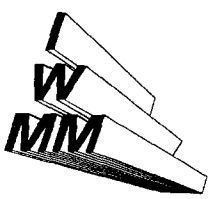
Lecture Notes in Computer Science 637
Edited by G. Goos and J. Hartmanis

Y. Bekkers J. Cohen (Eds.)

Memory Management

International Workshop IWMM 92
St. Malo, France, September 17-19, 1992
Proceedings



Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Volume Editors

Yves Bekkers
IRISA, Campus de Beaulieu
F-35042 Rennes, France

Jacques Cohen
Mitchum School of Computer Science, Ford Hall
Brandeis University, Waltham, MA 02254, USA

CR Subject Classification (1991): D.1, D.3-4, B.3, E.2

ISBN 3-540-55940-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55940-X Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

Storage reclamation became a necessity when the Lisp function *cons* was originally conceived[1]. That statement is simply a computer-oriented version of the broader precept: Recycling becomes unavoidable when usable resources are depleted. Both statements succinctly explain the nature of the topics discussed in the *International Workshop on Memory Management* (IWMM) that took place in Saint-Malo, France, in September 1992. This volume assembles the refereed technical papers which were presented during the workshop.

The earlier programming languages (such as Fortran) were designed so that the size of the storage required for the execution of a program was known at compile time. Subsequent languages (such as Algol 60) were implemented using a *stack* as a principal data-structure which is managed dynamically: information pushed onto a stack uses memory space which can be later released by popping.

With the introduction of structures (also called records) in more recent programming languages, it became important to establish an additional run-time data structure: the *heap*, which is used to store data-cells containing pointers to other cells. The *stack-heap* arrangement has become practically universal in the implementation of programming languages. An important characteristic of the cells in the heap is that the data they contain can become "useless" since they are not pointed to by any other cells. Reclamation of the so-called "useless cells" can be performed in an *ad hoc* (manual) manner by having the programmer explicitly return those cells to the run-time system so that they can be reused. (In *ad hoc* reclamation the programmer has to exercise great caution not to return cells containing valuable data.) This is the case of languages like Pascal or C which provide primitive procedures for returning useless cells. In the case of languages such as Lisp and Prolog reclamation is done automatically using a run-time process called *garbage-collection* which detects useless cells and makes them available for future usage.

Practically all the papers in this volume deal with the various aspects of managing and reclaiming memory storage when using a *stack-heap* model. A peculiar problem of memory management strategies is the unpredictability of computations. The undecidability of the halting problem implies that, in general, it is impossible to foresee how many cells will be needed in performing complex computations.

There are basically two approaches for performing storage reclamation: one is *incremental*, i.e., the implementor chooses to blend the task of collecting with that of actual computation; the other is what we like to call the *mañana* method - wait until the entire memory is exhausted to trigger the time-consuming operation of recognizing useless cells and making them available for future usage. A correct reclamation should ensure the following properties:

- *No used cell will be (erroneously) reclaimed.*
- *All useless cells will be reclaimed.*

Violating the first property is bound to have tragic consequences. A violation of the second may not be disastrous, but could lead to a premature halting of the execution due to the lack of memory. As a matter of fact, *conservative* collectors have been proposed to trade a (small) percentage of unreclaimed useless cells for a speedup of the collection process.

An important step in the collection is the identification of useless cells. This can be achieved by *marking* all the useful cells and *sweeping* the entire memory to collect useless

[1] The reader is referred to the chapter on the *History of Lisp*, by John McCarthy, which appeared in History of Programming Languages, edited by Richard L. Wexelblat, Academic Press, 1981, pp 173-183.

(unmarked) cells. This process is known as *mark-and-sweep*. Another manner of identifying useless cells is to keep *reference counts* which are constantly updated to indicate the number of pointers to a given cell. When this number becomes zero the cell is identified as useless. If the mark-and-sweep or the reference count techniques fail to locate any useless cells, the program being executed has to halt due to lack of storage. (A nasty situation may occur when successive collections succeed in reclaiming only a few cells. In such cases very little actual computation is performed between consecutive time-consuming collections.)

Compacting collectors are those which compact the useful information into a contiguous storage area. Such compacting requires that pointers be properly readjusted. Compacting becomes an important issue in paging systems (or in the case of hierarchical or virtual memories) since the compacted useful information is likely to result in fewer page faults, and therefore in increased performance.

An alternative method of garbage-collection which has drawn the attention of implementors in recent years is that of *copying*. In this case the useful cells are simply copied into a "new" area from the "old" one. These areas are called semi-spaces. When the space in the "new" area is exhausted, the "old" and "new" semi-spaces are swapped. Although this method requires twice the storage area needed by other methods, it can be performed incrementally, thus offering the possibility of *real-time* garbage-collection, in which the interruptions for collections are reasonably short.

The so-called *generational* garbage-collection is based on the experimental fact that certain cells remain used during substantial periods of the execution of a program, whereas others become useless shortly after they are generated. In these cases the reclaiming strategy consists of bypassing the costly redundant identification of "old generation" cells.

With the advent of *distributed* and *parallel* computers reclamation becomes considerably more complex. The choice of storage management strategy is, of course, dependent on the various types of existing architectures. One should distinguish the cases of:

1. Distributed computers communicating via a network,
2. Parallel shared-memory (MIMD) computers, and
3. Massively parallel (SIMD) computers.

In the case of distributed reclamation it is important that collectors be fault tolerant: a failure of one or more processors should not result in loss of information. The term *on-the-fly* garbage-collection is (usually) applicable to parallel shared-memory machines in which one or more processors are dedicated exclusively to collecting while others, called *mutators*, are responsible for performing useful computations which in turn may generate useless cells that have to be reclaimed.

Some features of storage management are *language-dependent*. Presently, one can distinguish three major paradigms in programming language design: *functional*, *logic*, and *object-oriented*. Although functional languages, like Lisp, were the first to incorporate garbage-collection in their design, both logic and object-oriented language implementors followed suit. Certain languages have features that enable their implementors to take advantage of known properties of data in the stack or in the heap so as to reduce the execution time needed for collection and/or to reclaim as many useless cells as possible.

In the preceding paragraphs we have briefly defined the terms: *mark-and-sweep*, *reference count*, *compacting*, *copying*, *incremental*, *generational*, *conservative*, *distributed*, *parallel*, *on-the-fly*, *real-time*, and *language-dependent features*. These terms should serve to guide the reader through the various papers presented in this volume.

We suggest that non-specialists start by reading the three survey papers. The first provides a general overview of the recent developments in the field; the second specializes in distributed collection, and the third deals with storage management in processors for logic programs. The other chapters in this volume deal with the topics of distributed, parallel, and

incremental collections, collecting in functional, logic, and object-oriented languages, and collections using massively parallel computers. The final article in this volume is an invited paper by H. G. Baker in which he proposes a "reversible" Lisp-like language (i.e., capable of reversing computations) and discusses the problems of designing suitable garbage-collectors for that language.

We wish to thank the referees for their careful evaluation of the submitted papers, and for the suggestions they provided to the authors for improving the quality of the presentation. Finally, it is fair to state that, even with technological advances, there will always be limited memory resources, especially those of very fast access. These memories will likely remain costlier than those with slower access. Therefore many of the solutions proposed at the IWMM are likely to remain valid for years to come.

July 1992

Yves Bekkers
Jacques Cohen

Program Committee

Chair	
Jacques Cohen	Brandeis University, Waltham, MA, USA
Members	
Joel F. Bartlett	DEC, Palo Alto, CA, USA
Yves Bekkers	INRIA-IRISA, Rennes, France
Hans-Jurgen Boehm	Xerox Corporation, Palo Alto, CA, USA
Maurice Bruynooghe	Katholieke Universiteit, Leuven, Belgium
Bernard Lang	INRIA, Le Chesnay, France
David A. Moon	Apple Computer, Cambridge, MA, USA
Christian Queinnec	Ecole Polytechnique, Palaiseau, France
Dan Sahlin	SICS, Kista, Sweden
Taiichi Yuasa	Toyohashi Univ. of Tech., Toyohashi, Japan

We thank all the people who helped the program committee in the refereering process, some of whom are listed below: K. Ali, M. Banâtre, P. Brand, A. Callebou, P. Fradet, S. Jansson, P. Magnusson, A. Mariën, R. Moolenaar, A. Mulkers, O. Ridoux, A. Saulsbury, T. Sjöland, L. Ungaro, P. Weemeeuw.

Workshop Coordinator

Yves Bekkers — INRIA-IRISA, Rennes, France

Sponsored by

INRIA
University of Rennes I
CNRS-GRECO Programmation

In cooperation with

ACM SIGPLAN

Table of Contents

Surveys

Uniprocessor Garbage Collection Techniques
Paul R. Wilson 1

Collection Schemes for Distributed Garbage
S.E. Abdullahi, E.E. Miranda, G.A. Ringwood 43

Dynamic Memory Management for Sequential Logic Programming Languages
Y. Bekkers, O. Ridoux, L. Ungaro 82

Distributed Systems I

Comprehensive and Robust Garbage Collection in a Distributed System
N.C. Juul, E. Jul 103

Distributed Systems II

Experience with a Fault-Tolerant Garbage Collector in a Distributed Lisp System
D. Plainfossé, M. Shapiro 116

Scalable Distributed Garbage Collection for Systems of Active Objects
N. Venkatasubramanian, G. Agha, C. Talcott 134

Distributed Garbage Collection of Active Objects with no Global Synchronisation
I. Puaut 148

Parallelism I

Memory Management for Parallel Tasks in Shared Memory
K.G. Langendoen, H.L. Muller, W.G. Vree 165

Incremental Multi-Threaded Garbage Collection on Virtually Shared Memory Architectures
T. Le Sergent, B. Berthomieu 179

Functional languages

Generational Garbage Collection for Lazy Graph Reduction
J. Seward 200

A Conservative Garbage Collector with Ambiguous Roots for Static Typechecking Languages
E. Chailloux 218

An Efficient Implementation for Coroutines
L. Mateu 230

An Implementation of an Applicative File System
B.C. Heck, D.S. Wise 248

Logic Programming Languages I

A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs
S. Duvvuru, R. Sundararajan, E. Tick, A. V. S. Sastry, L. Hansen, X. Zhong 264

Object Oriented Languages

Finalization in the Collector Interface
B. Hayes 277

Precompiling C++ for Garbage Collection
D.R. Edelson 299

Garbage Collection-Cooperative C++
A. D. Samples 315

Logic Programming Languages II

Dynamic Revision of Choice Points During Garbage Collection in Prolog [II/III]
J.F. Pique 330

Ecological Memory Management in a Continuation Passing Prolog Engine
P. Tarau 344

Incremental

Replication-Based Incremental Copying Collection
S. Nettles, J. O'Toole, D. Pierce, N. Haines 357

Atomic Incremental Garbage Collection
E.K. Kolodner, W.E. Weihl 365

Incremental Collection of Mature Objects
R.L. Hudson, J.E.B. Moss 388

Improving Locality

Object Type Directed Garbage Collection to Improve Locality
M.S. Lam, P.R. Wilson, T.G. Moher 404

Allocation Regions and Implementation Contracts
V. Delacour 426

Parallelism II

A Concurrent Generational Garbage Collector for a Parallel Graph Reducer
N. Röjemo 440

Garbage Collection in Aurora : An Overview
P. Weemeeuw, B. Demoen 454

Massively Parallel Architectures

Collections and Garbage Collection
S.C. Merrall, J.A. Padget 473

Memory Management and Garbage Collection of an
Extended Common Lisp System for Massively Parallel SIMD Architecture
T. Yuasa 490

Invited Speaker

NREVERSAL of Fortune - The Thermodynamics of Garbage Collection
H.G. Baker 507

Author Index 525

Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

Since liveness is a *global* property, this introduces nonlocal bookkeeping into routines that might otherwise be orthogonal, composable, and reusable. This bookkeeping can reduce extensibility, because when new functionality is implemented, the bookkeeping code must be updated.

The unnecessary complications created by explicit storage allocation are especially troublesome because programming mistakes often introduce erroneous behavior that breaks the basic abstractions of the programming language, making errors hard to diagnose.

Failing to reclaim memory at the proper point may lead to slow memory *leaks*, with unreclaimed memory gradually accumulating until the process terminates or swap space is exhausted. Reclaiming memory too soon can lead to very strange behavior, because an object's space may be reused to store a completely different object while an old pointer still exists. The same memory may therefore be interpreted as two different objects simultaneously with updates to one causing unpredictable mutations of the other.

These bugs are particularly dangerous because they often fail to show up repeatably, making debugging very difficult; they may never show up at all until the program is stressed in an unusual way. If the allocator happens not to reuse a particular object's space, a dangling pointer may not cause a problem. Later, in the field, the application may crash when it makes a different set of memory demands, or is linked with a different allocation routine. A slow leak may not be noticeable while a program is being used in normal ways—perhaps for many years—because the program terminates before too much extra space is used. But if the code is incorporated into a long-running server program, the server will eventually exhaust its swap space, and crash.

Explicit allocation and reclamation lead to program errors in more subtle ways as well. It is common for programmers to statically allocate a moderate number of objects, so that it is unnecessary to allocate them on the heap and decide when and where to reclaim them. This leads to fixed limitations on software, making them fail when those limitations are exceeded, possibly years later when memories (and data sets) are much larger. This "brittleness" makes code much less reusable, because the undocumented limits cause it to fail, even if it's being used in a way consistent with its abstractions. (For example, many compilers fail when faced with automatically-generated programs that violate assumptions about "normal" programming practices.)

These problems lead many applications programmers to implement some form of application-specific garbage collection within a large software system, to avoid most of the headaches of explicit storage management. Many large programs have their own data types that implement reference counting, for example. Unfortunately, these collectors are often both incomplete and buggy, because they are coded up for a one-shot application. The garbage collectors themselves are therefore often unreliable, as well as being hard to use because they are not integrated into the programming language. The fact that such kludges exist despite these problems is a testimony to the value of garbage collection, and it suggests that garbage collection should be part of programming language implementations.

In the rest of this paper, we focus on garbage collectors that are built into a language implementation. The usual arrangement is that the allocation routines of

the language (or imported from a library) perform special actions to reclaim space, as necessary, when a memory request is not easily satisfied. (That is, calls to the "deallocator" are unnecessary because they are implicit in calls to the allocator.)

Most collectors require some cooperation from the compiler (or interpreter), as well: object formats must be recognizable by the garbage collector, and certain invariants must be preserved by the running code. Depending on the details of the garbage collector, this may require slight changes to the code generator, to emit certain extra information at compile time, and perhaps execute different instruction sequences at run time. (Contrary to widespread misconceptions, there is no conflict between using a compiled language and garbage collection; state-of-the art implementations of garbage-collected languages use sophisticated optimizing compilers.)

1.2 The Two-Phase Abstraction

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such data objects are referred to as *garbage*. The basic functioning of a garbage collector consists, abstractly speaking, of two parts:

1. Distinguishing the live objects from the garbage in some way, or *garbage detection*, and
2. Reclaiming the garbage objects' storage, so that the running program can use it.

In practice, these two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique.

In general, garbage collectors use a "liveness" criterion that is somewhat more conservative than those used by other systems. In an optimizing compiler, a value may be considered dead at the point that it can never be used again by the running program, as determined by control flow and data flow analysis. A garbage collector typically uses a simpler, less dynamic criterion, defined in terms of a *root set* and *reachability* from these roots. At the point when garbage collection occurs[2] all globally visible variables of active procedures are considered live, and so are the local variables of any active procedures. The *root set* therefore consists of the global variables, local variables in the activation stack, and any registers used by active procedures. Heap objects directly reachable from any of these variables could be accessed by the running program, so they must be preserved. In addition, since the program might traverse pointers from those objects to reach other objects, any object reachable from a live object is also live. Thus the set of live objects is simply the set of objects on any directed path of pointers from the roots.

Any object that is not reachable from the root set is garbage, i.e., useless, because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore can't affect the future course of the computation, and their space may be safely reclaimed.

[2] Typically, this happens when allocation of an object has been attempted by the running program, but there is not sufficient free memory to satisfy the request. The allocation routine calls a garbage collection routine to free up space, then allocates the requested object.

1.3 Object Representations

Throughout this paper, we make the simplifying assumption that heap objects are self-identifying, i.e., that it is easy to determine the type of an object at run time. Implementations of statically-typed garbage collected languages typically have hidden "header" fields on heap objects, i.e., an extra field containing type information, which can be used to decode the format of the object itself. (This is especially useful for finding pointers to other objects.)

Dynamically-typed languages such as Lisp and Smalltalk usually use *tagged* pointers; a slightly shortened representation of the hardware address is used, with a small type-identifying field in place of the missing address bits. This also allows short immutable objects (in particular, small integers) to be represented as unique bit patterns stored directly in the "address" part of the field, rather than actually referred to by an address. This tagged representation supports polymorphic fields which may contain either one of these "immediate" objects or a pointer to an object on the heap. Usually, these short tags are augmented by additional information in heap-allocated objects' headers.

For a purely statically-typed language, no per-object runtime type information is actually necessary, except the types of the root set variables.[3] Once those are known, the types of their referents are known, and their fields can be decoded [App89a, Gol91]. This process continues transitively, allowing types to be determined at every pointer traversal. Despite this, headers are often used for statically-typed languages, because it simplifies implementations at little cost. (Conventional (explicit) heap management systems often use object headers for similar reasons.)

2 Basic Garbage Collection Techniques

Given the basic two-part operation of a garbage collector, many variations are possible. The first part, distinguishing live objects from garbage, may be done in several ways: by *reference counting*, *marking*, or *copying*. [4] Because each scheme has a major influence on the second part (reclamation) and on reuse techniques, we will introduce reclamation methods as we go.

2.1 Reference Counting

In a reference counting system [Col60], each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g., when a pointer is copied from one place to another by an assignment, the object's count is incremented. When an existing reference to an object is eliminated, the count is

[3] *Conservative* garbage collectors [BW88, Wen90, BDS91, WH91] are usable with little or no cooperation from the compiler—not even the types of named variables—but we will not discuss them here.

[4] Some authors use the term "garbage collection" in a narrower sense, which excludes reference counting and/or copy collection systems; we prefer the more inclusive sense because of its popular usage and because it's less awkward than "automatic storage reclamation."

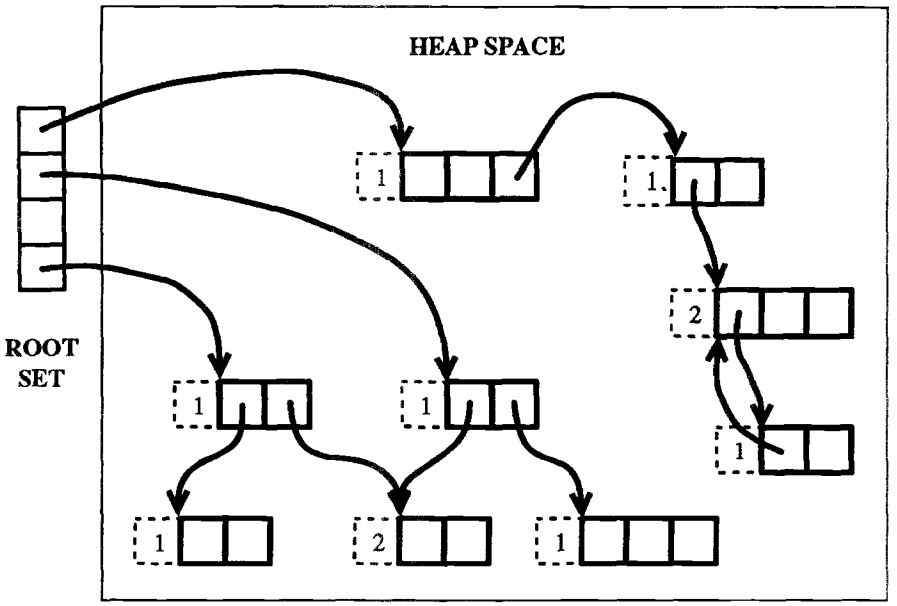


Fig. 1. Reference counting.

decremented. (See Fig. 1.) The memory occupied by an object may be reclaimed when the object's count equals zero, since that indicates that no pointers to the object exist and the running program could not reach it.

(In a straightforward reference counting system, each object typically has a header field of information describing the object, which includes a subfield for the reference count. Like other header information, the reference count is generally not visible at the language level.)

When the object is reclaimed, its pointer fields are examined, and any objects it holds pointers to also have their reference counts decremented, since references from a garbage object don't count in determining liveness. Reclaiming one object may therefore lead to the transitive decrementing of reference counts and reclaiming many other objects. For example, if the only pointer into some large data structure

becomes garbage, all of the reference counts of the objects in that structure typically become zero, and all of the objects are reclaimed.

In terms of the abstract two-phase garbage collection, the adjustment and checking of reference counts implements the first phase, and the reclamation phase occurs when reference counts hit zero. These operations are both interleaved with the execution of the program, because they may occur whenever a pointer is created or destroyed.

One advantage of reference counting is this *incremental* nature of most of its operation—garbage collection work (updating reference counts) is interleaved closely with the running program's own execution. It can easily be made completely incremental and *real time*; that is, performing at most a small and bounded amount of work per unit of program execution.

Clearly, the normal reference count adjustments are intrinsically incremental, never involving more than a few operations for any given operation that the program executes. The transitive reclamation of whole data structures can be deferred, and also done a little at a time, by keeping a list of freed objects whose reference counts have become zero but which haven't yet been processed yet.

This incremental collection can easily satisfy real time requirements, guaranteeing that memory management operations never halt the executing program for more than a very brief period. This can support *real-time* applications in which guaranteed response time is critical; incremental collection ensures that the program is allowed to perform a significant, though perhaps appreciably reduced, amount of work in any significant amount of time. (A target criterion might be that no more than one millisecond out of every two-millisecond period would be spent on storage reclamation operations, leaving the other millisecond for "useful work" to satisfy the program's real-time purpose.)

There are two major problems with reference counting garbage collectors; they are difficult to make *efficient*, and they are not always *effective*.

The Problem with Cycles The effectiveness problem is that reference counting fails to reclaim *circular* structures. If the pointers in a group of objects create a (directed) cycle, the objects' reference counts are never reduced to zero, *even if there is no path to the objects from the root set* [McB63].

Figure 2 illustrates this problem. Consider the isolated pair of objects on the right. Each holds a pointer to the other, and therefore each has a reference count of one. Since no path from a root leads to either, however, the program can never reach them again.

Conceptually speaking, the problem here is that reference counting really only determines a *conservative approximation* of true liveness. If an object is not pointed to by any variable or other object, it is clearly garbage, but the converse is often not true.

It may seem that circular structures would be very unusual, but they are not. While most data structures are acyclic, it is not uncommon for normal programs to create some cycles, and a few programs create very many of them. For example, nodes in trees may have "backpointers," to their parents, to facilitate certain operations. More complex cycles are sometimes formed by the use of hybrid data structures

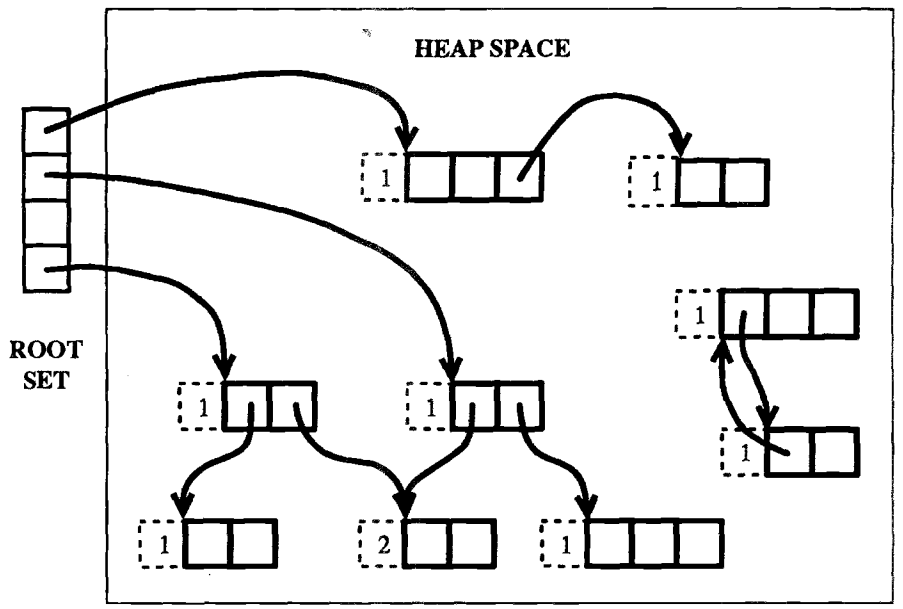


Fig. 2. Reference counting with unreclaimable cycle.

which combine advantages of simpler data structures, and the like.

Systems using reference counting garbage collectors therefore usually include some other kind of garbage collector as well, so that if too much uncollectable cyclic garbage accumulates, the other method can be used to reclaim it.

Many programmers who use reference-counting systems (such as Interlisp and early versions of Smalltalk) have modified their programming style to avoid the creation of cyclic garbage, or to break cycles before they become a nuisance. This has a negative impact on program structure, and many programs still have storage "leaks" that accumulate cyclic garbage which must be reclaimed by some other means.[5] These leaks, in turn, can compromise the real-time nature of the algorithm,

[5] [Bob80] describes modifications to reference counting to allow it to handle some special cases of cyclic structures, but this restricts the programmer to certain stereotyped

because the system may have to fall back to the use of a non-real-time collector at a critical moment.

The Efficiency Problem. The efficiency problem with reference counting is that its cost is generally proportional to the amount of work done by the running program, with a fairly large constant of proportionality. One cost is that when a pointer is created or destroyed, its referent's count must be adjusted. If a variable's value is changed from one pointer to another, *two* objects' counts must be adjusted—one object's reference count must be incremented, the other's decremented and then checked to see if it has reached zero.

Short-lived stack variables can incur a great deal of overhead in a simple reference-counting scheme. When an argument is passed, for example, a new pointer appears on the stack, and usually disappears almost immediately because most procedure activations (near the leaves of the call graph) return very shortly after they are called. In these cases, reference counts are incremented, and then decremented back to their original value very soon. It is desirable to optimize away most of these increments and decrements that cancel each other out.

Deferred Reference Counting. Much of this cost can be optimized away by special treatment of local variables [DB76]. Rather than always adjusting reference counts and reclaiming objects whose counts become zero, references from the local variables are not included in this bookkeeping most of the time. Usually, reference counts are only adjusted to reflect pointers from one heap object to another. This means that reference counts may not be accurate, because pointers from the stack may be created or destroyed without being accounted for; that, in turn, means that objects whose count drops to zero may not actually be reclaimable. Garbage collection can only be done when references from the stack are taken into account.

Every now and then, the reference counts are brought up to date by scanning the stack for pointers to heap objects. Then any objects whose reference counts are still zero may be safely reclaimed. The interval between these phases is generally chosen to be short enough that garbage is reclaimed often and quickly, yet still long enough that the cost of periodically updating counts (for stack references) is not high.

This *deferred reference counting* [DB76] avoids adjusting reference counts for most short-lived pointers from the stack, and greatly reduces the overhead of reference counting. When pointers from one heap object to another are created or destroyed, however, the reference counts must still be adjusted. This cost is still roughly proportional to the amount of work done by the running program in most systems, but with a lower constant of proportionality.

There is another cost of reference-counting collection that is harder to escape. When objects' counts go to zero and they are reclaimed, some bookkeeping must be done to make them available to the running program. Typically this involves linking the freed objects into one or more "free lists" of reusable objects, out of which the program's allocation requests are satisfied.

---

patterns.

It is difficult to make these reclamation operations take less than several instructions per object, and the cost is therefore proportional to the number of objects allocated by the running program.

These costs of reference counting collection have combined with its failure to reclaim circular structures to make it unattractive to most implementors in recent years. As we will explain below, other techniques are usually more efficient and reliable.

(This is not to say that reference counting is a dead technique. It still has advantages in terms of the immediacy with which it reclaims most garbage,[6] and corresponding beneficial effects on locality of reference;[7] a reference counting system may perform with little degradation when almost all of the heap space is occupied by live objects, while other collectors rely on trading more space for higher efficiency. Reference counts themselves may be valuable in some systems. For example, they may support optimizations in functional language implementations by allowing destructive modification of uniquely-referenced objects. Distributed garbage collection is often done with reference-counting between nodes of a distributed system, combined with mark-sweep or copying collection within a node. Future systems may find other uses for reference counting, perhaps in hybrid collectors also involving other techniques, or when augmented by specialized hardware. Nonetheless, reference counting is generally not considered attractive as the primary garbage collection technique on conventional uniprocessor hardware.)

For most high-performance general-purpose systems, reference counting has been abandoned in favor of *tracing* garbage collectors, which actually traverse (trace out) the graph of live objects, distinguishing them from the unreachable (garbage) objects which can then be reclaimed.

2.2 Mark-Sweep Collection

Mark-sweep garbage collectors [McC60] are named for the two phases that implement the abstract garbage collection algorithm we described earlier:

1. *Distinguish the live objects from the garbage.* This is done by tracing—starting at the root set and actually traversing the graph of pointer relationships—usually by either a depth-first or breadth-first traversal. The objects that are reached are *marked* in some way, either by altering bits within the objects, or perhaps by recording them in a bitmap or some other kind of table.
2. *Reclaim the garbage.* Once the live objects have been made distinguishable from the garbage objects, memory is *swept*, that is, exhaustively examined, to find all of the unmarked (garbage) objects and reclaim their space. Traditionally, as with reference counting, these reclaimed objects are linked onto one or more free lists so that they are accessible to the allocation routines.

[6] This can be useful for *finalization*, that is, performing "clean-up" actions (like closing files) when objects die [Rov85].

[7] DeTreville [DeT90] argues that the locality characteristics of reference-counting may be superior to those of other collection techniques, based on experience with the Topaz system. However, as [WLM92] shows, generational techniques can recapture some of this locality.

There are three major problems with traditional mark-sweep garbage collectors. First, it is difficult to handle objects of varying sizes without fragmentation of the available memory. The garbage objects whose space is reclaimed are interspersed with live objects, so allocation of large objects may be difficult; several small garbage objects may not add up to a large contiguous space. This can be mitigated somewhat by keeping separate free lists for objects of varying sizes, and merging adjacent free spaces together, but difficulties remain. (The system must choose whether to allocate more memory as needed to create small data objects, or to divide up large contiguous hunks of free memory and risk permanently fragmenting them. This fragmentation problem is not unique to mark-sweep—it occurs in reference counting as well, and in most explicit heap management schemes.)

The second problem with mark-sweep collection is that the cost of a collection is proportional to the size of the heap, including both live and garbage objects. All live objects must be marked, and all garbage objects must be collected, imposing a fundamental limitation on any possible improvement in efficiency.

The third problem involves locality of reference. Since objects are never moved, the live objects remain in place after a collection, interspersed with free space. Then new objects are allocated in these spaces; the result is that objects of very different ages become interleaved in memory. This has negative implications for locality of reference, and simple mark-sweep collectors are often considered unsuitable for most virtual memory applications. (It is possible for the "working set" of active objects to be scattered across many virtual memory pages, so that those pages are frequently swapped in and out of main memory.) This problem may not be as bad as many have thought, because objects are often created in clusters that are typically active at the same time. Fragmentation and locality problems are is unavoidable in the general case, however, and a potential problem for some programs.

It should be noted that these problems may not be insurmountable, with sufficiently clever implementation techniques. For example, if a bitmap is used for mark bits, 32 bits can be checked at once with a 32-bit integer ALU operation and conditional branch. If live objects tend to survive in clusters in memory, as they apparently often do, this can greatly diminish the constant of proportionality of the sweep phase cost; the theoretical linear dependence on heap size may not be as troublesome as it seems at first glance. As a result, the dominant cost may be the marking phase, which is proportional to the amount of live data that must be traversed, not the total amount of memory allocated. The clever use of bitmaps can also reduce the cost of allocation, by allowing fast allocation from contiguous unmarked areas, rather than using free lists.

The clustered survival of objects may also mitigate the locality problems of reallocating space amid live objects; if objects tend to survive or die in groups in memory [Hay91], the interspersing of objects used by different program phases may not be a major consideration.

At this point, the technology of mark-sweep collectors (and related hybrids) is rapidly evolving. As will be noted later, this makes them resemble copying collectors in some ways; at this point we do not claim to be able to pick a winner between high-tech mark-sweep and copy collectors.

2.3 Mark-Compact Collection

Mark-compact collectors remedy the fragmentation and allocation problems of mark-sweep collectors. As with mark-sweep, a marking phase traverses and marks the reachable objects. Then objects are *compacted*, moving most of the live objects until all of the live objects are contiguous. This leaves the rest of memory as a single contiguous free space. This is often done by a linear scan through memory, finding live objects and "sliding" them down to be adjacent to the previous object. Eventually, all of the live objects have been slid down to be adjacent to a live neighbor. This leaves one contiguous occupied area at one end of heap memory, and implicitly moving all of the "holes" to the contiguous area at the other end.

This sliding compaction has several interesting properties. The contiguous free area eliminates fragmentation problems so that allocating objects of various sizes is simple. Allocation can be implemented as the incrementing of a pointer into a contiguous area of memory, in much the way that different-sized objects can be allocated on a stack. In addition, the garbage spaces are simply "squeezed out," without disturbing the original ordering of objects in memory. This can ameliorate locality problems, because the allocation ordering is usually more similar to subsequent access orderings than an arbitrary ordering imposed by a garbage collector [CG77, Cla79].

While the locality that results from sliding compaction is advantageous, the collection process itself shares the mark-sweep's unfortunate property that several passes over the data are required. After the initial marking phase, sliding compactors make two or three more passes over the live objects [CN83]. One pass computes the new locations that objects will be moved to; subsequent passes must update pointers to refer to objects' new locations, and actually move the objects. These algorithms may be therefore be significantly slower than mark-sweep if a large percentage of data survives to be compacted.

An alternative approach is to use a *two-pointer algorithm*, which scans inward from both ends of a heap space to find opportunities for compaction. One pointer scans downward from the top of the heap, looking for live objects, and the other scans upward from the bottom, looking for a hole to put it in. (Many variations of this algorithm are possible, to deal with multiple areas holding different-sized objects, and to avoid intermingling objects from widely-dispersed areas.) For a more complete treatment of compacting algorithms, see [Knu69, CN83].

2.4 Copying Garbage Collection

Like mark-compact (but unlike mark-sweep), *copying* garbage collection does not really "collect" garbage. Rather, it moves all of the *live* objects into one area, and the rest of the heap is then known to be available because it contains only garbage. "Garbage collection" in these systems is thus only implicit, and some researchers avoid applying that term to the process.

Copying collectors, like marking-and-compacting collectors, move the objects that are reached by the traversal to a contiguous area. While compacting collectors use a separate marking phase that traverses the live data, copying collectors integrate the traversal of the data and the copying process, so that most objects

need only be traversed once. Objects are moved to the contiguous destination area as they are reached by the traversal. The work needed is proportional to the amount of live data (all of which must be copied).

The term *scavenging* is applied to the copying traversal, because it consists of picking out the worthwhile objects amid the garbage, and taking them away.

A Simple Copying Collector: "Stop-and-Copy" Using Semispaces. A very common kind of copying garbage collector is the *semispace* collector [FY69] using the *Cheney* algorithm for the copying traversal [Che70]. We will use this collector as a reference model for much of this paper.[8]

In this scheme, the space devoted to the heap is subdivided into two contiguous *semispaces*. During normal program execution, only one of these semispaces is in use, as shown in Fig. 3. Memory is allocated linearly upward through this "current" semispace as demanded by the executing program. This is much like allocation from a stack, or in a sliding compacting collector, and is similarly fast; there is no fragmentation problem when allocating objects of various sizes.

When the running program demands an allocation that will not fit in the unused area of the current semispace, the program is stopped and the copying garbage collector is called to reclaim space (hence the term "stop-and-copy"). All of the live data are copied from the current semispace (*fromspace*) to the other semispace (*tospace*). Once the copying is completed, the tospace semispace is made the "current" semispace, and program execution is resumed. Thus the roles of the two spaces are reversed each time the garbage collector is invoked. (See Fig. 4.)

Perhaps the simplest form of copying traversal is the Cheney algorithm [Che70]. The immediately-reachable objects form the initial queue of objects for a breadth-first traversal. A "scan" pointer is advanced through the first object, location by location. Each time a pointer into fromspace is encountered, the referred-to-object is transported to the end of the queue, and the pointer to the object is updated to refer to the new copy. The free pointer is then advanced and the scan continues. This effects the "node expansion" for the breadth-first traversal, reaching (and copying) all of the descendants of that node. (See Fig. 5. Reachable data structures in fromspace are shown at the top of the figure, followed by the first several states of tospace as the collection proceeds—tospace is shown in linear address order to emphasize the linear scanning and copying.)

Rather than stopping at the end of the first object, the scanning process simply continues through subsequent objects, finding their offspring and copying them as well. A continuous scan from the beginning of the queue has the effect of removing consecutive nodes and finding all of their offspring. The offspring are copied to the end of the queue. Eventually the scan reaches the end of the queue, signifying that all of the objects that have been reached (and copied) have also been scanned for

[8] As a historical note, the first copying collector was Minsky's collector for Lisp 1.5 [Min63]. Rather than copying data from one area of memory to another, a single heap space was used. The live data were copied out to a file, and then read back in, in a contiguous area of the heap space. On modern machines this would be unbearably slow, because file operations—writing and reading every live object—are now many orders of magnitude slower than memory operations.

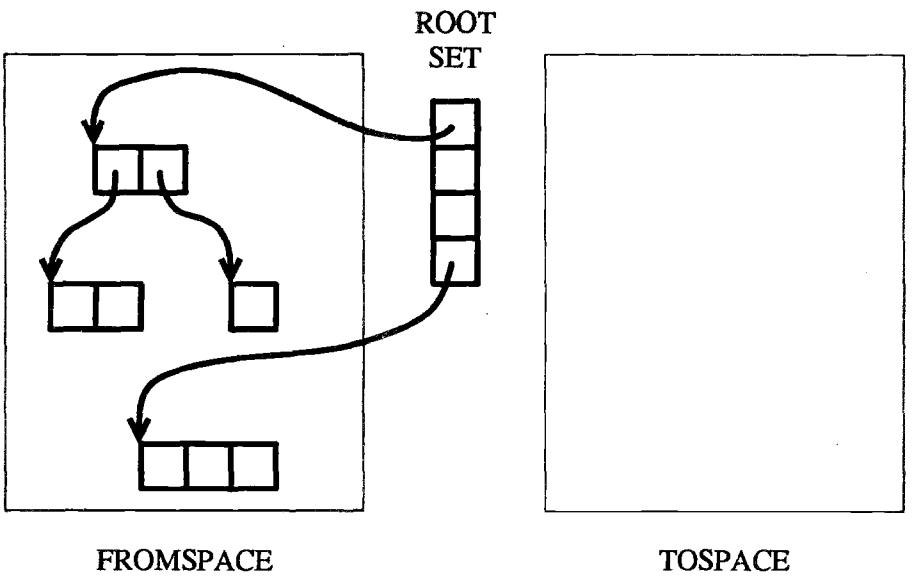


Fig. 3. A simple semispace garbage collector before garbage collection.

descendants. This means that there are no more reachable objects to be copied, and the scavenging process is finished.

Actually, a slightly more complex process is needed, so that objects that are reached by multiple paths are not copied to tospace multiple times. When an object is transported to tospace, a *forwarding pointer* is installed in the old version of the object. The forwarding pointer signifies that the old object is obsolete and indicates where to find the new copy of the object. When the scanning process finds a pointer into fromspace, the object it refers to is checked for a forwarding pointer. If it has one, it has already been moved to tospace, so the pointer it has been reached by is simply updated to point to its new location. This ensures that each live object is transported exactly once, and that all pointers to the object are updated to refer to the new copy.

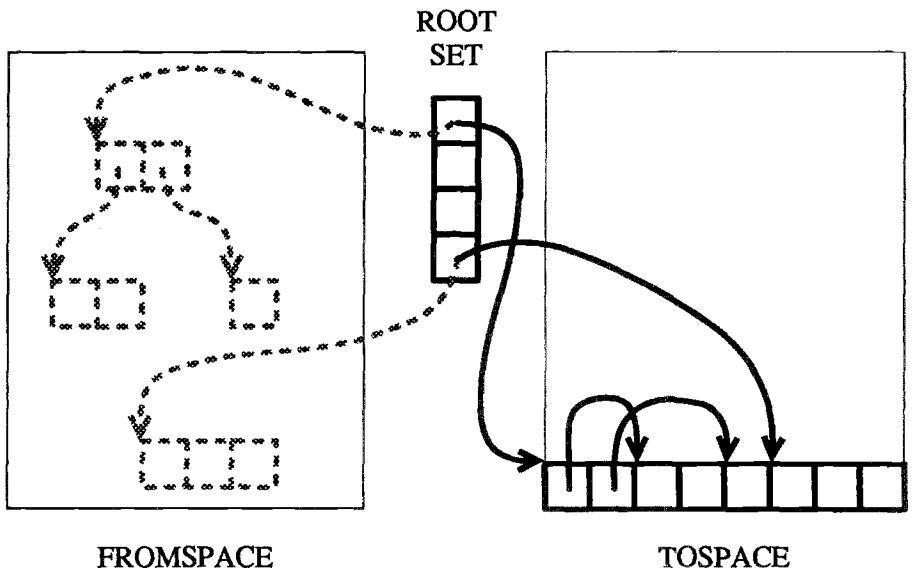


Fig. 4. Semispace collector after garbage collection.

Efficiency of Copying Collection. A copying garbage collector can be made arbitrarily efficient if sufficient memory is available [Lar77, App87]. The work done at each collection is proportional to the amount of live data at the time of garbage collection. Assuming that approximately the same amount of data is live at any given time during the program's execution, decreasing the frequency of garbage collections will decrease the total amount of garbage collection effort.

A simple way to decrease the frequency of garbage collections is to increase the amount of memory in the heap. If each semispace is bigger, the program will run longer before filling it. Another way of looking at this is that by decreasing the frequency of garbage collections, we are increasing the average age of objects at garbage collection time. Objects that become garbage before a garbage collection needn't be copied, so the chance that an object will *never* have to be copied is

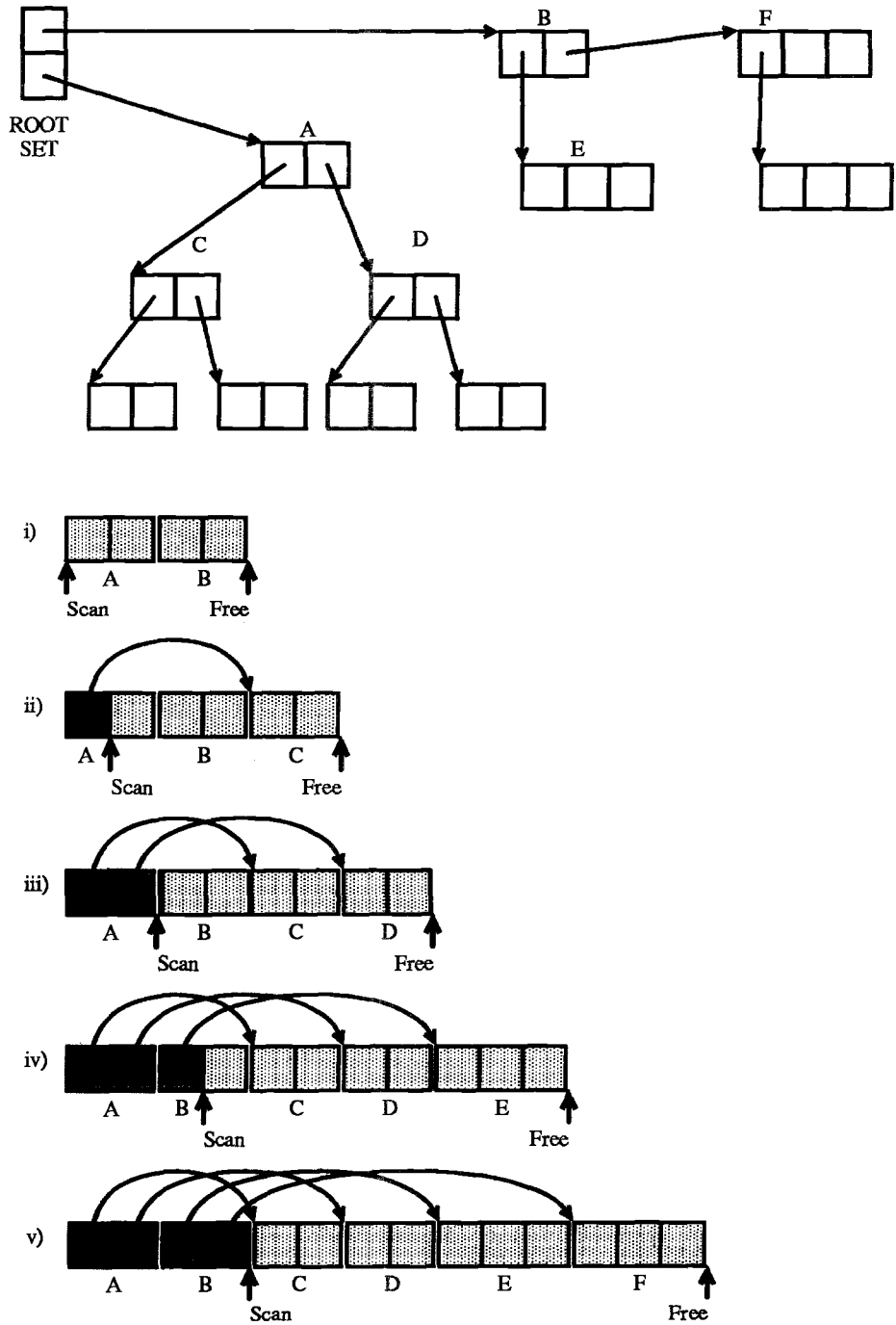


Fig. 5. The Cheney algorithm of breadth-first copying.

increased.

Suppose, for example, that during a program run twenty megabytes of memory are allocated, but only one megabyte is live at any given time. If we have two three-megabyte semispaces, garbage will be collected about ten times. (Since the current semispace is one third full after a collection, that leaves two megabytes that can be allocated before the next collection.) This means that the system will copy about half as much data as it allocates, as shown in the top part of Fig. 6. (Arrows represent copying of live objects between semispaces at garbage collections.)

On the other hand, if the size of the semispaces is doubled, 5 megabytes of free space will be available after each collection. This will force garbage collections a third as often, or about 3 or 4 times during the run. This straightforwardly reduces the cost of garbage collection by more than half, as shown in the bottom part of Fig. 6.

2.5 Non-Copying Implicit Collection

Recently, Baker [Bak92] has proposed a new kind of non-copying collector that with some of the efficiency advantages of a copying scheme. Baker's insight is that in a copying collector, the "spaces" of the collector are really just a particular implementation of sets. Another implementation of sets could do just as well, provided that it has similar performance characteristics. In particular, given a pointer to an object, it must be easy to determine which set it is a member of; in addition, it must be easy to switch the roles of the sets, just as fromspace and tospace roles are exchanged in a copy collector.

Baker's non-copying system adds two pointer fields and a "color" field to each object. These fields are invisible to the application programmer, and serve to link each hunk of storage into a doubly-linked list that serves as a set. The color field indicates which set an object belongs to.

The operation of this collector is simple, and isomorphic to the copy collector's operation. Chunks of free space are initially linked to form a doubly-linked list, and are allocated simply by incrementing a pointer into this list. The allocation pointer serves to divide the list into the part that has been allocated and the remaining "free" part. Allocation is fast because it only requires advancing this pointer to point at the next element of the free list. (Unlike the copying scheme, this does not eliminate fragmentation problems; supporting variable sized objects requires multiple free lists and may result in fragmentation of the available space.)

When the free list is exhausted, the collector traverses the live objects and "moves" them from the allocated set (which we could call the fromset) to another set (the toset). This is implemented by unlinking the object from the doubly-linked fromset list, toggling its mark field, and linking it into the toset's doubly-linked list.

Just as in a copy collector, space reclamation is implicit. When all of the reachable objects have been traversed and moved from the fromset to the toset, the fromset is known to contain only garbage. It is therefore a list of free space, which can immediately be put to use as a free list. (As we will explain in section 3.3, Baker's scheme is actually somewhat more complex, because his collector is incremental.) The cost of the collection is proportional to the number of live objects, and the garbage ones are all reclaimed in small constant time.

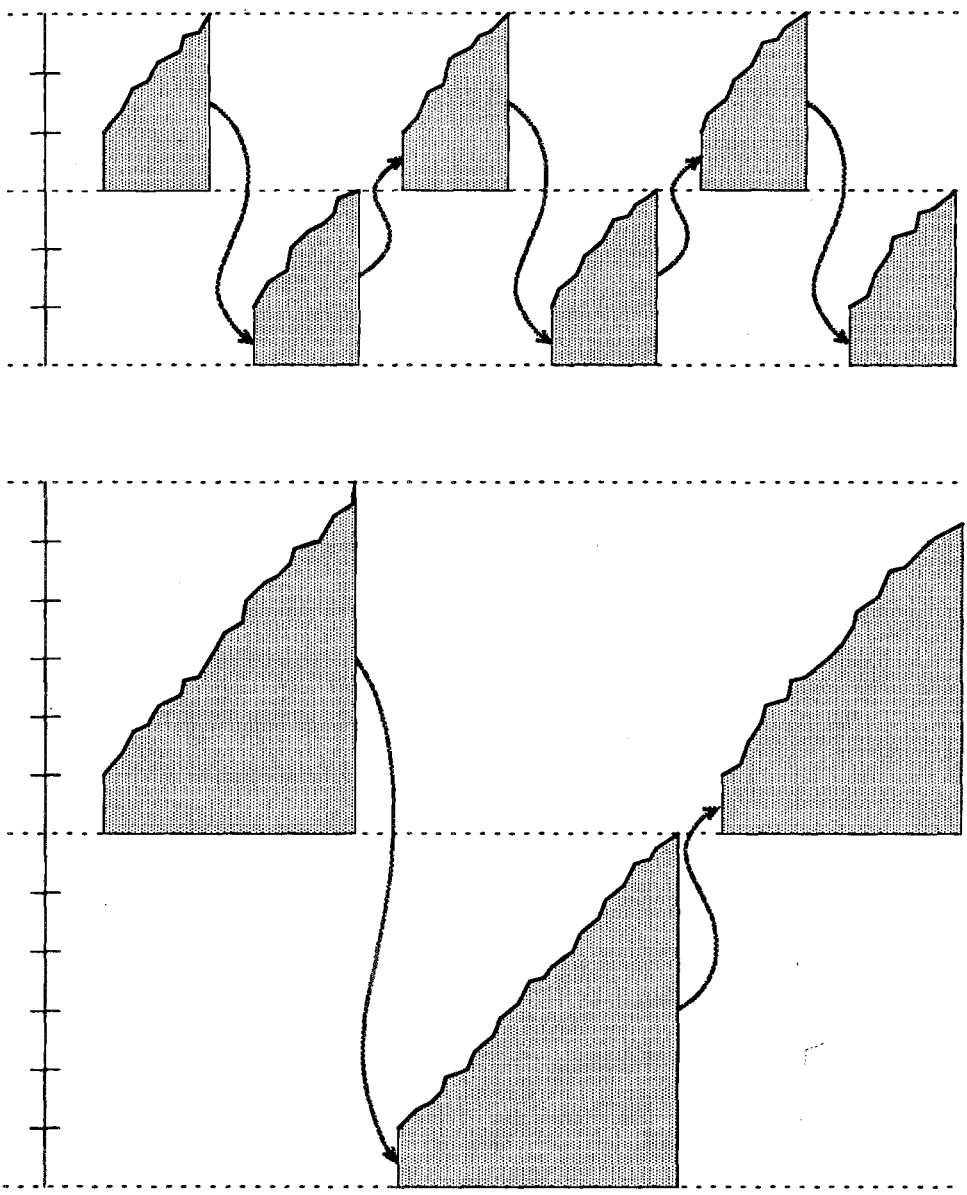


Fig. 6. Memory usage in a semispace GC, with 3 MB (top) and 6 MB (bottom) semispaces

This scheme has both advantages and disadvantages compared to a copy collector. On the minus side, the per-object constants are probably a little bit higher, and fragmentation problems are still possible. On the plus side, the tracing cost for large objects is not as high. As with a mark-sweep collector, the whole object needn't be copied; if it can't contain pointers, it needn't be scanned either. Perhaps more importantly for many applications, this scheme does not require the actual language-level pointers between objects to be changed, and this imposes fewer constraints on compilers. As we'll explain later, this is particularly important for parallel and real-time incremental collectors.

2.6 Choosing Among Basic Techniques

Treatments of garbage collection algorithms in textbooks often stress asymptotic complexity, but all basic algorithms have roughly similar costs, especially when we view garbage collection as part of the overall free storage management scheme. Allocation and garbage collection are two sides of the basic memory reuse coin, and any algorithm incurs costs at allocation time, if only to initialize the fields of new objects.

Any of the efficient collection schemes therefore has three basic cost components, which are (1) the initial work required at each collection, such as root set scanning, (2) the work done at per unit of allocation (proportional to the amount of allocation, or the number of objects allocated) and (3) the work done during garbage detection (e.g., tracing).

The latter two costs are usually similar, in that the amount of live data is usually some significant percentage of the amount of garbage. Thus algorithms whose cost is proportional to the amount of allocation (e.g., mark-sweep) may be competitive with those whose cost is proportional to the amount of live data traced (e.g., copying).

For example, suppose that 10 percent of all allocated data survive a collection, and 90 percent never need to be traced. In deciding which algorithm is more efficient, the asymptotic complexity is less important than the associated constants. If the cost of sweeping an object is ten times less than the cost of copying it, the mark-sweep collector costs about the same as as copy collector. (If a mark-sweep collector's sweeping cost is billed to the allocator, and it's small relative to the cost of initializing the objects, then it becomes obvious that the sweep phase is just not terribly expensive.) While current copying collectors appear to be more efficient than current mark-sweep collectors, the difference is not large for state-of-the art implementations.

Further, real high-performance systems often use hybrid techniques to adjust tradeoffs for different categories of objects. Many high-performance copy collectors use a separate *large object area* [CWB86, UJ88], to avoid copying large objects from space to space. The large objects are kept "off to the side" and usually managed in-place by some variety of marking traversal and free list technique.

A major point in favor of in-place collectors (such as mark-sweep and treadmill schemes) is the ability to make them *conservative* with respect to data values that may be pointers or may not. This allows them to be used for languages like C, or off-the-shelf optimizing compilers [BW88, Bar88, BDS91], which can make it difficult or impossible to unambiguously identify all pointers at run time. A non-moving

collector can be conservative because anything that looks like a pointer object can be left where it is, and the (possible) pointer to it doesn't need to be changed. In contrast, a copying collector must know whether a value is a pointer—and whether to move the object and update the pointer. For example, if presumed pointers were updated, and some were actually integers, the program would break because the integers would be mysteriously changed by the garbage collector.

2.7 Problems with a Simple Garbage Collector

It is widely known that the asymptotic complexity of copying garbage collection is excellent—the copying cost approaches zero as memory becomes very large. Treadmill collection shares this property, but other collectors can be similarly efficient if the constants associated with memory reclamation and reallocation are small enough. In that case, garbage detection is the major cost.

Unfortunately, it is difficult in practice to achieve high efficiency in a simple garbage collector, because large amounts of memory are too expensive. If virtual memory is used, the poor locality of the allocation and reclamation cycle will generally cause excessive paging. (Every location in the heap is used before any location's space is reclaimed and reused.) Simply paging out the recently-allocated data is expensive for a high-speed processor [Ung84], and the paging caused by the copying collection itself may be tremendous, since all live data must be touched in the process.)

It therefore doesn't generally pay to make the heap area larger than the available main memory. (For a mathematical treatment of this tradeoff, see [Lar77].) Even as main memory becomes steadily cheaper, locality within cache memory becomes increasingly important, so the problem is simply shifted to a different level of the memory hierarchy [WLM92].

In general, we can't achieve the potential efficiency of simple garbage collection; increasing the size of memory to postpone or avoid collections can only be taken so far before increased paging time negates any advantage.

It is important to realize that this problem is not unique to copying collectors. *All* garbage collection strategies involve similar space-time tradeoffs—garbage collections are postponed so that garbage detection work is done less often, and that means that space is not reclaimed as quickly. On average, that increases the amount of memory wasted due to unreclaimed garbage.[9]

While copying collectors were originally designed to improve locality, in their simple versions this improvement is not large, and their locality can in fact be *worse* than that of non-compacting collectors. These systems may improve the locality of reference to long-lived data objects, which have been compacted into a contiguous area. However, this effect is swamped by the pattern of references due to allocation.

[9] Deferred reference counting, like tracing collection, also trades space for time—in giving up continual incremental reclamation to avoid spending CPU cycles in adjusting reference counts, one gives up space for objects that become garbage and are not immediately reclaimed. At the time scale on which memory is reused, the resulting locality characteristics must share basic performance tradeoff characteristics with generational collectors of the copying or mark-sweep varieties, which will be discussed later.

Large amounts of memory are touched *between* collections, and this alone makes them unsuitable for a virtual memory environment.

The major locality problem is not with the locality of compacted data, or with the locality of the garbage collection process itself. The problem is an *indirect* result of the use of garbage collection—by the time space is reclaimed *and reused*, it's likely to have been paged out, simply because too many other pages have been allocated in between. Compaction is helpful, but the help is generally *too little, too late.* With a simple semispace copy collector, locality is likely to be worse than that of a mark-sweep collector, simply because the copy collector uses more total memory—only half the memory can be used between collections. Fragmentation of live data is not as detrimental as the regular reuse of two spaces.[10]

The only way to have good locality is to ensure that memory is large enough to hold the regularly-reused area. (Another approach would be to rely on optimizations such as prefetching, but this is not feasible at the level of virtual memory—disks simply can't keep up with the rate of allocation because of the enormous speed differential between RAM and disk.) *Generational* collectors address this problem by reusing a smaller amount of memory more often; they will be discussed in Sect. 4. (For historical reasons, is widely believed that only copying collectors can be made generational, but this is not the case. Generational mark-sweep collectors are somewhat harder to construct, but they do exist and are quite practical [DWH+90].

Finally, the temporal distribution of a simple tracing collector's work is also troublesome in an interactive programming environment; it can be very disruptive to a user's work to suddenly have the system become unresponsive and spend several seconds garbage collecting, as is common in such systems. For large heaps, the pauses may be on the order of seconds, or even minutes if a large amount of data is dispersed through virtual memory. Generational collectors alleviate this problem, because most garbage collections only operate on a subset of memory. Eventually they must garbage collect larger areas, however, and the pauses may be considerably longer. For real time applications, this may not be acceptable.

3 Incremental Tracing Collectors

For truly real-time applications, fine-grained incremental garbage collection appears to be necessary. Garbage collection cannot be carried out as one atomic action while the program is halted, so small units of garbage collection must be interleaved with small units of program execution. As we said earlier, it is relatively easy to make reference counting collectors incremental. Reference counting's problems with efficiency and effectiveness discourage its use, however, and it is therefore desirable to make tracing (copying or marking) collectors incremental.

In most of the following discussion, the difference between copying and mark-sweep collectors is not particularly important. The incremental tracing for garbage

[10] Slightly more complicated copying schemes appear to avoid this problem [Ung84, WM89], but [WLM92] demonstrates that *cyclic* memory reuse patterns can fare poorly in hierarchical memories because of recency-based (e.g., LRU) replacement policies. This suggests that freed memory should be reused in a LIFO fashion (i.e., in the opposite order of its previous allocation), if the entire reuse pattern can't be kept in memory.

detection is more interesting than the incremental reclamation of detected garbage.

The difficulty with incremental tracing is that while the collector is tracing out the graph of reachable data structures, the graph may change—the running program may *mutate* the graph while the collector "isn't looking." For this reason, discussions of incremental collectors typically refer to the running program as the *mutator* [DLM+78]. (From the garbage collector's point of view, the actual application is merely a coroutine or concurrent process with an unfortunate tendency to modify data structures that the collector is attempting to traverse.) An incremental scheme must have some way of keeping track of the changes to the graph of reachable objects, perhaps re-computing parts of its traversal in the face of those changes.

An important characteristic of incremental techniques is their degree of conservatism with respect to changes made by the mutator during garbage collection. If the mutator changes the graph of reachable objects, freed objects may or may not be reclaimed by the garbage collector. Some *floating garbage* may go unreclaimed because the collector has already categorized the object as live before the mutator frees it. This garbage *is* guaranteed to be collected at the next cycle, however, because it will be garbage at the *beginning* of the next collection.

3.1 Tricolor Marking

The abstraction of *tricolor marking* is helpful in understanding incremental garbage collection. Garbage collection algorithms can be described as a process of traversing the graph of reachable objects and coloring them. The objects subject to garbage collection are conceptually colored white, and by the end of collection, those that will be retained must be colored black. When there are no reachable nodes left to blacken, the traversal of live data structures is finished.

In a simple mark-sweep collector, this coloring is directly implemented by setting mark bits—objects whose bit is set are black. In a copy collector, this is the process of moving objects from fromspace to tospace—unreached objects in fromspace are considered white, and objects moved to tospace are considered black. The abstraction of coloring is orthogonal to the distinction between marking and copying collectors, and is important for understanding the basic differences between incremental collectors.

In incremental collectors, the intermediate states of the coloring traversal are important, because of ongoing mutator activity—the mutator can't be allowed to change things "behind the collector's back" in such a way that the collector will fail to find all reachable objects.

To understand and prevent such interactions between the mutator and the collector, it is useful to introduce a third color, grey, to signify that an object has been reached by the traversal, but that *its descendants may not have been.* That is, as the traversal proceeds outward from the roots, objects are initially colored grey. When they are scanned and pointers to their offspring are traversed, they are blackened and the offspring are colored grey.

In a copying collector, the grey objects are the objects in the unscanned area of tospace—the ones between the scan and free pointers. Objects that have been passed by the scan pointer are black. In a mark-sweep collector, the grey objects correspond to the stack or queue of objects used to control the marking traversal,

and the black objects are the ones that have been removed from the queue. In both cases, objects that have not been reached yet are white.

Intuitively, the traversal proceeds in a wavefront of grey objects, which separates the white (unreached) objects from the black objects that have been passed by the wave—that is, there are no pointers directly from black objects to white ones. This abstracts away from the particulars of the traversal algorithm—it may be depth-first, breadth-first, or just about any kind of exhaustive traversal. It is only important that a well-defined grey fringe be identifiable, and that the mutator preserve the invariant that no black object hold a pointer directly to a white object.

The importance of this invariant is that the collector must be able to assume that it is "finished with" black objects, and can continue to traverse grey objects and move the wavefront forward. If the mutator creates a pointer from a black object to a white one, it must somehow coordinate with the collector, to ensure that the collector's bookkeeping is brought up to date.

Figure 7 demonstrates this need for coordination. Suppose the object A has been completely scanned (and therefore blackened); its descendants have been reached and greyed. Now suppose that the mutator swaps the pointer from A to C with the pointer from B to D. The only pointer to D is now in a field of A, which the collector has already scanned. If the traversal continues without any coordination, C will be reached again (from B), and D will never be reached at all.

Incremental approaches There are two basic approaches to coordinating the collector with the mutator. One is to use a *read barrier*, which detects when the mutator attempts to access a pointer to a white object, and immediately colors the object grey; since the mutator can't read pointers to white objects, it can't install them in black objects. The other approach is more direct, and involves a *write barrier*—when the program attempts to write a pointer into an object, the write is trapped or recorded.

Write barrier approaches, in turn, fall into two different categories, depending on which aspect of the problem they address. To foil the garbage collector's marking traversal, it is necessary for the mutator to 1) write a pointer to a white object into a black object *and* 2) destroy the original pointer before the collector sees it.

If the first condition (writing the pointer into a black object) does not hold, no special action is needed—if there are other pointers to the white object from grey objects, it will be retained, and if not, it is garbage and needn't be retained anyway. If the second condition (obliterating the original path to the object) does not hold, the object will be reached via the original pointer and retained. The two write-barrier approaches focus on these two aspects of the problem.

Snapshot-at-beginning collectors ensure that the second condition cannot happen—rather than allowing pointers to be simply overwritten, they are first saved so that the collector can find them. Thus no paths to white objects can be broken without providing another path to the object for the garbage collector.

Incremental update collectors are still more direct in dealing with these troublesome pointers. Rather than saving copies of all pointers that are overwritten (because they *might* have already been copied into black objects) they actually record pointers stored into black objects, and catch the troublesome pointers at their destination,

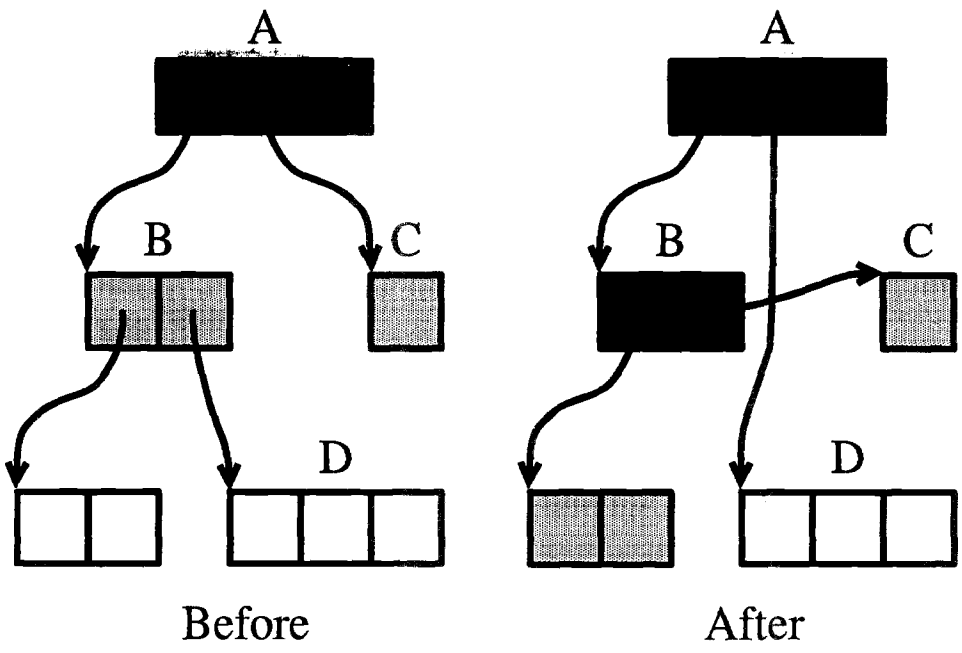


Fig. 7. A violation of the coloring invariant.

rather than their source. That is, if a pointer to a white object is copied into a black object, that new copy of the pointer will be found. Conceptually, the black object (or part of it) is reverted to grey when the mutator "undoes" the collector's traversal. (Alternatively, the pointed-to object may be greyed immediately.) This ensures that the traversal is updated in the face of mutator changes.

3.2 Baker's Incremental Copying.

The best-known real-time garbage collector is Baker's incremental copying scheme [Bak78]. It is an adaptation of the simple copy collection scheme described in Sect. 2.5, and uses a *read barrier* for coordination with the mutator. For the most part, the copying of data proceeds in the Cheney (breadth-first) fashion, by advancing the scan pointer through the unscanned area of tospace and moving any referred-to objects

from fromspace. This *background scavenging* is interleaved with mutator operation, however.

An important feature of Baker's scheme is its treatment of objects allocated by the mutator during incremental collection. These objects are allocated in tospace and are treated as though they had already been scanned—i.e., they are assumed to be live. In terms of tricolor marking, new objects are *black* when allocated, and none of them can be reclaimed; they are never reclaimed until the next garbage collection cycle.[11]

In order to ensure that the scavenger finds all of the live data and copies it to tospace before the free area in newspace is exhausted, the rate of copy collection work is tied to the rate of allocation. Each time an object is allocated, an increment of scanning and copying is done.

In terms of tricolor marking, the scanned area of tospace contains black objects, and the copied but unscanned objects (between the scan and free pointer) are grey. As-yet unreached objects in fromspace are white. The scanning of objects (and copying of their offspring) moves the wavefront forward.

In addition to the background scavenging, other objects may be copied to tospace as needed to ensure that the basic invariant is not violated—pointers into fromspace must not be stored into objects that have already been scanned, undoing the collector's work.

Baker's approach is to couple the collector's copying traversal with the mutator's traversal of data structures. The mutator is never allowed to see pointers into fromspace, i.e., pointers to white objects. Whenever the mutator reads a (potential) pointer from the heap, it immediately checks to see if it is a pointer into fromspace; if so, the referent is copied to tospace, i.e., its color is changed from white to grey. In effect, this advances the wavefront of greying just ahead of the actual references by the mutator, keeping the mutator inside the wavefront.[12]

It should be noted that Baker's collector itself changes the graph of reachable objects, in the process of copying. The read barrier does not just inform the collector of changes by the mutator, to ensure that objects aren't lost; it also shields the *mutator* from viewing temporary inconsistencies created by the collector. If this were not done, the mutator might encounter two different pointers to versions of the same object, one of them obsolete.

This shielding of the mutator from white objects has come to be called a *read barrier*, because it prevents pointers to white objects from being read by the program at all.

The read barrier may be implemented in software, by preceding each read (of a potential pointer from the heap) with a check and a conditional call to the copying-and-updating routine. (Compiled code thus contains extra instructions to implement

[11] Baker suggests copying old live objects into one end of tospace, and allocating new objects in the other end. The two occupied areas of tospace thus grow toward each other.

[12] Nilsen's variant of Baker's algorithm updates the pointers without actually copying the objects—the copying is lazy, and space in tospace is simply reserved for the object before the pointer is updated [Nil88]. This makes it easier to provide smaller bounds on the time taken by list operations, and to gear collector work to the amount of allocation—including guaranteeing shorter pauses when smaller objects are allocated.

the read barrier.) Alternatively, it may be implemented with specialized hardware checks and/or microcoded routines.

The read barrier is quite expensive on stock hardware, because in the general case, any load of a pointer must check to see if the pointer points to a fromspace (white) object; if so, it must execute code to move the object to tospace and update the pointer. The cost of these checks is high on conventional hardware, because they occur very frequently. Lisp Machines have special purpose hardware to detect pointers into fromspace and trap to a handler[Gre84, Moo84, Joh91], but on conventional machines the checking overhead is in the tens of percent for a high performance system.

Brooks has proposed a variation on Baker's scheme, where objects are *always* referred to via an indirection field embedded in the object itself [Bro84]. If an object is valid, its indirection field points to itself. If it's an obsolete version in tospace, its indirection pointer points to the new version. Unconditionally indirecting is cheaper than checking for indirections, but would still incur overheads in the tens of percent for a high-performance system. (A variant of this approach has been used by North and Reppy in a concurrent garbage collector [NR87].) Zorn takes a different approach to reducing the read barrier overhead, using knowledge of important special cases and special compiler techniques. Still, the time overheads are on the order of twenty percent [Zor89].

3.3 The Treadmill

Recently, Baker has proposed a non-copying version of his scheme, which uses doubly-linked lists (and per-object color fields) to implement the sets of objects of each color, rather than separate memory areas. By avoiding the actual moving of objects and updating of pointers, the scheme puts fewer restrictions on other aspects of language implementation.[13]

This non-copying scheme preserves the essential efficiency advantage of copy collection, by reclaiming space implicitly. (As described in Sect. 2.5, unreached objects on the allocated list can be reclaimed by appending the remainder of that list to the free list.) The real-time version of this scheme links the various lists into a cyclic structure, as shown in Fig. 8. This cyclic structure is divided into four sections.

The *new* list is where allocation of new objects occurs during garbage collection—it is contiguous with the free list, and allocation occurs by advancing the pointer that separates them. At the beginning of garbage collection, the new segment is empty.

The *from* list holds objects that were allocated before garbage collection began, and which are subject to garbage collection. As the collector and mutator traverse data structures, objects are moved from the from list to the to list. The to list is initially empty, but grows as objects are "unsnapped" (unlinked) from the from list (and snapped into the to list) during collection.

The new list contains new objects, which are allocated black. The to-list contains both black objects (which have been completely scanned) and grey ones (which have

[13] In particular, it is possible to deal with compilers that do not unambiguously identify pointer variables in the stack, making it impossible to use simple copy collection.

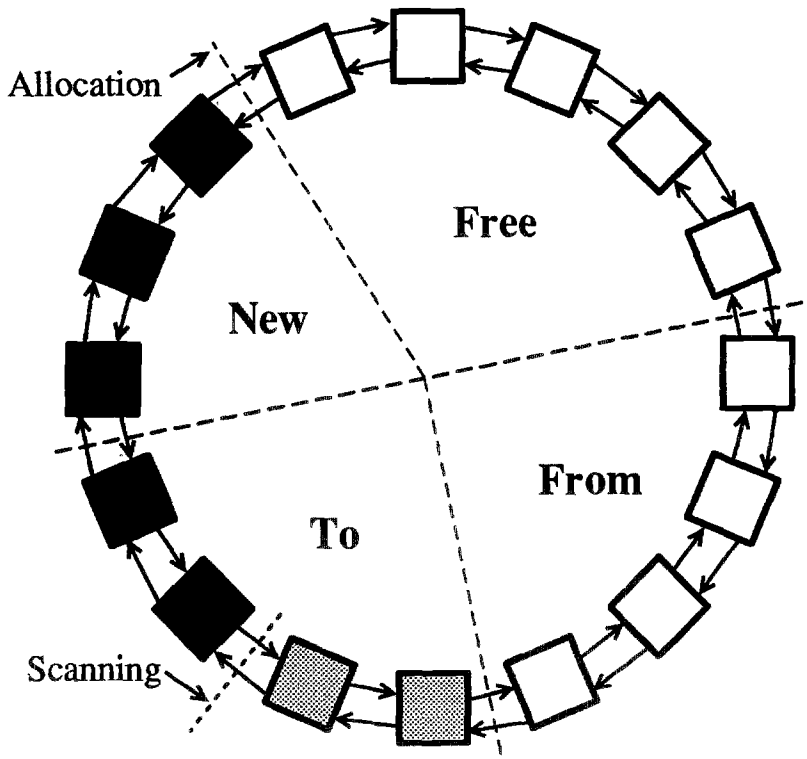


Fig. 8. Treadmill collector during collection.

been reached but not scanned). Note the isomorphism with the copying algorithm—even an analogue of the Cheney algorithm can be used. It is only necessary to have a scan pointer into the from list and advance it through the grey objects.

Eventually, all of the reachable objects in the from list have been moved to the to list, and scanned for offspring. When no more offspring are reachable, all of the objects in the to-list are black, and the remaining objects in the from list are known to be garbage. At this point, the garbage collection is complete. The from list is now available, and can simply be merged with the free list. The to list and the new list both hold objects that were preserved, and they can be merged to form the new to-list at the next collection.[14]

[14] This discussion is a bit oversimplified; Baker uses four colors, and whole lists can have their colors changed instantaneously by changing the sense of the bit patterns, rather

The state is very similar to the beginning of the previous cycle, except that the segments have "moved" partway around the cycle—hence the name "treadmill."

Baker describes this algorithm as being isomorphic to his original incremental copying algorithm, presumably including the close coupling between the mutator and the collector, i.e., the read barrier.

Conservatism in Baker's scheme. Baker's garbage collector uses a somewhat conservative approximation of true liveness in two ways.[15] The most obvious one is that objects allocated during collection are assumed to be live, even if they die before the collection is finished. The second is that pre-existing objects may become garbage after having been reached by the collector's traversal, and they will not be reclaimed—once an object has been greyed, it will be considered live until the next garbage collection cycle. On the other hand, if objects become garbage during collection, and all paths to those objects are destroyed *before* being traversed, then they *will* be reclaimed. That is, the mutator may overwrite a pointer from a grey object, destroying the only path to one or more white objects and ensuring that the collector will not find them. Thus Baker's incremental scheme incrementally updates the reachability graph of pre-existing objects, only when grey objects have pointers overwritten. Overwriting pointers from black objects has no effect, however, because their referents are already grey. The degree of conservatism (and floating garbage) thus depends on the details of the collector's traversal and of the program's actions.

3.4 Snapshot-at-Beginning write-barrier algorithms

If a non-copying collector is used, the use of a read barrier is an unnecessary expense; there is no need to protect the mutator from seeing an invalid version of a pointer. *Write barrier* techniques are cheaper, because heap writes are several times less common than heap reads. *Snapshot-at-beginning* algorithms use a write barrier to ensure that *no* objects ever become inaccessible to the garbage collector while collection is in progress. Conceptually, at the beginning of garbage collection, a *copy-on-write* virtual copy of the graph of reachable data structures is made. That is, the graph of reachable objects is fixed at the moment garbage collection starts, even though the actual traversal proceeds incrementally.

Perhaps the simplest and best-known snapshot collection algorithm is Yuasa's [Yua90]. If a location is written to, the overwritten value is first saved and pushed on a marking stack for later examination. This guarantees that no objects will become unreachable to the garbage collector traversal—all objects live at the beginning of garbage collection will be reached, even if the pointers to them are overwritten. In the example shown in Fig. 7, the pointer from B to D is pushed onto the stack when it is overwritten with the pointer to C.

Yuasa's scheme has a large advantage over Baker's on stock hardware, because only heap pointer writes must be treated specially to preserve the garbage collector

---

than the patterns themselves.

[15] This kind of conservatism is not to be confused with the conservative treatment of pointers that cannot be unambiguously identified. (For a more complete and formal discussion of various kinds of conservatism in garbage collection, see [DWH+90].)

invariants. Normal pointer dereferencing and comparison does not incur any extra overhead.

On the other hand, Yuasa's scheme is more conservative than Baker's. Not only are all objects allocated during collection retained, but *no* objects can be freed during collection—all of the overwritten pointers are preserved and traversed. These objects are reclaimed at the next garbage collection cycle.

3.5 Incremental Update Write-Barrier Algorithms

While both are write-barrier algorithms, snapshot-at-beginning and *incremental update* algorithms are quite different. Unfortunately, incremental update algorithms have generally been cast in terms of parallel systems, rather than as incremental schemes for serial processing; perhaps due to this, they have been largely overlooked by implementors targeting uniprocessors.

Perhaps the best known of these algorithms is due to Dijkstra *et al.* [DLM+78]. (This is similar to the scheme developed independently by Steele [Ste75], but simpler because it does not deal with compactification.) Rather than retaining everything that's in a snapshot of the graph at the *beginning* of garbage collection, it heuristically (and somewhat conservatively) attempts to retain the objects that are live at the *end* of garbage collection. Objects that die during garbage collection—and before being reached by the marking traversal—are not traversed and marked.

To avoid the problem of pointers escaping into reachable objects that have already been scanned, such copied pointers are caught at their *destination*, rather than their source. Rather than noticing when a pointer escapes *from* a location that hasn't been traversed, it notices when the pointer escapes *into* an object that *has* already been traversed. If a pointer is overwritten without being copied elsewhere, so much the better—the object is garbage, so it might as well not get marked.

If the pointer is installed into an object already determined to be live, that pointer must be taken into account—it has now been incorporated into the graph of reachable data structures. Such pointer stores are recorded by the write barrier—the collector is notified which black objects may hold pointers to white objects, in effect reverting those objects to grey. Those formerly-black objects will be scanned again before the garbage collection is complete, to find any live objects that would otherwise escape. (This process may iterate, because more black objects may be reverted while the collector is in the process of traversing them. The traversal is guaranteed to complete, however, and the collector eventually catches up with the mutator.)

Objects that become garbage during garbage collection may be reclaimed at the end of that garbage collection, not the next one. This is similar to Baker's read-barrier algorithm in its treatment of pre-existing objects—they are not preserved if they become garbage before being reached by the collector.

It is less conservative than Baker's and Yuasa's algorithms in its treatment of objects allocated by the mutator during collocation, however. Baker's and Yuasa's schemes assume such newly-created objects are live, because pointers to them may get installed into objects that have already been reached by the collector's traversal. In terms of tricolor marking, objects are allocated "black", rather than white—they are conservatively assumed to be part of the graph of reachable objects. (In Baker's

algorithm, there is no write barrier to detect whether they have been incorporated into the graph or not.)

In the Dijkstra *et al.* scheme, objects are assumed *not* to be reachable when they're allocated. In terms of tricolor marking, objects are allocated *white*, rather than black. At some point, the stack must be traversed and the objects that are reachable *at that time* are marked and therefore preserved.

We believe that this has a potentially significant advantage over Baker's or Yuasa's schemes. Most objects are short-lived, so if the collector doesn't reach those objects early in its traversal, they're likely never to be reached, and instead to be reclaimed very promptly. Compared to Baker's or Yuasa's scheme, there's an extra computational cost—by assuming that *all* objects allocated during collection are reachable, those schemes avoid the cost of traversing and marking those that actually *are* reachable. On the other hand, there's a space benefit with the incremental update scheme—the majority of those objects can be reclaimed at the end of a collection, which is likely to make it worth traversing the others. (In Steele's algorithm, some objects are allocated white and some are not, depending on the colors of their referents [Ste75]. This heuristic attempts to allocate short-lived objects white to reclaim their space quickly, while treating other objects conservatively to avoid traversing them. The cost of this technique is not quantified, and its benefits are unknown.)

3.6 Choosing Among Incremental Techniques

In choosing an incremental collection design, it is instructive to keep in mind the abstraction of tricolor marking, as distinct from mechanisms such as mark-sweep or copy collection. For example, Brooks' collector [Bro84] is actually a write barrier algorithm, even though Brooks describes it as an optimization of Baker's scheme.[16] Similarly, Dawson's[Daw82] copy collection scheme is cast as a variant of Baker's, but it is actually an incremental update scheme, similar to Dijkstra *et al.*'s; objects are allocated in fromspace, i.e., white.

The choice of a read- or write-barrier scheme is likely to be made on the basis of the available hardware. Without specialized hardware support, a write barrier appears to be easier to implement efficiently, because heap pointer writes are much less common than pointer traversals.

Appel, Ellis and Li [AEL88] use virtual memory (pagewise) access protection facilities as a coarse approximation of Baker's write barrier[AEL88, AL91, Wil91]. Rather than checking each load to see if a pointer to fromspace is being loaded, the mutator is simply not allowed to see any page that might contain such a pointer. Pointers in the scanned area of tospace are guaranteed to contain only pointers into tospace. Any pointers from fromspace to tospace must be from the unscanned area, so the collector simply access-protects the unscanned area, i.e., the grey objects. When the mutator accesses a protected page, a trap handler immediately scans the

[16] The use of uniform indirections may be viewed as *avoiding* the need for a Baker-style read barrier—the indirections isolate the collector from changes made by the mutator, allowing them to be decoupled. The actual coordination, in terms of tricolor marking, is through a write barrier.

whole page, fixing up all the pointers (i.e., blackening all of the objects in the page); referents in fromspace are relocated to tospace (i.e., greyed) and access-protected.

Unfortunately this scheme fails to provide meaningful real-time guarantees in the general case. (It does support concurrent collection, however, and greatly reduces the cost of the read barrier.) In the worst case, each pointer traversal may cause the scanning of a page of tospace until the whole garbage collection is complete.[17]

Of write barrier schemes, incremental update appears to be more effective than snapshot approaches—because most short-lived objects are reclaimed quickly—but with an extra cost in traversing newly-allocated live objects. This cost might be reduced by carefully choosing the ordering of root traversal, traversing the most stable structures first to avoid having the collector's work undone by mutator changes.

Careful attention should be paid to write barrier implementation. Boehm, Demers and Shenker's [BDS91, Boe91] incremental update algorithm uses virtual memory dirty bits as a coarse pagewise write barrier. All black objects in a page must be re-scanned if the page is dirtied again before the end of a collection. (As with Appel, Ellis and Li's copy collector, this coarseness sacrifices real-time guarantees, while supporting parallelism. It also allows the use of off-the-shelf compilers that don't emit write barrier instructions along with heap writes.)

In a system with compiler support for garbage collection, a list of stored-into locations can be kept, or dirty bits can maintained (in software) for small areas of memory, to reduce scanning costs and bound the time spent updating the marking traversal. This has been done for other reasons in generational garbage collectors, as we will discuss in Sect. 4.

4 Generational Garbage Collection

Given a realistic amount of memory, efficiency of simple copying garbage collection is limited by the fact that the system must copy all live data at a collection. In most programs in a variety of languages, *most objects live a very short time, while a small percentage of them live much longer* [LH83, Ung84, Sha88, Zor90, DeT90, Hay91]. While figures vary from language to language and program to program, usually between 80 and 98 percent of all newly-allocated objects die within a few million instructions, or before another megabyte has been allocated; the majority of objects die even more quickly, within tens of kilobytes of allocation.

(Heap allocation is often used as a measure of program execution, rather than wall clock time, for two reasons. One is that it's independent of machine and implementation speed—it varies appropriately with the speed at which the program executes, which wall clock time does not; this avoids the need to continually cite hardware speeds.[18] It is also appropriate to speak in terms of amounts allocated for

[17] Ralph Johnson has improved on this scheme by incorporating lazier copying of objects to fromspace [Joh92]. This decreases the maximum latency, but in the (very unlikely) worst case a page may still be scanned at each pointer traversal until a whole garbage collection has been done "the hard way".

[18] One must be careful, however, not to interpret it as the ideal abstract measure. For example, rates of heap allocation are somewhat higher in Lisp and Smalltalk, because more control information and/or intermediate data of computations may be passed as pointers to heap objects, rather than as structures on the stack.

garbage collection studies because the time between garbage collections is largely determined by the amount of memory available.[19] Future improvements in compiler technology may reduce rates of heap allocation by putting more "heap" objects on the stack; this is not yet much of a problem for experimental studies, because most current state-of-the-art compilers don't do much of this kind of lifetime analysis.)

Even if garbage collections are fairly close together, separated by only a few kilobytes of allocation, most objects die before a collection and never need to be copied. Of the ones that do survive to be copied once, however, *a large fraction survive through many collections.* These objects are copied at every scavenge, over and over, and the garbage collector spends most of its time copying the same old objects repeatedly. This is the major source of inefficiency in simple garbage collectors.

Generational collection [LH83] avoids much of this repeated copying by segregating objects into multiple areas by age, and scavenging areas containing older objects less often than the younger ones. Once objects have survived a small number of scavenges, they are moved to a less frequently scavenged area. Areas containing younger objects are scavenged quite frequently, because most objects there will generally die quickly, freeing up space; copying the few that survive doesn't cost much. These survivors are *advanced* to older status after a few scavenges, to keep copying costs down.

(For historical reasons and simplicity of explanation, we will focus on generational copying collectors. The choice of copying or marking collection is essentially orthogonal to the issue of generational collection, however [DWH+90].)

4.1 Multiple Subheaps with Varying Scavenge Frequencies

Consider a generational garbage collector based on the semispace organization: memory is divided into areas that will hold objects of different approximate ages, or *generations*; each generation's memory is further divided into semispaces. In Fig. 9 we show a simple generational scheme with just two age groups, a New generation and an Old generation. Objects are allocated in the New generation, until its current semispace is full. Then the New generation (only) is scavenged, copying its live data into the other semispace, as shown in Fig. 10.

If an object survives long enough to be considered old, it can be copied out of the new generation and into the old, rather than back into the other semispace. This removes it from consideration by single-generation scavenges, so that it is no longer copied at every scavenge. Since relatively few objects live this long, old memory will fill much more slowly than new. Eventually, old memory will fill up and have to be garbage collected as well. Figure 11 shows the general pattern of memory use in this simple generational scheme. (Note the figure is not to scale—the younger generation is typically several times smaller than the older one.)

The number of generations may be greater than two, with each successive generation holding older objects and being scavenged considerably less often. (Tektronix

[19] Allocation-relative measures are still not the absolute bottom-line measure of garbage collector efficiency, though, because decreasing work per unit of allocation is not nearly as important if programs don't allocate much; conversely, smaller percentage changes in garbage collection work mean more for programs whose memory demands are higher.

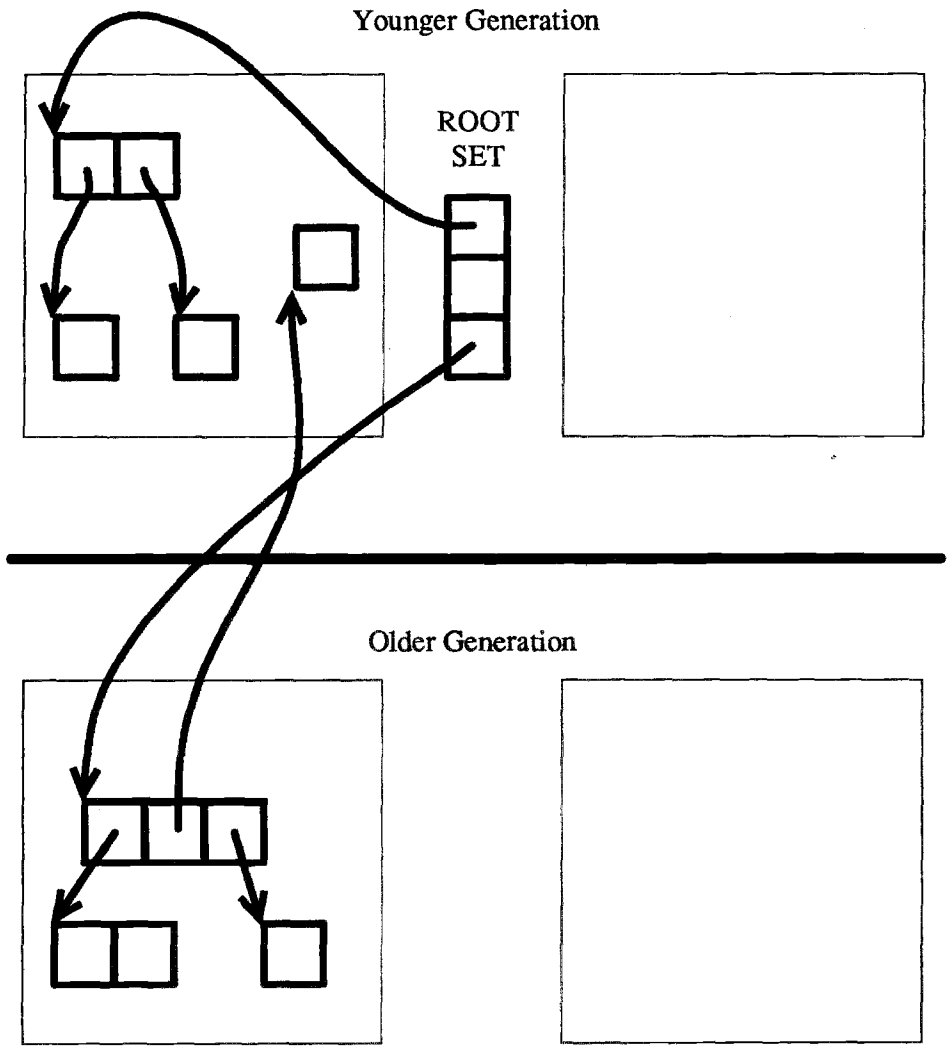


Fig. 9. A generational copying garbage collector before garbage collection.

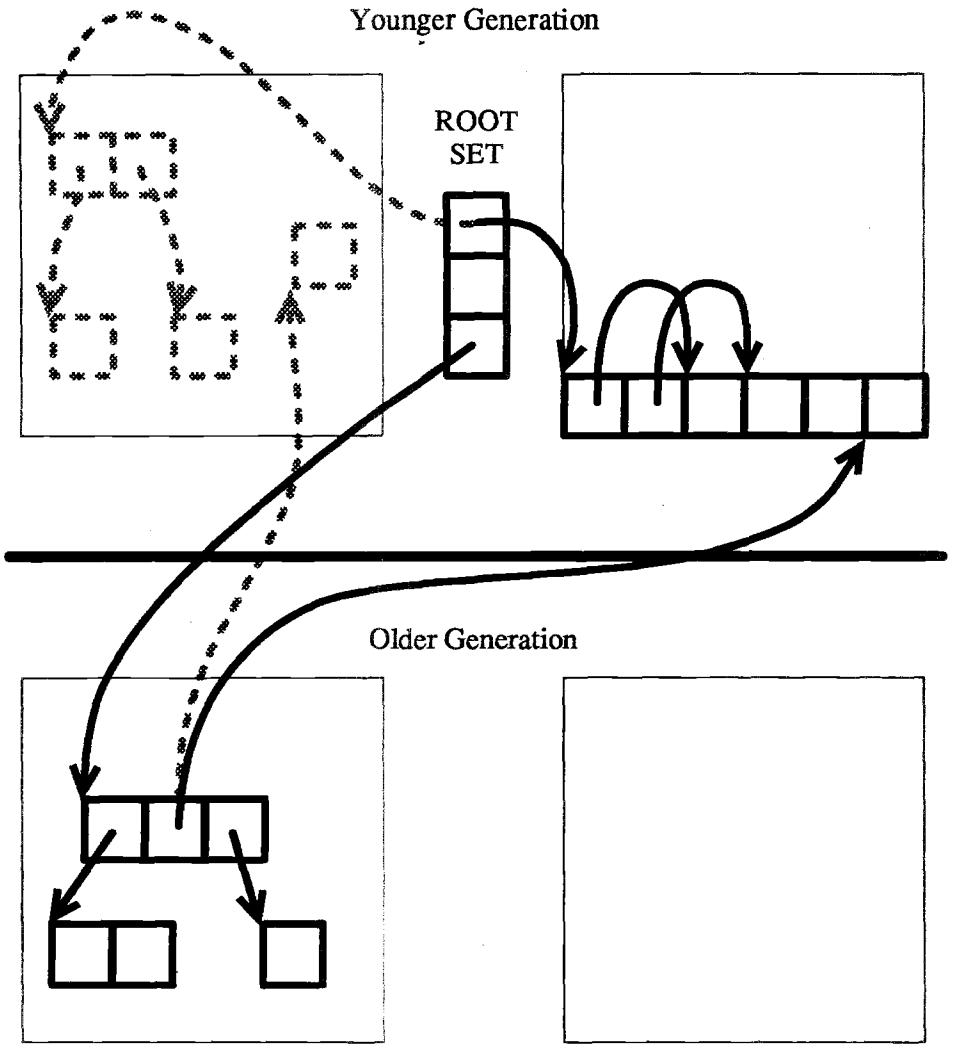


Fig. 10. Generational collector after garbage collection.

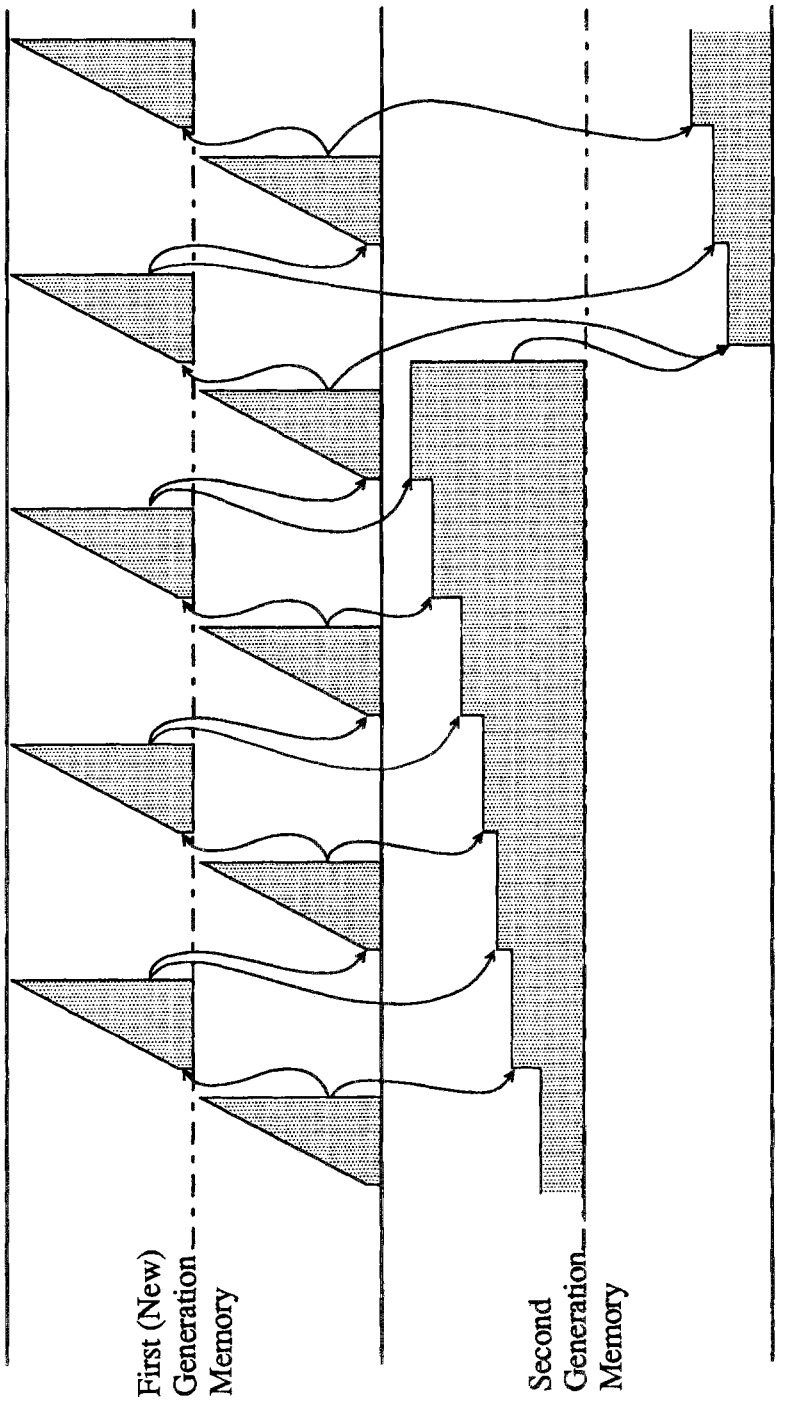


Fig. 11. Memory use in a generational copy collector with semispaces for each generation.

4406 Smalltalk is such a generational system, using semispaces for each of eight generations [CWB86].)

4.2 Detecting Intergenerational References

In order for this scheme to work, it must be possible to scavenge the younger generation(s) without scavenging the older one(s). Since liveness of data is a global property, however, old-memory data must be taken into account. For example, if there is a pointer from old memory to new memory, that pointer must be found at scavenge time and used as one of the roots of the traversal. (Otherwise, an object that is live may not be preserved by the garbage collector, or the pointer may simply not be updated appropriately when the object is moved. Either event destroys the integrity and consistency of data structures in the heap.)

In the original generational collection scheme [LH83] scheme, no pointer in old memory may point directly to an object in new memory; instead it must point to a cell in an indirection table, which is used as part of the root set. Such indirections are transparent to the user program. This technique was implemented on Lisp machines such as the MIT machines [Gre84] and Texas Instruments Explorer [Cou88]. (There are minor differences between the two, but the principles are the same.[20])

Note that other techniques are often more appropriate, especially on stock hardware. Using indirection tables introduces overhead similar to that of Baker's read barrier. A *pointer recording* technique can be used instead. Rather than indirecting pointers from old objects to young ones, normal (direct) pointers are allowed, but the locations of such pointers are noted so that they can be found at scavenge time. This requires something like a write barrier [Ung84, Moo84]; that is, the running program cannot freely modify the reachability graph by storing pointers into objects in older generation.

The write barrier may do checking at each store, or it may be as simple as maintaining dirty bits and scanning dirty areas at collection time [Sha88, Sob88, WM89, Wil90].[21]; the same mechanism might support real-time incremental collection as well.

The important point is that all references from old to new memory must be located at scavenge time, and used as roots for the copying traversal.

Using these intergenerational pointers as roots ensures that all reachable objects in the younger generation are actually reached by the collector; in the case of a copy collector, it ensures that all pointers to moved objects are appropriately updated.

As in an incremental collector, this use of a write barrier results in a *conservative approximation* of true liveness; any pointers from old to new memory are used as

[20] The main difference is that the original scheme used per-generation *entry* tables, indirecting and isolating the pointers into a generation. The Explorer used *exit* tables, indirecting the pointers *out of* each generation; for each generation, there is a separate exit table for pointers into *each* younger generation[Cou88].

[21] Ungar and Chambers' improvement [Cha92], of our "card marking" scheme [WM89, Wil90] decreases the cost per heap write by using whole bytes as dirty bits. Given the byte write instructions available on common architectures, the overhead is only three instructions per potential pointer store, at an increase in bitmap size and per-garbage collection scanning cost.

roots, but not all of these roots are necessarily live themselves. An object in old memory may already have died, but that fact is unknown until the next time old memory is scavenged. Thus some garbage objects may be preserved because they are referred to from objects that are floating (undetected) garbage. This appears not to be a problem in practice [Ung84, UJ88].

It would also be possible to track all pointers from new memory into old memory, allowing old memory to be scavenged independently of new memory. This is more costly, however, because there are typically many more pointers from old to new than from new to old. This is a consequence of the way references are typically created—by creating a new object that refers to other objects which already exist. Sometimes a pointer to a new object is installed in an old object, but this is considerably less common. This asymmetrical treatment allows allows object-creating code (like Lisp's frequently-used **cons** operation) to skip the recording of intergenerational pointers. Only non-initializing stores into objects must be checked for intergenerational references; writes that initialize objects in the youngest generation can't create pointers into younger ones.

Even if new-to-old pointers are not recorded, it may still be feasible to scavenge a generation without scavenging newer ones. In this case, *all* data in the newer generations may be considered possible roots, and they may simply be scanned for pointers [LH83]. While this scanning consumes time proportional to the amount of data in the newer generations, each generation is usually considerably smaller than the next, and the cost may be small relative to the cost of actually scavenging the older generation. (Scanning the data in the newer generation may be preferable to scavenging both generations, because scanning is generally faster than copying; it may also have better locality.)

The cost of recording intergenerational pointers is typically proportional to the rate of program execution i.e., it's not particularly tied to the rate of object creation. For some programs, it may be the major cost of garbage collection, because several instructions must be executed for every potential pointer store into the heap. This may slow program execution down by several percent. (It is interesting to note that this pointer recording is essentially the same as that required for a write barrier incremental scheme; the same cost may serve both purposes.)

Within the framework of the generational strategy we've outlined, several important questions remain:

1. *Advancement policy.* How long must an object survive in one generation before it is advanced to the next? [Ung84, WM89]
2. *Heap organization.* How should storage space be divided and used between generations, and within a generation [Moo84, Ung84, Sha88, WM89]? How does the resulting reuse pattern affect locality at the virtual memory level [Ung84, Zor89, WM89], and at the level of high-speed cache memories [Zor91, WLM92]?
3. *Traversal algorithms.* In a tracing collector, the traversal of live objects may have an important impact on locality. In a copying collector, objects are also reordered in memory as they are reached by the copy collector. What affect does this have on locality, and what traversal yields the best results [Bla83, Sta84, And86, WLM91]?
4. *Collection scheduling.* For a non-incremental collector, how might we avoid or

mitigate the effect of disruptive pauses, especially in interactive applications [Ung84, WM89]? Can we improve efficiency by careful "opportunistic" scheduling [WM89, Hay91]? Can this be adapted to incremental schemes to reduce floating garbage?

5. *Intergenerational references.* Since it must be possible to scavenge younger generations without scavenging the older ones, we must be able to find the live pointers from older generations into the ones we're scavenging. What is the best way to do this [WM89, BDS91, App89b, Wil90]?

5 Conclusions

Recent advances in garbage collection technology make automatic storage reclamation affordable for use in high-performance systems. Even relatively simple garbage collectors' performance is often competitive with conventional explicit storage management [App87, Zor92]. Generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically observed tendency of objects to die young; stock hardware incremental techniques may even make this relatively inexpensive for hard real-time systems.

We have discussed the basic operation of several kinds of garbage collectors, to provide a framework for understanding current research in the field. A key point is that standard textbook analyses of garbage collection algorithms usually miss the most important characteristics of collectors—namely, the constant factors associated with the various costs, including locality effects. These factors require garbage collection designers to take detailed implementation issues into account, and be very careful in their choices of features.

Features also interact in important ways. Fine-grained incremental collection is unnecessary in most systems without hard real-time constraints. Coarser incremental techniques may be sufficient, because the modest pause times are acceptable [AEL88, BDS91], and the usually-short pauses of a stop-and-collect generational system may be acceptable enough for many systems [Ung84, WM89]. (On the other hand, the write barrier support for generational garbage collection could also support an incremental update scheme for incremental collection; if this recording is cheap and precise enough, it might support fine-grained real-time collection at little cost.)

In this introductory survey, we have not addressed the increasingly important areas of parallel [Ste75, KS77, DLM+78, NR87, AEL88, SS91] and distributed [LQP92, RMA92, JJ92, PS92] collection; we have also given insufficient coverage of conservative collectors, which can be used with systems not originally designed for garbage collection [BW88, Bar88, Ede90, Wen90, WH91]. These developments have considerable promise for making garbage collection widely available and practical; we hope that we've laid a proper foundation for discussing them, by clarifying the basic issues.

Acknowledgments

I am grateful to innumerable people for enlightening discussions of heap management over the last few years, including David Ungar, Eliot Moss, Henry Baker, Andrew

Appel, Urs Hoelzle, Mike Lam, Tom Moher, Henry Lieberman, Patrick Sobalvarro, Doug Johnson, Bob Courts, Ben Zorn, Mark Johnstone and David Chase. Special thanks to Hans Boehm, Joel Bartlett, David Moon, Barry Hayes, and especially to Janet Swisher for help in the preparation of this paper.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, Santa Clara, California, April 1991.

[And86] David L. Andre. Paging in Lisp programs. Master's thesis, University of Maryland, College Park, Maryland, 1986.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App89a] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[App89b] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.

[App91] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*, pages 89–100. MIT Press, Cambridge, MA, 1991.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak92] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988.

[BDS91] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, Toronto, Ontario, Canada, June 1991.

[Bla83] Ricki Blau. Paging on an object-oriented personal computer for Smalltalk. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Minneapolis, Minnesota, August 1983. Also appears as Technical Report UCB/CSD 83/125, University of California at Berkeley, Computer Science Division (EECS), Berkeley, California, August 1983.

[Bob80] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[Boe91] Hans-Juergen Boehm. Hardware and operating system support for conservative garbage collection. In *IEEE International Workshop on Object Orientation In Operating Systems*, Palo Alto, California, October 1991. IEEE Press.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 108–113, Austin, Texas, August 1984.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language.* PhD thesis, Stanford University, March 1992.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In Norman Meyrowitz, editor, *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 119–130, September 1986. Also published as *ACM SIGPLAN Notices 21*(11):119-130, November, 1986.

[Daw82] Jeffrey L. Dawson. Improved effectiveness from a real-time LISP garbage collector. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 159–167, August 1982.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DeT90] John DeTreville. Experience with concurrent garbage collectors for modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conf. Record of the Seventeeth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, Las Vegas, Nevada, January 1990.

[Ede90] Daniel Ross Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly-typed programming languages. In *SIGPLAN Symposium on Programming Language Design*

and Implementation, pages 165–176, June 1991. Toronto, Ontario, Canada.

[Gre84] Richard Greenblatt. *The LISP Machine*. McGraw Hill, 1984. D.R. Barstow, H.E. Shrobe, E. Sandewall, eds.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[JJ92] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[Joh91] Douglas Johnson. The case for a read barrier. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, Santa Clara, California, April 1991.

[Joh92] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1(1):46–58, March 1992.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, chapter 2.3.5, pages 406–422. Addison-Wesley, Reading, Massachusetts, 1969.

[KS77] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposum on Foundations of Computer Science*, pages 120–131, Providence, Rhode Island, October 1977.

[Lar77] R. G. Larson. Minimizing garbage collection as a function of region size. *SIAM Journal on Computing*, 6(4):663–667, December 1977.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming*, pages 39–50, Albuquerque, New Mexico, January 1992.

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Min63] Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I Memo 58, Project MAC, MIT, Cambridge, Massachusetts, 1963.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Nil88] Kelvin Nilsen. Garbage collection of strings and linked data structures in real time. *Software, Practice and Experience*, 18(7):613–640, July 1988.

[NR87] S. C. North and J. H. Reppy. *Concurrent Garbage Collection on Stock Hardware*, pages 113–133. Number 274 in Lecture Notes in Computer Science. Springer-Verlag, September 1987.

[PS92] David Plainfosse and Marc Shapiro. Experience with fault tolerant garbage collection in a distributed Lisp system. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science Series.

[RMA92] G. Ringwood, E. Miranda, and S. Abdullahi. Distributed garbage collection. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, California, February 1988. Also appears as Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory, 1988.

[Sob88] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science Department, Cambridge, Massachusetts, 1988.

[SS91] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 16–32, Phoenix, Arizona, October 1991.

[Sta84] James William Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Programming Languages and Systems*, 2(2):155–180, May 1984.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 1–17, San Diego, California, September 1988. ACM. Also published as *ACM SIGPLAN Notices 23*(11):1–17, November, 1988.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. Also distributed as *ACM SIGPLAN Notices 19*(5):157–167, May, 1987.

[Wen90] E.P. Wentworth. Pitfalls of conservative garbage collection. *Software, Practice and Experience*, 20(7):719–727, July 1990.

[WH91] Paul R. Wilson and Barry Hayes. The 1991 OOPSLA workshop on garbage collection in object oriented systems. In *Addendum to the proceedings of OOPSLA '91*, Phoenix, Arizona, 1991.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Ontario, Canada, October 1990. Also in *SIGPLAN Notices 23*(1):45–52, January 1991.

[Wil91] Paul R. Wilson. Operating system support for small objects. In *IEEE International Workshop on Object Orientation In Operating Systems*, Palo Alto, California, October 1991. IEEE Press. Revised version to appear in *Computing Systems.*

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *SIGPLAN Symposium on LISP and Functional Programming*, San Francisco, California, 1992.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN Conference on Object Oriented Programming Sys-*

tems, Languages and Applications (OOPSLA '89), pages 23–35, New Orleans, Louisiana, October 1989.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.

[Zor89] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Electrical Engineering and Computer Science Department, Berkeley, California, December 1989. Also appears as Technical Report UCB/CSD 89/544, University of California at Berkeley.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.

[Zor91] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, May 1991.

[Zor92] Benjamin Zorn. The measured cost of conservative garbage collection. Technical report, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, 1992.

This article was processed using the LaTeX macro package with LLNCS style

COLLECTION SCHEMES FOR DISTRIBUTED GARBAGE

Saleh E Abdullahi, Eliot E Miranda and Graem A Ringwood
Department of Computer Science
Queen Mary and Westfield College
University of London
LONDON E1 4NS
yakubu | eliot | gar @dcs.qmw.ac.uk

Abstract: With the continued growth in interest in distributed systems, garbage collection is actively receiving attention by designers of distributed languages [Bal, 1990]. Distribution adds another dimension of complexity to an already complex problem. A comprehensive review and bibliography of distributed garbage collection literature up to 1992 is presented. As distributed collectors are largely based on nondistributed collectors these are first briefly reviewed. Emphasis is given to collectors which appeared since the last major review [Cohen, 1981]. Collectors are broadly classified as those that identify garbage directly and those that identify it indirectly. Distributed collectors are reviewed on the basis of the taxonomy drawn up for nondistributed collectors.

1.0 Introduction

Garbage collection is a necessary evil of computer languages which employ dynamic data structures. Abstractly, the state of a computation expressed in such languages can be understood as a rooted, connected, directed graph. Some edges, *roots*, are distinguished in that they provide entry points into the graph. The vertices of the computation graph are represented by *cells*, the units of allocation and deallocation of contiguous segments of store. (Nothing will be assumed about the sizes of cells.) Edges of the graph are represented by pointer fields

within cells. Roots are pointers to vertices from the execution stack, global variables or registers. As a computation proceeds the graph changes by the addition and deletion of vertices and edges. As a result, some portions of the graph become disconnected. These disconnected subgraphs are known as *garbage.*

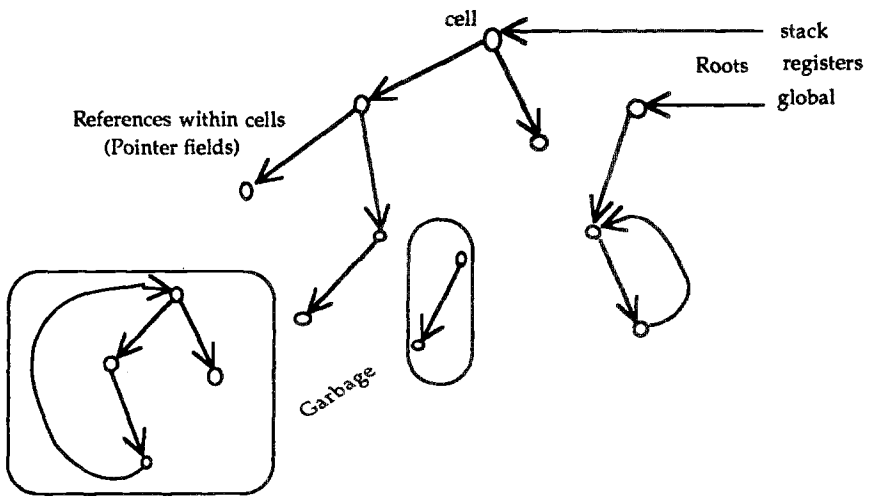


Fig 1. A representative, though small, state of a computation.

Without reutilization, the finite store available for allocating new vertices diminishes to zero. The process by which the store occupied by discarded cells can be reutilized is called *garbage collection.*

The earliest forms of store management placed the responsibility for allocation and reclamation on the programmer. Today this is considered too errorprone if not burdensome and a wide variety of languages provide automatic allocation and reclamation as part of their runtime system. Recent reports for various languages are: Smalltalk [Krasner, 1983; Ungar, 1984; Caudill, 1986; Miranda, 1987]; Prolog [Appleby et al, 1988]; ML [Li, 1990]; C++ [Bartlett, 1990; Detlefs, 1990a; Edelson and Pohl, 1990], Modula-2+ [DeTreville, 1990; Juul, 1990] and Modula-3 [Hudson and Diwan, 1990].

An important addition to the terminology of garbage collection was introduced by Dijkstra [1978]. The process which adds new vertices and adds and deletes edges is called the *mutator.* The mutator is an abstraction of the running program. The process which reclaims garbage is called the *collector.* Historically, the major

disadvantage of automatic collection was that it significantly detracted from the performance of the mutator, both by introducing unpredictable, long pauses and using large proportions of available processing cycles. Measurements of early Smalltalk-80 implementations indicate that 20% to 70% of the time was spent collecting garbage [Krasner, 1983]. For Lisp, collection overheads of between 10% and 40% were reported [Steele, 1975; Wadler, 1976], with pause times of 4.5 seconds every 79 seconds [Foderaro, 1981]. Over the previous decade much progress has been made; and the current state-of-the-art for Smalltalk-80 is less than 5% collector overhead, with typically better than 100 millisecond pause-times [Ungar, 1992].

Efficient garbage collection is so useful and so difficult to make unobtrusive that it has been a field of active research for over three decades. It constitutes a major concern for language designers. Knuth [1973] invented some and analysed other collectors which appeared prior to 1968. Cohen [1981] performed a public service with a survey of papers up to 1981. While there have since been numerous papers on garbage collection, they have tended to be language specific. Some languages allow optimizations which are not generally applicable. The semantics of a language does restrict the topology of the computation graph and graphs may be: cyclic; acyclic or tree-like. The topology in turn restricts the type of collector which can be employed.

A significant complication to the problem of garbage collection since Cohen [1981] has arisen with the spreading web of distributed systems [Bal, 1990]. According to Bal: "A distributed computing system consists of multiple autonomous processors, *nodes*, that do not share primary memory, but cooperate by sending messages over a communications network." The advantages of distribution are:

- improved performance through parallelism;
- increased availability and reliability through redundancy;
- reduced communication by dispersion of processing power to where it is needed and
- incremental growth through the addition of nodes and communication links.

The convincing factor is the economic consequences to which these advantages give rise. While distributed applications can be built directly on top of operating systems, Bal [1990] puts forward convincing arguments for programming

languages which contain all the necessary constructs for distributed programming. For such languages, the computation graph is distributed over a number of *nodes*. The absence of a homogeneous address space and the high cost of communication relative to local computation make distributed garbage collection a significantly more complex problem than collection on a single node.

The purpose of this paper is to give as comprehensive as possible a review and bibliography of distributed garbage collectors, subject to space limitations. As distributed collectors are generally based on nondistributed collectors the latter are first briefly classified. Special attention is given to incremental and concurrent collectors which are directly relevant to distribution. Emphasis will be placed on papers and trends published since Cohen [1981]. The majority of these papers relate to object-oriented languages, but for the ideal of treating different languages uniformly, herein, objects will be referred to as cells. The final section reviews distributed collectors on the basis of the taxonomy drawn up for single node collectors.

2.0 Single Node Collectors

Following Cohen [1981] the collection process consists of:

1) *identification* and
2) *reclamation* of garbage for reuse.

The way in which garbage is identified distinguishes two classes of collectors. Garbage identification can be made *directly*, identifying cells that become disconnected from the computation graph or *indirectly* by identifying the cells forming the computation graph; what then remains must be garbage (and unallocated store).

The form of reclamation is dependent on how the free store is managed. It can either be managed as a *freelist* (equally well a bitmap or buddy system) or a *heap*. If managed by a freelist, garbage is coalesced into the list. If managed as a *heap*, the division between the allocated and unallocated store is indicated by a single pointer, the *top of heap*, and reclamation can be performed either by compacting or by copying.

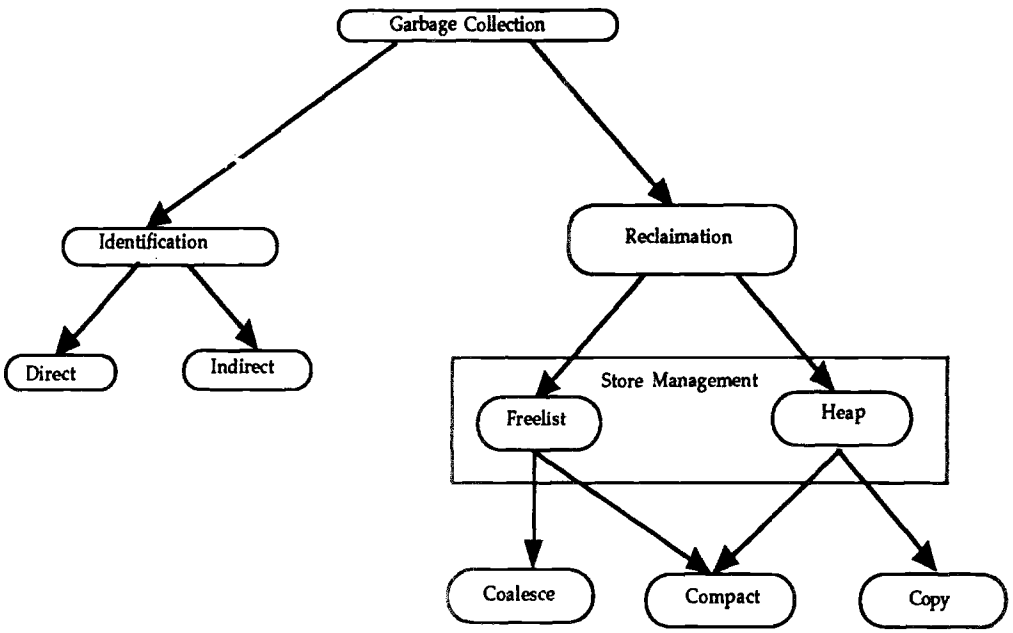


Fig 2. The garbage collection problem.

Various collectors have been proposed which seek to optimise different criteria. Some aim to minimize the total percentage time spent collecting garbage; some aim to minimize the period of time taken in any one invocation of the collector (to provide predictable performance for realtime or interactive programming); some aim to minimize the space overhead (the memory required to identify and collect garbage); some are concerned with localization which is important for the efficient use of virtual memory. The next section gives a brief survey developing a taxonomy in terms of the advantages and disadvantages of different species.

2.1 Direct identification of garbage

Direct identification of garbage can be made using a *reference count*. In its simplest form, a cell holds a count of the number of references to it [Collins, 1960]. If as a result of a mutator operation the count falls to zero, the cell is garbage, since it can no longer be reached from a root. The collector can immediately reclaim

the cell and recursively decrement the counts of its referents and reclaim those whose count also fall to zero. Naturally enough, this process is known as *recursive freeing*.

A feature of reference counting is that garbage is reclaimed immediately it is identified. One of a number of disadvantages of reference counting is the space overhead of the count. It has been observed [Krasner, 1983] that the majority of cells have a small reference count. Consequently, the size of the count field of a cell is chosen to be smaller than is needed to represent all possible references. Typically, systems allocate one byte to hold the reference count. Once a count reaches the ceiling, *saturation*, it is not altered and no longer accurately reflects the number of references to a cell. To cheapen the test for saturation a count is saturated if the signed byte is negative, allowing the count to record from 0 to 127 references.

Clark's measurements of LISP programs (see [Deutsch and Bobrow, 1976; Field and Harrison, 1988]) show that about 97% of list cells have a reference count of 1. This suggests an extreme form of saturation using a *singlebit* count [Friedman and Wise, 1977]. A clear bit is used to indicate a single reference to cell. When a second reference to the cell is created the bit is set. Once set the bit cannot be cleared because it cannot be determined, without great cost, if the cell has more than one reference.

To reclaim cells that acquire more than one reference during their lifetime, it is necessary to employ a second type collector. Because of the predominance of single references, this collector will be invoked considerably less often than if it were used on its own. Singlebit reference counts are efficient and have the additional advantage in that they can be stored in cell pointers rather than the cell itself. Duplicating a pointer then does not require access to the cell to adjust the count.

2.2 Indirect identification of garbage

A second disadvantage of reference counting is the difficulty it has with reclaiming circular structures. The reason for this is the locality of indentification. It is expensive to determine if the destruction of one local pointer has disconnected a portion of the graph. A disconnected cyclic structure will have no vertices connecting it with the roots of the computation graph but each of its cells will have a nonzero reference count.

Some reference counting schemes do exist that attempt to reclaim cyclic garbage, but they are tedious, complex [Friedman and Wise, 1979], lack generality [Bobrow, 1980] and have significant computational overhead [Brownbridge, 1985; Hughes, 1985; Rudalics, 1986; Watson, 1986]. The problem can be overcome by requiring that the programmer explicitly break cycles of references or, more typically, by supplementing reference counting with second collector that identifies garbage indirectly [Goldberg and Robson, 1983].

Collectors that identify garbage indirectly take a global aspect. Traversing the computation graph from the roots and visiting all vertices will identify those cells which are definitively not garbage. By default, the unvisited part of the store is garbage or unallocated. By such means, cyclically connected subgraphs which become disconnected *are* (indirectly) identified and can be collected.

Mark-and-sweep collectors postpone collection until the free store is exhausted. Mutation is then temporarily suspended. Identification and reclamation are treated as sequential phases. The first phase traverses the computation graph *marking* all accessible cells. In its simplest form a single *markbit* is sufficient to indicate whether or not a cell is pointed to by other cells reachable from a root. This markbit is comparable with a singlebit reference count. A difference is that for the markbit, those cells whose counts are equal to zero are declared garbage while the others are part of the computation graph. The marking phase concludes when all accessible cells have been marked. A *sweep* of the entire store reclaims the unmarked cells and clears the marked ones [McCarthy, 1960]. Singlebit reference counting is further distinguished from mark-and-sweep by the periods in which the bits holds accurate information. For mark-and-sweep the

information is only consistent at the end of the sweep phase. For reference counting it is made consistent after every mutation.

The free storage can be managed as a freelist or a heap. With heap management reclamation can be achieved by compaction. For fixed size cells, compaction can be performed by sweeping the heap twice [Cohen, 1967]. In the first pass, two pointers are used, one starting at the bottom of the heap, the other at the top. The pointer to the top of the heap scans down until it points to a marked cell. The pointer to the bottom of the heap scans up until it points to an unmarked cell. At this point, the contents of the marked cell are copied to the unmarked cell (assuming the cells are the same size.), the markbit cleared and *forwarding pointer* to the new cell placed in the old position. When the two pointers meet, all marked cells have been unmarked and compacted in the upper part of the heap. The second scan is needed for readjusting pointers to moved cells. Any cells that refer to cells in the compacted area are adjusted by following forwarding pointers.

Martin [1982] combines the marking phase with a rearrangement of the pointers so that they can be moved more readily. Carsson, Mattsson and Bengtsson [1990] present a variation in which during the mark phase the pointer fields of the accessible cells (not the whole cells) are copied into a table and the cells are marked as visited. After sorting the addresses the reachable cells are compacted by sliding the cells to one end of the store.

If the store is managed as a freelist and the computation graph contains cells of differing sizes, allocation will in general *fragment* the free store. When an allocation request is made, the free list may contain no free cells of the required size, but may contain cells larger than that required. Typically, the allocator will satisfy the request by splitting a larger cell into an allocated cell, and a remaining free fragment. Over time, the freelist becomes composed of smaller and smaller fragments. Eventually a situation occurs where no free cell is large enough to meet the allocation yet the total size of free space is sufficient. The allocation can be met by coalescing the fragments into a single, or at least larger, cells. This is done by compacting the cells forming the computation graph. Some systems use compaction as an independent storage management technique to backup another garbage collection scheme. For example, in BrouHaHa Smalltalk [Miranda, 1987] the allocator checks that the total size in free cells is sufficient and if so invokes

the compactor. A mark-and-sweep garbage collector is used as a last resort if compaction would prove futile.

3.0 Incremental and Concurrent Collectors

Section 2 identified three processes associated with garbage collection: mutation (M), indentification (I) and reclamation (R). What distinguishes the majority of collectors up to [Cohen, 1981] is that these processes are sequenced. As reference counting reclaims garbage as soon as it is detected, mutation can be followed by cascades of IR operations as a result of recursive freeing. In contrast, indirect identification postpones collection until the free store is exhausted; only at the end of each MIR cycle is the store, generally, in a consistent state.

As Ungar [1984] reported, Fateman found that mark-and-sweep takes up 25% to 40% of the computation time of Franz-Lisp programs. Wadler [1976] reported that typical Lisp programs spend from 10% to 30% of their time performing collection. As such, mark-and-sweep is unsuitable for reactive (interactive and realtime) applications, because even if the garbage collector goes into action infrequently, on such occasions as it does it requires large amounts of time.

While reference counting is somewhat better in this respect because the grain size of the processes is smaller, a significant amount of time is spent in identification [Steel, 1975; Ungar, 1984]. Every mutator operation on a cell requires that the counts of its referents' be adjusted. Furthermore, significant time is spent in recursive freeing: 5% on Berkeley Smalltalk and 1.9% on Dorado Smalltalk implementations [Ungar, 1984]. Because recursive freeing is unbounded, the simple form of reference counting in which the collector immediately reclaims all the cells freed by a mutation is also unsuitable for reactive applications.

3.1. Deferred, direct collectors

The overhead of immediate reference counting can be reduced by deferring recursive freeing. Using doubly linked freelist store management [Weizenbaum, 1962; 1963], a newly deallocated cell can be placed on the end of the freelist but its referents not immediately processed. This cell is considered for reuse when it advances to the head of the list. Only at this time are the counts of its referents decremented; any falling to zero are added to the end of the freelist.

This *deferred* reference counting technique is time efficient and provides a smoother collection policy, one not so vulnerable to unbounded mutator delays of *immediate* reference counting. However, it is no longer true that after each MI operation all garbage has been identified let alone reclaimed. Collectors which, by design, do not necessarily identify and reclaim all garbage in a single invocation are said to be *incremental*.

A similar scheme is that of Glaser and Thomson [Field and Harrison, 1988], which uses a *to-be-decremented* stack instead of a doubly-linked list. In this scheme cells are added to the *to-be-decremented* stack if they have a count of one which requires decrementing. When cells are allocated from the stack their count is already one, hence this scheme manages to elide many garbage identification operations.

Deutsch and Bobrow [Deutsch, 1976] observe that, frequently, over a series of reference counting operations the net change in a cell's count will be small, if not nil. For example, when duplicating a cell reference as a stack parameter to a procedure call, the cell will acquire a reference that will be lost once the procedure returns. If adjusting such volatile references can be deferred, many garbage identification operations can be eliminated.

Baden [1983], proposes such a scheme for Smalltalk-80 which was used by Miranda [1987]. References to cells from roots, such as the stack, are not included in a cells count. Instead, root reference to cells are recorded in the *Zero Count Table* (ZCT). If a reference to a new cell is pushed on the stack (the typical way by which new cells join the computation graph), it is placed in the ZCT since it has a zero count and is only referenced from the stack. When a nonroot reference

counting operation causes a cells' count to fall to zero the cell is also placed in the ZCT because it might be referenced from a root. If the ZCT fills up or when no more free store is available, the collector initially attempts to reclaim cells in the ZCT. Firstly, reference counts are *stabilized* , made consistent, by increasing the count of all cells referred to from the roots. The ZCT is emptied by scanning and any referenced cell with a zero count is freed. Finally, the stack is scanned and the counts of all cells referred to from the stack are decremented. During this process any cells whose counts returns to zero are placed in the ZCT, since they are now only referenced from the stack.

Using this technique, stack pushes and pops reduce to ordinary data-movement operations, that is, they can be made without identification operations. Baden's measurements of a Smalltalk-80 system suggest that this method eliminates 90% of the reference count manipulations, and reduces the total time spent on reference counting by half [Baden, 1983]. A slight disadvantage is that sweeping the ZCT causes a pause in mutation, however typical pause times are of a few milliseconds [Miranda, 1987]. A further disadvantage is the extra storage required by the ZCT between reclamations.

3.2 Concurrent mark-and-sweep

The major advantage of deferred reference counting is that garbage collection is fine grained and interleaved with mutation, making it suitable for interactive and realtime applications [Goldberg, 1983]. The major disadvantage of indirect identification is the long interruptions of the mutator by the collector. Dijkstra [1978] described a modification of mark-and-sweep in which the mutator and the collector operate concurrently. Put another way, the collector operates *on-the-fly*. It was in the context of this algorithm that the terminology mutator and collector processes was coined.

In the simple mark-and-sweep scheme, of Section 2, concurrency is prevented by interference of identification by the mutator. If a reference to a new cell is added after the sweep has passed over it, the new cell will not be correctly identified as part of the computation graph. Dijkstra achieves a decoupling of the mutator

from the collector by introducing a third state for a cell. The three states, referred to as colours: *white* (unmarked); *black* (marked) and *gray*, can be represented by two mark bits. The mutator prevents collection of a newly allocated white cell by turning it grey at the time of allocation.

Marking blackens any cell traced from a root. Cells will be either black, grey or white. As previously, white cells are unreachable from the roots. Grey cells will be those allocated since the last collection but missed by during the marking phase. In the sweep phase white cells are reclaimed and other shades are whitened. Baker [1992] has recently proposed a realtime collector similar to Dijkstra's where any invocation of the collector is bounded in time.

3.3 Scavenging collectors

The generality and modularity of mark-and-sweep account for the attention it has received in the past three decades. It can however be inefficient because of its global nature. The marking phase inspects all accessible cells while the sweeping phase traverses the whole store. The sweep time is proportional to the size of the store and in virtual memory systems, the collector may access numerous pages on secondary store, an inherently slow process.

When the store is managed as a heap the costly sweep phase of the mark-and-sweep collectors can be eliminated by combining the identification and collection phases. This requires two heaps, historically called *semispaces* [Baker, 1978]. The mutator begins operating in the *fromspace*. When there is no free space, the collector scavenges fromspace. A *scavenge* is a simultaneous traversal and copy of the computation graph from the fromspace to the *tospace*. This combination of copying and tree traversal has the added advantage of improving locality. When each cell is moved to tospace a forwarding pointer is left behind. After a scavenge, the fromspace becomes free, and can be reused. The two semispaces are *flipped* and the mutator continues.

Baker's original scheme is also realtime. Collection is interleaved with mutation but any invocation of the collector is bounded. A consequence of this is that the

mutator must handle forwarding pointers. If the mutator encounters a reference to a forwarding pointer it updates the reference, so avoiding subsequent forwarding.

Scavenging schemes trade space for time since they require two heaps. Consequently they have much higher space overheads than either mark-and-sweep or reference counting algorithms.

3.4 Generational scavengers

Lieberman and Hewitt [1983] observed that most newly created cells die young, and that long-lived cells are typically very long-lived. Their collector segregates cells into generations, each with its own pair of semispaces. Each generation may be scavenged without disturbing older ones giving rise to incremental collection. Younger generations to be scavenged more frequently. The youngest generation will be filled most rapidly, but when flipping very few of its cells survive. This drastically reduces the amount of copying needed to maintain the generation. Generations can be created dynamically when the youngest generation fills up with cells that survive several flips.

Ungar's [1984] *generation scavenging* collector exploits the same cell lifetime behaviour as Lieberman and Hewitt. This collector classifies cells as either *new* or *old*. Old cells reside in a region of memory called *Old Space* (OS). All old cells that reference new ones are members of the *Remembered Set* (RS). Cells are added to RS as a side effect of the mutator. Cells that no longer refer to new cells are removed from RS when scavenging. All new cells must be reachable from cells in RS. Thus, RS behaves as roots for new cells and any traversal of new cells can start from RS.

Three heaps are used for new cells: *new* space (NS) (a large nursery heap where new cells are spawned); *past survivor* (PS) space (which holds new cells that have survived previous scavenges), and *future survivor* (FS) space (which remains empty while the mutator is in operation). A scavenge copies live new cells from NS and PS to FS space, and flips PS and FS. At the end of the scavenge, no live

cells are left in NS and it can be reused. Cells that have survived more than a prescribed number of flips are moved to OS, a process called *tenuring*.

With Ungar's collector the mutator is stopped during scavenging. This allows dispensing with forwarding pointers which achieves performance gains. While explicitly not concurrent, the collector is incremental because generations are small, pause times are short. By carefully tailoring the size of NS, FS and PS an implementation of Ungar's scheme for Smalltalk manages to keep scavenge times to a median of 150 milliseconds occurring every 16 seconds [Ungar, 1984].

Although generational collectors collect intragenerational cycles, they cannot collect intergenerational, cycles of references through more than one generation. Further, some schemes do not attempt to scavenge older generations. [Ungar 1984] leaves the reclamation of such garbage to *offline reorganization*, where a full garbage collection is done after the system has stopped. The current ParcPlace [1991] Smalltalk-80 generational garbage collector is backed up by an incremental collector, a mark-and-sweep collector, and a compactor which garbage collects OS.

Although generation collectors are one of the most promising collection techniques, they suffer poor performance if many cells live a fairly long time, the so-called *premature tenuring problem*. Ungar and Jackson propose an adaptive tenuring scheme based on extensive measurements of real Smalltalk runs [Ungar, 1988; 1992]. This scheme varies the tenuring threshold depending on dynamically measured cell lifetimes. It also proposes a refinement that has been included in the ParcPlace [1991] collector. In systems like Smalltalk, interactive response is at a premium but the system contains many large cells that don't contain references to other cells, mainly bitmaps and strings. To avoid copying these cells they are segregated in a *LargeCellSpace*, and tenured to OS when necessary.

A generational scavenging collector that adapts to the allocation patterns of applications was recently presented by Hudson and Diwan [1990]. This generational scavenging collector has a variable number of fixed size (power of 2) generations. The generations are placed in store at contiguous addresses. The generation number is apparent from the most significant address bits. Each generation has its own tospace, fromspace, and RS (remembered set). RS is fed indirectly via a buffer containing addresses of possible intergenerational pointers.

The feeder may filter out duplicates, intragenerational pointers, and nonpointers. When scavenging more cells than a generation can accommodate, a new generation is inserted. To retain the ordering, the younger generations are shuffled backwards during scavenging.

Other generation-based collectors include: *opportunistic* collectors [Wilson and Moher, 1989]; *ephemeral* collectors and the Tektronix Smalltalk collector. In terms of usage, all three commercial U.S. Smalltalk systems (DigiTalk, Tektronix and ParcPlace systems) have adopted generational automatic storage reclamation [Ungar and Jackson, 1988]. The SML NJ compiler [Wilson, 1992] also uses a generational collector. Deimer *et al* [1990] have investigated a generational scheme combined with a conservative mark-and-sweep garbage collector designed for use with Scheme, Mesa and C intermixed in one virtual memory.

Wilson, Lam and Moher [1990] show that, typically, generational garbage collectors have poor locality of reference, but careful attention to memory hierarchy issues greatly improves performance. They attributed the small success recorded by several researchers in their attempts to improve locality in heaps to two flaws in the traversal algorithms. They failed to group data structures in a manner reflecting their hierarchical organization, and more importantly, they ignored the disastrous grouping effects caused by reaching data structures from a linear traversal of hash tables (i.e. in pseudo-random order).

Incremental collectors that copy cells when the mutator addresses them have also been looked at by White [1980] and Kolodner [Kolodner *et al*, 1989; Kolodner, 1991]. These reorder cells in the order they are likely to be accessed in the future, giving improved locality. However, the technique requires special hardware. Other reordering optimizations that don't require special hardware work by reordering pages within larger units of disk transfer [Wilson, 1992].

4.0 Distributed Collectors

Following Hudak and Keller [1982] distributed collectors are characterized by:

i) a set of *nodes*; comprising any number of processors sharing a single

address space;
ii) connected by a *communication network;*
iii) where each node holds a portion of the computation graph and
iv) each node has at least one mutator.

In distributed systems, processing is distributed over all nodes. Each node has direct access only to cells that reside in its local heap. A reference to a cell in the same node is said to be *local.* A reference to a cell on another node is said to be *remote.* Access to a remote cell is achieved by sending a message to the node that holds it, which then performs any necessary operation.

The issues of distributed garbage collection are very much the issues of distribution:

i) concurrency, communication and synchronization;
ii) communication overheads;
iii) messages may be lost, delivered out of order or duplicated;
iv) fault tolerance.

After discussing the effects of distribution on the computation graph the following sections present various distributed collectors based on the previous taxonomy. The final section addresses fault tolerance issues. Table 1 summarizes the main characteristics of the collectors described.

4.1 Distributed computation graphs

To exploit the parallelism of a distributed system, the computation graph has to be distributed over all nodes. The vertices of the graph are naturally partitioned according to physical distribution, but there is no principle that prevents a cell migrating between nodes. Each node could contain roots of the graph but it is more usual that the roots lie on the node on which the computation was initiated. A remote reference is necessarily indirect. It first references a local *export record.* The export record references an *entry record* on a remote node. In turn, the entry record directly references the remote cell.

The import and export records might naturally be grouped in tables but the export record could equally well be a *proxy* cell. The triple indirection causes some overhead for a remote reference which adds another dimension to the problem of nonlocality. The entry table acts as additional local roots for the local partition of the computation graph. The local roots and the entry table will allow the local part of a graph to be collected independently. Given the potential parallelism, incremental and concurrent collectors appear the most appropriate for distributed systems. The problem of collection, then, naturally decomposes into the problem of local collection and global collection of the entry and exit tables.

Further tables may be used to record the cells they reference remotely. El-Habbash, Horn and Harris [1990] use an additional *private table.* The private table provides location independent addressing. Storage is partitioned into clusters, each with its own set of tables. A cluster is a logical partition of cells (a passive node) in contrast to the natural physical partition (of active nodes). A *cluster* is a group of cells which are expected to form a locality set. Cells in the cluster reference other clusters via defined *ports.* The import table gives a location hint about each external cell referenced from the cluster. The export table is the entry point for the public cells in the cluster which can be externally referenced. Public cells in the cluster are given unique *public identifiers* (PIDs). Private cells are not known outside the cluster and can only be referenced by the cells in the same cluster. The private cells are given *local identifiers* (LIDs), which are, in fact, private table entries in the cluster.

Clusters are the unit of management, the objective being to increase the locality of reference within a cluster. Removing nonreferenced cells from a cluster is considered a contribution to increasing the locality of reference of the cluster. Subgraphs which are only reachable from the export table may be removed to that cluster's *archival* cluster. Whenever an archived cell is referenced from any cluster, that cell and its subgraph are moved into the cluster. In this way, cells may migrate from cluster to cluster, via archival clusters. Archived cells which are not referenced from any cluster will remain in the archival cluster. Starting from the roots in the cluster, and traversing the subgraphs rooted at them, any cells connected in these graphs must remain in the cluster. The other cells which are not reachable from the roots are moved away to maintain a high locality of reference in the cluster. Nonreachable public cells in the cluster cannot be considered as garbage because they may be referenced from other clusters, but on

the other hand they are not part of the locality in the cluster. The private cells which are not reached from any public cells (roots or nonroots) in the cluster are definitely garbage, and can be reclaimed. Archival collection is controlled by setting time limits.

A similar approach is used by Moss [1990] in the Mneme project. Mneme structures the heap of cells into *files*. A file has a set of persistent roots and contains a collection of cells that can refer to each other using short cell identifiers. Cells in one file can refer to cells in other files via a device called a *forwarder*. A forwarder is a local standin or proxy for a cell in another file. Thus, to refer to a cell in another file, one refers to a local cell marked as a forwarder; the forwarder can contain arbitrary information about how to locate the cell at the other end. Each file can be garbage collected independently. Moss calls the import table the *incoming reference table* (IRT). Both the Moss and El-Habbash collectors are intended for use in a persistent environment.

4.2 Distributed direct identification of garbage

The locality of identification in reference counting has a number of attractive consequences for distributed systems. The collector visits cells only when the mutator does. Cells can be reclaimed locally as soon as they become inaccessible. One of the earliest distributed reference counting collectors performs all of the reference counting operations by spawning remote asynchronous tasks on appropriate processors [Hudak and Keller, 1982]. This ensures that actions are atomic. The nontrivial part of the adaptation is to guarantee that indentification operations (increment and decrement reference counts) are executed in the order they were generated. If this were not the case, a reference count may prematurely reach zero. Simple remote reference counting requires synchronization of communication between cooperating nodes.

Lermen and Maurer [1986] ignore part of the problem by assuming that the underlying communication protocol preserves the order of messages. The assumption can be enforced if either the system provides fixed routing or provides a message protocol that indicates the order in which they are sent.

An extension of reference counting which eliminates both synchronization and the need to preserve the order of messages is *weighted reference counting* (WRC). It was developed independently by Thomas [1981], Watson and Watson [1987] and Bevan [1987]. The idea is that each cell is allocated a standard reference count when created and at all subsequent times the sum of weights on the pointers to a cell is equal to the reference count. A reference with a weight W is equivalent to W references each with a weight 1. When a reference is duplicated it is unnecessary to access the cell. Rather, the weight of the pointer is equally divided between itself and the copy. The sum of the weights then remains unchanged. In this respect, WRC can be understood as a generalization of singlebit reference counting when the bit is located with the pointer. The advantage for distribution, is that no communication is required when a remote reference is copied. When a reference is destroyed, however, the pointer weight must be decremented from the reference count of the cell in order to preserve the rule that sum of the weights must equal the reference count. As usual, if a cell's count falls to zero it can be reclaimed.

Because the reference weight is always a power of two to allow for duplication, the log of the weight can be stored instead of the whole weight. This provides an important reduction in the space requirement for each reference. However, when a weight is to be subtracted from a count it must be converted (by shifting). Indirection is used to handle *underflow* which occurs when a reference weight of one needs to be copied.

An unfortunate consequence of indirection is that a reference, its indirection and the cell to which it refers may reside on different nodes. In this case, accessing a cell requires additional messages. *Generational reference counting* (GRC), Benjamin [1989] solves this problem. Each reference is associated with a *generation*. Each cell is initially given a zero generation reference, any copy of an *ith* generation reference is an *(i+1)th* generation reference. Each cell has a table, called a *ledger*, which keeps track of the number of outstanding references from each generation. If a cell's ledger has no outstanding references from any generation, then the cell is garbage and its space can be reclaimed. GRC has a significantly lower communication overhead but greater computational and space requirements than ordinary reference counting. Its communication overhead is similar to WRC, namely one acknowledged message for each copy of

an interprocessor reference and a corresponding extra space associated with each reference.

Vestal [1987] describes a collector that uses a distributed fault tolerant reference counter. Each cell maintains a conservative list of sites referencing it. Each site of this list keeps the count of references it has for that cell. Atomic update of the list is required when a site first references a cell. The cycle-detection algorithm is seeded with some cell suspected of being part of a dead cycle. The algorithm essentially consists of trial deletion of the seed and checking if this brings all the counts in the cycle to zero.

4.3 Distributed indirect collectors

One of the first distributed indirect identification collectors was the marking-tree collector, [Hudak and Keller, 1982]. It is an adaptation to a distributed environment of the previously described Dijkstra [1978] concurrent mark-and-sweep. Each mutator and collector on each node has its own task-queue. Each task locks all cells it intends to access to prevent race conditions. To prevent deadlock, if a task finds that some cell was already locked all locked cells are released and the task requeued. Since cells involved in a task may reside on different processors, this locking mechanism introduces high processing time and communication overhead when the collector and the mutator have high degrees of contention to shared cells. There is a single root of the whole distributed graph. The collector collects one node after another beginning with the root node. It can reclaim all garbage including cycles. The marking-tree collector operates in a functional graph reduction environment and need not handle arbitrary pointer manipulation. Because it does not batch remote mark tasks, it imposes high message traffic. Space needed for storing these requests cannot be determined in advance.

Similar mark-and-sweep collectors also inspired by Dijkstra's parallel collector were described by Augusteijn [1987] and Vestal [1987]. All processors cooperate in both phases of the collection but marking can proceed in parallel with mutation. In Vestal's [1987] collector, the cell space is split into logical *areas* in which parallel

collection may occur. Areas are a logical grouping of cells, and there is no control over site boundary crossing. The space overhead is proportional to the number of cells and to the number of areas, since each cell maintains an array of four colours for each existing area in the system. This collector does not take advantage of locality: each collector performs a global transitive closure starting at the root of one area, hence crossing boundaries.

Mohammed-Ali [1984], Hughes [1985] and Couvert [see Shapiro *et al*, 1990] describe variants of mark-and-sweep collectors applicable to the distributed environment. For these all nodes synchronise at the start of a local mark phase; At the end they perform a global rendezvous to exchange information about the global reachability. Each node then proceeds in parallel to a local sweep phase. A global rendezvous is inherently costly and nonscalable.

Mohammed-Ali [1984] presented two different approaches, 'global' and 'local' collectors with minimal space overheads. In the global approach, mutation is globally suspended for the entire collection. The collector handles arbitrary pointer manipulations and resolve some of the space and communication problems of the marking-tree collector.

Mohammed-Ali's [1984] 'local' collector simplifies collection by simply abandoning the attempt to recover cyclic garbage that spans several nodes. Each node asynchronously and independently performs local collection without involving any other node. If the freed storage is large enough the node's mutator will continue. Otherwise, it will invoke global collection. To allow a node to perform local garbage collection, it has to know which of its local cells are reachable from remote cells. Cells that have references from other nodes are assumed to be accessible in each local garbage collection. This situation persists until the next global collection invocation.

In the collectors given by Mohammed-Ali, the issue of lost or transit messages is solved by first assuming that the communication channel between each pair of nodes is order-preserving. An alternative solution is to keep message counts in each node. Before a garbage collection is completed, a check is made to ensure that the number of reply messages equals the message count. The space overhead of the collectors are not easily determined. In addition to *InTable* and *OutTable*

which keep track of incoming and outgoing references, there is *TempTable* that keeps in transit references and several message queues.

Hughes' collector [Hughes, 1985] is based on Mohammed-Ali's 'local' collector but reclaims cyclic garbage. Its main idea is to pipeline a number of collections over the entire network. This is achieved with the use of a synchronous termination detection algorithm based on instantaneous communication. Synchronous termination, however, may invalidate the collector for architectures comprising many nodes. On the other hand, the approach may be unsuitable when local heaps are large since the contribution of one node must always consist of a complete scan of its local heap. In a special operating mode the creation of a remote reference has to be accompanied by an access to the referenced node [Rudalics, 1986].

A modification of the generation scavenging used for Berkeley Smalltalk [Ungar, 1984] was given by Schelvis and Bledoeg [1988] for a distributed Smalltalk collector. In addition to OS, NS, PS and FS which hold cells according to their age, there is additional subspace, RS, that contains all replicated cells . RS is like OS, except that it contains the same cells in the same order on every node. Newly created cells are stored in NS. When NS becomes full, it and PS are garbage collected by scavenging. The roots of the computation graph are the set of new and survivor cells referenced from OS, RS or remote nodes. This root set is dynamically updated by checking on stores of pointers to NS. All cells in the graph are moved to NS, except for sufficiently old cells, which are moved to OS. At the end of a traversal NS is empty. Since most new cells soon die, PS fills up relatively slowly and, therefore, collection of the much bigger OS and RS is necessary less frequently.

Detection of dead cells in the distributed system is accomplished by a system wide mark-and-sweep collector. All nodes are checked if they have pointers to a particular cell. The graph of living cells is traversed, the cells accessed are marked, and at the end the space of unmarked cells is reclaimed or "swept". Although, the global mark-and-sweep collector handles both local and distributed cycles well, it does not work properly when not all nodes are able or willing to cooperate.

4.4 Hybrid collectors

When local collectors are independent they need not be homogeneous. One node may employ reference counting, another concurrent mark-and-sweep. Global and local collection may employ different collectors. Bennett [1987] describes a scheme which uses both a reference counting collector and a mark-and-sweep collector in his prototype distributed Smalltalk-80 system. A single table in each node, the *RemoteCellTable* (RCT) holds local cells that are remotely referenced. Bennett relies on facilities provided by the local Smalltalk memory manager to enumerate local cells *(proxy cells)* that indirectly reference remote cells. There are two distributed garbage collectors in Bennett's scheme, a fast algorithm that does not reclaim internode cycles, and a slower one that does. The algorithms are initiated by a user on one of the nodes.

The first reference counting collector relies on remotely referenced cells in alternating collection phases being distinguishable. Each cell has a flag in the RCT that identifies cells created since the start of a collection phase. These are similar to the grey cells of Dijkstra's [1978] collector. During each phase, each node enumerates its local proxies and sends a message for each proxy that increases the *external* reference count of the remote cell in its RCT entry. After this marking phase all remotely referenced cells have a nonzero external reference count. Each node then scans its RCT and removes those cells with a zero external reference count that were created before the start of the collection. Any such cells not referenced locally will be reclaimed by the node's local garbage collector.

This algorithm does not detect and reclaim internode cycles. The second, slower collector is a distributed mark-and-sweep algorithm that proceeds from those cells in the RCT that also have local references. These cells are followed for references to proxies and messages are sent to the remote nodes of these proxies to continue the scan remotely. (Bennett's system is implemented on PS Smalltalk which employs deferred reference counting. The internal reference count of a cell is therefore readily available.) At the end of this phase internode cycles will not have been marked and can be removed from the RCT.

DeTreville [1990] combines reference counting and mark-and-sweep in a concurrent collector for Modula-2+. The collector was used in a distributed

between pairs of sites; no global mechanism is necessary. The collectors' interface is designed for maximum independence from other components.

Shapiro et al detail various message protocols. Given a reference, the finder protocol locates the cell referred to. This protocol also handles cell deletion and node crashes. Other protocols include reference-sending, cell-migration, cycle detection and abnormal termination protocols. To deal with lost messages or those in transit, events are timestamped by a local, monotonically increasing clock. Each transmitted message is stamped with the value of the clock on transmission.

In Shapiro et al, [1990], the universe of cells is subdivided into disjoint spaces. Each space maintains the vector of highest timestamps received from other nodes. Each disjoint space maintains a list of potential incoming and outgoing references, called respectively the Cell Directory Table (CDT) and the External Reference Table (ERT). A CDT entry is stamped with the clock value of the last received message.

When a mutator exports a reference to another node, it is first added to the local CDT. Both the CDT and the ERT are overestimates. Local garbage collection proceeds from both local roots and the CDT and will remove garbage entries in the ERT. In turn, this allows previously referenced CDTs to be collected. The interface between the global collector and other components (i.e. the mutator and the cell finder) is limited to just the CDT and ERT. Updates to a CDT or ERT can occur in parallel with other activities. No synchronization is needed between the global service and the local collector or mutator. The main weakness of the collector is that it fails to detect interspace cycles of garbage. It proposes migrating locally unreachable cells, leaving cycle removal to a local garbage collector. Total ordering of spaces is used to avoid thrashing but this has its limitations.

Lang et al [1992] describe a fault-tolerant distributed collector that is largely independent of how nodes collect their local space and doesn't need centralized control nor global stop-the-world synchronization. It allows for multiple concurrent collections, doesn't require migration of cells (cf Shapiro *et al*) and yet reclaims all garbage cells including distributed cycles.

between pairs of sites; no global mechanism is necessary. The collectors' interface is designed for maximum independence from other components.

Shapiro et al detail various message protocols. Given a reference, the finder protocol locates the cell referred to. This protocol also handles cell deletion and node crashes. Other protocols include reference-sending, cell-migration, cycle detection and abnormal termination protocols. To deal with lost messages or those in transit, events are timestamped by a local, monotonically increasing clock. Each transmitted message is stamped with the value of the clock on transmission.

In Shapiro et al, [1990], the universe of cells is subdivided into disjoint spaces. Each space maintains the vector of highest timestamps received from other nodes. Each disjoint space maintains a list of potential incoming and outgoing references, called respectively the Cell Directory Table (CDT) and the External Reference Table (ERT). A CDT entry is stamped with the clock value of the last received message.

When a mutator exports a reference to another node, it is first added to the local CDT. Both the CDT and the ERT are overestimates. Local garbage collection proceeds from both local roots and the CDT and will remove garbage entries in the ERT. In turn, this allows previously referenced CDTs to be collected. The interface between the global collector and other components (i.e. the mutator and the cell finder) is limited to just the CDT and ERT. Updates to a CDT or ERT can occur in parallel with other activities. No synchronization is needed between the global service and the local collector or mutator. The main weakness of the collector is that it fails to detect interspace cycles of garbage. It proposes migrating locally unreachable cells, leaving cycle removal to a local garbage collector. Total ordering of spaces is used to avoid thrashing but this has its limitations.

Lang et al [1992] describe a fault-tolerant distributed collector that is largely independent of how nodes collect their local space and doesn't need centralized control nor global stop-the-world synchronization. It allows for multiple concurrent collections, doesn't require migration of cells (cf Shapiro *et al*) and yet reclaims all garbage cells including distributed cycles.

In Lang et al [1992] nodes are organized into 'groups'. A group is a set of nodes willing to cooperate together in a group collection. Nodes cooperate to collect garbage local to a group by means of a concurrent mark-and-sweep collector. Each group gives a unique identifier to each GC cycle. Multiple overlapping group collections can be simultaneously active. When a node fails to cooperate, the group it belong is reorganized to exclude it and collection continues.

The collector uses export and entry records as described in Section 4.1 but calls them exit and entry items respectively. Entry items have a reference count of exit items referencing them (up to messages in transit). Reclaiming an exit item requires a decrement message to be sent to the referenced entry item. If this action brings its counter to zero, the entry item is reclaimed. This mechanism for reclaiming entry items (the only one available) is safe since non cooperative nodes (or nodes that are down) do not send decrement messages and thus the cells they refer to cannot be reclaimed. Messages with acknowledgements and timeout are used to detect failed or non cooperating nodes.

The distributed collection begins with group negotiation. Nodes cooperatively determine group formation. All entry items of nodes within the group are marked w.r.t. the group. An entry item is marked *hard* if it is "needed outside the group" or it is "accessible from a root of a node in the group". It is mark *soft* if it is only referenced from inside the group. The initial marks of the entry items of a group are determined locally to the group by means of a reference counter. The reference counter allows the determination of the number of references that are outside a group. The marks of entry items are then propagated towards exit items through local collection. Similarly, the marks of exit items are propagated towards entry items they reference (if it is within the group) through group collection. This is repeated until marks of entry or exit items of the group no longer evolve. At this point the group is disbanded.

At the end of the marking, all entry items that are directly or indirectly accessible from a root or from a node outside the group are marked *hard*. Entry items marked soft can only be part of inaccessible cycles local to the group and can thus be safely reclaimed by the reference counting mechanisms. In the case of dead cycles, dead entry items in the cycle eventually receive decrement messages from all the dead exit items that reference them. Hence their reference counts decrease

to zero and they are eventually reclaimed by the usual reference counting mechanism.

Liskov and Ladin [1986], describe a fault tolerant distributed garbage detection based on their highly available centralized service. This service is logically centralized but physically replicated and so claims to achieve high availability and fault-tolerance. A client dialogues with a single replica; replicas stay up-to-date by exchanging background "gossip" messages. The failure assumptions are realistic: nodes may crash (in a fail-stop manner) and recover, messages may be lost or delivered out of order. All cells and tables are assumed backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These requirements are needed for the centralized service to build a consistent view of the distributed system.

Liskov and Ladin's [1986] distributed garbage collector relies on local mark-and-sweep, extended with the ability to identify the part of the graph between some incoming and outgoing reference. Each local collector informs the centralized service about the paths. The root used for tracing is the union of its local root with the set of local public cells. Local collectors query the centralized service about the real accessibility of their public cells to better estimate their root. Dead intersite cycles are detected by the centralized service. Based on the paths transmitted, the centralized service builds the graph of internode references and detects dead cycles with a standard collector.

The problem of collection for reliable distributed systems was also addressed by Detlefs [1990a; 1990b; 1991]. Transactions in reliable, distributed systems are serializable and recoverable. An atomic collector must also preserve the consistency of data after hardware (and software) crashes. Thus, each transaction by the collector must be logged. After a crash, recovery can be redone by replaying the log of transactions or, if nonvolatile storage (disk) survives the crash, recovery may use this as the starting point if more efficient. Other work concerned with making garbage collection cooperate transparently with a transaction protocol was done by Kolodner [1989,1991].

5.0 Summary

An attempt has been made to give some structure to a review of distributed garbage collection. A problem has been that any conceptual scheme has so many exceptions. Collectors were broadly classified as those that identify garbage directly and those that identify it indirectly. Emphasis was given to collectors that appeared since the last major review of garbage collection [Cohen, 1981].

Table 1 gives a summary of characteristics of the distributed collectors described in the review. The collectors were evaluated in terms of the issues noted in Section 4.0 The following abbreviations are used in the table:

Msg => Message
Ack => Acknowledgement
Cnt => Count
M => Marking
C => Copying
RC => Reference Counting
GS => Generation Scavenging
Comm => Communication
Synchro => Synchronization

Where qualification is required, as in pause, space and communication overhead, a rank of low, medium and high is used. These are relative terms and an order or further explanation is, where available, given in brackets.

A comprehensive bibliography on the subject follows. The number of references in the bibliography bear witness to the attention garbage collection is receiving, particularly distributed garbage collection. Despite this attention, a lot still remains to be done. About 80% of the distributed collectors reviewed in this paper have not been implemented.

Acknowledgements

We would like to thank Andrew Nimmo, Tim Kindberg and Xu Wang for reading the draft and offering useful suggestions.

DISTRIBUTED GARBAGE COLLECTION ALGORITHMS

Algorithm	Local GC Tech	Pause	Space Overhead	Comm. Overhead	Synchro. Overhead	Support for locality of reference	Detection of inter-space cycles	Detection of lost Message(s)	Order of message	Handling of Node crash/down	Comment
Hudak Keller	M	Low (Parallel)	Collector/ Mutator Queue	Medium	Locking	None	Yes (Syswide marking	Not Considered	Must be preserved	Not Considered	It is non compacting. Mutation and collection are done in parallel.
Moh'd-Ali	M/C	High/ Low	Tables and Msg queues.	High (Master)	System wide marking	None	Partially (Only the Global Scheme)	Time-Stamp or Order-preserving or Msg-Cnt	Must be preserved	Not Considered	Gives a general protocol for distributed GC. Present both global and local GC schemes.
Liskov-Ladin	M	High (Sys-wide marking)	Back up of cells and tables	High (Replicas & central sev.)	Clock	Yes (Replicas)	Yes (Centralize Service)	Partially (Reply Mechanism)	Not Important	Yes	Fault tolerant based on highly available centralized services.
Rudalics	M/C	High (Sys-wide Marking)	State, Msg count, ParentId	Medium (Batch msg)	None	None	Yes (Syswide marking)	Partially (Msg Ack Cnt)	Not Important	Not Considered	Inspired from Cheney and Baker Collectors. Always collect garbage at next cycle
Lermen-Maurer	RC	Low (Recursive freeing)	Ref cnt, ack & incompl. replies	O (3x ref copied)	Reference deletion only	Not Considered	No	Partially (Ack Cnt)	Must be preserved	Yes (Reply & Ack Cnt)	Gives protocol for a dist. ref. counting
Augustein	M	Medium (Parallel)	Colouring	High (Synchro-nizer)	Synchro-nizer	Not Considered	Yes (Syn-chronizer)	Reply Mechanism, Msg-Cnt	Must be preserved	Not Discussed	Inspired from Dijkstra Parallel Collector but concurrency is on the gc process not processor
Bennett	M/ RC	Low	Remote Cell Table	High (Table Update)	None	Yes (Replicas)	Yes (Syswide Marking)	Partially (Reply Mechanism)	Not Discussed	Yes (Replica-tion)	For Distributed Smalltalk. Collection is done by prevention.
Bevan	R C	Low (Recursive freeing)	Reference Cnt and log of weights	O(Inter-process reference)	None	Partially (Store Access)	No	Not Considered	Not Important	Not Considered	Design for Distributed Systems. Uses indirec-tion which may be on diff PE with ref cells

DISTRIBUTED GARBAGE COLLECTION ALGORITHMS

Algorithm	Local GC Tech	Pause	Space Overhead	Comm. Overhead	Synchro. Overhead	Support for locality of reference	Detection of inter-space cycles	Detection of lost Message(s)	Order of message	Handling of Node crash/down	Comment
Watson-Watson	R C	Low (Recursive freeing)	Reference Cnt and Log of weights	O(Inter-process reference)	None	Partially (Store Access)	No	Not Considered	Not Important	Not Considered	Designed for Parallel Systems. Uses indirection which may be on diff PE with ref cells.
Benjamin	RC	Low (Recursive freeing)	Generation Count and Ledger	O(Inter-process reference)	None	Partially (Store Access)	No	Not Considered	Not Important	Not Considered	Design for Distributed Systems. Indirections are guaranteed to be on same PE with their ref
Schelvis Bledoeg	GS	Medium	Entrance Table, Info/ref	High	None	Yes (Replica heap)	(Incrementally) with AccessPath	Yes(Reply, AccessPath)	Not Important	Yes (Reply AccessPath)	Designed for Distributed Smalltalk
Shapiro Plainfosse Gruber	M	Low (Parallel)	(CDT,ERT)	Low	None	Yes (Spaces)	Yes (Migration)	Yes (Timestamp of events)	Not Important	Yes (Finder protocol)	Suitable for low-level Distributed object support System.
Lang-Queinnec-Piquer	M	Medium (currency)	Entry and Exit items	Medium (Batch)	Very Low	Groups	Yes (Group-wide concurrent marking & RC)	Yes (Msg Ack, Time-out)	Not Important	Yes (Group Re-organization)	Suitable for very large or world-wide nets of possibly heterogeneous processors.
Moss	C	Low	Incoming Reference Table	Undetermine	None	Yes (Files)	Yes (Migration)	Not Considered	Not Important	No	Aimed at GC large persistent store in a distri. environment
Habbash-Haris-Horn	M	Low	Import, Export and Private Tables	Low	None	Yes (Clusters)	Yes (Archival Cluster Migration)	Not Considered	Not Important	Not Considered	Aimed at GC in Object-Oriented distributed persistent environment
DeTreville	RC/ M	Medium	Undetermine	High (RPC)	Partially -For REF Assign'nt	Yes (Spaces)	Yes (Sys-wide Marking)	Not Discussed	Not Discussed	Not Discussed	Aimed at GC in a shared memory multiprocessor

6 Bibliography

Almes G, Borning A and Messinger E (1983) Implementing a Smalltalk-80 system on the Intel 432: a feasibility study, in *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley 175-187

Appleby K, Carlsson M, Haridi S, and Sahlin D (1983) Garbage collection for Prolog based on WAM, Comm. ACM **31**, 719-741

Augusteijn L (1987) Garbage collection in a distributed environment, in *PARLE'87 - Parallel Architectures and Languages Europe,* LNCS **259**, Springer-Verlag, 75- 93.

Baden SB (1983) Low-overhead storage reclamation in the Smalltalk-80 virtual machine, in *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, 331-342

Baker HG (1978) List Processing in real-time on a serial computer, Comm ACM **21**, 280-294

Baker HG (1992) The treadmill: real-time garbage collection without motion sickness, SIGPLAN NOTICES 27(3), March 1992, 66-70

Bal H (1990) *Programming Distributed Systems,* Prentice Hall

Ballard S and Shirron S (1983) The design and implementation of VAX/Smalltalk-80, in *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, 127-150

Bartlett JF (1990) A generational, compacting garbage collector for C++, ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Bates RL, Dyer D and Koomen JAGM (1982) Implementation of Interlisp on the VAX, *ACM Symposium on Lisp and Functional Programming,* Pittsburgh, Pennsylvania, 15-18 August 1982, 81-87.

Ben-Ari M (1984) Algorithms for on-the-fly garbage collection, ACM Transactions on Programming Languages and Systems **6**, 333-44.

Bennett JK (1987) The design and implementation of distributed Smalltalk, OOPSLA '87, SIGPLAN Notices **22**(12), 318-330.

Bengtsson M and Magnusson B (1990) Real-time compacting garbage collection, ECOOP/OOPSLA '90 Workshop on Garbage Collection.

Benjamin G (1989) Generational reference counting: A reduced communication distributed storage reclamation scheme in *Programming Languages Design and Implementation,* SIGPLAN Notices **24**, ACM Press, 313-321.

Bevan DI (1987) Distributed garbage collection using reference counting, in *PARLE '87 - Parallel Architectures and Languages Europe*, LNCS **259**, Springer-Verlag 176-187.

Boehm HJ and Weiser M (1988) Garbage collection in an uncooperative environment, Software Practice and Experience **18**(9), 807-820.

Bobrow, DG (1980) Managing reentrant structures using reference counts, TOPLAS **2**(3) 269-273.

Brooks RA, Gabriel RP and Steele GL (1982) S-1 Common lisp implementation, ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania, 15-18 August 1982. 108-113.

Brownbridge DR (1985) Cyclic reference counting for combinator machines, in *Functional Programming Languages and Computer Architecture*, LNCS **201**, Springer-Verlag, 273-288.

Carlsson S, Mattsson C and Bengtsson M (1990) A fast expected-time compacting garbage collection algorithm., ECOOP/OOPSLA '90 Workshop on Garbage Collection.

Chambers C Ungar D and Lee E (1989) An efficient implementation of SELF, A dynamically-typed object-oriented language based on prototypes. OOPSLA '89, SIGPLAN Notices **24**(10), ACM, 49-70.

Cohen J and Trilling L (1967) Remarks on Garbage Collection using a two level storage *(sic)* BIT 7(1), 22-30

Cohen J (1981) Garbage collection of linked data structures, ACM Computing Surveys **13**(3) 341-367.

Collins GE (1960) A Method for overlapping and erasure of lists, Comm. of the ACM **3**(12) 655-657.

Courts R (1988) Improving locality of reference in a garbage-collecting memory management system, Comm. of the ACM **31**(9) 1128-1138.

Dawson JL (1982) Improved effectiveness from a real-time lisp garbage collector, *1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania August 15-18. 159-167.

Dellar CNR (1980) Removing backing store administration from the CAP operating system, Operating System Review **14**(4) 41-49.

Detlefs DL (1990a) Concurrent garbage collection for C++, CMU-CS-90-119 School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA 15213.

Detlefs DL (1990b) Concurrent, atomic garbage collection., ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Detlefs DL (1991) Concurrent, Atomic Garbage Collection, PhD Thesis,Dept of Computer Science, Carnegie Mellon Univ, Pittsburgh, PA 15213 CMU-CS-90-177, November 1991.

Demers A, Weiser M, Hayes B, Boehm H, Bobrow D and Shenker S (1990) Combining generational and conservative garbage collection: framework and implementations, in ACM Symposium on Principles of Programming Languages, 261 - 269.

DeTreville J (1990) Experience with garbage collection for Modula-2+ in the Topaz Environment, ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Deutsch LP and Bobrow DG (1976) An efficient, incremental, automatic garbage collector. Comm ACM 19(9) 522-526.

Deutsch LP (1983) The Dorado Smalltalk-80 Implementation: Hardware architecture's impact on software architecture, in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 113-125.

Dijkstra EW, Lamport L, Martin A J and Steffens EFM (1978) On-the-fly garbage collection: An exercise in cooperation, Comm ACM **21**(11) 966-975.

Edelson D and Pohl I (1990) The case for garbage collector in C++, ECOOP/OOPSLA '90 Workshop on Garbage Collection.

El-Habbash A, Horn C and Harris M (1990) Garbage collection in an object oriented, distributed, persistent environment., ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Falcone JR and Stinger JR (1983) The Smalltalk-80 Implementation at Hewlett-Packard, in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 79-112

Fenichel RR and Yochelson JC (1969) A LISP garbage-collector for virtual-memory computer systems, Comm ACM **12**, 611-612

Ferreira P (1990) Storage reclamation., ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Field AJ and Harrison PG (1988) *Functional Programming*, Addison-Wesley.

Fisher DA (1974) Bounded workspace garbage collection in an address-order preserving list processing environment, Info. Processing Letters **3**(1), 29-32.

Foderaro, JK, Fateman, RJ (1981) Characterization of VAX maxsyma in Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, 14-19.

Friedman DP and Wise DS (1976) Garbage collecting a heap which include a scatter table, Info. Processing Letters **5**(6), 161-164.

Friedman DP and Wise DS (1977) The one-bit reference count, BIT (**17**), 351-359.

Friedman DP and Wise DS (1979) Reference counting can manage the circular environments of mutual recursion, Info. Processing Letters 8(1), 41-45.

Gabriel RP and Mansinter L M (1982) Performance of lisp systems, *1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania, 15-18 August 1982, 123-142.

Garnett NH and Needham RM (1980) An Asynchronous garbage collector for the Cambridge file server, Operating System Review **14**(4), 36-40.

Gelernter H, Hansen JR and Gerberrich CL (1960) A FORTRAN-compiled list processing language, JACM 7(2), 87-101.

Goldberg A and Robson D (1983) *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 674-681

Hansen WJ (1969) Compact list representation: definition, garbage collection, and system implementation, Comm ACM **12**(9), 499

Hayes, B (1990) Open systems require conservative garbage collectors, ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Hayes B (1991) Using key object opportunism to collect old objects, OOPSLA'91, SIGPLAN Notices **26**(11), ACM, 33-46

Hoare CAR (1974) Optimization of store size for garbage collection, Info. Processing Letters 2(6), 165-166.

Hudak, P(1982) Object and Task Reclamation in Distributed Applicative Processing Systems, PhD Thesis, University of Utah.

Hudak, P and Keller R M (1982) Garbage collection and task deletion in distributed applicative processing systems, ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania, August 1982, 168-178.

Hudak P (1986) A semantic model of reference counting and its abstraction (detailed summary), Proceedings of 1986 ACM Conference on Lisp and Functional Programming, Massachusetts Institute of Technology, 351-363.

Hudson, R and Diwan A(1990) Adaptive garbage collection for Modula-3 and Smalltalk., ECOOP/OOPSLA '90 Workshop on Garbage Collection.

Hughes, J (1984) Reference counting with circular structures in virtual memory, spplicative dystems, TR Programming Research Group, Oxford Univ.

Hughes, J (1985) A distributed garbage collection algorithm, in *Functional Programming Languages and Computer Architecture*, LNCS **201**, Springer-Verlag, 256 - 272.

Johnson D (1991) The case for a real barrier, Proceedings of the Fourth International Support for Programming Languages and Operating Systems (ASPLOS IV), 96-107.

Jones SLP (1987) *The Implementation of Functional Programming Languages*, Prentice-Hall.

Jonkers HBM (1979) A gast garbage compaction algorithm. Info. Processing Letters 9(1) 26-30.

Juul NC (1990) Report on the ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems.

Kafara D, Washabaugh D and Nelson J (1990) Garbage collection of actors, ECOOP/OOPSLA '90 Proceedings of Workshop on Garbage Collection, 126-34

Kain RY (1969) Block structures, indirect addressing and garbage collection. Comm ACM 12(7) 395-398.

Knuth DE (1973) *The Art of Computer Programming; Vol 1: Fundamental Algorithms*, Addision-Wesley, Reading, Mass.

Kolodner E, Liskov B and Weihl W (1989) Atomic garbage collection: managing a stable heap, *Proceedings of 1989 ACM SIGMOD International Conference on the Management of Data*, 15-25.

Kolodner E (1991) Atomic incremental garbage collection and recovery for Large stable heap, implementing persistent object bases: principles and practice, *Fourth International Workshop on Persistent Object Systems*, Morgan-Kaufmann Publishers, San Mafeo, California.

Krasner G (ed) (1983) *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley.

Lang, B, Queinnec C, and Piquer J (1992) Garbage collecting the world, Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1992.

Lermen C W and Maurer D (1986) A Protocol for Distributed Reference Counting, *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, Massachusetts Institute of Technology, 343-350.

Li K (1988a) Real-time concurrent collection in user mode, ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Li K Appel AW, and Ellis JR (1988b) Real-time concurrent collection on stock multiprocessors, *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 11-20.

Lieberman H and Hewitt C (1983) A real-time garbage collector based on the lifetimes ot objects. Comm ACM 26(6), 419-429.

Lindstrom G (1974) Copying list structures using bounded workspace, Comm ACM **17**(4), 198-202.

Liskov B and Ladin R (1986) Highly-available distributed services and fault-tolerant distributed garbage collection, in *Proceedings of the 5th symposium on the Principles of Distributed Computing*, ACM, 29-39

Martin JJ (1982) An efficient garbage compaction algorithm, Comm ACM **25**(8), 571-581.

McCarthy J (1960) Recursive functions of symbolic expressions and their computation by machine: Part I, Comm ACM **3(4)**, 184-195.

McCullough PL (1983) Implementing the Smalltalk-80 system: The Tektronix experience, in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 59-78

Meyers R and Casseres D(1983) An MC68000-Based Smalltalk-80 System, in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 175-187.

Miranda E (1987) BrouHaHa - a portable Smalltalk interpreter, OOPSLA'87, SIGPLAN Notices **22**(12), ACM, 354-365

Mohammed-Ali K A (1984) Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems, Academic Dissertation, Royal Institute of Technology, Dept of Computer Systems, Stockholm, Sweden.

Moon D.(1984) Garbage collection in a large lisp system, 1984 ACM Symposium on Lisp and Functional Programming, 235-246.

Morris FL (1978) A time- and Space-efficient garbage compaction algorithm. Comm ACM **21**(8), 662-665.

Morris FL (1979) On a comparison of garbage collection techniques, Comm ACM **22**(10), 571.

Moss JEB (1990) Garbage collecting persistent object stores, ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Newell A and Tonge FM (1960) An introduction to IPL-V, Comm ACM 3, 205 - 211.

Newman IA, Stallard RP and Woodward MC (1982) Performance of parallel garbage collection algorithms, Computer Studies **166**, Sept 1982.

Nilsen K and Schmidt WJ (1990) Hardware support for garbage collection of linked objects and arrays in real time, ECOOP/OOPSLA'90 Workshop on Garbage Collection. October 1990.

Nilsen K (1988) Garbage collection of strings and linked data structures in real time, Software Practice and Experience **18**(7), July 1988, 613 - 640.

North SC and Reppy JH (1987) Concurrent garbage collection on stock hardware

in *Functional Programming Languages and Computer Architecture*, LNCS **274** Springer-Verlag, 1987, 113 - 133.

ParcPlace (1991) Objectworks\Smalltalk Release 4 User's Guide, Memory Management 229-237

Queinnec C, Beaudoing B, and Queille J (1989) Mark DURING sweep rather than mark THEN sweep, in PARLE '89 - *Parallel Architectures and Languages Europe*. LNCS **365**, Springer-Verlag.

Rudalics M, (1986) Distributed copying garbage collection, *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, Massachusetts Institute of Technology, 364-372.

Schelvis M and Bledoeg E (1988) The implementation of a distributed Smalltalk, ECOOP Proceedings, August 1988 LNCS 322.

Schelvis M (1989) Incremental distribution of timestamp packets: a new approach to distributed garbage collection, OOPSLA'89, SIGPLAN Notices **24**(10), ACM, 37-48.

Schorr H and Waite WM (1967) An efficient machine-independent procedure for garbage collection in various list structures, Comm ACM **10**(8), 501-506.

Sharma R and Soffa M L (1991) Parallel generational garbage collection, OOPSLA 91, SIGPLAN Notices **26**(11), ACM, 16-32

Shapiro M, Plainfosse D and Gruber O (1990) A garbage detection protocol for a realistic distributed object-support system., ECOOP/OOPSLA'90 Workshop on Garbage Collection. October 1990.

Standish TA (1980) *Data Structures Techniques*, Addison-Wesley, Reading, Mass., 1980.

Steele GL (1975) Multiprocessing compactifying garbage collection. Comm ACM **18**(9), 495-508.

Thomas RE, (1981) A dataflow computer with improved asymptotic performance, MIT Laboratory for computer science report MIT/LCS/TR-265

Terashima M and Goto E (1978) Genetic order and compactifying garbage collector, Info. Processing Letters 7(1), 27-32.

Ungar DM and Patterson DA (1983) Berkeley Smalltalk: who knows where the time goes?, in *Smalltalk-80: Bits of History, Words of advice*, Addison-Wesley 189-206.

Ungar D (1984) Generation scavenging: a non-disruptive high performance storage reclamation algorithm, in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, 157-167.

Ungar D and Jackson F (1988) Tenuring policies for generation-based storage reclamation, OOPSLA'88, SIGPLAN Notices **23**(11), ACM, 1-17.

Ungar D and Jackson F (1992) An adaptive tenuring policy for generation scavengers, TOPLAS **14**(1), 1-27.

Vestal S C (1987) Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming. PhD Thesis, Dept. of Computer Science, University of Washington, Seattle WA (USA), January 1987.

Wadler PL (1976) Analysis of an algorithm for real-time garbage collection, Comm ACM **19**(9), 491-500.

Watson I (1986) An analysis of garbage collection for distributed Systems, TR Dept Comp. Sc., U. Manchester.

Watson P and Watson I (1987) An efficient garbage collection scheme for parallel computer architecture, in *PARLE '87 - Parallel Architectures and Languages Europe*, LNCS **259**, 432 - 443.

Wegbreit B (1972) A generalised compacting garbage collector, Computer Journal **15**(3) 204-208.

Weizenbaum J (1962) Knotted list structures, Comm ACM **5**(3), 161 -165.

Weizenbaum J (1963) Symmetric list processor, Comm ACM **6**(9), 524 -544.

White JL (1980) Address/memory management for a gigantic Lisp environment, or GC considered harmful, Conference Record of the 1980 Lisp Conference, 119-127.

Wilson PR and Moher TG (1989) Design of the opportunistic garbage collector, OOPSLA'89, SIGPLAN Notices **24**(10), ACM, 23-35.

Wilson PR (1990) Some issues and strategies in heap management and memory Hierarchies, ECOOP/OOPSLA'90 Workshop on Garbage Collection

Wilson PR, Lam MS and Moher TG (1990) Caching considerations for generational garbage collection: a case for large and set-associative caches., TR UIC-EECS-90-5, December 1990.

Wilson P R (1992) Comp.compiler Usenet discussion, February 1992

Wilson PR, Lam MS and Moher TG (1991) Effective "static-graph" reorganization to improve locality in garbage-collected systems, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, 177-191.

Wolczko M and Williams I (1990) Garbage collection in high-performance system, ECOOP/OOPSLA'90 Workshop on Garbage Collection. October 1990.

Woodward MC (1981) Multiprocessor garbage collection - a new solution, Computer Studies **115**

Zave DA (1975) A fast compacting garbage collector, Info Processing Letters 3, 167-169.

Zorn B (1989) Comparative performance evaluation of garbage collection Algorithms, PhD Thesis, EECS Dept, UC Berkeley.

Zorn, B (1990) Designing systems for evaluation: A case study of garbage collection., ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Dynamic Memory Management for Sequential Logic Programming Languages

Y. Bekkers, O. Ridoux and L. Ungaro

IRISA/INRIA-Rennes
Campus Universitaire de Beaulieu
35042 Rennes CEDEX - FRANCE
Email {bekkers,ridoux,ungaro}@irisa.fr

Abstract. Logic programming languages are becoming more complex with the introduction of new features such as constraints or terms with an equality theory. With this increase in complexity, they require more and more sophisticated memory management. This survey gives an insight into the memory management problems in sequential logic programming language implementations; it also describes the presently known solutions. It is meant to be understood by non-specialists in logic programming with good knowledge of memory management in general. We first describe a "usefulness logic" for run-time objects. Usefulness logic defines non-garbage objects. Next, memory management systems are presented from the most trivial original run-time system, with no real concern for memory problems, to elaborated run-time systems with memory management closely observing the usefulness logic. Finally, the choice of a garbage collection technique is discussed in relation with logic programming specificities.

1 Introduction

1.1 The scope of this survey

Logic programming languages are increasing in complexity with the introduction of new features such as suspension mechanisms, constraints or higher-order terms. With this increase in complexity, memory management becomes a primary concern. The problem with logic programming run-time systems is to avoid memory leaks which may be introduced by the implementation of the non-determinism of the language. Wadler [47] defines a *memory leak*[1] as a *feature of a program that causes it to use more space than one would expect.* Accuracy of memory management techniques in logic programming languages implementations is crucial. Accuracy concerns the ability to decide which objects are useless, regardless of a particular program, *only taking into account the characteristics of the representation function* implemented by the run-time system. Of course, taking into account the caracteristics of programs to define the completeness of memory management makes full accuracy impossible to achieve.

The paper contains five sections.

[1] Wadler calls it *space leak*

- In the first section, we explain, in an informal manner, the *usefulness logic* of standard logic programming run-time systems. Usefulness logic defines which objects are currently useful.
- Next, we briefly describe the implementations of the first logic programming system, and show its inefficiency.
- Then, we describe how usefulness logic can be implemented. Mechanisms specific to logic programming, namely "early reset" and "variable shunting", are presented.
- Then, memory management of objects used in the implementation of extensions to logic programming is discussed.
- In the last section, we discuss lower-level aspects of garbage collection, for example the pros and cons of "copy" or "mark and compact" techniques, or the ability to introduce some kind of generation garbage collector.

1.2 Other related issues

In this presentation, some important issues are left aside.

For instance, the memory management of OR-parallel systems which makes garbage collection algorithms more complex. Some important contributions are by Ciepielewski and Haridi [21], and Warren [52]. More recent contributions are [11, 23, 25, 27, 53]. Another contribution concerning concurrent logic programming with flat guarded Horn clauses is found in [44].

Parallel or real-time implementation of garbage collectors for logic programming is not presented. A study on this topic can, for example, be found in [10]. Real-time behavior was also one of the incentives in [37] to introduce some kind of generational garbage collector.

Another issue left aside here is the management of the program, called the *clause-base* in logic programming. In most logic programming systems, the clause-base can be modified using built-in predicates such as **assert** and **retract**. One would like the retracting of a clause to provide the opportunity of recovering memory. However, dynamic representation usually contains pointers to clauses. Those pointers may become dangling if the memory occupied by clauses is recovered as soon as they are retracted. The problems are magnified if "structure-sharing", see §3.2, is used. These problems have been studied in [31]. A dialect of Prolog, λProlog [35], proposes a more disciplined way of modifying the program. In this case, the dynamic part of the program is submitted to the general memory management scheme implemented for dynamic objects [14].

Other areas of memory may grow when executing a logic program. They are the symbol and constant tables. In most implementations, these areas are not reclaimed. Athough this is an important matter, this problem is left aside in this survey.

Nothing is said either, in this survey, on compile-time garbage collection, a promising technique which is just beginning to be investigated for logic programming purposes.

2 Peculiarities of logic programming systems

2.1 Run-time system of a non-deterministic language

Logic programming systems perform a *search* through a *search-tree*. Each node of the search-tree is decorated with a *goal-statement* which is a list of goals, i.e. terms containing *logical variables*. The root is decorated with the initial goal-statement. Transitions between search-nodes produce bindings of logical variables and the bindings produced along the path leading from the root to a given search-node form a *binding state*.

```
read(f(a)).
read(f(b)).

trans(U,V) :-   analyze(U,W),
                gen(W,V).

analyze(f(T),T).

gen(Z,g(Z)).

use(R) :- ...

:- read(X), trans(X,Y), use(Y).
```

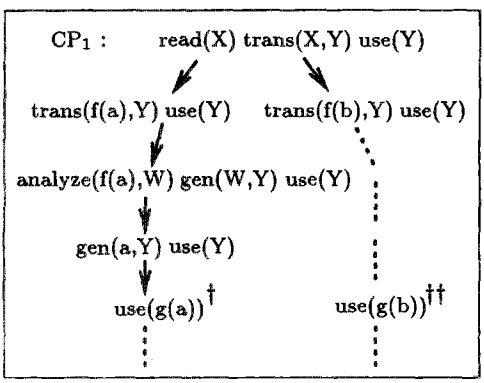


Fig. 1. A simple program and its search-tree

We will use the program in figure 1 throughout this survey. A search-tree for this program is partly shown on the figure.

Here is an explanation of the program for "non-prologers". The program represents a translation process. It reads, translates and finally uses terms, one at a time. The translation is done in two steps: an analysis step which extracts some of the content of the input term, and a generation step which integrates the extracted content within some new structure.

A characteristic of this program is that the predicate `read` has two clauses, hence, there are two possible input terms, `f(a)` and `f(b)`. They will be non-deterministically chosen by the run-time system and treated sequentially. Notice that these two terms stand for arbitrarily large data structures. The important thing is that, at some point of the treatment, some parts of these structures become useless for the rest of the computation.

2.2 Implementation of the search

A sequential Prolog system executes a depth-first traversal of the search-tree and the state of the system is a stack of goal-statements. This stack is called the *backtrack-*

stack and each of its elements is called a *choice-point.* A choice-point holds a goal-statement plus a reference to alternate clauses for that goal. For the sake of simplicity, we consider the "current goal-statement", CGS, as being at the top of that stack although it is usually held in registers.

In the example, one can see the two branches of the search-tree corresponding to the two input terms. Let us consider the search process as being at the node marked †. At this stage there are two active goal-statements:

- the current goal-statement `use(g(a))`,
- a single choice-point `read(X) trans(X,Y) use(Y)` called CP_1.

Usually, there is not much difference between two successive states in the stack, hence a new goal-statement is represented with some objects already used in the representation of previous (older) goal-statements. The modifications of structures, essentially variable substitutions, are recorded into a list called the *trail* which is used later to recover the previous states, i.e. *to backtrack.* In the example at position †, there are two active goal-statements, the current one, †, and CP1 to which the run-time system will backtrack to start the exploration of the second branch of the search-tree. At this stage of the computation the goal `use(Y)` is common to the two goal-statements and its representation can be shared by the representation of the the active goal-statements. Most implementations of logic programming languages follow this sharing method, which is called the *OR-sharing* technique. With such a technique, only the current choice-point can be readily accessed. In order to access saved choice-points, an interpretation of the trail is necessary.

2.3 Usefulness logic of logic programming run-time systems

We call *usefulness logic* of a programming language implementation the logic that determines which run-time objects are useful, without referring to a particular program. This notion is not usually exhibited, because it is trivial in the case of functional programming:

> *useful objects are those accessed when following references from some roots.*

But this notion is more complicated in logic programming implementations that call for the OR-sharing technique:

> *useful objects are those accessible from the choice-points, each one interpreted using its* own *binding state.*

The difficulty is that only the current binding state is readily represented. The usefulness logic of logic programming can be split into the three following principles.

1. The first principle, stated by Bruynooghe [17], is that all accesses to useful objects come from the active goal-statements. This suggests a marking procedure which executes a traversal of goal-statements found in the backtrack-stack. This principle implies that the well known "cut" operation of Prolog may recover some memory space, since it destroys choice-points. For non-prologers, the cut operation is an extra-logical predefined predicate that allows one to suppress choice-points. It is used for committing the search to some choices.

2. The second principle, pointed out by Bekkers et al. [9], is that some binding values may become useless in the course of the exploration of a branch in the search-tree. A given variable binding is relevant only for the goal-statements which are under the arc which produces the binding itself. If a variable is seen only from nodes higher in the search-tree than its binding, then this binding is useless.
   In the example, consider the search at point †, the substitution `[X <- f(a)]` is useless because, CGS has no access to `X`, and CP_1 accesses `X` but is above the arc which produces the binding. It is then possible to suppress the binding at that point. This technique is named *early reset* in [3]. The resetting of the variable and the discarding of its trail element are also described in [7]. Bruynooghe [17] describes a weaker version, called *virtual backtracking*, where variables are not reset to free, but where it is avoided to mark resetable binding values. The term early reset derives its name from the fact that variables, like `X` in the example, are reset to the free state before moving back up in the search-tree (i.e. before backtracking).
3. The third principle is that some variables may become irreversibly substituted [29]. We use the notation `<substitutions; list-of-goals>` to describe a goal-statement and exhibit the substitutions of variables.
   At position † in the search-tree, the goal-statements are:
   ```
   <[] ; read(X), trans(X,Y), use(Y)>
   <[Y <- g(a)] ; use(Y)>.
   ```
   Variable `Y` is free in a goal-statement and bound in another one. Hence, it is fair to represent the variable and its binding at the same time.
   But when the search is at node ††, there is only one goal-statement:
   ```
   <[Y <- g(b)] ; use(Y)>.
   ```
   In this case, variable `Y` is bound in every goal-statement. Hence, at position ††, it is useless to represent the variable `Y`. A more concise representation would be
   ```
   <[] ; use(g(b))>.
   ```
   Optimization consists in replacing an occurrence of a variable with its binding value. This has been named *variable shunting* in [29]. Some compile-time optimizations can be considered as a trivial form of variable shunting. In our example, the variable identifiers `U, V, T, Z, R` are treated by any decent compiler as naming devices and never lead to a variable creation at run-time. This is because there is no choice-point between the creation of the variable and its subtitution.

The three previous principles describe the requirement of an ideal memory management. It was gradually implemented in Prolog run-time systems. Several generations were necessary. Our presentation follows this historical and didactic progression.

3 Prehistory: lack of garbage collection

3.1 Reclaiming space upon backtracking only

The first Prolog interpreter (Marseille, [39]) was designed with no real concern for memory efficiency. Memory allocated during a procedure call is not reclaimed before

backtracking occurs. This is really not sufficient since it means, by analogy with functional languages, that a procedure body survives the procedure exit. However, reclaiming memory upon backtracking (*instant reclaiming* in short) is very important as it is a way of recovering an unbounded amount of memory at a constant cost. Hence, in general, systems should preserve such a capability.

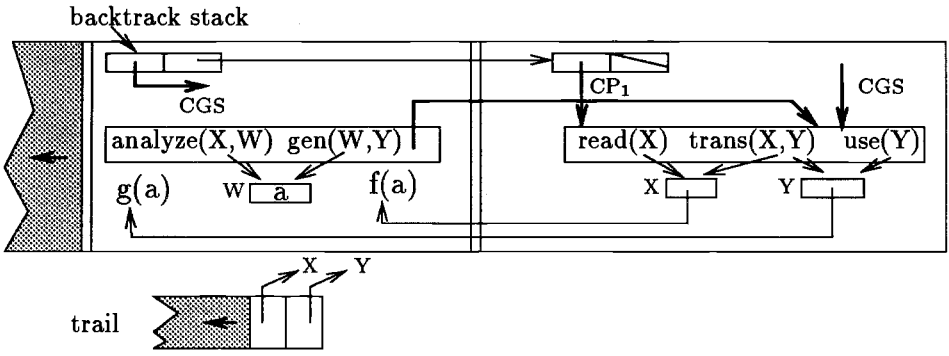


Fig. 2. Simple memory organization

Trivial memory management is illustrated on figure 2. Consider the computation as being at node † of the example. Goals `analyze(X,W)` `gen(W,Y)` are kept in memory and may eventually be buried under new objects, by the execution of the goal `use(W)`, although they are useless for the rest of the computation.

3.2 Structure-sharing versus term-copying

Mimicking the implementation of other languages, the first implementations of logic programming languages used classical Boyer and Moore's "structure-sharing" [13] to represent goals and terms. It is a well-known technique which represents a term with a pair of pointers, one to a static model in the program and the other to a dynamic "environment" which gives the values of variables. This is a "static" sharing, which uses pieces of program to represent dynamic objects. It must not be confused with the previously mentioned OR-sharing technique.

A new representation of terms, usually called *structure copying*, was proposed independently in [32] and in [16]. It uses copy and amounts to creating new data structures to represent a term instance. This method currently prevails over structure sharing because it simplifies garbage collection algorithms. The problem with structure-sharing comes from the indexed arrays used to represent the binding environments. Such arrays are difficult to compact. In most implementations, in classical WAM for example, only binding values are represented by copies. Lists of goals, located in the local stack, are usually represented with structure-sharing, but in [48] the choice is explicitly left open.

Few Prolog implementations use copies for the representation of lists of goals. Typical examples are Prolog implementations running on MALI [28, 14]. MALI is

a memory management machine which offers a kernel of commands well suited for logic programming: terms construction, terms traversal, variable binding, etc. It fully encompasses OR-sharing and does not preclude the use of the other sharings. Thanks to its Lisp-like term construction capability, goals copying is very natural in MALI.

It is hard to compare the memory efficiencies of the two schemes as one can construct examples of programs where structure sharing is better than structure copying, and it is also possible to construct programs where it is the contrary.

3.3 Splitting memory into local and global spaces

The Edinburgh implementation [49] is the first to benefit from memory management efforts. Memory space is divided into two spaces, the *local stack* and the *global stack*.

the local stack contains the "control part" of goal-statements, i.e. parts of goals that are statically known to become useless when deterministically returning from a procedure (execution of a goal). Hence, some memory can be reclaimed before backtracking, which is an improvement over previous systems.

the global stack contains the "data part" of goal-statements, i.e. the representation of binding values which are compound terms. In general, space in the global stack cannot be reclaimed before backtracking because objects there, like Lisp list constructions, can survive procedure exits.

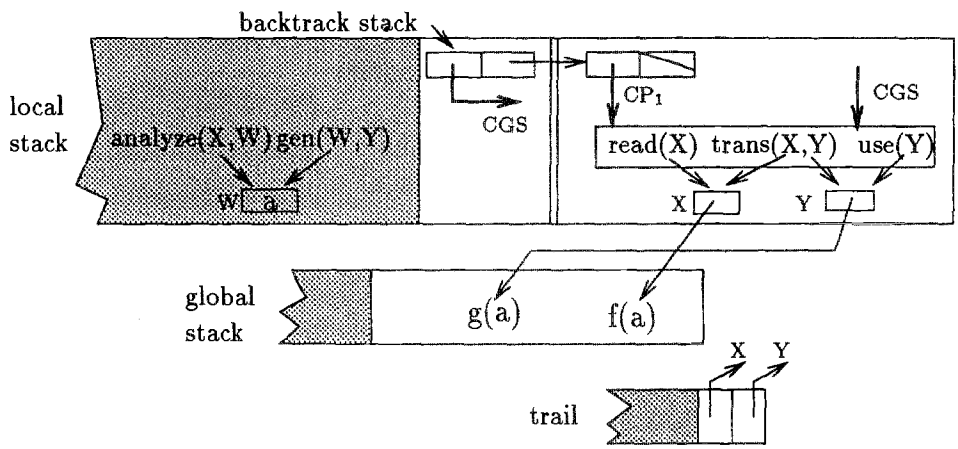


Fig. 3. Local and global spaces

A "deterministic return from a procedure" is such that no choice-point shares the representation of the procedure body. Indeed, the procedure body cannot be removed from the stack on exit if a choice-point maintains access to it. This is concretely detected by the presence of a choice-point above the procedure body in the local stack. This technique, sometimes called "environment protection" [2], is illustrated

on figure 3: consider the search at node †, goals such as `analyze(X,W) gen(W,Y)` have been removed, but not the goals `read(X) trans(X,Y)` because the choice-point CP_1 maintains access to them.

When a goal is "erased", the local stack abstractly decreases but actual decreasing depends on the concrete representation of goal-statements: in the case of a goal-copying technique, it really decreases as each goal is erased, but with a simple structure-sharing technique, see §3.2, it decreases at the procedure exit. An enhancement of the management of environments in structure sharing allows *environment trimming* [3]. It consists in ordering the variables in a clause so as to allow individual removal of each variable from the local, stack as soon as it disappears from goal statements.

With a local stack, classical tail recursion optimization (TRO) becomes possible [50] and allows the programming of infinite loops or "perpetual processes" which do not consume memory space [51].

The WAM [48, 2], which is usually considered as a standard for Prolog implementations, uses this two-stack organization.

4 Classical period: garbage collection

Despite the previous improvement, the global stack, also called the *heap*, needs to be garbage collected. It generally grows indefinitely during a perpetual process, even if the local stack does not. To complete its two-stack structure sharing scheme, Warren [49] proposes a mark and compact garbage collector for discarding inaccessible cells from either end of environments[2] in the global stack.

4.1 Blind traversal of goal-statements

Bruynooghe proposed an improved garbage collection scheme [17]. The proposal by Bruynooghe is to implement a traversal of the goal-statements found in the backtrack-stack, following all references without interpretation. However, such a traversal is blind in the sense that it follows references in a Lisp-like manner. This is conservative but not accurate because it does not respect the usefulness logic of logic programming systems (it satisfies principle 1, but neither 2 nor 3).

At step † of the example, all the objects in the heap, `g(a)` and `f(a)`, are marked and kept in memory (see figure 4).

4.2 Ignoring useless bindings, early reset

To comply with principle 2, "some binding values may become useless", it is necessary to consider the binding status of goal-statements. Several authors have implemented this principle [15, 8, 6, 37, 5, 3, 42]. The condition for applying early reset of variables is shown in figure 5. It shows that the variables which can be reset are those that are not seen by a goal statement older than the variable binding. The early reset technique consists in traversing each choice-point (goal-statement) in the backtrack-stack from top to bottom while marking traversed objects. Between traversals of

[2] they are called *frames* in his papers

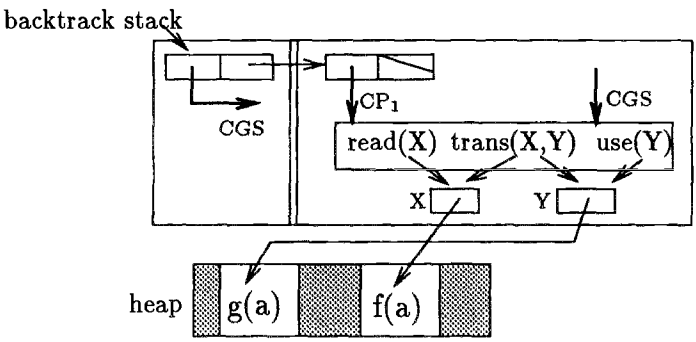


Fig. 4. Blind traversal of active goal-statements

consecutive choice-points, say CP_n and CP_{n-1}, the trail segment corresponding to CP_n is scanned in a search for trail elements corresponding to unmarked variables. The unmarked variables are unbound and their trail elements are discarded.

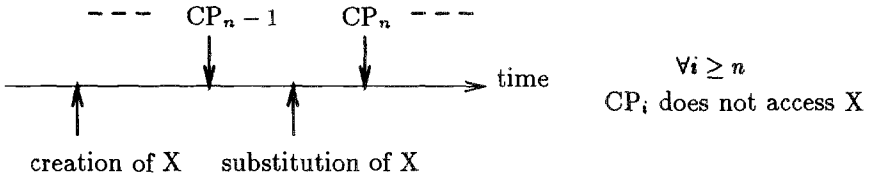


Fig. 5. Condition for applying early reset

Usually in the WAM, the trail is implemented as a stack, hence a compaction of the trail is required. With MALI, two other solutions have been implemented. The first solution is to represent the trail with a list and an element in this list is simply discarded by the garbage collector. The second solution is to put an extra field within the variables and to use this field as a link between substituted variables to represent the trail; in that case the trailing information is automatically discarded with the discarding of the variable. In usual WAM implementations, some variables are represented by a slot inside a data structure representing one of the terms in which the variable occurs. Hence, the last solution presented here is not realistic for such a representation.

The early reset algorithm is illustrated in figure 6: at step † in the example, variable `X` is only accessible from choice-point CP_1. It is free in this goal-statement, hence the space occupied by the binding value, `f(a)`, can be reclaimed. Note that term `f(a)` stands for an arbitrary large structure. As a result, the early reset gain of is unbounded.

This algorithm is correct because such a variable is not accessible from already visited goal-statements, since it is not marked, and is not seen as a bound variable

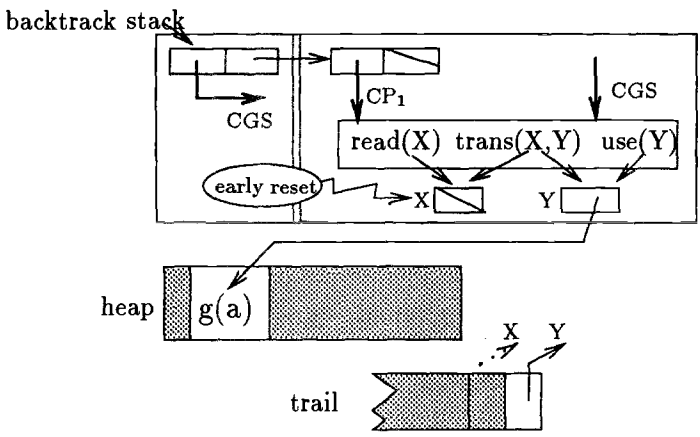


Fig. 6. Ignoring useless bindings: early reset

from the remaining goal-statements because they are older than the trail element (see figure 5). The binding value of such a variable is useless.

4.3 Shunting irreversibly substituted variables

Variable shunting consists in finding variables which are only seen in their bounded state, and then replacing pointers to such variables with their binding values. Such variables are those for which no choice points have been created (or they have been destroyed by cut operations) between creation time and binding time. There are several ways of implementing variable shunting.

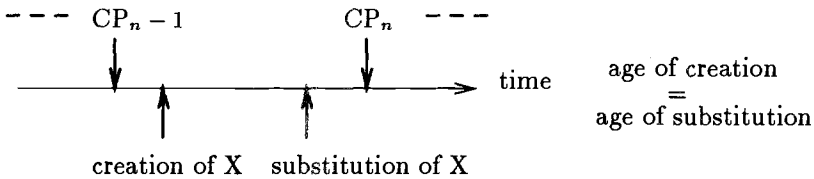


Fig. 7. Condition for applying variable shunting

First implementation, time-stamping: Let the *age* of a variable be the serial number of the choice-point just created after that variable. For systems in which the location of objects in memory respects the order of creation, the age of the variable can be directly deduced from its location; otherwise, it has to be explicitly memorized as a field in the variable.

In terms of age, variable shunting can be applied when the age of the variable is the same as the age of its substitution[3], as illustrated on figure 7 where both the creation of variable X and its substitution are time-stamped n. With a top-to-bottom traversal of goal-statements of the backtrack-stack, the algorithm works in two non-consecutive steps:

1. While scanning the trail segment corresponding to a choice-point CP_n, for any trail element concerning a variable of age n, the variable is made recognizable with a special tag called *shunted* and the trail element is taken out from the trail. Such a variable is not accessible in its free state and can be shunted.
2. When encountering a reference to a bound variable which has been tagged *shunted*, this reference is replaced by the binding value of the variable.

This algorithm which we call *time-stamping* has been implemented in [6, 10], two versions of MALI used in PrologII/MALI run-time system [28]. Variable shunting has also been implemented in SICStus Prolog [19], see [41] where benchmarks showing the usefulness of the approach are presented.

Second implementation, virtual saving: The general technique consists in traversing goal-statements in the backtrack-stack from bottom to top (the opposite of the previous traversal) and marking traversed objects. We call this technique *virtual saving* because it proceeds as if it were recreating the backtrack-stack. Virtual saving has been implemented in [38] a version of MALI used in λProlog run-time system [14].

To avoid traversing irrelevant binding values, the traversal (and marking) does not go through bound (non-shunted, see later) variables. As for the top-to-bottom traversal, a treatment is applied to the trail, between the traversals of two consecutive choice-points. The treatment used in the bottom-to-top traversal can be expressed as follows: let CP_{n-1} and CP_n be two consecutive choice-points, all choice-points up to $n-1$ have been marked; then two operations (sub-traitements) are performed onto the trail segment corresponding to CP_n.

In the first sub-treatement, the segment is scanned in a search for trail elements corresponding to unmarked variables; such variables are tagged "shunted" but not marked; their substitutions are untrailed.

Then, in the second sub-treatement, the binding value of every remaining trail element is traversed and marked. The binding values of shuntable variables do not need to be marked at that time. If necessary, they will be marked later while encountering shunted variables. While marking the binding values, some bound variables can, in turn, be met. Their bindings should not be followed. Either, it is a substitution relevant to that trail segment; hence, it will be marked by this second pass. Or, it is a substitution relevant to a trail segment not yet visited; therefore, to respect usefulness logic one must not follow the binding value.

So as to completely comply with principle 3 of usefulness logic, the two sub-treatements should be done sequentially, in the present order.

[3] The age of a substitution is the serial number of the choice-point just created after that substitution

As with the previous technique, every occurrence of a reference to a shunted variable is later replaced with the corresponding binding value.

This algorithm is correct because a trail element concerning an unmarked variable relates to a variable which was not seen as a free variable by previously traversed choice-points. In §6.1, it is shown that a top to bottom traversal followed by a bottom-to-top traversal of the backtrack-stack achieves early reset and variable shunting.

Not all substitutions are trailed: In most implementations, substitutions are recorded in the trail only if the variable is older than the last choice-point. If the substitution is not recorded in the trail, the variable is directly marked *shunted* when it is substituted.

About the interest of variable shunting: What variable shunting saves is only the representation of the variables. Hence, the gain is a constant amount if the representation of a variable has a constant size. However, as logical variables become more complex (types, constraints) their representation may become arbitrarily large (see §5.1) and the gain becomes more substantial.

Note that in most implementations of the WAM, a variable is usually represented by a slot in one of the structures in which it occurs. Therefore, in many cases the variable does not occupy any proper space, so that variable shunting may recover nothing. However, Sahlin showed in [40] that one can construct a program where variable shunting reclaims space in a WAM implementation. Another advantage with variable shunting is that it makes accesses to subsequent binding values more direct.

However, it is better not to be dependent on the "embodied-slot" trick because:

- some variables[4] need a proper representation,
- it does not work for complex variables,
- it complicates memory management.

For all these reasons some implementations may not use the trick.

5 Renaissance: implementing Prolog extensions

The development of extensions to Prolog started very early with PrologII [46] `dif/2` and `freeze/2` predefined predicates. The implementation of these two extensions has often been studied [12, 18]. Recent well-known extensions towards constraints programming can be found in PrologIII [22] and CHIP [24]. Another recent extension, λProlog, introduces λterms and their higher-order unification procedure. With these recent developments, new low-level mechanisms have been introduced to implement usefulness logic. Two requirements can be recognized:

1. suspending goals and awakening them on variable bindings;
2. rewriting terms.

[4] they are called *unsafe* variables in the WAM terminology

Two solutions have been proposed for implementing these mechanisms. The first solution consists in using a new binary structure which is an extension of the already known variable, the attributed variable, c.f. §5.1. The second solution consists in keeping the variable as is, while allowing multiple reversible substitutions thereof. The mechanism is usually implemented using a so-called *value trail*, c.f. §5.2.

5.1 Attributed variables

In order to implement extensions to Prolog, a new type of variable has been introduced; we call it *attributed variables*, as in [29]. It has also been called *closures* in [5] or *suspensions* in [18]. In these systems, attributed variables have been used to implement delay and constraint extensions into Prolog. An attributed variable is like a variable with an extra term attached to it, its *attribute*. Attributed variables have been generalized in [38] under the name of *mutable terms* (in short *muterms*). A Prolog level variant, called *meta-structure*, has also been proposed in [36] to provide a Prolog system with a user-extendible unification procedure.

From the point-of-view of memory management, the property of this binary structure is that the attribute is only accessible when the variable is free. This property must be used to improve the completeness of the garbage collector. The idea is to force the garbage collector to treat this structure as a normal variable. In this way, it straightforwardly reclaims the space occupied by the attribute when the attributed variable is shunted. Here the gain obtained with variable shunting becomes very important because attributes can be terms of any complexity.

Attributed variables, or their extensions, provide easily a reversible[5] term rewriting mechanism. For example, they have been used to implement λterms in λProlog [14][6]. With this method, histories of βreductions are automatically recorded in the trail through substitutions of attributed variables. For some reasons, essentially because of cut operations, the reversibility of rewriting may become useless. In that case, variable shunting automatically simplifies the representation of terms by discarding non-βreduced versions of terms.

With this method one can simulate multiple reversible assignments by creating a new attributed variable each time a new value has to be assigned.

5.2 Management of general affectation of variables

An alternative solution to attributed variables, is to allow direct multiple reversible affectations of the same variable. This solution has been implemented by Touraïvane [43] for PrologIII, a Prolog with constraints. It has also been used, under the name *value trail*, and *multi-value trail*, in CHIP [1], another constraint logic programming language, from ECRC.

Each time a variable is modified, its previous value is saved in the trail. Roots of accesses to useful objects are: *primary accesses* which come from the active goal-statements in the backtrack-stack and *secondary accesses* which come from values saved in some trail elements.

[5] undone upon backtracking

[6] In fact, for this application, muterms have used

As stated by the usefulness logic, some trail elements are useless (i.e. they must not be considered as secondary accesses). The following mechanisms are meant to implement a cleaning of the trail to fulfill principles 2 and 3 of the usefulness logic in the context of value trails.

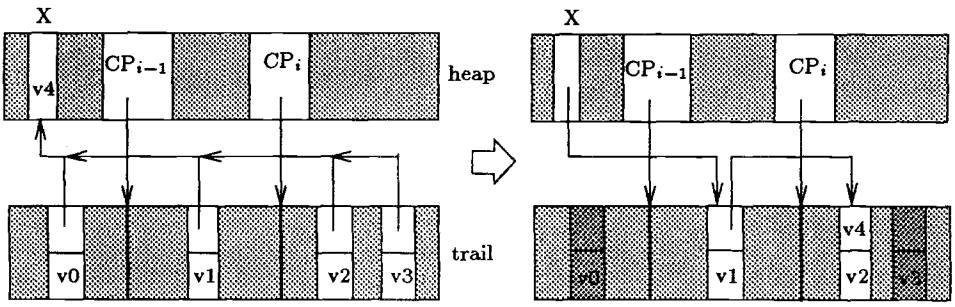


Fig. 8. Cleaning the trail

To specify which trail elements are useless, we use the already defined notion of age (§4.3). In the example of figure 8, the age of the variable `X` is $i-1$ and the ages of trail elements `v2` and `v3` are both $i+1$. There are two kinds of useless trail elements:

- those having the same age as the variable, for example `v0`; when backtracking, they restore a value to a variable which disappears immediately. This is a case of variable shunting.
- those for which there exists an older trail element with the same age for the variable, for example `v3`; they are useless because they restore a value which is immediately overwritten.

The overall algorithm consists in a "cleaning of the trail", to implement variable shunting, followed by a marking phase, to implement early reset, followed by a compaction of the stacks.

Cleaning the trail: The cleaning of the trail discards useless trail elements: this is done by scanning the trail from bottom to top and reversing pointers. Pointers are reversed to give access from the variables to the list of their values. See figure 8, at this stage there are only two trail elements in the trail.

The marking: Marking proceeds from primary accesses found in choice-points and secondary accesses found in the trail, in decreasing order of ages. As usual, to implement usefulness logic, each traversal is done according to an age; in the example

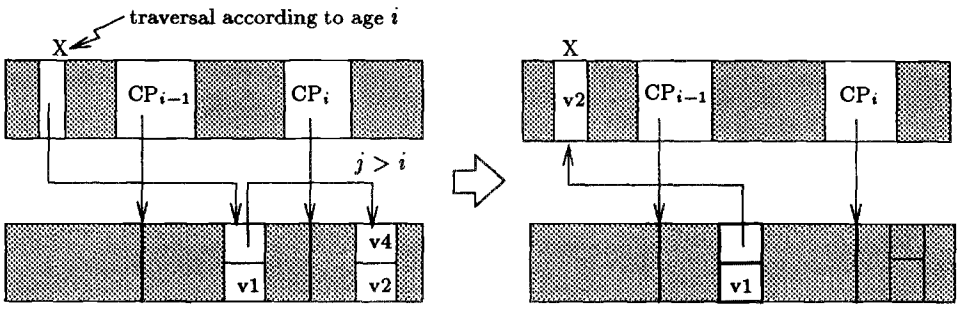


Fig. 9. Useless value of variable: early restoration

of figure 8, the value of `X` at age $i-1$ is `v1` and its value at age i is `v2`. When traversing a variable according to age i, let j be the age of the most recent trail element concerning this variable [7]:

- if $j \leq i$: the more recent value v of the variable is visible according to age i, so the value and all trail elements of the list are useful. Value `v` is immediately traversed according to age i and all trail elements of the list are marked.
- if $j > i$: the more recent value of the variable is a useless value (`v4` in figure 9). The actual value of the variable according to age i is the previous value `v2` saved in the trail. This value `v2` is immediately restored and traversed according to age i and all trail elements of the list, except for the last one, are marked. This operation, called "early restoration", corresponds to "early reset" in this implementation.

When traversing the trail section of age i, unmarked trail elements are discarded and values contained in marked trail elements are traversed under age $i-1$.

6 Different kinds of garbage collectors

Now that we have seen the requirements of memory management in the context of logic programming, we will discuss the implementation of garbage collectors in this context. At this level, classical traversal techniques which have been developed for functional programming can be borrowed:

- copying garbage collectors [26, 20], which are known to be simple and extensible towards real-time garbage collectors [4],
- mark and compact garbage collectors, usually a variant of the Morris algorithm [34]; regarding logic programming: the most interesting property of these garbage collectors is that they preserve the order of object locations. This makes

[7] remember that pointers in the trail have been reversed, so trail elements concerning a variable are easily found

the computation of the age of objects easy, and also allows instant reclaiming upon backtracking.

6.1 Copying versus mark and compact garbage-collectors

It is commendable to cause removal of elements from the search-stack to bring about an immediate recovery of memory. This is very cost-effective because an unbounded amount of memory can be recovered at a constant cost.

A frequent claim among implementors is that backtracking is incompatible with copying garbage-collectors. The reason that is given is that copying garbage-collectors usually move objects regardless of the structure of the backtrack-stack. Hence, instant reclaiming becomes impossible. This is why most existing implementations use mark and compact methods [5, 3, 42].

Assume, n is the number of non-garbage cells. The cost of a copy garbage collector is usually proportional to n. The cost of the compaction phase of mark and compact algorithm is generally proportional to the total amount of memory, i.e. the sum garbage and non-garbage. Sahlin proposes in [40] an improvement of the algorithm in [3] which makes it proportional to $nlogn$.

In the following, we describe how a copying technique, applied individually to each active goal-statement, can be combined to allow instant reclaiming.

Copying from the newest to the oldest choice-point: Copying from the newest to the oldest choice-point implements naturally early reset, §4.2. With the top-to-bottom copy garbage collector, variable shunting can be implemented at the extra expense of a new field within variables containing a time-stamp, see 4.3. However, the top-to-bottom copy places an object close to the most recent choice-point that uses it. This does not allow instant reclaiming because the same object may be used by older segments, see figure 10.

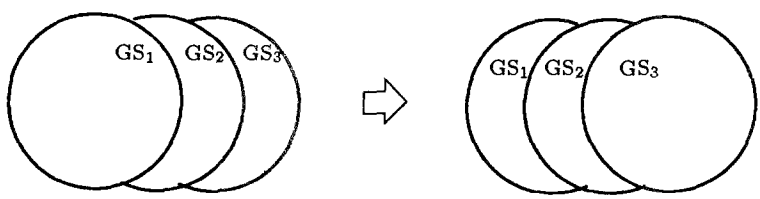


Fig. 10. Top to bottom copy collector modifies location order

Copying from the oldest to the newest choice-point: Applying a copying garbage-collector from the oldest to the newest choice-point preserves a sufficient amount of the creation order to implement instant reclaiming. This is the method

already mentioned as "virtual saving", see §4.3. Virtual saving is compatible with instant reclaiming because an object and the oldest choice-point that uses the object are placed in the same memory segment.

The drawback of the bottom-to-top copy method is that it does not help in implementing early reset of variables.

Combining bottom-to-top and top-to-bottom copies: The three desired behaviors can be obtained with a sequential combination of two copies:

- top-to-bottom copy to implement early-reset;
- then bottom-to-top copy to implement variable shunting and to allow instant reclaiming.

Both copies have a time complexity which is linear with respect to the number of useful cells. It is important to terminate the sequence with the bottom-to-top copy as it is the one which allows instant reclaiming.

A cheap but incomplete garbage collector is offered by the bottom-to-top copy alone. Both versions, the cheap one and the complete one, are offered in MALIv06 [38].

6.2 Segmented garbage collection for Prolog

Segmented garbage collection, a kind of generation garbage collection [45, 33, 30], adapted to Prolog, was first proposed by Pittomsvils, Bruynooghe and Willems [37]. These ideas have been developed and implemented by some authors [5, 3, 43]. The main idea for Prolog is well summarized by Appleby & all in [3]:

> When a choice-point is created, all structures in the heap that are not garbage will remain non-garbage until the choice-point is removed upon backtracking.

The main idea is to segment memory into two parts: an old segment and a new segment. The two segments are delimited by some choice-point CP_{GC}. Memory space is reclaimed in the new segment only. All objects in the old segment are useful. As usual, when references are created from the old segment to the new one, these references have to be considered as accesses. By chance, a Prolog run-time system records such reference creations in the trail, and the collector simply has to scan the trail to find them.

One drawback in relating the generational behavior of the collector to the search-stack is that the write boundary of the generational collector may be left (by the interpreting system) in such a state that the old segment is empty (or almost empty). For example, deterministic programs would never be in a position to benefit from the advantages of generation collection. A solution is to create artificial choice-points. This amounts to the use of the trail exclusively as a write boundary.

Early reset and segmented collectors - We can distinguish two strategies concerning the "early reset":

- the first strategy considers that all references from the old segment are useful [3]. This leads to an incomplete (conservative) garbage collection: the collector does not traverse structures in the old segment, but early reset is not applied on variables in the old segment.
- the second strategy traverses structures in the old segment in order to be able to correctly perform early reset. This seems to retain the advantages of generation garbage collection. However, in the case of a compacting collector, it saves the compacting of the old segment [43].

7 Conclusion

We have described the usefulness logic of run-time objects in logic programming language implementations. The usefulness logic of Prolog and its extensions highlights three points, which are:

1. only active choice-points are roots of useful terms;
2. some binding values may become useless, allowing early reset;
3. some variables may become irreversibly substituted, allowing variable shunting.

Complying with the third principle is particularily cost efficient for the implementation of extended Prolog systems. In λProlog, for example, it participates in cleaning useless versions of non-βreduced terms. In PrologIII and CHIP, it participates in cleaning the representation of constraints. In both cases, it automatically throws away versions that are useless for representing choice-points.

Modern logic programming systems implement the three points using two strategies related to the priority given to memory management and to the design of an abstract intermediate machine.

The first strategy is to enhance the implementation of an abstract machine already designed for some dialect of Prolog and to implement the usefulness logic for the data-structures of the preexisting machine. The main target of this strategy is the WAM.

The second strategy is to design a minimal package that implements the usefulness logic for some general data-structures, regarless of of any Prolog dialect. The package can then be used for designing an abstract intermediate machine. An example of this strategy is MALI.

The first strategy should be chosen for immediate efficiency of well-known Prolog dialects (say Standard Prolog). The second strategy is advantageously used for complex and/or experimental logic programming systems.

Usefulness logic is a difficult concept to formalize. It is basically an abstraction of an operational semantics for a programming language, which maps every object of implementation in the domain {useful,not-useful}. Moreover, it requires that the operational semantics should not betray the reality of memory usage: in Prolog, for instance, it must describe the OR-sharing mechanism.

This concept is seldom exhibited because it is trivial and implicit in memory management for functional languages, and it is a concept which has currently only few examples. We can connect the notion of usefulness logic to the solution that Wadler proposes to fix some space leaks problems [47]. He modifies a garbage collector so that it replaces every occurrence of `(car (cons x y))` (resp. `(cdr (cons x y))`) by `x` (resp. `y`). This means that the garbage collector is given a more intimate knowledge of the semantics of the language[8] than the ordinary box model. So, what Wadler proposes is in fact a new usefulness logic for non-strict functional programming. It is important to restrict the scope of the new usefulness logic to non-strict semantics because otherwise the equalities `(car (cons x y)) = x` and `(cdr (cons x y)) = y` are false (e.g. `(car (cons x BOTTOM)) = BOTTOM` in a strict semantics). This illustrates the connection between a usefulness logic and a semantics. The usefulness logic should be faithful to the semantics, but the semantics must be operational enough to express that `x` has always a smaller representation than `(car (cons x y))`.

8 Acknowledgements

This survey paper has greatly benefited from comments by Maurice Bruynooghe and Dan Sahlin.

References

1. A. Aggoun and N. Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. In *Séminaire de Programmation Logique de Trégastel*, pages 487–509, CNET, France, 1990.
2. H. Aït-Kaci. *The WAM: A (Real) Tutorial.* Technical Report 5, DEC Paris Research Laboratory, 1990. Revised in *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, 1991.
3. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on the WAM. *CACM*, 31(6), 1988.
4. H.G. Baker. List-processing in real-time on a serial computer. *CACM*, 21(4):280–294, 1978.
5. J. Barklund. *A Garbage Collection Algorithm for Tricia.* Technical Report 37B, UPMAIL, Uppsala University, 1987.
6. Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symp. Logic Programming*, IEEE, 1986.
7. Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog. In *Informatique–85, Symposium Soviéto–Français*, pages 111–117, Tallin, 1985.
8. Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog interpreters. In S-Å. Tärnlund, editor, *2nd Int. Conf. Logic Programming*, pages 343–351, Uppsala University, 1984.
9. Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A short note on garbage collection in Prolog interpreters. *Logic Programming Newsletter*, (5), 1983.

[8] Note that ordinary functional garbage collectors do not have this knowledge.

10. Y. Bekkers and L. Ungaro. Implementing parallel garbage collector for Prolog. In A. Voronkov, editor, *Russian Conf. Logic Programming*, Leningrad, 1991. LNCS 592.
11. D.J. Bevan. Distributed garbage collection using reference counting. In *PARLE*, pages 176–187, 1987.
12. P. Boizumault. A general model to implement *dif* and *freeze*. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming*, London, 1986. LNCS 225.
13. R.S. Boyer and J.S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, pages 101–116, Edinburgh University Press, 1972.
14. P. Brisset. *Compilation de λProlog.* Thèse, Université de Rennes I, 1992.
15. M. Bruynooghe. Garbage collection in Prolog implementations. In J.A. Campbell, editor, *Implementations of Prolog*, pages 259–267, Ellis Horwood, 1984.
16. M. Bruynooghe. The memory management of Prolog implementations. In *Logic Programming Workshop*, Debrecen, Hungary, 1980. Revised in S-Å. Tärnlund and K.L. Clark (editors), *Logic Programming*, pages 83–98, Academic Press, 1982.
17. M. Bruynooghe. A note on garbage collection in Prolog interpreters. In *1st Int. Conf. Logic Programming*, 1982.
18. M. Carlsson. Freeze, indexing and other implementation issues in the WAM. In J.L. Lassez, editor, *4th Int. Conf. Logic Programming*, pages 40–58, MIT Press, Melbourne, 1987.
19. M. Carlsson and J. Widèn. *SICStus Prolog User's Manual.* Research Report SICS/R88007C, SICS, 1988.
20. C.J. Cheney. A nonrecursive list compacting algorithm. *CACM*, 13(11):677–678, 1970.
21. A. Ciepielewski and S. Haridi. *Storage Models for Or-Parallel Execution of Logic Programs.* Technical Report Report TRITA-CS-8301, Royal Institute of Technology, Stockholm, 1983.
22. A. Colmerauer. Opening the Prolog III universe. *Byte Magazine*, 12(9), 1987. Special Issue on Logic Programming.
23. J. Crammond. A garbage collection algorithm for shared memory parallel processors. *Int. J. on Parallel Processing*, 17(6):497–522, 1988.
24. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Int. Conf. on Fifth Generation Computer Systems*, pages 693–702, Tokyo, 1988.
25. M. Dorochevsky, K. Schuerman, A. Véron, and J. Xu. Constraint handling, garbage collection and execution model issues in ElipSys. In A. Beaumont and G. Gupta, editors, *ICLP'91 Workshop on Parallel Execution of Logic Programs*, pages 17–28, 1991.
26. R.R. Fenichel and J.C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *CACM*, 12(11):611–612, 1969.
27. A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy reference counting: an incremental garbage collection method for parallel inference machines. In *5th Int. Conf. and Symp. on Logic Programming*, pages 1241–1256, 1988.
28. S. Le Huitouze. *Mise en œuvre de PrologII/MALI.* Thèse, Université de Rennes I, 1988.
29. S. Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Int. Work. Programming Languages Implementation and Logic Programming*, 1990. LNCS 456.
30. H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–429, 1983.
31. T.G. Lindholm and R.A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *4th Int. Conf. Logic Programming*, MIT Press, Melbourne, Australia, 1987.

32. C.S. Mellish. An alternative to structure-sharing in the implementation of a Prolog interpreter. In *Work. Logic Programming*, Debrecen, Hungary, 1980.
33. D. Moon. Garbage collection in a large Lisp system. In *ACM Conf. Lisp and Functional Programming*, 1990.
34. F.L. Morris. A time and space efficient garbage compaction algorithm. *CACM*, 21(8):662–665, 1978.
35. G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Symp. Logic Programming*, pages 810–827, 1988.
36. U. Neumerkel. Extensible unification by metastructures. In *Meta-Programming in Logic Programming*, pages 352–363, Leuven, Belgium, 1990.
37. E. Pittomvils, M. Bruynooghe, and Y.D. Willems. Towards a real-time garbage collector for Prolog. In *2nd Symp. Logic Programming*, IEEE, 1985.
38. O. Ridoux. *MALIv06: Tutorial and Reference Manual.* Publication Interne 611, IRISA, 1991.
39. P. Roussel. *Prolog : manuel de référence et d'utilisation.* Technical Report, G.I.A. Université Aix-Marseille, 1975.
40. D. Sahlin. *Making Garbage Collection Independent of the Amount of Garbage.* Research Report SICS/R–87/87008, SICS, 1987.
41. D. Sahlin and M. Carlsson. *Variable Shunting for the WAM.* Research Report SICS/R–91/9107, SICS, 1991.
42. J. Schimpf. Garbage collection for Prolog based on twin cells. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*, MIT Press, 1990.
43. Touraïvane. *La récupération de mémoire dans les machines non déterministes.* Thèse, Université d'Aix-Marseille, 1988.
44. K. Ueda and M. Morita. A new implementation technique for flat GHC. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 3–17, MIT Press, Jerusalem, 1990.
45. D. Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices, (ACM)*, 19(5):157–167, 1984.
46. M. Van Caneghem. *L'anatomie de PrologII.* Thèse de doctorat d'état, Université d'Aix-Marseille, 1984. Also in *L'anatomie de PrologII*, InterÉdition, Paris, 1986.
47. P.L. Wadler. Fixing some space leaks with a garbage collector. *Software — Practice and Experience*, 17(9):595–608, 1987.
48. D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.
49. D.H.D. Warren. *Implementing Prolog — Compiling Logic Programs, Vol. 1 and 2.* D.A.I. Research Report 39, 40, University of Edinburgh, 1977.
50. D.H.D. Warren. An improved Prolog implementation which optimises tail-recursion. In *Work. Logic Programming*, Debrecen, Hungary, 1980.
51. D.H.D. Warren. Perpetual processes — an unexploited Prolog technique. *Logic Programming Newsletter*, (3), 1982.
52. D.S. Warren. Efficient Prolog memory management for flexible control strategies. In *1984 Int. Symp. Logic Programming*, IEEE Computer Society Press, Atlantic City, NJ, 1984.
53. P. Weemeeuw and B. Demoen. A la recherche de la mémoire perdue, or: memory compaction for shared memory multiprocessors—design and specification. In S. Debray and M. Hermenegildo, editors, *2nd North American Conf. on Logic Programming*, pages 306–320, 1990.

This article was processed using the LaTeX macro package with LLNCS style

Comprehensive and Robust Garbage Collection in a Distributed System

Niels Christian Juul & Eric Jul

DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK 2100 Copenhagen Ø, Denmark
Phone: +45 35 32 18 18 Fax: +45 35 32 14 01 E-mail: {ncjuul|eric}@diku.dk

Abstract. The overall goal of the Emerald garbage collection scheme is to provide an efficient "on-the-fly" garbage collection in a distributed object-based system that collects *all* garbage. and that is robust to partial failures.
The first goal is to collect *all* garbage in the entire distributed system; we say that the collection is *comprehensive* in contrast to conservative collectors that only collect most garbage. Comprehensiveness is achieved by employing a system-wide mark-and-sweep collection based on concurrently running collectors, one on each node.
The second goal of our collector is to be robust to partial failures. When facing node failures the collector will progress in the available parts of the system and, when necessary, wait for temporarily unavailable nodes to become available again.
The scheme is being implemented on a network of VAXstations at DIKU. The full scheme employs two concurrent mark-and-sweep collectors on each node in the distributed system, one for comprehensiveness, one for expediency. Concurrency is achieved by using an object protection and faulting mechanism.

Keywords: Garbage collection [*mark-and-sweep, faulting, comprehensive*], Distributed systems [*distributed control, termination detection, fault-tolerance*], Concurrency, Object-oriented systems, Robustness, Emerald, Algorithm.

1 Introduction

The first goal of our distributed garbage collection scheme is to collect *all* garbage in a entire distributed system. We have introduce the term *comprehensive collection* to denote such schemes. In contrast, partial or conservative collection is a priori non-comprehensive. In general, the garbage collection problem can be formulated as a graph problem, where the vertices in the graph are objects and each directed arc represents a reference from one object to another. In contrast to a more conservative collection, a comprehensive collection essentially needs to perform a system-wide traversal of the graph in order to identify which objects are still in use and which are garbage. We attain a comprehensive collection by combining a basic mark-and-sweep collection scheme with mechanisms for concurrency and distribution.

Unfortunately, any comprehensive collector in a distributed system will, due to large network overheads, have problems collecting garbage fast enough to keep up with new object allocations. Thus, it is necessary to supplement our collector with an expedient, but conservative collector. In this paper, we concentrate on the issues relating to comprehensive collection.

The second primary goal is *robustness* to partial failures. In large distributed systems, the probability of failures becomes significant. Thus, we cannot expect the entire system to be available concurrently long enough to complete a comprehensive collection. This has lead us to investigate robust garbage collectors. A robust garbage collection scheme must cope with both short and long term unavailable parts of the system—partly by adapting its behavior to the situation, and partly by compromising on its goals.

While still being comprehensive, the collector must be able to survive during temporary unavailability. The collector must progress in the available parts of the system and wait for the needed, but unavailable, parts to become available again.

When more permanently unavailable parts block the comprehensive collection, both robustness and expediency demand that garbage is collected in the available parts. This may be achieved by another collector that collects garbage in the available parts of the system only. Such a collection cannot be comprehensive as long as the "liveness" of references from unavailable parts is unknown. Based on a conservative estimate of root objects, taking objects potential reachable from unavailable parts into account, this supplementary collector may collect garbage in the available part of the system. By careful selection of the additional root objects, this collection may be nearly comprehensive in the available parts of the system.

The full garbage collection scheme, which is based on at least two collectors on each node, must also reduce the latency introduced into applications. Thus, each collector works concurrently with other processes. The necessary synchronization constraints introduced by this concurrency are achieved by protecting objects, that have not been traversed by the collector, from being mutated by other processes.

Our approach has been to implement such a collection scheme for the Emerald Language [Hutchinson 87b, Raj 91]. Emerald is a distributed, object-based system [Black 86, Black 87, Jul 88b], based on a compiler generating native machine code [Hutchinson 87a] and a run-time system [Jul 88a], designed to take advantage of run-time garbage collection.

The objects handled by the run-time system in Emerald may be *migrated* between the nodes of the distributed system. Immutable objects may be *replicated* instead of moved; the replicas will always have consistent states because their state does not change. To survive node crashes, any object may *checkpoint* its current state to other nodes and/or stable storage. A checkpoint is a passive copy, from which an object may be recovered, if the real object has been lost during a node failure. Emerald is based on reliable internode communication, thus only nodes may fail. In our model, nodes are autonomous and have *failed-stop* semantics.

In summary, the Emerald garbage collection scheme does a comprehensive and concurrent collection of all garbage in the distributed system, while being robust to node failures. The full scheme employs two faulting, mark-and-sweep collectors on each node in the distributed system.

Before presenting our comprehensive and robust solution to the distributed garbage collection problem in details, the goals of distributed garbage collection are described (Sect. 2). Based on these, an overview of recently and related work on distributed garbage collection is given (Sect. 3), followed by a sketch of the basic Emerald garbage collection scheme (Sect. 4).

The Emerald solution is detailed in the following sections. First (Sect. 5), we discuss

comprehensive garbage detection in a failure-free distributed system, and next (Sect. 6), the expedient collection of local garbage is discussed. Then robustness to failures is added, using distributed control and a distributed termination detection algorithm (Sect. 7). Sect. 8 gives some remarks on storage reclamation, and finally, we summaries our contribution (Sect. 9).

2 Goals in Distributed Garbage Collection

Distributed garbage collection is not only faced with the traditional problems of garbage collection, the very nature of distribution poses further challenges. specifically, robustness to partial failures is a goal that impacts all other goals. We identify the following general goals in garbage collection schemes. The consequences, when taking robustness into consideration, are described in the rightmost column:

	General goals	Robustness considerations
Comprehensiveness	*All* garbage is collected, e.g., no memory leakage.	Continually adapting to the current available parts, while waiting for unavailable parts to become available again as necessary.
Concurrency	The collector and mutators run concurrently on all nodes.	Mutators must not be blocked by a collector due to unavailable parts.
Expediency	Delivery of garbage for recycling in a speed comparable to the speed of new allocation requests.	Collection must complete despite unavailable parts.
Efficiency	Limited overhead per byte of storage collected introduced by each step and the total number of steps needed.	Failures and their circumvention must be handled efficiently. Additional overhead due to robustness must be limited and mainly paid when failures are present.
Correctness	Only garbage must be collected, e.g., no dangling references.	References to unavailable parts and references in checkpointed files must remain valid.

An ideal scheme would fulfill *all* of these goals; unfortunately, this is not possible in general. A comprehensive collection depends on all nodes in the distributed system, thus, the mere presence of communication delays in distributed garbage collection often rules out the possibility of fulfilling the goals of comprehensiveness and expedience by a single collector. Thus, a trade-off between comprehensiveness and expediency is necessary.

By trading off comprehensiveness, we achieve a partial collection, i.e., only part of the garbage is collected. Figure 1 describes the various degrees of partial collection, from collection of nothing, to collection of all garbage, with a broad variety of conservative collectors, which are able to collect only part of the garbage, in between . From the most

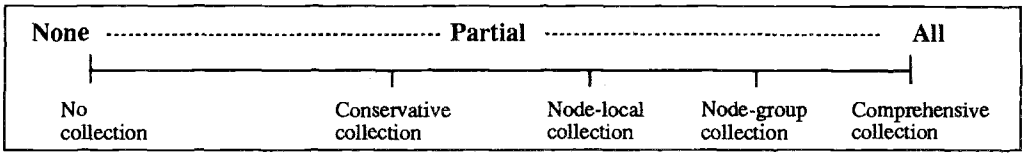


Fig. 1. The degree of partial garbage collection

conservative collectors like Boehm-Weiser [Boehm 88] and Mostly-Copying [Bartlett 88] to comprehensive collectors like the ones implemented for POOL [Augusteijn 87] and Emerald [Juul 92]. In between, we find collectors based on smaller or larger parts of the system with conservative estimates of references from other parts. In distributed systems these may span from one node, over groups of nodes, to nearly all nodes. As a supplementary, expedient collector Emerald employs such a node-local collector on each node. The Galileo and SOR projects are examples of such node-group collectors [Mancini 91, Shapiro 90]. Also [Lang 92] describes distributed collection schemes, which eventually reclaims all inaccessible objects. The partitioning and grouping techniques in the distributed environment, has many similarities to non-distributed techniques like area collection and the partitioning used by generational scavenging.

3 Related Work

Distributed systems like POOL [Augusteijn 87, Beemster 90], Galileo [Mancini 91], Argus [Liskov 86], and those implemented in the SOR project [Shapiro 90, Shapiro 91] all employ garbage collection. These distributed garbage collectors fulfill most of the previous mentioned goals. They do, however, compromise on various aspects of the goals to be able to achieve some of the others.

In general, the non-comprehensive collectors are able to collect more efficiently, while being both expedient and robust to node-failures. The solution described by Shapiro to be implemented in project SOR is robust but may fail to collect all garbage, instead, it has the potential for being expedient. Lang, Queinnec, and Piquer further refine the scheme based on independent node collectors to cooperate for various groups of nodes. The scheme is expected [Lang 92] to be comprehensive under the assumption that each distributed (interconnected) graph of garbage objects is contained in a group of nodes, which do not fail during the collection of the group, i.e., failures during the collection are not tolerated.

The collector for Galileo is implemented as a global stop-and-copy collection of objects on stable storage. If nodes become unavailable a fault-tolerant adaption enables a non-comprehensive collection to complete. The scheme may limit the collection to a subset of nodes and thus only blocking mutators on those nodes. A comprehensive collection is, however, not guaranteed.

A comprehensive collection is often more costly and assumes a simple failure model or no failure at all. The collection scheme for the POOL system is based on global synchronization of node-local mark-and-sweep collectors, and to be comprehensive it depends on complete node availability. The garbage collector for POOL, as described in [Augusteijn 87], does not cope with failures in the distributed system.

The idea of node-local collectors that cooperate has also been used by Liskov and Ladin in Argus. The cooperation is made possible by introducing a logically centralized, but physically replicated, highly available service for cross-node references. Any such reference is registered at the service, which also does cross-node garbage cycle detection. The service is based on synchronized local clocks and bounded delays of cross-node message exchanges.

4 The Emerald Garbage Collection Scheme

Based on the ideas in [Jul 87, Jul 88a, Jul 88b], [Juul 92] describes the design and implementation of a garbage collection scheme for the distributed, object-based system, Emerald, which fulfills all the goals listed in Sect. 2. The solution combines the comprehensiveness of mark-and-sweep collection with distribution and concurrency. The primary methods to achieve concurrency and robustness are distributed control, object protection and faulting, and the use of more than one collector concurrently.

Robustness to partial failures in a distributed system can be achieved when all parts work independently. Still cooperation is needed, but distributed control makes partial failures less threatening to the system. Robustness to partial-failures in distributed systems has lead us to implement our garbage collector without using centralized control.

4.1 The Basic Mark-and-sweep Algorithm

Our basic algorithm is based on a *concurrent* variant of mark-and-sweep garbage collection. The mark-phase is done concurrently with user processes running by *protecting* non-marked objects from being used by the running processes with a *garbage collection fault* mechanism similar to a page-fault mechanism in a virtual memory system [Appel 88, Appel 91]. In Emerald such a protection and faulting mechanism is already available in the implementation of *remote invocation*. Thus, the utilization of the mechanism by the garbage collector is nearly free.

The mark-phase of our basic algorithm uses the traditional three color settings: white, gray, black. Objects are marked either white (potentially garbage), gray (alive with references under consideration), or black (alive with references considered). Furthermore, a root set of objects is given, i.e., the active processes and the "always present" objects.

The mutators may run without problems in the black objects and the algorithm assures that mutators execute in black objects only. They may reference other black or gray objects, but the gray objects are protected. Thus, a mutator, which tries to use a gray object, will be suspended while a fault handler marks it black and traverses it to ensure that the objects, it references, are marked and protected. This way mutators are only faced with black objects. When the collection is started, all objects are white, and all mutators are stopped. Before each mutator is resumed, it is marked black, and its references is marked at least gray. The mark-phase is finished when all gray objects have been traversed and marked black, i.e., when the gray set is empty.

4.2 The Garbage Collection Invariants

The following invariants form the basis of our garbage collection algorithm. A detailed description of the basic algorithm and its implementation is found in [Juul 92]. The

general assumption is that *garbage stays garbage*. The collection scheme is based upon this assumption and five invariants.

Invariant 1 (Progression).

During garbage collection objects become darker, never lighter, i.e., shading is a monotone function moving objects from white to gray, and from gray to black.

Invariant 2 (Mutators).

Mutators execute in black objects only.

Invariant 3 (No black-to-white references).

No black-to-white references, i.e., a black object contains references to gray and black objects only.

Invariant 4 (Faulting).

Gray objects are protected, thus any attempt to access a gray object is withhold until the faulting mechanism has changed the object from gray to black (by shading its references at least gray).

Invariant 5 (Termination).

No gray objects indicates that black objects are the surviving ones, whereas the whites are all garbage (and thus reclaimable).

4.3 The Dual Collector Scheme

The Emerald garbage collection scheme consists of two sets of collectors, which are all applied concurrently. The *global* scheme, using one collector on each node in the system, continuously adapts to the current situation and strives to fulfill comprehensiveness while giving up on expediency. The *local* scheme foresees the failures of many parts of the system by performing an independent and expedient, but non-comprehensive, local collection on each node.

The comprehensive collection is achieved by one concurrent mark-and-sweep collector on each node, which cooperate as one global garbage collector across the entire network of Emerald nodes. The set does a comprehensive collection of all garbage, while various parts of the distributed system may be temporarily unavailable. A second set of collectors does an independent, partial collection on each node. These node-local collectors do a more expedient collection of local garbage without being comprehensive. Both sets of collectors proceed simultaneous and in parallel (*on-the-fly*) with the running processes. Each set of collectors adds robustness to the garbage collection scheme. The global collection by waiting for needed but unavailable nodes to become available again while progressing the collection in the available parts of the system. Whereas each local collector is able to collect local garbage while the rest of the system is unavailable. This further adds efficiency and expedience to the scheme, as most objects tend to be short lived and local [Lieberman 83, Schelvis 88, Jul 88b, Rudalics 86].

5 Comprehensive Garbage Collection

In terms of a graph, a comprehensive collection must partition the distributed graph of objects, connected by references, in two very well-defined parts. One containing *exactly all* the objects reachable from the distributed root set, and another containing the rest, i.e., *all* the garbage. During a comprehensive garbage collection, the graph must be traversed from the root set to identify the reachable objects, i.e., the closure of the root set. To ensure the collection of all garbage, the references in the root set and the references inside the objects must be identified exactly.

In general, any traversing algorithm has this property. Thus, the basic algorithm, even in the distributed case, can be based on either mark-and-sweep or copying collection. With focus on garbage *detection*, which is the harder part of the problem in the distributed case, the mark-and-sweep algorithm has been chosen due to its nice separation of garbage detection from garbage reclamation. Our current implementation pays little attention to compaction and locality of references. Though copying collectors may waste up to half of the available memory, they might be considered in future implementations where compaction is combined with object mobility.

In the comprehensive garbage collection scheme in Emerald any node may take initiative to a new *global collection cycle* and inform the other nodes in the distributed system about the decision. Each collector progress on its own node by initiating the collection and doing the marking. References to non-resident objects must, however, be treated differently. To the mutators on the node, we pretend that the non-resident objects are already black, while we accumulate references to them in the *non-resident gray set*. When the gray set of resident objects has been emptied, the non-resident objects are handled by sending a shade request to the node hosting the object. Meanwhile, remote requests to shade objects resident on our node are handled by putting these references in our gray set of resident objects. Each shade request is acknowledged by the node hosting the object, to let the requesting node remove the reference from the non-resident gray set. Thus, a gray reference will stay in the non-resident gray set until the node hosting the object guarantees that the object is at least gray, i.e., gray or black.

The mark-phase is finished when both gray sets are empty on all nodes. This global state is stable, in contrast to the state *both gray sets empty* on a single node. The global state is detected by a two-phase commit protocol. For simplicity, the global termination detection could be detected by approving a coordinator node. The current solution is, however, prepared for robustness to partial failures.

The cooperating collectors, constituting the global collection, may run very independent on each node. They only need to coordinate their actions on three topics:

1. When to start, i.e., when mutators must be stopped and the local part of the distributed set of root objects constructed.
2. During the mark-phase, i.e., when a non-resident object is shaded by requesting the node hosting the object and acknowledging the action back to the requesting node.
3. To determine when the mark-phase is finished, a distributed termination detection protocol must detect the *all gray set is empty* situation.

All nodes may decide to initiate a new cycle of the comprehensive collection and let this knowledge sieve to the other nodes. By adding the information (the cycle number) about a progressing collector in all inter-node messages, any node will become aware of

the situation before it engages in the transfer of objects or references with the started nodes. Though the garbage collectors may start at different wall clock time, we are able to identify a common logical clock where no collector were started before and all were started after.

6 A Dual Node-Local Garbage Collector

Due to both expediency and robustness the set of global comprehensive collectors has been extended with another set of independent local collectors.

The local collectors provide expediency by not depending on inter-node communication, and robustness by only using available nodes. We have chosen not to add a node-group collector scheme as a third, intermediate scheme. We find the clustering of nodes irrelevant to our current testbed of only a dozen nodes, to pay the additional cost of maintaining tables of incoming and outgoing references from each node. Furthermore, the two collector scheme is enough to fulfill our goals, thus a third scheme would not add substantially benefits.

The node-local collector is conservative in its definition of the root set, but not in its identification of references between objects. The conservative approach is acceptable, as this collector is supplementary to the comprehensive scheme. The node-local collector will collect local garbage only, i.e., garbage which has never been reachable from other nodes.

The implementation is fairly simple. It takes advantage of a general mechanism that marks objects potentially reachable from other nodes as *ReferenceGivenOut*. When a reference to a resident object is exported to another node, the object is marked as known from outside. Though that reference may later be dropped, the object stays marked during the rest of its lifetime. These marked objects are added to the root set of the node-local collector. When this collector finds references to non-resident objects, they are simply skipped.

Beside the extended root set and the missing needs to communicate with other nodes, the local collector uses exactly the same algorithm as does the global collector on each node. The two collectors on each node need, however, to synchronize, as they are working on the same data. Thus, they have their own data structures and mark fields for color information. To prevent either of them from reclaiming objects, later needed by the other, the two collectors may not have a non-empty set of resident gray objects concurrently. On each node this is achieved by only starting the local collector when the global has an empty resident gray set, which is achievable without communication delays, and by only starting the global collector between the end of the mark-phase and the start of the next local collection.

7 Robust Garbage Collection

The comprehensive global garbage collection scheme is threatened by node failures, as these influence the global state as well as the individual collectors.

The failure model is *failed-stop*, thus, by keeping its main state on stable storage, the collector on each node may always restart in a globally well-defined state. As an aside, a node failure is by itself an effective garbage collection, as a restarting node gets all

storage reclaimed, except checkpointed objects. These are saved, e.g., on stable storage, and recovered after the failure. This means that live references may reside on nodes currently unavailable due to a failure.

From a global point of view, node failures may influence the garbage collection as follows:

Before a garbage collection	No harm.
During the start of a new collection	Nodes may become out of step concerning the global garbage collection state.
During the mark-phase	The global invariants about colors and mutators may be broken.
Finishing the mark-phase	The global termination detection needs reliable information about all nodes.
During the sweep-phase	No harm.

Robustness to node failures, i.e., to partial failures of the distributed system, must take the above situations into account. The problems occur exactly in the situations where the collectors on each node need to coordinate their action (see the listing of the very same three points in Sect. 5).

7.1 Starting a New Collection

It is fairly easy to assure that restarted nodes adapt to the global situation. The presented mechanism to ensure synchronization by tagging all inter-node messages, will also force a restarted node to enter the same garbage collection cycle before it engages in mutating the object graph. If it keeps running locally only, it may, however, not be aware of the progressing global garbage collection on the other nodes, until one of these sends out shade requests or tries to detect global termination. This will eventually happens, and when it does, the restarted node may immediately adapt to the situation, without breaking the global invariants. From a global point of view, all actions, done by the restarted node until then, have taken place before the logical global clock of the start of the mark-phase.

7.2 A Robust Mark-Phase

During the mark-phase, node failures have several impacts. Both while a node is failed and when it is recovered.

When recovering, a node will restart its collection, but come up with references to non-resident objects. It will need to send out shade requests for these references, even though it might have done so before the failure occurred. Shading is an idempotent function, thus no invariant is broken, only performance is degraded (but this is insignificant compared to the node crash and reboot sequence). The scheme also covers the cases where a shade request has been sent but the reply was lost. For better performance, a node may save its received acknowledgements to remote shade requests on stable storage and use this information when recovered.

While a node is unavailable, other nodes may have references to objects on it. They cannot shade these objects until the node is available again. For simplicity, we have implemented the shade request mechanism as a repeated broadcast with exponentially back-off. Thus, non-acknowledged requests will be sent out until they are eventually acknowledged.

7.3 Distributed Termination Detection

The detection of the global state *all gray set are empty* can be difficult when nodes may fail independently and randomly often. To achieve comprehensiveness we must ensure that all nodes have finished their mark-phase and that no request is in transit.

The latter is ensured by the acknowledgement of shade requests. The references in the non-resident gray set are kept there until an acknowledgement is received by the shade reply mechanism (Sect. 5).

The two-phase commit protocol can be started by any node. A node may do so when it believes that the collection is done. Such a decision is based on its own status and the network traffic. More precisely, both gray sets of the node must be empty, and no nodes must have been broadcasting shade requests for a while. The termination detection protocol is robust to temporary node failures; it only depends on nodes being pairwise available. The current implementation depends on each node being aware of all other nodes in the system. This assumption holds in the current Emerald prototype. A system with a very large number of nodes should use another protocol.

8 Storage Reclamation

Though we emphasis on garbage detection, a short presentation of the reclamation part of our garbage collection scheme is given here to complete the picture.

The sweep-phase of all our collectors (both local and global) has been relinquished from the mark-phase in a scheme similar to the *mark-during-sweep* scheme proposed in [Queinnec 89]. On each node one common sweeper takes care of the sequential traversal of the node-local heap.

The scheme is based on marking with the current garbage-collection cycle number, instead of marking objects black, gray, or white. The gray information is kept aside already, as this information is used by the faulting mechanism also. During the mark-phase the current cycle number represent the color black, and the previous cycle, the color white. Objects marked with lower numbers are identified garbage, waiting for the sweeper to pass by and reclaim them. At the end of the mark-phase the previous cycle number becomes a garbage indicator, just like white indicates garbage when the mark-phase is finished. When the next garbage collection cycle is started, the cycle number is incremented, effectively turning all objects to be considered white until they are marked again. New objects are always born with the current cycle number, i.e., black.

The sweeper is interleaved with the allocation routines, i.e., each time a new allocation request is received, the allocator tries to reclaim the same amount of storage from the heap. It does so starting from its current sweep position in the heap and moves forwards (viewing the heap as a circular list of objects) until the requested amount is reclaimed or no more objects marked with old cycle numbers exist.

9 Conclusion

The Emerald garbage collection scheme has reached its goals by running two mark-and-sweep collectors on each node in the distributed system:

- The global collectors cooperate to achieve a *comprehensive collection* of all garbage.
- The global collectors are *robust to node failures.* The collection progresses on the available nodes as fast as possible before they wait for the still needed, but failed, nodes to become available again. All nodes need not be available concurrently.
- The local collection ensures that local garbage is collected on a per node basis independent of the current status of other nodes, thus achieving an *expedient collection.*
- The garbage collection *faulting* mechanism has made concurrency with mutators possible and thus limit the length of pauses introduced by garbage collection on user processes.

The measurements and experiments with the implementation will present a definite evaluation of the presented collection scheme. Due to a very untimely disk crash whereby parts of the current implementation were lost, an evaluation of the prototype is, unfortunately, not available as this article goes to press.

Acknowledgement

We gladly acknowledge the comments and proof-reading by Birger Andersen. Also the comments from the anonymous referees helped clarifying several points in the presentation; they are hereby acknowledged.

References

[Appel 88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 23(7), pages 11–20, ACM, SIGPLAN, Association for Computing Machinery, Georgia, USA, July 1988.

[Appel 91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV Proceedings in: *SIGPLAN Notices* 26(4), pages 96–107, ACM SIGARCH/SIGOPS/SIGPLAN and IEEE Computer Society, TC MM / TC VLSI / TC OS, ACM Press, Santa Clara, California, USA, April 1991. Simultaneous published as SIGARCH *Computer Architecture News* 19(2) and SIGOPS *Operating Systems Review* 25, special issue.

[Augusteijn 87] Lex Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, pages 75–93, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1987.

[Bartlett 88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* WRL Research Report 88/2, Digital, Western Research Laboratory, Palo Alto, CA, USA, February 1988.

[Beemster 90] Marcel Beemster. Back-end aspects of a portable POOL-X implementation. In Pierre America, editor, Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, pages 193–228, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In OOPSLA'86, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 21(11), pages 78–86, October 1986.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Boehm 88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice & Experience*, 18(9):807–820, September 1988.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-01.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report.* Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, Washington, October 1987. Also available as DIKU Report (Blue series) no. 87/22, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark and as TR no. 87-29, Department of Computer Science, University of Arizona, Tucson, Arizona.

[Jul 87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 105–106, Association for Computing Machinery, December 1987. Extended abstract only; full paper published as [Jul 88b].

[Jul 88a] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, December 1988. Technical Report no. 88-12-6. Also available as DIKU Report (Blue series) no. 89/1 from Department of Computer Science, University of Copenhagen, Denmark.

[Jul 88b] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobilty in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Juul 92] Niels Christian Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System, Emerald.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, 1992. In preparation.

[Lang 92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, ACM SIGPLAN and ACM SIGACT, Association for Computing Machinery, Albuquerque, New Mexico, USA, January 1992.

[Lieberman 83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Liskov 86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th annual ACM Symposium on Principles of Distributed Computing (PODC'5)*, pages 29–39, Association for Computing Machinery, Vancouver (Canada), August 1986.

[Mancini 91] Luigi V. Mancini, Vittoria Rotella, and Simonetta Venosa. Copying garbage collection for distributed object stores. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

[Queinnec 89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING sweep rather than mark THEN sweep. In E. Odijk, M. Rem, and J.-C. Syre, editors, PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, pages 224–237, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1989.

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software – Practice & Experience*, 21(1):91–118, January 1991.

[Rudalics 86] Martin Rudalics. Distributed copying garbage collection. In William L. Schelis and John H. Williams, editors, *1986 ACM Symposium on LISP and Functional Programming, Proceedings of*, pages 364–372, ACM SIGPLAN / SIGACT / SIGART, Association for Computing Machinery, Cambridge, Massachusetts, USA, August 1986.

[Schelvis 88] Marcel Schelvis and Eddy Bledoeg. The implementation of Distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, ECOOP'88, European Conference on Object-Oriented Programming, Proceedings published in: *Lecture Notes in Computer Science* 322, pages 212–232, Springer-Verlag, Oslo, Norway, August 1988.

[Shapiro 90] Marc Shapiro, David Plainfossé, and Olivier Gruber. *A garbage detection protocol for a realistic distributed object-support system.* Rapport de Recherche INRIA 1320, INRIA-Rocquencourt, Paris, France, November 1990.

[Shapiro 91] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

This article was processed using the LaTeX macro package with LLNCS style

Experience with a Fault-Tolerant Garbage Collector in a Distributed Lisp System

David Plainfossé and Marc Shapiro

INRIA Project SOR, Rocquencourt BP 105, 78153 Le Chesnay CEDEX, FRANCE
David.Plainfosse@inria.fr

Abstract. In order to evaluate our fault-tolerant distributed garbage collection protocol, we have built a prototype implementation within a distributed Lisp system, *Transpive*, replacing Piquer's native indirect reference count distributed garbage collector. This paper presents our protocol and highlights implementation issues on Transpive. In particular, we describe the prototype and the alterations required to fit into the Transpive distributed programming model. The message and CPU performance of our protocol are measured and its fault-tolerance evaluated. We conclude that the cost of our protocol is close to Piquers's , although our protocol has greater functionality.

1 Introduction

Garbage collection (GC) has recently become of increasing interest in distributed systems [6, 9]. The motivations for such a service are numerous. First, transparency: Just as modern distributed systems support transparent, uniform placement of and invocation on both local and remote objects, so should they also support transparent object management, including reclamation. Second, storage management is a complex task, not to be managed by users. Distributed GC is even harder than local GC because the local collectors must be coordinated, to consistently keep track of changing references between spaces. This consistency problem is further complicated by the common failures of distributed systems such as lost, duplicated, and late messages, and crashes of individual spaces.

Distributed garbage collection poses a challenging problem: reclaiming all kinds of data structures while achieving efficiency, scalability and fault-tolerance. In spite of the difficulty, a number of proposals have attempted to design a distributed GC that fulfills all these requirements. The great number of incomplete proposals (see Sect. 6) reflects how difficult the challenge is. However, the combination of several complementary techniques may lead to an almost perfect algorithm. For instance, combining Lang *et al.* cyclic distributed GC [6] with our fault-tolerant algorithm could gain a fault-tolerant and cyclic garbage collector.

To address these issues, we have designed a fault-tolerant distributed garbage collector protocol, hereafter called the SGP protocol [13, 14] based on reasonable, weak assumptions. It scales to any number of nodes. It continues to function correctly in the presence of lost, duplicated, or out-of-order messages, or of (fail-stop) node crashes; it allows objects to migrate or become deleted while referenced.

In order to evaluate the SGP protocol, we have prototyped it on a distributed Lisp, Transpive [12], implemented at INRIA, running on a multi-Transputer board hosted by a Sun server machine. For the purpose of this evaluation, we replaced Piquer's original Indirect Reference Count (IRC) garbage collector [11], provided with Transpive, with a prototype implementation of the SGP protocol. SGP provides all the functionality of Piquer's GC, and in addition is resilient to message or site failures. The motivations for this approach are the following:

- ease of prototyping the algorithm in a functional language,
- use of an existing, easy-to-use, clean, distributed programming environment,
- existence of a local tracing collector as required by SGP,
- possibility of comparison with Piquer's GC.

The organization of this paper is the following. Section 2 describes briefly the SGP protocol, and highlights mechanisms implemented on Transpive. Section 3 reviews the distributed programming model of Transpive and its implementation. In particular, issues relevant to the SGP implementation are highlighted. Then, in Sect. 4, we further describe the implementation itself. Section 5, presents performance measurements of our prototype implementation. Section 7 concludes the paper. We compare our performance results with Piquer's.

2 Brief Description of the SGP protocol

We consider a collection of *spaces* connected by an unreliable non-FIFO channels. A space is either a process, a processor or a group of machines. Spaces may contain one or more applications called *mutators* performing independant computations. Mutators allocates dynamically objects in their space. An Object is located in a single space but may migrate. Objects may contain references to other objects located in the same or in remote spaces. Local objects are accessed through standard pointers whereas remote objects are accessed via remote pointers. An object accessible from at least one remote space is called *public*, as opposed to *private* objects. A public object belongs to a single space, its *owner*. Public and private objects are dynamic sets. That is, any non-garbage private object may become public and vice-versa. For instance, a public object only remotely referred by a single space may migrate to that space. After migration, the object is considered as private. Basically, the distributed GC is in charge of tracking remote accessibility of public objects. Private objects do not concern us and are reclaimed by the local GCs.

The mutator rests upon two separate layers of object management (see Fig. 1). The bottom layer is independent of object semantics, structure, or programming language: this is the distributed garbage collection specified in [14] and described briefly herein. The distributed garbage collection only propagates accessibility information supplied by the upper layer.

The upper layer is a (language-specific) run-time, extended to interface with our distributed GC. In the upper layer, one finds storage management (object allocation,

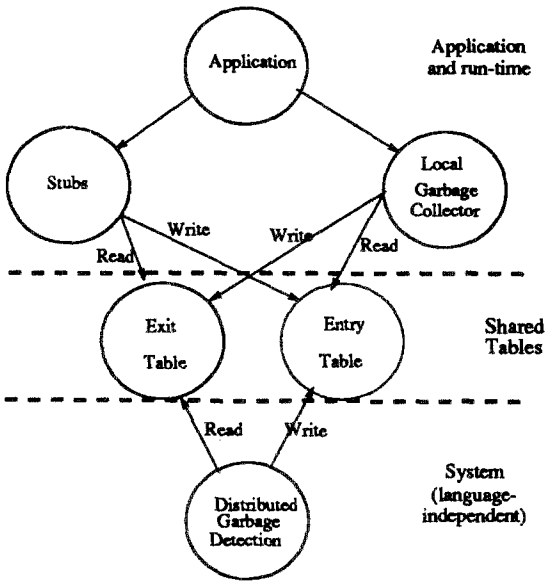


Fig. 1. Relationships between processes and main data structures

and local tracing garbage collection) as well as remote invocation functions (i.e. communication stubs).

The two layers share information in the form of incoming and outgoing references. An incoming reference is called an *entry item* and an outgoing reference is called an *exit item*. Cooperation between layers is limited to simple interactions to maintain consistency between entry and exit items.

Mutators in different spaces communicate via RPC-style invocation, i.e. by messages. An invocation is mediated by mechanically-generated stubs for marshalling and unmarshalling messages; a stub interfaces between the application and the system, encoding typed information into a typeless form. The arguments and results in an invocation contain any mixture of pure data, references, and migrating objects. When sending or receiving a message, the stub creates either an entry or exit item for the reference or the object embedded in the message.

To provide fault-tolerance, extra time and ownership information is piggy-backed onto the existing mutator messages. Occasional control messages are exchanged, in the background, to remove inaccessible entry items.

The SGP protocol relies on the existence of any standard local tracing garbage collector. The distributed protocol is based on a conservative extension of reference counting. Each space maintains a list of potential incoming and outgoing references, respectively called the *entry table* and *exit table*. Both the entry table and the exit table are conservative estimates. If two different spaces possibly refer to a single object of space A, each will be assigned an entry item in A. This differs from reference counting, and in particular from Piquer's IRC, because we need an entry per remote space to deal with unreliable communication. This policy renders entry item

deletion an idempotent operation and permit tolerating lost or duplicated message. In the former case any subsequent control messages received will allow us to reclaim previously garbage entry items. The latter case will have no effect since all garbage entry items would have been been previously collected.

Local garbage collection proceeds from the union of the local root and the entry tables and removes object and entries in the exit table. Since local GC starts from the union of the local root with the (conservatively estimated) entry table, all non-reachable local objects are true garbage. Each local GC cleans the entry table of useless entry items. In turn, exit tables are used to clean remote entry table, yielding successively better estimates.

When an exit item on space A is deleted, the corresponding entry item on space B can be removed. To this effect, a *delete message* can be sent from space A to space B. However this message can be duplicated or lost. To guard against loss, periodic *use messages* are sent from A to B containing the list of all existing exit items on A pointing to B; by comparison space B can deduce entry items that are not reachable, and remove them. In the remainder of the paper, *control message* refers to both use and delete messages.

One common problem in distributed systems is the message delivery delay. Messages containing references must be taken into account to guard against unsafe reclamation. Suppose that one space B sends a message to a space A containing a reference to a given object, say x. At the same time, a control message is sent from space A to space B to inform that the remote pointer on object x has been discarded. If object x is not locally referenced upon receiving the delete message, it will be remove from the entry table and collected at the next local GC.

To avoid this problem, we keep on each space a vector of highest timestamps and we timestamp entry items. When sending a reference, the stub creates the entry items and store in it the value of the local clock. The same value is used to timestamp the mutator message. Upon receiving a mutator message, the receiver compares the timestamp value extracted from the message with the one found in the vector of highest timestamps. This vector contains a space identifier and an associated timestamp for each remote space. A timestamp is increased each time a message is received. If the corresponding entry in the vector does not yet exist the initial value can be taken from the message. Control messages carry the current value of the timestamp vector corresponding to the target space. Upon receiving a control message, the timestamp value found in the message is compared to the value in the entry items to detect messages in transit.

Since our distributed protocol is based on reference counting, it fails to collect cycles[1].

[1] A separate sub-protocol [14] copes with inter-space cycles but its description is out the scope of this paper as it has not been implemented on this prototype.

3 Transpive

A garbage collector interacts closely with the programming model, as shown in Sect. 2. In particular, the way references are created, copied and sent is a crucial issue. For this reason, we first describe the programming model of Transpive, concentrating on key points related to the SGP implementation. Transpive is a distributed Lisp designed to provide a programming model as close as possible to a centralized Lisp, and in particular:

- to provide location-transparent invocation,
- to supply the basic functionality required by a distributed application through a small number of concepts,
- to provide a set of extensions, easily portable to another Lisp or runtime systems.

Transpive is layered on a Lisp interpreter.[2] One Lisp interpreter runs on each Transputer processor and interacts with the others through message passing. The underlying runtime system ensures FIFO, reliable message channels. Consequently, we have simulated message failures to evaluate the fault-tolerance aspects of SGP.

3.1 Sending and Receiving Messages

Transpive provides a function, **ext-send()**, to send a typed message to Transpive thread, addressed by a identifier **target_id** and a port number **port_n**. The **function** argument is used for marshalling/unmarshalling the **msg** given as argument. We have extended this function to accept an added argument:

```
ext-send (msg function  thread_id port_n delay)
```

The last argument, **delay**, simulates messages failures: out-of-order, delayed, lost, or duplicated messages. The target thread **thred_id** receives the message by calling the function **receive_from_any**:

```
msg := receive_from_any()
```

A Transpive message is a structure composed of several fields :

```
struct msg {
     data        ; the message data
     source      ; the sender thread_id
     target      ; the target thread_id
     send_type   ; the type of the message
     function    ; function for marshalling and  unmarshalling
     ; additional SGP fields:
     timestamp   ; value of the sender Lisp local clock
     delay       ; simulates unreliable messages}
```

[2] The current implementation runs on Le_Lisp [4], a fast Lisp interpreter implemented at INRIA. But the distributed model of Transpive is generic and easily portable to another Lisp dialect or functional language.

All these fields have default values. The **function** is used to marshal and unmarshal the object referenced by the **data** field. Several alternative marshalling semantics are provided by Transpive. Transpive servers use an efficient marshalling function **low_level** which does not generate remote pointers but copies values to the target Lisp.

We have added two fields to the standard Transpive message structure. The SGP protocol timestamps mutator messages to protect against unsafe, late or duplicate messages (see Sect. 2). The **timestamp** field is managed by the stubs.[3] This extension has no consequence on the other message functions. The field **delay** is used to simulate message failures, and is set from the **delay** argument of **ext-send()**.

The effect of the **delay** value is the following:

0 corresponds to the default normal delivering FIFO order,
$+n$ corresponds to a delayed message,
-1 corresponds to a lost message,
-2 corresponds to a duplicated message.

The delay is enforced by the **receive_from_any()** function which delivers the message to the application according to the value of the **delay** field. The $+n$ value indicates the number of times the messages is read in the queue without being delivered to the corresponding thread.

3.2 Remote References

Transpive supports transparent fined-grained object sharing. Lisp is a typeless language which only manipulates cons cells. Consequently, Transpive allows one to pass and access remotely any cons cell. The creation of remote references is totally transparent to the programmer. The corresponding data structures are created as a side effect of message passing. Specifically, stubs are responsible for detecting cons cells in messages and creating the corresponding entry or exit items to access the remotely referenced objects. Lisp does make any distinction between references and plain objects. Therefore, in that model, a reference is created for each cons cell passed in the message. Thus, each cons cell of a list may be accessed independently from other cells. This policy is required to keep the same semantics as any local Lisp. However, it creates a large amount of exit and entry items and worsens locality.

Transpive provides a cache memory associated with remote references. On first access to a public object through a remote reference, a replica of the object is copied to the local cache of the referencing Lisp. All subsequent read accesses to this object will be local, in the cache.

Conversely, an attempt to write a replica invalidates all other replicas. Ownership of the object is migrated to the Lisp which has attempted the write access. This scheme is well adapted to functional languages, such as Lisp, where read accesses are much more frequent than writes.

[3] Actually, the timestamp field is initialized at creation of a message. Stubs are responsible for updating the timestamp associated with each descriptor as explained in Sect. 2.

A public object always points to a *descriptor*. For this purpose, Transpive objects have been extended with an extra field, called *back pointer*, to access their corresponding descriptor. In order to save space, plain private objects don't refer to any descriptor and their back pointer is set to NULL. Depending on the existence or not of a cached replica, a descriptor acts either as a local handler on its cached replica, or as a remote pointer to a public object. With respect to the SGP model, a Transpive descriptor acts partly both as an entry and an exit item. It contains the following fields when corresponding to an exit items:

- The identification of the owner Lisp where the object is located,
- an OID which uniquely identifies the object throughout the system,
- a status, indicating whether the cached replica is valid or not,
- a "weak pointer" to the local replica; this pointer is not taken into account by the local garbage collector. Initially, this pointer is set to NULL. It is updated upon receiving a copy.

An additional field is present when the descriptor acts as an entry item:

- a list of pairs ((lisp_id timestamp) ...). The first one identifies a Lisp holds holds a remote pointer to that particular object. The second one is increased each time a message containing a reference to that object is sent to that lisp.

Each Lisp maintains a table of valid descriptors (TOD) indexed by OID of public objects. As a descriptor contains only a weak pointer to the local replica, another data structure, the list of public objects (LPO), is maintained to prevent public objects from being collected by the local GC. When an object becomes public, it is added to the LPO. When the last remote reference to this object is discarded, and becomes again private, it is removed from the LPO.

Upon sending a Lisp list composed of cells, the stub generates an OID and allocates a descriptor for each cell of the list. For instance, a call to `ext-send((1 2 3))` will lead to generate three OIDs and three descriptors for `(1 2 3)`, `(2 3)` and `(3)`. This is inherent in the Lisp object model and unfortunately leads to large space overhead.

Transpive provides, along with descriptors, a number of functions to access and set each field of a descriptor. These functions will be used in the remainder of this paper and are introduced to improve readability of source Lisp code. They are listed below in the same order as the list of fields given above:

```
desc:= get_desc(obj)          ; reads obj's back-pointer and returns descriptor
get_owner(get_desc(obj))      ; returns owner field of obj's descrpitor
get_oid(desc)                 ; get oid's field of desc
get_replica (desc)            ; get cached replica through weak pointer
set_owner(desc, lisp_id)      ; set owner field with the Lisp identifier lisp_id
set_oid(desc, oid)            ; set oid field of the argument descriptor desc
get_timestamp(desc, lisp_id) ; get the timestamp value embodies in
the descriptor for a particular Lisp
```

The function `get_desc(obj)` takes an object as argument and returns its back pointer (i.e. its descriptor). The function `get_replica(desc)` returns the cached replica object (if it exists) of the descriptor `desc`.[4]

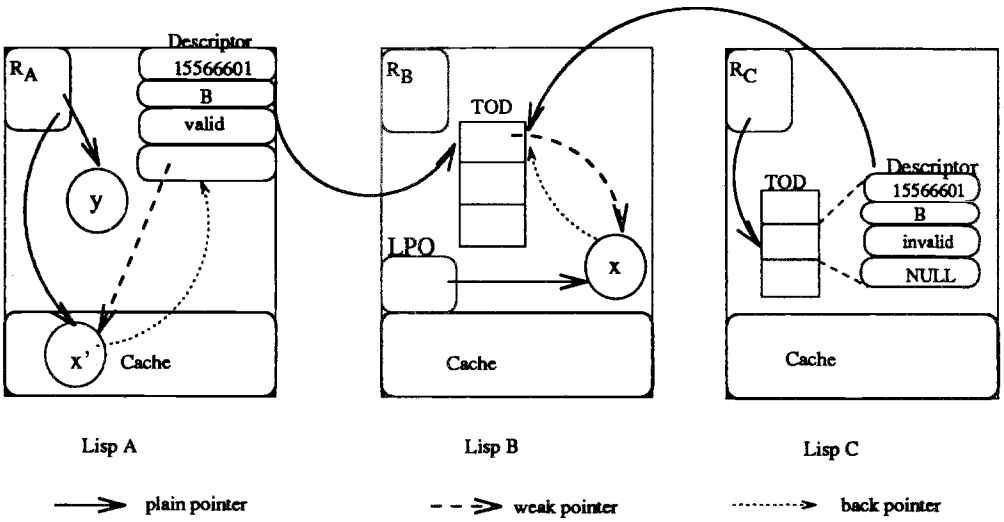


Fig. 2. remote references, descriptors and cache memory

Figure 2 shows three Lisps running on three different Transputers. Object x, owned by Lisp B is public and remotely accessible from Lisps A and C. Lisp A has already accessed object x from some root R_A and therefore has a replica of x in its local cache. A's reference to x points directly to the cached replica x'. In contrast, Lisp C has never accessed object x although it is accessible from its root R_C. Consequently, the reference to x points to the corresponding descriptor. Object y is a plain private object accessible from its local root R_A. Note that y does not refer to any descriptor because it is not public.

We have extended this descriptor to handle administrative information specific to the SGP protocol. As stated in Sect. 2, the SGP model assumes one entry item per remote space. In Transpive, a single descriptor may be referenced by several remote Lisps, and a counter associated to each descriptor embodies the corresponding reference count. We have adapted the SGP model to fit into the Transpive implementation of remote references. A new field has been added to each descriptor, pointing to a list of pairs. Each pair contains a Lisp identifier and a timestamp value. The Lisp identifier refers to a Lisp which remotely points to the corresponding public object. The timestamp value is updated each time a reference to the object is sent to this Lisp. The counter has been retained for compatibility but serves no useful purpose.[5]

[4] We tried to use the same variables names in the paper. In particular a descriptor will always be named as desc in pseudo code.

[5] However, we are in the process of removing them to compare memory consumption

4 Prototyping the SGP on Transpive

In the SGP protocol, a remote reference is created when a mutator passes a reference in a message. In other words, a creation message is a mutator message containing at least one reference. A *use message* (see Sect. 2) is sent by our collector to inform the owner Lisp which remote references have been discarded. We briefly describe here how we have implemented our protocol using the Transpive mechanisms introduced in Sect. 3.

4.1 Timestamps

Each Lisp maintains a vector of highest timestamps called the HTS vector. The HTS vector is updated each time a Lisp receives a mutator message. To handle the HTS vector, we have modified the original Transpive server of messages. This server receives all the messages exchanged between mutators and forwards them to the target thread. Actually, a call to the `ext-send()` function sends a message to a target thread via this server. Each time the message server receives a mutator message it extracts the timestamp, updates the corresponding entry of the HTS vector, then queues the message for the receiver.

```
PROCEDURE server_msg()
   msg : message;
   WHILE true do
        msg := receive_from_any()
        IF msg.timestamp GREATER THAN hts[msg.sender] THEN
           hst[msg.sender] := msg.timestamp
           queue the message to the target thread
        ELSE
           ignore the message
END END END
```

4.2 Cleanup of Public Objects

In Transpive, garbage collection of a descriptor occurs in several steps. Figure 3, shows the sequence of events involved in the collection of a public object. As stated earlier in Sect. 3, a descriptor is useless when the back pointer of its local replica does not reference it. Here are the relevant events:

1. On Lisp A the last reference to the local replica x' of x is discarded.
2. On Lisp A, a local GC occurs. The replica is collected and its descriptor pointer is updated.
3. The matching descriptor on Lisp A is then collected by the cleaning function `cleanup_tod`.
4. Consequently, a control (use) message is sent to the owner Lisp B.

between the SGP and IRC protocols.

5. On Lisp B, the SGP server receives the delete message and removes the corresponding object from the LPO.
6. On Lisp B, public object x is not locally referenced. It will be collected at the next local GC.

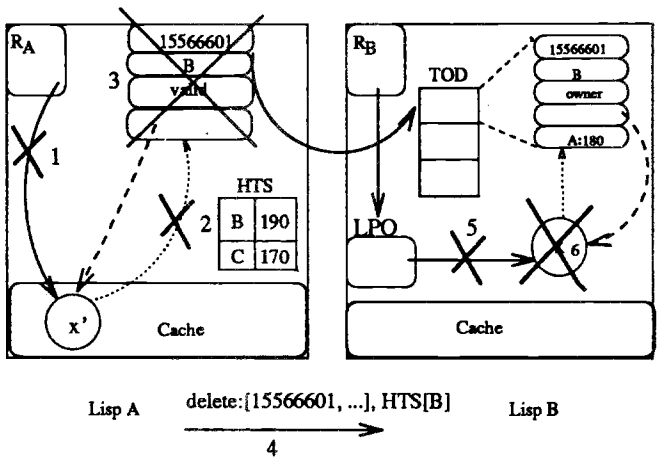


Fig. 3. Chronology of events in garbage collecting public objects x of Lisp B

SGP model assumed that entry items were collected by the local GC. The distributed programming model of Transpive enforces a totally different scheme for collecting descriptors. As stated earlier in Sect. 3.2, SGP's exit table is modelled by the TOD in the Transpive model and exit items are represented by descriptors. In contrast to exit items, a descriptor refers either a local object or a replica. Conversely each public object or replica points to its descriptor through its back pointer. Consequently, a descriptor cannot be collected as long as its replica is not collected. For this reason, garbage collection of descriptors proceeds in two steps. First, the local cached replica is collected and its back pointer is set to NULL. Later, the previously pointed descriptor is considered as garbage and will be collected by a cleanup function as shown by the code below:

```
FUNCTION cleanup_tod() : descriptors_list
  BEGIN critical section
      FOREACH desc IN TOD DO
         IF replica's back pointer refers to desc THEN
            add desc's OID to the use_list for the corresponding owner_lisp
         ELSE
            removes desc from the tod
      END END
   END critical section
   return TOD;
END
```

This function `cleanup_tod()`, cleans the TOD by removing useless descriptors. Each time a descriptor is detected, its OID is added to the `oids_list` if the descriptor is still valid. As an optimization, the messages are not sent at once to the corresponding Lisp, but rather buffered.[6] A high priority Transpive daemon `flush_msg_list` is responsible for traversing the list of messages and sending control messages as shown by the code below :

```
PROCEDURE flush_msg_list()
  BEGIN critical section
   FOREACH (target_lisp oids_list) pair IN use_list DO
     msg.timestamp := get_hts(target_lisp)
     msg.data := oids_list
     ext-send(msg, low_level(), target_lisp, GC_PORT, random())
   END
   reset use_list to nil
 END critical section
END
```

A control message is composed of the following fields:

- the corresponding entry of the vector of highest timestamps `hts`,
- the list, `l_obj_id`, of valid OIDs depending

Message lists are composed of pairs (`lisp_id l_obj_id`). The first element is a lisp identifier and the second a list of OIDs. The function traverses the whole `use_list` and sends to each target Lisp a corresponding subset of the valid OIDs. Note that `ext-send` calls are done with a delay argument. The function `random()`, in the code fragment above, generates a random value corresponding to either a delay, a lost, or a duplicated message. The function `get_hts` is used to timestamp the control messages and prevents a public object to be discarding if a reference is in transit (see Sect. 2). All these control messages are sent to a specific Transpive port associated with our SGP server. Note that we have to bypass the normal reference marshalling scheme in order to avoid the creation of remote pointers when sending control messages. The `low_level` Transpive marshalling function is used to avoid the creation of descriptors. This function marshalles control messages as a vector of integers and bypassed the traditional reference sending layer.

4.3 SGP Server

Each Lisp runs a server dedicated to receiving and processing control (delete and use) messages. The former kind contains a vector of unreachable OIDs whereas the latter contains whole sublist of the reachable OIDs between two Lisps (see Sect. 2). The `sgp_server` is activated each time a control message is sent from some remote Lisp by the `flush_msg_list` mentioned above. It extracts the relevant components of the control message and forwards them to appropriate function to update the local TOD.

[6] We haven't tried yet to piggy back delete or use messages on mutator messages.

```
PROCEDURE sgp_server()
   WHILE true DO
      ; wait for control message
      msg := receive_from_any()
      oids_list := msg.data
      msg_timestamp := msg.timestamp
      ; process control messages
      FOREACH oid IN oid_list
         desc := get_desc(oid)
         IF get_timestamp(desc) >= msg_timestamp THEN
            delete_desc(desc)
         ELSE
            a message in transit contains a reference to this descriptor
END END END END
```

4.4 SGP Interface with local GC

Le_Lisp provides an number of system signals. For instance, the signal **gcalarm** is activated just after each local garbage collection. This signal can be used to check the collection process to detect, for instance, a memory overflow. This signal invokes a user-defined function, which is a Null function by default. In our case, this function is responsible for cleaning the TOD and the LPO . Although it was stated in [14] that this cleanup could occur in parallel with other processing (only update of individual elements needs to be atomic), we implemented the whole procedure in a critical section as a quick first approximation.

```
PROCEDURE gcarlarm_sgp()
BEGIN
   BEGIN critical section
      ; removes from LPO previous public objects
      LPO := cleanup_lpo(LPO)
      ; cleanup the TOD of useless descriptors
      TOD := cleanup_tod(TOD)
   END critical section
END
```

The problem with this scheme is that the cleanup of remote pointers is bound to some local GC. This can lead to a memory overflow (in both implementations i.e Piquer's and SGP) since TOD cleanup is always delayed until after a local GC at the remote Lisp. This problem arises when a Lisp holds many remote pointers but uses only a small amount of its local memory. Since local GC is invoked on the basis of memory use, garbage remote references may not be collected for a long while. As a result, a high number of potentially garbage public objects are not collected, leading to a memory overflow. To avoid this problem, the cleanup protocols should be called not only after local GC, but also periodically.

5 Experiments

In this section, we analyse the measured performance of our SGP prototype and compare it with the IRC implementation. Two kinds of performance are discussed: the number of messages and their frequency andthe CPU overhead due to both kind of distributed GC.

5.1 CPU overhead

We have measured the CPU overhead due to our SGP implementation. We compare these results with the native distributed GC of Transpive. We have run two applications: a merge sort and a matrix multiplication. The measurements were taken on a Parsytec board composed of four Transputers (T800) with one megabyte of memory each, hosted by a Sun. We have measured each application twice on the same data to take into account the copies of objects. Since Transpive copies public objects, results are always better the second time. However these measures have been repeated dozens of times to be sure of the results and have shown extremely low variance.

Table 1. CPU performance measurements

Application	CPU time in seconds					Overhead (%)	
	Without DGC		IRC		SGP	SGP/IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5 4.1	17%	5%
(sort 200)	5.6	4.4	6.7	5.2	8.1 5.9	20%	12%
(multmat 20 20)	11.1	7.8	12.0	8.7	13.5 9.8	11.9%	12.3%

Table 1 shows the performance measurements. The results conform with Piquer's. We have disconnected the function responsible for sending control messages on each Lisp in order to avoid interrupting applications, but we kept all the control data management in order to measure the overhead on mutators until sending control messsage. The overhead measured is due to managing control data structures this represents the mutator part of the SGP protocol. Our SGP implementation is on average 10% slower than Piquer's and 20% slower than without any DGC. This slight overhead is encouraging. First, our basic motivation was to evaluate the SGP prototype; as a consequence, we did not pay too much attention to optimizations and kept a big part of Piquer's data structure management (for compatibility reasons). Second, the fault-tolerance property of the SGP protocol requires a some additional work, compared with Piquer's approach which largely justifies some added cost. For instance, we update descriptor timestamps each time a reference is sent.

5.2 Message Overhead

A second kind of measurements concerns the number of control messages sent, and their frequency. Our message sending protocol is different from Piquer's and slows

down local processing a little because group OIDs into a single structure, instead of sending a unique OID per control message. We have chosen, the former policy because it reduces message traffic. As shown on Table 2, this "buffering" strategy dramatically reduces the number of control messages sent in SGP compared with IRC protocol. Note that the number of control messages sent does vary a little between the two executions. Note also that we obtain the same results whatever the size of the list in the merge sort application. This shows that our message sending policy is somewhat independent of the number of objects sent between Lisps, although this buffering strategy may retain a big amount of floating garbage. For that reasons, this strategy should not have two much impact on control message frequency. This is particulary true when locality is very poor and the number of remote references is large, such as in Transpive.

Table 2. Control message measurements

Application	Control Messages					
	IRC		SGP		IRC - SGP	
(sort 100)	31	28	10	8	21	20
(sort 200)	41	39	10	8	31	31
(multmat 20 20)	101	96	20	18	81	78

6 Related Work

Distributed garbage collection is a difficult problem which has only been addressed partially. One key reason is that while most proposals rely on centralized techniques, adapting such techniques to distributed environments is not a straightforward task. Stop the world algorithms require costly termination mechanisms when facing distribution, whereas reference counting is completely defeated by common messages failures. In order to adapt those techniques to distributed environments, many recent proposals try to relax traditional invariants [2, 11, 15] whereas others rely on reliable communication protocols [3, ?, 6, 10]. The former family algorithms is usually based on reference counting. Therefore they cannot garbage collect distributed cycles and must assume that such graphs are rare. The second family ensure better liveness but all known algorithms are not resilient to message failures [6], may be completely defeated by space failures [3], or fail to address large network [9]. Our protocol belongs to the former family and bears some similarities to a number of proposals based on reference counting [2, 11]. Unlike those approaches, however, we maintain an entry item per source space that permits us to tolerate message loss whilst avoiding the dangers of duplicated messages.

Dickman [2] proposes *Optimizing Weighted References Counting* improving traditional *Weighted Reference Counting* [1, 15] in two aspects: message failures resilience and indirection cells. Resilience to message failures is provided through a weak invariant that requires that each object weight (total weight) is always greater or

centralized service to build a consistent view of the distributed system. Each local collector informs the centralized service about incoming and outgoing references, and about the paths between incoming and outgoing references. The path computation is expensive but necessary for reclamation of distributed garbage cycles. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects garbage (including dead cycles) with a standard tracing algorithm. The centralized service informs LGCs of accessibility of objects.

In a later paper [5] Ladin and Liskov simplify and correct the deficiencies of the above proposal, adopting Hughes' algorithm and loosely synchronized local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for an accurate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralized service determines the garbage threshold date, making a termination protocol unnecessary.

Recently Lang *et al.* [6] describe an original proposal to combine reference count and mark and sweep. The algorithm collect distributed cycles within predefined groups. Groups are dynamic collections of spaces (i.e a space may be removed or added during garbage collection) and may overlap or include other groups. The algorithm relies both on counters and local GC to perform mark and sweep within a group. Reference counts must be kept accurate, hence message failures are not tolerated. Group GC is conservative with respect to inter-group references: any subgraph referenced from outside the group is not collected until a larger group is formed encompassing the entire graph; therefore liveness is not guaranteed. Thus, large cycle reclamation requires extending group size such that the group includes all spaces that hold a cycle vertex. Distributed garbage collection of very large networks is proposed through a hierarchy of included groups. Included groups benefit from larger groups GC that perform some of their work. However, large group GCs are longer than smaller ones and therefore retain more floating garbage. For that reason, the authors assume that large group GCs are rare compared to small group GCs.

In [8] Lins and Jones combine *Weighed Reference Counting* with Lins'local algorithm for *Cyclic Reference Counting* [7] to address distribution issues. As a result, they propose a simple algorithm to garbage collect cycles in a distributed environment. The general idea of the algorithm is to perform a local mark-scan whenever a reference to a shared graph is deleted. That is, a mark-scan is initiated each time an object is suspected of belonging to a garbage cycle (i.e when its counter is decremented down to one). The mark phase decrements counters each time it visits an object belonging to the subgraph. At the end, all nodes with counters equal to zero are part of a dead cycle and may be safely reclaimed. Lins [7] improves the basic idea to perform the mark-scan lazily. Spurious objects are not scanned at once but instead they are queued in a special list. When the allocator fails to supply memory the corresponding list is scanned in order to reclaim potential garbage cycles. Unfortunately, mark-scan of subgraphs must be computed in critical sections. In other words, two different spaces cannot invoke cycle detection concurrently.

equal to the sum of all remote reference weights (partial weights). The weak invariant permit tolerating message loss but duplicated message remains problematic. The algorithm avoid the creation of indirections cells when partial weights cannot be split. However, this is enforced through a special `null weight` value. In this case, the total weight is always greater than the sum of partial weights preventing the object from being reclaimed by error. However, liveness is not ensured for *weak* objects which conform only the weak invariant. For this reason, the author assumes than the algorithm is always used in conjunction with a global tracing collector to reclaim garbage distributed cycles and weak objects.

In [11] Piquer describes his Indirect Reference Count (IRC) algorithm which improves Weighted Reference Count [1] by avoiding indirection cells. The algorithm also eliminates the need for increment messages that may conflict with decrement messages in traditional schemes. Thus, creation and duplication of a remote pointer are performed locally without informing the space where the object is located. In order to achieve local creation/duplication, remote pointers have been extended with a new field, named an indirect pointer. The indirect pointer serves only distributed GC purposes, and refers either to an object or to another remote pointer. The whole set of remote pointers referencing a single object forms a distributed graph which can be traversed using indirect pointers. Mutators never use indirect pointers, instead relying on the direct pointers to access objects in a single hop. As with others proposals relying on reference counting, the IRC algorithm is not resilient to message failures: liveness is not enforced against message loss and safety is not preserved against duplicated message.

Mancini and Shrivastava [10] describes an efficient and fault-tolerant distributed garbage collector based on reference counting. Resilience to space or message failures is supported granted to an RPC mechanism extended with detection and killing of orphans. A special protocol is used to cope with duplication of remote references. This protocol makes an early short-cut of potential indirections even if they are never used. Two alternatives are proposed to deal with distributed cycles : traditional and inefficient global mark and scan, and per object cycle detection based on an heuristic. The first one is notoriously inefficient and the second one does not collect all cycles.

Hughes [3] describes an elegant algorithm based on timestamps and local tracing. The algorithm timestamps objects and relies on the premise that garbage objects' timestamps remain constant whereas non-garbage objects' timestamps increase monotically. A timestamp threshold is computed to distinguish garbage from non-garbage objects. Objects that carry timestamps less than the threshold can be safely reclaimed. Unfortunately, the threshold computation relies on a termination algorithm which is notoriously costly and not scalable. Moreover, the algorithm is not resilient to space failures since a failed space prevents increasing the threshold, hence blocking garbage collection on all other nodes.

In contrast to many proposals that attempt to compute on each space the global accessibility of objects. Liskov and Ladin [9] rely on their highly available centralized service to compute global accessibility of objects on a single space. This service is physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are assumed to be backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These assumptions allow the

7 Conclusion

We have experimented with the SGP protocol on Transpive. The choice of Transpive allowed us to quickly implement the SGP protocol and to learn few lessons, although the distributed model of Transpive is quite different from SGP's. The original SGP model did not take into account the replication of objects. Consequently, we have adapted the SGP protocol into the replication model of Transpive. As a result, collection of out-going references —descriptors in the Transpive model— is more complex and slower than we expected. This increases the conservative aspect of the SGP and can be troublesome if memory is heavily in demand. A solution to decrease the delay for collecting out-going references is to decoupled local GC from SGP. Moreover, the fine grained sharing support of Transpive is definitely an uncooperative environment for a distributed GC. In particular, memory consumption is heavy since it requires a huge number of entries in the control data structures. As a consequence, it increases the overhead of SGP on application and the frequency of local GC. The performance results are encouraging but need to be improved , to minimize the overhead on applications. The buffering policy reduces dramatically the number of control messages. The resilience to message failures has been demonstrated. This result validates our design guideline. However, the fault-tolerance to space failures and duplicate messages remain to be investigated. Although, the SGP design relies on a very different distributed programming model, the prototype behaves correctly with respect to the safety property. It demonstrates that the SGP protocol is generic and adaptable. Therefore, it is a good candidate for a system service.

Acknowledgments

The design of the SGP protocol has been done in collaboration with **Olivier Gruber** of INRIA/RODIN. We wish to thank our colleagues of INRIA/ICSLA for their help and their availability to answering questions on Transpive, in particular **José Piquer**, and **Luis Mateu**. Many thanks also to **Daniel R. Edelson** and **Peter Dickman** for commenting on drafts of this paper.

References

1. BEVAN, D. I. Distributed garbage collection using reference counting. In *PARLE'87—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1987), no. 259 in Lecture Notes in Computer Science, Springer-Verlag, pp. 117–187.
2. DICKMAN, P. Optimising weighted reference counts for scalable fault-tolerant dis tributed object-support systems. (submitted to publication), 1992.
3. HUGHES, J. A distributed garbage collection algorithm. In *Functional Languages and Computer Architectures* (Nancy (France), Sept. 1985), J.-P. Jouannaud, Ed., no. 201 in Lecture Notes in Computer Science, Springer-Verlag, pp. 256–272.
4. J. CHAILLOUX, M. DEVIN, J. M. H. Le_lisp : A portable and efficient lisp system. In *Proc. 1984 ACM Symposium on Lisp and Functionnal Programming* (Aug. 1984), pp. 108–120.

5. LADIN, R., AND LISKOV, B. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.* (Yokohama (Japan), June 1992).
6. LANG, B., QUEINNEC, C., AND PIQUER, J. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.* (Albuquerque, New Mexico (USA), Jan. 1992).
7. LINS, R. D. Cyclic reference counting with lazy mark-scan. Tech. Rep. TR-77, University of Kent, Computing Labotory Canterbury (England), Aug. 1991.
8. LINS, R. D., AND JONES, R. Cyclic weighted reference counting. Tech. Rep. TR-95, University of Kent, Computing Labotory Canterbury (England, Mar. 1992.
9. LISKOV, B., AND LADIN, R. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing* (Vancouver (Canada), Aug. 1986), ACM, pp. 29–39.
10. MANCINI, L., AND SHRIVASTAVA, S. K. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Jounal 34*, 6 (Dec. 1991), 503–513.
11. PIQUER, J. M. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1991), vol. I of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 150–165.
12. PIQUER, J. M. *Parallélisme et distribution en Lisp.* PhD thesis, Ecole Polytechnique, Massy France, Jan. 1991.
13. PLAINFOSSÉ, D., AND SHAPIRO, M. Distributed garbage collection in the system is good. In *Proc. of the International Workshop on Object-Orientation in Operating Systems* (1991), pp. 94–99.
14. SHAPIRO, M., GRUBER, O., AND PLAINFOSSÉ, D. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), Nov. 1990.
15. WATSON, P., AND WATSON, I. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1987), no. 259 in Lecture Notes in Computer Science, Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style

Scalable Distributed Garbage Collection for Systems of Active Objects*

Nalini Venkatasubramanian**, Gul Agha and Carolyn Talcott
email: nalini@cs.uiuc.edu, agha@cs.uiuc.edu, clt@sail.stanford.edu

[1] University of Illinois, Urbana-Champaign, IL 06120, USA
[2] Stanford University, Stanford, CA 94305, USA

Abstract. Automatic storage management is important in highly parallel programming environments where large numbers of objects and processes are being constantly created and discarded. Part of the difficulty with automatic garbage collection in systems of *active* objects, such as actors, is that an active object may not be garbage if it has references to other reachable objects, even when no other object has references to it. This is because an actor may at some point communicate its mail address to a reachable object thereby making itself reachable. Because messages may be pending in the network, the asynchrony of distributed networks makes it difficult to determine the current topology. Existing garbage collection schemes halt the computation process in order to determine if a currently inaccessible actor may be potentially active, thus precluding a real-time response by the system. We describe a generation based algorithm which does not require ongoing computation to be halted during garbage collection. We also outline an informal proof of the correctness of the algorithm.

Keywords: actors, asynchrony, distributed systems, generation scavenging, network clearance, broadcast and bulldoze communication, snapshot.

1 Introduction

We describe a garbage collection algorithm, HDGC (hierarchical distributed garbage collection), for systems of active objects distributed across a network of nodes. An important advantage of our algorithm is that it is non-disruptive: it does not halt or otherwise interfere with the ongoing computation process. A novel feature is the recording of a GC-snapshot to obtain a consistent local and global view of the accessibility relation. The algorithm is described in terms of the actor model. However, it is applicable to any language supporting dynamic creation and reconfiguration of objects (passive or active), executed on a network with a *global name space* distributed across the nodes [3]. The HDGC algorithm can be adapted to a wide range

* This research was partially supported by DARPA contract NAG2-703, by DARPA and NSF joint contract CCR 90-07195, by ONR contract N00014-90-J-1899, and by the Digital Equipment Corporation.

** Current address: Hewlett Packard Company, 19111 Pruneridge Avenue MS44UT, Cupertino, CA 95014, USA.

[3] This memory architecture is often referred to as *distributed shared memory*

of parallel architectures including fine, medium or large grained MIMD machines, message passing, shared memory or distributed shared memory machines, or networks of workstations. This paper presents the conceptual aspects of the algorithm. An implementation effort is in progress. There are numerous possible optimizations. These are discussed briefly in the conclusion.

The Actor Model [Hew77, Agh86] provides a good abstraction for discussing concurrent computation in distributed systems. Here, the universe contains computational agents called *actors*. Each actor has a conceptual location (its *mail address*) and a *behavior*. The only way one actor can influence the actions of another actor is to send the latter a communication. Communication between actors is asynchronous, and every communication sent will be delivered after some finite but unbounded delay (fairness of mail delivery). If an actor α knows the mail address of an actor β, then β is called a *forward* acquaintance of α and α is called an *inverse* acquaintance of β. An actor can send communications only to its forward acquaintances. Mail addresses of actors may be communicated: thus the interconnection topology is dynamic. We call the actor addresses occurring in a communication the acquaintances of that communication. On receiving a communication, an actor processes the message and as a result may cause one or more of the following actions: (1) creation of a new actor, (2) alteration of its behavior and its acquaintances, (3) transmission of a message to an existing actor. Every actor is equipped with a mailbox that queues incoming communications.

In order to make sense of the notion of distributed memory management we need to refine the abstract actor model to account for local grouping of actors on nodes (processing units) and to account for the network interconnecting these nodes. We assume that the network consists of channels linking pairs of nodes. Each channel consists of a pair of directed links (one in each direction) with infinite message buffers. We require that message order is preserved across a single link and that the network routing satisfies certain progress-only constraints that will be made precise in the next section. In addition to normal messages between actors, there will also be special messages used for GC.

Traditionally, garbage is detected by starting with some pre-defined root set and forming the transitive closure of the acquaintance (referenced objects) relation. In actor-like systems there are two problems. First, the acquaintance relation is distributed and changes dynamically. Thus we must find a way of establishing a GC start time and determining the acquaintance relation as of this point in time, as a distributed snapshot. Second, simply following acquaintance links from the root set is not adequate. This is because, using that definition, a non-reachable actor can become reachable, at some later time, by communicating its address to a reachable actor. These problems are addressed in the HDGC algorithm by first obtaining a (locally and globally) consistent snapshot of the acquaintance relation, then computing reachability according to an algorithm that accounts for actors that are potentially reachable relative to the snapshot. The HDGC algorithm is *conservative*, i.e., it identifies only a subset of inaccessible objects during a GC. For example, a potentially reachable object may become inactive without communicating its mail address to any reachable object. However, all unreachable objects will be collected by some subsequent GC.

The remainder of the paper is organized as follows. In §2 we outline the full HDGC

algorithm. In §3 we present the algorithm for establishing a consistent snapshot of the acquaintance relation at the start of GC. In §4 we define reachability and present an algorithm for marking reachable objects. §5 contains an informal outline of a proof of correctness. §6 contains concluding remarks.

2 Hierarchical Distributed Garbage Collection

A hierarchical organization partitions a distributed system into smaller subsystems. These subsystems may in turn be further partitioned. The topmost level of the hierarchy is the entire system. The lowermost level of the hierarchy has a single node per subsystem. There may be zero or more intermediate levels. The organization of the distributed system into subsystems may be static or dynamic (cf. [LQP92]). The motivation for dividing a large, distributed system into smaller subsystems is to avoid the bottleneck inherent in global resource management.

To accurately determine garbage in a subsystem at any level other than the top level, it is necessary to know which internal actor addresses have been communicated to some external actor. Such actors are called the *receptionists* of the subsystem. They must be considered reachable (part of the root set) for a GC local to the subsystem. A receptionist table is constructed by adding an actor whenever a reference to that actor is passed out of the subsystem. This provides a conservative approximation to reachability. It can be improved by determining when entries in the receptionist tables are no longer accessible, but this requires global cooperation. The approximation can also be improved by maintaining a reference count of the number of outstanding references to each receptionist (cf. [SGP90, LQP92]). This also entails some overhead.

We present the HDGC Algorithm in the context of a two level hierarchy, i.e. global and node level collections. The generalization to hierarchies with intermediate levels is relatively straightforward. We can use any of the traditional algorithms for local GC. The best algorithm to use will depend on the granularity of the nodes as well as on particular application domains. It is not necessary for all nodes to use the same algorithm.

The HDGC algorithm consists of five steps: Pre-GC, DistributedScavenge, Local-Clear Initiation, Local-Clear, and Post-GC. There is a unique (per subsystem) special actor designated as the GC-root actor. Requests for GC go to the GC-root actor and sequencing of the GC steps are synchronized through the GC-root actor. Thus the algorithm does not require a global clock in the system. We describe the purpose of each step below. The steps are initiated and carried out by communication of GC related messages. Details are given in the following sections. The behavior of the GC-root actor will be described after these details have been filled in.

Step 1: Pre-GC. In a system with distributed state there is no uniquely determined global state. Thus to compute some property of the state it is generally necessary to determine a global snapshot that determines a consistent view of the state. In the case of the acquaintance relation for an actor system, the problem of obtaining a consistent global snapshot involves an additional subtilty. The asynchrony of communication together with the ability to communicate acquaintances means that at any given time, there can be communications in the network whose acquaintances

are no longer acquaintances of the sender, and not yet acquaintances of the receiver. This means that before a snapshot of the acquaintance relation can be taken, the network must be cleared of such communications. During the pre-GC step each node is notified that a GC has been initiated, and the network is cleared of messages in transit at the time GC was initiate. This defines a local start-of-GC time on each node that is globally consistent. Each node records GC information relative to its start-of-GC time that will persist throughout the duration of the GC. The combined local information forms a consistent global snapshot of the system state that is adequate to determine the reachability of each actor in the system. We call this the *GC snapshot*. A detailed description of information and of the process of recording the GC snapshot is presented in section 3.

Step 2: The Distributed Scavenge Phase. During this step, actors that are non-garbage relative to the GC snapshot are marked *touched*. The definition of non-garbage and the *distributed scavenge algorithm* for marking non-garbage actors is described in section 4.

Step 3: Local-Clear Initiation. Each node in the system is informed that the distributed scavenge phase has completed and local clearance begins. On each node, objects not marked touched are cleared from local memory, according the nodes method of memory management, and any other actions (updating receptionist tables, etc.) entailed by this reclamation are carried out.

Step 4: Local-Clear Phase. This step detects when all nodes have completed the local clearance initiated in the previous step.

Step 5: Post GC Broadcasts. This step informs each node that the current GC is complete: each node can now note that GC is no longer in progress and update necessary information to reflect this state. At the end of this step a new GC can be initiated at anytime.

Note that if GC is purely local, Step 2 becomes non-distributed and the synchronization provided by Steps 1, 3 and 5 are unnecessary.

3 Asynchrony in Distributed GC

In this section we describe how the start-of-GC time is established and how the recording of the GC snapshot is accomplished. The key idea is that in addition to ordinary (actor-to-actor) communications, new types of messages are introduced that propagate through the network in pre-established patterns, and can thus be used for various forms of synchronization. To describe these messages, we make additional assumptions about the network topology.

3.1 Message Routing in the Network

For simplicity we restrict our attention to networks of nodes that form two dimensional grids. Such a grid contains an $m \times n$ array of nodes. Each node is designated by a pair of integers (a_1, a_2), where $1 \leq a_1 \leq m$ and $1 \leq a_2 \leq n$. A node (a_1, a_2), is an *Fneighbor* of a node (b_1, b_2) if either $a_1 = b_1 + 1$ and $a_2 = b_2$, or $a_2 = b_2 + 1$ and $a_1 = b_1$. Similarly, a node (a_1, a_2), is a *Bneighbor* of a node (b_1, b_2) if $a_1 = b_1 - 1$ and $a_2 = b_2$, or $a_2 = b_2 - 1$ and $a_1 = b_1$. Connecting each Fneighbor/Bneighbor

pair of nodes X/Y is a channel comprised of a pair of unidirectional FIFO links, one from X to Y and one from Y to X. An *Fpath* is a path in the network that progresses only along Fneighbor links. A path in the network that progresses only along Bneighbor links is a *Bpath*. We call $(1,1)$ the *start node* of the system. It is the unique node from which there exists an Fpath to every other node in the system. Dually, we call (m,n) the *finish node*. It is the unique node from which there exists a Bpath to every other node in the system.

Ordinary messages are assumed to be routed from the node where the sender resides to the node where the receiver resides via paths that are *progress-only* in the sense that the paths contain at most one Fpath segment and at most one Bpath segment. Thus the route of an ordinary message is either an Fpath, a Bpath, an Fpath followed by a Bpath, or a Bpath followed by an Fpath.

In addition to ordinary messages, we introduce two kinds of node-to-node messages: *broadcast messages* and *bulldoze messages*. These messages propagate to every node in the system, and are used for synchronization and network clearance. The node-to-node messages may also contain information indicating actions to be carried out. *Broadcast messages* are propagated from the start node to all the nodes in the network, along some subset of links. The protocol for propagating a broadcast message is illustrated in Figure 1. Each node has a designated set of *broadcast predecessors* and *broadcast successors*. A node can issue a broadcast message to its *broadcast successors* only after it has received the message from all of its broadcast predecessors. The broadcast is considered complete when the finish node has received messages from all of its broadcast predecessors.

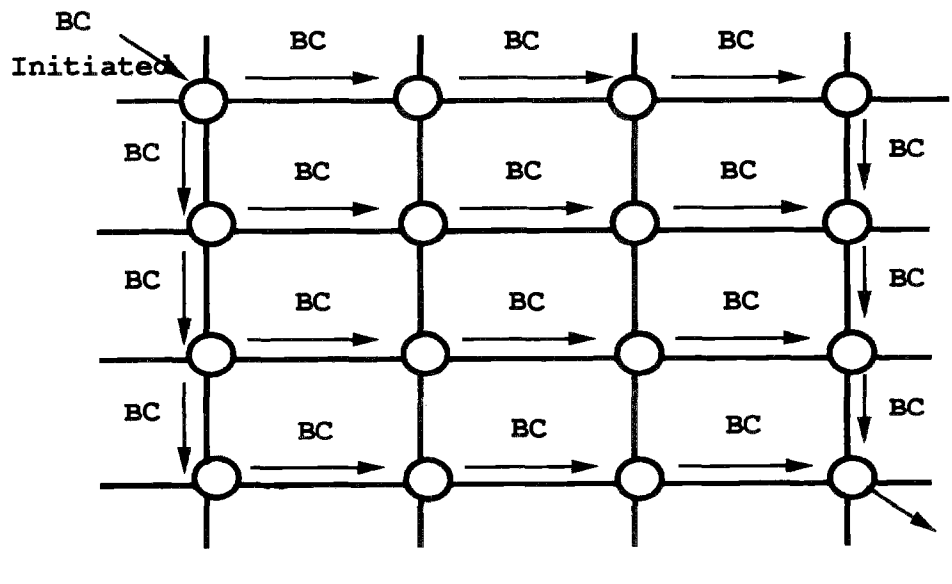


Fig. 1. The Broadcast Wavefront: The figure shows the broadcast messages traversing through the network as a wavefront. The broadcast messages are initiated at the *start* node and travel along indicated route to every node in the network.

There are two types of bulldoze messages, *Fbulldoze messages* and *Bbulldoze messages. Fbulldoze messages* are initiated at the start node and propagate along all Fneighbors links. Non-start nodes in the network issue an Fbulldoze message to their Fneighbors only after they receive Fbulldoze message from all of their Bneighbors. Dually, *Bbulldoze messages* are initiated at the finish node and propagate along all Bneighbor links, and non-finish nodes in the network can issue a Bbulldoze message to its Bneighbors only after it receives the Bbulldoze message from both its Fneighbors. The propagation of a bulldoze message forms a wave as illustrated in Figure 2. Bulldoze messages traverse every channel in the network and, by the FIFO assumption on links, force messages already in the network to be cleared along the direction of the bulldoze. A broadcast message does not in general traverse all forward links in the network. Thus the number of messages needed to accomplish a broadcast is less than the number of messages needed to accomplish a bulldoze.

3.2 Obtaining a Consistent GC Snapshot

A GC snapshot consists of acquaintance and active status information that determines a consistent global view of the state of the system at start-of-GC time. Each node records, for each of its actors, its GC-acquaintances, its GC-inverse-acquaintances, and whether or not it was active at start-of-GC time. The GC-acquaintances of an actor are the current acquaintances, plus any acquaintances in messages in the network prior to the start of actual garbage collection. This is a safe approximation of the actors acquaintances, and insures that actors actually forgotten by one actor but sent in messages during GC will not be lost. The GC-inverse-acquaintances of an actor the set of actors having that actor as a GC-acquaintance. This information is used to account for apparently unreachable actors that might communicate their mail addresses to a reachable actor. The GC acquaintance information is used only for GC and can be discarded when the GC for which it was created is complete.

For a global snapshot of the state of the system, we need to guarantee that both *local consistency* and *global consistency* have been achieved. Every node in the system needs a point of reference in time with respect to which it determines the accessibility or inaccessibility of actors in its memory. Once a node has established this point and recorded the necessary information, we have attained *local consistency. Global consistency* is a point in time when all participating nodes have agreed on a particular state of the distributed system.

In order to determine which messages were in the network prior to the start of GC and which entered after, ordinary messages are given *tags* to classify them as *old* or *new* messages. *Old* (resp. *new*) messages are messages which were created prior to (resp. after) the time of the GC snapshot. When GC is initiated, all messages in the network are tagged *old.* During the process of recording the GC snapshot, the network will be cleared of *old* messages by means of the forward and backward bulldoze messages explained above.

To obtain the GC snapshot, first a pre-GC message is broadcast to every node in the system. When a node receives the pre-GC broadcast message, it initializes the GC-acquaintances of each actor residing on that node with (1) its current acquaintances and (2) all acquaintances contained in messages currently residing in

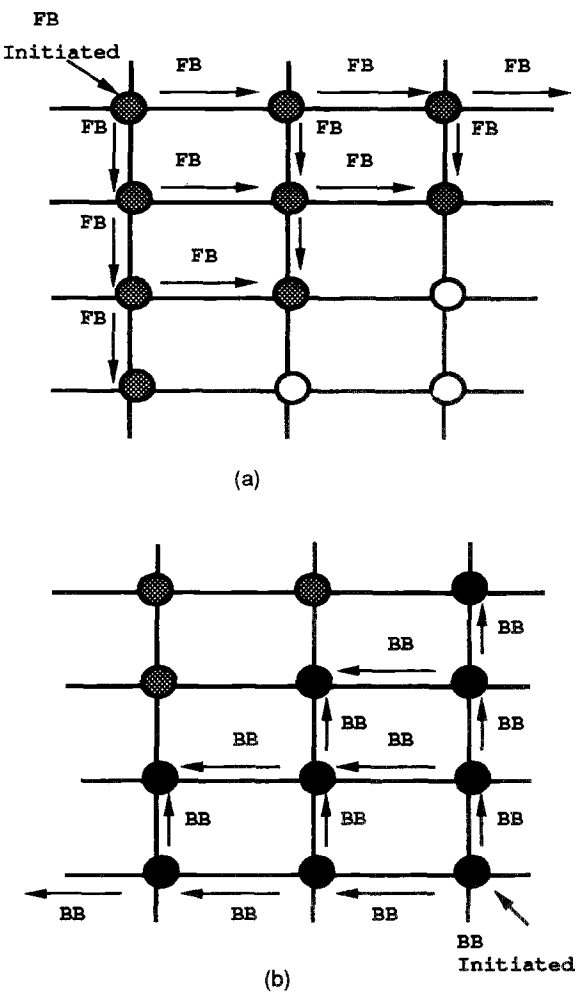


Fig. 2. The Forward and Backward Bulldoze Wavefronts: The figure shows the forward (FB) and backward bulldoze (BB) messages traversing through the network as a wavefront. The FB messages are initiated at the *start* node and travel along Fpaths until they reach the *finish* node. The BB messages are initiated at the *finish* node and travel along Bpaths until they reach the *start* node.

its mail queue. Any acquaintances contained in *old* messages subsequently obtained from the network are added to the GC-acquaintances. It also initializes GC-inverse-acquaintances to be empty. When the pre-GC broadcast is complete, a pre-GC Fbulldoze message is initiated (by the finish-node). When the pre-GC Fbulldoze message passes a node, it marks as active any objects with non-empty mailqueue. The active status of this node is retained for the current GC even though the node may become inactive during GC. Any messages subsequently communicated from that node are be tagged *new*. The *new* tag on a message guarantees the recipient of the message

that any acquaintances communicated in the message have already been accounted for. When the Fbulldoze message reaches the finish node a Bbulldoze message is initiated. When the Bbulldoze message passes a node, this signals that the recording of GC-acquaintances is complete. The node sends *I-know-you* messages from each of its actors to each GC-acquaintance of that actor. When an I-know-you message from actor A to actor B is received then actor A is added to the GC-inverse-acquaintances of actor B. A second forward and backward bulldoze phase is required to clear the network of I-know-you messages. This is initiated by the start node upon completion of the first backward bulldoze wave. When the second forward/backward bulldoze wave is complete, the start node sends a pre-GC-complete message to the root node. At this point, all *old* and *I-know-you* messages in the system have been cleared from the network and the snapshot information is recorded.

The backwards bulldoze messages are needed for both the recording of GC-acquaintances and GC-inverse-acquaintances, since the forward bulldoze only clears forwards links and there may be messages traversing backwards links that need to be recorded. To see this, note that after an object, say A, has received the pre-GC Fbulldoze message it can send only *new* messages. However, it may receive *old* messages from an actor H which has not yet received the pre-GC Fbulldoze message (see Figure 3).

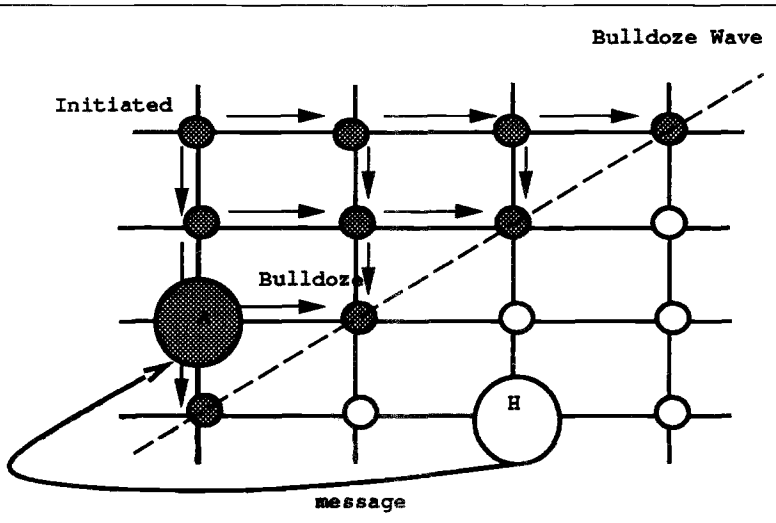


Fig. 3. The Bulldoze Wavefront in Progress: The bulldoze wavefront is halfway through the system. Although object A has started recording acquaintances and issues only *new* messages, it can receive and *old* messages from object H which has not yet received the bulldoze wave.

4 Detection of garbage

In this section we give a definition of reachability that takes into account the ability of an active object to become known by communicating its mail address. We then present an algorithm for marking objects that are reachable according to this definition. We conclude with a description of the behavior of the GC-root actor, which provides an overview of the complete HDGC algorithm.

4.1 Definition of reachability

The definition of reachable objects in an actor-based system is derived from the work of Kafura et al [KWN90]. The *root set* is a pre-defined set of actors from which reachability is traced. It includes actors referenced in the current computation state of the system (environment variables, control structures like stacks etc.). A GC snapshot of the system state determines a conservative approximation of the acquaintance relation. As mentioned in the introduction, in an actor computation, the transitive closure of this relation starting from the root set is not adequate to determine reachability, since an inverse acquaintance of a reachable actor may communicate its mail address at any point of time to its reachable acquaintance, thereby making itself reachable. Thus we cannot ignore the inverse acquaintances in determining reachability.

An actor which is currently processing messages or has messages pending in the network or in its mail queue is an *active actor*, otherwise, it is an *inactive actor*. An inactive actor which is not connected by the transitive closure of the inverse acquaintance relation to an active actor is a *permanently inactive actor*. An actor that is permanently inactive can never communicate its mail address and can be safely regarded as unreachable. The set of *reachable actors* is defined inductively as the least set such that:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- If an actor is reachable, then every inverse acquaintance of that actor which is not permanently inactive is reachable.

A *garbage actor* is an actor which is not reachable according to the above definition.

4.2 Distributed Scavenging

The algorithm for marking the reachable objects in the system, distributed scavenging, follows the inductive definition of reachability. To record the reachable objects, each object of the system has associated with it an *object-status* which may be *touched*, *untouched* or *suspended*. Touched objects are objects which are known to be reachable. Untouched objects have not yet been visited during GC. Objects that remain untouched at the completion of GC are unreachable. Suspended objects are inactive objects that are inverse acquaintances of reachable (touched) objects. If an active inverse acquaintance of such an object is found then the object will become touched. An object that remains suspended at the completion of GC is also

unreachable. When GC is initiated all actors in the system have status untouched. Any actors created after the start of GC on a node are marked as touched.

The marking of objects is accomplished by propagation of GC and GC^{-1} messages from the roots and by backpropagation of GC-ack and GC^{-1}ack messages. It is initiated at the GC-root by sending GC messages to all the root actors. It is complete when GC-acks have been received by the GC-root from all root actors. The process of *touching* the accessible nodes is carried out in accordance with the *Principle of Monotonicity* which states that once an actor has been marked as *touched* during a GC, it cannot subsequently be untouched or suspended during the same GC. Below we summarize the actions caused by receipt of one of the GC marking messages.
A GC message from actor B to actor A is processed as follows:

- if A is touched then a GC-ack message is sent to B from A
- if A is untouched then A becomes touched, and
  - a GC message is sent to each GC acquaintance of A,
  - a GC^{-1} message is sent to each GC inverse acquaintance of A,
  - When GC-ack/GC^{-1}ack messages have been received from all GC acquaintances and GC inverse acquaintances, a GC-ack is sent to B from A.
- if A is suspended then A becomes touched, and
  - a GC message is sent to the GC-acquaintances of A,
  - When GC-ack messages have been received from all GC acquaintances and outstanding GC^{-1}ack messages have been received from GC inverse acquaintances (to GC^{-1} messages sent at suspension time) then a GC-ack is sent to B from A.

A GC^{-1} message from actor B to actor A is processed as follows:

- if A is touched then a GC^{-1}ack is sent to B from A
- if A is untouched then
  - if A is active then A becomes touched, and proceeds as in the GC message case,
  - if A is inactive, then A becomes suspended and sends GC^{-1} messages to its GCinverseacquaintances. When GC^{-1}ack messages have been received from all GC inverse acquaintances, a GC^{-1}ack is sent to B from A.
- if A is suspended then it remains suspended and sends a GC^{-1}ack to B

This basic distributed scavenging algorithm can be adapted to provide a generational version by extending Ungar's Generation Scavenging scheme [Ung84]. A tag field associated with every actor which encodes the generation to which the actor belongs. When a GC is called, the generation bits in the tag field of accessible objects are altered. This is logically equivalent to moving the object from one generation to another. The *copy-count* bits, also a part of the tag field, are used to implement a tenuring policy and are incremented whenever the object survives a GC. When this count reaches a threshold value, the object is tenured from ScavengeSpace to Oldspace.

4.3 Behavior of a GC-root actor

An overall view of HDGC is given by describing the behavior of the GC-root actor. The GC-root actor remembers whether or not a GC is currently in progress. We summarize below the actions of the GC-root actor for each message it can receive.

- GC-initiate: This can come from any node wishing to initiate a GC. If a GC is not in progress, then a pre-GC broadcast is initiated at the start node and the GC-root remembers that a GC is in progress, otherwise the sender is informed that a GC is in progress.
- pre-GC-complete: This is sent by the start node when the second f/b bulldoze wave is complete. The distributed scavenge phase is initiated by sending GC messages to each root actor. When GC-acks have been received from all the root actors, a Local-Clear-Init broadcast is initiated at the start node. Local clearance is begun at each node when this broadcast is received.
- Local-Clear-Complete: This is sent by the finish node when the local clearance is complete. A post-GC broadcast is initiated at the start node.
- GC-complete: This is sent by the finish node when the post-GC broadcast is complete. Now each node marks all messages as old and all remaining actors as untouched [by flipping the interpretation of the tags]. The GC-root now remembers that GC is not in progress and is ready to initialize another GC.

5 Informal sketch of Correctness for HDGC

The correctness of the Hierarchical Distributed Garbage Collection Scheme is expressed by the following four theorems. The first two represent safety properties and the last two represent liveness properties.

Theorem 1. *A non-garbage actor will not be collected by the distributed garbage collection algorithm.*

Theorem 2. *The user program progresses as normal without any semantic interference with the distributed garbage collection algorithm.*

Theorem 3. *The HDGC scheme terminates for every execution.*

Theorem 4. *Every garbage object will eventually be collected.*

To establish these theorems we assume that a GC is initiated only under the following conditions.
Initial Conditions:

- *All actors in the system are untouched*
- *Messages in the system are of one kind - Old messages*

We recall the properties of actors and the underlying network that we have assumed.

1. There are a finite number of actors in the system.

2. Along a single link in the network messages are communicated in a FIFO fashion.
3. Message routing is progress-only in the sense described in section 2.
4. The mutator cooperates with the collector. Any new actors created during GC are created as touched actors, and any new messages created during GC are tagged as new.
5. The mutator does not interfere with the collector. The mutator does not modify data used during GC — the GC-acquaintances and GC-inverse-acquaintances of an actor, an actors active status and other GC status information, or a messages old/new tag.
6. A garbage actor can never become non-garbage.

We have not specified the details of how a node carries out it local clearance but we make certain requirements. Namely, that only untouched or suspended objects on a node are collected, and that local clearance at a node terminates. The correctness theorems follow from the HDGC step lemmas and GC invariant lemmas stated below. A rigorous proof of these lemmas is beyond the scope of this paper and will appear in a forthcoming publication.

5.1 HDGC Step Lemmas

The following lemmas express the crucial properties of each of the steps of the HDGC algorithm. For informal proofs of these lemmas, see [Ven91]. Recall that the *GC snapshot* consists of the GC-acquaintances, GC-inverse-acquaintances, and active status for each actor in the system. This information together with the root set determines a consistent global view of the reachability relation for the purposes of the GC.

Lemma 1. *1. The pre-GC step terminates*
2. At the end of pre-GC, all messages in the network are new and all objects existing prior to initiation of GC are marked untouched.
3. At the end of pre-GC, the GC snapshot is a consistent distributed snapshot of the acquaintance relation relative to the start-of-GC time.

Lemma 2. *1. The Distributed Scavenge phase marks all objects that are reachable according to the GC snapshot as* **touched**.
2. The DistributedScavenge phase marks all objects that are unreachable according to the GC snapshot as **untouched** *or* **suspended**.
3. The Distributed Scavenge phase terminates.

Lemma 3. *The Local-Clear-Initiation terminates and local clearance is initiated on every node in the system.*

Lemma 4. *The Local-Clear step terminates.*

Lemma 5. *The termination of GC is correctly detected and all the nodes in the system are informed of the same.*

Lemma 6. *1. The GC snapshot persists through out the duration of a given GC.*
2. The touching process is monotonic, i.e., once an actor has been marked as touched *during a GC, it cannot subsequently be untouched or suspended during the same GC.*
3. Only one GC can be active in the system at a point in time.

6 Conclusions and future work

In this paper, we have proposed a novel algorithm for garbage collection in scalable distributed systems of active objects called *hierarchical distributed garbage collection.* An informal sketch of the proof of correctness of HDGC has been outlined. A formal proof of correctness will appear in a forthcoming paper. To formalize the proof of a distributed garbage collection algorithm, we formally express the GC process as a transition relation and show that the possible computations of the system satisfy the step lemmas and that these in turn imply the desired correctness properties. The key concept for our formalization is to classify actors into object-level (application) actors and meta-level (system) actors. Meta-level actors can access information about object-level actors that other object-level actors cannot access. In particular, they can modify fields in the data structures representing object-level actors such as status, tags, mailqueue, acquaintances, and behavior. Some meta-level actors simply serve as resource managers for a node. This provides encapsulation of the resource management facilities, and allows us to deal with system management and application management within a single unified framework—the actor model.

Any mechanism for efficient GC in a large system must be conservative. Generational storage management techniques are conservative and they exploit characteristic reference patterns observed in many applications [Ung84]; we therefore believe that they are well-suited to machines with large numbers of processing elements. As we avoid physically moving objects across generations, this scheme also turns out to be less error prone because interprocessor management of forwarding pointers can get very complex and frustrating. In actor based systems, GC involves more than data deallocation. An actor is a basic entity within which behavior (code), communication information and task processing information is embedded. When an actor is deleted, all resource management responsibilities associated with an actor disappear. Memory management in Actors is more than a data management facility, it is a process management facility as well.

What we have avoided in this paper is a detailed discussion of optimizations to the HDGC scheme. A consideration of various deficiencies of the this scheme has revealed some optimizations which can reduce the time and space overheads encountered in synchronization, name translation and bookkeeping. In addition to possible optimizations, this research has also brought to the surface many interesting issues. Compaction of memory to obtain locality, static analysis for optimal actor allocation and placement, lifetime analysis, and extensions of the HDGC algorithm to exhibit fault tolerance and real-time behavior are a few. We believe that the ability to design efficient, scalable, concurrent systems does not lie in esoteric programming paradigms and architectures that are difficult to comprehend. It lies in representing applications as well classified, intuitive specifications and organizing hardware re-

sources to render flexible and manageable concurrency using natural strategies such as *hierarchical resource management.*

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actorrs. In Norman Meyrowitz, editor, *1990 ECOOP/OOPSLA Proceedings*, pages 126–134, Ottawa, Canada, October 1990. ACM Press.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *Nineteenth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 39–50, 1992.

[SGP90] Marc Shapiro, Olivier Gruber, and David Plainfosse. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, November 1990.

[Ung84] David M. Ungar. Generation scavenging - a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. Pittsburgh, PA, April 1984.

[Ven91] Nalini Venkatasubramanian. Hierarchical garbage collection in scalable distributed systems. Master's thesis, University of Illinois, Urbana-Champaign, Dept. of Computer Science, Urbana, IL, forthcoming 1991.

This article was processed using the LaTeX macro package with LLNCS style

Distributed Garbage Collection of Active Objects with No Global Synchronisation

Isabelle Puaut

IRISA / INRIA Campus de Beaulieu
35042 Rennes Cédex FRANCE.
e-mail: puaut@irisa.fr

Abstract. This paper presents an algorithm to perform distributed garbage collection of objects possessing their own thread of control (active objects). The relevance of garbage collection in a system of active objects is briefly discussed. The collector is comprised of a collection of independent local collectors loosely coupled to a global collector. The mutator (application), the local collectors and the global garbage collector run concurrently. Distributed cycles of garbage are detected. The algorithm does not require that the communication channels be reliable: messages may be lost, duplicated, or may arrive out of order. Moreover, local collectors are only loosely synchronised to help detecting global garbage.

1 Introduction

Parallel object-oriented languages and distributed object-based systems have appeared recently as a suitable paradigm for distributed computing [1]. Although not all these languages and systems use garbage collection, we argue that garbage collection is to be preferred to user-controlled memory management for the following reasons:

- programmer-controlled memory management is notoriously error-prone. The programmer tends to make two mistakes. One mistake is that he fails to free a resource when it is no longer used. This leads to performance degradation. The second mistake is that he returns a resource that is still used. Both mistakes are difficult to detect and recover from, especially in systems managing persistent data.
- a better division of responsibility is obtained when the system does what it does best (manage resources), and the programmer does what programmers do best (design systems). The task of programming becomes easier when no longer concerned with memory management. Programs become shorter and thus easier to maintain.
- in distributed applications, it is unlikely that a programmer could design a correct and efficient distributed algorithm for managing distributed data.

Additional reasons appear when using objects possessing their own thread of control (*active* objects). It is significantly more difficult to manage active objects than passive data because both reachability and state must be considered. Furthermore,

as active objects not only consume memory space, but also processing capacity, it is even more imperative that active garbage objects be identified quickly.

Garbage collection in an object-based system raises three distinct problems: distinguishing references from other data in objects, detecting garbage objects and reclaiming the space occupied by these objects. This paper focuses on the second problem.

The remainder of this paper is organised as follows. Section 2 presents briefly garbage collection in a system of active objects and justifies the need for the development of an original distributed garbage collector. Section 3 describes the principles of the algorithm assuming a reliable environment. Section 4 extends this algorithm to allow the parallel execution of the mutator and the collector, and to cope with unreliable communications. Section 5 analyses briefly the performance of the algorithm in terms of messages, time and space overhead. We conclude in Section 6.

2 Garbage Collection of Active Objects: The Problem

2.1 Definition of garbage in a system of active objects

In sequential programming languages with dynamic memory allocation (*i.e.* list processing languages, object-oriented languages), storage can be modelled by a directed graph: a node of the graph is a memory cell and an edge is a reference from one memory cell to another. A memory cell is said to be *garbage* if it cannot be accessed through a path from a distinguished cell (the *root*) leading to that cell (it is not *reachable* from the root cell) [2]. But this definition is not suited to systems managing active objects as discussed below.

In the following we assume that an object is composed of data and of one or several threads of control that operate on that data. The object data is a sequence of memory cells, each containing either an atomic value or a *reference* to another object. An object is *running* when at least one of its threads is executing. It is *inactive* when all of its threads are inactive. An object may activate another object through message passing, if it is running and if its data embeds a reference to the other object. When activating another object, an object may communicate a subset of its data to the activated object.

Note that the computing model described above is quite general and is used in many concurrent object-oriented languages (*e.g.* [3, 4, 5]). Therefore, the proposed garbage collector may be retained for a wide variety of concurrent object-oriented languages and systems.

Garbage collection in systems of active objects was first addressed for the actor computation model [6] and later refined in [7]. Only a brief definition of garbage in a system of active objects is given here; further details can be found in [7].

Informally, an object is garbage if its absence from the system cannot be detected by external observation, excluding from its consumption of memory and processor resources. To make this idea more concrete, *root objects* are introduced, to designate objects that are always needed. The root objects are the objects which have the ability to directly interact with the external world, via I/O devices, external naming, etc. Root objects are assumed to be always running. Intuitively, an object is *garbage* either if it is inactive and cannot be activated in the future, or if it cannot send

information to or receive information from a root object. In other words, an object is garbage if:

- it is not a root object,
- it cannot *potentially* receive a message from a root object,
- it cannot *potentially* send a message to a root object.

In the above definition, the term *potentially* requires further clarification. An object that cannot at a given time directly activate a root object (because it is either inactive or does not possess a reference to a root object) may do so later because it may be activated by another object that gives it a reference to the root object. There exists a set of transformations that change the system of objects from a representation of what can *currently* happen to what can *potentially* happen. Let us consider which objects in Figure 1 are garbage.

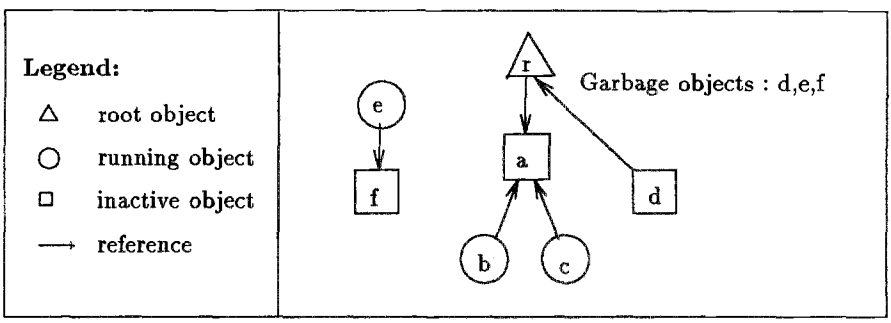


Fig. 1. A system of active objects

Objects e and f are garbage because they cannot potentially communicate with the root object r. Object d is garbage because it cannot be activated. Objects b and c are not garbage because they can communicate to object a their own reference and then can be indirectly activated by the root object r through a later on. Note that the definition of garbage in a system of active objects is actually different from the definition of garbage in sequential systems, which is based only on reachability. Objects b and c are not garbage, although they are not reachable from a root object. Note also that when using the definition of garbage given above, both running and inactive objects may be garbage (both e and f are garbage).

One key property of garbage objects is that they cannot become non-garbage (*stability* property). This is because an object is determined to be garbage only if there is no possibility of communication between it and a root object. Therefore, once an object is garbage, there is no sequence of transformation which could cause it to become non-garbage.

2.2 Why a new distributed garbage collector ?

Numerous garbage collection algorithms have been proposed since the birth of the first programming languages with dynamic memory allocation. Most of them apply only to non-distributed passive objects [2]. Fewer collectors have been developed for distributed systems (see [8, 9, 10] for examples) but the vast majority of them focus on determining object reachability which, as seen before, is too weak a criterion for detecting garbage in a system of active objects. Few algorithms detect distributed active garbage objects [11, 12] but either use a more limited model of computation or enforce global synchronisation.

A distributed garbage collector similar to the one proposed in this paper is the garbage collector described in [12] for a distributed system of actors. Like our garbage collector, this technique relies on independent local garbage collectors and a global garbage collector, both using marking. However, unlike our proposition, the garbage collector of [12] enforces global synchronisation to detect global garbage, and assumes reliable communications.

Another related garbage collector was developed for the EMERALD object-based programming system [3, 13]. While EMERALD provides active objects, the garbage collector designed for this system is based exclusively on object reachability (all running objects are designated as being root objects). Moreover, like the distributed garbage collector described in [12], it enforces global synchronisation to detect global garbage and assumes reliable message transmission.

The distributed garbage detection protocol described in [14], like ours, supports unreliable communication channels. No global mechanism is required to detect global garbage. This protocol only uses information local to each node or exchanged between pairs of nodes. However this protocol, unlike ours, only considers object reachability and does not detect distributed cycles of garbage.

Our algorithm is in some aspects similar to the one described in [15]. Like this algorithm, our global collector is based on (possibly out-of-date) information on inter-node references that permits the elimination of global synchronisation when detecting global garbage. Unlike [15], node crashes and crash recovery are not considered. Only node unavailability is supported. However, in contrast to [15], we detect garbage in a system of active objects and require neither synchronised clocks nor bounded message transmission delay.

3 Basic Principles of the Garbage Collector

3.1 System model

An object is an active entity whose data contains references to other objects. A reference to an object is a unique name that is not reused when the object is deleted. The universe of objects is subdivided into *spaces* (*e.g.* the local memory of a processor, or a disk unit). At any time, an existing object is located in exactly one space. Each space has its own local root object. The global root object is conceptually formed of the union of all local root objects. It is assumed that it is possible to distinguish a *local reference* (to an object in the same space) from a *remote reference* (to an

object in another space). Like many other garbage collectors (*e.g.* [14, 16]), indirection tables are used to distinguish between local and remote references. We also distinguish between *local objects* (referenced only by objects of the same space and having only local references), and *global objects* (having remote references and/or referenced by an object of another space). Local objects are assumed to be much more numerous than global objects. Objects communicate through message passing. The references embedded in a message are distinguished from atomic values. When two objects located in different spaces communicate, messages are sent across the corresponding spaces.

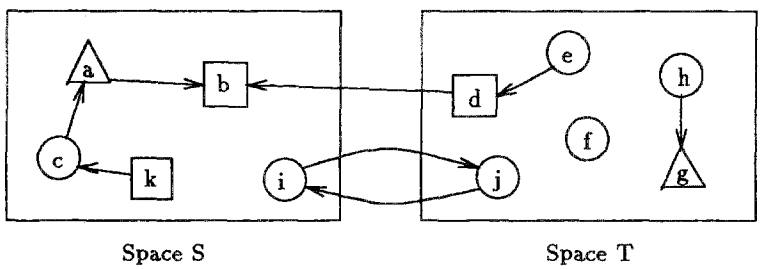


Fig. 2. A distributed system of active objects

An example of a distributed system of active objects is shown in Figure 2. Spaces are denoted with capital letters (S, T), objects with lower-case letters (a, b, c). A message sent from space S to space T is noted $S \longrightarrow T : \{contents\}$. A reference to an object x is noted $@x$. The term *mutator* is used to refer to the overall computation achieved by the objects.

In this section, message transmission is assumed reliable and FIFO (two successive messages from a space S to a space T are received in the order sent). It is also assumed that the mutator is halted during local garbage collection. These assumptions are made to simplify the description of the principles of the garbage collector and are relaxed in section 4.

The garbage collector is comprised of a collection of local garbage collectors (one per space) loosely coupled to a global garbage collector (Figure 3). Local garbage collectors identifies and reclaim objects that can be determined to be garbage by using only local information (*local garbage*). In the example shown in Figure 2, k and f are local garbage. Objects that need inter-space communication to determine if they are garbage are retained by the local garbage collectors. The number of such objects increases while the mutator executes.

The global collector identifies the garbage objects whose detection requires inter-space communication (*global garbage*). Periodically, each local collector running on a space S sends information to the global collector. This information includes the subgraph of S's object graph needed for the identification of global garbage. The global collector records the subgraphs sent by the local collectors and processes this information asynchronously in order to detect global garbage.

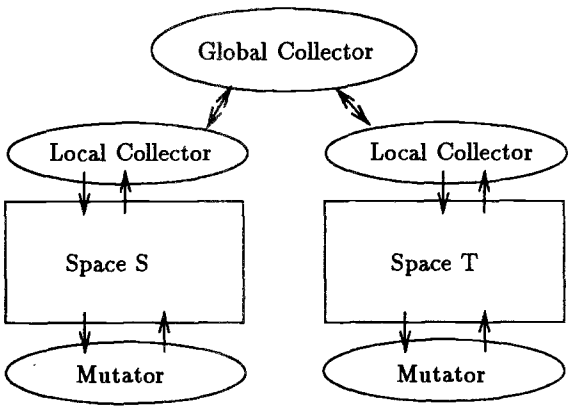


Fig. 3. Architecture of the distributed garbage collector

The local and global collectors are both based on a colouring algorithm, which is first described. The local collectors and the global collector are then presented in turn.

3.2 A simple colouring algorithm to detect garbage active objects

A simple algorithm based on marking is proposed in [7] to detect garbage in a system of active objects. It is assumed that the mutator is halted during the marking process. Three colours are used to mark the objects: objects coloured *white* cannot potentially communicate with a root object; objects coloured *grey* could communicate with a root object if they were activated; objects coloured *black* can potentially communicate with a root object. Initially, all objects are marked white, except for root objects which are marked black. The five rules given in table 1 are applied continuously until no new marking is done.

Table 1. Marking rules

Rule 1 : Mark black all objects referenced by black objects.
Rule 2 : Mark black all running objects having a reference to a black object.
Rule 3 : Mark black all running objects having a reference to a grey object.
Rule 4 : Mark grey all inactive objects having a reference to a black object.
Rule 5 : Mark grey all inactive objects having a reference to a grey object.

The first rule marks black the objects that can directly receive a message from a non-garbage object. Rule 2 marks black the objects that can directly activate a black object. Rule 3 marks black the objects that can directly activate a grey object.

Rule 4 marks grey the inactive objects having a reference to a non-garbage object. Rule 5 marks grey the inactive objects that could (if they were activated) send a message to a grey object.

When no new marking can be done, all black objects are considered to be non-garbage. Grey and white objects are garbage and can be reclaimed. An object is marked at most twice: once grey and once black. The termination of marking follows. Two algorithms implementing the above marking rules are given in [7].

3.3 Local garbage collector

Detection of local garbage

Let us consider an object x located in a space S that is potentially referenced by an object y located in another space. Object x must be retained even if it is inactive because it may be activated by y. A running object having remote references must be retained because it may activate a remote object. An inactive object containing a remote reference must be retained only if it can potentially be activated.

The marking rules described above are used to detect the local garbage of space S. Initially, the following objects of space S are coloured black: the root object of space S, remotely referenced objects and running objects having remote references. The inactive objects containing remote references are coloured grey. All the other objects of space S are coloured white. When marking is complete, all white and grey objects are garbage and can be reclaimed, without any synchronisation with the other spaces.

Table 2. Local garbage collection

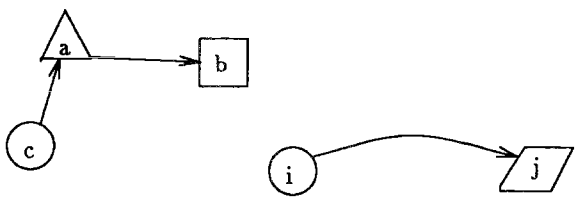
		Space S	Space T
Initialisation:	black	{a,b,i}	{j,g}
	grey	{}	{d}
	white	{c,k}	{h,e,f}
End of Local Collection:	black	{a,b,c,i}	{d,e,g,h,j}
	grey	{k}	{}
	white	{}	{f}
Local Garbage		{k}	{f}

The progress of local collection in space S of Figure 2 is shown in table 2. At the end of the local marking, objects k and f are identified as garbage and can be reclaimed.

Cooperation to the detection of global garbage

Periodically, a local collector running on a space S identifies the objects that are needed for the detection of global garbage. The references contained in these objects are then sent to the global collector that processes them asynchronously. The objects needed for detecting global garbage are the objects affected by remote references, that is objects that can potentially communicate with remote objects (they may be local or global to space S).

The objects needed for the detection of global garbage are identified by applying the marking rules given in section 3.2, in which colours are renamed. Initially, the running objects containing remote references and the remotely referenced objects are marked REMOTE (analogous to black). The inactive objects containing remote references are marked POSSIBLYREMOTE (analogous to grey). The other objects are marked LOCAL (analogous to white). When marking is finished, REMOTE objects are used to detect global garbage (objects a, b, c and i for the space S of Figure 2). All the references contained in these objects form a subgraph of S's graph of objects (shown below for the space S of the example).



The subgraph is then sent to the global collector as a list of edges. In our example, we get the list $(< a, root >< b, inactive >)$. $(< c, running >< a, root >).(< i, running >< j, unknown >)$. As the state of object j is not known by space S, it is sent as *unknown* to the global collector. The state of j will be known by the global collector when merging the informations sent by the local collectors.

3.4 Global garbage collector

The global collector is a logically centralized service that maintains a graph G which is the merge of the subgraphs sent by the local collectors. Since local collectors do not synchronise with each other when sending information to the global collector, the global collector must be able to detect whether G represents a consistent vision of the system state. This issue is examined before giving a more detailed description of the global collector.

Consistent global states and garbage collection

Let us consider a distributed system composed of n processes p_i, $1 \leq i \leq n$ communicating through message passing on reliable communication channels c_{ij}, $1 \leq i, j \leq n$, where c_{ij} denotes the communication channel between p_i and p_j.

Message transmission is assumed to be finite, but not necessarily bounded. Each process p_i has a private local state. The state of any communication channel is the set of messages sent by process p_i and not yet received by process p_j. The execution of a process consists of a sequence of *events*. The events are classified according to three categories: *send*, *receive*, and *internal* where internal events modify only the process local state. A global system state if comprised of the processes local states and the communication channels states. A global state is qualified as being *consistent* (or is called *global snapshot* [17, 18]) if for each message captured as received in a process local state, the message is captured as sent in the sender local state. An interesting feature of global snapshots is that they can be used to detect stable properties [17].

Consistency of a global system state may be determined through the use of *vector timestamps* [18, 19] that timestamp events occurring in a distributed system. Each process p_i has a clock VT_i consisting of a vector of length n, where n is the number of processes. With each event of process p_i, VT_i "ticks" by incrementing its own component of its clock, $VT_i[i]$. Ticking is considered to occur before any event; the timestamp of an event is the clock value after ticking. Each message gets a piggy-backed timestamp consisting of the sender clock. The receiver p_i of the timestamped message updates its clock with the componentwise maximum of its clock and the timestamp contained in the message, that is, $VT_i := sup(VT_i, t)$, where t is the timestamp of the message and $sup(C, C') = [max(C[1], C'[1]), ...max(C[n], C'[n])]$. Figure 4 shows an example of events timestamping using vector timestamps (arrows denote message transmission).

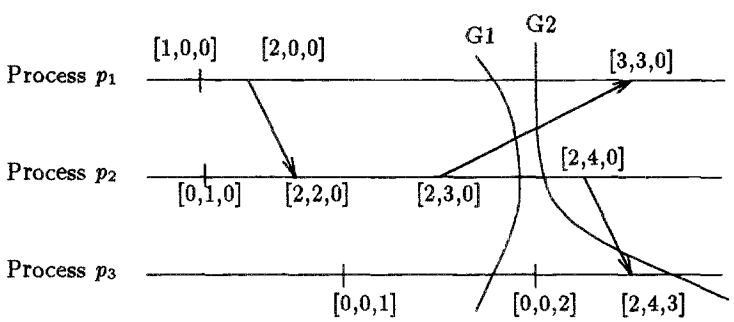


Fig. 4. Vector timestamps

Assuming that VT_i corresponds to the clock of process p_i timestamping its local state ls_i, a global state is consistent [18] if:

$$\forall i \, \forall j \quad VT_i[i] \geq VT_j[i]$$

For instance, in Figure 4, the system state in $G1$ is a consistent state, while the system state in $G2$ is not consistent ($VT_3[2] > VT_2[2]$): the message sent from p_2 to p_3 is captured as received in p_3 state and not yet sent in p_2 state.

Vector timestamps can be used to detect global garbage due to the stability property of garbage: once an object is garbage, it remains garbage forever.

The protocol

Events are timestamped using vector timestamps. Each space S maintains a vector timestamp VT_S. The information sent by the local collector of space S to the global collector is timestamped with the clock VT_S corresponding to the construction of this information. Note that the information sent to the global collector must contain the states of the communication channels in order to obtain a consistent state. This is described in the following paragraph.

The local collectors and the global collector communicate through asynchronous message passing. In this way, the local collectors are not blocked while global garbage is detected. Two types of messages are used (see table below): *Info* is sent by a local collector to the global collector, and contains the information needed to detect global garbage; *Delete* is sent by the global collector to a local collector to notify that some objects are garbage.

Global Collector

Maintains a global graph G composed of objects needed for the detection of global garbage. G is composed of the subgraphs sent by the local collectors and the clocks at which these subgraphs were computed

→ **Info**(S,Edges,Trans,VT_S)

Receipt of information from the space S; it includes a list of references (*Edges*), a list of references that are possibly in-transit (*Trans*) and the timestamp at which this information was computed.

← **Delete**(S,list_of_obj)

Sending of a message notifying S that the objects belonging to *list_of_obj* are garbage.

The global collector represents G as a set of edges labelled with the space that sent them. An edge is a tuple $(< id_1, state_1 >< id_2, state_2 >)$, where id_1 and id_2 are names of objects and $state_1$ and $state_2$ are states of objects (*i.e. running, inactive* or *root*). The global collector also stores the timestamps of the last information sent by the local collectors $(VT_1, .. , VT_n)$.

Upon receipt of an *Info*(S_i,*Edges,Trans*,VT_S) message, the global collector replaces the edges of G that are labelled with S_i by the edges contained in *Trans* and *Edges*. The timestamp VT_i of the old subgraph sent by S_i is replaced by VT_S. The global collector then checks if G is consistent (*i.e.* $\forall i \, \forall j \;\; VT_i[i] \geq VT_j[i]$). If G is not consistent, nothing is done. Otherwise, the global collector traces G using the marking algorithm described in paragraph 3.2. Initially, all objects whose state is *root* are marked black. The marking rules are applied until no new marking is done. When marking is finished, all the objects whose colour is white or grey are garbage. All white or grey objects labelled with space S are gathered in a list l. The message

Delete(S,l) is then sent to space S.

Recording the state of the communication channels

The global collector needs to know the states of the communication channels in order to establish a consistent state. The only informations relevant to garbage collection are the references contained in in-transit messages, hence this is the only information sent to the global collector. Note that a reference $@x$ contained in a message may be considered in-transit although received without incorrect behaviour of the global collector. Indeed, this implies only that G has an extra-edge containing $@x$ which results in delaying the detection of x as garbage. Consequently, each message need not be acknowledged. Each time a message is sent, the references contained in the message are stored; it only remains to free the storage needed to store these references.

Assuming FIFO channels, a message $m : S \longrightarrow T : \{..., @x, ...\}$ is received when T has acknowledged a message sent after m; the space needed to keep track of $@x$ can then be freed. In addition to the vector VT_S, used to timestamp the information sent to the global collector, each space S has a vector of clocks Hts_S (for *highest*), used to free the memory occupied by possibly in-transit references. $Hts_S[T]$ indicates that the last message received by S from T was sent at T's time $Hts_S[T]$. The following actions must be executed when sending and receiving messages:

- *Sending of* $m : S \longrightarrow T : \{..., @x, ...\}$
  (where m corresponds to the sending of a message from an object y to an object z)
  - increment VT_S,
  - store $trans = < z, x, T, VT_S[S] >$, that indicates that a reference from z to x is in transit and was sent to space T at S's time $VT_S[S]$,
  - send m to space T together with the vector clock of the sender and the sending clock of the last message from T to S:
    $m = S \longrightarrow T\{..., @x, ...VT_S, Hts_S[T]\}$.
- *Receipt by S of a message from* $T < ..., VT_T, Hts_T[S] >$.
  - update VT_S with VT_T,
  - update $Hts_S[T]$ with $VT_T[T]$,
  - delete all $trans = < x, y, T, VT_S[S] >$ such as $VT_S[S] \leq Hts_T[S]$.

When S sends information to the global collector, the list of possibly in-transit references are included in this information.

Guaranteeing global garbage is eventually deleted

While not formally proved, the global collector does not detect an object as garbage although it is not, because the global collector processes a global snapshot of the system. We wish to show there that all global garbage is eventually deleted. Three properties are needed to ensure that progress is made:

(Dyn1) A local collector sends new information to the global collector in finite time

This property can be ensured by performing the detection of local garbage at regular finite intervals, and sending information to the global collector after a fixed number of local garbage collections.

(Dyn2) In-transit references are known to be received in finite time

The technique used to keep track of the states of the communication channels, although it does not need extra communications, does not satisfy this property. Indeed, if a space S stops sending messages to space T after the receipt of a message m from T, T will never know m was received. In order to satisfy *(Dyn2)*, each space S must send to another space T at least one message in a finite time interval.

(Dyn3) A consistent state is detected in finite time

This property is not directly satisfied by the proposed protocol. Indeed, the global collector may detect a consistent state only after a very long delay (this delay may also be infinite). A practical approach to attempt reducing this delay, when the spaces physical clocks are loosely synchronised, is to send information to the global collector at predetermined physical times. The spaces are in this case loosely synchronised to send information to the global collector. However, although this technique may seem realistic from a practical point of view, it still does not ensure that a consistent state is obtained in finite time.

A *panic mode* of the global collector, less efficient that the *normal mode* given in the previous paragraphs is defined in order to satisfy *(Dyn3)*. In panic mode, all the spaces must synchronise with each other for obtaining a consistent state. The protocol given in [18] can be used for that purpose. Panic mode and normal mode can be combined into a judicious mixture depending on how short of free store the system is. We define a *nervous mode*, in which each local collector has a *panic threshold* PT. PT is the maximum allowable number of messages containing information sent to the global collector before a consistent state is obtained. When the number of messages sent to the global collector exceeds PT without obtaining a consistent state, the system enters the panic mode. A computation of global snapshot is initiated. In this way, the system can balance garbage collection costs against the urgency of its need for storage.

4 Extensions of the Basic Algorithm

4.1 Concurrency between mutator and collector

Halting the mutator while garbage objects are detected leads to unpredictable interludes in the computation. Such interludes are annoying for users running interactive computations and are unacceptable for applications having real-time requirements. The technique described in [12] can be used to execute concurrently the mutator and the local collectors: the mutator participates to the marking of objects when modifying the object graph. We focus here on allowing the local and global collectors to proceed in parallel, with as few synchronisation as possible.

The global and local collectors must cooperate for the deletion of global garbage. Indeed, the global objects contained in a space are shared between the local collector

of this space and the global collector: it would be incorrect to delete a global object while it is used by the local collector, *e.g.* during local marking. Deletion of a global object is achieved by the local collector of the space containing this object. Upon receipt of a *Delete(S,x)* request, x is marked as "deleted". The actual deletion of x is done by the local collector during its next reclamation cycle.

The marking of objects for the detection of local garbage and the marking of objects for the detection of global garbage are independent. Two distinct *colour* fields can be used to achieve these two marking cycles in parallel. The same *colour* field can however be used if these two marking cycles are done sequentially.

4.2 Unreliable communications

Until now, we have considered reliable and FIFO communication channels between spaces. In true distributed systems, additional features must be taken into account. A message may be lost, duplicated or arrive our of order. Byzantine failures are ruled out: message contents are not altered during transmission. Delivered messages arrive in finite (but not necessarily bounded) time. When considering that messages may be lost, it is assumed that transmission of sufficiently many messages will eventually cause at least one to be received.

Two kinds of messages exist in our system: mutator messages and collector messages, the latter being used for the detection of global garbage (*Info* and *Delete* messages). It is assumed that the mutator knows how to deal with unreliable channels (*e.g.* by sending again lost messages and removing duplicates). Our algorithm tolerates message loss, duplication and non-FIFO ordering independently of the solutions adopted by the mutator.

Message loss

Assuming that the local collectors send information periodically to the global collector, the loss of a message *Info* will only cause a delay in the detection of global garbage. The loss of a message *Delete(S,x)* is also tolerated since garbage objects remain garbage: the global collector will still detect x as garbage during its next marking phase.

The loss of a mutator message will not cause the incorrect deletion of objects. The only objects that could be incorrectly identified as garbage are the objects whose reference is contained in the lost message. All references contained in a message m sent from S to T are considered to be in transit until T has acknowledged a message sent by S after m (m is either received or lost). Thus, no object is incorrectly identified as garbage. If m contains the last reference to object x, x will eventually be detected as garbage (as soon as S will detect that m is either received or lost).

Non-FIFO ordering

Our use of timestamps to guard against possibly in-transit references works well if channels are FIFO, *i.e.* if messages are received in the order sent (if at all). With a small extension, our algorithm can tolerate some amount of non-FIFO ordering. Non-FIFO orderings are acceptable if the following acceptance condition is added for receiving mutator messages $m = S \longrightarrow T : \{...\}$:

$$Hts_T[S] > VT_S[S]$$

A message m sent from S to T is rejected, *i.e.* considered as being lost, if it arrives after a message sent from S to T after m (all late mutator messages are considered lost). Late *Info* messages received by the global collector are also eliminated, because they carry out-of-date information. An *Info* message sent by a space S is accepted if the following acceptance condition is verified:

$$VT < VT_S,$$

where VT_S is the timestamp of the last information sent by S. Non-FIFO ordering of *Delete(S,x)* messages are harmless since garbage objects remain garbage.

Duplicated messages

The duplication of an *Info* message is already taken into account by the acceptance condition of messages on the global collector, given in the previous paragraph. The duplication of a *Delete(S,x)* message is treated by making the action executed when this message is received idempotent: x is marked "deleted" if it still exists and is not already marked "deleted". Since object identifiers are not reused, there can be no confusion of object identifiers. If object names were reused, stronger assumptions, like bounded transmission delay would have to be done.

4.3 More availability

The global collector stores only (possibly out-of-date) information sent by the local collectors. It can therefore be replicated without any problem. A local collector sends information to a single replica of the global collector (the global collector is seen by its user as a centralized service). Information is then propagated to all other replicas in the background. This permits garbage to be collected even if some of the replicas are unavailable and removes the bottleneck of a centralized global collector.

5 Performance evaluation

Cost of the proposed garbage collection algorithm is briefly considered according to three different measures: in terms of messages, memory space, and computation time.

Messages

The only additional foreground messages are those needed to detect global garbage. Assuming messages have unbounded size, if n is the number of spaces in the system, only $n + 1$ messages are required to detect an object x as garbage (n *Info* messages and one *Delete(S,x)* message). However, although the number of sent messages is low, there may be a long delay between the time an object becomes garbage and the time it is effectively detected as garbage. However, all garbage

is eventually detected. In real systems, where messages have bounded size, more than one message may be sent in order to communicate information to the global collector. The number of edges, and thus the number of messages actually sent strongly depends on the percentage of global objects in a space.

Memory space

Four or two bits per object are required for marking, depending upon whether local marking and marking the objects used for the detection of global garbage are done in parallel or not.

Two clock vectors per space S are required : VT_S and Hts_S. In addition, the global objects must be identified. This is usually done by using two tables: one for remotely referenced objects and the other for objects containing remote references.

The global collector stores a set of edges necessary to detect global garbage. The memory space occupied by this set is strongly dependant on the degree of locality exhibited by the mutator. If n is the number of objects of the global graphe G, $2 * n$ bits are required for marking these objects. The set of edges can for example be represented as a list or a matrix depending on the structure of G. Assuming G is represented as a list of edges, each list element contains two object references. Experimental results on the structure of G will help knowing the best representation for G and the space occupied by this representation.

Finally, each message must be timestamped; however, such a requirement is already common in distributed systems.

Computation time

Mainly three factors have to be considered : local marking, that is used to detect local garbage, the collection of information by each space for the detection of global garbage and global marking, that actually detects global garbage. We use the algorithm described in [7] for marking. This algorithm has a time complexity of $O(n^2)$, where n is the number of objects.

6 Conclusions

In this paper we have proposed an algorithm to perform distributed garbage collection of active objects. Autonomous local garbage collectors detect and reclaim local garbage, without synchronising with each other. Global garbage (even forming distributed cycles) is identified asynchronously by a logically centralized global garbage collector. The computation is not halted while detecting garbage objects and only weak synchronisation is required between the local collectors. This weak synchronisation is paid by a delay in the detection of global garbage. The algorithm is based on weak assumptions on the communication channels: messages may be lost, duplicated or arrive out of order. However, there are several limitations of the collector described in this paper. These limitations, and future work needed for removing them, are discussed below.

First, some quantitative information on objects can improve the performance of the garbage collector. In particular, the following knowledge would be useful:

- the percentage of objects needed to detect global garbage,
- the variations of the lifetimes of objects,
- the percentage of cyclic structures.

The first point is of prime necessity because it allows to choose the frequency of global garbage detection. It also permits to know how much information is sent to the global collector. If too much information is sent, it would be interesting to study if an additional conservative technique for detecting global garbage (like the one described in [14]) could lower the overall cost of garbage collection. Having an idea of the lifetimes of objects would help knowing if a *generational* method [20] is adequate for the collection of garbage in a system of active objects. The percentage of cyclic structures, and more particularly of distributed cycles would show whether detecting distributed cycles is of prime importance or not. An implementation of the algorithm, currently under way in the GOTHIC distributed object-based system [21] will help tuning the algorithm.

Object migration was not considered here: its influence on the proposed garbage collector has to be taken into account. Finally, the algorithm presented in this paper has not been proved correct. This is an area which needs further investigation.

Acknowledgements

Thanks to Valérie Issarny, Michel Banâtre and Ciarán Bryce for valuable comments on earlier drafts of this paper.

References

1. Roger S. Chin and Samuel S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.
2. J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.
3. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.
4. M. Benveniste and V. Issarny. ARCHE : un langage parallèle à objets. Research report 642, IRISA, March 1992.
5. A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press Series in Computer Systems, 1987.
6. G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
7. D. Kafura, D. M. Washabaugh, and J. Nelson. Garbage collection of actors. In *Proc. of the 1990 ECOOP/OOPSLA Conference*, pages 126–133, 1990.
8. P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proc. of Conf. PARLE*, volume 259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, 1987. Springer Verlag.
9. L. Augusteijn. Garbage collection in a distributed environment. In *Proc. of Conf. PARLE*, volume 259 of *Lecture Notes in Computer Science*, pages 75–93, Eindhoven, 1987. Springer Verlag.

10. M. Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. In *Proc. of 1989 OOPSLA Conference*, pages 37–48, October 1989.
11. P. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proc. of the ACM Conf. on LISP and Functional Programming*, pages 168–178, 1982.
12. D. M. Washabaugh and D. Kafura. Distributed garbage collection of active objects. In *icdcs11*, pages 369–376, May 1991.
13. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
14. M. Shapiro, D. Plainfossé, and O. Gruber. A garbage detection protocol for a realistic distributed object-support system. Research report 1320, INRIA, November 1990.
15. B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proc. of 5th International Symposium on the Principles of Distributed Computing*, pages 29–39, Alberta, Canada, August 1986.
16. J. Hughes. A distributed garbage collection algorithm. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272, Nancy (France), September 1985. Springer Verlag.
17. K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
18. F. Mattern. Virtual time and global states in distributed systems. In *Proc. Int. Conf. on Parallel and Distributed Algorithms*, pages 215–226. North-Holland Publishing, 1988.
19. C.J. Fidge. Timestamps in message-passing systems that preserves the partial ordering. In *Proc. 11th Australian Comp. Conf.*, February 1988.
20. H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.
21. Michel Banâtre, Yasmina Belhamissi, and Isabelle Puaut. Some features of gothic: a distributed object-based system. In *1992 International Workshop on Object-Orientation in Operating Systems (I-WOOOS '92)*, Paris, France, September 1992.

Memory management for parallel tasks in shared memory

K.G. Langendoen H.L. Muller W.G. Vree
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
koen@fwi.uva.nl

Abstract. Three memory management strategies for shared-memory multiprocessors that support two-space copying garbage collection are presented. These strategies exploit the fork-join task structure of the divide-and-conquer paradigm by equipping each task with a private heap that can be locally collected independently of other processors and tasks. The memory management strategies use a virtual address space to allocate private heaps such that the efficient copying collectors do not need to be adapted to handle physically scattered heaps. When the allocation strategies run out of the virtual address space, an expensive compaction operation has to be performed. Results from a detailed simulation, however, show that this happens so infrequently that the costs are negligible in practice.

1 Introduction

An important property of logic, object-oriented, functional, and other high-level programming languages is their automatic management of dynamically allocated storage. The language support system provides the user with a virtually unlimited amount of storage by running a garbage collector to reclaim storage that is no longer in use. The efficiency of the garbage collector is important for the application's performance, especially when the underlying computational model (e.g., graph reduction) allocates lots of temporary storage.

From the three classes of garbage collection algorithms (reference counting, mark&scan, and copying collectors), the copying collectors perform best on systems with large memories [Har90] for two reasons. First, they only traverse live data, which usually accounts for only a small fraction of the total heap space, while mark&scan collectors access every heap cell twice. Secondly, copying collectors compact the live data into one consecutive block, which facilitates the fast allocation of (variable sized) nodes by advancing the free pointer instead of manipulating a linked list of free cells and managing the reference counts.

Now that shared-memory multiprocessors are widely in use, it is important to develop runtime support systems that efficiently manage the storage allocated by parallel programs. This raises the question of how to adapt the efficient sequential copying collectors to run on such parallel machines, while making the best use of the available hardware support.

Existing copying garbage collectors that support general purpose parallel applications on shared-memory multiprocessors collect the complete heap in shared memory at once. As a consequence, global synchronisation is needed to control the garbage collections (see Section 2) and half of the shared heap is reserved for to-space. For the class of (fork-join) task parallelism, however, it is possible to provide each task with a private heap, which can be locally collected independently of other processors and tasks; the memory manager exploits the properties of

fork-join parallelism: tasks only communicate when creating new tasks or returning results, while parent tasks wait for the results of all their children before resuming execution.

The scattered heaps of join tasks complicate the pointer classification of the local collector when copying live data to reclaim garbage space. The memory management strategies, as described in Section 3, handle this problem by allocating storage blocks in a virtual address space. A set of parallel benchmark programs (Section 4) is used to evaluate the basic scheme and two improvements on a multiprocessor simulator.

2 Copying garbage collection for multiprocessors

Cheney's two-space copying collection algorithm as described in [Che70] is the basis of many (parallel) copying garbage collectors. The available heap space is divided into two equal parts: the from-space and the to-space. During normal computation new nodes are allocated in from-space by advancing the free-space pointer through the from-space. When the heap space in the from-space has been consumed, all live nodes are evacuated (i.e. copied) to the empty to-space by the garbage collector.

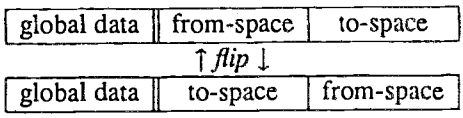


Figure 1: Memory layout for two-space collector

The evacuation starts with copying the nodes in from-space that are referenced by root pointers in the global data area, which contains for example the call stack. Then the nodes in to-space are scanned for pointers to objects in from-space that still have to be copied. This process is repeated until the to-space contains no more references to nodes in from-space. The strict separation of global data and the heap allows the collector to efficiently detect with one compare instruction whether a pointer refers to a node in from-space or not. After evacuating all live nodes, the roles of the two semi spaces are *flipped*, and ordinary computation is resumed.

A straightforward adaptation of a copying collector to run on a multiprocessor is to let all processors participate in a global evacuation operation: processors allocate large blocks of storage in the shared global heap, and if one processor detects the exhaustion of the (global) from-space, it synchronises with the other processors to start garbage collection. The evacuation of live nodes proceeds with all processors scanning parts (pages) of the to-space in parallel. To handle possibly shared data objects, processors lock each individual node in from-space when inspecting its status and, if necessary, copying it to to-space. This method is, for example, used in MultiLisp [Hal84] and GAML [Mar91].

To reduce the locking overhead of the above method, the Parlog implementation described in [Cra88] partitions the heap among the processors, so that each processor can collect its own part of the heap. Whenever a processor handles a *remote* pointer to a live node in another part of the heap, it places a reference to the pointer in the corresponding processor's Indirect Pointer Stack (IPS). After a plain evacuation operation, each processor scans its IPS buffer, which contains (new) roots into its private heap, updates the pointers to point to copies in to-space, and continues with scanning the new objects in to-space. Now only the accesses to the IPSes have to be guarded with locks instead of each heap object.

A rather different approach to use copying collectors on parallel multiprocessors is described in [AEL88]: one processor reclaims all the garbage, while the others proceed with their normal computational work. The synchronisation between the collector and the other processors (mutators) is accomplished through standard hardware for virtual memory. When the evacuation of live nodes starts, the collector copies all root nodes to the to-space, and marks the virtual memory pages of the to-space as inaccessible to the mutators. Then the mutators immediately resume execution in the to-space, while the collector scans the to-space page by page for references to nodes in from-space that still have to be evacuated. Whenever the collector has finished a page of the to-space, it makes that page accessible to the mutators. If a mutator tries to access an object in a not-yet-scanned page in to-space, the hardware generates an access violation trap. This triggers the collector to handle the referenced page immediately, whereafter the mutator resumes execution.

The common disadvantages of the above copying garbage collection algorithms for multiprocessors are that they waste half of the shared heap, which is reserved for the to-space, and that they require global synchronisation operations. The inherently global nature of these algorithms also raises efficiency problems when scaling to large (hierarchical) shared-memory multiprocessors: the single virtual memory collector can not keep up with many mutators, while the parallel scan of the other algorithms overloads the memory bandwidth.

3 Local copying garbage collection

A possible scheme for copying garbage collectors on shared-memory multiprocessors that has not been explored before is to provide each parallel process with its own heap and perform garbage collection per process locally. This approach is attractive since it avoids global synchronisation and cooperation of processors, while the reserved amount of to-space can be reduced by limiting the maximum heap size of a process and time-sharing a common to-space. Collecting a process, however, requires access to all global root pointers into the local heap. This makes the scheme unattractive for general parallel processes that can exchange arbitrary data, in particular heap pointers, since recording the roots from outside is a space and compute intensive task. Instead we restrict ourselves to a task model with limited communication capabilities: the divide-and-conquer model, also known as the fork-join model.

The divide-and-conquer paradigm is an important method to structure parallel applications and has been extensively studied, see for example [Vre89]. It (hierarchically) decomposes a problem into independent subproblems, solves those subproblems in parallel, and combines the results into the solution of the original problem. The divide-and-conquer paradigm is applicable to a wide range of applications and can be implemented efficiently on most parallel machine architectures since divide-and-conquer applications usually generate a controlled number of coarse-grain tasks with a restricted communication pattern: only at the begin and end of a task, data has to be exchanged.

The fork-join task structure of divide-and-conquer parallelism allows efficient incorporation of the above local copying collector scheme in a shared-memory multiprocessor. At runtime a divide-and-conquer application (recursively) unfolds into a tree shaped task structure, see Figure 2a. Each task is provided with a "private" part of the shared heap where it allocates storage during its execution. Interior tasks (1, 2, and 3) are suspended during the execution of their child tasks, so only leaf tasks (4, 5, 6, and 7) can reclaim their garbage locally; in Section 5

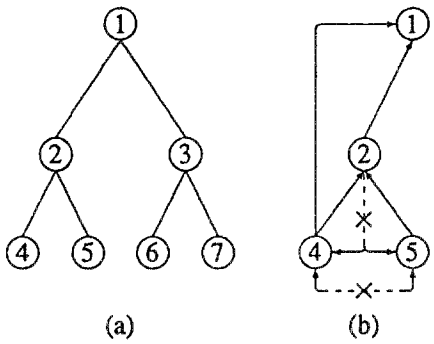


Figure 2: fork-join tree (a) with limited inter-task pointers (b).

we discuss how this restriction can be relaxed.

The garbage collection of a leaf task with a two-space copying collector requires the allocation of a contiguous to-space and access to all *root* pointers into the private heap. The latter requirement is hard to fulfil in general, but the divide-and-conquer model causes the leaf tasks to execute without any external interaction, hence, a leaf task can not pass a pointer to any other active task; there are no pointers between tasks 4 and 5 in Figure 2b. The absence of communication between leaf tasks, however, does not rule out data sharing since tasks can execute different subproblems that contain pointers to shared data in common ancestor heaps. For example, tasks 4 and 5 can share data that resides in the heap of task 2, or even in task 1. To prevent tasks from passing pointers through their ancestor's heap, we require that shared data is read-only. As a consequence, pointers from interior tasks to leaf tasks do not exist; for example, there are no pointers from task 2 to either task 4 or task 5 as shown in Figure 2b. Some programming languages already meet this "read-only" requirement. Others can satisfy it by preprocessing the parents' data before forking the child jobs. In [LV91b], for example, an adaptation for lazy functional languages is discussed.

Since the divide-and-conquer paradigm limits the inter-task pointers to references to ancestor data, there are no "external" root pointers into the heap of a leaf task. This allows the garbage of a leaf task to be reclaimed with a local sequential copying collector, which only scans the task's call stack for root pointers. Note the resemblance with generation scavenging garbage collectors [LH83] where often the youngest generation (cf leaf tasks) is collected, but not the older generations (cf interior tasks).

We would like to use the sequential copying collector for interior tasks too. When an interior task resumes execution it becomes a leaf task again since all its offspring has already ended their execution. By having the child tasks link their heap to the parent's heap when returning their result, the parent can also reclaim its garbage locally provided that it can handle a scattered heap. The scattered heap can not be avoided by reserving space in the parent heap in advance since the size of a result is unknown when creating a child task and results can become arbitrarily large.

3.1 Scattered heaps

Collecting a scattered heap is not straightforward if the private heaps of tasks are arbitrarily allocated in the multiprocessor's shared memory because then the linked-up heaps of interior tasks may interleave. This complicates the evacuation of live nodes since it is no longer possible

to distinguish pointers to objects in from-space and pointers to global (ancestor) data with a single compare instruction. For example, suppose the fork-join tree of Figure 2a has been laid out in memory as shown in Figure 3a. After leaf tasks 4 and 5 have terminated and linked their heap to the parent task, task 2 resumes execution and the storage configuration changes to 3(b); the heap of task 2 is no longer contiguous.

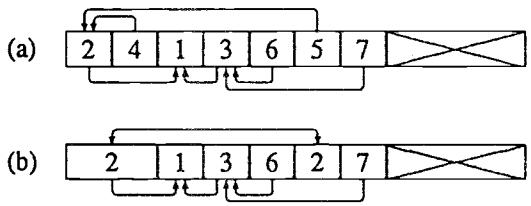


Figure 3: Storage layout with inter-task pointers.

When task 2 runs out of free space, it allocates a to-space at the right of task 7 and starts evacuating the live nodes. The search for pointers to live nodes in the heap of task 2 is complicated by the presence of heap 1, which breaks the simple memory layout of Figure 1 where global data and the from-space each have a contiguous address space. Note that task 2's internal pointers from the right part to the left part or vice versa must be distinguished from the inter-task pointers to 1. In principle the problem of distinguishing global and local data can be solved by means of a lookup table that records the owner of each storage block, but this would degrade performance because of extra memory references and table management overhead. Instead we will use a virtual address space to allocate storage such that task heaps never interleave with ancestor heaps.

3.2 The Basic Allocation Scheme, BAS

To support efficient evacuation of live data in scattered parent heaps, it is sufficient to enforce that a task's private heap is allocated to the right of all its *ancestor* heaps. This causes a strict separation of the task's (scattered) private data and its global ancestor data, so pointers can be classified with one instruction as in the sequential case. The basic allocation scheme (BAS) accomplishes the strict separation by always allocating a new heap at the right of the most recently allocated one. Virtual memory hardware is used to relocate the released physical space of the from-space to the right end after a garbage collect.

The basic scheme results in a window of physical memory moving from left to right through the virtual address space, see the example in Figure 4. When task 2 resumes execution in 4(d), its scattered heap encloses the heap of task 3, but this has no effect on the garbage collector since task 2 to does not refer to data of task 3; it only refers to data of task 1.

The window with available physical memory (W) has to be at least as large as the size of the largest private heap since tasks allocate their to-space in the window when collecting garbage. By limiting the maximum task size, we significantly lower the 50% waste of memory reserved for to-space of the (sequential) copying collectors since tasks can time-share W as a common to-space. The costs of this limit are that large tasks have to collect their garbage more often. Note that we can control this space-time trade-off by adjusting the value of the maximum task size. It suffices to reserve a $1/(p+1)$ fraction of the total memory size on a multiprocessor with

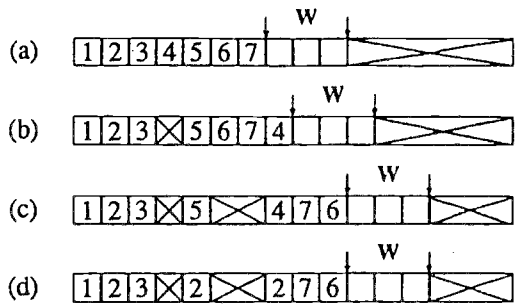


Figure 4: BAS: (a) initial configuration, (b) after collecting 4, (c) after collecting 7 and 6, (d) after resuming 2.

p processors, so p large tasks can execute in parallel. If the shared to-space is a bottleneck, which we do not expect in (small) shared memory systems, a pool of to-spaces can be provided.

When the window W has completely moved to the right and all virtual address space has been consumed, a global action is required to reclaim the unused holes in the virtual address space that have resulted from the local garbage collects. To preserve the ordering between the tasks, the virtual space is compacted by sliding the private heaps to the left. Besides adjusting the page tables, all physical pages have to be scanned for pointers to objects in virtually "moved" pages, so they can be relocated to their new positions. This expensive compaction method limits the usefulness of the storage allocation scheme to systems where the virtual address space greatly exceeds the size of the physical memory because then compactions are rarely needed.

3.3 The Virtual Allocation Scheme, VAS

We can improve the basic memory management's rapid consumption of the virtual address space by reusing holes on the fly. Holes in the virtual address space can be freely reused for new private heaps as long as the task ordering is preserved: tasks must be allocated to the right of their ancestors. Thus instead of always allocating memory at the right end, the enhanced Virtual Allocation Scheme (VAS) allocates a task's heap in the lowest free part of the virtual address space that lies to the right of the task's parent.

The VAS works well for the common case of a divide-and-conquer application that unfolds into a task tree with small interior tasks and big leaf tasks. After the interior control tasks have divided the work into independent components, the leaf tasks run for a long time to compute the partial solutions. Under the basic storage allocation scheme these leaf tasks move to the right each time the garbage collector is invoked, but under VAS these tasks remain in a small part of the virtual address space. A leaf task that needs to allocate a to-space can usually reuse the most recently released from-space of another task since there are no allocation constraints between leaf tasks; the only constraints are between interior tasks and leaf tasks.

Figure 5 shows the effects of VAS for the same example as with the basic scheme in Figure 4. Now the positions of leaf tasks 4, 5, 6, and 7 just permute, but do not shift to the right. In comparison with the basic scheme, the VAS administration is slightly more complicated since it has to record the holes in the virtual address space and the position of each task's parent.

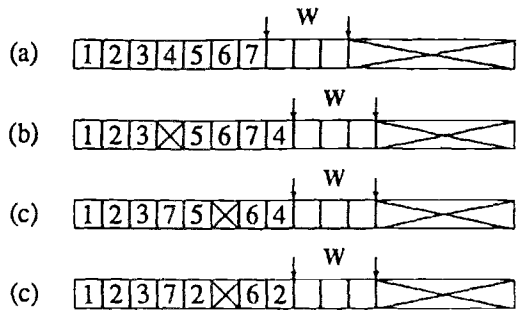


Figure 5: VAS: (a) initial configuration, (b) after collecting 4, (c) after collecting 7 and 6, (d) after resuming 2.

3.4 The Circular Allocation Scheme, CAS

Both previous storage allocation schemes use paging hardware to implement a large virtual address space. Obviously, this limits their applicability to multiprocessors with such hardware support, while those schemes also need a considerable amount of memory to store the page table. For example, the complete page table for a 4 Gbyte virtual address space on a MC88000 architecture with 4Kbyte pages occupies 4 Mbytes of physical memory. In addition the usage of a page as the unit of storage results in wasted heap space due to internal memory fragmentation. This has a strong effect on parallel applications that unfold into a large task tree where each interior task occupies a private page of memory that is only partially filled with useful data. Both sources of memory loss are tackled by the following allocation scheme that allocates storage in a virtual address space, but does not require paging hardware at all.

The Circular Allocation Scheme (CAS) uses a fixed translation scheme to map virtual addresses onto physical addresses. The upper bits of a virtual address are simply replaced by zeros to obtain the physical address. This gives a virtual address space that is wrapped circularly through the physical address space, see Figure 6. The "ghost" images of tasks 1, 2, and 3 cause a repeated pattern of holes in the virtual address space that extends right of the physical space.

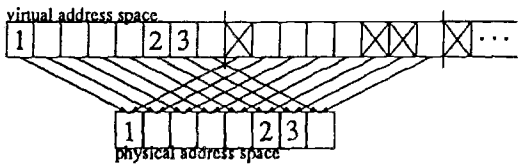


Figure 6: Circular Address Space

The CAS strategy uses the same allocation policy as VAS: a task's heap is allocated at the lowest available virtual address above the task's parent. Unlike VAS, however, CAS has to skip over the *ghost* images when looking for a free hole. For example, if task 3 wants to extend its heap with another two contiguous pages to the right of task 2, then CAS can not allocate it directly after its own heap, but has to allocate it in the large hole after the ghost image of task 1 as depicted in Figure 7.

Observe that the holes in the virtual address space are just a repetition of the physical holes. To take advantage of this redundancy by recording the status of the physical space only, the CAS strategy regards virtual addresses as the concatenation of a cycle-counter (most significant

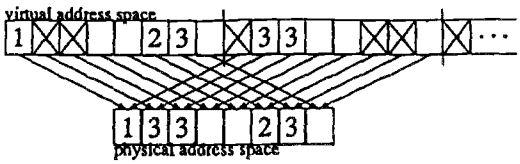


Figure 7: CAS after extension of task 3.

bits) and a base address in the physical space (least significant bits): addr ≡ cycle:base. When allocating storage to the right of a parent task located at address *cycle:base*, CAS first tries to locate a suitable hole at the right of the *base* in physical memory. If CAS succeeds then it returns *cycle:hole* as the start address of the new storage block, else CAS increases the *cycle* counter and starts looking at the beginning of the physical memory and returns *(cycle+1):hole* on success.

If the CAS strategy fails to allocate a large contiguous block due to external memory fragmentation, the scattered free space has to be compacted by sliding the tasks down to the left. This compaction only adjusts the *base* parts of pointers, but it is more expensive than with the two previous schemes since all data has to be copied as well. In the previous example compaction is needed when task 3 in Figure 7 wants to allocate 3 pages to perform garbage collection. The compacted memory layout is shown in Figure 8.

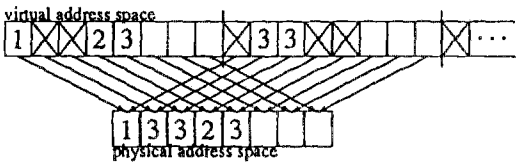


Figure 8: CAS after sliding compaction

Note that the sliding compaction has not compressed the virtual space, so an even more complex compaction method is needed when CAS runs out of the virtual address space: all *cycle* parts of pointers have to be cleared which requires a permutation of the tasks in physical memory to preserve the task ordering in the virtual address space.

The advantages of the CAS strategy are that there is no need to maintain the page tables since the address mapping is fixed; in fact it can be implemented in hardware by cutting the upper address pins of the processor! The fixed mapping also implies that CAS is not bound to the usage of pages, so heaps can have arbitrary sizes to avoid (internal) memory fragmentation. The CAS strategy, however, can only compete with the VAS strategy if both physical and virtual compaction operations are rarely needed.

4 Evaluation

To evaluate the performance of the three above mentioned memory allocation strategies, we have studied their behaviour by running a set of benchmark programs on a multiprocessor simulator. The programs are explicitly annotated to denote the parallel tasks, compiled and linked with one of three memory management schemes, and executed concurrently under control of a multiprocessor simulator that collects performance statistics. In particular we are interested

program	description	runtime	# tasks	mem. usage	# garb.coll.
fft v	A fast Fourier transform on a vector of v (= 2048) points, arrays are represented as lists [HV92].	1.5	15	1,846,985	14-36
queens n	A divide and conquer solution to the n (= 10) queens problem [LV91a].	1.4	165	325,121	164-172
wave i	A mathematical model of the tides in the North Sea. It consists of a sequence of i (= 10) iterations [Vre89].	1.7	41	197,072	59-61
comp_lab	An image processing application that labels all four connected pixels into objects [Sto87, ERW90].	1.6	465	1,178,347	464-476
15-puzzle	A branch and bound program to solve the 15-puzzle. The iterative deepening search strategy is used [Gla92].	28.0	24625	28,045,720	24624-24736

Figure 9: Benchmark programs; the simulated runtimes in seconds are for BAS on a 4 processor system; the memory usage is the number of words (32 bits) claimed in the heap; the range of garbage collects over all runs.

in the amount of memory wasted due to memory fragmentation, and the usage of the virtual address space. For CAS the number of physical compactions is also important information.

The benchmark programs are written in a lazy functional language that supports the divide-and-conquer paradigm through the *sandwich* primitive [Vre90], which is used to annotate parallel tasks explicitly at the source level. The programs, as described in Figure 9, have been selected on two criteria: the program should be non-trivial, and the program's task structure is expected to put strong demands on the memory allocation algorithm.

The FAST compiler [HGW91, LH92] translates the functional benchmark programs into equivalent C programs that can be used in combination with a copying garbage collector. The compiler, for example, maintains an explicit call stack to bring all pointers under control of the garbage collector. In addition the compiler recognises the *sandwich* construct and generates code to force the shared task data into read-only form, and to call the runtime support system with a list of tasks that need to be scheduled for execution.

The runtime support system, which is coded in C, handles the scheduling and memory allocation of tasks. Three different versions have been constructed, implementing the storage allocation schemes of Section 3. The common task scheduler employs a list-scheduling policy and maintains a global stack of runnable tasks, where newly created tasks are deposited and idle processors look for work.

When tasks run out of heap space, they double their heap size by allocating a new block if enough global memory is available, otherwise the garbage collector is invoked. When a task finishes, its result is compressed by invoking the garbage collector, whereafter the unused heap space is returned to the global pool. In this pilot implementation we have not directly made use of virtual memory hardware, but rather simulated the allocation schemes with one large chunk of physical memory. This suffices to collect the statistics about the memory consumption of the benchmark programs.

Instead of running the programs and run time support systems on a real shared-memory multiprocessor, we have simulated the parallel execution. This has the advantage that we can

easily use different design parameters, and can obtain statistics without disturbing the execution. The multiprocessor simulator is a stripped version of the MiG simulator [MLH92], which is developed to study cache coherency and bus saturation effects of parallel functional programs. Instead of tracing the memory references at the level of bus transactions as in the original, the multiprocessor simulator only counts the number of executed instructions, loads, and stores, but semaphore primitives are fully simulated to get realistic synchronization behaviour. Because of this underlying execution model we have restricted ourselves to simulations of systems with 4 processors. For a single processor system, the simulator provides accurate execution times for the benchmark programs within a 15% range of the actual measured times on a SUN 4/690.

BAS

At first, we study the behaviour of the basic allocation scheme of Section 3.2, which always allocates new storage at the right end of the virtual memory space. Table 1 summarises the results of the benchmark programs for the basic scheme with 1024 word (= 4Kbyte) pages. The column labeled "physical" lists the maximal amount of heap words in use at any moment in time during the execution of the application. This number does not include code and static data that are located in separate segments, nor does it include the space needed for the page tables, but it does account for the memory fragmentation inside pages. The second column contains the highest virtual address used by the application, and it shows that the simplistic basic scheme consumes large quantities of virtual memory space. The 15-puzzle, for example, allocates 200 times as much virtual space as physically needed.

program	physical	virtual	claim rate
fft 2048	1,261,568	3,917,823	2.67 Mw/s
queens 10	77,824	1,443,839	1.04 Mw/s
wave 10	49,152	804,863	0.47 Mw/s
comp_lab	208,896	4,738,047	2.89 Mw/s
15-puzzle	726,016	142,187,519	5.13 Mw/s

Table 1: Performance statistics of the basic scheme.

The ratio between virtual and physical memory usage depends strongly on the application's input parameters and cannot be used as a meaningful characteristic in general. Instead we have listed the application's claim rate (in Mwords/second) that shows how fast virtual memory is consumed. The high claim rate of the 15-puzzle is partly caused by the large number of tasks, which results in considerable memory fragmentation inside pages. The claim rate indicates how frequently a compaction of the virtual address space is needed. In our benchmark, the claim rates are at most ca. 5 Mwords/second, so an application can execute in a 1 Gword virtual address space for at least 200 seconds without a compaction on a system with four 20 MIPS processors. A 16 node processor system will (if the program has enough parallelism) consume the same virtual space in roughly 50 seconds. A compaction would take ca. 1 second per Mbyte of physical memory.

VAS

The results of using the VAS strategy are shown in Table 2. In comparison with the basic scheme, the benchmark applications under VAS use slightly more physical memory, but the virtual memory consumption has been significantly reduced to within a factor 2 of the application's physical memory requirement. Therefore an application is unlikely to need an expensive compaction to compress the virtual memory space, hence, the compaction operation probably does not have to be implemented at all.

program	physical	virtual
fft 2048	1,261,568	1,572,863
queens 10	80,896	105,471
wave 10	49,152	73,727
comp_lab	234,496	517,119
15-puzzle	849,920	898,047

Table 2: Performance statistics of the VAS strategy.

The simulator records the allocation overheads, like managing the list of free pages, of the memory management schemes. The differences, however, are marginal and only account for ca. 0.5% of the total execution time in the usual case that no compactions are needed.

CAS

First we have run the benchmark programs under CAS with the same pagesize (1024 words) as the basic and VAS strategies. The results in Table 3 show the number of compactions to recover from physical memory fragmentation besides the physical and virtual memory usage

program	physical	virtual	compacts
fft 2048	1,261,568	1,703,935	0
queens 10	76,800	159,743	14
wave 10	49,152	73,727	0
comp_lab	241,664	492,543	4
15-puzzle	775,168	803,839	0

Table 3: CAS performance, pagesize 1024 words.

The difference in physical memory usage under CAS in comparison to VAS is caused by their difference in allocation time, which results in different task scheduling decisions. The virtual memory usage under CAS exceeds the physical memory usage only by a small factor, just like for VAS. Note that only the queens and comp_lab applications perform compactions to compress the physical memory space.

Next we have run CAS with a small pagesize of 32 words to lower the internal memory fragmentation, see Table 4. To our surprise some programs need more physical and virtual memory; only the queens and 15-puzzle benefit from the small pagesize. The increase is caused by the internal overhead to administrate the linked list of heap blocks. The "wasted" space forces the large tasks to allocate another block just before finishing their computation, and since tasks double their heapsize when running out of storage only a small fraction is actually used.

program	physical	virtual	compacts
fft 2048	1,586,176	2,359,295	0
queens 10	49,760	109,759	9
wave 10	60,000	134,143	10
comp_lab	245,792	452,671	4
15-puzzle	467,072	1,066,655	1

Table 4: CAS performance, pagesize 32 words.

The number of compactions listed in the performance results is a worst case value since the applications have been simulated on a multiprocessor with the minimum amount of physical memory needed by the specific application. Adding about 50% extra memory decreases the number of compactions to zero in all cases. Thus the CAS scheme performs well if the amount of physical memory in the shared-memory multiprocessor is somewhat larger than the absolute minimum required by the application.

Stressing the allocation schemes

The benchmark results for VAS and CAS show that the applications can be efficiently executed in a surprisingly small virtual address space. This is a consequence of the scheduler that traverses the fork-join tree in a depth-first manner, hence at any moment the allocation strategies only have to satisfy a logarithmic number of the task allocation constraints (depth of the tree). To test the limits of the allocation schemes we therefore created a synthetic application, called *spine*, that unfolds into a degenerated tree: a linear list. The spine of interior tasks forces the allocation schemes to allocate new tasks at the right end. The results for a spine of length 512 on a 4 processor system with 1024 word pages are presented in Table 5.

strategy	physical	virtual	comp	claim rate
BAS	393,216	17,105,919	-	8.6 Mw/s
VAS	393,216	2,117,631	-	1.1 Mw/s
CAS	393,216	793,599	27	0.4 Mw/s

Table 5: Performance statistics of *spine*.

The synthetic spine program allocates virtual address space somewhat faster than the benchmark applications: a claim rate of 8.6 Mwords/second versus 5.1 for the 15-puzzle. The large difference in virtual address consumption between the basic scheme and VAS is caused by leaf tasks that have allocated address space far beyond the growing spine: whenever such a leaf task finishes its computation, the garbage collector is invoked to compress the result and the reclaimed space at the right of the spine can be reused for new tasks. The CAS strategy needs even less virtual address space because of the 27 physical compactions: they also reclaim the virtual address space that resides in the currently highest cycle.

The total amount of virtual space claimed by the *spine* program can be made arbitrarily large by increasing the length of the spine, but the moderate claim rate limits the virtual compaction frequency to a low value for all three memory management strategies.

5 Conclusions and future work

It is possible for divide-and-conquer applications on shared-memory multiprocessors to use a local copying garbage collector; executing tasks reclaim their garbage independently of other tasks and processors. The avoidance of global synchronisations to control garbage collects has been achieved by assigning private heaps to individual tasks. The memory management strategies handle the resulting scattered heaps of join tasks by allocating storage in a virtual address space so that active (leaf) tasks never interleave with their suspended ancestors. By time-sharing the to-space and limiting the maximum task size, the 50% waste of memory reserved for the to-space can be significantly reduced.

From the simulation results we conclude that both the VAS and CAS schemes are feasible memory management strategies since in exceptional cases only they have to issue an expensive compaction operation to reclaim wasted address space. The synthetic *spine* program shows that applications can consume an unlimited amount of virtual address space, but for the benchmark programs the needed size of the virtual address space is less than three times the physically required amount of memory, hence, no compactions are required. The CAS strategy, however, does occasionally compress the data in physical memory to overcome fragmentation of free space. The number of these physical compactions is negligible when the applications run in a memory whose size is 1.5 times the application's minimally required amount of physical memory.

The multiprocessor simulator also provides figures for the performance consequences of CAS and VAS since the runtime support code for managing the free list, compacting the address space, etc. is traced too. However, since the simulated differences are small we will not draw conclusions about their relative performance. Instead we will measure the difference in a real implementation; at the moment, both CAS and VAS are being implemented on a 4 node 88000 multiprocessor with 64 Mbytes of main memory and MMUs to support virtual memory.

The current implementation is based on the divide-and-conquer paradigm (the *sandwich* annotation). We plan to extend the memory management strategies to cover the more general spark&wait paradigm (for example, *futures* in LISP). This is not trivial since the parent task continues to execute in parallel with its child tasks, while both CAS and VAS assume that parent tasks are waiting and do not need to be garbage collected. We foresee that this conflict can be solved by splitting the parent's heap in two parts: a fixed public part that contains data shared with child tasks, and a private part where the parent can claim and collect storage as usual.

Acknowledgements

We would like to thank our colleagues Pieter Hartel, Rutger Hofman, Marcel Beemster, and the referees for their helpful comments on draft versions of this paper.

References

[AEL88] A. Appel, J. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *Proceedings conf. on Programming Language Design and Implementation*, pages 11–20, 1988.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *CACM*, 13(11):677–678, November 1970.

[Cra88] J. Crammond. A garbage collection algorithm for shared memory parallel processors. *Int. Journal of Parallel Programming*, 17(6):497–522, 1988.

[ERW90] H. Embrechts, D. Roose, and P. Wambacq. Component labelling on a MIMD multiprocessor. prepublished report, Dept. of Comp. Sci., Katholieke Universiteit Leuven, Belgium, 1990.

[Gla92] J. Glas. The parallelization of branch and bound algorithms in a functional programming language. Master's thesis, Dept. of Comp. Sys, Univ. of Amsterdam, April 1992.

[Hal84] R. Halstead. Implementation of multilisp: Lisp on a multiprocessor. *Proceedings ACM symp. LISP and Functional Programming*, pages 9–17, 1984.

[Har90] P. H. Hartel. A comparison of three garbage collection algorithms. *Structured programming*, 11(3):117–127, 1990.

[HGW91] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. Technical report CSTR 91-03, Dept. of Electr. and Comp. Sci, Univ. of Southampton, UK, January 1991.

[HV92] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. Technical report CS-92-02, Dept. of Comp. Sys, Univ. of Amsterdam, May 1992.

[LH83] Henry Liebermann and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 23(6):419–429, 1983.

[LH92] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. Technical report CS-92-03, Dept. of Comp. Sys, Univ. of Amsterdam, May 1992.

[LV91a] K. G. Langendoen and W. G. Vree. Eight queens divided: An experience in parallel functional programming. In R. Dietrich J. Darlington, editor, *Declarative programming*, pages 101–115, Sasbachwalden, West Germany, November 1991. Springer Verlag.

[LV91b] K. G. Langendoen and W. G. Vree. FRATS: a parallel reduction strategy for shared memory. In M. Wirsing J. Maluszynski, editor, *3rd Programming language implementation and logic programming, LNCS 528*, pages 99–110, Passau, West Germany, August 1991. Springer Verlag.

[Mar91] L. Maranget. GAML: a parallel implementation of lazy ML. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 102–123, Cambridge, Massachusetts, September 1991. Springer Verlag.

[MLH92] H. L. Muller, K. G. Langendoen, and L. O. Hertzberger. MiG: Simulating parallel functional programs on hierarchical cache architectures. Technical report CS-92-04, Dept. of Comp. Sys, Univ. of Amsterdam, June 1992.

[Sto87] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *J. Parallel and Distributed Computing*, 4(1):95–115, February 1987.

[Vre89] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, December 1989.

[Vre90] W.G. Vree. Implementation of parallel graph reduction by explicit annotation and program transformation. *Mathematical Foundations of Computer Science 1990, LNCS 452*, pages 135–151, 1990.

Incremental Multi-threaded Garbage Collection on Virtually Shared Memory Architectures

Thierry Le Sergent, Bernard Berthomieu

Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
e-mail: lesergen@laas.fr, bernard@laas.fr

Abstract: This paper describes a multi-threaded and incremental garbage collector operating on shared memory architectures. The technique was developed for parallel implementations of the language LCS, a high level parallel programming language.

An incremental, page trap based, collection algorithm operates locally on each of the processors. Processors alternatively plays the role of mutator and collector. The processors cooperate for collection and mutation; idling processors perform part of the collection task for the others until they acquire some work. The progress of collectors versus allocations is controlled by a scan credit mechanism that guarantees a responsive execution of the application. There is no static partitioning of the storage among the processors; pages are dynamically allocated to any processor, for specific purposes.

Two implementations are discussed: the first is suitable for operation on a shared memory architecture; the second provides garbage collection services as added functionality to a distributed shared virtual memory service.

1 Introduction

This paper presents a garbage collection technique suitable for multi-threaded applications running on multiprocessor targets with a shared memory abstraction. The shared memory abstraction can be provided by the architecture, or virtually, obtained through the Distributed Shared Virtual Memory paradigm [LH89][CBZ91]. The multi-threaded applications may be the result of compilation of programs written in programming languages with explicit or deduced parallelism. The different threads of control of the application are assumed to share a single address space, a global storage allocation and reclamation discipline is thus required.

These techniques were designed for implementing a parallel virtual machine and compiler for the language LCS [Ber88][BGG91], an extension of the language Standard ML [MTH90] with processes based upon the CCS formalism [Mil80]. However, most of the results are independent of this context; similar techniques would be required for parallel implementations of many programming languages with transparent memory management. In distributed implementations of LCS, the algorithms presented here are complemented by algorithms for distributed scheduling, load balancing, and a few other distributed functions

which cannot be discussed here. A complete account of the experiment is presented in the first author's forthcoming thesis [LeS92].

The starting point of our technique is [ELA88]. The heap space is organized in pages; an incremental page based collection algorithm operates locally on each of the processors. Unlike Ellis et al., we reject the idea of having specialized collector or mutator threads. Instead, each thread here alternatively plays the role of mutator and collector. This allows a better cooperation of several threads for performing a single collection, when required, or working together to make the whole application progress. We can have at any time a varying number of collectors and/or mutators; the progress of collection versus allocation is controlled by a specific scan credit mechanism. The several processes share a virtual space of references, as in [Hal84]; but the assignment of storage to each processor is not statically determined. Instead, pages are dynamically allocated to the processors, for specific purposes. The page management algorithms guarantee the consistency of the information held in the pages.

Integrating the garbage collection mechanism with the other paradigms present in the underlying architecture received much attention. Two implementations of the allocation/reclamation algorithms are proposed. The first is suitable for shared memory services directly provided by physically shared memory architectures. A second targets systems with physically distributed memories. In this case, the page allocation and collection services are provided together with a virtually shared memory service, as added functionality, rather than running on top of it.

Section 2 reviews and discusses versus our goals a number of mechanisms taken from existing garbage collection techniques. The collection method is presented in section 3. Section 4 suggests a canonical implementation on shared memory architectures while section 5 investigates an implementation with a distributed shared virtual memory service. We conclude with some remarks prompted by a prototype implementation of the algorithms and discussion of some possible variants and enhancements.

2 Architecture

2.1 Copying collection techniques

Garbage collecting techniques have been developed for a long time; reference [Coh81] surveys a number of algorithms. A convenient terminology for discussing memory allocation and reclamation techniques is that of *mutators* and *collectors*. A thread is a mutator when it allocates new storage, or updates existing storage; it is typically the application to be run. A thread is a collector when the operations it performs are relevant to storage reclamation. The storage is assumed to be organized as a set of *cells*, possibly referring each others and residing in a *heap*. A number of references to such cells, called the *root*, constitute the context (or registers) of each mutating thread.

Among the methods based upon transitivity of references, the copying methods certainly are the most satisfying. They consist in copying the reachable cells from a first space called the *fromSpace*, into another space, called the *toSpace*. They yield fast collectors, since only the cells reachable from the root are visited; they are able to reclaim circular structures and they have the effect of compacting the heap, which reduces page thrashing. However, copying

techniques generally yield a poor storage occupancy, but most of that space is not permanently required.

The basic stop and copy version of the technique suffers the essential drawback of requiring the mutator to stop while the collector is proceeding, resulting in relatively long latency. Baker's variant [Bak78] was aimed at solving this drawback. The mutator is resumed immediately after the cells referenced in the root data have been copied and their references updated in the root data. The cells copied are left unscanned. The mutator must only operate on cells in *toSpace*. For achieving this, each access from the mutator to a cell residing in *fromSpace* forces its copy into *toSpace*, if it was not copied yet, and updates its reference. A pointer, **B**, points to the next available address for copying cells; another pointer, **S**, points after the last cell scanned. New cells are allocated from a pointer **T**, initially set to the last available address of *toSpace*, and decreasing towards the copy pointer **B**.

Conditions are established to make the whole mutator and collector process *real-time*, in a certain sense; Baker's algorithm is incremental. For each allocation of one word of storage, k words are scanned, where k is some non negative constant. For a program which has a maximum cell requirement of N words, with a half-heap size of t, the parameter k must be greater than N/(t-N) to guarantee that scanning is terminated before pointer **B** reaches **T** [Bak 78].

Baker's method is costly if the test it requires at each access of a cell for determining if it stands in *toSpace* or in *fromSpace* is done by software. Ellis et al. [ELA88] satisfactorily solved this drawback by using the memory protections facilities provided by the hardware to prevent access of unscanned cells by the mutator. The heap is organized in pages; to enforce the property that the mutator only sees scanned cells, all pages of the heap containing unscanned cells (i.e. between pointers **S** and **B**) are read-protected. An access to these pages by the mutator would be trapped by the hardware and an exception would be raised; the access exception is handled by scanning the faulty page and then relaxingits protection, allowing the mutator to resume.

2.2 Garbage collection for multi-threaded applications

We will only consider here parallel applications operating on a single virtual address space. The complexity of collection algorithms proposed for multiple address spaces applications results from the absence of an observable global state, and the problem of migrations of cells between the processors [Rud86].

The approach taken in [Hal84], also relying on a single virtual address space, is to have a global heap statically and equally partitioned, each sub-heap being under control of one of the (virtual) processors. Each processor executes on its sub-heap a collection algorithm based upon Baker's, with its *toSpace*, local roots, and using local pointer variables **T**, **S** and **B**. The changes brought to the basic algorithm for multiprocessor operation consist of a lock-bit associated with each address in *toSpace* and each cell in *fromSpace*, to ensure atomicity of the copy and update operations. A global synchronization ensures that all processors flip their spaces before one of them starts the next collection.

The Multilisp treatment suffers two major drawbacks. Firstly, garbage collection is initiated when any of the sub-*toSpaces* is full, whatever the content of the others, yielding in practice much more frequent garbage collections than would be required in the single-heap

case. Secondly, each processor must copy for itself all the cells only locally referenced (the gray areas in figure 1 below), with no possible help from the other processes for scanning these.

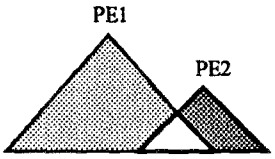


Figure 1. Sharing of heap cells

2.3 Architecture

The goal is to design the memory management layer for a virtual machine running LCS, an experimental parallel programming language. LCS programs, in which parallelism is made explicit, lend themselves naturally to parallel evaluation. However, typical LCS programs use a very large number of threads, from a few to several thousands, or tens of thousands, most of them very short lived. Consequently, the LCS virtual machine is constituted of a medium, fixed, number of processes, each handling execution of a number of LCS threads. A distributed scheduler, with a load balancing algorithm, distributes execution of the many LCS threads on the fewer processes.

The shared memory abstraction is adequate for our purpose. LCS, like many other high level languages, handles values which are generally trees, possibly with circularities. It would be very expensive to transmit copies of such values by messages between processes; LCS values are transmitted "by references". Another argument for the shared memory abstraction is that typical LCS programs do not heavily rely on side-effects, yielding, in practice, relatively few write access conflicts on the shared heap.

From the previous review, we will retain the two-space copying and compacting method and the incremental technique [Bak78]. We will retain from [ELA88] the idea of using page protection traps for protecting parts of the heap, with a recovery mechanism. This, despite the added complication of a heap granularity distinct from the constituents of the heap (the cells), makes Baker's method of incremental collection efficient. However, we will not retain the specialized mutator and collector threads advocated in this reference, and have instead threads which may be alternatively mutator or collector.

The page allocation mechanism allows processors to share the work of incremental scan; the progress of allocation versus collection will be controlled by a credit mechanism. While an incremental criterion similar to Baker's is maintained, idling processes are requested to perform some scan until they acquire some work. Collection may then progress "in advance" over the mutation, and this will reduce the amount of scan to be done after future allocations. Using this mechanism, most of the potential advantages of collectors operating in parallel are recovered, while preventing the storage overflow problems which are difficult to avoid with asynchronous parallel collectors. In [ELA88], this particular problem is delayed, rather than avoided, by flipping spaces when occupancy of *toSpace* reaches some threshold, rather than when *toSpace* is full. Alternatively, [BSD91] prevents overflow by controlling the scheduling of the collector and mutator threads.

All processes share a single address space, but, conversely to [Hal84], the heap will not be statically partitioned over the different processors; we will make use instead of a paged heap, with pages dynamically allocated to any process. This treatment avoids the two major drawbacks of a statically partitioned heap; no process plays a particular role.

The collection technique is presented in the next three sections. The routines have been split as "front-end" routines, those interfaced with the application, and "back-end" routines, those directly using the storage services provided by the operating system.The front-end routines are discussed in the next section. The back-end routines may benefit from an encoding matching the underlying operating system services. Two implementations of the back-end routines are discussed. The first, in section 4, is suitable for any virtual shared memory target. The second, in section 5, provides these routines together with a distributed shared virtual memory service [LH89].

3 The multi-threaded garbage collector, front end routines

3.1 Storage services and layout of the storage of processes

The storage for each thread is constituted of a number of segments. Each segment is a contiguous address range partitioned in pages, all of the same size. A segment may be shared by several threads. We will assume that the pages which constitute a segment can be individually attached or detached from the address space of a thread, and that any access from a thread to a page not currently part of its address space raises an exception for which a handler may be set. These services are available on many stock operating systems; setting page protections at user level is a feature of Unix SVR4 and Mach based operating systems. Depending on the operating system, page protection facilities are either provided at thread level (otherwise sharing their address space), or at process level (possibly sharing part of their virtual space); we will use indifferently the words "thread", "process", "processor", or "site" in the following sections, to mean a thread of control associated with some shared storage for which page protections may be locally manipulated.

No assumptions are made about atomicity of updates on shared pages. The algorithms will use page locks where atomicity need be enforced. Any attempt to lock a page already locked should make the requesting process wait until the page is unlocked.

Each process will make use of three segments: two heap areas of identical size, referred to as *toSpace* and *fromSpace*, and a *stack* area. Each process shares with all others the heap areas. Though each process privately extends or shrinks its stack segment, it is convenient to have the stack segments shared by all processes so they can get help from the others for scanning their stacks. We took this choice here, though the alternative choice of having process stacks in private areas could have been taken as well, at the price of some extra complications for detecting end of collection. Finally, each process privately owns a set of registers and the three pointer variables, **S**, for scan, **B**, for copy, and **T**, for allocation of new cells; they also handle a number of private or shared auxiliary variables. To avoid contention on variables **S**, **B** and **T**, the processes will operate on different pages for each of these operations.

Pages are dynamically allocated to the processes, following a strategy to be made precise in a latter section, and released after use. The different pages may hold information of differ-

ent nature (e.g. none, unscanned copies of cells, coherent cells, etc.), we will talk about the state of a page to mean the nature of the information it holds. The use of the storage, and the cooperation between the processes for achieving collection and allocations, are performed through of a number of page allocation and liberation routines, one for each "kind" of pages needed. These routines encapsulate the necessary synchronizations, and bookkeeping, for pages.

3.2 Heap allocation

A page will be identified by any address in the range it provides. New cells are allocated from a pointer T; the procedure `Allocate` is always called with a current T defined. The usual "end-of-heap" test is replaced here by an "end-of-page" test. When the cell (of size n) to be allocated does not fit on the current page, the current page is freed and as many pages as required for allocating this cell are requested by a call to `allocPageT`. The procedure provides pages for n contiguous addresses, attaches them to the address space of the process, and sets the local pointer T. If the required amount of pages cannot be allocated, then a flip is initiated. Procedure `fill` fills the space allocated and advances pointer T.

```
Allocate (n,content) =
      (if spaceLeftOnPage(T) < n
       then (freePageT();
             Scan();
             if not(allocPageT(n))
             then (Flip();
                   if not(allocPageT(n)) then FAIL));
      fill(T,n,content));
```

In general, cells may overflow the page size. The solution proposed in [ELA88] is to associate with each page a "crossing" flag, set for pages not beginning with a new cell. The alternative taken here is slightly more space consuming, but allows to scan pages individually. It consists of associating with the crossing flag the minimum information needed to scan the part of the cell starting the page. This added information allows the scanner to see that part of cell as a complete cell; it would typically hold an indication of the nature of the cell, which determines a scanning method, and the size taken on the page. This information cannot stand in the pages themselves since it must not take any address slot. When crossing a page boundary, procedure `fill` releases the current T page by a call to `freePageT`, and fills the crossing information for the following page.

3.3 Incremental Scan

Each thread alternatively plays the role of mutator and collector. Following Baker's incremental technique, each allocation of a word in the heap forces to scan a number K of words. The scan is performed after the page currently filled is released, rather than performed after each allocation; this reduces the overhead on allocations. A procedure `allocPageS` provides, and attaches, the next page to be scanned. It sets the scan pointer S, if some incremental

scan is required, or, otherwise, returns false. Procedure `freePageS` releases page S when its scan is complete. This is repeated until a call to `allocPageS` returns false, meaning either that no more incremental scan is currently required, or that collection is complete. The pages are locked while being scanned, to prevent concurrent scanning by another processor (`allocPageS` locks, `freePageS` unlocks).

```
Scan () =
      while allocPageS()
      do (scanPage();
          freePageS());
```

Mutation continues after end of collection has been detected, until *toSpace* is full; this also contributes to decrease the average cost of allocations. The progress of scan versus allocations is controlled by a credit mechanism soon to be described.

3.4 Copying

Copies of cells resulting from scan are done on pages allocated by a third procedure, `allocPageB`. Copies are allocated from a pointer B. For allowing the scanner to work on a page basis, the cells are copied on pages distinct from those on which new cells are allocated. The procedure `allocPageB` locks the pages it allocates, attaches them to the address space of the thread, and sets pointer B. `freePageB` detaches the current B page, so that any further access to this page will provoke an access violation, and unlocks the page. The lock prevents other processors from attempting a scan of the page while it is being filled. In practice, `scanPage` would make use of the crossing information for pages, for determining the location and size of the first cell on the page.

```
scanPage () =
      if allocPageB(threshold)
      then (for all cell in page of S
            do scanCell(cell);
            freePageB())
      else FAIL;
```

All processes share, and have authorized access to, all pages of *fromSpace*. To prevent cells from being copied concurrently, the page on which an old cell stands is locked until the cell is copied, its copy-bit updated, and its forwarding address set.

```
scanCell(cell) =
      for all cell' referenced in cell
      do (lock(cell');
          if not(copied(cell')) then Copy (cell');
          update reference to cell' in cell;
          unlock(cell'));
```

The Copy routine copies cells from B and sets their copy-bit and forwarding address. The low level procedure copy advances B; it also releases the current B page and fills the crossing information of the following page when crossing a page boundary. If a cell does not fit on the current B page, then that page is freed and as many contiguous pages as required are requested by a call to allocPageB.

```
Copy (cell) =
      (if spaceLeftOnPage(B) < size(cell)
       then (freePageB();
             if not(allocPageB(size(cell))) then FAIL);
       forward=B;
       copy(cell,B);
       mark_copied(cell);
       set_forwarding_addr(cell,forward));
```

3.5 Scan on page access exception

Any attempt from a processor to access a page not part of its current address space raises an access exception. Upon an access exception, a handler routine, Handle, is invoked, which performs some treatment, and the computation is resumed at the instruction that caused the exception. There are a number of reasons for which a page accessed may not be part of the current address space of a process, including the case where the page has been filled with copies of cells, but has not been scanned yet. As in Ellis's algorithm, the mutator expects to see scanned cells only.

On that instance of access exception, the handling procedure first attaches the page to the address space of the process. The page is then locked to prevent concurrent scan by another process, and it is scanned. Upon completion of scan, the page is freed and unlocked (by a call to freePageS). Conversely to [ELA88], all page protections here can be manipulated at user level.

From now on, that particular page holds cells with consistent references, and no exception will be raised when accessing it from that process. But the page is still non-attached by the other processes. It is assumed that, given a page of *toSpace*, a process can decide if this page was scanned or not. The handling routine, which is the same for all processes, should first check that the faulty page has not been scanned yet, before scanning it; otherwise it just attaches the page.

3.6 Stack allocation and scanning

Stacks are the most usual technique for implementing parameter passing. If application stacks were encoded in the heap, with stack frames encoded as cells, then only the stack pointer would be part of the root references of a process. But, essentially for efficiency reasons, function call stacks are often encoded as arrays, in a dedicated area. It is this entire array which must then be considered root data by the collector. Moreover, stacks may grow very large on languages favoring recursion, such as LCS; scanning the full stacks at flip time, as should be

done for all root references, might break the low latency requirement for the mutation process, as noticed in [Bak78] and [ELA88].

Stack frames are allocated on contiguous pages in an area distinct from the heap. After a flip, all the pages below the topmost stack page are detached from the address space of the thread. Only the topmost page is scanned as part of the root data. The other pages will be scanned on access exceptions or by incremental scan. This strategy implies that stack pages may be scanned individually, without knowledge of the boundaries of stack frames. This constraint is easily satisfied if multi-words values are allocated not in the stack area, but in the heap instead, at the price of an extra indirection. The stack should only hold references to heap cells and constant values, which must be encoded as if they were occurring in a heap cell.

On an access exception on a stack page, the handler routine scans the faulty stack page and the mutator is resumed. Stack pages are scanned by the previous `scanPage` routine (we may assume a predefined crossing information, identical for all stack pages), only words at addresses below the stack pointer are scanned.

As for the heap, the stacks must be incrementally scanned. A number k' of stack words must be scanned for each word allocated, so that the stacks, of total size s, are scanned before *toSpace* is filled. Parameter k' depends upon the size s of the stacks, known at flip-time. The smallest number of words allocated between two flips is $t/(k+1)$, where t is the size of *toSpace* and k is Baker's parameter for the heap. Allocating that amount of words should force a complete scan of the stacks; consequently: $k' = (k+1)*s/t$[1].

Practically, the words in the stacks and those in the heap are globally considered, and $K=k+k'$ words are scanned for each word allocated. Parameter K is adjusted at flip time, from a fixed k and a parameter k' computed from the previous equation. The routine `allocPageS` allocates either an heap page or a stack page, as long as some is available. Furthermore, since we choose to have the stack areas shared by all processes, any process is able to scan the stack of any other. Finally, in order to favor cells which have the longest life expectancy, the stack pages are allocated by `allocPageS` from bottom to top.

3.7 Idling processes and the scan credit

Processors idling by lack or work are requested, through a `background` procedure, to perform some scan until they get some work (by a scheduling and load balancing mechanism not discussed here). The words scanned from the `background` procedure or upon access exceptions are considered as words scanned "in advance" for allocations to come (a credit on the amount of words to be incrementally scanned).

In order to fulfil the low latency requirement, the credit mechanism is implemented with two credits: A *local* credit (one per thread), decremented by `freePageT` (of the amount of words allocated on the page, multiplied by constant K) and incremented after scanning a page

---

1. Baker uses k' = k*s/n, where n is the amount of words in the heap when collection is complete. The difference comes here from the fact that flip does not occur at end of collection, but when *toSpace* if full. n is unknown at flip-time, then, but an upper bound is deducible from k and t, by n = t*k/(k+1).

(upon an access exception handling or a local incremental scan), and a *shared* credit, incremented by `freePageS` when it is invoked from the `background` procedure.

When allocating, a process uses a maximum amount of its local credit. If the process has to do some scan (i.e. if its local credit is negative), then it first checks the shared credit. If the shared credit cover its debt, then no scan is performed and the shared credit is decremented of the local debt. Otherwise, the local debt is decreased of the shared credit, the shared credit is reset and scan proceeds. Finally, when a processor is idling, it first transfers its local credit to the shared credit, so that other processes can benefit of it, before scanning for the other processors.These computations, as well as the test deciding if incremental scan is required, are encapsulated in the procedures `freePageT`, `freePageS` and `allocPageS`.

The scan credit mechanism brings two main advantages. Locally, it makes incremental scan lazier; the amount of words scanned on access traps or by idling processors are subtracted from the amount of words to be incrementally scanned. Globally, It allows the processors to cooperate for scanning, some of them performing part of the scan task for the others, with an actual parallelism. The number of collectors is dynamic and the algorithm controls the progress of the collectors versus the amount of data allocated.

3.8 Flip

A thread notices that *toSpace* is full when a call of `allocPageT` returns false; it then initiates a flip. A synchronization of all threads is necessary before the local flips take place, to prevent the different threads from working with different views of the heap. Then, all threads stop mutation, release the pages they held, and start a local `flip` procedure. It must be noticed that, when a flip occurs, the previous collection is necessarily complete; all threads are then either idle or mutating.

The `flip` procedure itself first consists of inverting the roles of the *to* and *from* spaces. Each thread then attaches to its address space all the pages of *fromSpace* (the former *toSpace*) and detaches all the pages of *toSpace*. The pages of *toSpace* will be attached on request. All stack pages are also detached, except the topmost page. The thread then initializes its local bookkeeping information and variables (including a T page for non-initiators), and scans its registers and top stack page. Mutation is subsequently resumed.

The initial attachment of the pages of *fromSpace* avoids nested access exception handling. The initial detachment of all pages of *toSpace* is the consequence of the strategy retained for page management. This strategy is to force an access exception in a process every time this process has no precise knowledge of the content of a page; the access exception handler will determine the exact nature of the contents of the faulty page, and perform the adequate treatment. To avoid detachment of pages attached in several processes, the page management guarantees that, once a page is made available to a mutator, it will never need to be detached from any process until the next flip.

3.9 States of pages, and transitions

Pages have states, identifying the nature of their contents. Initially, and just after a flip, the pages of *fromSpace* have no significant state and are attached in all processes. The *stack* pages

have one among two states: Scanned, or Unscanned. The pages of *toSpace* are initially detached from all processes and all of them initially have state Empty. The figure below depicts the different possible states and transitions of a page in *toSpace*, with meanings obvious from the previous sections.

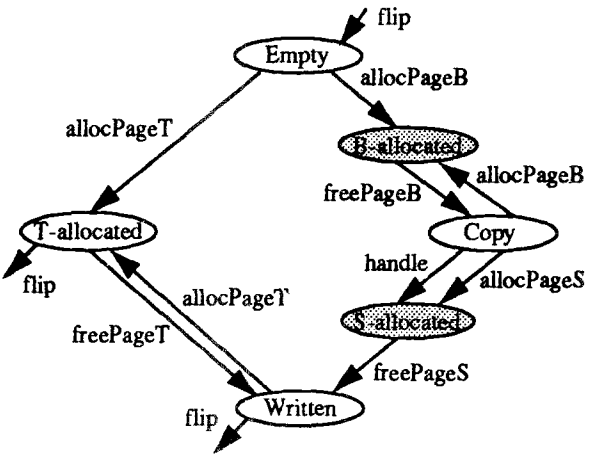


Figure 2. The different states of a page in *toSpace*

Consistency of the information held by the pages requires that the pages in gray states in figure 2 are locked by the processor which set them in that state; they must not be available to the other processors until they leave that state.

Pages may be released by `freePageB`, `freePageT` or `freePageS` even if not completely filled. In order to maintain a reasonable mean page occupancy of the storage, a policy of reallocating partially filled pages may be adopted. Given a threshold `h`, a page will be said full if less than `h` address slots are available on that page. Partially filled pages in state `Copy` will be reallocated by `allocPageB`, partially filled pages in state `Written` will be reallocated by `allocPageT` (when the size requested is not greater than threshold `h`, in both cases). This explains some of the transitions in the above graph.

4 A canonical implementation of the back-end routines

4.1 Page pools

Pages must be allocated according to their state. A convenient encoding, easing retrieval of pages, is to store addresses identifying these pages in different data structures, according to their states. We shall need five, shared, pools of pages:

The EMPTY pool holds pages which have never been allocated. If empty pages are chosen to be the consecutive pages following the last allocated page in the *toSpace* segment, then it is not necessary to actually record their addresses; a shared page pointer protected by a semaphore is sufficient, incremented of the page size after each allocation of an empty page.

The WRITTEN pool holds the partially filled pages in state Written. Totally filled pages in state Written need not be recorded since they will never be reallocated. A semaphore protecting a linked list of page addresses is a suitable encoding.

The next three pools hold pages in state Copy. The STACK pool holds all unscanned stack pages (which can be thought of as pages in state Copy); the COPYP pool holds partially filled pages and the COPY pool holds full pages. Implementation of the access exception handler and of the page allocation routines will be made easier if, first, we have the ability of removing a page identified by its address from these pools, and, second, if the three pools are protected as a whole from concurrent updates. These pools may be represented by double-linked lists, and a semaphore, requested by `Handle`, `allocPageS`, `allocPageB` and `freePageB` should protect access to all three pools.

In the handler routine for access exception, it is necessary to check if a page is in state Copy, before scanning it. To avoid scanning the pools then, it is convenient to keep that information in a page table. This page table must also record some additional bookkeeping information which is summarized now.

4.2 The page table

The page table must be shared by all processes. What need to be recorded for operation of the algorithms is a Scanned/Unscanned bit for each stack page, a lock bit for each heap page, plus, for each page in *toSpace*:

- A Copy/Written bit.
- A crossing information. Its minimum size is that of a few bits specifying a scanning method, plus the number of bits for recording at most the page size (in words).
- The next available address slot on the page (for reallocations, scan, etc.); need to be large enough to hold the page size (in words).
- Room for encoding the pools discussed above as linked lists.

Let us assume that each page holds 4K bytes and that *toSpace* holds at most 16M bytes (that should cover the needs for most applications). Then 64 bits per page of *toSpace* are largely sufficient for storing the bookkeeping information. That amounts to 0.2 percent of *toSpace*; a fairly reasonable amount. In addition, a bit table, with one bit per page, is required for the stack area.

4.3 Page allocators and protection violation handler

Procedure `allocPageT` will first try to reuse a partially filled page from pool WRITTEN, if any is available and if the size requested is not greater than threshold `h`. Otherwise, it will attempt to allocate a range of pages from pool EMPTY. `freePageT` sets the state of page **T** to `Written` and, if the page is only partially filled, records it in the pool WRITTEN. The other effects of `allocPageT` and `freePageT` have been discussed in section 3.

Procedure `allocPageS` repeatedly tries to find an unscanned page until either one is available, or no more scan is required, either because end of collection has been detected, or because no incremental scan is currently required. In each attempt, it first tries to allocate an unscanned stack page (from bottom to top) from the STACK pool, then it attempts to use a

page from pool COPY, and finally a page from pool COPYP. While searching for a page in these pools, it requests the semaphore protecting these pools. Procedure `freePageS` sets the state of page S to `Written`, and saves it into the pool WRITTEN if partially filled.

`allocPageS` is also in charge of detecting end of collection. A shared `Unscanned` counter is maintained to detect end of collection. This counter records the number of pages allocated for copies of cells which are still unscanned. This counter is initially set to the number of unscanned stack pages after flip; it is incremented every time a page is taken from EMPTY by `allocPageB`, and decremented every time a page is released by `freePageS`. Collection is complete when this counter reaches 0. Obviously, this counter must be protected from concurrent updates.

The scan credit mechanism is easily implemented with two counters, one being shared and protected by a semaphore. Both credits are initially 0, and reset to 0 by flip. The local credit is decremented by `freePageT` of the amount of words allocated on the page being freed since it was last allocated (multiplied by the incremental scan parameter K), and incremented by `freePageS` of the amount of words on the page freed. The shared credit is incremented by `freePageS` when called from the `background` procedure invoked when processors are idling.

Procedure `allocPageB` first checks if the size requested fits into the space available on the page S being scanned. If this is the case, then it is this page which is also allocated for copying, and B is set to the first available address on that page. Otherwise, `allocPageB` tries to allocate either a partially filled page in state `Copy`, from pool COPYP, if any, or a range of pages from pool EMPTY. If pages B and S are distinct, then procedure `freePageB` detaches the current B page, sets its state to Copy, stores it in the COPY pool (if full), or COPYP pool (if partially filled), and finally unlocks it.

The handler for access exceptions must request the semaphore protecting the three Copy pools before attempting to scan the faulty page. It then locks the page and checks if its Copy bit is still set. If this is the case, it removes the page from the pool it lied into. The Copy pool semaphore is then released; the page is scanned, if required, and is unlocked.

4.4 Flip

All processes must flip synchronously. There may be several processes simultaneously noticing the necessity of a flip; multiple simultaneous flip initiations must be prevented. It may also happen that some processes do not locally notice the necessity of a flip (e.g. because these processes currently do not allocate data in the heap); so a mechanism is needed to make all processes aware of the necessity of a flip.

Selection of an initiator is easily implemented with the help of a semaphore. Among the potential initiators, the first that could acquire the resource broadcasts a signal to the other processors and waits for all of them to reply. It then releases the semaphore and broadcasts another signal, enabling local flips on all processors. This solution is basically similar to that used for Mul-T [KHM89].

A number of variables must also be initialized at flip time. These include the `Unscanned` counter, the shared and local credit counters, the current pointer T (needs to be allocated for non-initiators; the initiator does it from the `Allocate` procedure), and the in-

cremental scan parameter K, which must be adjusted after each flip since it depends upon the total stack sizes at flip time. We can assume that the initiator takes care of initialization of the shared counters, in addition to the initialization of its local data.

5 Providing Garbage Collection with a Distributed Shared Virtual Memory service

5.1 Goals

This section addresses the implementation of the algorithm on distributed memory architectures. The concept of Distributed Shared Virtual Memory (DSVM for short) [LH89][CBZ91] provides a shared memory abstraction for a physically distributed memory architecture. A first approach would be to implement the algorithms discussed in sections 3 and 4 on top of a DSVM service, since it supplies the required abstraction (assuming some implementation of semaphores is available).

An alternative, hopefully yielding better performances by reducing the page traffic, is to provide a garbage collection service as an added functionality to a DSVM service, rather than built on top of it. This section specifically investigates this issue. The abstract procedures discussed in section 3 need not be altered, only the implementation of back-end routines should. Page management for the allocation and garbage collection service will be integrated, in some sense, with the page management required for providing the DSVM service.

We will assume our application to run on a number of (virtual) processors, or "sites". All processors virtually share a single address space through a DSVM service. As before, each processor owns part of the roots of the application and shares with the others all cells in the heap, and possibly its stack pages. The management of control information will rely here on message passing.

5.2 Page managers

The allocation/collection algorithms require to maintain some bookkeeping information for pages (crossing information, state information, etc.). These information cannot be saved within the pages, since that would break the required continuity of address space across pages. The solution taken is to partition the management of state information for pages between the processors, the local bookkeeping information being kept in some specific area on each processor. We will furtner assume that each processor is statically assigned a range of pages to manage, and that each processor is aware of the distribution of page management.

This strategy is exactly Kai Li's Fixed Distributed Manager strategy for providing the distributed shared memory service [LH89]. Upon receiving an access request for a page it manages, a processor takes the adequate decision, according to the state of that particular page, and then, once access is enabled for that page, the page is transferred to the requester.

5.3 Allocation of pages for new cells

Locally, the processors must organize the information concerning the pages they handle; the method proposed in section 4, with a page table and page pools, is still adequate here. In order to minimize the communication traffic, the processors preferably allocate for their own usage the pages they manage. In order to allow all pages to be allocated, the processors that manage pages that may be allocated for storing new cells (or free pages, for short), are organized as a virtual ring; a distributed algorithm takes care of allocation of these pages to the whole set of processors. Initially, all sites are in the ring, and the successor of each site is given by some predefined ordering known by all sites.

Every site records the remote site that replied to its last request for a free page, initially set to the next site in the ring; this site is referred to as its *remote free page allocator* in the sequel. A site receiving an allocation request for a free page will handle it as if it was a local request. If the request can be satisfied locally, then a range of addresses is returned to the requester. Otherwise, two cases may occur: either the request may still be satisfied by another site in the ring, in this case the processor transmits the request to its own *remote free page allocator*; or no site in the ring may satisfy the request, in this case, a failure message is sent to the requester, which will initiate a flip upon reception of the message.

If a demand is satisfiable, then it must be satisfied before it realizes a complete turn of the ring of sites managing free pages. Any site not managing any free page will never manage any in the future; sites may only leave the ring. The condition is checked as follows: each site visited by a request compares the identifier of the requesting site (say i) with its own identifier (say j) and that of its *remote free page allocator* (say k). The previous condition, and the way remote free page allocators are maintained, imply that no site between j and k (in the initial ring) manages free pages, thus, the request cannot be satisfied if the requester stands between j and k (in this ordering).

Upon a successful reply to its request for free pages, a processor updates its *remote free page allocator* to be the site that satisfied its request, except in the infrequent case of a multi-pages request. In this last case, the *remote free page allocator* is not updated since the request may have visited sites that manage free pages, though none of them could cover its demand.

Freeing a page in state Written consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in sections 3 and 4.

5.4 Allocation of pages for scan

For incremental scan, the processors must be able to allocate the whole set of pages that require scan. Conversely to the case of free pages, or of pages for copies, it is not sufficient here to organize the sites managing pages to be scanned as a ring. The problem is that the number of pages to be scanned managed on a site does not monotonically decrease. As long as a processor manages pages in state Empty, these may be allocated for copying cells and consequently become pages to be scanned. Using a particular allocation strategy for allocating pages for copies, and with some additional constraints on cell sizes, the ring structure could still be used, as shown in [LB91], but these constraints are not assumed here.

The sites managing pages to be scanned will be organized as a virtual distributed queue. Initially, all sites agree on the first and the last site in the queue (these can be statically determined, or dynamically, for instance at flip time). The algorithms guarantee that any request for a page to be scanned either will reach the first site in the queue, which will reply to the request, or, if the queue is empty, will reach a site that detected end of collection, which will propagate this information to the requester.

An implementation of the algorithm is feasible using four variables per site. Each site records the last site having provided to it a page to be scanned, that's its *remote allocator for pages to scan*. In addition, each site in the queue maintains its *successor* site in the queue (initially the next site according to some predefined ordering, or itself if the site is the last in the queue). Each site also maintains a variable holding the site it believes to be the last in the queue (the *last* for short), and, finally, a flag indicating if the site has detected end of collection. The insertion and removal protocols for maintaining the queue guarantee that, by transitivity of the *remote allocator for pages to scan* references (resp. *last* references), the first (resp. the last) site in the queue is actually reached. Further, they guarantee that the *successor* variable on each site either holds an indication that the site is not in the queue, or, if that site is in the queue, holds its successor in the queue (itself if it is the last in the queue). If the queue is empty, then the last known site in the queue necessarily detected end of collection, and has its specific flag set.

For the same reasons than for allocation of free pages, each site preferably allocates for itself the pages to be scanned that it manages. To help detection of end of collection, each site locally maintains an `Unscanned` counter, handled similarly to the `Unscanned` counter discussed in section 4, but relative to the pages it manages. This counter maintains the number of pages allocated for copies (local or remote) the scan of which did not terminate yet. A request for a page to be scanned (local or remote) is delayed by a site until either a page to be scanned is locally available, or the `Unscanned` counter reaches zero.

When a request cannot be locally satisfied, the site, if not in the queue or if not the first in the queue, transmits the request to its own *remote allocator for pages to scan*, which handles the request as if it was locally issued. Any site visited which is aware of end of collection replies negatively to the requester, making it aware of end of collection. If the receiver of a request is currently the first in the queue, then, either it currently manages some pages to be scanned, in which case it sends one to the requester, or it does not manage any of them, in which case it leaves the queue and then transmits the request to its *successor* in the queue. Furthermore, if the receiving site is also the last in the queue and does not manage any page to be scanned, then collection is complete; the site sets its end-of-collection flag, replies negatively to the request, making the requester aware of end of collection, and leaves the queue (which becomes empty). A site enters the queue, following the last in the queue, when not currently in the queue and acquiring the management of a page holding copies (see 5.5).

This solution may appear rather "centralized"; a queue is actually the most natural structure here, due to the non-monotonicity of the number of pages to be scanned on each site. In addition, it should be noted that the solution proposed realizes a balancing of pages to be scanned over the sites. Acquiring a page to be scanned from the queue may lead to copy some

cells which, as long as the requesting site can provide pages for these copies, will be locally copied. This treatment has thus the potential effect of locally creating pages to be scanned.

Freeing a scanned page consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in section 3 and 4.

5.5 Allocation of pages for copies

The allocation of pages for copies uses an algorithm similar to the one used for allocation of free pages. However, it is necessary that these two virtual rings are distinct, since the local allocation strategies for free pages and pages for copies differ (for copies, one first tries to allocate a partially filled page in state Copy, rather than Written). Each site maintains a *remote allocator for pages for copies*, managed similarly to the *remote allocator for free pages.*

The algorithm for detection of end of collection assumes that collection is complete if and only if all local `Unscanned` counters hold the value 0. In order to maintain the property, a site able to reply to a request for a page for copies, for itself or for another site, must join the queue of sites managing pages to be scanned (if not in the queue yet) before returning the required range of pages to the requester.

Feeing a page holding copies consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in section 3 and 4.

5.6 Page locks and page protection violations

The pages in states S-allocated or B-allocated, as well as the pages holding cells being copied, must be locked to enforce consistency of their content. Instead of using a lock bit here, as was done in section 4, it is more efficient, considering that the shared memory is obtained through exchanges of pages, to request these pages to their manager with exclusive write access. The protection will be relaxed when the page is freed by the processor to which it was granted. It is the manager of the page which will enforce the exclusive access and make all requests for that page wait until the page is freed. Implementation of this primitive should not require any additional effort, since it is also required for implementing the DSVM service.

When an access violation is trapped by a processor, it must not necessarily scan the page. If the same distribution of the page management task is taken for both the DSVM service and the collection service, then handling access traps in the mutation process does not require any additional message. In both cases, it is enough, when trapping a protection violation, to ask the manager of the page for the action to be taken.

5.7 Distributing the scan credit mechanism

As in section 4, a local credit counter is maintained by each processor. The shared credit counter is here distributed over the processors; the actual shared credit is the sum of these locally maintained shared credits. The following discipline is adopted for maintaining the shared credit counters:

- When an idle processor scans for another, it decrements the remote shared counter.
- An idle processor first transfers its credits (local and shared) with its first request for a page to be scanned, since the credits cannot be used locally.

- When using the shared credit, a processor does not use the full available credit (which is distributed over the processors), but uses the shared credits of the processors it visited for finding a free page, and, if scanning is still necessary, the shared credit of all processors visited for finding a page to be scanned.

With this method, the distribution of the shared credit mechanism does not require any additional message.

5.8 Flip

The necessary synchronization of all processors preceding the flip requires a distributed synchronization algorithm. This algorithm must prevent from multiple, simultaneous, flips, by electing an initiator among the sites that noticed the necessity of flip, and must ensure that all sites flip. An ad hoc algorithm, requiring a pause on each processor corresponding to the time necessary for transmitting four messages, is discussed in [LeS92], to which the reader is redirected for details.

6 Discussion and Experiments

6.1 Enhancements

We assumed so far a fixed number of threads, each alternatively acting as a mutator or as a collector. Allowing a varying number of threads, instead of a fixed number, may be convenient for implementation of programming languages in which programs typically use fewer threads than LCS, with longer life expectancy. This may also be convenient for dynamically adapting the execution of LCS programs on the parallel virtual machine to the physical resources available. Augmenting, or reducing, the number of threads in the shared memory case (section 4) does not imply a large work, the main task for an entering process is to acquire a local context from one of the other threads; similarly, a leaving process must transmit its root data to one of the remaining processes. This can be achieved through some load balancing mechanism. In the distributed case, in addition, the storage managed by a leaving process has to be redistributed over the other processes, and an entering process must acquire from the others some storage to manage, unless the storage is simultaneously updated. In any case, all sites must be made aware of introductions or deletions of sites, and a reconfiguration of the page tables and distributed algorithms is required.

Dynamic extension of the heap size and/or stack size is another desirable enhancement of the basic algorithms.

To avoid having to move the contents of the heap, heap size adjustment would typically occur at flip-time, before the local flips take place. At that time, the content of *fromSpace* is irrelevant and that area can be replaced by another, of the required size, possibly at another place in the virtual space. Just before the next flip, the current *fromSpace* will be adjusted accordingly, to match the size of the current *toSpace* (or according to some heuristics for heap size dynamic adjustment). Besides moving an (empty) area, the adjustment also consists of updating the page tables accordingly.

Dynamically extending the stack should not be difficult too, provided the stack does not hold any reference to itself (that hypothesis was assumed in section 3). If the stack segment cannot be grown in place, then it must be detached and reattached at a place where it can be grown. As for the heap, an update of the page tables is required.

An enhancement of heap allocation that does not appear feasible at that time, unless taking strong limitations on the size of cells, is to use page protection traps for avoiding the end-of-page test required in routine `Allocate`, similarly to the solution proposed in [App88]. The problem here is that the heap may contain interleaved attached and detached pages, or ranges of pages. Consequently, the fact that some address is enabled (resp. not enabled) at a given distance from pointer T in some process does not imply that all addresses in between are enabled (resp. not enabled).

Other desirable enhancements include implementing a generational collector on this ground. This has to be investigated, but we cannot foresee any major reasons preventing from adding generational capabilities to our collector.

6.2 Experiments

The incremental scan, plus the use of page protection traps for enforcing the scan of pages holding copies, should provide a low latency collection mechanism with an acceptable loss of performances compared to a stop and copy collector. This has been confirmed by the experiments.

A version of the (sequential) LCS virtual machine equipped with an incremental collector based upon the algorithms given in section 3 was prototyped, and its performances were compared with those of the currently available implementation of LCS, which uses a stop and copy-depth-first collector, and with those of a version using a stop and copy-bread-first collector. The different versions were compared on various benchmarks; the table in figure 3 below shows the results for a benchmark consisting of running an implementation of the Knuth-Bendix rewrite-rule completion algorithm on an example set of rules. All collection algorithms were implemented in C; all were run with a heap-size of 16MB (8MB per space); all runs required 58 flips.

	s/c b-first	s/c d-first	incr 8kB	incr 64kB
Collection time/Total time (%)	19.7	26.3	24.3	22.1
Relative mutation time	1	0.97	1.14	1.11
Relative collection time	1	1.42	1.50	1.28
Relative total time	1	1.07	1.21	1.14

Figure 3. Performances of stop and copy and incremental versions

For this benchmark, an overhead of 21% in total mutation+collection time was observed for the incremental version, compared to the stop and copy-breadth-first version. For reasons which will not be detailed here, the current implementations of LCS use a depth-first variant of the stop-and-copy collection algorithm. The depth-first version is slightly faster than the breadth-first version, for mutation, but significantly slower for collection (the algorithm uses

a pointer-reversal technique for achieving a depth-first copy of the active cells). Compared to the latter version, the overhead of the incremental algorithm decreases to 13% in total time.

The 21% overhead observed for the incremental version, versus the stop and copy-breadth-first version may seem high, compared to the 4% overhead claimed in [ELA88]; this figure requires some comments.

- First, the stop-and-copy versions have been used for several years and have been carefully optimized, which was not the case for the incremental version. The overhead should be slightly reduced by a careful optimization of the incremental version.
- The Collection time/Total time ratio observed for this benchmark is rather high; it was more often below 10% in the other benchmarks we tried. Considering that the overhead is greater for collection than for mutation, the typical total overhead should be lower.
- Another factor that influences the overhead is the ratio between words scanned on access faults and those scanned by incremental scan; again, this ratio was rather high here. Not surprisingly, it has been observed that the overhead decreases with that ratio. This stresses the need for a careful implementation of the access handler routine and of the page protection mechanisms, both at the application level and at operating system level.
- Finally, we exercised the incremental version for several page sizes, ranging from 8kB to 64kB. The smaller number of access faults on heap pages resulting from larger pages makes the overhead decrease when the page size increases. But, obviously, using large pages affects responsiveness of the applications.

As a conclusion, the typical overhead of the sequential incremental version, compared to the fastest stop-and-copy version, can be expected to be around 10%. This may be considered an acceptable overhead, considering the benefits of the incremental version with respect to responsiveness.

The scan credit mechanism, and the global page allocation, should allow processes to effectively cooperate for both collection and mutation. A simplified version of the parallel collection algorithm has been prototyped to run on a stock workstation running an SVR4 based operating system, and integrated in a preliminary version of the parallel virtual machine for LCS. The virtual machine is constituted of a number of processes (typically four to sixteen), running on distinct processors when allowed by the hardware. The effect of lazy scanning of the heap due to the credit mechanism could be precisely observed: processes lacking of work (including those waiting for I/O operations to complete) perform some scan for themselves, and then for the other processes. This is particularly interesting for interactive applications; most of the scan work is then done while the user is typing commands, with less overhead on the computations themselves.

Unfortunately, no fair performance figures can be provided yet for the parallel implementation. The task of properly integrating the memory management layer with the other components of the virtual machine is still in progress. However, an additional overhead is to be expected, compared to the sequential incremental version, due to the locks required for enforcing consistency of pages, in the shared memory version, or to the latency of page transfers, in the distributed version. With several processors running a single-threaded application, we should recover the advantages of parallel collectors; for multi-threaded applications, there

should be an overhead on total mutation time, compared to the same application run on a single processor, but we should also observe an improvement in elapsed time.

References

[App 88] A. W. Appel, Simple Generational Garbage Collection and Fast Allocation, *Software Practice and Experience* 19(2):171--183, February 1988

[Bak 78] H. G. Baker, Jr., List Processing in Real Time on a Serial Computer, *Communications of the ACM*, 21(4), April 1978

[Ber 88] B. Berthomieu, LCS: une implantation de CCS, In A. Arnold, editor, *Troisième colloque C-cube*, Angoulème, France, Décembre 1988

[BDS 91] H-J. Boehm, A. J. Demers, and S. Shenker, Mostly Parallel Garbage Collection, In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989.

[BGG 91] B. Berthomieu, D. Giralt, and J.-P. Gouyon, LCS Users Manual, LAAS Report 91226, CNRS-LAAS, June 1991

[CBZ 91] J. B. Carter, J. K. Bennet, and W. Zwaenepoel, Implementation and Performance of Munin, In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[Coh 81] J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys*, 13(3), September 1981

[ELA 88] J. R. Ellis, K. Li, and A. W. Appel, Real-time Concurrent Collection on Stock Multiprocessors, In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988. Also Digital SRC Research Report number 25

[Hal 84] R. H. Halstead Jr., Implementation of Multilisp: Lisp on a Multiprocessor, In 1984 *ACM Symposium on LISP and Functional Programming*, August 1984

[KHM 89] D. A. Kranz, R. H. Halstead Jr., and E. Mohr, Mul-T: A High-Performance Parallel Lisp, In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989

[LB 91] T. Le Sergent, and B. Berthomieu, Un ramasse miettes distribué incrémental sur une mémoire virtuelle partagée distribuée, LAAS Report 91373, CNRS-LAAS, Novembre 1991.

[LeS 92] T. Le Sergent, Méthodes d'exécution, et machines virtuelles parallèles, pour l'implantation distribuée du langage de programmation LCS, PhD. thesis, 1992 (forthcoming).

[LH 89] K. Li, and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, 7(4):321--359, November 1989

[Mil 80] R. Milner, *A Calculus of Communicating System*, LNCS volume 64, Springer-Verlag, 1980

[MTH 90] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990

[Rud 86] M. Rudalics, Distributed Copying Garbage Collection, In *1986 ACM Conference on LISP and Functional Programming*, August 1986.

Generational Garbage Collection for Lazy Graph Reduction

Julian Seward*
sewardj@uk.ac.man.cs

Abstract

Although the LISP community have been exploiting the fruits of generational garbage collection for some time, little attempt has been made to apply these ideas in lazy functional language implementations. This paper attempts to plug that gap.

The action of overwriting an unevaluated thunk with its final value, known as *updating*, is central to lazy reduction systems. Unfortunately, updating creates pointers from older to younger generations. A simple two-generation scheme which allows heap occupancy to approach 100% is presented. This collector is a hybrid semispace and mark-scan collector. We show that keeping track of old-to-new pointers imposes virtually zero time and space overhead on the mutator. Consequently a net performance gain can be had by using generational collection.

This paper describes how a generational collector was incorporated into a standard G-machine interpreter. Detailed performance measurements presented indicate that a significant improvement in overall performance is achieved, compared to both semispace and compacting mark-scan collectors. Some interesting variants of the basic scheme are discussed. Finally, a possible compiled-code implementation is presented.

Keywords

Garbage collection, Generational, Graph reduction, Lazy, Functional, Updating.

1 Introduction

1.1 The problem

Recent work indicates there is much to be gained by employing a garbage collection strategy which exploits cell lifetimes. Quite a few generational schemes have been suggested and implemented with considerable success. A recent example is Standard ML of New Jersey [App92].

---

* Author's address: Department of Computer Science, Victoria University of Manchester, Oxford Road, Manchester M13 9PL, UK. Tel: +44 61 275 6291 Fax: +44 61 275 6236

Observations show that if a cell survives one garbage collection, it is likely to survive several more. The central principle of any generational collector is to segregate cells which look like they will last a long time, and collect them much less frequently than the rest. When heap occupancy is high, this could be a big win over non-generational schemes.

A key problem is how to deal with pointers from older to newer generations. Such pointers are created in a lazy reduction system when unevaluated expressions, or *thunks*, are overwritten with their final value. This action is known as *updating*. With strict languages like SML and LISP, updating is unnecessary, so old-to-new pointers only appear if assignment is used. This may explain the lack of takeup of generational ideas in the lazy arena.

This paper describes how a generational collector was incorporated into a standard G-machine interpreter [Joh87]. Detailed performance measurements indicate that a significant improvement in overall performance is achieved, compared to both semispace and compacting mark-scan collectors. The mutator is not significantly impeded by the need to detect old-to-new pointers, and heap utilisation may approach 100%.

1.2 Structure of paper

Three main sections discuss theory, results and further work:

- **Section 2** derives a suitable generational collector by merging two well-known non-generational collectors. Next, relevant details of the G-machine implementation in question are examined. A strategy for dealing with updates is defined, and we consider under what circumstances this will work well.

- **Section 3** presents detailed performance results for the generational collector. We also compare its performance to the same G-machine using a semispace collector, and a compacting mark-scan collector. This provides an illuminating insight into the relative strengths and weaknesses of the new collector.

- **Section 4** introduces some optimisations which have not yet been implemented. It also discusses how this collector might be integrated into a compiled-code reduction system which employs "info-table" style cell tags, as implemented in the STG machine [Pey91] and the Chalmers G-machine [Joh87].

2 A generational collector

2.1 A starting point: the semispace collector

Two important properties that a good sequential garbage collector should possess are:

- **Compaction.** It is widely accepted that allocating from a contiguous block is essential for good mutator performance.

- **Efficiency.** Functional language implementations place tremendous demands on their collectors. For example, the Chalmers LML compiler [Aug84] frequently achieves an allocation rate well in excess of a megabyte per second on widely

available workstations. If we demand that garbage collection takes at most 20% of execution time, we imply a *minimum* recovery rate of five times the allocation rate. A few back-of-the-envelope calculations reveal that the recovery rate required begins to approach the instruction rate of the processor. Clearly, efficiency is paramount.

Possibly the most promising candidate is the semispace collector [Che70]. It is very simple to implement. An absolutely crucial property is that collection time depends only on the number of *live* cells in the heap. Consequently we can achieve a recovery rate asymptotically approaching infinity simply by making the heap arbitrarily large[1].

The high recovery rate of Chalmers LML mentioned above is attained by employing just such a collector. Unfortunately, the semispace scheme has three flaws, all of which we now attempt to correct.

1. **Heap occupancy is limited to 50%.** Not only does this cause a serious under-utilisation of a valuable resource, it is also extremely annoying to find that one's program has run out of space when there are megabytes of memory which could be used if only the collector made better use of available resources.

   Proponents of semispace collection have in the past claimed that a virtual memory system alleviates the problem since "the unused semispace is simply paged out, freeing up real memory for the current semispace". Recently, a few voices of dissent have pointed out that this causes large amounts of paging in practice. The author would like to go further and point out that argument is absolutely invalid. For the argument to stand would require disk I/O transfers to operate sustainedly at memory speeds.

   In any case, not everyone has a virtual memory machine.

2. **Old cells are copied repeatly.** This problem is shared with all non-generational compacting schemes. It seems a pity to waste time indiscriminately moving old cells again and again given that we can identify the majority of them at very little cost.

3. **Locality is appalling.** The mutator cyclically visits every cell in the heap before returning to the start. This constitutes worst-case behaviour from the viewpoint of both the cache and the virtual memory system.

This all looks like bad news. Let us restate the advantages of semispace collection:

1. **Speed.** In a sparsely occupied heap the semispace method outperforms all others by a considerable margin.

2. **Simplicity.** Ease of implementation is important.

3. **Space.** No auxiliary data structures are required.

If it were possible to use this collector as the basis of a hybrid system, we might be on to a good thing. It is important to get heap utilisation as high as possible, so we next look at a second collector.

---

[1] In practice, collection time is also proportional to the number of roots, but this effects all implementations equally.

2.2 The compacting mark-scan collector

Such a collector has the advantage of allowing heap utilisation up to 100%. Operation is three-phase, as follows:

1. **Mark the accessible graph.** This involves a recursive traversal, starting from all root pointers.

2. **Compact.** All live cells are slid to one end of the heap, leaving a contiguous free block.

3. **Fix up pointers.** Since all live cells have moved, it is necessary to adjust pointers to cells so as to reflect their new locations. Root pointers are similarly adjusted.

This is expensive. Phase (2) involves a complete scan of the heap even if occupancy is low. So simply adding an arbitrary amount of memory does not necessarily increase collector efficiency in this case. In practice, measurements show that at low occupancy, this collector performs badly compared to the semispace collector. When occupancy gets higher, though, they are more evenly matched.

A naïve implementation requires an auxiliary stack to guide the mark phase. In the worst case, this can be as big as the heap itself. We also need a table in which to record the new locations of cells after phase (2), although in practice it is possible to re-use the mark-stack for this.

At the cost of considerable extra complication, both the mark and fixup phases can be done in constant space. Marking using pointer reversal alleviates the need for a mark stack [Pey87]. Using Jonker's in-place compaction algorithm [Jon79], references to cells are chained together, so the new-address table is eliminated.

2.3 Merging the two

The following combination, suggested in [San91], is a variant of the generational collector employed in SML-NJ [App92]. Appel's collector, like this one, divides the heap into two generations. However, both generations are collected by copying, so occupancy cannot exceed 50%, a serious limitation. We employ the compacting mark-scan scheme to collect the older generation, and thereby allow occupancy arbitrarily close to 100%.

As depicted in Figure 1, the heap is divided into three regions, **OldSpace**, **ToSpace** and **FromSpace**. The latter two are equally sized. Old cells are kept in the section delimited by **HeapStart** and **OldEnd**. New cells are allocated in **FromSpace**, moving towards **HeapEnd**. Eventually **FromSpace** becomes full. A copying collection then moves all live cells in **FromSpace** to **ToSpace**, adding them to the end of **OldSpace**. **OldEnd** is moved along to reflect this, so the cells collected enter the **OldSpace**. Such an event is called a *minor collection*. The remaining space is split again and allocation in the now diminished **FromSpace** resumes.

After some number of minor collections have gone by, **OldEnd** will have advanced past **OldMax**. We then perform a compacting mark-scan collection of the entire heap, that is to say, of **OldSpace**, since a minor collection has just been performed. Hopefully, this *major collection* causes **OldEnd** to retreat considerably towards **HeapStart**, in

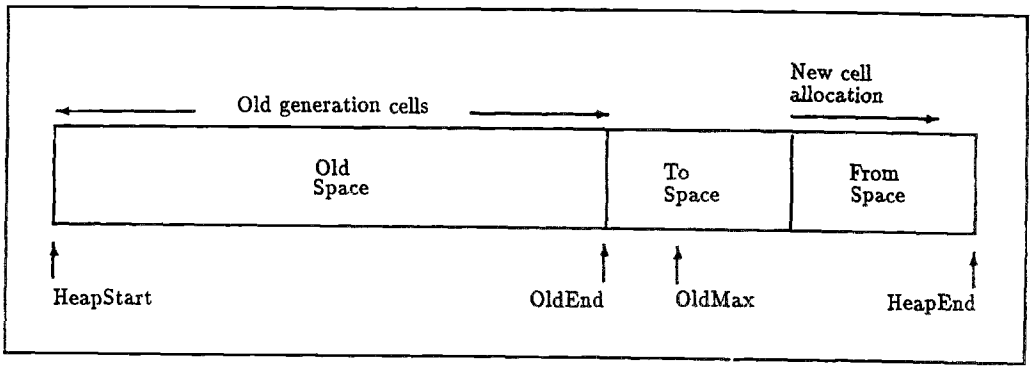


Figure 1: Heap organisation for generational collection

which case we proceed as normal. But if **OldEnd** still exceeds **OldMax**, it is necessary to deem the heap full and abandon execution. Clearly, then, maximum allowable occupancy is defined by **OldMax**. This setting also has a bearing on overall efficiency. Experimentation shows a value of 90% gives acceptable performance.

Alas, there is a problem. When a cell in **OldSpace** is updated, it may acquire pointers to cells in **FromSpace**. During a minor collection, **FromSpace** cells are moved, invalidating such pointers. So it is necessary to keep a record of all such updates, and fix up the **OldSpace** cells after each minor collection. But more than that, such cells must act *as a source of roots* during the minor collection, for it is concievable that a **FromSpace** cell is referred to only from **OldSpace**.

So we need an auxiliary array of pointers to old-space cells which have been updated. Henceforth, this array is referred to as the *forward reference table*. Supposing a cell at location x is updated. Under what conditions is it necessary to put x into the table?

- x must be in **OldSpace**: $x \leq$ **OldEnd**.
- The new contents of x must contain one or more pointers.
- At least one of these pointers should point to **FromSpace**.

The second and third tests are optional. They reduce the required forward reference table size somewhat, but not a lot. When a minor collection is performed, the table acts as a source of roots.

The viability of our scheme depends on

- The required size of the forward reference table.
- How much of an overhead the update tests impose on the mutator.

In the next section we present figures which indicate that for typical programs, at least 97% of all updates are in the new generation. Consequently, if **FromSpace** contains, say, 300000 closures, approximately 10000 words of forward reference table are needed. In the STG machine, all closures contain at least 2 words, from which it is possible to conclude

that the table size is only about 1.5% of overall heap size. Old-space updates occur when there is a significant delay between the creation of a thunk and its updating, because in the delay, the root cell may well be transported into the old generation. Most programs update the majority of their thunks quickly, and thereby cause few old generation updates. However, a degenerate case that has up to 20% of its updates in the old generation has been discovered. This is somewhat alarming, but hopefully it does not occur very often.

We need, therefore, a way to deal with the rare situation where the forward reference table becomes full. In this case the minor collection can be started early, at the cost of a little extra inefficiency.

2.4 Details of implementation

The collector was incorporated into a G-machine interpreter written in Sun Pascal. The system is based closely on the supercombinator language presented in [Pey87]. Importantly, the only data structures supported are lists. Programming in the language is a bit like working with a lazy variant of LISP. All heap cells are the same size, with the following layout:

- **Tag** - 1 byte
- **Mark bit** - 1 byte
- **Left pointer** - 4 bytes
- **Right pointer** - 4 bytes

The equal sizedness of all cells has an important bearing on this paper: all updates are done by copying. There are *no indirection nodes*. Despite this, most of the work described here is also relevant to systems using indirection nodes.

The use of an interpreter to gather timing information is also of some concern. Whether or not the relative timings presented below accurately reflect what would happen in a compiled-code implementation is a matter for debate. Nevertheless, our interpreter does make significant demands on the heap. Running on a Sun-4/330, it averages around 200000 G-codes per second with heap allocation varying from 45000 to 60000 cells per second. This is roughly equivalent to half a megabyte per second, a third of the observed allocation rate of Chalmers LML on the same machine. It seems therefore reasonable to assume the conclusions drawn below are valid for compiled-code implementations too.

3 Performance results

3.1 Test programs

Four test programs were employed.

- **STG**: A STG machine simulator. This 3800 line program parses a Core[2] file, converts it to STG code, then runs a STG machine simulation. The parser and

[2] As discussed in [Pey91], Core is a simple functional language used as an intermediate form in the Glasgow Haskell compiler.

Core-to-STG conversions take a lot of heap. Simulation was kept to a minimum so as to hold heap residency high.

- **TreeGrow**: Binary tree traversal. This program was specifically designed to take a lot of heap and have a mean-to-peak residency ratio approaching 100%. It builds a large binary tree which is subsequently traversed a number of times, counting the number of nodes. To make things more interesting, the tree is inverted before counting. The inversion forces a complete copy to be made. Consequently average residency is about 75% of peak.

- **Primes**: Generates an infinite list of primes using a lazy sieve of Erasthones. This program has a low residency, and is extremely lazy.

- **Queens**: All 92 solutions to the 8 queens problem (generated in an inefficient way, so as to give significant run time). Also relatively low residency.

Timings below were obtained by running each program three times under the relevant conditions and averaging. Unless otherwise stated, measurements were obtained with a Sun-4/330 with 32 megabytes of real memory. Results are organised as follows.

Firstly, variation of the following quantities with heap size is shown:

- Garbage collection time.

- Proportion of updates in the old generation.

- Maximum size of the forward reference table.

Next we demonstrate that the gains from generational collection far outweigh the additional mutator cost, especially when heap residency is high, whilst still giving performance as good as semispace collection when the heap is nearly empty. This is done by running the **TreeGrow** and **Queens** programs using semispace and compacting mark-scan (CMS) collectors as well as with the generational scheme.

The generational scheme presented above has one parameter which can be adjusted: the maximum proportion of the heap that the old generation may occupy. In terms of Figure 1, this is the value of **OldMax**. We show how collection time varies with this quantity and thereby justify the choice of 90% used in all other measurements.

3.2 Generational collector performance

All heap sizes quoted are in cells. For the purposes of comparison, the maximum known residencies of the test programs are shown below. Average mutator time is also shown. Variation of mutator time with heap size is very small.

Program	Maximum Residency	Avg Mutator Time
STG	89000	70.3 ± 0.3
TreeGrow	112000	48.2 ± 0.2
Primes	20800	329.5 ± 2.5
Queens	22500	616.0 ± 8.2

3.2.1 Collection Time vs Heap Size

Collection time is presented as a percentage of average mutator time.

Heap Size	STG	TreeGrow	Primes	Queens
100000	10.9	-	16.3	2.08
120000	7.17	-	14.8	1.87
140000	5.94	9.19	13.7	1.84
160000	5.34	10.1	12.9	1.64
180000	5.01	11.2	12.4	1.50
200000	4.97	7.23	11.5	1.50
240000	5.05	8.26	10.6	1.35
280000	4.43	6.20	9.92	1.21
350000	4.40	6.55	9.04	1.20
400000	4.20	4.37	8.56	1.03
500000	3.59	4.34	7.98	1.01

The high-residency cases, **STG** and **TreeGrow** are encouraging. Given that **STG** has a maximum residency of 89000 cells, it is impressive that the collection time is only 11% of mutator time with a 100000 cell heap. Even small increases in heap size cause this figure to fall rapidly.

The **TreeGrow** program shows similar good collection times even when the heap is only marginally larger than the maximum residency. In this case, collection time falls off jerkily as the heap expands. This curious phenomenon merits further investigation.

The **Queens** program does not exercise the heap very much, hence the low collection times. It is doubtful whether a semispace collector could do much better in this case.

Unfortunately the **Primes** program, which also has a moderate residency, runs against these otherwise hopeful results. This is due to the rather unusual dynamic behaviour of the program. Section 3.2.4 discusses the matter further.

3.2.2 Forward Reference Table Size vs Heap Size

The sizes presented are the maximum observed forward reference table size at minor-collection time.

Heap Size	STG	TreeGrow	Primes	Queens
100000	510	-	4418	2009
120000	802	-	4330	2005
140000	903	37	4210	2010
160000	1113	39	4435	1996
180000	675	37	4384	2013
200000	809	36	4314	1991
240000	1193	37	4339	2000
280000	1624	47	4338	1998
350000	2293	40	4493	1997
400000	2733	48	4387	2016
500000	3393	45	4315	1904

The results are curious, but encouraging. For **TreeGrow**, **Primes** and **Queens**, the table size is essentially independant of heap size. Why could this be? A plausible hypothesis is as follows. As the heap expands, the interval between new-space collections increases. Consequently, it becomes more and more likely that updates overwrite new-space rather than old-space cells. On the other hand, since the mutator runs for longer between minor collections, it has the potential to generate more updates and therefore more old-space updates. Let the term *thunk delay* denote the number of reductions which elapse between the creation of a thunk and the updating of it. If we now assume that the thunk delays have a negative exponential distribution, the two phenomena could cancel each other out.

The **STG** results suggest that, despite a few kinks, table size in this case is proportional to heap size. Even in this case, that's quite acceptable. From all the measurements above, the maximum table size is 4493 entries. Hence, table size to heap size is

$$\frac{4493}{500000 \times 2} = 0.45\%$$

This is a conservative estimate, assuming all closures (in a variable-closure-size heap) are 2 words long. As mentioned in Section 2.3, it is not a disaster even if the table does overflow from time to time: we simply have to start the minor collection before **FromSpace** is full.

3.2.3 Old Space Updates as a proportion of Total Updates

This title is not quite accurate. What is actually shown is the percentage of all updates which require an entry in the forward reference table. Recall that such an entry need be made only for those old-space updates for which the overwriting cell contains pointers, one or more of which point to the new generation.

Heap Size	STG	TreeGrow	Primes	Queens
100000	1.56	-	31.3	0.876
120000	1.53	-	27.2	0.799
140000	1.40	0.246	24.0	0.787
160000	1.19	0.235	21.5	0.745
180000	1.10	0.208	19.8	0.692
200000	1.14	0.142	17.7	0.687
240000	1.17	0.140	15.0	0.625
280000	1.06	0.097	13.1	0.594
350000	0.89	0.086	10.6	0.562
400000	0.90	0.068	9.31	0.526
500000	0.81	0.044	7.69	0.492

Once again, the **Primes** result spoils otherwise encouraging news. We look at this in the next section. For the others, it is instructive to see that the maximum proportion of updates entered in the forward reference table is about 1.6%, and in many cases dramatically lower.

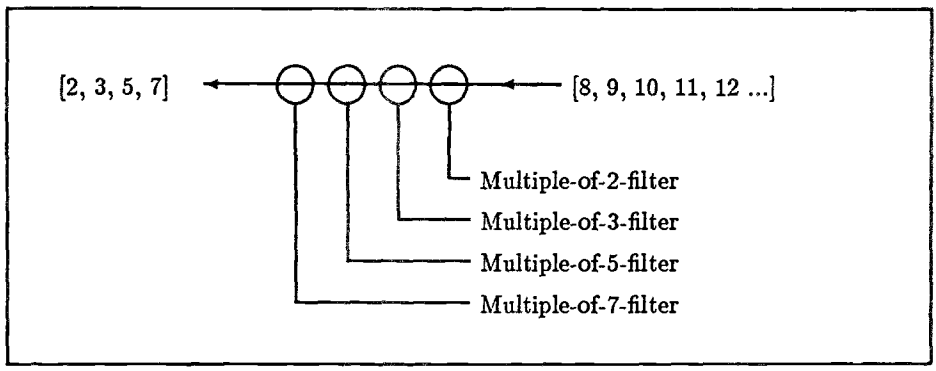


Figure 2: The infinite sieve of Erasthones, immediately after 7 has been discovered prime

Although not listed here, the total number of updates in old-space, whether or not the overwriting cell contains pointers, was also recorded. This indicates that about two-thirds of all old-space updates had to be entered into the forward reference table. Ignoring **Primes**, this means that at worst 97.6% of all updates are *not* in old-space. This is important, because we can detect a non old-space update simply by comparing the updated address with the old-space-end pointer, **OldEnd**. Provided both values are in registers, this imposes a tiny overhead on this 97.6% of all updates.

3.2.4 Effect of dynamic behaviour on performance

Many systems rely on statistical properties of the problem in hand to get good performance in the majority of cases. Freak cases that do not "play along" cause severe performance degradation. For example, virtual memory systems and caches lose effectiveness if the programs being run do not exhibit enough locality. Similarly, if everyone in the country decides to telephone the Prime Minister at the same time, the phone system will virtually collapse.

Unfortunately for us, a generational collector is just such a system. For it to work well, it is necessary for old-space updates to only be a very small proportion of all updates. We observe that the **Primes** program has a large number of old-space updates and hence an inordinately large collection time. An important question is why there should be so many old-space updates.

Figure 2 shows the sieve in progress. The printing mechanism "pushes" a barrage of filters along the candidate list of numbers [8, 9, 10, 11, 12 ...]. Candidates which get through are prime and join the output list [2, 3, 5, 7]. Each new prime also attaches a "is-not-a-multiple-of-me" filter to the barrage. In the example, the next successful candidate, 11, attaches a fifth filter.

After a while, many prime numbers will have been produced, so there is a correspondingly large collection of filters. To get the next prime, the printing mechanism kicks the leftmost filter, which in turn kicks its neighbour. Demand is propagated through all filters, right up to the candidate list. Now, clearly the first few filters, for multiples of 2, 3, 5 and 7, filter out the majority of candidates, so it may be a long time before a hopeful makes it back to the leftmost filters. These have all been awaiting update for a

long time, and may well have been transported to old-generation by a minor collection. Consequently, a heavy burden is imposed on the collector.

So there is at least one program to which generational collection does not respond well. Is that a good reason to abandon this approach? We believe not.

- Although collector performance in this case is not as good as we would like, it is not bad: perhaps two or three times the cost of semispace collection in a moderate-sized heap.

- The other three programs examined respond extremely well to generational collection. In particular, it is significant that the STG program, of considerable complexity (3800 lines of code), works well. It seems a pity to throw away this large average-case performance gain.

- One hopes that most programs update most of their thunks quickly, so average-case behaviour is almost always exhibited.

Examining the distribution of thunk delays (defined in Section 3.2.2) provides more information than merely looking at the average value. It would appear that the effectiveness of generational collection in an environment with updates depends centrally on this distribution. Further work in this area could be very useful.

3.3 Performance relative to non-generational collectors

3.3.1 What measures are of significance?

In this section, we hope to answer the following questions:

- Does generational collection really give significant performance benefits?

- To what extent is the mutator impaired by the need to record old-space updates?

What is the best way to measure collector efficiency? First, observe that for most types of collectors, efficiency can be improved to some extent simply by making the heap larger. Observe also that in many cases, the utility of functional programs is limited not by collection time, but by the size of heap needed to keep collection time down to a reasonable level.

Other workers [San91] have measured GC efficiency in terms of the rate at which free heap is reclaimed. We suggest a more natural measure is to directly relate how heap size and overall run time are related. Results below are phrased in this way.

Two programs, **TreeGrow** and **Queens** were selected as well-behaved representatives with high and low residencies respectively. They were run with a range of heap sizes, using a semispace collector, a compacting mark-scan collector, and the generational collector. In the latter case, old-space maximum size was set to 90% of overall heap size, as for the measurements above.

3.3.2 The TreeGrow program

Average mutator time

Collector	Mutator Time
Semispace	50.0 ± 1.6
CMS	48.0 ± 2.6
Generational	48.2 ± 0.4

Mutator time with each different collector was averaged over all runs. Measurement noise seems to swamp any detectable variation in the generational case.

Overall run time

Heap Size	Semispace	CMS	Generational
130000	(heap overflow)	112.3	57.1
140000	(heap overflow)	97.6	52.9
150000	(heap overflow)	88.5	53.3
160000	(heap overflow)	83.5	53.6
170000	(heap overflow)	79.3	53.8
180000	(heap overflow)	76.3	54.4
190000	(heap overflow)	73.2	51.8
200000	(heap overflow)	71.0	52.1
250000	113.1	-	-
300000	77.6	62.0	51.5
400000	63.6	59.9	50.5
500000	59.4	57.0	50.5
600000	57.6	56.5	50.7
700000	55.0	56.6	50.5
800000	56.2	56.0	50.7
900000	53.2	55.2	50.5
1000000	52.2	57.0	50.4

The generational collector outperforms the other two under at all heap sizes, spectacularly so when the heap is relatively small. Recall that maximum residency of this program is about 112000 cells. It is noteworthy that generational GC time does not improve much once heap size exceeds about 140000 cells. This suggests that residency can get up to about 80% with practically no performance penalty. The second significant observation is that the generational collector outdoes the semispace collector even when the heap is very large, the best-case for semispace collection.

Required heap sizes

Finally we phrase the question the other way round, and ask how much heap is needed to get overall run time down to a given level. Heap sizes are approximate.

Overall time	Semispace	CMS	Generational
65	390000	270000	≤125000
60	490000	400000	125000
55	700000	900000	140000

These figures speak for themselves.

3.3.3 The Queens program

Peak residency of 22500 cells gives extremely low residency at all heap sizes used (22.5% down to 2.25%), so the semispace collector can be expected to do well.

Average mutator time

Collector	Mutator Time
Semispace	621.5 ± 21.3
CMS	614.1 ± 3.4
Generational	617.8 ± 11.8

As before, measurement noise predominates.

Garbage collection time

Because heap occupancy is so small, garbage collection time is similarly small. Presenting overall run times is made rather meaningless by the measurement noise for mutator time, so only the garbage collection time is shown. Observe in many cases how it declines to less than 1% of mutator time.

Heap Size	Semispace	CMS	Generational
100000	38.38	76.94	12.85
200000	17.45	65.30	9.65
300000	11.26	61.52	7.53
400000	7.97	60.35	6.58
500000	7.03	58.23	6.54
600000	5.59	58.19	6.42
700000	5.06	57.27	5.86
800000	4.32	56.74	5.30
900000	3.65	56.20	4.99
1000000	3.07	56.55	4.89

As expected, compacting mark-scan collection does badly because of the low residency. The semispace collector draws ahead of the generational collector as peak occupancy sinks beneath about 4% (corresponding to 562500 cells), but collection time in both cases is so small that this makes little difference.

The conclusion to be drawn here is that the generational collector outperforms the other two once heap residency is above a few percent, and does spectacularly well in the important cases where residency approaches 100%. Also significant is the fact that no significant impairment of mutator performance can be detected in the generational case.

3.4 Old space size as a proportion of total heap size

At any given heap size, there is another parameter to adjust: what proportion of the heap the old-space may occupy. In terms of Figure 1, this is the position of **OldMax**. Since this value defines the absolute maximum heap occupancy allowed, we would like to get it as high as possible consistent with reasonable performance. However, both a very high and very low setting degrade efficiency, since:

- As **OldMax** decreases, the frequency of major collections increases. Major collections are expensive.
- As **OldMax** increases, the frequency of minor collections increases, so the interval between them decreases. This erodes the effectiveness of generational collection, since it decreases the average age of cells moved to the old generation.

  A secondary consideration is that minor collections, although cheap, are not free, especially if there are a lot of roots around (a large G-machine stack or a lot of old-space updates).

This suggests there is some mid-range setting which gives optimal efficiency. Clearly, this depends both on the dynamic properties of the particular program being run, and on the relative speeds of the compacting mark-scan and semispace collectors. We can only hope the former effect does not make much difference, select a "representative" test program and conduct some trial runs to arrive at a value.

Figures below are for the **STG** program[3]. Absolute heap size is held constant at 200000 cells whilst the proportion allocated to the old-generation is varied from 45% to 97.5%. The program has a maximum residency of 89000 cells (44.5% of the heap).

Old-space size	Total GC time
45%	6.22
50%	5.14
55%	4.98
60%	4.78
65%	4.66
70%	4.28
75%	4.39
80%	3.98
85%	4.21
90%	4.31
95%	4.51
97.5%	4.52

[3] Run on a Sun-4c/60, 8 megabytes real memory.

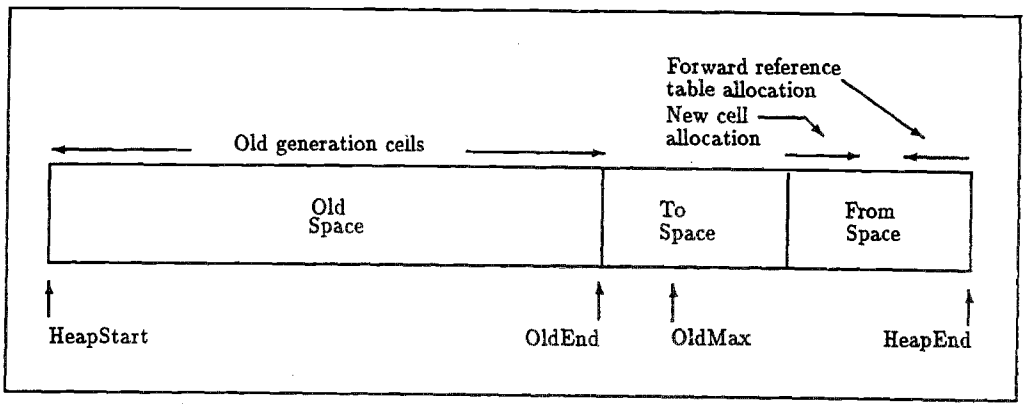


Figure 3: Putting the forward reference table in the heap

Despite a small wiggle, this suggests that any size from 70% to 90% will do, and even going up to 97.5% does little harm. This is good, since it means heap utilisation approaching 100% is quite feasible.

4 Optimisations to the basic scheme

Once it is established that generational collection does not impose an excessive burden on the mutator, a large design space opens up. In this section, some ideas which may improve performance are discussed. None of them have been implemented, though, due to lack of time.

4.1 Putting the forward reference table in the heap

Having a forward reference table which may overflow, or be underutilised, independantly of what's going in the heap, is a nuisance. An obvious place to put the table is right at the top of the heap, as shown in Figure 3. Minor collections now occur when new-cell allocation crashes into the top of the table. This is the usual "two stacks in one array" trick in disguise. The advantage is it gets rid of a separate table. On the other hand, it does make minor collections a little more frequent, with the associated costs discussed in Section 3.4. Hopefully, if the program being run is well-behaved, it will not generate many old-space updates, so the diminution in **FromSpace** size is minimal.

Another, more elegant solution is available for systems which can update using indirection nodes. All old-space updates are done with indirections. Each indirection node contains a field which is used to point to another indirection node, so, as old-space updates occur, a linked list of indirection nodes is built up. This list is, in effect, the forward reference table.

4.2 Making FromSpace larger than ToSpace

The reason we make **ToSpace** the same size as **FromSpace** is to cope with the worst case minor-collection wherein all **FromSpace** cells are live. In practice this rarely happens. The collector is at its most effective when the number of live **FromSpace** cells is minimised. Observations show that typically less than 10% of **FromSpace** cells are live at minor collection time, with many programs getting below 3%. Values above 30% are rare. Andrew Appel quotes a corresponding figure of 1.87% in [App92].

Given that increasing **FromSpace** size increases collector efficiency, it seems a great pity to waste most of **ToSpace**. The Big Idea here is to monitor **FromSpace** residency, and dynamically adjust the relative **ToSpace/FromSpace** ratio accordingly. It appears that **FromSpace** residency changes relatively slowly, so guessing the required next **ToSpace** size, as, say, 50% more than the previous **FromSpace** residency keeps things safe in the face of **FromSpace** residency increases of up to 50% between adjacent minor collections.

Since we clearly cannot *guarantee* anything about residency changes, there must be a way to deal with the case where the live cells of **FromSpace** do not fit into the allocated **ToSpace**. We observe (as in [San91]) that the heap is always in a consistent state during semispace collection. Consequently, if **ToSpace** overflows during a minor collection, the minor collection can be abandoned and the major collector called instead. Given that major collections are expensive, this had better happen only extremely rarely. The key question is how to make a good guess of how big to make **ToSpace** after each minor collection. All manner of clever schemes come to mind: it will be interesting to see how well they perform.

4.3 Multiple minor collections before merging

All the previous variants suffer from an annoying deficiency in that the old generation grows ever larger at every minor collection. This forces the occasional major collection even if none of the live cells collected during minor collections is really destined to become "genuine" old generation data.

It is easy to modify the collector to do multiple minor collections before merging the result of a minor collection onto the old generation. An interesting question is exactly when such a merge should occur. We might stipulate that this happen whenever the semispace residency exceeds, say, 20%. For a program with low residency, this means the system acts perpetually like a semispace collector. Alternatively, the merging could take place every n'th minor collection.

Quite how such a scheme affects performance is unknown. Since the existing implementation works well in the majority of cases, perhaps development should concentrate on improving the worst case, as typified by the **Primes** program.

A word of warning about these complicated schemes is in order. The more parameters which can be twiddled, the smaller is the chance that we will ever arrive at an optimal setting, or even that one setting is optimal for all programs. Building mathematical models of collector-mutator performance may help, but at the end of the day it is often down to time consuming experimentation.

4.4 Compiled code implementation

A popular way to implement fast case-analysis in compiled implementations is by having the tag field of each cell point to a so-called "info table". For every possible action, the info table contains a pointer to the code that performs the action for this type of cell. Further details may be found in [Joh87] and [Pey91].

In [San91], the details of doing both semispace and compacting mark-scan collection using info tables are presented. The only remaining problem is how to create entries in the forward reference table.

When a cell is updated, we need to decide whether to put its address in the table. In the crudest approach, this involves a comparison of the address being updated with **OldEnd**, followed if necessary by a call to a routine `EnterIntoFPTable` which enters this address into the table. In the following C code, the address being updated is in `UpdAddr`. The code then is:

```
if ( UpdAddr <= OldEnd ) EnterIntoFPTable ();
```

Statistics from Section 3 show the vast majority of updates fail the test. Assuming `UpdAddr` and **OldEnd** are in registers, as explained in [Pey91], the overhead for most updates could be as low as two or three instructions.

A more refined approach only inserts an old-space updated cell into the forward reference table if it contains pointers and one or more of these points to new-space. What we need here is to invent a new method, which:

- Does nothing if the cell contains no pointers.
- Does nothing if the cell contains pointers, but they all point to old-space.
- Otherwise, adds the address of the cell to the forward reference table.

To achieve this, it is necessary to allocate a new info table slot for this action, and write code to perform it for every kind of cell. Since these pieces of code "know" the layout of the cells they operate on, this is a fairly cheap operation. Supposing that `EnterIntoFPTableDISP` is the displacement for this method in info tables, our code might now look like this (neglecting typecasting):

```
if ( UpdAddr <= OldEnd )
      (* ((*UpdAddr) + EnterIntoFPTableDISP)) ();
```

In English, this means: "if `UpdAddr` is in **OldSpace**, call the `EnterIntoFPTable` method for the cell pointed to by `UpdAddr`".

5 Conclusions

This paper provides strong evidence that generational garbage collection is viable for lazy reduction systems. The small added mutator costs are greatly outweighed by better garbage collector performance. We showed how the central question of recording old-space updates can be dealt with at very little space and time cost.

It is also, unfortunately, apparent that generational collection makes certain assumptions about statistical-level dynamic behaviour of programs. The primary factor here is the distribution of delays between the creation of a thunk and its updating. It may be that these have a negative exponential distribution. Generational collection works well when most such delays are very small. In the few cases where these assumptions do not hold, performance will not be as favourable. However, there is a strong argument to be made that for the vast majority of programs these assumptions are valid. Significantly, a 3800 line program runs very well with this collector.

Finally, some modifications were suggested. Whether or not these are a good idea awaits further work.

6 Acknowledgements

A special thank-you to David Rushall, who implemented large sections of our G-machine interpreter **LazySu** on which this work is based. Dave also wrote much of the STG machine simulator used as a benchmark, and spent many hours debugging and testing.

Financial support was provided by the Science and Engineering Research Council.

Bibliography

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, August 1984.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13:677–678, November 1970.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[Jon79] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9:26–30, July 1979.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[Pey91] S.L. Peyton Jones. The spineless tagless g-machine: second attempt. Publication forthcoming, 1991.

[San91] Patrick M. Sansom. Dual-mode garbage collection. In *Proceedings of the 1991 Glasgow Functional Programming Workshop*, 1991.

A Conservative Garbage Collector with Ambiguous Roots for Static Typechecking Languages

Emmanuel CHAILLOUX[1,2]

[1] LIENS (URA 1327) : Laboratoire d'Informatique de l'École Normale Supérieure
Address: 45 rue d'Ulm, 75230 Paris Cedex 05, France.
Electronic mail: Emmanuel.Chailloux@ens.fr
[2] LITP (URA 248) : Laboratoire d'Informatique Théorique et Programmation
Institut Blaise Pascal - 4, place Jussieu - UPMC - 75252 Paris Cedex 05, France.

Abstract. In a static typechecking language, such as ML, the type information produced by the typechecker can be forgotten during execution. But in many cases, a minimal type information (tag) is needed for the Garbage Collector (GC). In this paper, I propose a simple, safe and efficient GC algorithm which does not use any tags to distinguish immediate values and pointers.
This GC is a conservative `Mark&Sweep` (which does not move objects) with ambiguous roots (there is a possibility of ambiguity between immediate values and pointers). It is used for a runtime library added to C for an ML compiler to C (CeML) where basic data types are identical to those in C (int, float). However the GC uses a disambiguating strategy which is shown to be safe. It can be used also for other polymorphic languages with static typechecking and uniform data representation.

Introduction

CeML [6] is a new ML dialect (derived from CAML [18]) which differs mainly by its typechecker and its module system. The CeML typechecker includes a new functional type constructor which indicates the function arity. Its compilation model uses the C language as a portable assembly language. Because of some characteristics of functional languages, the C language is not the best target language because it has no exception handler and no memory management. It is thus necessary to add a runtime library which includes a memory management, a mechanism for total and partial applications and exceptions. This paper presents only the memory management of this runtime library (completely described in [5],[6]).

The main goals of this ML implementation are :

- to be as efficient as C when ML programs are written in imperative style and to be as efficient as best ML implementation for functional style programs.
- to be portable : the C generated programs must be running on different computers.
- to be interoperable : the C generated programs can be merged with other safe generated C programs from other compilers.

For this purpose, the runtime library is driven by these constraints, in particular by the memory management. First, I shall present these constraints and their implications for the implementation. I shall then describe a **Mark&Sweep** [13] with ambiguous roots [4] [3] and its distinguishing algorithm between basic values and pointers, in comparison with another GC (`Stop&Copy` [14] with tags). This will lead to a final discussion of the use of tag objects and to a comparison with other GC without tags and other ML implementations.

1 Constraints for the CeML Implementation

To each CeML variable the CeML code generator associates a C variable, and to each CeML function a C function. With its more informative typechecker the C functions also have an arity corresponding to the CeML arity detected during typechecking. The closure environment is given to a C function as supplementary arguments by λ-lifting [9](because there are no local functions in C). It seems interesting to use the C calling protocol and the same data representation as C for CeML basic data types. With the direct mapping between ML and C variables, it is preferable not to move objects, in particular the variables during GC, because in this case it is necessary to push pattern matching variables into the root set. For example, for the following function :

```
let rec sum_list = function [] -> 0 | a::l -> a+(sum_list l);;
```

there are two cases which depend on whether the GC moves objects or not :

- if objects are not moved, then only the argument of **sum_list** function is pushed;
- if objects are moved, there are two pushes for **a** and **l** in the second case of the pattern matching.

From this, I obtain the following constraints :

- data representation is uniform (each value uses 32 bits) :
  - basic data types are the same as in C : **int** and **float**,
  - structured values are represented by a pointer (also 32 bits);
- C calling protocol is used;
- objects do not move;
- root set must be independent from the C stack;
- C functions manage the trace of their arguments and their local variables into the root set.

These constraints are interesting because immediate values are not tagged (this is not necessary in ML [1]), and they allow the direct use of the C functions particularly the arithmetic operators. Because ML is a polymorphic language, the static typechecking is not sufficient to distinguish the immediate values and pointers at compile time. So, the root set can contain immediate values and it is necessary to make a distinction between immediate values and pointers.
The following GC is a conservative GC with uniform data representation without tags for immediate values.

2 Memory Management

Here, I present a new **Mark&Sweep** algorithm with ambiguous roots. I describe the data representation, the partitioning of memory, the root set and the algorithm used during the **Mark** phase to distinguish immediate values and pointers.

2.1 data type representation

All objects have a uniform representation (e.g. 32 bits). In order to distinguish pair, list and other concrete types, a type field is necessary. Records, vectors, strings and closures are considered to be different vectors and need two fields : one for the number of elements and the other for the type (cf. figure 1).

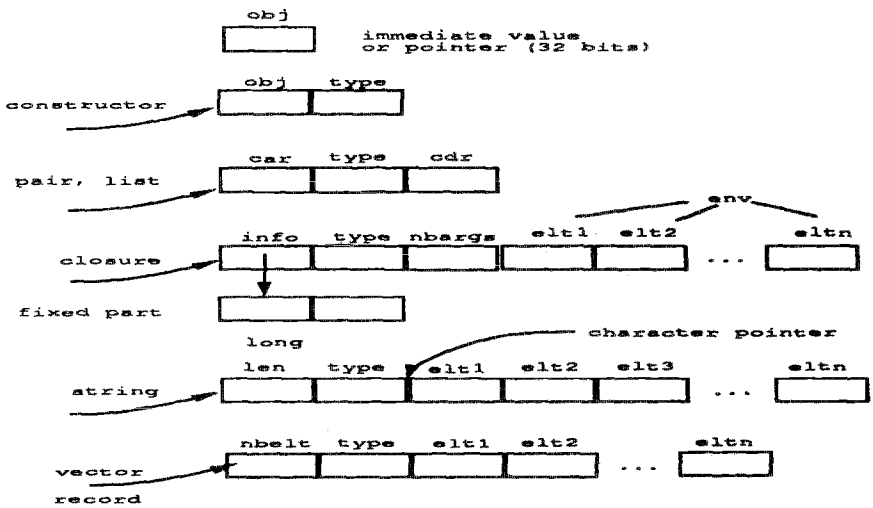


Fig. 1. Data type representation

Integers and floating point numbers use a word as well. Double precision floating point numbers are not implemented, but they can be represented as a pointer toward a four word storage. Records and vectors have the same representation. Strings are a special kind of vectors. Each element (word) contains four characters. The address of the first element corresponds to the C character pointer (**char** *) of the string. Closures are represented by a fixed field, which contains arity of the C function, and a variable part including its environment values.

2.2 partitioning of memory

The heap is partitioned into chunks (cf. figure 2). Each chunk contains objects of the same size (powers of two). There are *nbzones* sets of chunks called zone (from 2^3 to

$2^{2+nbzones}$ bytes). Objects greater than one chunk are arranged into several chunks. This partitioning of memory is a variant of the **BIBOP** (Big Bag Of Pages [16]) algorithm. For our implementation, the chunk size is four kilobytes and *nbzones* is equal to ten.

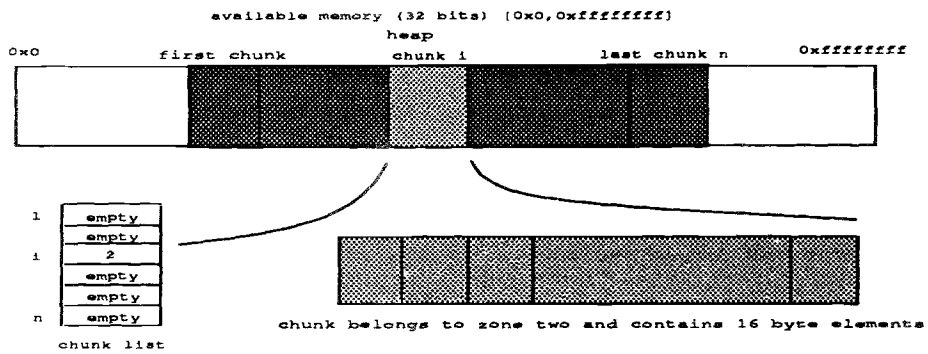


Fig. 2. Partitioning memory

2.3 free lists

Each predefined zone has a list of available elements (cf. figure 3).

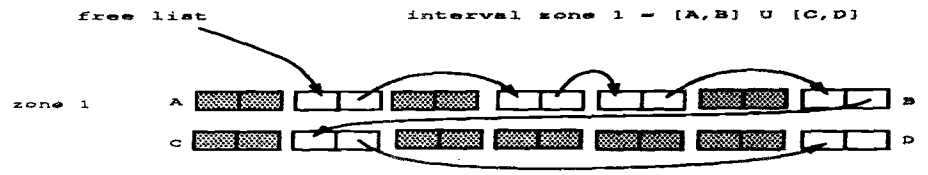


Fig. 3. Free Lists

2.4 root set

The root set is represented by a statically allocated independent stack. This stack allows for the preservation of immediate values or pointers. Sometimes there is a double use between this stack and the C stack, but in order to be independent of the C stack this representation is necessary. This feature preserves programs against the C optimizations which can move out objects from the C stack. Each C function

needs to trace its arguments and its local variables into root set when it is necessary (when the typechecker can not determine if an argument is a immediate value).

This stack is also used for the general apply mechanism (when a direct call to a C function is not possible because the argument is a closure, or during a partial application).

2.5 initial memory state

At the beginning, the heap is empty. Each zone can grow dynamically. The first allocation for a zone is ten chunks. The following allocations are computed by the growing function, after the `Sweep` phase. This orientation allows for a control of the heap evolution.

2.6 allocation

There are two kinds of object allocation.

The first one is used for small objects less than one chunk in size. This first case has two alternatives. If the object size is known (for example one *cons* uses four words) and the corresponding free list is not empty, then the allocation is completed; if not, a GC is invoked. If the object size is not known, then the zone to be used has to be computed.

The second one is used for big objects. If the object size is greater than a chunk, then the object uses several contiguous chunks.

2.7 recovery

When a zone is full, one must recover some space. There are two phases. The first phase (`Mark`) marks each object indicated by the root set, and the second (`Sweep`) preserves only these objects.

Mark. For each value inside the root set, a discriminating algorithm distinguishes between an immediate value and a pointer. In this last case, the structured object is marked and the process is applied to its structure elements. Actually, this algorithm uses the C stack, but it can change by using pointer reversal as described in [10].

Sweep. For each chunk in use, its corresponding free list is updated by all the unmarked elements. This algorithm explores all used memory. This is an implementation which wants to be simpler. If the responsible zone which raised the GC is too small after memory recovery, then a new chunk is allocated for this zone.

3 Distinguishing Algorithm

When the GC explores an object, it runs four tests to assure that it is in the presence of a pointer, as follows :

- is the pointed object in the heap?

- does this address belong to a chunk in use?
- is the pointed object correctly aligned for this chunk?
- is there an object allocated to this address?

If the answer to these four questions is yes, then the object can be an address and the pointed object will be marked; otherwise, the object is an immediate value. Because all the pointed objects have a type field, it is possible to check if there is an object allocated to this address.

3.1 safety of the algorithm

Let H be the heap, corresponding to the interval $[H_B, H_E]$, and let C_i be a chunk inside H. Each chunk has the same size which is noted ***chunk_size***. Each chunk contains elements of the same size : $element\_size(C_i) = 2^{2+n}$ *bytes with* $1 \leq n \leq nbzones$ (for example I use $nbzones = 10$). Each chunk is well aligned inside the heap : C_B is the corresponding address of the beginning of the chunk which verifies C_B *modulo chunk_size* $= 0$ (this is a simplification of $C_B - H_B$ *modulo chunk_size* $= 0$, because H_B *modulo chunk_size* $= 0$).
If **obj** is a CeML value, we can determine if **obj** is a basic value (**val**) or a value inside the heap (**adr**) :

if **obj** $\notin [H_B, H_E] \rightarrow$ **val**
else *let* $C = which\_chunk(obj)$
    let $shift = (obj - C_B)$ *modulo* $(element\_size(C))$
    if $shift \neq 0 \rightarrow$ **val**
    else if **obj**[1] $= empty \rightarrow$ **val**
        else $\rightarrow$ **adr**

If an immediate value, such as an integer, has a value which cannot be distinguished from a correct memory address, then this integer is considered as a pointer and the correct ML object pointed at this address is explored and preserved.

3.2 probability of bad distinction

Conflict probability is the probability of being inside the heap times the probability of being well aligned. The worst case appears when all the chunks are used.

$$P_{conflict} = P_{insideheap} * P_{wellaligned}$$

where for a heap of m Mbytes and for a chunk size of 4 kilobytes, we have $m * 2^8$ chunks, then the address space contains 2^{32} bytes, and gives the following probability :

$$P_{insideheap} = \frac{m * 2^{20}}{2^{32}} = \frac{m}{2^{12}}$$

I call "zone" the set of chunks corresponding to the same size elements. The object size inside a zone i is 2^{i+2} bytes. The probability of being well aligned inside a zone is the product of the probability of being inside a zone times the probability of being well aligned inside this zone.

I assume the same space occupation for each zone. The probability of being well aligned inside long objects is comparatively small.

$$\begin{aligned} P_{\in i * wellalignedinsidei} &= P_{\in i} * P_{wellalignedinsidei} \\ &= \frac{1}{nbzones} * \frac{1}{2^{i+2}} \end{aligned}$$

Then :

$$\begin{aligned} P_{wellaligned} &= \sum_{i=1}^{nbzones} P_{\in i * wellalignedinsidei} \\ &= \sum_{i=1}^{nbzones} \frac{1}{nbzones} * \frac{1}{2^{i+2}} \\ &= \frac{1}{nbzones} * \sum_{i=1}^{nbzones} \frac{1}{2^{i+2}} \\ &= \frac{1}{nbzones} * \frac{1}{2^2} * \sum_{i=1}^{nbzones} \frac{1}{2^i} \\ &\leq \frac{1}{nbzones} * \frac{1}{2^2} * 1 = \frac{1}{4 * nbzones} \end{aligned}$$

And :

$$\begin{aligned} P_{conflict} &= P_{insideheap} * P_{wellaligned} \\ &\leq \frac{m}{2^{12}} * \frac{1}{4 * nbzones} = \frac{m}{nbzones * 2^{14}} \end{aligned}$$

With a use of 5 Mbytes of memory for the heap ($m = 5$) and ten zones ($nbzones = 10$), the error probability $P_{conflict}$ is $\frac{1}{2^{15}}$.

This small probability is acceptable (1/32768). In general though, the full use of chunks is not equiprobable. In fact, there are more small objects than large objects such as constructors, references, lists, pairs and closures. The factor is about 4 times greater in a more standard case, but the entire heap is not always in use and the distinction has a smaller probability.

3.3 remark

This GC cannot be used in the Lisp family (or any dynamic typechecking language) because it is not possible, dynamically, to determine the object type. This feature is a consequence of the same representation of the immediate values and pointers.

That, an integer can have the value of a correct pointer, for example, represents a CeML object. In this case, the `typeof` function returns a bad result.
For this reason, the symbol = which is polymorphic has the semantic of **eq** (equality for immediate values or sharing for structured objects) and not the semantic of **equal** (equality of structures). This is a problem for the programmer, but this is more coherent with the language definition because the ML polymorphism is parametric, i.e. it does not look at the form of the function arguments.

4 Comparison and discussion

First, I present a comparison between the previous GC algorithm and a **Stop&Copy** GC for CeML. This comparison allows for a discussion of tags and boxed objects for the GC. I then compare CeML with other ML dialects.

4.1 comparison with a Stop&Copy GC for CeML

Another GC was implemented for CeML. It was a **Stop&Copy** with boxed objects inside the heap. When the arguments are given to the function they need to be unboxed. Inside the heap, each object has a word to describe its type. The main difference concerns the immediate values. Because objects can move, there is a different representation for the values and the variables into the root set. The variable address is pushed into the root set.

I give the execution time (Unix user time) for three examples in figure 4 : `Itlist`, `Oct` and `MapOct`. They manipulate all three polymorphic functions : `Itlist` (iteration on the lists) tests the optimized total application, `Oct` (Church integers) the general apply mechanism and `MapOct` (Church integer lists) the partial application.

The times are given in seconds. The times in parentheses correspond to new optimizations which are not supported by the **Stop&Copy** version.

DS3100	Stop&Copy	Mark&Sweep (AR)
`ItList`	0,8	0,7 (0,3)
`Oct`	1,8	0,7
`MapOct`	7,2	4,3 (2,9)

Fig. 4. Experimental results

In the two CeML GC implementations, which establish a direct correspondence between ML functions and C functions, it is possible to use some C tools for the execution profiling. For example, the `pixie` tool, for the MIPS computer, profiles the optimized C program. This tool gives the time and the call number for the allocation and recovery functions. For example, the example `Oct` does not use recovery memory. The difference between the two times comes from the boxing and unboxing of objects needed for the **Stop&Copy** GC. This feature is discussed in the next subsection (4.2).

4.2 problems with tags or boxed objects

With this `pixie` tool it is possible to interpret these experimental results. For the `Stop&Copy`, each value inside the heap or the root set is boxed by its type. The immediate values are boxed when they belong to a structured object, but they are not boxed when they are given as arguments to a C function (in order to follow the implementation constraints). From this data representation some difficulties appear with polymorphic functions. The different kinds of polymorphism are well described in [15], but the terminology does not distinguish between polymorphic parameters and polymorphic results. To make this distinction, I note the term *polymorphism in input* for a function if one or more parameters are polymorphic and the term *polymorphism in output* if its result is polymorphic. Let us look at these two cases :

- *polymorphism in input.* When an argument, corresponding to a polymorphic parameter, must be stored inside the heap, allocation needs to know the argument's type. The type of each argument corresponding to a polymorphic parameter must then be given as a supplementary argument to the C function. The number of arguments grows for the polymorphic function.
- *polymorphism in output.* If a function is *polymorphic in output*, its result can be given to a second function *polymorphic in input*. In this case, the second function needs to know the result's type. These two kinds of applications thus coexist. The first one is the standard application which returns the result value. The second one returns the result value with its type. For the following function :

  ```
  let double f x = f ( f x);;
  ```

  `f` needs to know the type of `(f x)` dynamically. In this case the application mechanism becomes slow.

These problems are similar if the data representation is not uniform ([12]).

Another possibility is to use one bit (tag) to make the distinction between immediate values and pointers. This solution is adopted in Lisp systems. This is reasonable for dynamic typechecking languages because at any time a Lisp program can check the object type (since it is not satisfying to be ambiguous). In ML, though, the solution which preserves the uniform data representation and takes up little memory, loses the entire 32 bits of immediate values (to 31 bits) and the addressable memory space. One possibility is to use the higher bit to distinguish immediate values and pointers for each created integer in which case it is necessary to mask this bit. We have the same problem if the lower bit (lowtag) equals 1 for the immediate values is chosen. The last standard possibility is to use the lower bit which equals 0 for the immediate values, but in this case the multiplication and the division of integers are no longer efficient and the pointers are no longer aligned. In this latter case, each memory access must be indexed. As a result, the performances are very dependent on the processor. In any given case, if it is possible to find a good tag for a data type, this data representation is not good for the set of data types, particularly in the case of a compiler to C language.

4.3 related work on GC without tags

My GC was inspired by the work of Boehm and Weiser ([4]). Their GC has a different representation of chunks, which are a multiple of 4Kb and contain mark bits. They use the C stack as root set. This point increases the tests to determine immediate values and pointers (because inside the C stack we can find many immediate values which should not be pushed into the root set).

Bartlett in [3] distinguishes two object classes : those which are directly referenced by the root set and the others. The first ones are left in place and the other objects can be moved. This method can be used for ML. With a direct mapping between pattern variables and local C variable though, we need to push the pattern variables into the root set. Then the root set increases.

In [8], Goldberg associates a frame_gc_routine for each function call which knows how to trace the function frame. For the polymorphic functions, their frame_gc_routine are parametrized by the types of the polymorphic function. In the case of a polymorphic function nested inside another polymorphic function, the type informations are given at each level. For that, the GC starts from the bottom of the stack and the deduction of type for the polymorphic arguments reflects the execution process. This method works well for ML, but a complete implementation is needed to measure its performances.

Edelson presents in [7] a generational **Mark&Sweep** collector for C++. he uses a *buddy* [10] system for the allocator. His GC is *type-accurate*, i.e. every value that the collector interprets as a pointer is statically typed to be a pointer. This is the opposite for ML which does not have this information because the polymorphic functions do not make any distinctions between immediate values and pointers.

4.4 comparison with others ML dialects

I compare this CeML implementation with the main ML compilers divided in two families CAML and SML for the following implementations : CAML, CAML-LIGHT [11], SML-NJ [2] and SML2C [17]. The compiler performances are measured for ten programs (figure 5) on a DecStation DS3100 (MIPS R2000). These tests describe the main characteristics of the functional languages (data representation, complete and partial application, polymorphism, pattern matching, exceptions and imperative features) and allow us to measure the performances of these GC. The times are given in seconds by the Unix command `time`. Only the user time is considered.

An ML implementation depends mainly on its compilation model for the application, on data type representation and on its GC algorithm. It is thus not possible to separate the GC time from the execution time. There are only two programs which use only the C stack rather than using the heap in CeML. For all other programs though, the heap is fully used and the GC performances can be compared. CeML performances are satisfactory in comparison to the C compiler (in many cases the C program generated by the CeML compiler is very similar to the equivalent hand-written C program) and with the best ML implementations. There are however two types of programs that do not verify these good performances : the very functional programs (close to λ-calculus as in the case of Church integers) and programs which use too much exception handling. In the first case, the partial application and the

DS3100	CAML			SML		
Test	V2-6.1	light	CeML	NJ 0.66	SML2C	What is mainly tested?
1 Fibonacci	6.7	42.0	**2.5**	4.7	14.5	integers
2 Takeuchi	18.5	12.4	**0.7**	4.6	11.3	function calls (3 args)
3 Integral	4	6.7	**1.4**	**1.4**	3.8	floats, loop
4 CountStr	12.5	**1.3**	1.9	6.6	10.3	strings
5 Reverse	14.6	9.6	**2.2**	2.6	6.4	list processing
6 Sieve	7.5	13.2	**2.4**	4.0	10.7	list processing, functionals
7 ItList	4.6	7.2	3.1	**2.1**	4.0	list processing, functionals
8 Church int	5.4	10.4	6.4	**1.2**	4.8	functionals, polymorphism
9 TakExcept	24.2	18.3	15.4	**7.2**	14.5	exceptions
10 SigmaVect	4.6	29.0	**1.3**	5.1	9.9	vectors

Fig. 5. Experimental results

application of closures given as arguments forbid application optimization. In the second case, the exception handling mechanism is above all dependent on the C compiler implementation. On Unix system, the `setjmp/longjmp` library, which is used in CeML runtime library, generates too much work (it saves the bitmasks for signals,...) to implement the exceptions well. Since the partial application and the intensive use of exceptions are in fact scarce, they do not put into question the CeML implementation choices.

With its more informative typechecker (arity of functions, expressions decorated with their types), the CeML compiler yields excellent optimizations for the total application and the immediate value manipulation. But these optimizations are not always possible. When the application depends on a functional argument, then its arity is lost for the application optimization. The execution speed then varies according to the ratio of non-optimized application / optimized application. Its GC, with ambiguous roots, avoids the tagging of immediate values. The distinguishing algorithm, between an immediate value and a pointer, slows the GC down, but the benefits achieved through the uniform representation of data is, in most cases, greater than the slowdown. It is particularly well suited to the implementation of parametric polymorphism.

Conclusion

This GC, `Mark&Sweep` with ambiguous roots, satisfies the initial constraints, i.e. permits the representation of basic ML types by basic C types and the use of the direct C function call. It also allows us to push the parameter variables and local variables, but not the pattern matching variables. This GC seems simple (less than 700 lines) but it is more efficient than the other implementation of a `Stop&Copy` for CeML. Moreover it facilitates good optimizations for the application (as seen in the previous examples) which is the other main feature for the functional language compilers. Finally, the efficiency of a CeML program depends on the ratio between optimized application and non-optimized application (as the profiling C tools show).

It is completely independent of the C implementation, but it is appropriate for a word size (32 bits, or in the future 64 bits). Finally, it is pleasant to profit from the static typechecking for the GC and to show the good properties of ML for its implementation.

Acknowledgement

I would like to thank Bernard Serpette for the discussions on "to tag or not to tag" and for his remarks on the draft of this paper.

References

1. APPEL, A. Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation* (1989).
2. APPEL, A., MCQUEEN, D., AND DAVID, B. A Standard ML Compiler. *Functional Programming Languages and Computer Architecture* (1987).
3. BARTLETT, J. F. Compacting Garbage Collection with Ambiguous Roots. Tech. Rep. 88/2, Digital Equipement Corporation (WRL), Feb. 1988.
4. BOEHM, H., AND WEISER, M. Garbage Collection in an Uncooperative Environment. *Software - Practice and Experience* (Sept. 1988).
5. CHAILLOUX, E. *Compilation des langages fonctionnels : CeML un traducteur ML vers C.* Thèse d'université, Université Paris VII, Nov. 1991.
6. CHAILLOUX, E. An Efficient Way of Compiling ML to C. In *Workshop on ML and its Applications* (San Francisco, June 1992), ACM SIGPLAN.
7. EDELSON, D. A Mark-and-Sweep Collector for C++. In *Principles Of Programming Languages* (Albuquerque, 1992), ACM.
8. GOLDBERG, B. Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *Programming Language Design and Implementation* (1991), ACM.
9. JOHNSSON, T. Lambda lifting: transforming programs to recursive equations. In *Conference on Functional Programing Languages and Computer Architecture. LNCS 201* (Nancy, 1985), ACM, Springer Verlag.
10. KNUTH, D. *The Art of Computer Programming : Fundamental Algorithms*, vol. 1. Addison Wesley 3821, 1973.
11. LEROY, X. The ZINC experiment : an economical implementation of the ML language. Tech. Rep. 117, INRIA, Feb. 1990.
12. LEROY, X. Unboxed Objects and Polymorphic Typing. In *Principles Of Programming Languages* (Albuquerque, 1992), ACM.
13. MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM* (1960).
14. MINSKY, M. A Lisp Garbage Collector Algorithm Using Serial Secondary Storage. Tech. Rep. Memo 58, MIT, Cambridge, Massachussets, 1963.
15. MORRISON, R., DEARLE, A., CONNOR, R. C. H., AND BROWN, L. An Ad Hoc Approach to the Implementation of Polymorphism. In *Transaction on Programming Languages ans Systems* (1991), ACM.
16. STEELE, G. L. Data Representation in PDP-10 Mac Lisp. In *MACSYMA Users Conference* (1977).
17. TARDITI, D., AND ACHARYA, A. A Guide to SML2C. Tech. rep., CMU-CS, June 1991.
18. WEIS, P., APONTE, M. V., LAVILLE, A., MAUNY, M., AND SUAREZ, A. The CAML Reference Manual. Tech. Rep. 121, INRIA, Sept. 1990.

This article was processed using the LaTeX macro package with LLNCS style

An Efficient Implementation for Coroutines

Luis Mateu

INRIA-Rocquencourt &
Universidad de Chile

Abstract. Emulating coroutines with first-class continuations imposes an unacceptable overhead in managing function frames when there is an intensive exchange of control. This paper presents a high-performance implementation for a restricted class of continuations. These continuations are exploited in a simple coroutine mechanism, reaching a rate of 430,000 control exchanges per second on a modern RISC processor. As an extra feature, first-class continuations are recovered from the restricted class.
Keywords: coroutines, continuations, garbage collection, dynamic variables, shallow binding

1 Introduction

A coroutine is a kind of concurrent process, getting and passing control explicitly. The simplest way to implement them is to use multiple stacks, one for each coroutine. The problem with this approach is that whenever memory resources are limited, the deepest function recursion must be traded off against the maximal number of simultaneous coroutines. Yet, predicting the deepest function recursion is generally impossible.

On the other hand, Scheme [Rees & Clinger 86] has abstracted a wide variety of control structures —including coroutines and escapes— into just one general control operator named `call-with-current-continuation` (or, in its abbreviated form, `call/cc`). This operator reifies its *continuation* into a first-class function, which can then be treated just as any other function in Scheme. The continuation of `call/cc` is the rest of the computation from its application point. In Scheme, coroutines can be obtained by reifying the continuation of a computation to emulate the exchange of control [Haynes *et al* 86].

Scheme continuations can be implemented by allocating function frames in the general heap, where they are managed by a normal garbage collector. In this way, there is no trade off to be solved because all frames share the same heap and deep function recursion is treated by normal heap exhaustion. With some optimizations [Clinger *et al* 88], this memory organization has a small overhead for normal procedural applications.

However, we state in Sect. 2 that Scheme continuations could not be an effective way to emulate coroutines, because once a continuation has been reified, the only way to recycle the captured frames is by triggering an expensive general garbage collection, in which all the objects are involved.

* Postal address : INRIA, Bât. 8, Domaine de Voluceau-Rocquencourt, B.P. 105, 78153 LE CHESNAY CEDEX, France. Email Address: mateu@margaux.inria.fr.

The goal of this paper is to introduce a fast implementation technique for a restricted class of continuations. These continuations are used in a simple coroutine mechanism, solving effectively those problems having a natural solution with coroutines, i.e. the performances are competitive with alternative procedural solutions. If concurrency were to be added among the features of Scheme, to have a fast coroutine mechanism (i.e. context switch facility) is also of paramount importance and is solved by our model.

The basic idea is to allocate frames in a dedicated heap, managed by a generational *Stop and Copy* garbage collector. We add a new object class, the *hooks*, which are used to store the continuation of suspended coroutines. Thus continuations can be only held in hooks. When the frame memory is exhausted, an inexpensive garbage collection recycles unreachable frames. This collection is cheap since it is only applied to the frame heap compared, as in the Scheme case, to the whole general heap. This is possible, since the roots are found in the hooks, which are bounded by the number of coroutines.

In Sect. 3 we present the set of coroutine primitives and we show that they recover the Scheme notion of first class continuations. Also, we apply them to solve the *same-fringe* problem in an elegant way. In sections 4 and 5 we implement them.

In Sect. 7 we compare the performances of our heap organization to several stack organizations, showing that the main overhead comes from the locality loss in memory access. So in Sect. 8 we introduce an optimization for normal applications which reduces most of this overhead. With this optimization and others, the execution time overhead for normal applications is around 11%, compared to a stack based implementation providing no coroutine facility. In Sect. 9 we compare the performances of a coroutine based solution of the *same-fringe* problem against the trivial procedural solution. Some possible extensions are discussed in Sect. 10.

2 First-class continuations and the same-fringe problem

The *same-fringe* problem determines whether the sequence of leaves —the fringe— of two trees are the same. This problem is easily solved with three coroutines as shown in the next section. The first compares the leaf sequences returned sequentially by the other two coroutines, each of them traversing one of the trees recursively. When arriving at a leaf, a coroutine traversing a tree passes the control to the comparing coroutine. Later, the coroutine is resumed at the same point where it had been suspended, to continue traversing that tree. In this section we examine the performances of a garbage collector when first-class continuations are used to emulate suspension and resumption of the coroutines in the *same-fringe* problem.

As stated in the introduction, a trivial implementation of Scheme continuations is achieved by allocating frames in the general heap. Unfortunately, memory allocated for frames is much more important that memory allocated for normal objects so garbage collection activity increases, thus degrading performance. This heavy frame allocation is not visible in a stack organization, because frame lifetime is very short, thus frames are popped as soon as they are pushed.

For applications not using the Scheme continuations intensively, several implementation strategies are discussed in [Clinger *et al* 88] and [Hieb *et al* 90]. These

strategies reduce the associated overhead by using a stack cache to execute normal call/return behavior, but transferring frames from the stack to the general heap when a continuation is reified. In some strategies frames are also transferred from the heap to the stack when a continuation is invoked.

Now, let us consider using first-class continuations to emulate the coroutines in the *same-fringe* problem. The suspension of a tree traversal is achieved by reifying its continuation, and the resumption by invoking it. To traverse a tree recursively, a function is called at every internal node. This function allocates a frame in the stack cache. However, sooner or later that frame will be transferred to the heap by a continuation capture at a leaf. Therefore any optimization introduced to treat normal call/return behavior will be useless, because all frames will be captured by a continuation.

Considering that the size of each transferred frame is at least the size of a cons cell, and there is an additional space overhead in creating a callable continuation, we become aware that the garbage collection activity will be much more important than in a trivial solution which flattens the trees into lists prior to comparison. Thus performances will be unacceptably slow for the first-class continuation solution.

3 Coroutines as second class continuations

In fact, the aim of using coroutines to solve the *same-fringe* problem is firstly, to decrease the additional memory requirements to allocate a new list in the tree flattening solution, and secondly, to reduce the execution time overhead incurred in managing that memory. When emulating coroutines with first-class continuations, we can see that the former is successful, because the surviving frames at memory exhaustion are just those present in the branch of the current node in the tree traversal, and not the whole. Yet, for the latter, it is just the opposite that has been obtained.

Therefore, beginning with this section, we will be concerned with reducing the memory management overhead incurred to treat coroutines when frames are allocated in a heap. To achieve this goal we will introduce a coroutine definition based on continuations. Although these continuations are not first-class functions as in Scheme, we will show that `call/cc` can still be obtained from our coroutines.

We start by defining the new type *hook*. A hook is a continuation holder encapsulating a limited set of legal operations. Hooks are first-class objects, i.e. they have an unlimited extent and they can be passed as arguments to functions, returned from functions, and stored in data structures. They are created and manipulated with the following operations (an accurate semantics is presented in the appendix):

- `(coroutine f)` with `f`= `(lambda (hook) ...)`
  This primitive is used to create a coroutine. It allocates a new hook filled with the continuation of the `coroutine` form. Then the `f` function is applied on the hook. When `f` returns, the continuation currently held in the hook is invoked on the returned value.
- `(escape hook val)`
  This primitive allows a coroutine to exit, passing control to another coroutine. It invokes the continuation held in `hook` on `val`. This means that `val` is returned

as the value of the `coroutine` or `suspend/resume!` form that was the last to fill the hook.

- **(suspend/resume! hook val)**
  This primitive allows the suspension of the current coroutine, resuming another previously suspended coroutine. Therefore this is a kind of explicit context switch mechanism between coroutines. It exchanges the current continuation with the continuation held in `hook`, and invokes this latter on `val`.

The primitives `coroutine` and `suspend/resume!` are used to capture continuations just as `call/cc` in Scheme. However, continuations can't be obtained as first-class objects, because there is no legal operation reading the hook contents directly. Yet, the original first class continuations are recovered by defining:

```
(define (call/cc fun)
  (coroutine
    (lambda (hook)
      (fun
        (lambda (value)
          (escape hook value))))))
```

The inner lambda that is passed to `fun` emulates the Scheme continuations. It is actually a first-class function because closures are first-class in Scheme. Note that the continuation held in a hook is not lost when `escape` is used, so it can be reinvoked. In this way, multiple returns from function applications are also recovered.

Nevertheless, just using this newly defined `call/cc` gives no performance advantages over the Scheme first-class continuations. We will see that an efficient implementation can be conceived for applications creating a moderate number of coroutines but heavily exchanging control, as in the following solution for the *same-fringe* problem:

```
;;;a leaf reader
(define (make-walker tree)
  (coroutine
    (lambda (hook)
      ;;a recursive traversal
      (define (walk tree)
        (cond
          ((not (pair? tree))
            (suspend/resume!
              hook tree))
          (else
            (walk (car tree))
            (walk (cdr tree)))))
      ;;returns the hook
      ;;to the client
      (suspend/resume! hook hook)
      ;;starts the traversal
      (walk tree)
      ;;signals the end
      'end )))
```

```
;;;The comparator
(define (same-fringe tree-a tree-b)
  ;;starts the two leaf readers
  (define hook-a
    (make-walker tree-a))
  (define hook-b
    (make-walker tree-b))
  ;;loops on the leaves
  (let loop ()
    (let ((leaf-a (suspend/resume!
                    hook-a 'void))
          (leaf-b (suspend/resume!
                    hook-b 'void)))
      (cond
        ((not (eq? leaf-a leaf-b))
              #f)
        ((eq? leaf-a 'end)
              #t)
        (else
              (loop))))))
```

4 Implementing second class continuations

Let us consider a Scheme implementation passing arguments in registers and allocating a fixed size frame at function entry. This frame is allocated in a special heap controlled by the frame memory manager presented in next section. A frame contains the following fields:

- **tag**: A frame identifier used by the frame memory manager. This tag can be a pointer to a structure containing the frame layout.
- **retaddr**: The caller return address.
- **oldfp**: The caller frame address. The fields **retaddr** and **oldfp** represent the implicit continuation passed to every function.
- Some optional static frame pointers: Present only when the function accesses variables located in lexically enclosing functions.
- Some variables: The programmer defined variables, the function arguments and some intermediate values, which are held in registers initially, but need to be saved when a function call uses some of those registers and also at register exhaustion.

Upon function entry, a frame is allocated and initialized with the caller information. The frame address is placed in a dedicated register named *current frame pointer* or simply **fp**. At return, **fp** is restored with the caller frame address and a jump to the caller return address is done. Since a frame can still be useful even after function return, it can't be freed as easily as in a stack implementation. When there is no more memory for allocating frames, the frame memory manager reorganizes the heap by pruning frames that are no longer reachable from the current frame pointer or a continuation held in a hook.

A hook is a structure allocated in the general heap. It contains a **tag** identifier used by the memory manager and a field named **cfp** which is a pointer to a *capture frame* structure. A capture frame is a special frame created at a continuation capture for storing the information needed to invoke that continuation. Excepting a hook, there is no other first-class object pointing to a capture frame. A capture frame is allocated in the special frame heap and contains the following fields:

- **tag**: A capture frame identifier.
- **retaddr**: The return address to jump to when invoking the continuation.
- **oldfp**: The frame address of the function being suspended.
- **hook**: The address of the hook involved in the continuation capture and which will be linked to this capture frame.
- **nextcfp**: This pointer is used by the frame memory manager.

Figure 1 shows the linking between a hook, a normal frame and a capture frame.

Using these hook and capture frame structures, the coroutine primitives are implemented as follows:

- (**coroutine f**): A capture frame **cfp** is allocated with **tag**, **retaddr** and **oldfp** filled as upon normal function entry. Next, **hook** is allocated to hold the continuation of the **coroutine** form. Then the following code is executed:

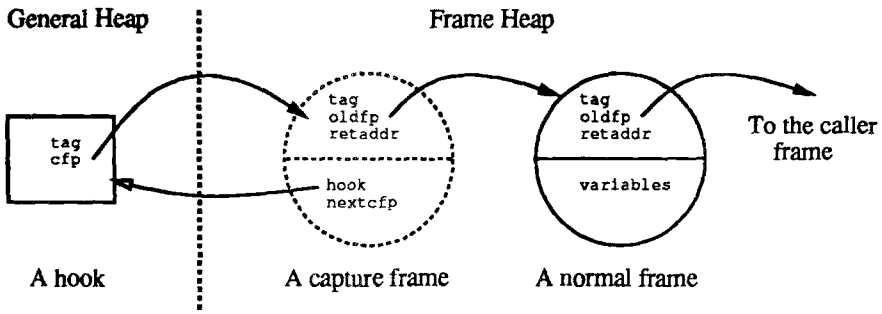


Fig. 1. Hook and frame linking.

```
cfp→hook= hook          ;; The capture frame and the hook
hook→cfp= cfp           ;; are made to point at each other.
cfp→nextcfp= listcfp    ;; The capture frame is chained in a list,
listcfp= cfp            ;; for reasons which we will explain later.
fp= cfp                 ;; The capture frame becomes the current
escape(hook, f(hook))   ;; frame and f is invoked.
```

Note that upon normal return of `f`, this form does an escape through the current hook contents.

- (**escape hook val**): A normal return is done as if the current frame was the capture frame referenced by **hook**.

```
fp= hook→cfp
return val
```

- (**suspend/resume! hook val**): A capture frame **cfp** is allocated with **tag**, **retaddr** and **oldfp** filled as upon normal function entry. Then the following code is executed :

```
fp= hook→cfp            ;; The capture frame in hook
                        ;; becomes the current frame.
hook→cfp= cfp           ;; Then hook is linked to cfp.
cfp→nextcfp= listcfp    ;; The capture frame is chained
listcfp= cfp            ;; in a list, as in coroutine.
return val              ;; A normal return is made from
                        ;; the new current frame.
```

Figure 2 shows the frame and hook chaining for an example of function call tree. Frames have been enumerated according to allocation order. While working with frame 1, a coroutine is created, so the hook H and the capture frame 2 are allocated. Then the frame 3 is allocated for the function starting that coroutine. Next, using **suspend/resume!** through the hook H, that coroutine is suspended and the work with frame 1 resumed, so the capture frame 4 is allocated. A function call allocates frame 5 where a **suspend/resume!** through hook H creates the capture frame 6 and resumes the work with frame 3. Another function call allocates the frame 7 from where an escape through hook H is done, resuming the work with frame 5. Finally

a normal return resumes the work with frame 1 from where a last function is called allocating frame 8.

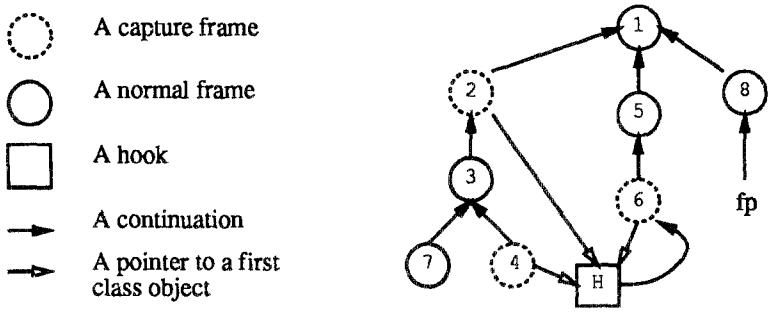


Fig. 2. An example of function call tree.

Initially, the hook H has been linked to frame 2, but the two successive **suspend/resume!** operations link it to frame 4 and then frame 6. Note that frame 7 hasn't been captured, so its memory is available for new allocation. In the same way, the capture frames 2 and 4 aren't reachable from any hook, so the memory taken by frames 2 and 4 and then frame 3 is also available.

5 An efficient memory manager for frames

We split memory management into two almost independent garbage collectors. The first is the general garbage collector managing first class objects such as cons cells, symbols, vectors, etc. and especially hooks. The second is the frame only memory manager —including capture frames— which is presented in this section. This frame memory manager is a simplification of the generational *Stop and Copy* garbage collector described in [Nakajima 88] and [Appel 89].

Frame allocation is implemented as follows. At the beginning of a cycle, there is an empty buffer which we will name the primary buffer. Two registers **hp_limit** and **hp** point at the start and the end of this buffer. A frame is allocated by subtracting its size from **hp** and comparing the new **hp** against the **hp_limit** register to test the buffer overflow. It is important to note that there is no need to initialize frames, instead the primary buffer is cleaned of dangling references by initializing it to zero or nil after a general garbage collection.

When the primary buffer overflows, it holds the frames for the complete function call tree from the beginning of the cycle. As stated in the previous section, some branches of this tree are unreachable, so the reachable ones are appended to another buffer which we will name the secondary buffer. Then a new cycle begins with an empty primary buffer.

The reachable frames are firstly, those captured by the current frame which signaled the overflow, and secondly, frames that have been captured by a continuation held in a hook. However, for the latter, it has been necessary to create a capture

frame in the primary buffer from where they are reachable. Therefore **coroutine** and **suspend/resume**! chain the capture frames that they create in a list which we will name the primary capture frame list or simply **listcfp**. In addition, some of these capture frames are no longer referenced by a hook, because their initial hook has been used to capture another continuation, so they are considered unreachable unless another continuation captures them.

We outline a simple copying collector to transfer frames to the secondary buffer. We say that this collector prunes frame trees.

1. For each capture frame C in **listcfp**:
   (a) If C points to a hook no longer linked to C, continue with the next capture frame in **listcfp**.
   (b) Reverse the dynamic chain obtained from C by following the **oldfp** field as far as a frame located in the secondary buffer or a frame that has been marked as already transferred.
   (c) For each frame F in this new chain:
      i. Make a copy of F in the secondary buffer. This copy will be named F'. The size of F is determined from the **tag** field.
      ii. Link **oldfp** in F' to the copy of the caller frame which is just the previously transferred frame.
      iii. Set a mark in the **tag** field in F indicating that F has been transferred.
      iv. Link **oldfp** in F to the address of F'.
      v. If F has some static pointers, since the referenced frames are in the dynamic chain, they have already been transferred, so relink any static pointer to its new address. This address is found in the **oldfp** field of the referenced frame.
   (d) Let C' be the copy of C. C' points to a hook having the **cfp** field linked to C. Link **cfp** to C'. Then chain C' into a list which we will name the secondary capture frame list.
2. Transfer in a similar way the dynamic chain obtained from the current frame pointer.
3. Set **listcfp** to the empty list.

Afterwards, the execution must be resumed with a primary buffer reduced to the size of the remaining memory in the secondary buffer. Thus, when the primary buffer overflows again, all new frames are guaranteed to find room in the secondary buffer, even when all of them are reachable. When this frame pruning is triggered after a primary buffer overflow, we call it a *minor pruning*.

If the primary buffer becomes too small —assume a quarter of its initial size— make a *major pruning*. A major pruning exchanges the primary and secondary buffers and the primary and secondary capture frame lists, and then does a normal pruning. Most old frames transferred to the secondary buffer aren't reachable, so they won't be recopied and the secondary buffer will regain a reasonable size.

Figure 3 shows the primary and secondary buffers before pruning the tree of Fig. 2. Frame 1 is the only frame already located in the secondary buffer. Figure 4 shows the buffers once the frames reachable through hook H have been transferred. Finally, Fig. 5 shows the buffers once the pruning has finished.

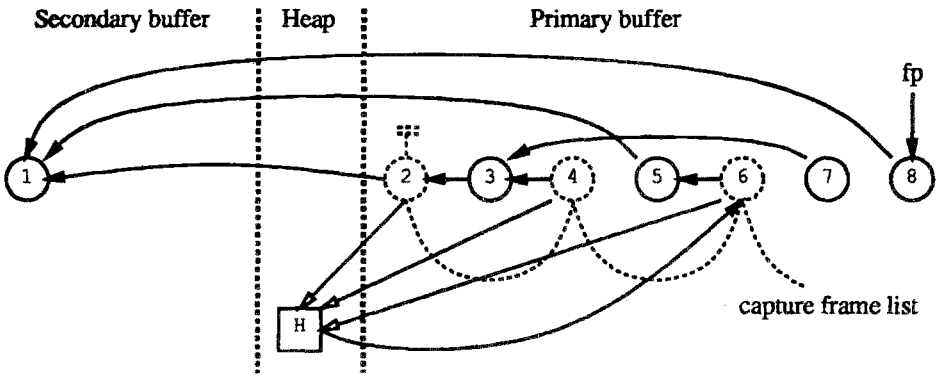


Fig. 3. Frame tree before pruning.

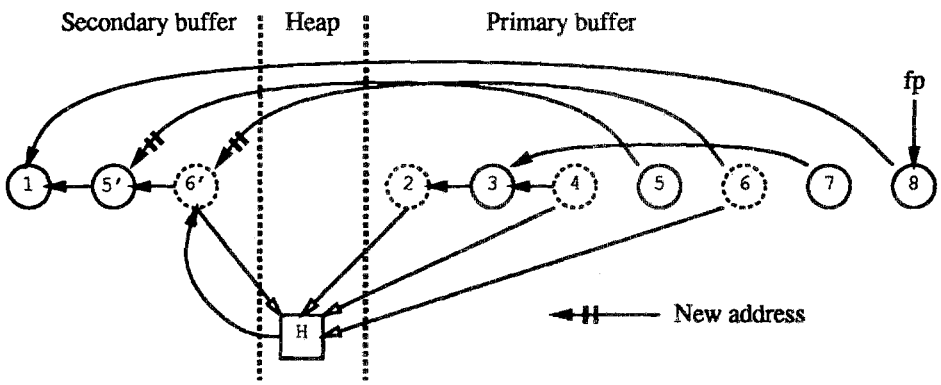


Fig. 4. Buffer contents after transferring frames reachable through hook H.

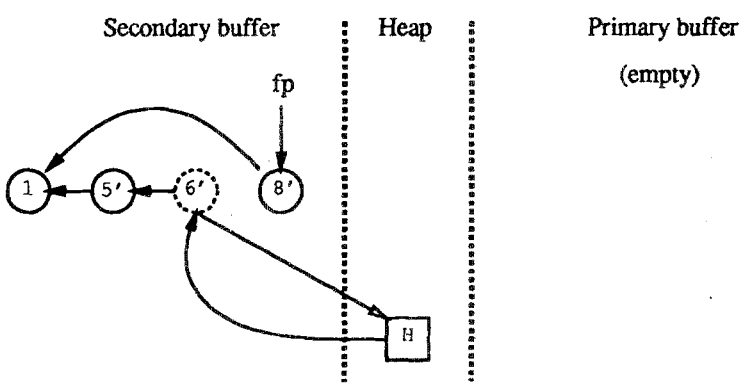


Fig. 5. Final pruned frame tree.

Having an unlimited extent, a hook might become garbage for the rest of the execution. Its captured frames will therefore appear to be reachable forever to the frame memory manager. These useless frames will fill the secondary buffer increasing the pruning frequency and so degrading performance. Hence, if after a major pruning the secondary buffer is filled up to a given percentage, a general garbage collection must be triggered to discard the unreferenced hooks. However, this situation should occur rarely because it is the normal heap exhaustion which will trigger the collection.

6 Synchronization between first-class object and frame memory management

During a general garbage collection, a subtle synchronization with the frame memory manager must be done to recover the frame space captured by garbage hooks. To achieve this goal, while doing the general garbage collection, a special major pruning is carried out simultaneously as follows:

1. Make a minor pruning to free the primary buffer.
2. Exchange primary and secondary buffers.
3. Set the secondary capture frame list to empty.
4. Transfer the frames reachable from the current frame to the secondary buffer. The pointers held in those frames are roots for the general garbage collector.
5. Start the general garbage collector.
6. For each hook proved reachable by the general garbage collector:
   (a) Transfer the frames captured by the hook to the secondary buffer.
   (b) Chain the copy of the associated capture frame in the secondary capture frame list.
   (c) The pointers held in the transferred frames are roots for the general garbage collector.

Note that the method used by the general garbage collector doesn't matter. It could be a *Stop and Copy* or a *Mark and Sweep* collector with or without generations, etc.

Our frame memory manager and the general memory manager, as a whole, can be seen as a generational garbage collector [Lieberman & Hewitt 83], with the first two generations reserved for frames only. The complexity associated with generational memory management comes from the need to trace pointers from older generations to newer generations. These pointers are the result of object mutating operations such as `set-car!`, `vector-set!`, etc. However, frame chaining can't be altered in the frame heap, so pointers from secondary buffer to primary buffer can't exist and therefore no tracing is needed. Yet, coroutine suspension can link a hook —in an old generation— with a newer frame, hence the necessity to introduce capture frames which just serve to trace hook mutations.

7 Performance analysis for applications lacking coroutines

In this section we evaluate the overhead that the memory organization described in previous sections carries to applications not using coroutines. To measure this overhead we have implemented it in F1 [Seniak 91]. F1 is a compiler for a small lisp, generating assembler code for Sparcs. It uses the 31 Sparc registers as much as possible, but without calling on window registers. F1 doesn't treat floating point numbers, so no test is needed in integer arithmetic operations.

We show in table 1 the timings for the Gabriel benchmarks [Gabriel 85]. Timings include our heap organization as well as various stack implementations which test or don't test overflow and chain or don't chain frames. We include also the performances of a mixed stack/heap strategy which we explain in the next section. The measurements have been done on a SUN/670MP, having a 64 Kb cache.

To measure the overhead associated with tagging, we added the tags to the frame chaining stack organization, then this overhead was the additional execution time. To measure the overhead associated with frame pruning, we doubled the work by transferring surviving frames to a third intermediate buffer while pruning, then another pruning transferred the same frames to the secondary buffer. The pruning overhead was the difference with respect to our heap organization. Having the overhead of frame chaining, tagging and pruning, the remaining unexplained overhead was due to the locality loss in memory access.

Therefore the proposed organization has an overhead of 18% when compared with a stack implementation testing the overflow. However with the optimizations described in the next section, we reduce that overhead to a 11%.

The surprisingly small overhead of frame pruning is explained because the mean lifetime of frames is very short in conventional applications. The frames surviving to a minor pruning are those that are reachable from the current frame, excepting those that are already in the secondary buffer. Therefore, the number of transferred frames is the depth of the call tree at primary buffer overflow, less the minimal depth reached during the cycle. However, considering the locality of function call depth from which Sparc register windows are inspired, this number is very small. In fact, we have measured a 1 to 2% of frames surviving to a minor pruning, and a 2 to 5% surviving to a major pruning.

The locality loss is the main penalty for this heap organization. A normal stack organization presents a high degree of locality, explained also by the locality of function call depth. However, our organization allocates frames sequentially, flushing cache lines at almost every function call. In fact the 5.2% was obtained solely when the primary buffer was limited to a size by 16 or 32 Kb, to give an opportunity to the 64 kb cache to hold it completely, otherwise the overhead was greater.

8 Optimizing performances of frame allocation

The following minor variation is inspired from the stack/heap strategy described in [Clinger *et al* 88].

Upon normal function entry, after frame allocation, the `fp` and `hp` registers have the same value. If this still holds at return, no capture frame could be allocated,

so the frame memory can be reused safely. Therefore, at return, the `fp` and `hp` are compared and when they are equal, the allocated frame is freed by adding its size to the `hp` register. Moreover, when there is no captured continuations, a function calling another function will get the `hp` register with the same value that it had before the call, so it will also free its frame at return. Thus, frames will be pushed and popped in the primary buffer, just as in a stack, and the application will exhibit the locality of a stack organization.

During a continuation capture, the `hp` register is adjusted to allocate a capture frame. When that continuation is invoked, the `hp` register is not restored, so its further comparison against `fp` will fail, and all frames allocated before the continuation capture will not be freed. In this way, the frames captured by a continuation are guaranteed not to be reused for new allocation. Frames allocated after the capture frame will continue with the normal push and pop discipline.

Therefore, in normal applications, there is a gain in the locality of memory access, but there is also a loss in performing the test at function return. Although this test is useless when coroutines are exploited intensively, we have adopted it, because we desired a minimal penalty for normal applications and the measurements had shown that the gains were greater than the losses.

Another optimization adopted is the suppressing of frame tagging. In fact, the tags can be placed at primary buffer overflow for the reachable frames only. The tags can be deduced from the return address by using a binary tree, a hash table or, in some architectures, just including it in the code around the return address. The performance of an implementation with the stack/heap optimization and tag suppression is shown in table 1.

Finally, other optimizations are possible, even though we didn't adopt them. First, the overflow test at function entry can be suppressed by placing the frame heap in the stack space of a Unix process and organizing it as in Fig. 6. In this way, the test is done at continuation captures only, while at deep recursion the primary buffer grows automatically. Second, the return test can also be suppressed in functions which are known not to capture continuations by inspecting the static function call tree. In addition, those functions don't need the frame chaining. Third and last, established optimizations such as function integration and inlining can be applied to further reduce the call/return overhead.

9 Performance analysis for the same-fringe problem

Tables 2, 3 and 4 show the performances obtained by three solutions for the *same-fringe* problem. Each version compares 10 times two trees containing 100,000 cons cells.

The first solution is based on coroutines, so it does no allocation in the general heap, instead it captures a continuation at every leaf. Although the stack/heap optimization is present, it is useless, and therefore there is a low locality in memory access. Table 2 shows performances when varying the primary buffer size and the total frame heap size.

The second solution uses `call/cc` to emulate the coroutine context switch. For implementing `call/cc` we used a variation of the stack/heap strategy where a general

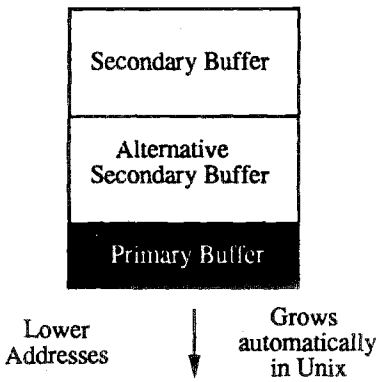


Fig. 6. A buffer organization allowing the suppression of the overflow test and simplifying the first ancestor search when implementing shallow binding. At a major pruning, frames are transferred from the secondary buffer to the alternative secondary buffer or vice versa.

garbage collection is triggered immediately after a frame heap overflow. We are constrained to do so because the first-class status of Scheme continuations exclude any frame pruning without proving that the continuation which captured a frame is not referenced by another first-class object.

The third solution flattens the trees, i.e. it chains the leaves of each tree into two lists before comparing them, resulting in a high general garbage collection activity. We used a Stop and Copy collector with no generations.

The measurements show that the version using `call/cc` is far the slowest. They show also that there is a performance crosspoint between the coroutine and the tree flattening solutions when objects survive in average one collection cycle in the tree flattening version. Therefore the latter will win especially when coupled to a generational garbage collector where objects rarely survive to one generation.

However, we think that coroutines must not be seen as a means to speed up applications. Instead, they are a powerful abstraction tool which can greatly simplify programming. The aim of presenting these measurements is to show that our coroutines aren't expensive even when used intensively as in *same-fringe*. In fact, in the coroutine version, *same-fringe* reaches a rate of 430,000 **suspend/resume!** per second. Yet, if *same-fringe* is considered as a subpart of a more complex system, the coroutines solution could win when a generational approach is not desired, because the programming paradigm involves too much object mutations, such as in object oriented systems.

10 Extensions

In this section we discuss some possible extensions to the memory organization described previously. We start by introducing a hint to implement shallow binding for dynamic variables; next, we discuss a way to treat dynamic escapes efficiently; and finally, we consider allocating dynamic objects in the frame heap.

Dynamic variables have been traditionally implemented in high-performance Lisps by using shallow binding, because the time needed to create, access and delete a dynamic variable is constant. However, combining shallow binding with coroutines or first-class continuations introduces a subtle complication. When transferring control between coroutines, the dynamic environment must first be unwound from the current frame up to a common ancestor with the target frame, and then, rewound down to the target frame.

This overhead in restoring dynamic environments can discourage language designers to add dynamic variables, because even when these variables are not used, coroutine users must pay at least the cost of finding the common ancestor to discover that there is no dynamic environment to restore. This search can be accelerated by chaining the frames containing dynamic variables in a special list. Yet, adding a single dynamic variable to the program, introduces an additional overhead in any control exchange between coroutines.

Therefore we point out an interesting property for the buffer organization of Fig. 6 when using the frame pruning of Sect. 5. For every frame f_1 pointing to a frame f_2, the following holds:

$$\text{address}(f_1) < \text{address}(f_2)$$

Hence, finding the common ancestor between two frames is as easy as unwinding the two dynamic chains up step by step, alternating in such a way that the chain containing the lower address frame is unwound first. The unwinding stops when the same frame is found. Thus, when there is no dynamic environment to be restored, the overhead associated with the search of the common ancestor is reduced to a single test.

Another desirable extension is a way to treat *dynamic extent continuations* efficiently. These continuations are useful to implement fast *escapes* such as `longjmp` in C. A dynamic extent continuation can only be invoked by a function that is a child of the function which captured that continuation. The capture of a dynamic extent continuation is implemented by allocating a hook and a *dynamic capture frame*, much as `coroutine` is implemented. A dynamic capture frame is a capture frame, but the fact that the former is referenced by a hook isn't enough to consider that frame reachable as is the case for the latter: the former must also be referenced by a reachable frame. Thus, there is no need to trigger an expensive general garbage collector to recover frames captured by a dynamic extent continuation. Detecting the illegal use of a dynamic continuation is achieved as follows: at a frame pruning, when a hook is linked to a dynamic capture frame considered no longer reachable, that hook is redirected to a special capture frame containing an error handler in its return address.

Finally, in a stack based organization, dynamic extent objects can be allocated efficiently in the stack. In our heap organization, such objects can also be treated efficiently, because the compiler can be modified to include layout information to be used at frame pruning. This information must describe where to find pointers to dynamic objects in the frames of functions allocating, or receiving in arguments, such objects.

11 Conclusions

In conceiving our memory organization for frames, we were inspired from [Appel 87] which states that garbage collection can be faster than stack allocation if very large heaps are coupled with a Stop and Copy collector. Although the original idea is not practical with current memory configurations, we found that it was reasonable when applied to Spaghetti stacks [Bobrow & Wegbreit 73], because stack memory requirements are much smaller than heap memory requirements. Then, we realized that adding generations to frame pruning was easy, because pointers from older frames to newer frames can't exist. Finally, experimentation established that frames have a very short life time, reaching to a point where frame pruning is almost costless. Therefore, large frame heaps are not recommended because the smaller ones are more efficient in presence of memory caches.

In this paper we concentrated on proving that this memory organization can be used effectively to implement a simple class of coroutines. However, we are conscious that the primitives we have introduced don't have all the desired power required from coroutines. Yet, they don't extract the full power of the memory organization. For example, additional power can be obtained by including two new primitives to capture a continuation in a previously existing hook and to displace a continuation from one hook to a second hook. Such additions would not affect the performances of the frame pruner.

Acknowledgements

The author wishes to thank Nitsan Séniak for his valuable explanations about the working of his F1 compiler, Christian Queinnec for interesting discussions concerning the semantics of continuations and helpful remarks on the writing of this paper, and David De Roure for precious improvements brought to this paper.

References

[Appel 87] Andrew W. Appel: "Garbage Collection Can Be Faster Than Stack Allocation," *Information Processing Letters* 25, 1987, 275-279.

[Appel 89] Andrew W. Appel: "Simple Generational Garbage Collection and Fast Allocation", *Software-Practice and Experience*, 19(2), February 1989, 171-183.

[Bobrow & Wegbreit 73] Daniel G. Bobrow and Ben Wegbreit: "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM*, 16(10), October 1973, 591-603.

[Clinger *et al* 88] William D. Clinger, Anne H. Hartheimer and Eric M. Ost: "Implementation Strategies for Continuations," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, 124-131.

[Gabriel 85] Richard P. Gabriel: *Performance and Evaluation of Lisp Systems*, the MIT Press, 1985.

[Haynes *et al* 86] Christopher T. Haynes, Daniel P. Friedman and Mitchell Wand: "Obtaining Coroutines with Continuations", *Computer Languages*, 11(3/4), 1986, 143-153.

[Hieb *et al* 90] Robert Hieb, R. Kent Dybvig and Carl Bruggeman : "Representing Control in Presence of First-Class Continuations," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20-22, 1990, 66-77.

[Lieberman & Hewitt 83] Henry Lieberman and Carl Hewitt : "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM*, 26(6), June 1983, 419-429.

[Nakajima 88] Katsuto Nakajima : "Piling GC — Efficient Garbage Collection for AI Languages —," *Parallel Processing*, M. Cosnard, M. H. Barton and M. Vanneschi (Editors), Elsevier Science Publishers B.V. (North Holland), IFIP, 1988, 201-204.

[Rees & Clinger 86] Jonathan A. Rees and William Clinger, eds.: "The Revised^3 Report on the Algorithmic Language Scheme," *SIGPLAN Notices*, 21(12), December 1986.

[Seniak 91] Nitsan Séniak : *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels*, Thèse de Doctorat de l'Université Paris 6, October 1991.

Appendix: Defining coroutines from Scheme continuations

In this appendix we give a semantics for the primitives presented in Sect. 3. The formal behavior is obtained by defining each primitive from Scheme continuations.

```
;;; Coroutine creation
(define (coroutine fun)
  (call/cc
    (lambda (k)
      (fun (make-hook k)))))

;;; Escaping
(define (escape hook val)
  ((hook-ref hook) val))

;;; Suspension and resumption
(define (suspend/resume! hook val)
  (call/cc
    (lambda (k)
      (let ((old-k (hook-ref hook)))
        (hook-set! hook k)
        (old-k val)))))
```

```
;;; The hook abstraction
(define (make-hook k)
  (vector k))

(define (hook-ref hook)
  (vector-ref hook 0))

(define (hook-set! hook k)
  (vector-set! hook 0 k))
```

Table 1. Performances of different implementations for frame allocation. The first three columns correspond to the execution time of three stack implementations. The first one does no stack overflow test nor frame chaining, the second adds stack overflow test and the third adds both. We consider the second implementation as the "normal" stack implementation. The following four columns show the execution time of our heap implementation and the relative overhead associated with tagging, frame pruning and locality loss, compared to the normal stack implementation. Finally, the last column shows the performances of our organization with the stack/heap optimization and tag elimination. All execution times are expressed in seconds and the percentage appearing under each of them is the relative overhead of the corresponding implementation, compared with the normal stack implementation. Note that the sum of the relative overheads associated with frame chaining, tagging, pruning and locality loss is the relative overhead of the heap implementation.

Gabriel benchmark timings (in seconds)								
Benchmark name	Stack			Heap				Stack/ Heap
	-ovf test	normal	+frame chaining	total	tag	pruning	loc. loss	
Puzzle	0.822	0.823	0.836	0.890				0.883
	-0.1%		1.6%	8.1%	1.0%	0.1%	5.5%	7.3%
Boyer	0.885	0.930	1.058	1.210				1.102
	-4.8%		13.7%	30.1%	5.6%	4.3%	6.5%	18.5%
Destru	1.700	1.730	1.770	1.835				1.790
	-1.7%		2.3%	6.1%	1.4%	0.3%	2.0%	3.5%
Browse	1.590	1.625	1.700	1.840				1.765
	-2.2%		4.6%	13.2%	3.4%	2.5%	2.8%	8.6%
Div-rec	0.145	0.155	0.170	0.202				0.180
	-6.5%		9.7%	30.6%	6.5%	6.5%	8.1%	16.1%
Div-iter	0.140	0.140	0.140	0.142				0.140
	0.0%		0.0%	1.8%	0.0%	0.0%	1.8%	0.0%
Deriv	0.290	0.298	0.315	0.345				0.320
	-2.5%		5.9%	16.0%	3.4%	0.0%	6.7%	7.6%
Dderiv	0.338	0.345	0.372	0.415				0.388
	-2.2%		8.0%	20.3%	5.1%	2.9%	4.3%	12.3%
Triangle	8.895	9.190	9.935	10.810				10.420
	-3.2%		8.1%	17.6%	4.8%	1.1%	3.5%	13.4%
Traverse	5.730	5.875	6.355	7.200				6.495
	-2.5%		8.2%	22.6%	5.2%	0.3%	8.9%	10.6%
Tak	0.043	0.046	0.053	0.061				0.055
	-5.2%		15.8%	33.3%	7.8%	2.8%	6.9%	21.0%
Average	-2.8%		7.1%	**18.2%**	4.0%	1.9%	5.2%	**10.8%**

Table 2. Performances of a *same-fringe* solution based on coroutines. The columns show the total size of the frame heap, the primary buffer size, the total amount of memory allocated for frames, the percentage of frames surviving to a minor pruning, the number of minor and major prunings, and finally, the execution time using the stack/heap implementation.

Same-fringe with coroutines						
frame heap size (KB)	primary buffer size (KB)	frames allocated (KB)	copied frames	minor prunings	major prunings	execution time (secs.)
128	8	97735	1.77%	12997	49	9.4
128	16	97735	1.57%	6300	32	9.3
128	32	97735	0.87%	3101	26	9.4
256	64	97735	0.47%	1538	7	9.6
512	128	97735	0.25%	766	1	10.9

Table 3. Performances of a *same-fringe* solution with `call/cc`. The columns show the total heap size, the total amount of memory allocated (mainly formed of frames and continuations), the memory copied during garbage collection (including the trees), the number of garbage collections and the execution time using the stack/heap variation where the general garbage collector is triggered immediately after a frame heap overflow.

Same-fringe with `call/cc`				
heap size (KB)	allocated objects (KB)	copied objects (KB)	number of GCs	execution time (secs.)
7000	96117	189916	81	81.6
10000	96117	82063	35	43.4
12000	96117	58616	25	35.1
15000	96117	42204	18	29.3
20000	96117	28135	12	24.3

Table 4. Performances of a *same-fringe* solution with tree flattening. The columns show the total heap size for a Stop and Copy collector, the memory allocation in cons cells, the total memory copied during garbage collection (including the two trees), the execution time with a stack implementation and the execution time with our stack/heap implementation.

Same-fringe with tree flattening					
heap size (KB)	allocated objects (KB)	copied objects (KB)	number of GCs	execution time (secs.)	
				stack	stack/heap
10000	25781	23905	9	13.6	14.0
12000	25781	32905	9	16.8	17.1
15000	25781	11249	4	8.9	9.3
20000	25781	8759	3	8.2	8.4

An Implementation of an Applicative File System*

Brian C. Heck and David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405-4101 USA
Fax: +1 (812) 855-4829
Email: heckb@cs.indiana.edu

Abstract. A purely functional file system has been built on top of pure Scheme. It provides persistent structures and massive storage expected of file systems, without explicit side-effects like read and write. The file system becomes an additional, lazy argument to programs that would read from it, and an additional result from functions that would alter it.
Functional programming on lazy structures replaces in-place side-effects with a significant storage management problem, handled by conjoining the heap to the file system. A hardware implementation of reference counting is extended out to manage sectors, as well as the primary heap. Backing it is a garbage collector of heap and of disk (i.e. UNIX's fsck), needed only at reboot.

CR categories and Subject Descriptors:
D.4.2 [**Storage Management**]: Storage hierarchies; D.1.1 [**Applicative (Functional) Programming**]; E.2 [**Data Storage Representations**]: Linked representations; H.0 [**Information Systems**].
General Term: Design.

Additional Key Words and Phrases: Reference counting heap, mark/sweep garbage collection, hardware, Scheme, functional programming.

1 Motivation

The acceptance of functional, or applicative, programming languages has been encouraging. However, as often as they are taught and used, their scope of application has been restricted to formalism, to toy algorithms, and to simple systems. True, "purity" has been constrained at higher levels, to yield success stories like Lisp's (with side effects), ML's [11] (without laziness), and maybe Haskell's [5] (under UNIX I/O). With such constraints, however, they are unlikely to fulfill their acknowledged promise for parallel processing.

In a formal context applicative languages are often used as the foundation for semantics, for program analysis, and for rigorous documentation or proof. Although one can argue that important, non-"toy" algorithms (like divide-and-conquer tree searching or Strassen's matrix multiplication) were first invented and are best taught

* Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 90-02797.

using functional style; nevertheless, even these are used in production from C or Fortran source code.

Similarly, it has long been understood that Landin's streams [9], or lazy evaluation, allows a simple system to be expressed as an applicative program [4, 6] whose input is the stream **stdin** and whose output is the stream **stdout**. However, two things essential to a full operating system are still missing from such models: some kind of indeterminism, *e.g.* to interleave multiple, asynchronous inputs; and a persistent file system to cushion users against failures. A reliable file system is essential to allow recovery from a crash without rehearsing all of history, beginning when the system was first installed. Designing and implementing the file system is the goal of this project.

We expect two things from an Applicative File System (AFS). The first, already mentioned, is the ability to establish critical data structures in persistent media—commonly on a magnetic disk. Then all of ephemeral, primary memory might be lost, yet the system can rapidly be restarted from data on disk, recovering the system to a persistent configuration that recently preceded the catastrophe.

The second seems incidental: that files are larger than structures in main memory. Sometimes they are stored in secondary memory because of sheer bulk; often they are static over long periods. These properties derive from physical properties of the storage medium, and our own habits in using it.

However, AFS must behave within the constraints of pure functional programming: that the only operation is applying a function to arguments. Side effects are not available; thus, the programmer can neither "read" nor "write" a file. He can, however, traverse one stream-like parameter, and generate another one as a result. Either may be "bound" to a name in a distinguished environment that is commonly called a file directory.

The remainder of this paper is in six parts. The next section briefly contrasts common serial structures with, in particular, linked trees, setting up the importance of reference-counting hardware in Section 3. Similarly, Section 4 deals with streamed input and output, setting up the file system described in Sections 5 and 6, formally in the former, operationally in the latter. Finally, Section 7 presents a simple running example, and Section 8 contrasts this with past work and offers conclusions.

2 Side-effected Aggregates vs. Recopied Trees.

The only memory model we use is Lisp's (actually Scheme's) heap. Every data structure is built from binary nodes, a member of the recursive domain:

$$S = E + S \times S,$$

where E is a flat domain of elementary items. Conventional vectors, and their conventional in-place updates are not used. If static, they are easily mimicked with linked lists or (better yet) trees; so they are unnecessary. Moreover, it is possible to recopy a perfect tree of n nodes, incorporating one change, by creating only $\lg n$ new nodes. Therefore, side-effect-free updates are simulated cheaply and we can prohibit side-effects, consistent with functional programming and lazy evaluation.

Since the only non-trivial structure is a list, the only file structure is a persistent list. Because file updates, similarly, cannot be done by means of side-effects, it is possible to sustain two, perhaps several, successive incarnations of a single file simultaneously—merely as different lists, likely with shared substructure.

¿From the perspective of the database manager we have simplified the file system; we need only to keep the most current of several surviving incarnations, even while older ones are still bound and traversed elsewhere in the system. Later we shall describe how lists in memory migrate off to disk as files, and vice versa.

The inital perspective is that the file consistency problem has simply been traded for for a massive storage management problem. The manager or garbage collector needs to handle both binary nodes in primary (ephemeral) memory and sector-objects in secondary (persistent) memory. The remainder of this paper describes how that system was built and how it runs.

3 Reference Counting vs. Garbage Collection

We have built a system that has a hybrid storage manager; it has reference counting machinery both in main memory as well as on disk, and it is backed up with garbage collectors in both places.

A foundation to the system is the hardware implementation of Reference Counting Memory (RCM) [17, 18]. RCM is reported elsewhere, and the interesting story here is how AFS was laid over it. However, a brief overview of the hardware is necessary first.

3.1 Reference Counting in Hardware

RCM has been implemented as a device on a NeXT computer. Although its design would support full memory speed, the first prototype appears as eight megabytes of microsecond memory. It is configured as a half-meganode heap and an equal amount of serial memory that "roots" RCM. A second version is being designed for parallel processing; the description below presumes that there are several RCMs.

Every write of a pointer in RCM is a read-modify-write. That is, a new pointer is overwritten at a memory location only with removal of a former reference in that location during the same memory cycle. The algorithm to write a pointer is dispatched from a processor to memory where the following C code is executed uninterruptibly (as a remote procedure local to `destination`):

```
struct node
{
   integral RefCt;
   node *left, *right;
};
void store(pointer, destination)
  node *pointer, **destination;
{
   dispatch incrementCount(pointer);
```

```
    dispatch decrementCount(*destination);
    *destination = pointer;
}
```

All these operations occur essentially in parallel, subject to two constraints: the increment is dispatched before the decrement, and the former content at the destination is used just before it is overwritten. Both the fetch from *destination* and the store there occur during the same memory cycle (read-modify-write). The sequentiality of these three steps in uniprocessor C satisfies the constraints of a uniprocessor, although we intend them to be nearly simultaneous in hardware. Again, the increment and decrement are dispatched on-line, but they complete off-line.

Reference-counting transactions can be interleaved with similar ones dispatched from other memories to the same *destination*, as long as increments/decrements arrive at the targeted reference count as some merging of the orders in which they were dispatched. A unique, non-caching path between any source-destination pair, as on a bus or a banyan net meets this constraint.

At the destination address, both increments

```
void incrementCount(p)
  node *p;
{
    if Sticky(p->RefCt) ;else p->RefCt++ ;
}
```

and decrements occur as atomic transactions.

```
void decrementCount(p)
  node *p;
{
    if (Sticky(p->RefCt) || --(p->RefCt)) ;else FREE(p) ;
}
```

The use of FREE above indicates return to the local available space list. *Each of these three operations requires only finite time;* a node can return to available space still containing live, yet-counted pointers [15]. Thus, one memory location can, on one hand, handle a store and, on the other, act on a couple increments or decrements during one memory cycle.

RCM is controlled by reading or writing to special memory registers. Notably, new nodes of two types are allocated by reading from distinguished addresses. Because of this and because increments, particularly, must not be deferred, RCM is written-through the cache and read without caching.

In addition, RCM has on-board support for the rotations required in Deutsch-Schorr-Waite marking and has an on-line sweeper so that memory cycles for garbage collection can beat stop-and-copy. Early benchmarks show it running MachScheme (hobbled to use a recursive stack) faster than equivalent code using a RAM heap [18].

MachScheme is MacScheme [13] ported to the Mach operating system on the NeXT computer, and subsequently revised to use RCM for its heap of binary nodes. We acknowledge Lightship for granting source-code access to MacScheme.

3.2 AFS over MachScheme over RCM

AFS is then implemented over MachScheme which uses MacScheme's tagged pointers, extended for RCM hardware. RCM provides two types of nodes in its randomly allocated heap space. Nodes having addresses of the form $8n$ are terminal nodes (floating-point numbers); nonterminal binary nodes have addresses of the form $8n+4$.

MachScheme's tagging system and RCM design allows us 18-bit addresses into our file system. With 1024-byte sectors this yields a quarter-gigabyte file system. However, the effect of internal fragmentation reduces this. Moreover, the present tests were run on a prototype file-system of 4000 sectors, so to demonstrate a space-constrained configuration.

AFS's directory structure is modeled after UNIX [12, 14]. All files have an associated data node (dnode) similar to a UNIX inode except that there are no indirect pointers. Dnodes contain a single pointer to the unique first sector of the file.

Initially, we wanted to have files collected by reference counting on disk, like those in UNIX. When we examined our design for sharing data we discovered that we really needed to maintain reference counts on each sector. So we began to use typed pointers for references from heap space to sectors, but this required reference counting to interfere, dispatching additional disk transactions on every write or overwrite of a sector reference; they would be prohibitively expensive. What we needed was a second RCM dedicated to keeping reference counts for sectors with count information dispatched from the RCM MachScheme was using. We, therefore, stole enough nodes from the existing RCM to dedicate one per sector for the prototype AFS.

As a result, all the reference counts both for nodes in memory, and for sectors on disk are maintained by the same circuits in RCM. The difference is that a node whose count drops to zero implicitly returns to available space. However, sector reference counts are "nailed down" so that they cannot return to zero (and be handled like an RCM binary node, instead of as a whole sector); thus, their counts must periodically be scanned to find one-counts—to be returned to the sector pool. (This scan will be eliminated under the next version of RCM.)

3.3 Garbage Collection on Disk

It is desirable that any garbage collection be deferred as much as possible because such a lengthy traversal is slow, particularly under multiprocessing and on disk. Garbage collection on disk parallels UNIX's **fsck**; it is slow but it can be necessary—especially for recovery after a catastrophe.

Moreover, MachScheme has its own garbage collector, using RCM's mark/sweep hardware. In order that these two collectors not interfere with each other, the RCM-resident counts on disk sectors are stored in two parts corresponding to the RCM-sourced and to the on-disk references. Thus, MachScheme's internal collector recomputes the former, but does not traverse the disk. **fsck** roots from the current disk

directory, and traverses only the persistent memory; it requires a "quiet system," just like UNIX.

A sector on disk is composed of two parts: most of it is a compressed representation of the Scheme expression. A preorder-sequential representation is used, with tags indicating types immediately following. Circularities are detected and represented appropriately. Ten-bit pointers refer to positions in structure within that sector (uncounted), or to expanded 24-bit (counted) off-sector references. They form the second part of the sector—compressed at the end.

Thus, **fsck** need not traverse the first part of any sector, but must traverse and count the references at the end. And when a sector is condemned because its reference count drops to zero, all those forwarding references must be dereferenced.

3.4 Directories and Write-only-memory

The "file system" we discuss here is not built free-standing. In fact, it is nothing more than a permanent UNIX file of four kilosectors, which models a small, private random-access disk. UNIX utilities see it as a jumble of bits, and its UNIX-directory entry is irrelevant to the description below.

Within it is at least one—-possibly several successive—directories that we have generated. The most recent one of those represents the "current" file system. As discussed in Section 2, "creating," "deleting," or "changing" the binding of a file's name is effected by recopying the directory (itself a file) to include that alteration. Thenceforth, the `file_system` is bound to the newer directory.

After a file binding is created, the bound structure soon migrates onto disk. Because any file binding must now refer to a structure entirely disk-resident, huge files no longer need to consume heap in main memory. Manipulation of these files remains the same as if they were resident in that heap—except for the delays associated with file access, needed to copy sectors from it back into the heap.

The remaining problem is how to assure recovery of the file system after a system crash. The entire file system is still rooted in some ephemeral register of the computer, even though its entire content is resident on disk. However, the persistent information there is useless unless its root can be found.

One word of permanent memory (or disk) is set aside as *write-only-memory*, to receive a copy of the root of the file system after every update. (This contrasts with Section 2 protocols.) Operationally, this seems to be a side effect, but this binding is *completely invisible to a running system,* which uses its own ephemeral register as its root of the file system. In effect, the memory-resident root of the file system is copied into permanent memory, but nothing in this lifetime can use that copy; therefore, it may as well "not exist."

Whenever a conventional operating system is rebooted, it uses this distinguished address into permanent storage in order to root and to restore the file system as it was when last stable. The system is restored to a configuration from the not-too-distant-past, and comparatively little is lost, just as in UNIX.

Of course, the former streams, `stdin` and `stdout`, are lost during a catastrophe, and the reboot establishes a new `stdout'` and provides a new `stdin'`, presumably initiated by a user aware that the crash occurred and likely inquisitive of what files survive in the aftermath.

4 Stdin and Stdout

Stdin and **stdout** are classically treated as special streams/files with read-once and write-only privileges respectively. As discussed in Section 1, many researchers have suggested using streams for I/O. Merging these two approaches under AFS provides an elegant solution. (These ideas are not implemented in the current AFS due to the eager nature of Scheme.)

As in any file in the system, **stdin** should become manifest in main memory when a read operation is attempted upon it. The data structure representing **stdin** should be a lazy list. The tail of the list would be a suspension which, when thawed, creates a list with the head being a character read from the real input device (or a **port-not-ready** token) and the tail being a copy of the original tail suspension. This new pair is placed in the tail position of the manifest portion of the lazy list and control returns. To get a character from the **stdin** file one simply takes the head of the list. The user is responsible for keeping track of where she is in the **stdin** data structure. Each computational thread has an incarnation of the file system, so each may have a different opinion of what the next character to be read is. There is no restriction that keeps a thread from rebinding its own **stdin** to a different device or file.

No side-effects are necessary to maintain **stdin** because only the main thread can write to the master file system. Until the main thread updates its version of **stdin**, the entire stream of characters up to the last actually read from the device can remain memory resident (or swapped to disk). Updating the main file system to reflect the current consumption of **stdin** by the main thread would be achieved by the user installing what she perceives to be the tail of the current **stdin** as **stdin'** in a new file system. An ideal time to perform this update would be just before spawning a new thread. Due to reference counting, the list of characters would be automatically collected as soon as all references to the older **stdin** no longer exist.

Stdout may be handled by the classic **write-only-file** viewpoint with the file system piping the **stdout** file to a logical device (perhaps unique for each file system which is active). The operating system may map each of the logical devices to an actual output device performing any merge operations necessary.

5 Functionality

Notation from formal semantics is used to specify the types of AFS primitives.

Finite Sets

$\pi \in \mathrm{P}$	Persistent memory addresses
$\mu \in \mathrm{M}$	Main memory addresses
$\iota \in \mathrm{I}$	Identifiers as file names

Domains

$S = E + (S \times S)$	S-expressions
$\alpha \in A = P + M$	Addresses
$\delta \in D = I \rightarrow P_{\perp}^{+}$	Directories

$\rho \in R = A \to \mathcal{N}$ Reference Count
$\chi \in C = P \to M_\perp$ Cache

An interface to the file system has been provided through the following commands. All commands take an implicit file system and return a new system implicitly.

mkf : $D \to I \to S \to D$
Makes S persistent and associates I with that persistent structure in the new file system.

rmf : $D \to I \to D$
Returns a new file system with no directory entry for I

getf : $D \to I \to S \times D$
Returns the data structure associated with I in D plus a new D.

mkdir : $D \to I \to D$
Returns a new file system with an entry for the directory I.

lnh : $D \to I \to I \to D$
Returns a new file system with an association between the existing file (the second identifier) with the first identifier via a hard link.

lns : $D \to I \to I \to D$
Returns a new file system with an association between the existing file (the second identifier) with the first identifier via a soft link.

fsck : $D \to D$
Returns a new file system after a disk garbage collection. Intended to be run only by the main thread.

createfs : $I \to \perp$
Used to build a new, empty file system in the UNIX file I. The user must install the system to use it.

load-fs : $I \to D$
Installs the file system contained in the UNIX file I into the current Scheme session.

close-fs : $D \to \perp$
Stores the current file system into the UNIX file it originated from and removes all file systems from the Scheme session.

6 System Operations

AFS users have the ability to create hard and soft links in much the same manner as in UNIX. (*Cf.* UNIX commands `ln` and `ln -s` respectively.) As in UNIX, hard links are counted references and soft links are uncounted. One interesting change, due to the elimination of side-effects, is a modification in the behavior of hard links. (Soft

links behave exactly the same as their UNIX counterparts.) Following a hard link in UNIX returns the most current version of the file; this behavior stems from UNIX overwriting the file to install the new contents. However, AFS does not install the new data structure into the existing file system. Figure 1 contains before and after diagrams resulting from writing new "contents" to an existing hard link.

Initially, in the directory δ_1, hard links ι_1 and ι_2 correlate to the disk resident structure beginning with sector π_1. Issuing the command "(**mkf** ι_1 (**cons** 6 15))" creates a new file system in which δ_2, ι_2 still points to its old contents, but ι_1 does not. In δ_1 (the root of the old file system), both links still point to the old data structure, π_1.

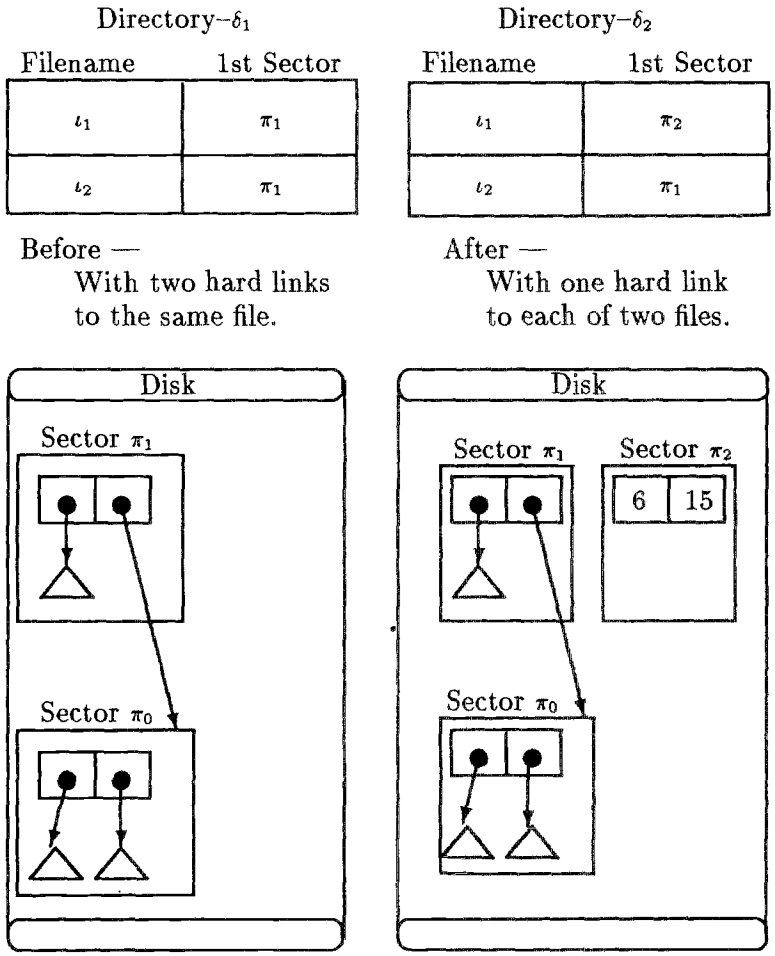


Fig. 1. Result of requesting (**mkf** ι_1 (**cons** 6 15)).

6.1 File Creation

A request for migration of a data structure α to disk initiates a preorder-sequential compression [7] of α into a set of sectors P^+. At some time (undefined but "soon") after a request to make a data structure a file, the data structure will become persistent. Currently, AFS has eager behavior inherited from Scheme, but implicit laziness could be inserted. Lazy file write operations are consistent with most current operating systems' views of files and buffers; until all buffers are flushed and the file committed, no guarantees can be made about the state of the file. Laziness should be transparent to the user except after a catastrophic system failure after which the file system is guaranteed to come back up in some stable, pre-existing state; we just cannot say exactly what state. (The user will learn to program confirmations to `stdout` that depend on successful commits to migration.)

Graphs migrated to disk may produce arbitrary graphs of sectors. In the trivial case of a flat list of nonterminal nodes, all having unary reference counts, the file becomes a flat list of sectors. One consideration for migration of graphs to disk is that multiply referenced nodes provide opportunities for data sharing and introduce a possibility of circularity. We assert that circularity in manifest graphs can be detected and sector reference counts corrected to allow reference counting to collect the disk structures [3].

As for suspensions, we would install them into the file system *verbatim*, that is, unexpanded. The suspensions would remain persistent on disk, but heap resident versions could be thawed and allowed to continue their work. Because Scheme is eager, however, we don't provide for suspensions yet.

Figure 2 shows the data structure rooted at μ_1 before and after it becomes persistent. The content of μ_1 is copied into a new heap node μ_2. This is immediately followed by adding an entry to the resident sector cache associating the address of the file's first sector, π_1 (a preallocated sector) with μ_2. (The resident sector cache which maps resident sector addresses to RCM addresses ($\chi : P \rightarrow M$) reduces the likelihood of having multiple copies of sectors heap resident, but more importantly, it prevents costly disk accesses if a needed sector is found to have a cache entry.) Next, the contents of μ_1's `car` and `cdr` fields are overwritten by forwarding pointers to the file's first sector, π_1.

The modifications to the original data structure to insert forwarding pointers are side-effects only at the level of implementation, not with respect to language semantics. Further, we assert there is no net consumption of heap space by the migration process. The recovery of the nodes containing forwarding pointers is completed during the next garbage-collection cycle. Thus the space needed for the new node μ_2 is offset by recovery (in the future) of the pre-existing node μ_1.

During compression, any multiply referenced, nonterminal nodes (μ_k) encountered are treated as separate trees and are rooted at the head of a new sector π_k (and have their contents replaced by forwarding pointers as μ_1's was in the above example). A pointer to π_k is written into the current, compression buffer and the μ_k is marked as disk resident (via a forwarding pointer as discussed below). Next, μ_k is stacked for later compression onto disk (rooted at π_k). The placement of multiply referenced, nonterminal nodes at sector heads allows the sharing of that single data structure by all its referents.

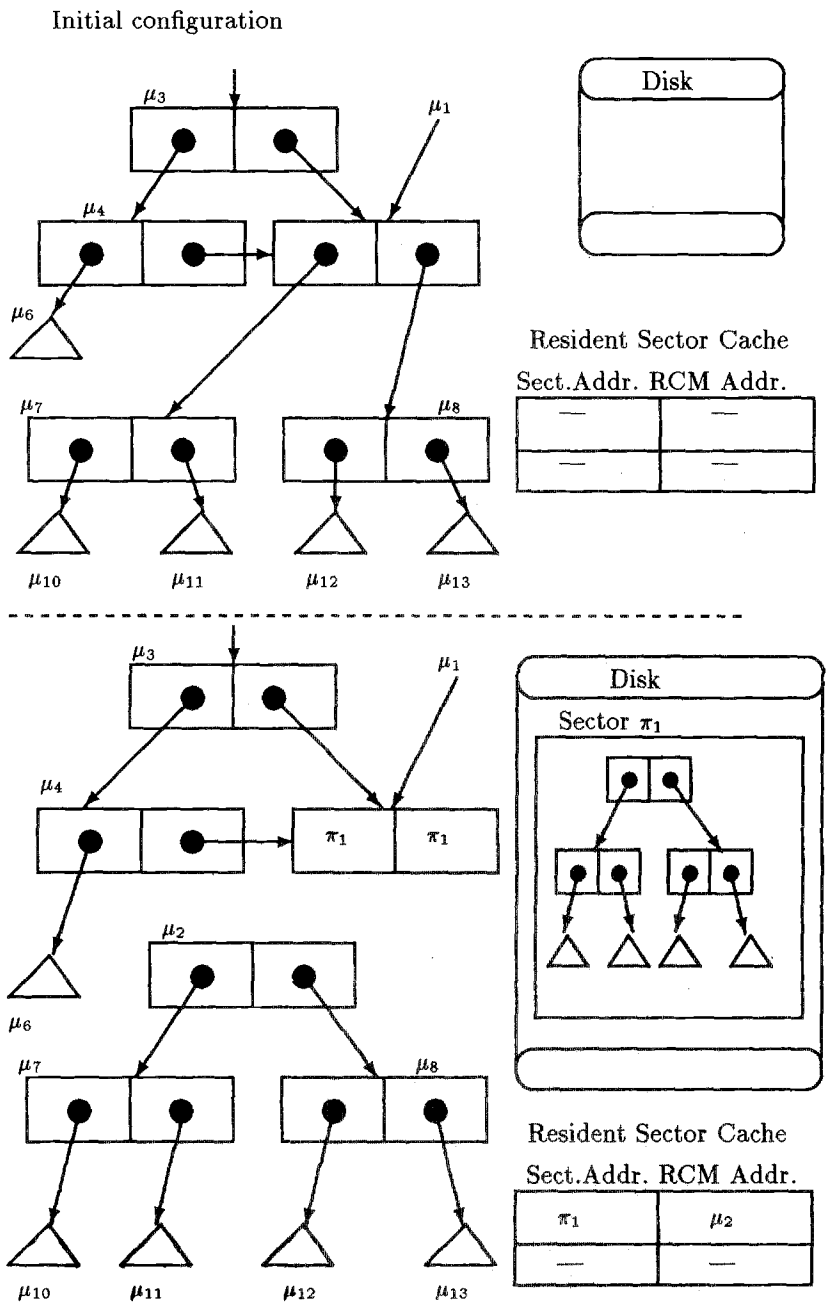


Fig. 2. Example of Data Migration: making μ_1 persistent.

While traversing the data structure, data from all uniquely referenced nodes and multiply referenced terminal nodes is copied into the current compression buffer. (If the buffer is full, a pointer to an empty sector is added and the buffer is written to the sector preallocated for it. Processing continues with the new, empty buffer.) Multiply referenced, terminal nodes are not assigned to unique sectors (to reduce internal fragmentation).

Sector pointers encountered in the heap during compression are added to the current sector as normal data along with one additional entry. At the end of every sector, a set of off-sector references, as mentioned in Section 3.3, is maintained. This list enables the disk's garbage collector to recalculate sector reference counts without scanning the sector.

6.2 File and Sector Migration from Disk to Heap

A getf command is treated as a request to load the root sector of the file into the heap. (The same mechanism is used to render sector addresses encountered in the heap into heap addresses.) The sector address is clashed against the resident sector cache. In the case of a hit ($\chi\pi \neq \perp$), the address of the heap resident copy of the data structure is returned. A cache miss ($\chi\pi = \perp$) forces a load of the desired sector into the heap and the addition of an entry in the cache to reflect that action. A traversal of the entire graph stored in the file will cause the file to be loaded into the heap one sector at a time. This is not to say that the entire file will be heap resident at any given time; if the file is sufficiently large that may not be possible).

7 Examples

The first example is a file editor taken from Friedman's and Wise's [4] paper. The example editor consumes a list of commands (stored in the file "commands"), applies the commands to the contents of another file "rhyme", and returns a list. We then install the list as a new version of the file "rhyme" in a new file system. The old version of "rhyme" is never deleted; it is automatically reclaimed when no longer in use.

The editor has six basic commands:

type	–Prints characters up to the next newline.
repos	–Repositions the cursor at the beginning of the text.
dele	–Deletes the character to the right of the cursor.
ins	–Inserts the given string to the left of the cursor.
find	–Finds the first occurrence of the given string to the right of the cursor. Success is reported if the string is found. Otherwise the cursor is advanced to the end of the text and failure is reported.
subst	–Locates the target string just as the find command and replaces it with the replacement string. Reports success or failure in the same manner as find.

```
MacScheme* Top Level Version 1.9 (Development)
>>> (createfs "fsys")
#t
>>> (load-fs "fsys")
#t
>>> (mkf "rhyme"  (rcmlist #\t #\h #\e #\space  #\q #\u #\i #\c #\k
         #\space #\b #\r #\o #\w #\n #\newline #\f #\o #\x))
#t
>>> (mkf "commands"
     (rcmlist
      (rcmlist find (rcmlist #\q #\u #\i #\c #\k #\space))
      (rcmlist type)
      (rcmlist dele)
      (rcmlist dele)
      (rcmlist ins (rcmlist #\d))
      (rcmlist subst (rcmlist #\newline) nil)
      (rcmlist repos)
      (rcmlist subst (rcmlist #\q #\u) (rcmlist #\s))
      ))
#t
>>> (mkf "rhyme" (editor (getf "commands") (getf "rhyme")))

#\:"find"(#\q #\u #\i #\c #\k #\space)
"Found: "(#\q #\u #\i #\c #\k #\space)
#\:"type"#\b#\r#\o#\w#\n
#\:"dele"
#\:"dele"
#\:"ins"(#\d)
#\:"subst"(#\newline)()
"Found"(#\newline)
#\:"repos"
#\:"subst"(#\q #\u)(#\s)
"Found"(#\q #\u)
#\:#t
```

```
>>> (getf "rhyme")
(#\t #\h #\e #\space #\s #\i #\c #\k #\space #\d #\o #\w #\n #\f
#\o #\x)
>>>
```

7.1 Example of Catastrophic Failure

When the system fails, AFS attempts to return to a previous, stable state. It will attempt to bootstrap the last version installed in the write-only memory discussed in Section 3.4. In the example below a system is created with thirty files (each with a unique copy of the same S-expression for convenience). While executing we cause the system to fail (via an interrupting control–C). To see how the system fared under a failure we list the files to determine which file was last created. According to the directory, "seed24" is the last file created intact. File "seed23" was probably in the process of being written to disk, but since the new file system (with "seed23" installed) was not written to the write-only-memory before the failure, the file is lost. This behavior is consistent with most current file systems.

```
MacScheme* Top Level Version 1.9 (Development)
>>> (createfs "fsys")
#t
>>> (load-fs "fsys")
#t
>>> (mk-manyfiles "seed" (rcmlist 1 (rcmlist 4 5.4)) 30)
^C
Program received signal 2, Interrupt
(gdb) q
prototype: ffsrnrcmsch
MacScheme* Top Level Version 1.9 (Development)
>>> (load-fs "fsys")
#t
>>> (ls)
("f" "seed24" "Fri Mar 27 21:10:07 1992
" "f" "seed25" "Fri Mar 27 21:10:07 1992
" "f" "seed26" "Fri Mar 27 21:10:07 1992
" "f" "seed27" "Fri Mar 27 21:10:07 1992
" "f" "seed28" "Fri Mar 27 21:10:07 1992
" "f" "seed29" "Fri Mar 27 21:10:06 1992
```

```
" "d" ".." "Fri Mar 27 21:09:49 1992
" "d" "." "Fri Mar 27 21:09:49 1992
")
>>> (getf "seed24")
(1 (4 5.4))
>>>
```

8 Conclusions

Present treatment of the migration of data between layers of memory under functional programming falls into two categories. Sometimes the problem is set aside, and the language enjoys an absence of restriction on transactions that are "outside" the program, as in ML, Lisp 1.5, or Scheme. Such languages do not extend very well to parallel processing.

Other languages attempt to isolate the problem: Backus's ASM in FP [1], William's and Wimmer's *histories* in FL [16], and Lucassen's and Gifford's *effect streams* in FX [10] are all serious efforts to encapsulate the "impure" file activity in order to isolate it from the "pure" functional portion of the language. Haskell [5] follows this tack. Alternatively, the time or scope for creation of persistent structures has been restricted [2].

In contrast, this research neither partitions nor encapsulates data. This treatment, in fact, would be transparent to the user if she were not required to participate, contributing some important declarations about her data. (ObjectStore [8], a general database system, similarly depends on only a few type assertions.) This is *experimental* work; one test of success is just to build a hierarchical memory in a purely functional environment without any "barriers." Another is to make it work well.

We have succeeded in the first test. The design and construction effort was not straightforward, but the difficulties we encountered all had an elegant solution, appropriately within the scope of the tools we had chosen.

Scheme is hardly an ideal language for this experiment; its lack of lazy evaluation corrupts the transparent implementation of UNIX pipes for **stdin** and **stdout**. However, a remarkably convincing test for such a generalized file system is to bring it to life under a general-purpose programming environment. We have built a production environment.

References

1. John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, **21**,8 (August 1978), 613–641.
2. David J. McNally and Antony J. T. Davie. Two models for integrating persistence and lazy functional languages. *SIGPLAN Notices*, **26**,5 (May 1991), 43–52.

3. Daniel P. Friedman and David S. Wise. Garbage collecting a heap which includes a scatter table. *Information Processing Letters* **5**,6 (Dec 1976), 161–164.

4. Daniel P. Friedman and David S. Wise. Aspects of applicative programming for file systems. In *Proc. of ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices* **12**,3 (Mar 1977), 41–55.

5. Paul Hudak, Simon Peyton Jones, and Philip Wadler (eds.). Report on the Programming Language Haskell. *SIGPLAN Notices* **27**,5 (May 1992), R1–R164.

6. Peter Henderson, Geraint A. Jones, and Simon B. Jones. *The LispKit Manual.* Tech. Monograph PRG-32 (2 vols.), Programming Research Grp., Oxford Univ. (1983).

7. Donald E. Knuth. *The Art of Computer Programming* **1** (2nd edition), Reading, MA, Addison Wesley (1973).

8. Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system, *Comm. ACM* **34**,10 (Oct 1991), 50–63.

9. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation: Part I. *Comm. ACM* **8**,2 (Feb 1965), 89–101.

10. John M. Lucassen and David K. Gifford. Polymorphic effect systems. *Conf. Rec. 15th ACM Symp. on Principles of Programming Languages* (Jan 1988), 47–57.

11. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* Cambridge, MA, MIT Press (1990).

12. D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Tech. J.* **57**,6 (Jul–Aug 1978), 1905–1930.

13. Lightship Software. MacScheme © *Version 1.9 development.* Beaverton, OR (1989).

14. K. Thompson. UNIX Implementation. *Bell System Tech. J.* **57**,6 (Jul–Aug 1978), 1931–1946.

15. J. Weizenbaum. Symmetric list processor. *Comm. ACM* **6**,9 (Dec 1963), 524–544.

16. John H. Williams and Edward Wimmers. Sacrificing simplicity for convenience: where do you draw the line? *Conf. Rec. 15th ACM Symp. on Principles of Programming Languages* (Jan 1988), 169–179.

17. David S. Wise. Design for a multiprocessing heap with on-board reference counting. In P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (Sept 1985), 289–304.

18. David S. Wise, Caleb Hess, Willie Hunt, and Eric Ost. Uniprocessor performance of a reference-counting hardware heap. Tech. Rept., Computer Science Department, Indiana Univ. (in preparation).

A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs

S. Duvvuru, R. Sundararajan, E. Tick,
A. V. S. Sastry, L. Hansen, and X. Zhong

University of Oregon, Eugene OR 97403, USA

Abstract. As large amounts of garbage are generated in concurrent logic programs, it becomes necessary to salvage used memory frequently and efficiently, with a garbage collector. Another approach is to detect when a data structure becomes garbage and reuse it. In concurrent languages it is particularly difficult to determine when a data structure is garbage and suitable for destructive update. Dynamic schemes, such as reference counting, incur space and time overheads that can be unacceptable. In contrast, static-analysis techniques identify data objects whose storage may be reused. Information from static analysis can be used by the compiler in generating appropriate instructions for reuse, incurring little or no runtime overhead. In this paper we present a new static-analysis technique for reuse detection. We present empirical performance measurements comparing the new scheme against binary reference counting (MRB). It is shown that a simple static analysis can achieve most of the benefits of MRB and improve the execution time.

KEYWORDS: static analysis, garbage collection, abstract interpretation

1 Introduction

In logic and functional programming languages, variables have the *single-assignment property*, i.e., a variable can be bound to a value at most once. In logic programming languages, a logical variable starts its life as an undefined cell and may later hold a constant, a pointer to a structure, or a pointer to another variable. These languages do not allow in-place update of bound data structures. Abstractly, the effect of an update is achieved by creating a new copy of the structure, with some new portion inserted into the copy.

The single-assignment property is elegant because it is possible to use the availability of data as a means of process synchronization, similar to dataflow computation. However, this has the undesirable effect of resulting in large memory turnover, due to excessive copying. Copying is wasteful in terms of both execution time and storage requirement. The lack of economy in memory usage results in the prodigious memory requirements by programs that update aggregate data structures. Garbage collection needs to be invoked frequently as the heap space is limited. Large memory bandwidth requirements and poor cache utilization hinder construction of scalable and high performance architectures for parallel logic languages.

If it is known that there are no references to a data object (from any process other than the one inspecting the object), then the structure can be reclaimed and used in

building other structures if necessary. The detection and reuse of such data objects can be done either at compile-time through static analysis, known as *compile-time garbage collection*, or at runtime (with additional data structures and instructions), or by a combination of both.

In this paper, we present (in brief) a new static-analysis method for compile-time garbage collection. We restrict ourselves to committed-choice logic programming languages [19]. In Section 2, we provide a brief introduction to committed-choice logic programming languages and review alternative methods for garbage collection in such languages: the MRB scheme and Foster and Winsborough's static-analysis technique [9]. In Section 3, we describe a new static-analysis scheme in the framework of abstract interpretation. In Section 4, we present performance measurements comparing our technique to MRB. Conclusions are summarized in Section 5.

2 Review

We start with a brief introduction to committed-choice logic programming before discussing garbage collection methods. A committed-choice logic program [19] is a set of guarded Horn clauses of the form $H :- A_1, \ldots, A_m : T_1, \ldots, T_n \mid B_1, \ldots, B_p$ for $m, n, p \geq 0$. H is the clause head, A_i is an "ask" guard, T_i is a "tell" guard, and B_i is a body goal. For *flat* languages, the guards can only be built-ins such as `X = t`.[1] We say that a goal a is reduced to a set of goals $body_i$ (or clause i commits) if a successfully *matches* with the head of a clause i (i.e., without causing any bindings to the variables of the goal) and the guards of clause i succeed. The ask guards must succeed without binding any goal variable, whereas the tell guards can bind variables.

A collection of clauses with the same head predicate and arity is known as a procedure. When a goal can commit to more than one clause in a procedure, it commits to one of them non-deterministically and the execution of other candidates is aborted. Otherwise, structures appearing in the head and guard of a clause cause *suspension* of execution if the corresponding argument of the goal is not sufficiently instantiated. For example, in order for a goal `foo(X)` to commit to the following clause,

```
foo([A|B]) :- true : true | bar(A,B).
```

the argument `X` of the goal must already be bound to a list structure, whose head (car) and tail (cdr) may be any term, even unbound variables. In the rest of the paper, structures appearing in the head are referred to as *incoming structures*.

Logic languages dynamically construct and decompose terms, usually allocated on a heap. The problem is to make efficient use of memory, specifically to reclaim heap space which is no longer accessible. Heap space is allocated in varying sizes, the management details of which do not concern us in this paper.

Although general reference counting [6] has the disadvantages of significant memory requirements and runtime overheads, the scheme can be efficiently approximated with binary reference counting. A prime example of this approach is the Multiple Reference Bit (MRB) garbage collection scheme [4]. Advantages of the MRB are ease of

[1] Upper case identifiers denote variables and lower case identifiers denote atoms.

hardware implementation and reasonable execution speed [13]. In this scheme, memory is incrementally reclaimed with special `collect` instructions that are generated for each incoming structure. We illustrate this technique with abstract machine code for the above clause:

```
foo_1_top:
        try_me_else          foo_1_0  ; set up continuation
        wait_list            1        ; [A|B]
        read_car_variable    1,2      ; X2 = A
        read_cdr_variable    1,3      ; X3 = B
        collect_list         1        ; collect list cell
        put_value            1,2      ; set up call to bar(A,B)
        put_value            2,3
        execute              2,bar_2_top
```

The `collect_list` instruction attempts to reclaim the list cell. The attempt succeeds if the MRB is *off*, in which case the cell is added to a free list. Separate free lists are maintained for structures of varying sizes. When a structure is created, instead of freshly allocating it from the heap, it is allocated from a free list. If the free lists are used in a LIFO (last in, first out) manner, it is likely that the reclaimed cell is still in the cache.

As mentioned above, `collect` instructions are generated for each incoming structure. The MRB method *per se* does not involve compile-time determination of when `collect` instructions are needed, and when they are not. `Collect` will reclaim the memory allocated to a structure *only if* the structure is singly referenced. In other words, all structures, regardless of their potential for reuse, will incur the overhead of `collect` instructions.

In contrast to MRB, which is a dynamic reuse method, we can apply static-analysis techniques to the problem. The application of static program analysis to infer properties of programs, and the use of this information to generate specialized and efficient code, have proved to be quite successful in functional and logic languages. Analysis techniques for functional languages (e.g., [11, 10]) do not carry over to logic programming because of the complexity of unification and the representation of bindings of the logical variable. Analysis techniques for sequential Prolog (e.g., Mulkers [15] and Bruynooghe [2]) do not extend easily to concurrent logic languages, where no assumptions can be made on the order of execution of the goals or about the interleaving of their execution. On the one hand, our approach is simpler than that for Prolog because committed-choice programs do not backtrack and therefore do not require trailing. On the other hand, the analysis is more complex because we cannot reason about when some goal will start or finish executing, and the related problem of concurrent interleaving.

One of the major implementation issues in functional languages is the efficient implementation of the update operator on array data structures. The straightforward implementation would take linear time in the size of the array, as opposed to constant time update in imperative languages. Through static analysis of liveness of the aggregates, the update operations can be optimized. The related research done in this area are detecting single-threadedness of the store argument of the standard semantics of imperative languages [18], and update analysis for a first-order lazy functional language with flat aggregates by defining a non-standard semantics called *path semantics* [1]. Both analyses have been formulated for sequential implementa-

tions.

Another important implementation area is shape analysis, the static derivation of data structure composition. Recent work by Chase *et al.* [3] describes a more accurate and efficient analysis technique than previous methods. They also describe how this storage shape graph (SSG) method can be followed by reference-count analysis. We do not discuss shape analysis for logic programs here. Furthermore, recently there has been work on reordering the expressions in a strict functional language with the objective of making most of the updates destructive [17].

Foster and Winsborough [9] address the reuse problem for concurrent logic programs with static analysis. They sketch a collecting semantics for Strand programs in which a program state is associated with a record of the program components that operated on it. The collecting semantics is then converted into an abstract interpretation framework by supplying an abstract domain in order to identify single consumers. The details of their method have not yet been reported (unpublished draft [8]), hence it is premature to compare their scheme with ours.

3 Proposed Static Analysis

The reference counting schemes previously reviewed have the main deficiency of excess runtime overheads. We are not aware of any successful (efficient) implementation of general reference counting for a parallel language. Binary reference counting, for instance MRB, adds runtime overheads to the abstract machine instruction set in which it is implemented. In this section, we propose a dataflow-analysis technique to detect opportunities for reuse in committed-choice programs and use this information (in the next section) to generate reuse instructions.

There are four distinct ways in which a variable can be used for sharing information in concurrent logic programs. They are: Single producer-Single consumer (*SS*), Single producer-Multiple consumer (*SM*), Multiple producers-Single consumer (*MS*), and Multiple producers-Multiple consumer (*MM*). Since a variable may be bound at most once in logic languages, the notion of multiple producers implies that there are several potential producers but only one succeeds in *write-mode* unification. In a successful committed-choice program, all other potential producers perform *read-mode* unification. Ueda [23] defines the class of *moded* FGHC programs to be those in which there are no competing producers. In legal moded FGHC programs, *MS* and *MM* variables do not exist. Saraswat [16] proposes a related language, Janus, which allows only *SS* variables, each appearing only twice: as an "asker" and "teller," explicitly annotated by the programmer.

The purpose of our analysis is to determine which type of communication, *SS* or *SM*, applies to each of the program variables. This information is used by the compiler to generate reuse instructions (see Section 4.1). The algorithm is safe for non-moded programs, but little reuse will be detected in programs where multiple producers and consumers abound. Since most (not all) programming paradigms can be implemented in moded programs, we expect accurate information to be produced from our simplified analysis, for a large class of programs.

Structures appearing in the head and guard of a clause imply *incoming* data. Thus if such a head structure is determined to be reusable, we can recycle its cells when constructing a structure in the body. Consider the following clause:

```
p(X,t(L,C,R),Y) :- X < C : Y = t(NL,C,R) | p(X,L,NL).
```

If the second argument in the head, `t(L,C,R)`, may be reused, it would be best to reuse it when constructing the structure `t(NL,C,R)` in the body. This is known as *instant reuse* or *local reuse*. However, if no immediate use is possible, the reclaimed cells should be stored away for future use. This is known as *deferred reuse*, and is equivalent to the `collect` operation defined earlier in the context of MRB.

We assume that the programs have been translated to a *flattened, canonical form*.[2] In this form, all head arguments are unique variables. Flattening is achieved by moving all unification in the head into the guard. Also, all procedure calls contain only variables. For example, a call to procedure `P(t)` is flattened into a unification `X=t` and a subsequent call `P(X)`. In the following presentation, data objects that have a single producer and single consumer are referred to as *single-threaded* and data objects that have multiple consumers are referred to as *multiple-threaded*. We assume, for the analysis, that the top-level components of structures appearing in the head or guard are bound to non-variables. This assumption is discharged by a runtime check (discussed in Section 3.3). We also assume that *structure sharing analysis* has been done, and that the results of the analysis are available. We envision sharing analysis similar to [21, 14], with two modifications. First, the analysis must work for concurrent languages. Second, if it is determined that two variables may share, no subsequenting grounding can undo this sharing.

3.1 Multiple Threadedness of Structures and Components

To propagate the threadedness information safely and precisely, we have to understand how the multiple threadedness of a structure affects the multiple threadedness of its components and vice versa. The three questions that arise are:

- Is a substructure of a multiple threaded structure multiple threaded?
- Is it always the case that the structure becomes multiple threaded if one of its substructures is multiple threaded?
- How does threadedness of one component of a structure affect another component within the same structure?

If a structure is multiple threaded, it means that there are several consumers accessing the structure. Each consumer can potentially access any substructure, implying that each substructure may also have multiple consumers. Thus multiple threadedness of a structure implies the multiple threadedness of its components.

Multiple threadedness of a component of a structure, however, does *not always* mean that the structure becomes multiple threaded. Suppose a structure is built in the body of a clause and it contains a head variable which is multiple threaded. A head variable is simply a reference to an incoming argument which has already been created. Only a *pointer* to that actual parameter resides in the structure built in the body. Because the variable is not created inside the current structure, the reuse of the structure does not affect the contents of the multiple-threaded component. Therefore the structure does not become multiple threaded.

[2] This is to simplify the presentation and is not a limitation of our method.

Now suppose a structure is built in the body of a clause and it contains a variable local to the clause body (i.e., the variable does not appear in the head) and the variable is multiple threaded. If the implementation creates variables *inside* structures, then the the reuse of the structure may change the contents of the multiple-threaded variable. In this case, we have to make the structure multiple threaded, to be safe. If the implementation creates variables *outside* structures (the structure arguments are linked to the variables by pointers), then multiple threadedness of a component would *never* make the structure multiple threaded.

The answer to the third question depends on the sharing of the components of the structures. If two subterms of a structure share, then multiple threadedness of one makes the other subterm multiple threaded. In this paper, we assume that we have the sharing information for all the program variables.[3]

3.2 Abstract Domain and Operations

A variable can take values from the two-point complete lattice L whose least element is *SS* (Single producer/Single consumer) and the top is *SM* (Single producer/Multiple consumer). The dataflow analysis is summarized below. A formal treatment of the analysis is given in [22].

Initialization The initial abstraction of the threadedness of variables is based on the number of occurrences of a variable (and the variables it shares with) in the head and the body. All occurrences of the same variable in the head and the guards are counted as a single occurrence and each occurrence of a variable in the body is counted individually.

If a variable occurs two or fewer times, it is initialized to *SS*. If it occurs more than two times, it is initialized to *SM*. Note this implies that variables that occur only in the guard (the *Ask* or the *Tell* part) are initialized to *SS*. The variables that occur only in the guard will inherit their threadedness from other structures with which they are matched/unified.

As an example of initialization and its interaction with sharing, consider the following:

```
p(X,Y,Z) :- true : Y=Z | q(X,Y), r(Z).
```

Variable `X` is initialized to *SS* because it occurs only twice. Assume that the tell goal may cause `Y` and `Z` to share. Without considering sharing, the number of occurrences of `Y` and `Z` are two each. However, considering sharing, we count four occurrences of each. Thus we initialize each to *SM*.

Head-Goal Matching and Guard Execution The reduction of a goal by matching with the head of a clause and successfully executing the guards is approximated as follows. Since we are dealing with canonicalized programs, goal-head matching and the execution of guards involves a sequence of $X = Term$ equations, where X is a variable and *Term* may be a variable, a constant or a structure. If both X and *Term* are *SS* then they remain *SS*. Otherwise they and their subterms become *SM*, and so do the variables that share with them. This gives us the initial abstract entry substitution.

[3] If such information is not available, then we assume that all variables within a structure share, and perform the analysis.

In reality, we further differentiate the guard into *Ask* and *Tell* parts, as reviewed in Section 2. This is a minor distinction, but it allows a more precise derivation of threadedness information. Since Ask guards cannot bind goal variables, threadedness does not propagate among goal variables. Tell guards are treated most generally, allowing threadedness propagation among both goal and clause variables. For a formal description of the abstract execution mechanism, see [22].

Local Fixpoint Computation Given an entry substitution for a clause, its exit substitution is computed as follows. For a unit clause, the exit substitution is the same as the entry substitution. Otherwise, compute the success substitution of literal one, using the entry substitution of the clause as the call substitution. Using the success substitution of literal i as the call substitution of literal $i+1$, compute the success substitution of literal $i+1$. Since the body goals may execute concurrently, we need to safely approximate their interleaved execution. This is accomplished by treating the success substitution of the last body goal as the call substitution of the first body goal and repeating the above process until a fixpoint is reached [5]. This is called "local fixpoint computation." The success substitution of the last body goal (after the local fixpoint computation) is the exit substitution of the clause.

Abstract Success Substitution When all the body goals have been solved, the current substitution is known as the *abstract exit substitution* of the clause. Restricting the abstract exit substitution of a clause i, to the variables in the environment of goal a (which unified with the head of clause i) and then composing it with the abstract call substitution of a gives us one *abstract success substitution* of goal a with respect to clause i. The least upper bound (lub) of the abstract success substitutions of all the clauses whose heads matched with the goal a is called the abstract success substitution of goal a. Since we do not know at compile time which clause of a procedure will commit to a goal, we have to take the least upper bound of the abstract success substitutions of all the clauses in a procedure.

Global Fixpoint Computation Since a program in general may contain recursive clauses, finding the abstract success substitution of goal involves a fixpoint computation. To compute the abstract success substitution of a goal defined by recursive procedure, we use the non-recursive clauses of the procedure to compute an initial approximate success substitution. Using this as the first approximation of the success substitution of the recursive calls in a recursive clause, we compute the success substitutions of the recursive clauses and use this as the next approximation. The process is iterated until a fixpoint is reached. We call this "global fixpoint computation." Both the local and global fixpoint computations will terminate since our abstract domains are finite and the abstract domain operations are monotonic [20].

We outline the analysis for the QuickSort program as listed in Fig. 1. First, the initial abstract substitution is computed for each program variable in each clause. All program variables are initially assigned *SS*. On completing the analysis, we obtain abstract substitutions for all program variables, which identifies which of the incoming arguments are of types *SS* and *SM*. In the example above, it was determined that all four potential applications of reuse are safe: variables X_1 in $q/2$, $a/3$,

```
q(X1,X2) :- X1=[] : X2=[] | true.
q(X1,X2) :- X1=[X3|X4] : X9=[X3|X8] |
     s(X4,X3,X5,X6),
     q(X5,X7),
     q(X6,X8),
     a(X7,X9,X2).

a(X1,X2,X3) :- X1=[] : X2=X3 | true.
a(X1,X2,X3) :- X1=[X4|X5] : X3=[X4|X6] | a(X5,X2,X6).

s(X1,X2,X3,X4) :- X1=[X5|X6], X5 >  X2 : X4=[X5|X7] | s(X6,X2,X3,X7).
s(X1,X2,X3,X4) :- X1=[X5|X6], X5 =< X2 : X3=[X5|X7] | s(X6,X2,X3,X7).
s(X1,X2,X3,X4) :- X1=[] : X3=[], X4=[] | true.
```

Fig. 1. Flattened, Canonical Form of QuickSort Program

and $s/4$. In these four cases, the incoming variables represent incoming structures in the head of the unnormalized clauses. A compiler can generate the appropriate reuse instructions after the code for inspecting the head arguments. In the next section we discuss the problem that arises if at runtime, unbound variables occur in the top level of the structure.

3.3 The Problem of Unbound Structure Arguments

The presence of uninstantiated variable(s) in the top-level of a structure renders the structure unsuitable for reuse, even if the structure is single threaded. The reason is because a producer of the unbound variable may bind its value *after* the enclosing structure has been reused! This might result in an erroneous unification failure.

This problem may be avoided by always allocating variable cells outside of structures and placing only pointers, to the variable cells, inside structures. While this permits reuse, it introduces extra dereference operations and may also increase memory consumption. These tradeoffs have been quantitatively analyzed by Foster and Winsborough [9].

If outside allocation is not the storage management policy, then a variable check of the top-level arguments must be conducted at runtime. Usually when a structure is inspected by a consumer, the components of the structure are decomposed, and copies of the elements placed in machine registers, or in the environment of the consumer. During this decomposition, the runtime check is relatively cheap to perform.

4 Experimental Results

In this section we review two alternative committed-choice language instruction-set extensions for exploiting reuse information. The extensions are from the Strand abstract machine and the PDSS emulator. These extensions are similar, and deserve some explanation to put our empirical performance measurements in context. A performance comparison between our method and MRB is presented. The main purpose of this analysis is to illustrate that our analysis technique in fact works!

4.1 Reuse Instruction Sets

Foster and Winsborough [9] describe a reuse instruction set for the Strand abstract machine. The extension includes:

`test_list_r(L,H,T)` — If a register `L` references a list structure, then place a reference to the list in reuse register R, and place references, to the head and tail, in `H` and `T`, respectively.

`assign_list_r(L)` — Place a reference to the list structure, referenced by reuse register R, into register `L`, and let the structure pointer point to the head of the list.

`reuse_list_tail_r(L)` — Place a reference to the list structure, referenced by reuse register R, into register `L`, and let the structure pointer point to the tail of the list. Here we avoid the write mode unification of the head cell.

The reuse instructions use an implicit operand, the reuse register R, or a set of reuse registers. The reuse register is effectively a fast "free list" of currently reusable structures. This method is an efficient way of managing reusable dead structures for deferred reuse.

In contrast, the PDSS system [12] implements a reuse instruction set, designed around the MRB method, for the KL1 abstract machine. Deferred reuse is based on the `collect` operation which places reusable structures in free lists. When a new structure is required, it may be allocated from the free list. PDSS also includes includes instructions for instant reuse. The extensions for instant reuse include:

- `put_reused_func(Rvect,OldVect,Atom)` — Set the `Rvect` to point to the same location as `Oldvect`, set the name of functor to `Atom`.
- `put_reused_list(Rlist,OldList)` — Set `Rlist` to point to the same location as pointed by `OldList`.

Instant reuse is more efficient than deferred reuse since the intermediate move onto the free list is avoided. However, recall from Section 3.3 that a runtime variable check is needed for each structure argument. In our empirical experiments with reuse analysis, presented in the next section, the PDSS system was used. Since PDSS allocates unbound variables outside of structures, variable checks are not needed, so our comparison is fair.

4.2 Performance

Measurements were made with the PDSS emulator running on a Sun SparcStation I. Six small benchmark programs were analyzed: append, insert, primes, qsort, pascal, and triangle. Insert constructs a binary tree of integers. Prime uses the Sieve of Eratosthenes to generate prime numbers. Qsort is the standard quicksort algorithm previously shown. Pascal generates the n^{th} row of Pascal's Triangle. Triangle finds all solutions to the triangle puzzle of size 15. For each benchmark, three compiled versions were generated:

Naive: A version with no collect nor reuse instructions, used as a basis for comparison.

Collect: A version with collect instructions as generated by the existing PDSS compiler.

Reuse: A version with instant reuse instructions (where appropriate) and no collect operations.

Both reuse *and* collect can be used together in a hybrid scheme. However, in this study we wish to compare the efficacy of reuse with that of the MRB scheme, and therefore we do not present results concerning the hybrid. The heap usage patterns in the benchmarks are presented in Table 1.

Table 1. Heap Usage and Execution Speed: Comparison of No Optimization, Collect, and Reuse

	Heap Usage (Words)			Execution Time (sec)			
Program	**Naive**	**Collect**	**Reuse**	**Naive**	**Collect**	**Reuse**	**% Save ‡**
insert	1,500,000	6,314	6,314	52.2	53.6	50.5	5.8
append	5,000,000	6,202	6,202	111.0	114.3	106.5	6.8
prime	323,786	12,128	12,158	11.1	11.3	10.5	7.1
qsort	8,000,000	61,725	61,725	234.0	237.5	221.7	6.7
pascal	167,070	127,072	147,270	12.1	12.9	12.3	4.7
triangle	543,809	539,523	543,809	60.5	63.8	60.5	7.9

‡ Reuse compared to Collect Optimization

The measurements in Table 1 reflect the behavior we expected from the benchmarks. The benchmarks illustrate classes of full, partial, and no-reuse programs. Insert, append, prime and qsort extensively use stream-based single producer/single consumer communication. Our algorithm predicted potential for full instant reuse, as confirmed in the table. In these benchmarks the heap memory requirements of the reuse and collect versions are nearly identical. This demonstrates that it is possible to achieve as much efficiency as collect in benchmarks where a large number of single-threaded structures are constructed.

In pascal, where only $50\% = (167,070 - 147,270)/(167,070 - 127,072)$ of reusable structures were actually reused, the heap requirements of the reuse version are only slightly higher than that of the collect version. Inability to reuse all local data is due to the imprecision of the analysis. In triangle, the board structure is multiple threaded, i.e., has multiple consumers. The collect operations almost never succeed in reclaiming memory and the memory requirements of the collect and naive versions of the program are nearly the same. Since reuse is not possible, the reuse version is the same as the naive version.

The memory requirements of the naive versions of the benchmarks can be several times higher than the reuse and collect versions. The extent of memory reuse is highly program dependent however. These measurements are meant only to illustrate how our algorithm can exploit reuse when conditions are right.

To further illustrate the effectiveness of static analysis, the execution times of the benchmarks are also presented in Table 1. By compiling reuse into the program, the execution speed is consistently better than that obtained through the MRB

optimization. By implementing reuse more efficiently (than in PDSS), the savings may be even higher. Note that in PDSS, even after stop-and-copy garbage collection, the naive version of a benchmark runs as fast as, if not faster than, the collect version. This lends support to our claim that in the absence of special hardware to implement MRB, a good garbage collector is important. To better the combination of reference counting and a good garbage collector, memory reuse is necessary.

In programs with a preponderance of multiple-consumer communication, the collect operation simply adds runtime overhead without reclaiming memory. In such a case, the program performs better if the collect operations are simply removed. This is evident in triangle, for instance, where our analysis determined that there is no scope for reuse. Thus the reuse version (i.e., naive version) outperforms the collect version.

In programs where the majority of the structures have single consumers, the collect operation is avoided wherever instant reuse is possible. This reduces the overhead of free-list management. The reuse version is 6–8% faster than the collect version in insert, append, prime, and qsort, where 100% instant reuse was possible. Even in pascal, where instant reuse was only 50% as memory-efficient as collect, the reuse version was 4.7% faster.

5 Conclusions and Future Work

We have shown that simple compile-time analysis can have as much benefit, in reusing structure memory during program execution, as a hardware-based scheme such as MRB. We have sketched an algorithm (formal details of which can be found in [22]) that enables the compiler to automatically emit reuse instructions. Empirical results indicate that reuse is comparable to the MRB scheme in terms of amount of memory saved, and the execution speed is improved (4.7%–7.9%) in some cases.

One source of imprecision in our analysis is the inability to determine at compile time the execution ordering among the goals. For instance, we anticipate that our analysis is less precise than similar analysis for sequential Prolog, but there is little hope of remedying this. Another source of imprecision is that mode analysis is currently not taken into account, potentially resulting in overly conservative derivation of the threadedness of output variables [22]. Furthermore, if structure sharing analysis can provide detailed information about the shared subterms, then propagation of threadedness can be made more precise. Of these three problems, integration of mode analysis information seems most promising.

A key input to the analysis method is sharing information. Future plans include implementing both sharing and reuse analysis within the Monaco system, a native-code, shared-memory multiprocessor compiler for flat committed-choice languages [7]. Instant reuse instructions must be incorporated into the architecture at the appropriate level. With a robust implementation, the utility of the technique, for large benchmarks, within a high-performance system, can be determined.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds granted by Sequent Computer Systems Inc. We thank the anony-

mous referees for helping us clarify the role of sharing in our analysis.

References

1. A. Bloss. *Path Analysis and Optimization of Non-Strict Functional Languages.* PhD thesis, Yale University, Dept. of Computer Science, New Haven, May 1989.
2. M. Bruynooghe *et al.* Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *International Symposium on Logic Programming*, pages 192–204. San Francisco, IEEE Computer Society, August 1987.
3. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–309, White Plains, NY, June 1990. ACM Press.
4. T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.
5. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232. Austin, MIT Press, October 1990.
6. L. P. Deutsch and D. G. Bobrow. An Efficient Incremental, Automatic Garbage Collector. *Communications of the ACM*, 19:522–526, September 1976.
7. S. Duvvuru. Monaco: A High Performance Architecture for Concurrent Logic Programs. Master's thesis, University of Oregon, June 1992.
8. I. Foster and W. Winsborough. A Computational Collecting Semantics for Strand. Research report, Argonne National Laboratory, 1990. unpublished.
9. I. Foster and W. Winsborough. Copy Avoidance through Compile-Time Analysis and Local Reuse. In *International Symposium on Logic Programming*, pages 455–469. San Diego, MIT Press, November 1991.
10. P. Hudak. A Semantic Model of Reference Counting and Its Abstraction. In *Conference on Lisp and Functional Programming*, pages 351–363, Cambridge, 1986. ACM Press.
11. P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Languages. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 300–314, New Orleans, January 1985. ACM Press.
12. ICOT. *PDSS Manual (Version 2.52e).* 21F Mita Kokusai Bldg, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, February 1989.
13. Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming*, pages 907–921. Cleveland, MIT Press, October 1989.
14. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, pages 154–165. Cleveland, MIT Press, October 1989.
15. A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *International Conference on Logic Programming*, pages 747–762. Jerusalem, MIT Press, June 1990.
16. V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.
17. A. V. S. Sastry and W. Clinger. Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. Technical Report CIS-TR-92-14, University of Oregon, Computer Science Department, 1992.

18. D. Schmidt. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.
19. E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
20. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge MA, first edition, 1977.
21. R. Sundararajan. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. Technical Report CIS-TR-91-06, University of Oregon, Department of Computer Science, October 1991.
22. R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming.* Washington D.C., MIT Press, November 1992.
23. K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.

This article was processed using the LaTeX macro package with LLNCS style

Finalization in the Collector Interface

Barry Hayes

bhayes@cs.stanford.edu Stanford University, Department of Computer Science, Stanford, CA 94309, USA

Abstract. When a tracing garbage collector operates, it treats the objects and pointers in the system as nodes and edges in a directed graph. Most collectors simply use the graph to seek out objects that have become unreachable from the root objects and recycle the storage associated with them.
A few collector designers have hit on the idea of using the trace to gather other information about the connectivity of the graph, and notify user-level code when an object is reachable from the roots, but only in a restricted way. The user-level code typically uses this information to perform some final action on the object, and then destroys even the restricted access to the object, allowing the next pass of the garbage collector to recycle the storage. *Finalization* is useful for appropriating the power of garbage collection to manage non-memory resources. The resource in question can be embodied in a memory object with finalization enabled. When the memory resource is reachable only through restricted paths, the non-memory resource can be recycled and the restricted access destroyed. The users of the resource need not coordinate to manage, nor do they need to know that the resource is precious or needs finalization.
This paper presents system-level details of five different implementations of finalization in five different systems, and language-level details of several languages that have defined similar mechanisms. These comparisons highlight several areas of concern when designing a system with finalization.

1 Introduction

Garbage collection is sometimes trying to serve two antithetical goals. First, some languages and systems see collection solely as a way to make a finite memory resource appear larger. The programmer need not worry about memory because there is a large supply, and the garbage collector helps maintain the fiction. In this role, the collector must be invisible, lurking in the shadows, and no side effects of collection should be apparent to other code in the language.

Other languages and systems see garbage collection as a valuable opportunity to learn more about the connectivity of objects, and try to make the information gleaned by the garbage collector available to other code. The most common use of this information is to implement weak or soft pointers. Another is to drive *finalization.* Finalization takes many varied forms, but the goal is to communicate information about the connectivity of objects from the garbage collector to other elements of the system.

With finalization this connectivity information is used to let a module that manages a resource know when no module other than itself has any remaining pointers to an object in its resource pool. When it knows this, it can invoke code on the object to do any clean-up that might be required, and return the resource to the pool. Without this connectivity information, the users of such a resource are required to cooperate in the management of the object, and the code required can be difficult to write, verify, and maintain. Making the connectivity information available allows the resource to be managed in a simple way, and lends the power of garbage collection to the management of other resources.

2 A Short History

Finalization seems to have grown from two different roots in computer science: the desire to have soft pointers for ease in engineering, and the desire to do correctness proofs in the presence of exception handling.

Soft pointers, also called weak pointers, are pointers that are not traced or counted by the garbage collector. Typically, when the collector notes that there are no hard pointers to an object, it collects the storage associated with the object and sets the soft pointers to a known value, often zero or NIL[1]. Soft pointers allow a process to monitor an object and know if it has been collected without interfering with the collection of the object.

Closely related to soft pointers are *populations*. A population is a clever kind of hash table — a "key" can be used to find a "value". Often these keys and values are simply objects, and the address of the key is hashed to find the location of the key/value pair. But when all other references to the key have vanished from the system, it will never be used to look up the associated value. A population differs from a simple hash table in that it is in bed with the collection system and does not allow this reference to retain the key[2]. Populations exist in many modern Lisp system.

The other related concept, error recovery, is particularly relevant to systems where the central focus is more on the data types than on the code, and where correctness concerns are important. Many languages include a facility whereby a block of code can have an attached clause that is executed if the block terminates

[1] One of the earliest soft pointer implementations was Interlisp-D's XPOINTERs [Xer85]. It did not change the values of soft pointers, but would just deallocate the object. Users of soft pointers could have all the problems associated with dangling references that garbage collection was supposed to have solved for them. Soft pointers were one of the aptly-named "unsafe" features of the language.

[2] If the garbage collector is also copying objects, the address of the object will be changing from time to time, and that too provides motivation to make the garbage collector and the population implementation interconnected.

abnormally. This is sometimes called "unwind protection," since it protects the block in question from the call stack unwinding that occurs automatically when an error throws control from the location of the error to a handler for that error. The unwind protection code is expected to take any necessary activity to clean up after the error-exit, and maintain any invariants needed in the program. Any program using semaphores, for example, benefits from unwind protection, in that an error between the points where the resource is locked and unlocked could otherwise cause the resource to remain locked. The unwind protection code can clean up after the error, and might be expected to return the resource to a consistent state and unlock it.

Often, the invariants are more closely associated with the data types than with the code, and a correct program would have nearly the same unwind protection associated with every block that declared an instance of that type. For example, if a block declares a file, it might be expected that when the block is exited, either normally or because of an error, the file's buffers will be flushed, and the file will be closed. It would be perfectly acceptable to include the code to do this in an unwinding clause of every block that declared a file, but for two things: programmers would invariably miss a few, leading to subtle bugs, and the code would be less readable for the constant clutter. Instead, the declaration of the type can be extended with what is in essence the common unwind clause, and each block containing a declaration can be assumed to have such a clause.

This type-centered formulation of final action extends to dynamically allocated objects as well. When an object is about to be freed, either explicitly or by a garbage collector, the same unwind phrase can be run. All instances of the type, allocated on the stack or heap, receive the same final treatment, and have a chance to correct any invariants before they are returned to storage. C++ destructors are the best known exemplar of this style, but C++ has no native garbage collection, and only experimental exception handling.

The issues involved for finalization of stack variables center around exceptions and error recovery [SMS81], and the issues involved for finalization of heap objects center around the topology of the connections between objects.

3 Survey of Finalization

This section is a survey of systems where finalization is available. Where I have been able to find out details about how the collection system works, I have presented as complete a description of the system as I can. Previous work [AN88] has identified a set of properties that might be desired from finalization. Some systems I know only through language reference manuals and reports, and for these systems the summary is often quite brief. I encourage anyone with knowledge of other finalization systems or more complete knowledge of any of these systems to contact me.

3.1 Lisps

Almost every Lisp dialect has some form of hash tables, and a few have populations that garbage collect inaccessible keys.

Scheme allows files to be closed automatically provided "it is possible to prove that the [file] will never again be used for a read or write operation." [AAB+91, Section 6.10.1] The garbage collector can be seen as constructing such a proof.

T [RAM84], a Lisp variant influenced by Scheme, has finalization for files but for no other data types. Files are a highly trusted client of the collector, and the collector explicitly calls a file routine to close all inaccessible files near the end of the collection. The files are known to be inaccessible by use of T's extensive weak pointer system.

I have heard rumors that other Lisp implementations have similar finalization hooks for trusted clients, but have been unable to track down any definitive sources.

3.2 Sun NeWS

The NeWS package from Sun is a windowing system using a liberally extended PostScript, and includes a conceptually parsimonious finalization interface [Sun90]. There are two operations on pointer values, *soften* and *harden*. By default, a pointer value is hard, but these operations take a pointer value of either firmness and turn it into the firmness desired. A third operator, *soft*, queries the firmness of a pointer without changing it.

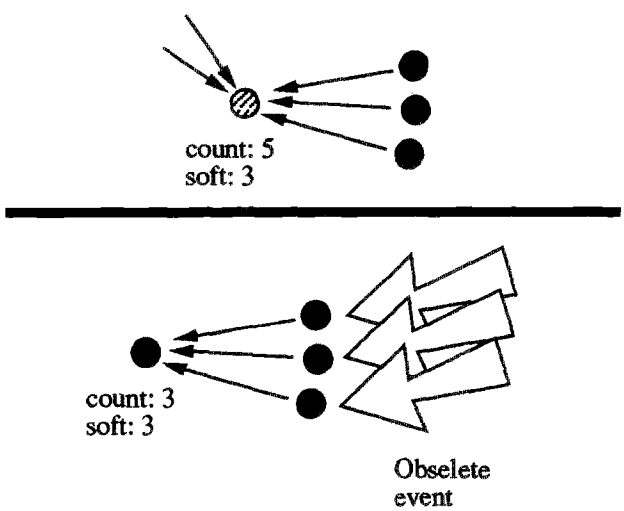


Fig. 1. NeWS Finalization

The garbage collector counts references, and maintains both a total count and a count of the soft references for each object[3]. Both reference counts are updated as needed every time a pointer is changed — they are always accurate between execution of any two PostScript operators.

Whenever the counts of total references and soft references become equal and are not both zero, the system generates an *Obsolete* event for that object. This can happen only if a hard reference is deleted or made soft. It is expected that every holder of a soft pointer will have expressed interest in the event.

3.3 Euclid

Euclid allows a module, implementing an abstract data type, to "include an *initial* action which is executed whenever a new variable of the module type is created, and a *final* action which is executed whenever such a variable is destroyed." [LHL+77, page 22] If several module variables are declared, they are initialized in order of declaration and finalized in reverse order. This is to allow later-declared objects to access fully initialized, previously declared objects at initialization, and to guarantee that at finalization no object will attempt to access a finalized object.

Euclid requires that initialization and finalization also run when an object is explicitly allocated and freed. Presumably, the finalization code would also run if the object is implicitly deallocated by garbage collection, but the definition does not make this clear.

A sticky point comes up when trying to finalize dynamically allocated objects in a sensible order. The system would like to guarantee that no object would have any methods invoked on it after it has been finalized, but two or more objects may cyclicly reference one another. If the collector finalizes one of the objects in the cycle, it may still be reachable from another that requires access to the now finalized object. There is no information available that would allow the collector to choose objects to finalize wisely under these conditions. Finalization order of cyclic structures is a problem in other languages, and will be examined in Section 4.4.

3.4 C++

The C++ language is not defined to have a garbage collector, but has *constructors* and *destructors* quite similar to the concepts found in Euclid [ES90]. The destructor is a method of an object type, and will be called by the system when the storage for an instance of that type is about to be returned to the system. It will be called as a consequence of explicitly *deleting* the object, if the object is on the heap, and it also will be called when a block declaring the object is exited, either normally when

[3] There is also a third class of references, *uncounted*, that is not available to the user, but is used by the system to break cycles among its structures.

the evaluation of the block is finished, or abnormally when the block is exited with a break, continue, return, or goto.

Within a block the order of construction and destruction is defined just as in Euclid: in declaration order for construction, and in reverse order for destruction. Types in C++ have multiple inheritance, and so the initializers and finalizers for each of the base classes, if any, have to be run at construction and destruction of objects. "[To initialize a class object] the base classes are initialized in declaration order [...], then the members are initialized in declaration order [...], then the body of [the initializer] is executed. The declaration order is used to ensure that sub-objects and members are destroyed in reverse order of initialization." [ES90, page 292]

One problem with C++ destructors stems from compiler-generated temporaries. Compiler-generated temporaries have no obvious scope, and so it is not clear when to run the destructor method. Adding multiple threads of control to C++ in the presence of destructors may also prove difficult, since pointers to objects may be passed out of the static scope where the object is created.

There have been several proposals to date for adding garbage collection to C++ [Bar89, Ede90, Det91]. One of these [Det91] explicitly disables destructors due to worries about compatibility. This is correct if the only purpose of the destructor is to explicitly delete other objects it references — the collector will do just that without any help — but will fail if the destructor has other effects.

3.5 Modula-3

Modula-3 has garbage collection without finalization, but extensions have been proposed [Hud91]. This proposal allows destructors similar to C++ and after each collection invokes the destructors for the unreachable objects in order from youngest to oldest. This is the same order they would be invoked in if the objects were stack-allocated, but the problems in using this order of finalization for heap-allocated objects is not addressed by the proposal.

Most of these problems occur when the objects form cycles of reference, and it seems reasonable that finalization should take the topological order, rather than the chronological order, of the objects into account when ordering finalization. This problem will be discussed in more detail in Section 4.4, and is common to almost all implementations.

3.6 Ada 9X

Ada has no finalization, but the Ada 9X revision does [DoD91b, Section 7.4.6]. Finalization is available for *limited types*, a restricted abstraction where assignment is not defined. Ada disallows objects of limited types in contexts where implicit

assignment or copy would be needed, and so avoids any problems that arise in finalization of temporary values [DoD91a, Section 3.2.3.1]. Finalization actions occur when the scope of the program unit finishes, for static variables, and when objects are explicitly deallocated, for allocated variables.

In addition, packages [DoD91b, Section 7.4.6] have a form of finalization. When a generic package has an *exit handler*, exiting the scope where the package is instantiated will cause the handler to run, and the package can take final actions.

The two methods are similar, but if coordination among objects of the same type is required at finalization, the use of limited types seems superior to exit handlers.

3.7 ParcPlace Smalltalk

Finalization in the Smalltalk system available from ParcPlace [Par90] is similar to the NeWS finalization, but is a more direct descendant of populations. There is a special type of array called a *weak array*, containing weak pointers; weak pointers are not available anywhere in the system except these weak arrays.

The garbage collection subsystem contains both a generational collector for young objects and an incremental collector for old objects. Both are tracing collectors — the generational collector is a copying scavenger and the incremental collector is trace and sweep.

Pointers from weak arrays are traversed last in garbage collection, and when an object is found to be inaccessible except through a weak array, the collection system frees the storage associated with the object and stores the value zero in any weak pointer that is a reference to a reclaimed object. The object is truly collected, and the zeroing guarantees that there will be no dangling references.

To give finalization information back to the user, this simple weak pointer scheme has been combined with a notification step much like the Obsolete event in NeWS. After each garbage collection, any weak array that has had a pointer zeroed is sent a "changed" message. The zeros in the array give the indices of elements that have been collected; it is the responsibility of the user's code to make sure that any data needed after the element is collected are present elsewhere, and that the "changed" message is propagated to the collected object's *executor*, which is expected to take the action needed to maintain the invariants.

Figure 2 shows an object held by a weak array, as well as other pointers. When the other pointers are deleted, the object does not go away immediately. When the garbage collector discovers that the object is reachable only via the weak array, the object is collected and the array notified.

The usual programming idiom is to make the executor a shallow copy of the object. This is a distinct object with identical values for all of its variables — it points to the same objects that the finalized object points to, and any information needed

to preserve invariants should be designed into the shared parts of the structure, not the finalized object. By the time the executor gets control, the memory allocated to the finalized object has been freed, and the object is unavailable to the executor.

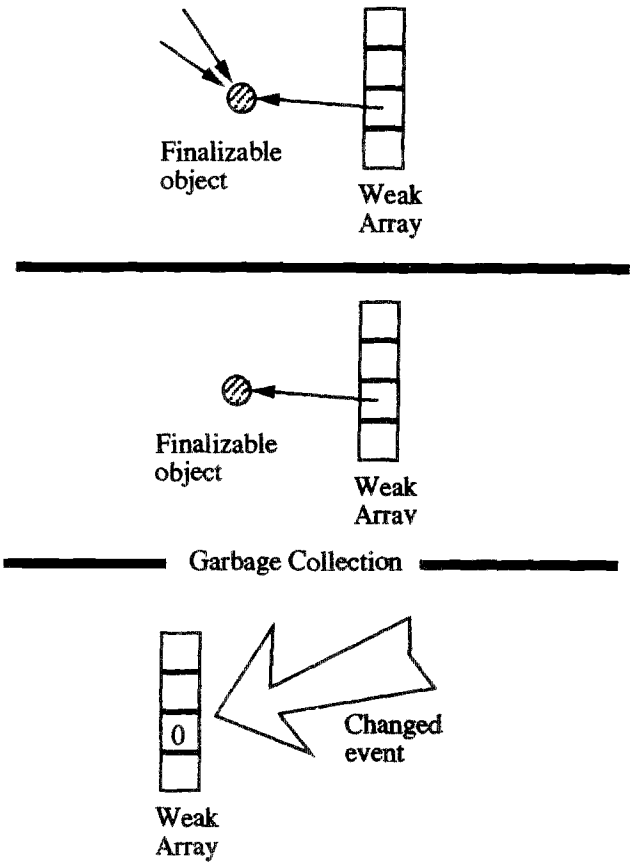


Fig. 2. Objectworks Finalization

3.8 AML/X

The object-oriented design language AML/X [NLT+86] included a reference-counting garbage collection system and finalization package. One of the driving goals in the system was to study the interfacing between object-oriented systems and procedural systems. Procedural protocols involving return of resources were enforced by using finalization methods of objects [AN88, Atk89], but natural cycles in the objects prevented some of the finalizations from occurring.

In order to get more reliable finalization of cycles, the design was carried from the reference-counting collector to a tracing collector [AN88]. Each object has three bits, "mark," "destroy," and "colour." The mark bit is the usual tracing collection mark. The destroy bit shows if an object's finalization method must be or has been called. The colour bit is used to ensure that only one object in a cycle is finalized.

All objects have the colour and mark bit initially set to zero, and collection is a five-phase process:

Mark Mark all objects reachable from the system roots, and reset the "destroy" bit on all reachable objects. Unreachable objects that were finalizaed by a previous collection may have the destroy bit still set.

Identify Candidates Examine each finalizable object in turn, setting the "destroy" bit in all unmarked, uncoloured, finalizable objects that do not have their destroy bit set, and setting the "colour" bit in them and all unmarked objects reachable from them. This phase colours objects that cannot be reached from the roots, but that are reachable from finalizable objects.

Prune If there are unreachable cycles of finalizable objects, at most one in each cycle should be selected. To do this, every finalizable object is visited in turn, and if it is coloured and its destroy bit is set, the destroy bit is cleared in all unmarked objects coloured in the previous phase. The destroy bit in the first object encountered in each cycle will not be reset unless the entire cycle is reachable from another finalizable object.

Scan Each allocated object that is neither marked not coloured is deallocated, and the mark and colour bits are cleared.

Finalize For every object with the "destroy" bit set, call its finalization method.

The designers of this system noticed a few flaws in it. First, the system's arbitrary choice of one object in a cycle can easily be wrong. It is difficult for users to predict the effects of finalization when cycles are involved. This is a problem inherent in cyclic finalizations, not a problem with this specific system.

Second, if an object's finalization method causes the object to become reachable from the roots, the "destroy" bit is still set and the finalization method will be called again on the next garbage collection. While an object from a resource pool may need to have its finalization called many times through its lifetime, it seems as if some kind of explicit "enable" call for finalization is needed to tell the difference between the return of an object to a pool, and the return of an object to a client.

The complexity of finalization in the tracing system, including the retracing of objects[4], drove one of the authors to consider other finalization techniques [Atk89]. The new proposal relies on weak pointers, much as the Objectworks system does.

[4] The P-Cedar system, outlined in Section 3.10 uses a different marking strategy and only a single mark bit to get almost exactly the same effect with none of the retracing.

Each finalizable object is paired with a *forwarding object.* All clients needing access to the finalizable resource are given pointers to the forwarding object instead, and all method calls to the forwarding object are passed to the client object — the forwarding object is invisible to the clients. The system maintains a weak pointer to each forwarding object, and so when the last client pointer is deleted the forwarding object is collected. The garbage collector then notifies a list of clients that a collection has occurred, and any finalizable object that has had its forwarding object collected can be finalized.

This system and the Objectworks system differ in the level that forwarding objects are defined — this system make them primitive, and Objectworks requires the users to roll their own or create variants. In addition, the propagation of the information that indicates that an object has been freed is more clearly defined in Objectworks. There does not seem to be an implementation of Atkins's system, and that allows many issues to remain unaddressed.

3.9 Cedar — The Early Years

D-Cedar[5], as implemented on the Xerox D-machines, uses a concurrent reference counting collector and a secondary trace and sweep collector [Rov85]. The reference counts are not always accurate for two reasons: references from the stacks are not counted, and the stacks are scanned conservatively. When an object's reference count goes to zero, it is placed in a special zero count table but not deallocated, since there may still be pointers to it from the stacks. Occasionally the garbage collector conservatively scans the stacks looking for bit patterns that, if they are pointers, point to objects with reference counts of zero[6]. In the end, any object that has a reference count of zero and is not pointed to from a stack is collected or finalized.

There is no finalization available for objects declared statically in Cedar, but typical programming practice is to explicitly create any objects needed in the block and assign them to local reference variables. Some time after the block exits, the collector will discover that the objects are no longer reachable, and they will be reclaimed. The order of initialization is under user control, and the order of finalization is determined by the topology of the interconnections among the objects.

[5] The Cedar system, including the Cedar language, has been implemented twice: the first implementation ran on Xerox's family of machines, the Dorado, the Dandelion, the Dolphin and the Daybreak. The second implementation was designed for portability, and currently runs on a number of standard platforms including Unix and Posix. The storage management has changed almost completely between the two implementations. To keep the discussion on an even keel, the first implementation will be called D-Cedar, for Dorado-Cedar, and the second P-Cedar, for portable Cedar.

[6] The implementation is more complex, in that all processes are halted just long enough for the stacks to be copied, and the conservative search for pointers occurs in these copies. The collector runs concurrently with all other active processes.

The model for finalization in D-Cedar is that a package will manage objects of a certain type, and will be responsible for maintaining any invariants associated with those objects. New objects of that type will only be created by calls to the package, and when the package returns an object it may still have several private pointers to that object. The clients need not do anything specific to manage the object, but when the clients destroy the last pointer to the object, the package, which still holds pointers to the object, should be notified that the clients can no longer use the object.

Cedar is a typed language, and finalization in D-Cedar is strongly linked to types. Associated with any finalizable type are a *finalization queue* and a positive number indicating the count of *package references*. Any particular object of a finalizable type can be explicitly enabled for finalization by a call to the storage manager. This call sets a bit in the object's header, and decrements the object's reference count by the package reference count. From that point, the object is reference counted normally.

When an object has a zero reference count and there is no pointer to it from the stack, it is freed if the finalization bit in that object is not set. If the bit is set, the collector clears the bit, sets the reference count for the object to be the package count, determined from the object's type, and adds the object to the finalization queue for that type, allowing the package to do whatever is required with the object to maintain its invariants[7].

In practice, the use of a type-wide count of package references proves to be fragile. The package must ensure that it has the same number of pointers to every object enabled for finalization, and the writing of packages where finalization is used is a delicate affair. Catastrophic failures occurs when a dropped package pointer makes a reference count negative.

Notice also that the object is changed from finalizable to not finalizable when the finalization is run. The object may be explicitly set finalizable again, setting the bit and reducing the reference count, but it is not automatic. This helps prevent errors where the finalization code runs for each garbage collection without making progress on finalizing the object. Instead, the code will be run once, and if the object is neither made reachable nor re-enabled for finalization, it will be collected.

---

[7] When the package count is zero, this is what is sometimes called *resurrection semantics*, since the object has no pointer to it, and yet the collector creates one to enqueue the object. If you feel uncomfortable with the idea that the collector is creating a pointer to an object after the user has discarded all the pointers to it, recall that the call to the collector to enable finalization allows the system to squirrel away a pointer to the object, and that it is this pointer that is used to enqueue the object. In fact, that pointer exists — it is simply compressed into a single bit in the header of the object.

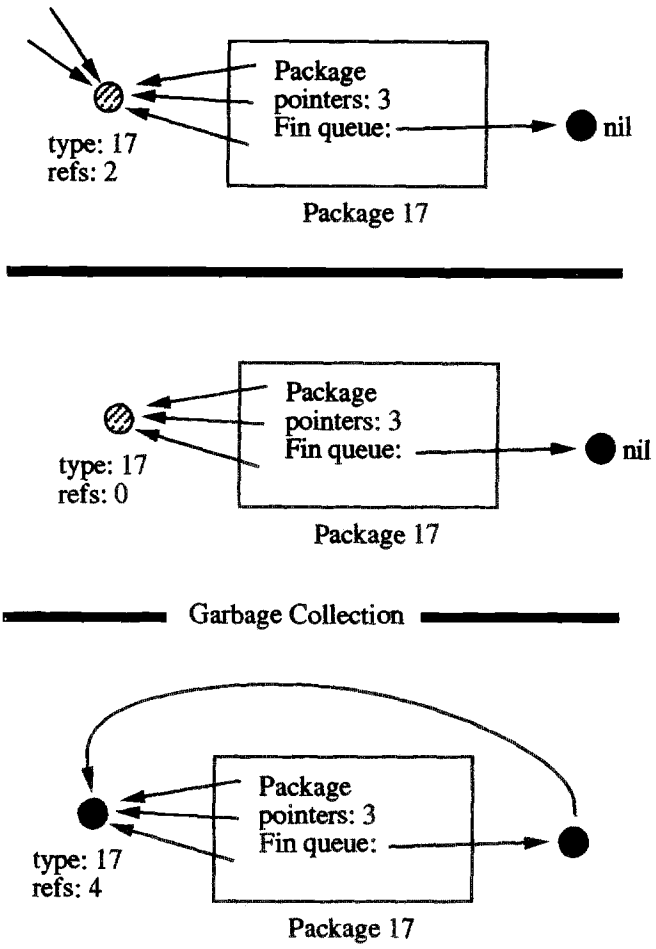


Fig. 3. D-Cedar Finalization

3.10 Cedar II — The Revenge

Some of the weaknesses in the storage management in D-Cedar were addressed in P-Cedar [ADH+89]. The reference counting collector was replaced with a conservative, generational, mark-and-sweep collector. This freed the programmer from the burden of breaking reference cycles in complex data structures[8].

[8] One of the common uses of finalization in D-Cedar was to break these cycles. For example, a tree where every leaf points back to the root will never be collected by simple reference counting — when the last reference to the root other than the leaves is deleted, the root will still have a non-zero count. A package count for the root would not make sense, since the count at the root is the number of leaves, and that is variable, rather than structural. But if a node is added above the root and all access takes place through that node, that extra node can be enabled for finalization. When all references to it are gone, the leaves

Most of the work of finalization has been put into a package distinct from the garbage collector. The finalization package is still tightly coupled to garbage collection, but the split seems valuable to insulate the two functions — collection and finalization — from each other.

To enable an object for finalization, a client calls a routine in the finalization package with a pointer to an object and a finalization queue. The package returns a pointer called a finalization handle. Strictly speaking, it is the handle itself, not the object, that is enabled for finalization. The only important operations available on a finalization handle are disable finalization, re-enable finalization, and dereference. The disable/re-enable calls do the obvious, and the dereference call returns a pointer to the object that was a parameter to the enable call that returned that handle. A disabled finalization handle functions simply as an indirect pointer to the object. The finalizer keeps the state needed to finalize objects; this is not the responsibility of its clients, and they often ignore the finalization handle returned, knowing that the finalization will occur nonetheless.

The collector traces from the roots, but does not trace through the finalization package's state to objects that are enabled for finalization[9]. At this point, any finalizable object that has been seen by the trace is accessible from the roots, and should not be finalized.

Only some of the objects unreachable from the roots are put on their finalization queues. The intent is to mimic the effects of the reference counting finalization of D-Cedar by only finalizing those objects that are not reachable from other finalizable objects. If P points to Q, and both are finalizable and unreachable from the roots, the system would like to finalize P. When P is put on its finalization queue, Q is now reachable from the roots via the queue, and should not be finalized.

The objects to be queued are found by another marking phase of the collector. It traces all of the pointers from unmarked finalizable objects, but does this without initially marking the finalizable objects themselves. After this marking is finished, any marked finalizable object was either marked by the first phase, and so is reachable from the roots, or was marked in the second phase, and so was reachable from a finalizable object[10]. Any unmarked finalizable object is reachable through neither

---

still point to the root, but there are no pointers to the uber-root. The finalization for the uber-root can walk the tree down to the roots and NIL the backpointers, allowing the reference counter to discover that the tree's storage can be reclaimed.

[9] The finalization package does not, in fact, keep pointers to the objects. The object pointed to by a handle contains a field that is a disguised copy of the pointer to the object, and the finalization package keeps a list of the currently enabled finalization handles. The collector does not recognize the disguised pointers as pointers when doing the trace.

[10] At the moment, a finalizable object that is reachable from itself but no other finalizable object is *not* finalized. This is considered to be a bug and will be changed. This might present a danger to an object that has access to an object of the same type as itself, since it might have to check the identity of the object against itself. For example, we

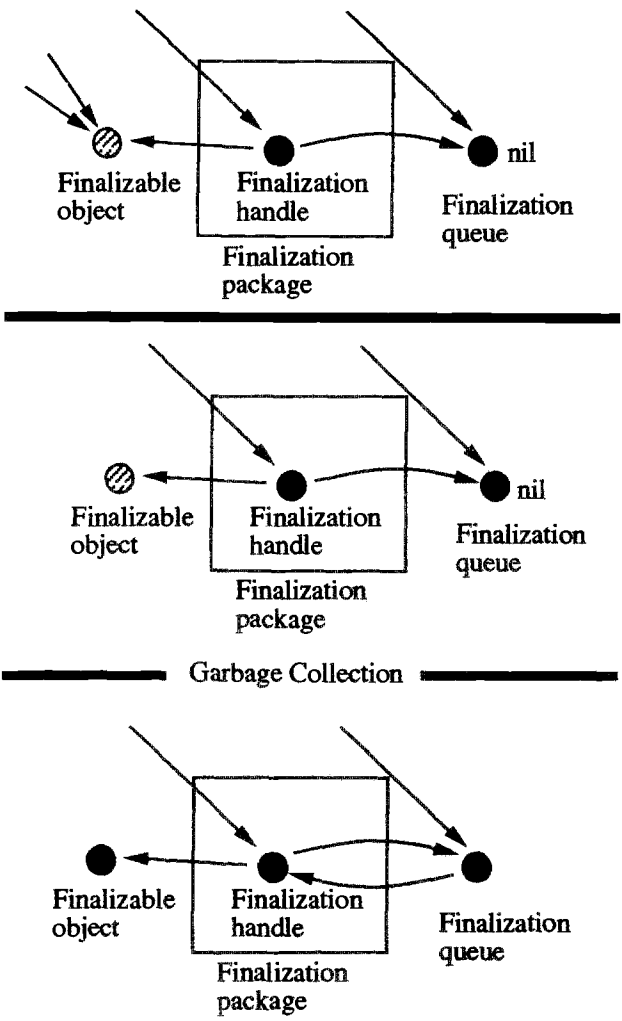


Fig. 4. P-Cedar Finalization

method. The finalization package can safely enqueue each unmarked finalizable object, and change its state to indicate that it has discharged its duty — the enqueued objects are no longer enabled.

might have a system that writes a record to a log file for each file finalized. Imagine the problems that might occur when the log file is closed.

4 Finalization Issues

As should be apparent, there is no consensus of opinion on how to design the interface between the collector and its clients to allow for finalization. There are at least four major decisions that a designer should consider in specifying a finalization interface: decoupling, promptness, locking, and cycles.

4.1 Decoupling

Aside from the Lisps that have a restricted group of clients for finalization, systems must be careful in how the results of tracing are communicated to the clients. If a collector were to directly call some kind of finalization routine for a client, it would risk aborting or looping the collection thread. Event and message systems have a natural way of dealing with this problem, since the messages provide a decoupling between the collector and its clients. Likewise, the Cedar queues let the collector enqueue an object without worrying about the consequences of calling general client code. Another option would be to fork a process from the collector for each client needing to be finalized.

4.2 Promptness

When a resource is recovered soon after it is no longer reachable from its clients, we say that it is *promptly reclaimed.* This measure is often applied to garbage collection, and it is also a useful measure of finalization. The range of promptness in the systems in Section 3 is broad.

The NeWS system recovers memory as soon as an object's reference count drops to zero, and sends Obsolete events as soon as the last hard pointer is deleted. Promptness is a benefit of being a reference counting collector. But objects that are unreachable from the roots and involved in cycles will never be recovered or finalized, and so these objects can hardly be said to be promptly reclaimed. Reference counting is both a boon and a bane for promptness.

Inexact collection, such as is found in generational collectors, has a similar effect. An object may be considered reachable because there is a pointer to it from an old object that is unreachable, but hasn't been collected yet. No object can guarantee that it will be promptly collected or finalized.

Conservative collection confounds the issue even more [BW88]. Not only can an object remain uncollected or unfinalized because of genuine pointers to it, but bit patterns that the collector treats as pointers are enough to keep objects uncollected. An application may earnestly exercise great care to NIL all of the pointers to an object to make finalization or collection occur, only to have an errant integer prohibit it.

Current collectors do not offer clients any options to help regulate the promptness of the service they get, and as object bases get larger and larger, generational collectors will make this problem worse and worse. It seems as if a good rule of thumb for finalization clients is to consider finalization to be a frill, and not to rely on it for promptness or correctness.

4.3 Locking

All too often, the garbage collector and finalization routines are overlooked as a source of parallelism in code, and parts of an aggregate structure will be finalized even when other clients might still be using it. For example, consider a tree with back-pointers from the leaves to the root, and an extra finalizable node at the top that will break all the cycles to aid a reference counting collector. Great care must be taken to ensure that this finalization does not occur while there is still a pointer to an internal node. The process holding that pointer might be surprised to find that the back-pointer is NIL.

Unfortunately, the simple ways of locking out the finalization are not guaranteed to work. For example, it might be thought that any process holding a pointer to the finalizable uber-root node while examining the other nodes would be safe, but compilers would look on the reference to the uber-root as dead, since it is not used, and might optimize away the load of the pointer. This is just another manifestation of collector/optimizer interactions, but lifted into the domain of finalization [Cha87, Boe91].

Neither of the obvious solutions to this problem are attractive. The optimizer can be prevented from performing some useful optimizations, but that's a performance cost many might blanch to pay. Modules could supply client calls to ensure that structures are not finalized when there is an active client, but this seems a violation of modularity, since it reveals to clients of the package that objects are finalized.

4.4 Cycles

When two or more finalizable resources are clients of one another in a cyclic order, finalization becomes much more complex. The system might decide to take a conservative view and finalize no object in the system. This guaranteed lack of promptness leads to resource leaks similar to memory leaks from cycles in a reference counting collector.

The system may try to guess an object in the cycle to finalize first, but if it chooses incorrectly, another object may try to make client calls to the first finalized object. The client of the finalized object may be unable to do anything reasonable now that the resource embodied in the first object has been rescinded [AP87].

Any system that chooses to finalize all the objects in a cycle in an unpredictable order dooms the programmer to adjudicating the correct order by use of mutual exclusion primitives. But this violates modularity, since each object must be aware of the cycles it might be involved in.

To break the symmetry of the cycle, some objects might use soft pointers to point to other objects to indicate that they do not require the softly-held resource at finalization, and would be willing to be finalized after the other resource. But this means that the softly-held resource might be finalized while its client is still firmly-held and active if the holder of the soft pointer is its sole client. It must be ready to have the resource finalized at any point, not only just prior to its own finalization.

Finalization of cyclic structures is a problem that the garbage collector cannot solve without further development on the interface between the languages and the finalization package. The current interfaces seem to be too narrow to address all of the situations that arise.

5 Conclusions

In many new systems and languages, garbage collection is considered indispensable. Programs can be crafted without worrying about memory leaks or dangling pointer bugs — the memory will be recycled when it is no longer needed and not before because the collector can guarantee when a memory object is no longer reachable from the roots.

If other system elements are allowed access to the information gathered by the garbage collector, they can make decisions about non-memory resource recycling, and allow these resources to be managed in much the same way as garbage collected memory.

Several recent researchers have tried to marry C++ destructors and garbage collection to get finalization, but previous efforts in finalization and the problems that have been encountered by users of finalization does not seem to be well known.

At least four systems, Objectworks, NeWS, D-Cedar, and P-Cedar, have been built with a garbage collector that allows user code some access to the information gathered by the collector. All four of these systems are in current use, but the finalization features seem to be little-known in the systems community, and even the memory management community.

6 Thanks

Many people helped with the gathering of information for this paper. Much of the leg-work for citations was done by Frank Jackson, who also checked that my description of ParcPlace Smalltalk was reasonably accurate. In that same vein, thanks to Stuart

Marks from Sun for the NeWS information, and Carl Hauser of Xerox PARC for the Cedar information.

This work was funded by Dr. John Koza, the Northern California Chapter of ARCS, Inc., and Xerox.

A A Garden of Delights

Many people have been active in the discussion of changes to finalization at Xerox PARC. The most active of these are Hans Boehm, Alan Demers, Carl Hauser, Christian Jacobi[11], and myself. There is a set of canonical examples we have been using in discussion of problems with and extensions to finalization. Much of the text of this section was provided by Carl Hauser; otherwise, it is hard to credit any particular example to any particular person.

A.1 File Descriptors

This example involves collection of non-memory, low level resources, and shows a simple case where finalization is valuable.

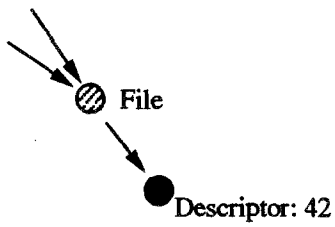


Fig. 5. Simple Finalization

Unix file descriptors are a resource that should be garbage-collected: to first approximation, if there are no copies of a file-descriptor in a program, then that file descriptor should be closed and freed. By allocating a memory object for each opened file descriptor, and uniformly passing the memory object around as the representative of the open file descriptor (instead of the file descriptor), we know to close the file descriptor when the memory object becomes unreachable.

A.2 Buffered File

In this example, two objects, each of which should logically be enabled for finalization, point to each other and form a cycle.

[11] Christian has shown the patience of a saint in letting us discuss the issues to death before we actually do anything that will let him fix the problem with his code.

Suppose we have an OpenFile abstraction, a BufferedFile abstraction and a Buffer abstraction. OpenFile objects contain file descriptors and hence need finalization enabled. They also refer to a Buffer object — the next buffer to be filled, for example. BufferedFile is one of perhaps many abstractions built on OpenFile, and each BufferedFile supplies the buffers used by its underlying OpenFile. BufferedFile's objects need finalization enabled to allow write buffers to be flushed prior to closing the underlying OpenFile object. The Buffer abstraction provides a field in each object for recording the owner of that buffer object so that low-level system interrupts that refer to memory locations in the buffer can be correctly forwarded to the proper OpenFile.

Since the Buffers are owned by BufferedFiles, this gives us a cycle containing two finalizable objects: a BufferedFile points to an OpenFile, which points to a Buffer, which points back to the BufferedFile.

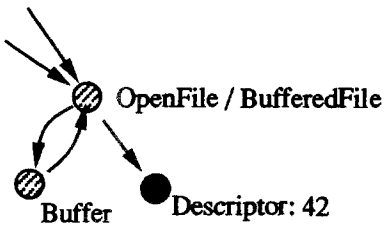


Fig. 6. Cyclic Finalization

A.3 Courier Handles

This example deals with "resurrection," when the finalization code for an object makes the object accessible rather than letting it become garbage. In this example, objects that are difficult to recreate are kept in a cache. Rather than being deallocated, they are recycled.

In D-Cedar, handles for open Courier connections were serially reusable: a client could use a connection for awhile, then give it back to the Courier package which might eventually hand it to some other client. This was useful because opening a connection is a heavyweight operation, so clients talking in rapid succession to the same machine got a performance boost by reusing an already-established connection. To return connections to the pool when clients dropped them, finalization was enabled for each CourierHandle. The Courier package gave out a handle to a single client. When the client finished, it either gave the handle back or dropped it. In the latter case, finalizing the object provided the missing giveback call.

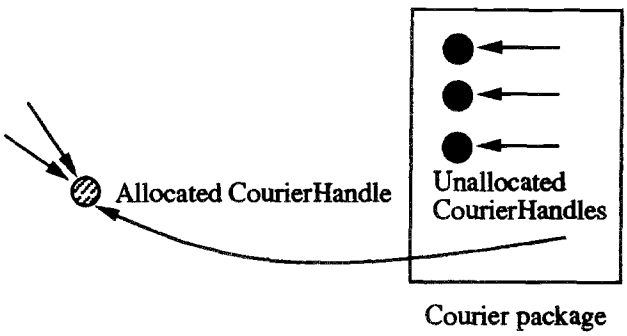


Fig. 7. Caching Finalizable Objects

A.4 Log Files

Examples of this kind were not mentioned in the main body of the article, but constitute another interesting problem with finalization. In this example, two different finalization actions need to be attached to a single object, since it needs to be finalized at two levels of abstraction.

Given an OpenFile abstraction, we might want to build a LogFile — a file to which clients could log records text records. When a LogFile is no longer accessible, we would like to write one last record to it saying that the log file has now been closed. The finalization for the LogFile, which writes the record, must be run before the finalization for the OpenFile, which closes the file.

A.5 Population of Finalizable Objects

This is another example where multiple finalizers exist for the same object, one of which may resurrect the object. The order of finalization is important, but not apparent to the system.

We may want to build a cache of OpenFiles, to ensure that two clients requesting the same file share the same OpenFile. If the cache's finalizer is run first, it cannot know when the file's finalizer has finished, and risks having two OpenFiles for the same file — one newly opened and one not quite yet finalized. If on the other hand the OpenFile's finalizer is run first, it cannot know if any pointers to the file remain, and a call might occur after finalization.

References

[AAB+91] H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, Dybvig R. K., D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steel Jr., G. J. Sussman, and M. Wand.

Revised[4] report on the algorithmic language Scheme. *ACM LISP Pointers*, IV(3), November 1991.

[ADH+89] R. Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. *SIGPLAN Notices*, 24(7):261–269, July 1989.

[AN88] Martin C. Atkins and Lee R. Nackman. The active deallocation of objecs in object-oriented systems. *Software Practice and Experience*, 18(11):1073–1089, November 1988.

[AP87] S. G. Abraham and J. H. Patel. Parallel garbage collection on a virtual memory system. In *14th Annual international symposium on computer architecture*, page 26, June 1987.

[Atk89] Martin C. Atkins. *Implementation Techniques for Object-Oriented Systems*. PhD thesis, University of York, Dept. Computer Science, University of York, Heslington, York, YO1 5DD, England., 1989.

[Bar89] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical report, Digital Western Reseaerch Laboratory, October 1989.

[Boe91] Hans-J. Boehm. Simple gc-safe compilation. In *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1992*, October 1991. Available by anonymous ftp from cs.utexas.edu in pub/garbage/GC91.

[BW88] Hans-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cha87] David R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, November 1987.

[DoD91a] Department of Defense. *Mapping Rational*, volume I of *Ada 9X Mapping*. Intermetrics, Inc., Cambridge, Massachusetts, August 1991.

[DoD91b] Department of Defense. *Mapping Specification*, volume II of *Ada 9X Mapping*. Intermetrics, Inc., Cambridge, Massachusetts, August 1991.

[Det91] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, 1991.

[Ede90] D. Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, Mass, 1990.

[Hud91] Richard L. Hudson. Finalization in a garbage collected world. In *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1992*, October 1991. Available by anonymous ftp from cs.utexas.edu in pub/garbage/GC91.

[LHL+77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Notices*, February 1977.

[NLT+86] Lee R. Nackman, Mark A Lavin, Russell H. Taylor, Walter C. Dietrich, Jr., and David D. Grossman. AML/X: a programming language for design and manufacturing. In *Proceedings of the Fall Joint Computer Conference*, pages 145–159, November 1986.

[Nyb89] Karl A. Nyberg, editor. *The Annotated Ada Reference Manual*. Grebyn Corporation, Vienna, Virginia, 1989. [An annotated version of ANSI/MIL-STD-1815A-1983, The Ada Reference Manual].

[Par90] ParcPlace Systems. *ObjectWorks / Smalltalk User's Guide, Release 4*. ParcPlace Systems, Inc, Mountain View, CA, 1990.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meechan. The T manual. Technical report, Yale University, January 1984.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, staticly-checked, concurrent language. Technical Report CSL-84-7, Xerox

Corporation, July 1985.

[SMS81] Richard L. Schwartz and P. M. Melliar-Smith. The finalization operation for abstract types. In *Proceedings of the 5th International Conference on Software Engineering*, pages 273–282, San Diego, California, March 1981.

[Sun90] Sun Microsystems. *NeWS 2.1 Programmer's Guide.* Sun Microsystems, Inc, Mountain View, CA, 1990.

[Xer85] Xerox Corporation. *Interlist Reference Manual*, volume 1. Xerox Corporation, Palo Alto, CA, October 1985.

This article was processed using the LaTeX macro package with LLNCS style

Precompiling C++ for Garbage Collection

Daniel R. Edelson

INRIA Project SOR, Rocquencourt BP 105, 78153 Le Chesnay CEDEX, FRANCE
Daniel.Edelson@inria.fr

Abstract. Our research is concerned with compiler-independent, efficient and convenient garbage collection for C++. Most collectors proposed for C++ have either been implemented in a library or in a compiler. As an intermediate step between those two, this paper proposes using precompilation techniques to augment a C++ source program with code to allow mostly type-accurate garbage collection. There are two key precompiler transformations. The first is automatic generation of *smart pointer* classes. The precompiler defines the smart pointer classes and the user utilizes them instead of raw pointers. These smart pointers supply functionality that allows the collector to locate the root set for collection. The second transformation augments the C++ program with code that allows the garbage collector to locate internal pointers within objects. This paper describes the precompiler and the garbage collector. The paper includes a brief (1500 word) survey of related techniques.

1 Introduction

The lack of garbage collection (GC) in C++ decreases productivity and increases memory management errors. This situation persists principally because the common ways of implementing GC are deemed inappropriate for C++. In particular, tagged pointers are unacceptable because of the impact they have on the efficiency of integer arithmetic, and because the cost is not localized.

In spite of the difficulty, an enormous amount of work has been and continues to be done in attempting to provide garbage collection in C++. The proposals span the entire spectrum of techniques including:

- compiler-based concurrent atomic mostly-copying garbage collection [12],
- library-based reference counting and mark-and-sweep GC [27],
- library-based mostly copying generational garbage collection [5],
- library-based reference counting through *smart pointers* [28, 29] (Smart pointers are discussed momentarily),
- library-based mark-and-sweep GC using smart pointers [16],
- compiler-based GC using smart pointers [22],
- library-based mark-and-sweep and generational copying collection using macros [21], and,
- library-based conservative generational mark-and-sweep GC [8, 11].

The vast number of proposals, without the widespread acceptance of any one, reflects how hard the problem is.

In the past, we have proposed implementing GC strictly in application-code: GC implemented in a library. The problem with this approach is that it requires too much effort on the part of the end-user. The user must first customize/instantiate the library, and then follow its rules. This is a tedious and error-prone process.

To solve our goal of compiler-independence, while keeping the associated complexity to the user to a minimum, we are now proposing *precompiling* C++ programs to augment them for garbage collection. The user still needs to cooperate with the collector, but the likelihood of errors is reduced. In addition, the precompiler can perform transformations that are independent of the actual garbage collection algorithm in use, making it very useful for experimentation in GC techniques.

A Word About Smart Pointers

A number of the systems that are considered in the related work section use smart pointers, as does this collector. Therefore, this paper begins with an introduction to the term.

C++ provides the ability to use class objects like pointers; these objects are often called smart pointers [38]. Smart pointers allow the programmer to benefit from additional pointer semantics, while keeping the syntax of the program largely unchanged. Smart pointers use *operator overloading* to be usable in expressions with the same syntax as normal pointers. For example, the overloaded assignment operator = permits raw pointers to be assigned to smart pointers and the overloading indirect member selection operator -> permits smart pointers to be used to access data members and operations of the referenced object.

Smart pointers can be used for a variety for purposes. For example:

- reference counting [10, 27, 28, 29, 39],
- convenient access to both transient and persistent objects [36, 39],
- uniform access to local or distributed objects [24, 35, 37],
- synchronizing operations on objects [39, p. 464],
- tracing garbage collection [16, 18, 27],
- instrumenting the code,
- or others.

Section 2 discusses how existing systems use smart pointers for memory management. A discussion of various issues concerning smart pointers can be found in [17]. Our implementation of smart pointers is presented in §3.1.

2 A Brief Survey of Related Work

There is a significant body of related work, in the general field of GC, in C++ software tools, and specifically in collectors for C++.

2.1 Conservative GC

Conservative garbage collection is a technique in which the collector does not have access to type information so it assumes that anything that might be a pointer actually is a pointer [7, 8]. For example, upon examining a quantity that the program

interprets as an integer, but whose value is such that it also could be a pointer, the collector assumes the value to be a pointer. This is a useful technique for accomplishing compiler-independent garbage collection in programming languages that do not use tagged pointers.

Boehm, Demers, et al. describe conservative, generational, parallel mark-and-sweep garbage collection [7, 8, 11] for languages such as C. Russo has adapted these techniques for use in an object-oriented operating system written in C++ [32, 34]. Since they are fully conservative, during a collection these collectors must examine every word of the stack, of global data, and of every marked object. Boehm discusses compiler changes to preclude optimizations that would cause a conservative garbage collector to reclaim data that is actually accessible [6]. Zorn has measured the cost of conservative garbage collection and found that it compares favorably not just with manual allocation, but even with optimized manual allocation [44].

Conservative collectors sometimes retain more garbage than type-accurate collectors because conservative collectors interpret non-pointer data as pointers. Often, the amount of retained garbage is small, and conservative collection succeeds quite well. Other times, conservative techniques are not satisfactory. For example, Wentworth has found that conservative garbage collection performs poorly in densely populated address spaces [41, 42]. Russo has found that the programming style must take into account the conservative garbage collector: naive programming leads to inconveniently large amounts of garbage escaping collection [33, 34]. For example, he has found it necessary to disguise pointers and manually break garbage cycles [33]. To aid the programming task, he is investigating augmenting the conservative garbage collector with *weak pointers* [30], i.e. references that do not cause objects to be retained. Finally, we have tested conservative garbage collection with a CAD software tool called ITEM [16, 26]. This application creates large data structures that are strongly connected when they become garbage. A single false pointer into the data structure keeps the entire mass of data from being reclaimed. Thus, our brief efforts with conservative collection in this application proved unsuccessful.

As these examples illustrate, conservative collection is a very useful technique, but it is not a panacea. Since it has its bad cases, it is worthwhile to investigate type-accurate techniques for C++.

2.2 Partially Conservative

Bartlett's *Mostly Copying Collector* is a generational garbage collector for Scheme [14] and C++ [39] that uses both conservative and copying techniques [4, 5]. This collector divides the heap into logical pages, each of which has a *space-identifier*. During a collection an object can be promoted in one of two ways: it can be physically copied to a to-space page or the space-identifier of its present page can be advanced.

Bartlett's collector conservatively scans the stack and global data seeking pointers. Any word the collector interprets as a pointer may in fact be either a pointer or some other quantity. Objects referenced by such roots must not be moved because, as the roots are not definitely known to be pointers, the roots cannot be modified. Such objects are promoted by having the space identifiers of their pages advanced. Then, the root-referenced objects are scanned with the help of information provided by the application programmer; the objects they reference are compactly copied to

the new space. This collector works with non-polymorphic C++ data structures, and requires that the programmer make a few declarations to enable the collector to locate the internal pointers within collected objects.

Detlefs implements Bartlett's algorithm in a compiler and uses type information available to the compiler to generalize the collector. Bartlett's first version contains two restrictions, the first of which is later eliminated:

1. internal pointers must be located at the beginnings of objects, and
2. heap-allocated objects may not contain *unsure* pointers.

An unsure pointer is a quantity that is statically typed to be either a pointer or a non-pointer. For example, in "union { int i; node * p; } x;" x is an unsure pointer.

Detlefs relaxes these by maintaining type-specific map information in a header in front of every object. During a collection the collector interprets the map information to locate internal pointers. The header can represent information about both sure pointers and unsure pointers. The collector treats sure pointers accurately and unsure pointers conservatively. Detlefs' collector is concurrent and is implemented in the *cfront* C++ compiler.

2.3 Type-Accurate Techniques

Kennedy describes a C++ type hierarchy called OATH that uses both reference counting and mark-and-sweep garbage collection [27]. In OATH, objects are accessed exclusively through application-level references called *accessors*, that are very similar to stubs because they duplicate the interfaces of their target objects. Accessors implement reference counting on the objects that they reference. The reference counts are used to implement a three-phase mark-and-sweep garbage collection algorithm [9] that proceeds as follows. First, OATH scans the objects to eliminate from the reference counts all references between objects. After that, all objects with non-zero reference counts are root-referenced. The root-referenced objects serve as the roots for a standard mark-and-sweep collection, during which the reference counts are restored. Like normal reference counting, this algorithm incrementally reclaims some memory. In addition, however, this algorithm reclaims garbage cycles.

In OATH, a method is invoked on an object by invoking an identically-named method on an accessor to the object. The accessor's method forwards the call through a private pointer to the object. This requires that an accessor implement all the same methods as the object that it references. Kennedy implements this using preprocessor macros so that the methods only need to be defined once. The macros cause both the OATH objects and their accessors to be defined with the given list of methods. While not overly verbose, the programming style that this utilizes is quite different from the standard C++ style and such long macros can make debugging difficult.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [23], building on work by Appel [2], who in turn builds on techniques that were invented for Algol-68 and Pascal. Goldberg's compiler emits functions that know how to locate the pointers in all necessary activation records of the program. For example, if some function $\mathcal{F}$ contains two pointers as local variables, then another function would be emitted to mark from those pointers during a collection. The emitted function would be called once for every active

invocation of $\mathcal{F}$ to trace or copy the part of the datastructure that is reachable from each pointer. The collector follows the chain of return addresses up the run-time stack. As each stack frame is visited, the correct garbage collection function is invoked. A function may have more than one garbage collection routine because different variables are live at different points in the function. Clearly, this collector is very tightly coupled to the compiler.

Yasugi and Yonezawa discuss user-level garbage collection for the concurrent object-oriented programming language ABCL/1 [43]. Their programming language is based on active objects, thus, the garbage collection requirements for this language are basically the same as for garbage collection of Actors [13, 25].

Ferreira discusses a C++ library that provides garbage collection for C++ programs [21]. The library supplies both incremental mark-and-sweep and generational copy collection, and supports pointers to the interiors of objects. The programmer renders the program suitable for garbage collection by placing macro definitions at various places in the program. For example, every constructor must invoke a macro to register the object and every destructor must invoke a complementary macro to un-register the object. Another macro must be invoked in the class definition to add GC members to the class, based on the number of base classes it has. To implement the remembered set for generations, the collector requires a macro invocation on every assignment to an internal pointer. Ferreira's collector requires that the programmer supply functions to locate internal pointers. It can also scan objects conservatively to work without these functions.

Maeder describes a C++ library for symbolic computation systems whose implementation uses smart pointers and reference counting [29]. The library contains class hierarchies for *expressions*, *strings*, *symbols*, and other objects that are called *normal*, and reference-counting smart pointers are used exclusively to access the objects. To improve the efficiency of assignment of reference counted pointer assignment, the address of a discrete object serves as a replacement for the NULL pointer. This means that pointers do not need to be compared with NULL before being dereferenced to modify the reference count. The smart pointers support debugging by allowing the programmer to detect dangling references: rather than being deleted, an object is marked *deleted* and subsequent accesses to the object cause an error to be reported. Other functionality allows the programmer to detect memory leaks by reporting objects that are still alive when the program terminates.

Madany et al. discuss the use of reference counting in the *Choices* object-oriented operating system [28]. The hierarchy of operating system classes is shadowed by parallel smart pointer classes, called *ObjectStars*. By programmer convention, the system classes are accessed exclusively through ObjectStars, which implement reference counting on their referents. As identified by Kennedy in [27], returning reference counting smart pointers from functions can sometimes result in dangling references. This was observed to be true of the ObjectStars, and therefore the following convention was adopted: Whenever an ObjectStar is returned from a function, it must first be assigned to a variable; it cannot be immediately dereferenced [15]. This prevents that particular error.

3 Garbage Collecting C++ Code

The program's dynamically allocated garbage collected objects are collectively referred to as the *data structure*. The collector's job is to determine which objects in the data structure are no longer in use and to reclaim their memory. The application has pointers into the data structure; these pointers are called *roots* and are collectively referred to as the *root set*. Any object in the data structure that can be reached by following a chain of references from any root is *alive*. The other objects are *garbage* and should be reclaimed. The two hard problems are: 1) finding the roots, and 2), locating pointers inside objects, called *internal pointers*.

3.1 Roots and Smart Pointers

This system uses smart pointers that implement indirection through a root table. All of the direct pointers are concentrated in the root table and can therefore be located by the collector. The term *root* is used to refer to the smart pointer objects. In contrast, the built-in pointers, i.e. the pointers that are directly supported by the compiler and the hardware, are called *raw pointers*.

A problem with smart pointers is that they can be nontrivial to code [17]. The problem arises from emulating the implicit type conversions of raw pointers. For example, a raw pointer of type `T*` can be implicitly converted to type `const T*`, based on the safety of converting an unrestricted pointer to a pointer that permits only read accesses. Also, derived class pointers can be converted to base class pointers, reflecting the *isa* relationship between a derived class and its base classes. C++ allows smart pointers to emulate these type conversions using *user-defined* type conversions. The need to add these user-defined type conversions makes generation of the smart pointer classes inconvenient. They cannot be automatically produced from a parameterized type, a *template*, because that does not supply the necessary type conversions. While macros or inheritance can abbreviate the process, some coding is required. Emitting smart pointer class definitions, rather than necessitating hand coding, is one of the tasks of the precompiler.

The Root Table. The data structure that allows the collector to find the root set is the *root table*. It it implemented as a linked list of *cell arrays*. Each cell array contains its list link and many direct pointer *cells*. A cell may be *active*, in which case it contains a direct pointer value, or it may be *free*, in which case it is in the free list. A diagram of this data structure is presented in Fig. 1.

The application's smart pointers point to pointer cells rather than directly to objects; the cells, in turn, contain the direct pointers. C++ objects implement this in the following way. The initialization code for a root, i.e. the *constructor*, gets a cell from the free list, optionally initializes the cell, and makes the root point to the cell. The de-initialization code for a root, the *destructor*, adds the root's cell to the free list. The overloaded indirection operators first dereference the indirect pointer to fetch the direct pointer and then dereference the direct pointer. The overloaded assignment operator causes assignment to a root to assign to the direct pointer rather than to the indirect pointer.

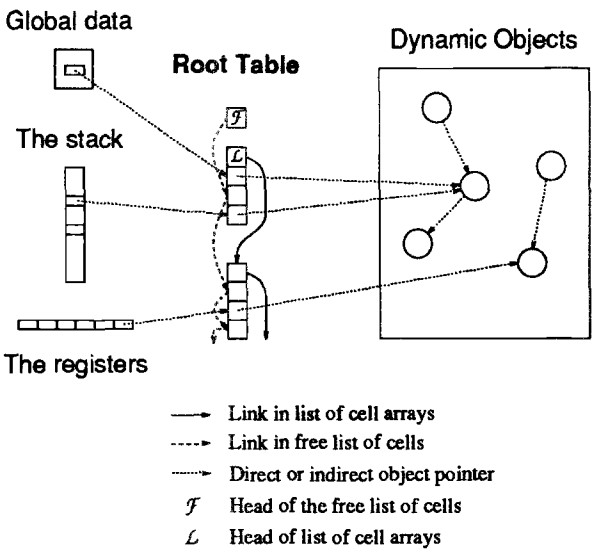


Fig. 1. The root table

Linked list removal usually requires a test and conditional branch to check for the end of the list. In this implementation, however, when a cell is removed from the free list, its value is immediately fetched. That fetch is used to avoid the test and branch. The last page of the last cell array is *read-protected* [3]. Attempting to load the link stored in the first cell on the read-protected page causes the program to receive a signal. The signal handler unprotects the page, links in and initializes a new cell array, and read-protects the last page of the new array. A new diagram of a cell array is presented in Fig. 2; the shaded area illustrates the read-protected region.

Smart Pointer Class Definitions. For every application class two smart pointer classes are generated. One of them emulates pointers to mutable objects and the other emulates pointers to const objects. When the application classes are related through inheritance, the precompiler gives the derived class smart pointers user-defined type conversions to the base class smart pointer types. A detailed description of this organization can be found in [17].

The precompiler parses the program to determine what smart pointer classes are needed and writes the classes to a file. Then, the preprocessed and otherwise transformed application code is appended.

A typical smart pointer class is shown in Fig. 3. This shows the smart pointer class for const objects. The associated smart pointer class for mutable objects derives from this class.

Smart Pointer Efficiency. Each smart pointer takes up two words in memory, one for the indirect pointer and one for the direct pointer. The actual space overhead is greater than that because the root table grows in increments of 8 kilobytes.

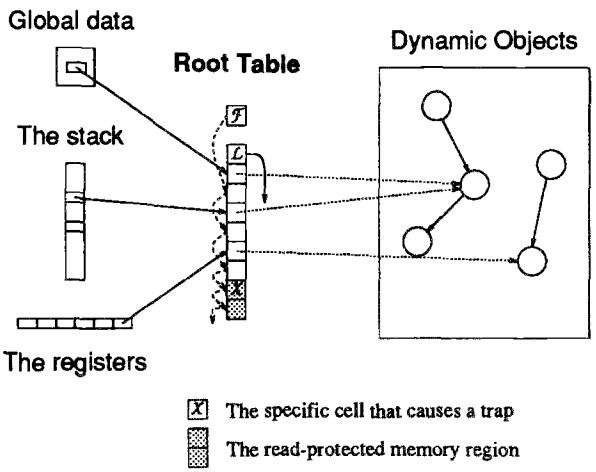


Fig. 2. The protected page of a cell array

The last cell array has its last page read protected. When the protection violation occurs, a new array is allocated and linked to the others.

```
class Root_C_T {
  protected:
    const T * * iptr;  // The indirect pointer

  public:
    const T & operator*()            const { return **iptr; }
    const T * operator->()           const { return *iptr; }
    void operator=(const T * p)            { *iptr = p; }
    void operator=(const Root_C_T r)       { *iptr = *r.iptr; }
    int operator==(const void * vp)  const { return *iptr == vp; }
    int operator==(const T * tp)     const { return *iptr == tp; }
    int operator!=(const void * vp)  const { return *iptr != vp; }
    int operator!=(const T * tp)     const { return *iptr != tp; }
    int operator==(const Root_C_T r) const { return *iptr == *r.iptr; }
    int operator!=(const Root_C_T r) const { return *iptr != *r.iptr; }

    const T * value()                const { return *iptr; }

    Root_C_T()                  { iptr = (T**) _gc_RootTable.pop(); }
    Root_C_T(const T * p)       { iptr = (T**) _gc_RootTable.pop(p); }
    Root_C_T(const Root_C_T & r) { iptr = (T**) _gc_RootTable.pop(*r.iptr); }
    ~Root_C_T()                 { _gc_RootTable.push(iptr); }
};
```

Fig. 3. A smart pointer class for const objects of type T

Measurements of the efficiency of these smart pointers show them to be more expensive than raw pointers but less expensive than reference counted pointers [16]. If a global register can be dedicated to the Root Table, then initializing a new smart pointer requires two memory references and destroying one requires one memory reference. Without a dedicated global register, the cost of each of construction and destruction is increased by one memory reference. Accesses through a smart pointer pay a one memory reference penalty due to the level of indirection.

3.2 Locating Internal Pointers

Locating pointers within managed objects is the second task of the precompiler: the precompiler parses type definitions and emits a *gc()* function per garbage-collected type. This function identifies the internal pointer members to the garbage collector.

Internal Pointers and Type Tags. For every managed type the precompiler emits a gc() function. The gc() function invokes an *internal pointer*, or *ip()*, function on every pointer member of an object. The ip() function is global to the program and defined inline for efficiency. As an example, in the existing mark-and-sweep collector, the ip() function pushes internal pointer values onto the mark stack.

The precompiler emits code to register each managed type with the collector. Registration consists of a call to _gc_register() that passes in the type's gc() function pointer. Each such registration causes the garbage collector to generate and return a new type tag. Subsequent memory allocation requests pass in the tag, which is stored in the object's allocator meta-information.

Three type tags are predefined: one for objects that contain no pointers, one for objects that are entirely pointers, and one for *foreign* objects. Foreign objects are only reclaimed manually, i.e. they are never garbage collected, and there is no type information available for them. They are called foreign because they are ignorant of the presence of the garbage collector. Support for foreign objects permits this memory allocator to be the only one in the program; it can satisfy the dynamic memory needs of the standard libraries by treating their allocation calls as requests for foreign objects. Foreign objects are not examined by the collector; they should only reference collected objects through smart pointers, not through raw pointers.

The C++ feature that makes this process convenient is overloadable dynamic storage allocation operators: new and delete. These operators permit every class to supply functions to handle memory allocation and deletion. In this case, operator new for a managed class passes in the type tag to the memory allocator. The default, global operator new passes in the type tag for foreign objects. A call to malloc(), which circumvents new, also allocates a foreign object.

Figure 4 shows some sample input to the precompiler; the transformations for locating internal pointers are shown in Fig. 5.

3.3 Finalization

If the programmer specifies a static member function named T::_gc_finalize(T*), then that becomes the finalization function [31] for objects of type T. As in Cedar, finalization can be enabled or disabled for individual objects; the collector maintains a

```
class CL {
private:
  CL    * ptr1;
  OTHER * ptr2;
  static void _gc_finalize(CL *);  /* optional */
  ...
public:
  ...
};
```

Fig. 4. A class with internal pointers

```
class CL {
private:
  CL    * ptr1;
  OTHER * ptr2;
  static void _gc_finalize(CL *);
  ...
public:
  ...
private:
  static _gc_tag_t _gc_tag;
  static void _gc_(CL *);
public:
  void * operator new(size_t sz)   { return malloc(sz,_gc_tag); }
  void   operator delete(void * p) { return free(p); }
};

// The inline ip() function...
inline void _gc_ip_(void * ptr) { _gc_MarkStack.push(ptr); }

// Emitted in exactly one .C file ...
void CL::_gc_(CL * ptr)                    // the type's gc() function
{
   _gc_ip_(ptr->ptr1);
   _gc_ip_(ptr->ptr2);
}

// register type CL with the collector
_gc_tag_t CL::_gc_tag = _gc_register(&CL::_gc_, &CL::_gc_finalize);
```

Fig. 5. The internal pointers transformation

bit with every object indicating whether or not the object needs finalization. By default, finalization is enabled for an object whose class has a finalization function; a library call is available to disable or to re-enable finalization for any object.

There are no restrictions on what a finalization function can do. This means that a finalization function, which is only called when the object is unreachable, may make the object reachable. Therefore, in order not to create dangling references, an object is never reclaimed in a turn when it is finalized; it is only reclaimed after another collection confirms that it is unreachable and that finalization is disabled for it [19, 31].

A finalize function must be **static**, therefore, it may not be virtual (i.e. dynamically bound). However, since it is allowed to invoke virtual functions, the effect of a virtual finalize function is easily obtained.

3.4 Garbage Collection

The collector divides the heap into blocks that are used to allocate objects of uniform size. Using an integer division operation and knowledge of where blocks begin and end, the collector is able to make a pointer to the interior of an object (an *interior pointer*) point to the beginning of the object. This is potentially expensive because integer division can be expensive on RISC processors. Nonetheless, this ability is needed because a pointer to the beginning of the object is necessary to locate the object's type tag and mark bit. Forbidding interior pointers is impossible, firstly because the collector is sometimes conservative, also because multiple inheritance in C++ is generally implemented using interior pointers.

Garbage collection begins by examining every cell of the root table. For each cell, the collector determines if the cell points to a page that is part of the heap. If so, the value is pushed onto the mark stack. After all the roots have been pushed, the collector begins the marking traversal.

Every time a value is popped from the mark stack, the collector determines whether or not the value points into the heap, and if so, what object it references. The collector fetches the object's mark bit. If the mark bit was already set, the pointer is ignored. Otherwise, the bit is set and the type tag is fetched from the allocator's meta-information. The type tag indexes into an array of type descriptors that contain the **gc()** and finalization function pointers. The **gc()** function is called with a pointer to the object; the **gc()** function pushes the internal pointers onto the mark stack.

After the mark phase, the collector performs finalization and reclamation. For every object, one of three cases is true:

1. The object is unmarked and has finalization enabled: The object is finalized and its finalization bit is unset.
2. The object is allocated, unmarked, has finalization disabled, and is not a foreign object: The object is reclaimed. If the object's page is now empty of objects, then the page is added to the free page list. Otherwise, the object is added to the free list for its size.
3. Neither of the above is true: No action is taken.

After this, garbage collection is finished and the application resumes execution. In the next version this phase will be incremental.

3.5 this Pointers

In C++, whenever a method is invoked on an object, a pointer to the object is passed to the method on the stack. This pointer is called the this pointer. Through the this pointer the method can access the object's instance data. These pointers are part of the root set, so the garbage collector should consider them.

There are a number of different ways of finding the this pointers. For example, the precompiler could add a root local variable to every member function and assign the this pointer to the root. This would be invasive and inefficient for small member functions. Another way is to coarsely decode the stack and treat the first argument to every function conservatively in case the argument is a this pointer. This requires information about the stack frame layout that only the compiler has. In particular, the first argument to a function call is not necessarily always placed at a consistent offset in the stack frame. Thus, this either requires knowledge about the stack frame layout for each individual function, or it requires treating virtually the entire stack conservatively.

A pure copying collector must accurately find all pointers during every collection. However, a collector that does not move *all* objects does not necessarily need to find all the pointers. We do not to attempt to locate the this pointers. Instead, the programmer must ensure that the following property is always true:

> *There may not be an object whose* only *reference is through one or more* this *pointers.*

For example, the following code is illegal:

```
int main(void)
{
  (new T)->method_X();
  ...
}
```

This code is invalid because in method_X(), the object's only reference is the this pointer.

If the programmer suspects that this restriction is accidentally being violated, the collector can be configured to scan the stack conservatively in addition to using the root table. Then, the collector can report the presence of pointers on the stack to objects that would otherwise be reclaimed. The debugger can then be used to determine what code is responsible.

3.6 Controlling the Precompiler

By default, all of the class, struct, and union types that the precompiler sees are assumed to be garbage collected. Thus, for all such types, the precompiler performs its two transformations. In fact, a great many of these types are likely not to be

managed. For example, while the vast majority of C++ files include the standard header file <iostream.h>, emitting smart pointer types for the iostream classes would unnecessarily slow down compilation because there is no need for them. As an optimization, therefore, there are precompiler-specific #pragmas to control generation of garbage collection information, either at the granularity of the individual type, or at much coarser granularity. (This functionality permits programmers to take control of storage management for certain types if they so choose.)

3.7 Translation Unit Management

The precompiler processes every file in a multi-file program. Therefore, it is likely to see the class definitions multiple times. Some of the transformations are performed every time a file is compiled; others must be performed more selectively. In particular, the modifications to the class definitions are always performed so that all of the code in the program sees the same definitions. However, the gc() functions must not be replicated every time the class definitions are seeing because that would define these functions multiple times.

The precompiler uses the following heuristic to make the decision in most cases: Produce the gc() functions in the same file that defines the first non-inline function of the class. This rule tells the precompiler when to emit the gc() function for every class that has at least one non-inline function. If a managed class does not have a non-inline function member, then the precompiler will issue a warning. The user must then add a #pragma to the class telling the precompiler what file should contain the definition of the class's gc() function. This technique is used in some C++ compilers to determine when to emit the *vtbl* for a class [20, §10.8.1c].

3.8 Status

The design and development of this system are both underway. The smart pointers and the garbage collector are operational. The precompiler has been prototyped using an existing C++ compiler as the starting point. The modified C++ compiler parses the user's C++ code and emits smart pointers and other declarations. The precompiler does not yet reintegrate the emitted code back into the original source program. A complete reimplementation of the precompiler is in progress.

The SOR group at INRIA Rocquencourt has designed and is developing a distributed garbage collection algorithm [35]. The distributed garbage collector requires local garbage collectors with support for finalization. This garbage collector serves as the foundation for the distributed garbage collector.

3.9 Future Work

This collector will be used as a platform for research on the interaction between the collector and the virtual memory system. The areas of future research include VM synchronized incremental and generational collection, and influencing collection decisions based on the state of the virtual memory system.

4 Conclusions

C++ is a very well designed language considering its goals, however, the complexity of its semantics is daunting. Adding to that complexity by requiring manual storage reclamation makes programming in C++ difficult and error-prone.

Precompiling C++ programs for garbage collection is more convenient for the programmer than a pure library-based approach. Simultaneously, it is portable and not tied to any particular compiler technology. Also, it should reclaim more garbage then a purely conservative approach.

A number of other systems use smart pointers, generally for reference counting. Automatic generation of smart pointer classes can be of benefit to those projects. Similarly, the transformation that locates internal pointers is independent of the implementation of the GC algorithm, and could be used by other C++ garbage collectors.

There are three main benefits to our approach. First, the precompiler can be used as a garbage collector front-end and as a smart pointer generator. Second, this is a convenient platform for research in garbage collection techniques and issues, and will be used as such. Finally, the collector makes programming in C++ less complex and safer, and may make garbage collection available to a large part of the C++ programming community.

Acknowledgements

This work has been supported in part by Esprit project 5279 *Harness*.
I am grateful to **Marc Shapiro** for supporting this work.

References

1. ACM. *Proc. PLDI '91* (June 1991). SIGPLAN Not. 26(6).
2. APPEL, A. W. Runtime tags aren't necessary. In *Lisp and Symbolic Computation* (1989), vol. 2, pp. 153–162.
3. APPEL, A. W., AND LI, K. Virtual memory primitives for user programs. In *ASPLOS Inter. Conf. Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA (USA), Apr. 1991), pp. 96–107. SIGPLAN Not. 26(4).
4. BARTLETT, J. F. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, Feb. 1988.
5. BARTLETT, J. F. Mostly copying garbage collection picks up generations and C++. Tech. Rep. TN-12, DEC WRL, Oct. 1989.
6. BOEHM, H.-J. Simple gc-safe compilation. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
7. BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. Mostly parallel garbage collection. In *Proc. PLDI '91* [1], pp. 157–164. SIGPLAN Not. 26(6).
8. BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Softw. – Pract. Exp. 18*, 9 (Sept. 1988), 807–820.
9. CHRISTOPHER, T. W. Reference count garbage collection. *Softw. – Pract. Exp. 14*, 6 (1984), 503–508.

10. COPLIEN, J. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.
11. DEMERS, A., WEISER, M., HAYES, B., BOEHM, H., BOBROW, D., AND SHENKER, S. Combining generational and conservative garbage collection: Framework and implementations. In *Proc. POPL '90* (Jan. 1990), ACM, ACM, pp. 261–269.
12. DETLEFS, D. Concurrent garbage collection for C++. Tech. Rep. CMU-CS-90-119, Carnegie Mellon, 1990.
13. DICKMAN, P. Trading space for time in the garbage collection of actors. In unpublished form, 1992.
14. DYBVIG, K. R. *The SCHEME Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 1987.
15. DYKSTRA, D. Conventions on the use of ObjectStars, 1992. Private communication.
16. EDELSON, D. R. Comparing two garbage collectors for C++. In unpublished form, 1992.
17. EDELSON, D. R. Smart pointers: They're smart but they're not pointers. In *Proc. Usenix C++ Technical Conference* (Aug. 1992), Usenix Association, pp. 1–19.
18. EDELSON, D. R., AND POHL, I. A copying collector for C++. In *Proc. Usenix C++ Conference* [40], pp. 85–102.
19. ELLIS, J. Confirmation of unreachability after finalization, 1992. Private communication.
20. ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Feb. 1990.
21. FERREIRA, P. Garbage collection in C++. Workshop on GC in Object Oriented Systems at OOPSLA '91, July 1991.
22. GINTER, A. Cooperative garbage collectors using smart pointers in the C++ programming language. Master's thesis, Dept. of Computer Science, University of Calgary, Dec. 1991. Tech. Rpt. 91/451/45.
23. GOLDBERG, B. Tag-free garbage collection for strongly typed programming languages. In *Proc. PLDI '91* [1], pp. 165–176. SIGPLAN Not. 26(6).
24. GROSSMAN, E. Using smart pointers for transparent access to objects on disk or across a network, 1992. Private communication.
25. KAFURA, D., WASHABAUGH, D., AND NELSON, J. Garbage collection of actors. In *Proc. OOPSLA/ECOOP* (Oct. 1990), pp. 126–134. SIGPLAN Not. 25(10).
26. KARPLUS, K. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI* (Pasadena, CA, 20-22 March 1989), C. L. Seitz, Ed., MIT Press, pp. 101–118.
27. KENNEDY, B. The features of the object-oriented abstract type hierarchy (OATH). In *Proc. Usenix C++ Conference* [40], pp. 41–50.
28. MADANY, P. W., ISLAM, N., KOUGIOURIS, P., AND CAMPBELL, R. H. Reification and reflection in C++: An operating systems perspective. Tech. Rep. UIUCDCS-R-92-1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Mar. 1992.
29. MAEDER, R. E. A provably correct reference count scheme for a symbolic computation system. In unpublished form, 1992.
30. MILLER, J. S. *Multischeme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, MIT, 1987. MIT/LCS/Tech. Rep.-402.
31. ROVNER, P. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Tech. Rep. CSL–84–7, Xerox PARC, 1984.
32. RUSSO, V. Garbage collecting an object-oriented operating system kernel. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
33. RUSSO, V. There's no free lunch in conservative garbage collection of an operating system, 1991. Private communication.

34. RUSSO, V. Using the parallel Boehm/Weiser/Demers collector in an operating system, 1991. Private communication.
35. SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing* (Vancouver, Canada, Aug. 1992), ACM.
36. SHAPIRO, M., GOURHANT, Y., HABERT, S., MOSSERI, L., RUFFIN, M., AND VALOT, C. SOS: An object-oriented operating system—assessment and perspectives. *Comput. Syst. 2*, 4 (Dec. 1989), 287–338.
37. SHAPIRO, M., MAISONNEUVE, J., AND COLLET, P. Implementing references as chains of links. In *Workshop on Object Orientation in Operating Systems* (1992). *To appear.*
38. STROUSTRUP, B. The evolution of C++ 1985 to 1987. In *Proc. Usenix C++ Workshop* (Nov. 1987), Usenix Association, pp. 1–22.
39. STROUSTRUP, B. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1991.
40. USENIX ASSOCIATION. *Proc. Usenix C++ Conference* (Apr. 1991).
41. WENTWORTH, E. P. *An environment for investigating functional languages and implementations.* PhD thesis, University of Port Elizabeth, 1988.
42. WENTWORTH, E. P. Pitfalls of conservative garbage collection. *Softw. – Pract. Exp.* (July 1990), 719–727.
43. YASUGI, M., AND YONEZAWA, A. Towards user (application) language-level garbage collection in object-oriented concurrent languages. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
44. ZORN, B. The measured cost of conservative garbage collection. Tech. Rep. CU-CS-573-92, University of Colorado at Boulder, 1992.

This article was processed using the LaTeX macro package with LLNCS style

GC-cooperative C++

A. Dain Samples*

Dept. of Electrical and Computer Engineering
University of Cincinnati
Cincinnati, OH 45221-0030
Dain.Samples@uc.edu

Abstract. A garbage collector for C++ should maintain the spirit of the language as much as possible and yet provide mechanisms for reliable development and debugging of programs utilizing garbage collection. This paper proposes a design of such a system, including a minimal set of language changes, and compile-time and runtime environment enhancements. The design provides support for many different kinds of garbage collection strategies (copying, mark and sweep, generational, ...), does not impose extensive overhead on runtime objects that do not use garbage collection, and imposes as few restrictions on programming style as possible.

1 Introduction

This work builds on and is to be contrasted with the work of Edelson [1–3], Bartlett [4,5], Detlefs [6], and Ginter [7]. All of these efforts have concentrated on user-defined garbage collectors; that is, garbage collectors written in C++ as currently defined without specific support from the compiler. The goal of a library of garbage collectors unsupported by the compiler and language is laudable but, as their efforts show, fraught with difficulties.

This proposal will concentrate on the design of the C++ language and compiler to support (cooperate with) garbage collection (GC). Whereas previous work has concentrated primarily on building garbage collectors within the current definition of C++, we are asking the question "What minimal changes to the language and compiler would be necessary to provide direct, efficient support for a garbage collection facility?" Such a facility would effectively be predefined much as the *malloc* and *free* functions are defined; the facility is assumed to be part of the language environment without specifying how it is implemented. A design of a GC facility for C++ should not preclude *a priori* GC strategies; specifically, it should not preclude copying collectors.

The design is preliminary and is presented for criticism and discussion. Consider it a manifesto of desirable GC properties, an exploration of design ramifications, and some explicit suggestions for implementation. We are currently developing a prototype to empirically evaluate some of the design decisions.

To streamline what follows, we define some terms. C++ classes that have been declared collectible are referred to as *gc-classes* or *gc-types*, their instantiations as

* This work supported by the Languages Development Unit of Borland, International

gc-objects, and the space in which they are collected as *gc-space*. Objects that are pointed to by an application from outside gc-space are said to be *rooted*, and the pointers are called *roots*. Gc-objects to which there exists a path of pointers from a rooted object are said to be *grounded*; rooted objects are trivially grounded. Pointers to or into gc-objects are called *gc-pointers*. Gc-pointers *into* gc-objects are also called *embedded* pointers (also called *interior* pointers by some). Variables that contain bit-patterns that could be interpreted as gc-pointers but might actually be something that just happens to look like a gc-pointer (e.g., an integer, a sequence of characters, etc.) are called *ambiguous roots*. The act of informing the GC system that a pointer may point to a gc-object is called *registration*.

Garbage collection strategies are categorized as either *in-place* or *copying* collectors. In-place collectors do not move live objects once they are allocated, whereas copying collectors may change the memory address of objects at any time during garbage collection.

2 Previous Work

Edelson has implemented at least two different collectors, an in-place mark-and-sweep using root indirection [2] and a copying collector using root registration [3]. Root indirection means the application can only manipulate indirect pointers through a table; the collector uses the table to track objects in gc-space. Root registration means the application manipulates direct pointers, and a separate mechanism is used to track pointers which the collector needs. Both collectors use *smart pointers*: class definitions that use their constructors, destructors, and an overloaded -> operator to manipulate pointers in a controlled manner [8]. The C++ compiler was not modified in either prototype to provide any cooperation for GC.

Bartlett's mostly-copying scheme [4,5] cleverly combines copying and *conservative* collection [9]. Those objects that are referenced by ambiguous roots are not copied, while all other live objects are copied. The *to-* and *from-spaces* central to the copying scheme are implemented by *page promotion*; pages of *from-space* memory containing objects referenced by ambiguous roots are simply redesignated as *to-space* pages. In his applications, memory fragmentation and uncollected garbage rates were surprisingly reasonable and a function of the selected page size [4]. He later modified his collector to use a generation scavenging scheme [5].

Detlefs [6] also uses Bartlett's mostly-copying scheme but found that his application (the AT&T *cfront* C++-to-C translator) had far too many ambiguous roots and retained too much garbage. Most of the ambiguous roots came from extensive use of *unions* in data structures; he found it necessary to restructure many of these unions to alleviate the problem. He borrows from Modula-3 [10] the notion of *traced* and *untraced* objects and pointers; this is analogous to our collected versus uncollected objects and pointers. However, we argue below that *traced* and *untraced* do not adequately capture the notion of pointers that root objects.

Detlefs' implementation walked a thin line: he rejects any scheme that requires a *GC cooperative* compiler, and yet proposes a scheme that nevertheless requires enhancements to the C++ compiler. He justifies this decision by the fact that his scheme can easily be integrated into existing compilers and that his approach promotes portability.

Ginter's report [7] analyzes the impact on the C++ language if user-defined garbage collectors are to be supported. His recommendations include enhancing the declaration syntax of smart pointers and implementing mechanisms for determining the structure of objects at runtime. His report points out many of the difficulties of supporting user-defined collectors in C++, some of which appear inherent to the language.

We do not summarily reject the notion of user-defined collectors; however, the problems pointed out by Ginter are difficult ones, and providing even incomplete support requires significant changes to the C++ language. For instance, Ginter, Edelson, and Detlefs all point out the problem for user-defined collectors when dealing with C++'s `this` pointer. The `this` pointer is allocated on the stack for every call of a member function. However, the user cannot extract the address of *this* and therefore cannot register it with the GC system: this can be done only by the compiler. Hence, our approach.

We are proposing that C++ compilers provide a garbage collection environment as part of the system, not as a library routine. In this report, we explore what C++ compilers can do, the language enhancements required, and some of the implementation issues. We also explore the possible GC strategies that one can reasonably expect C++ to support.

3 Goals of the Design

As mentioned, the primary goal of this design is to retro-fit C++ with garbage collection facilities while not changing the spirit of the language. That spirit, based on the spirit of C, can be summarized provocatively as: (1) Programmers should be able to do anything they want, subject to their willingness to deal with the consequences (side-effects and interactions of features whose invariants are violated). (2) No one pays for any feature of the language that they do not use.

Based on this spirit, we consequently adopt the following goals for the system.

Adding GC to C++ should not force overhead on users that do not use it. When GC is used, it should not interfere with software modules that do not use it. GC must not interfere with the current and/or standard definitions of the language. That is, a GC facility should not cause re-definition or loss of language features.

The dual of this requirement is that programmers that use GC must take the responsibility either for not violating the invariants of the design, or for knowing how to deal with the consequences. Almost any restriction imposed by a GC-cooperative C++ compiler can be bypassed by a sufficiently determined programmer. The compiler therefore can take responsibility only for maintaining its GC invariants, but cannot be expected to find any and all ways in which users may have violated those invariants.

The language should be changed as little as possible. A strict goal of no change was seen rather quickly to be unrealistic. Either all runtime objects would have to be garbage collected—a *major* change to the design of the language and a violation of the spirit of the language—or, at a minimum, programmers would need language support to be able to specify which (classes of) objects are to be garbage collected. A design consequence of this goal is that a program can have two dynamic memory

spaces: the space dedicated to objects that are to be garbage collected, and the usual 'heap' of dynamically allocatable memory.

Use of a garbage collector must not impose a complete dichotomy on runtime objects: collected objects must be able to refer to, contain, and be contained in non-collected objects, and *vice versa*. Gc-objects are not restricted to exist only in gc-space: gc-objects should be instantiable as global, automatic, or heap objects, if the programmer so desires. Obviously, only those that are in gc-space can and will be automatically reclaimed. While it may be necessary to add data to an object to support GC, it must be the case that objects of a type have exactly one memory representation, no matter where they reside (global space, stack, heap, or gc-space). Furthermore, it should be possible for the user to declare that a specific object is to be automatically reclaimed. For instance, the programmer should not have to declare a special class or data type to have arrays of characters (strings) automatically reclaimed.

Arrays of gc-objects must be supported.

The design should support pointers into objects and pointer-to-member. For instance, if a gc-object has an array of characters as an element, the programmer should still be able to traverse that array with a pointer-to-char. Compiler and runtime mechanisms must exist that allow correct reclamation of gc-objects that have such 'embedded' pointers into them. ('Embedded' here refers to where the pointer is pointing, and not to the location of the pointer itself.)

C unions and Pascal tagless variant records are a way of hiding information from the compiler and/or exposing representation information to the user. Unions should be supported wherever possible. (However, unions can be supported only with non- or mostly-copying GC strategies; cf. §7).

It would be nice if the design did not preclude supporting multiple and discontiguous gc-spaces, where each gc-space may have a different GC strategy. The prototype being constructed does not yet support multiple gc-spaces, and so this paper does not pursue this goal directly. The problem of cross-registration of root pointers between GC strategies is complex enough that more data is needed on the desirability of this feature to justify a design to support it at this time.

4 Impact on Language Design

Syntactic changes to the C++ language definition consist only of three additional keywords: **collected**, **embedded**, and **heap**. The programmer must be able to specify which objects are to be reclaimed automatically, so classes of objects to be garbage collected are declared to be of type **class collected** (Figure 1).

If a program uses embedded pointers, then the programmer must either declare such pointers to be **embedded** (Figure 2)[2], or ensure that any object so referenced is rooted by some pointer to the object itself. We also note that declaring a pointer **embedded** does not mean that its value *is* an address inside a gc-object, only that it *may* be a pointer into a gc-object. The garbage collector will be required to determine if it is.

[2] Examples are cumulative, and use declarations from previous examples.

```
class collected Gtype {
public:
   char  carr[32];
   :
   } Gobject;
```

Fig. 1. Declaration of a gc-type

```
void eg1(void)
{
  char embedded* ep;
  char* notep;

  ep    = &Gobject.carr[3];     // OK
  notep = &Gobject.carr[3];     // Illegal
      // At the least, the compiler should give
      // a warning message for the above.
  notep = (char *)&Gobject.carr[3]; // Legal
}
```

Fig. 2. Embedded pointers

The problem pointed out by Ginter [7] concerning the creation of **this** pointers is solved by the use of embedded pointers. In his example (Figure 3, translated into our notation), the call of the member function **Xfcn** creates a **this** pointer on the stack that points to the (uncollected) **Y** object that is a member object of a **base** object. The **base** object is allocated in gc-space, so the **this** pointer is actually an undeclared embedded pointer. The assignment of **this** to **X** in **Xfcn** is therefore legal in Ginter's construction, although potentially creating a serious error for the programmer. Since objects of type **member** are never allocated in gc-space except as member elements of other objects, we do not want to have to declare the **member** class to be **collected**. Ginter suggests that an error message be emitted every time pointer to a **traced** object is stored into an **untraced** pointer. This is not acceptable since every invocation of a member function of **member** would create such an error message.

Our solution is to declare class **member** what it is: embedded. Declaring **class embedded member**{...} means that the **this** pointer on the stack is appropriately registered without forcing the unnecessary overhead of making **member** objects collectible, and without the proliferation of runtime error messages. Now, the assignment **X = this** can be flagged as an error by the compiler as the assignment of an embedded pointer into an non-gc-pointer, a much more reasonable error message.

Gc-objects may be allocated on the heap by using the keyword **heap** with the **new** operator; likewise, non-gc-objects that are to be automatically reclaimed use the keyword **collected** in a similar manner (Figure 4). If neither modifier is present, then an object of the indicated type is allocated from the space appropriate to that

```
class member* X;
class member {
public:
   void Xfcn(void) { X = this; }
};

class collected base {
   class member Y;
};

void fcn(void) {
   class base* Z;
   Z = new base;
   (Z->Y).Xfcn();
   :
}
```

Fig. 3. Example of the `this` problem (from Ginter)

type. We also note in passing that the declaration of `cstr` in Figure 4 must be declared `embedded`.

```
void eg2(void)
{
  char* str            = new char[64];           // allocated on heap
  Gtype* Gptr          = new Gobject;            // allocated in gc-space
  Gtype* Gptr2         = new heap Gobject;       // allocated on heap
  char embedded* cstr = new collected char[32]; // allocated in gc-space
}
```

Fig. 4. Use of `new` operator

5 Runtime Organization

The following description of the runtime organization is sufficiently detailed to allow a straightforward (and probably slow) implementation, and does not go into any detail on optimization techniques. This design has concentrated on generality while keeping an eye on implementation issues with the belief that the design does not preclude optimizations.

5.1 Object Formats

Three pieces of information are required about an object: how to traverse the object (what other gc-objects does this one point to), GC-specific data (e.g., mark bits), and finalization information (e.g., is there a destructor to be called when this object is deallocated). This information is maintained in two objects, the *gc-wrapper*[3], and the *gc-descriptor*. There is one gc-wrapper for each gc-object allocated, and one gc-descriptor for each gc-type.

The first field of each gc-object is a pointer to its gc-wrapper; this field is hidden from the programmer in the same way the virtual function table pointer is hidden. One of the fields of the gc-wrapper is a pointer to the appropriate gc-descriptor. (It is an implementation question as to whether a gc-object with virtual functions will have two hidden pointers, or whether the gc-wrapper and the virtual table are reached via a single pointer in the gc-object.)

The gc-descriptor, generated from the declaration of the gc-type in much the same way as described by Detlefs [6], contains the location of all gc-pointers contained in objects of this type.

Consider the code in Figure 5. All three pointers are referencing three views of the same object. If any two of the pointers are set to `nil`, then the remaining pointer must root the object. Therefore the collected base class component(s) of a gc-object must refer to the outermost containing object. This is effected by having the gc-wrapper for the contained base object be the gc-wrapper for the containing object. Therefore, in Figure 5, `gcw_B == gcw_D`. This, of course, has implications for the initialization (construction) of gc-objects.

The 'embedded' concept frees us from requiring that base classes of collected classes must also be collected. If base classes of gc-classes are not themselves gc-classes, then declaring them **embedded** maintains the type discipline without the overhead they would acquire if declared **collected**. If the base classes are also gc-classes, then pointers to the base components of a gc-object do not need to be declared **embedded** since the gc-wrapper for the contained base object (which is also the gc-wrapper for the outermost containing object) contains all of the information necessary for the collector.

5.2 Arrays of Gc-Objects

Arrays of gc-objects (let's call them gc-arrays) require a special gc-wrapper that each gc-object in the array references as its own gc-wrapper. The gc-array's gc-wrapper will contain a pointer to the base of the array, the number of elements in the array, and the descriptor for the individual elements of the array. Gc-pointer-to-gc-object behaves exactly as pointer-to-object behaves. If there is a reference to any element of the gc-array, then the entire array can be scavenged.

[3] The name gc-wrapper betrays an earlier design in which the data actually surrounded the object. This presented problems for the definition of the C++ *sizeof* operator, so the information was at least conceptually pulled into a separate runtime object.

```
class collected Base { int i; };
class embedded  Vile { int i; };
class collected Derived: Vile, Base { int i; };
Base *bp;
Vile embedded* vp;
Derived *dp;

main()
{
  dp = new Derived;
  bp = (Base*)dp;
  vp = (Vile embedded*)dp;
  ...
}

// layout of Derived objects:
//  gc_wrapper*         gcw_D       dp points here
//  int                 Vile::i     vp points here
//  gc_wrapper*         gcw_B       bp points here
//  int                 Base::i
//  int                 Derived::i
```

Fig. 5. Base and derived classes

5.3 Non-Gc-Objects Allocated in Gc-Space

If a programmer does not wish to track references to a particular dynamic object, that object can be allocated in gc-space with a **new collected** call. For instance, the function

```
char embedded*
copy_name(char *cp)
{
    char embedded* gcp = new collected char[strlen(cp)+1];
    strcpy(gcp, cp);
    return gcp
}
```

returns a pointer to an instance of an array of characters that will be reclaimed automatically when there are no references to the string. The declaration of any character pointer that might end up pointing to or into this array of characters must be declared **embedded**.

To track these non-gc-objects allocated in gc-space, a gc-type that contains the non-gc-object or the array of non-gc-objects as its only member is constructed by the compiler. Such non-gc-objects may not contain gc-objects or pointers to gc-objects. Objects containing gc-pointers or gc-objects should probably be declared **collected**

anyway, but the restriction avoids problems with the registration of gc-objects or pointers-to-gc-objects in objects that are suddenly become gc-objects.

5.4 Gc-Wrapper Format

Each gc-object and gc-pointer has a gc-wrapper; elements of a gc-array all share the same gc-wrapper, and a gc-object that has member gc-objects or gc-pointers shares its gc-wrapper with its members. Hence the requirement for the pointer-to-object field within a gc-wrapper in order to find the address of the outermost containing object. The purpose of the gc-wrapper is two-fold: (1) to customize the information necessary to trace an object or pointer, and (2) to be the proxy for the object or pointer in a root set.

Currently, there are three gc-wrappers defined in the system; their formats are indicated by the template class definitions in Figure 6.

```
enum gcwTag {
     GCW_OBJ,
     GCW_OBJPTR,
     GCW_ARRAY,
     };

template<class collected T> class gcw_obj {
    gcwTag      tag;        // = GCW_OBJ
    T*          object;
    gc_desc*    gd;
    gc_data     d;    };    // e.g., mark bits

template<class collected T> class gcw_obj {
    // for global and local pointers
    gcwTag      tag;        // = GCW_OBJPTR
    T**         object;  };

template<class collected T> class gcw_obj {
    // for arrays with elements of type T
    gcwTag      tag;        // = GCW_ARRAY;
    T*          base;
    gc_desc*    gd;         // the descriptor for the elements
    int         n;       }; // nof elts of the array
```

Fig. 6. Format of the gc-wrappers

It is interesting to note that, like virtual table pointers, gc-wrappers cannot be directly defined within a C++ class hierarchy. Figure 7 shows a straightforward attempt to implement pointers to gc-wrappers for all gc-objects by deriving all collected objects from a **GC_CLASS** type. The code fragment in the **main** procedure attempts to access the gc-wrapper for a **UserDerived** object. Because of C++'s

inheritance design, however, there are two gc-wrapper pointers to which the name **gcwp** might refer in this context: the one associated with the **UserDerived** object by inheritance from **GC_CLASS**, and the one associated with the base class **UserBase**. The language definition of C++ currently does not have a mechanism for unambiguously identifying either these two gc-wrapper pointers.

```
class GC_CLASS { public: GC_WRAPPER* gcwp; };
class UserBase : public GC_CLASS { ... }; // a collected class
class UserDerived : public GC_CLASS, // because UserDeriveds are collected
                    public UserBase   // because user derived
                    { ... };
main() {
     UserDerived* udp = new UserDerived;
     ... udp->gcwp ...
     }
```

Fig. 7. C++ ambiguous reference problem

While it is not within the proper scope of this paper to propose a solution to this problem with C++, perhaps a partial solution would be to allow some disambiguation via class names; e.g., **udp->UserBase::gcwp** and **udp->GC_CLASS::gcwp**. However, it is still possible to construct a class hierarchy that has ambiguities that cannot be so resolved. Suffice it to say that, because of this problem, a full prototype of this design cannot be implemented within the C++ class hierarchy design without significant work-arounds.

5.5 Implications for Allocation

The greatest change to C++ implied by these design decisions involve the constructors and the allocator. The compiler and runtime system will have to be modified to give each gc-object's constructors sufficient information to initialize the gc-wrappers. More accurately, this information must be given to the registration routines that determine how an object will be registered. The registration routines are conceptually a part of the operation of the **new** operator for gc-classes. Specifically, two pieces of information must be passed to the registration routines for gc-objects and gc-pointers:

1. *Where* is the gc-object/gc-pointer being allocated? There are four possible values:
   (a) On the stack (auto/local variables)
   (b) Static/global space
   (c) On the heap
   (d) In gc-space
2. *How* is the gc-object/gc-pointer being allocated?
   (a) By itself.

(b) As an element of an array; in which case the registration routines will need to know the address of the first object in the array.
(c) as a (sub-)member of a gc-object; in which case the registration routines will also need the base address of the outermost gc-object.

'Sub-member' refers to object member nesting. For example,

```
class collected Gin { ... };
class           Nog { ... Gin gin; ... };
class collected Gout{ ... Nog nog; ... };
class           Nog2{ ... Gout gout; ... };
```

The registration routines for gc-class **Gin** need to know the address of the containing gc-object. Therefore, when a **Nog2** is allocated, the address of **Gout**, not **Nog2** is passed to the registration routines for **Gin**. This requirement also holds for base class/derived class relationships.

Of course, user-defined **new** operators are prohibited for gc-classes.

5.6 Performance Considerations

The actual implementation of the gc-descriptor may be a table of offsets, or it may be executable code that recursively invokes GC routines to mark or scavenge reachable objects. Which is the best implementation has not yet been determined: the table of offsets is smaller but slower, while the executable code is faster but requires more memory. Detlefs [6] defined several encoding schemes for a trace description of gc-objects.

Allocation and initialization of gc-wrappers when creating a gc-object or gc-pointer will slow down allocation, and will need to be engineered carefully. In general, every gc-object must have a gc-wrapper created and initialized for it since the gc-object may be referenced by a traced gc-object. If it can be determined that an object will never be so referenced (e.g., the address of a stack-allocated gc-object is never computed) then the gc-wrapper is not necessary.

However, not every gc-object outside of gc-space need be registered: gc-objects must be registered only if they contain references to other gc-objects. Automatic (local, or stack) variables present a special performance problem since each invocation of a function may require each gc-object declared local to the function to be registered. There are several optimizations that may be effective in reducing this overhead; our first prototype sidesteps the whole issue by *not* registering objects on the stack. Instead, in our experiments, we will measure the performance of conservative collection of the stack versus that of full registration. This is to be contrasted with the work of Appel [11], Goldberg [12], and Diwan, Moss, and Hudson [13] in which a descriptor of the stack frame guides (non-conservative, precise) collection.

As mentioned above, the Modula-3 distinction between **traced** and **untraced** pointers is not sufficient for this environment; combined with **collected** and (implied) uncollected objects, **embedded** declarations actually improve the ability of the compiler to generate more efficient code. When an embedded gc-pointer points into the middle of a gc-object, the beginning of that object must be found (which has implications for the allocator; our prototype will be using the beginning-of-object

bitmap allocator described in Detlefs [6]). This search does not need to be performed for pointers known to point to the head of a gc-object, only for pointers that have been declared **embedded**.

Since we allow gc-objects to be allocated anywhere, gc-pointers will always have to be checked to see if they point into gc-space. If this overhead is too onerous, then we may need to explore either restricting gc-pointers to point *only* into gc-space, or allowing the programmer to specify which gc-pointer variables always point into gc-space and those which may point elsewhere.

5.7 Root Sets

The root sets are the collection of objects outside of gc-space that may reference gc-objects in gc-space. Each set is designed to simplify the registration and tracing of each of the different kinds of objects and where they are allocated. For instance, gcr_statics records the location of gc-objects and pointers in global data space. This is most simply implemented as a compiler-initialized array of pointers. On the other hand, the gcr_heap set may be most efficiently implemented as a doubly-linked list. Therefore, we make no assumptions as to the implementation of the root sets. Indeed, one implementation might be to have only one root set that contains the union of all of the following sets.

gcr_statics: The set of all globally allocated gc-objects.
gcr_heap: The set of all objects allocated on the heap.
gcr_stack: The set of gc-objects allocated on the runtime stack.
gcr_staticptrs: The set of all globally allocated pointers to gc-objects.
gcr_stackptrs: The set of all pointers-to-gc-objects allocated on the stack.
gcr_heapptrs: The set of all pointers-to-gc-objects allocated on the heap.
gcr_staticembptrs, gcr_stackembptrs, gcr_heapembptrs: The set of all embedded pointers (pointers pointing into a gc-object). Embedded pointers require more processing by the collector, so it makes sense to isolate them into their own root sets. If experience shows otherwise, we can simplify things and include embedded pointers into the root sets for gc-object-pointers.

There is a potential problem with gc-objects falsely being added to a root set. If a gc-object or gc-pointer is put in a root set, then there must be a way of removing it from that set when the object/pointer is reclaimed. For instance, if the compiler sees the following:

```
class collected G1 { ... };
class           G2 { ... G1 g1; ... };
class collected G3 { ... G2 g2; ... };
```

then, assuming a straightforward implementation of root registration, any time a new **G2** is created on the heap, a **G1** object is added to the **gcr_heap** root set. In a straightforward implementation, a **G1** object will be added to the heap root set even when a **G2** object is allocated in gc-space as part of a **G3** object. We must carefully handle the root set to make sure that the reference to **g1** in the root set allows **G3** objects to be reclaimed. That is, when the only active reference to a **G3** object is

the root set reference to its sub-member `g1`, then `G3` needs to be reclaimed *and* `g1` needs to be removed from the root set.

Edelson [2] calls these *weak pointers*. That is, they are pointers that do not make the pointed-at object 'live'. The object is live only if an application pointer reaches it from another live object or root. The easiest solution is for the collector to check each member of the heap-space registration set to see if any gc-object in that set is not in heap-space, but rather is in gc-space. If such an object is found in the heap-space set, it is removed from the registration set and not traversed.

6 Programmer Impact

There are still pitfalls in C++ that the programmer must aggressively guard against. The programmer can get into serious trouble rather innocently. For instance, there is not much to be done to guard against the following error except perhaps to provide some run-time checking measures and suggest alternative coding styles. Consider the code in Figure 8. If the call on `randomFcn` causes a garbage collection, the array, which has no references to it anymore (`p` points beyond the end of the gc-array, and `polygon` has been set to `nil`), may be collected, in which case the last call on `adjust()` will fail. Even if `polygon` is not set to `nil`, the array might be moved. In that case, `p` will most likely not be updated correctly because it is not pointing into the gc-array. This is one place where problems caused by C++-encouraged coding styles cannot be overcome. The programmer will just have to be aware of the possibility of this error, and code appropriately. In Figure 8, the programmer should consider the pointer `p` local to the loop.

One possibly mitigating solution to this problem is to append a byte to the end of each gc-array. The address of this byte would keep the one-beyond pointers pointing into the gc-array's space, rooting the array and allowing the pointer to be updated if the array were moved.

Even more disastrous and mysterious bugs will occur when programmers overwrite information at the end of an array (e.g., with an off-by-one error on the length of the array). What follows the array in gc-space could possibly be the gc-array's gc-wrapper (depending on the allocation strategy used for gc-wrappers) or the gc-wrapper field at the head of another gc-object. Such errors may cause very mysterious behavior in the garbage collector, and could be very difficult to track down. Again, this is not a problem isolated to garbage collection environments: it is an error that is almost encouraged by C++'s pointer definition of arrays.

7 Evaluation

The goal of this design was to find a minimum set of language enhancements and attendant compile-time and runtime environment enhancements to support as many different garbage collection strategies as possible. For any one garbage collection strategy used by a C++ compiler, not all of this design will be necessary. For instance, we are currently constructing a prototype that will keep track of root sets for

```
class collected Point
{
    float x, y;
    void    adjust();
};

main()
{
    Point* polygon = new Point[100];
    Point* p;
    ...
    // adjust polygon
    for (p = &polygon[0]; p < &polygon[100]; p++) {
        p->adjust();
        }
    polygon = nil;
    randomFcn();
    --p->adjust(); // can fail!
}
```

Fig. 8. Array reference problem

global and heap spaces, but will do a conservative trace of the runtime stack. Therefore, gc-objects and gc-pointers allocated on the stack will not incur the overhead of root set registration.

In-place and copying differ primarily in the fact that copying collectors must find every gc-pointer to or into a moved (and therefore live) gc-object in order to update the pointer value; in-place collectors need only to find one valid gc-pointer for each live gc-object. Copying collectors are possible to the extent the compiler can register its temporaries. Expression temporaries derived from gc-pointers should be considered embedded and registered as such. If the compiler cannot maintain this invariant, then copying collectors will not work. Bartlett's mostly-copying scheme is the next best compactifying garbage collector that can be used in such an environment.

The only way tagless unions can be supported is if all pointers contained in the union are treated as embedded pointers; i.e., all pointer fields of all possible overlaid objects are first checked to see if they do indeed point to or into a gc-object. That gc-object can then be considered reachable. Because this is a form of conservative garbage collection, any gc-object marked by one of these possible pointers cannot be moved because the pointer is only a possible pointer and cannot be updated. Therefore, if a copying GC strategy is used, tagless unions containing pointers to gc-objects cannot be supported at all.

We have successfully met many desirable goals. Only gc-objects and gc-pointers have associated overhead. Objects that are not collected and do not reference collected objects are the same as they were. In addition, we have solved some problems noted by previous work in this area. The language changes are minimal, and previous work has shown the efficacy of many of the ideas incorporated into the system.

In summary, the design is robust enough to support many different kinds of garbage collection strategies, and offers many opportunities for efficient implementation of specific strategies.

References

[1] Daniel Ross Edelson, "Dynamic Storage Reclamation in C++," Master's Thesis, Univ. of California, Santa Cruz, UCSC-CRL-90-19, June 1990.

[2] Daniel Ross Edelson, "A Mark-and-Sweep Collector for C++," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albuquerque, NM (Jan. 19-22, 1992).

[3] Daniel Ross Edelson & Ira Pohl, "A copying garbage collector for C++," *Usenix C++ Conference Proceedings*, Washington, D.C. (Apr. 1991).

[4] Joel F. Bartlett, "Compacting Garbage Collection with Ambiguous Roots," Digital Equipment Corp., Western Research Center, WRL 88/2, Feb. 1988.

[5] Joel F. Bartlett, "Mostly-Copying Collection Picks Up Generations and C++," Digital Equipment Corp., Western Research Center, TN-12, Oct. 1989.

[6] David L. Detlefs, "Concurrent Garbage Collection for C++," Carnegie-Mellon Univ., CMU-CS-90-119, Pittsburgh, PA, May 1990.

[7] Andrew Ginter, "Design Alternatives for a Cooperative Garbage Collector for the C++ Programming Language," Dept. of Computer Science, Univ. of Calgary, Research Report No. 91/417/01, Calgary, Alberta, Canada, Jan. 1991.

[8] Bjarne Stroustrup, "The evolution of C++: 1985 to 1987," *USENIX C++ Workshop Proceedings* (1987).

[9] Hans-Juergen Boehm & Mark Weiser, "Garbage Collection in an Uncooperative Environment," *Software–Practice Experience* 18 (Sept. 1988), 807–820.

[10] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow & Greg Nelson, *Modula-3 Report (revised)*, Digital Equipment Corp., Western Research Center, 1988.

[11] Andrew W. Appel, "Runtime tags aren't necessary," *Lisp and Symbolic Computation* 2 (1989), 153–162.

[12] Benjamin Goldberg, "Tag-Free Garbage Collection for Strongly Typed Programming Languages," *Proceedings of the ACM-SIGPLAN 1991 Conference on Programming Language Design and Implementation* 26 (June 26-28, 1991), 165–176.

[13] Amer Diwan, Eliot Moss & Richard Hudson, "Compiler Support for Garbage Collection in a Statically Typed Language," *Proceedings of the ACM-SIGPLAN 1992 Conference on Programming Language Design and Implementation* 27 (June 17-19, 1992), 273–282.

This article was processed using the LaTeX macro package with LLNCS style

Dynamic Revision of Choice Points during Garbage Collection in Prolog [II/III]

Jean François PIQUE

Groupe Intelligence Artificielle,
Université Aix-Marseille II,
Luminy case 919,
13288 Marseille CEDEX 9, France
Email: jfp@gia.univ-mrs.fr

Abstract. We describe a technique allowing efficient reconsideration of all the current choice points whenever the garbage collector is activated. It turned out that this extra work usually saves time since the garbage collector will have less structures to consider. Moreover this extra work will be deducted from future work, if the search tree is not pruned. This technique also allows recovery of space in the local stack of the wam, thus making garbage collection of this stack worthwhile.
There is no additional cost in space if a trail with (address,value) pairs is used. The Dynamic revision algorithm can however be implemented with a standard trail if space the same size as the trail contents is made available at garbage collection time.

1 Introduction

The detection of the determinism of execution of a Prolog procedure enables the environment (mainly the "local variable" space) to be recovered immediately when the procedure terminates. It also increases the number of heap cells which can be reclaimed by the garbage collector since choice points are avoided. However the detection of this determinism, though fundamental for memory recovery, is an operation which often cannot be taken very far during compilation.

To deal with this problem, the most frequent solution is to generate instructions which enable this case to be detected dynamically. One constructs an index on the first argument of the procedure, and code is generated which enables the instantiation of this argument to be used [Warren 83]. This method is an acceptable compromise in relation to the time required for compilation and the increase in size of the code generated (between 20% and 100%). It makes use of the fact that the programmer has a tendency to use first the arguments which are known at the time of the call.

Some implementations create an index on several arguments and/or integrate into the selection code certain primitives located at the start of the rule body. However the code generated quickly becomes complex and voluminous (especially when the number of arguments grows). Worse, It has been shown that complete indexing is NP-complete in the worst case [Hickey and Mudambi 89]. In addition, it is not compatible with incremental modification of the procedure.

Here we propose a technique which makes it possible to re-examine the choice points left in place. It consists, for each choice point, in restarting unification on the heads of as yet unresolved clauses. This method therefore performs a selection which takes into account *all* the arguments *and* the current constraints (disequations in the Prolog II+

compiler; disequations, inequations, list and boolean constraints in Prolog III), since their processing is included inside unification [Pique 90a]. Moreover, using this technique it is easy to also take into account the test primitives and the arithmetical primitives compiled "in line" and which are located at the start of the rule body.

From the point of view of garbage collection, dynamic revision enables a better result to be obtained than is possible using global compilation with complete indexation because the constraints of the current system are also taken into account.

The deletion of choice points is very useful because the number of living cells is decreased and the time required to perform garbage collection is therefore also decreased. It turns out that in a lot of cases dynamic revision plus garbage collection does not last as long as garbage collection alone. In addition, it should be noted that the duration of this operation will be partly deducted from future execution time (this is only true if no cuts are involved) since choice points can easily be updated to refer to the next effective alternative found by dynamic revision.

As a side effect the technique encourages declarative programming since it happens to cancel 'a posteriori' (i.e. when the garbage collector is activated) the space overhead which can occur with cut free programming. New Prolog programmers often leave a lot of choice-points around, and with this technique they will be able to run their programs anyway. An interesting feature that could be added to the system is the possibility to issue a warning whenever unnecessary choice points are found.

In constraint systems, the unused parts of words containing free variables are used to store information on the constraints of these variables [Van Caneghem 86]. A restoration stack (or trail) consisting of pairs (address of the cell to restore, value to re-establish) must therefore be used to be able to restore the constraints during backtracking. A trail with pairs is needed for the implementation of constraints like *dif* in Prolog II [Colmerauer 81] and for more general constraint solvers like Prolog III [Colmerauer 90], CHIP [Dincbas et al. 88], and CLP(R) [Jaffar 87]. In a Prolog without constraints, only the address of the cell to be restored has to be memorized in the trail, since the value to re-establish is always the same. Constraints can however be implemented using a standard trail and space in the copy stack [Le Huitouze and Ridoux 86, Carlsson 87].

The *systematic* use of pairs in the trail (which was not the case in the first Prolog II implementations) leads to more efficient code and allows easy implementation of several interesting techniques like variable time-stamping and dynamic revision of choice points. Dynamic revision is possible with a standard trail if space with a copy of the trail content is made available at garbage collection time, which is tantamount to create pairs with two tables.

2 Dynamic Revision

Dynamic choice points revision enables a certain number of choice points to be eliminated before activation of the Prolog garbage collector. This is done by attempting to unify each rule head which has not yet been resolved until either unification succeeds (the choice point then remains in place), or all the rules of the procedure have been exhausted (the choice point is then deleted). The difficult part is to be able to run alternatives left in the past without destroying the present state of computation.

The most suitable time to start dynamic revision is when the garbage collector is activated. To be able to activate and control alternative head unifications, it is necessary to differentiate the first real call (i.e. non expanded "in line") in a Prolog clause. It may also

be useful to be able to distinguish "in line" built-ins calls with and without side effects if such "in line" calls exist.

We therefore introduce two new instructions: *call_1, execute_1*. These instructions replace the instructions *call*, and *execute* when the latter are in the first call position:

```
L_a/0:                          a :- b, c, d.
   try_me_else L_a2
   allocate 0
   call_1  0, L_b/0, 0
   call    0, L_c/0, 0
   deallocate
   execute 0, L_d/0
L_a2:                        a :- b.
   trust_me_else fail
   execute_1 0, L_b/0
```

These instructions are also a good place for testing the advent of special events such as: coroutine activation, user interrupt, need to call the garbage collector, external interruption requiring asynchronous start of a Prolog program [Pique 90b], and so on. The *proceed* instruction is also extended in order to perform this test.

Compared to the implementation described by Carlsson [Carlsson 1987], this method performs the test earlier and in the environment of the rule concerned. However both methods are equivalent for the sake of dynamic revision.

2.1 Technique of choice point re-examination

For each choice point currently in place, we simulate a backtracking and a continuation of resolution on the possible altenatives. In order to be able to restore the current state of the bindings, we use the fact that a trail entry contains a field for the old value of the variable cell: the pseudo backtracking is performed by *exchanging* the value located in the trail with the value located in the stacks (environment stack and copy stack). We have thus re-established the content of the stacks before the call to the procedure, but *without having displaced the tops of the stacks*. The information concerning the state of the bindings of the current resolvent is therefore still present, but the bindings which are older than the considered choice point are now in the upper part of the trail.

As an example, Figure 1 represents a state at time t_n before the activation of dynamic revision. *x1, x2, x3* are free variables, *a1,a2,a3* their addresses, and *b1,...,b4* are bindings obtained in the following steps:

cur.choice time	x1	x2	x3
t_{n-2}	free	free	free
t_{n-1}	b1	b2	free
t_n	b4	b2	b3

One should notice that in a constraint system, there may be several trail entries for the same variable. For example the state of a variable may change from *free* to several *constrained* states and then to *bound*. The order in which the value of a cell is exchanged with the trailed value is therefore very important: from top to bottom when going in the past, and bottom to top when going back from the past to the present time.

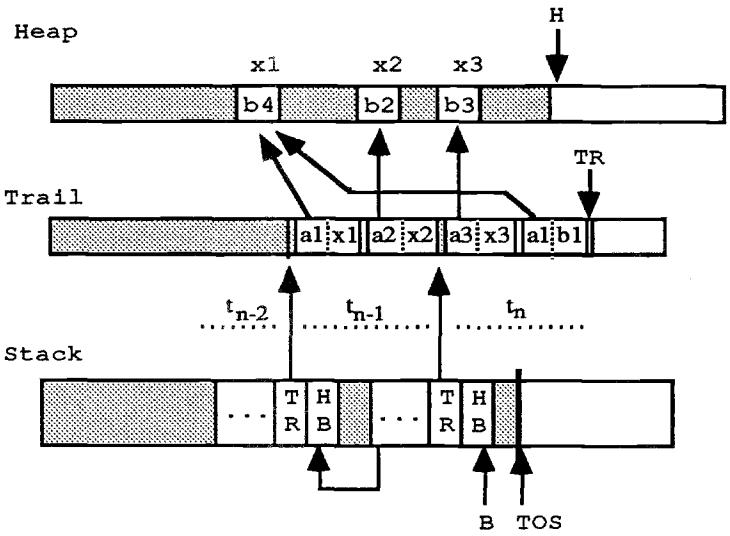


Fig. 1

Figure 2 represents the stack modification which is performed before the revision of the last choice point. This operation is equivalent to a backtracking as far as concerns the binding of variables. However as far as concerns the state of the tops of the stacks it differs: this state *must* not be modified.

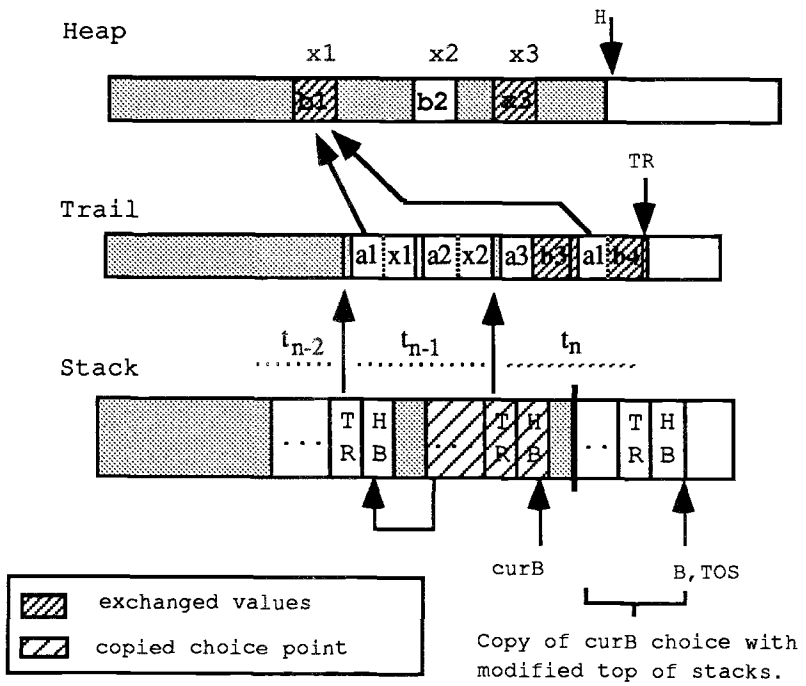


Fig. 2

When a choice point is reconsidered, the arguments of the corresponding goal are reconstituted from the information memorized in the choice point. The instruction pointer is positioned on the alternative memorized in the choice point, and execution runs until a rule head unifies or until all the rules in the procedure have been considered without success. *If there exists a rule head which unifies, the choice point is maintained, otherwise the choice point is deleted.*

This operation performs unification of the as yet unresolved rule heads. Consequently, the current values of the stack tops must first be memorized in the fields corresponding to the considered choice point, since the resolution may cause failures, and thus restoration of stack pointers. It is also essential that when the re-examination of choice points is activated there still remains enough space in the stacks to perform a resolution.

Figure 3 describes the state of the stacks before the revision of the second choice point:

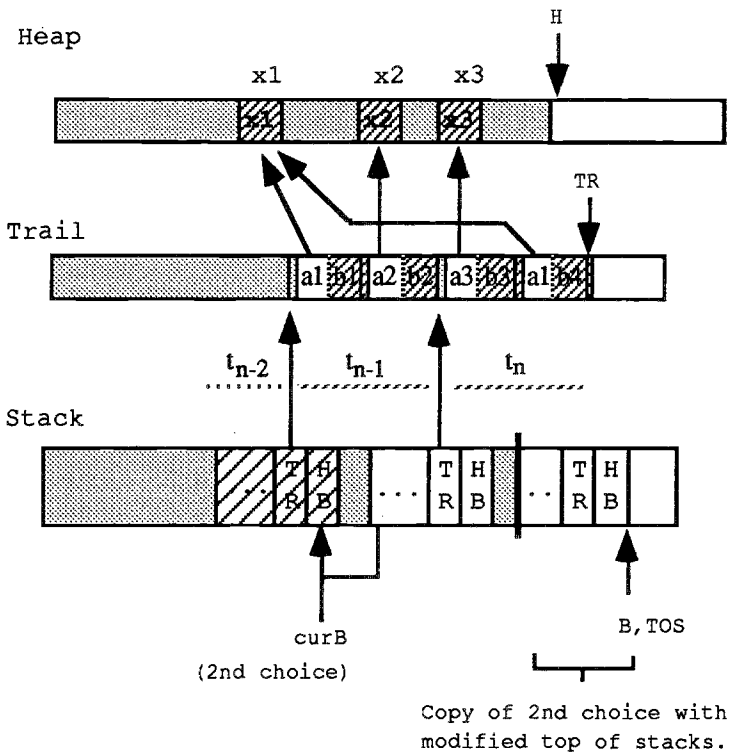


Fig. 3

During unification of the head, an environment may be created by the *allocate* instruction[1]. This creation must not occur in relation to the examined choice point, but in relation to the actual top of the stack. In most implementations this does not pose a problem since there is a top pointer for the environment stack (TOS), and environments and choice points are allocated relative to this pointer. However, one should consider the

---

[1] The heap may also be extended by execution of the unification instructions.

way the TOS value is computed in case of bactracking. If it is directly computed from the value of the B register (which is usual), the stack may be overwritten if an *allocate* instruction is encountered during the search for an alternative. Parameterizing the behavior of failure code is unacceptable, since the speed of Prolog depends so much on its efficiency. Worse, the compiler often performs some optimizations based on the hypothesis that the choice point and environment pointers have the same value when these two structures are created [Turk 86].

To escape these problems, the solution is *to copy the considered choice point frame on top of the stack* before the search for an alternative. This solution also has the advantage of being compatible with all means of implementing the *allocate* instruction. Only the copied choice point needs then to be modified to reflect the current tops of stack.

When a choice point is maintained, its fields must be restored, and the bindings eventually created by a head resolution must be removed. This is achieved by performing the equivalent of a backtracking on this choice point, and *then* by re-establishing the initial values of the fields.

2.2 Stopping execution, modified instructions

To stop execution at the relevant time, we need to modify the behavior of the following instructions[1] during dynamic revision:

- The instructions *call_1, execute_1,* or *proceed* stop execution and set the "choice point maintained" variable at true. Execution stops after unification of the rule head and processing of the builtins which are located at the start of the rule body and compiled "in line".
- The instructions *trust* and *trust_me_else* do not delete the choice point, but put the address of a *break* instruction in the field containing the address of the alternatives.
- The *break* instruction stops execution of one choice point revision and positions the "choice point maintained" variable at false.

The following additional modifications are not indispensable, but make it possible to benefit from the exploration of alternatives performed for reconsideration of the choice point:

- A *current_alternative* additional register is created for the sake of dynamic revision.
- The instructions *retry, retry_me_else, trust* and *trust_me_else* memorize the contents of field P of the choice-point (i.e. the address of the instruction) in the *current_alternative* register, before changing the contents of the field.
- If the choice point is maintained, the content of the P field is replaced by the value of the *current_alternative* register.

---

[1] It is easy to modify the semantics of a virtual instruction in an emulator, without any cost. However, in case of direct code generation, a test for modified semantics should be added (most current CISC implementation use an emulator).

2.3 Impact on execution times

It should be noted that the duration of this task will be partly deducted from future execution time (this is only true if no *cut* is involved). In addition, this operation can reduce substantially the task of the garbage collector, since the number of structures to mark and shift is reduced. This choice point deletion phase enables the following:

- Recovery of space in the global stacks by making certain structures inaccessible.
- Recovery of space in the local stack when the deleted choice point is located at the top of the stack (B > E) or if the environments are garbage collected.

2.4 Algorithm

In the following, we list the main algorithm in a Pascal like language.TOS denotes a register containing the address of the top of the environment and choice point stack. The main routine *choice_revision* is called before activation of the garbage collector.

```
VAR
   B : ^ ChoicePoint;                    { Current choice point pointer }
   E : ^ Environment;                    { Current environment pointer  }
   TOS : union ( ^ ChoicePoint; ^ Environment );     { Top of stack }
   ...
   top_B   : ^ ChoicePoint;
   top_H   : ^ PrologCell;
   top_TR  : ^ TrailPair;
   top_TOS : union ( ^ ChoicePoint; ^ Environment );

PROCEDURE choice_revision;
   VAR    cur_B, up_B : ^ choicePoint;
      TR1, TR2 : ^ TrailPair;
   BEGIN
   save_all_WAM_registers(P,CP,N,B,E,H,HB,TR);
   save_Ai_registers(N);
   top_TR := TR;
   top_H  := H;
   top_TOS:= TOS;
   top_B  := B;

   set_mode_stop_at_call_or_break;
   cur_B := top_B;
   up_B  := top_B;
   TR2   := top_TR;
   WHILE cur_B <> bottom_stack DO
      BEGIN
      TR1 := cur_B^.TR;
      exchange_trail_downwards(TR1,TR2);
      IF choice_maintained(cur_B) THEN
         up_B := cur_B;
      ELSE IF  cur_B = top_B THEN              { Discard top choice point }
         BEGIN up_B := cur_B^.B;  top_B := cur_B^.B END
      ELSE                                        { Discard from chain }
         up_B^.B := cur_B^.B;
      cur_B := cur_B^.B;
      TR2   := TR1;
      END;
   exchange_trail_upwards(bottom_trail,top_TR);       { restoration }
   TR := top_TR;
```

```
   H  := top_H;
   restore_all_WAM_registers;
   restore_Ai_registers(N);
   B  := top_B;                    { The last maintained in place }
   HB := top_B^.H;
   IF E < B THEN
      TOS := B;
   ELSE
      TOS := E + Environment_size(E,P,CP);
   unset_mode_stop_at_call_or_break;
   END;

                                   { Examines the choice point cur_B }
FUNCTION choice_maintained( cur_B : ^ choicePoint ) : boolean;
   BEGIN
   restore_Ai_from_choice_point(cur_B);
                    { Defining the state of the wam for this revision }
   TOS := top_TOS;
   TR  := top_TR;
   H   := top_H;
   HB  := cur_B^.H;
   P   := cur_B^.P;
         { copy of choice point to allow a safe local backtracking }
   B := copy_choice_to_top(cur_B);
   B^.TR := top_TR;
   B^.H  := top_H;
   RUN_PROLOG();                      { Try to find an alternative.}
   IF NOT stop_at_break THEN         { There exists one alternative: }
      BEGIN                          { discard unification bindings. }
      restore_trail(top_TR,TR);
      cur_B^.P := current_alternative;        { Skip failing rules }
      choice_maintained := TRUE;              { Keep choice }
      END
   ELSE            { No alternative resolves: discard choice point }
                   { fail has already made the unbindings }
      choice_maintained := FALSE;
   END;
```

The procedure *exchange_trail* exchange in a given trail segment the trail values with the corresponding heap cells.

The procedure *set_mode_stop_at_call_or_break* changes the behavior of the wam instructions as described above:

```
PROCEDURE set_mode_stop_at_call_or_break;
   BEGIN
   - Retry and retry_me_else: same as usual + memorize B^.P in
     register "current_alternative".
   - Trust and trust_me_else set B^.P at break address, and
     do the same as above.
   - Call_1, execute_1, and proceed stop the machine
     with "stop_at_break" false.
   - Break stops the machine with "stop_at_break" true.
   END;
```

2.5 Examples

The first test program is a tail recursive procedure extracting the positive values of a given list. The program is run on a list of N values with alternate positive and negative values

and is created in a deterministic way (for N=5, L=[5,-4,3,-2,1,0]). The *positive* procedure will then create a choice point one time out of two.

```
ptest(N,L1) :- alternate(N,L), positive(L,L1).

positive( [], [] ).
positive( [X|L], [X|L1] ) :- X >= 0, positive( L, L1 ).
positive( [X|L], L1 )     :- X <  0, positive( L, L1 ).
```

The following measurements show garbage collection results when the program is run with a copy stack of 30 Kbytes (the implementation uses 8 bytes for one Prolog cell and 32 +8*(nb of arg) bytes for a choice point). *gc1..gcn* denote activations of the garbage collector. A sequence of numbers in the table should be interpreted in the following order (case 3 and 4 only apply in case of dynamic revision):

1. Number of cells collected in the copy stack.
2. Number of cells collected in the trail stack.
3. Number of choice points killed during dynamic revision (in parenthesis).
4. Tos address and Tos reduction after dynamic revision (in bytes).

```
?- ptest(1500,L).
```

Table 1.

	Standard wam	With dynamic revision
	Copy, Trail	Copy, Trail, Bkill, Tos
gc1	0,0 -> abort	1040, 259, (259), 888044 (-12432)
		solution

If we consider the *positive* program, we notice that it is running in mode (+,-). Therefore, each time the second clause is executed, a choice point is created, a value is pushed on the trail, and an element of the initial list is included in the query list. This explains the observed results (table 2): there are as many trail entries collected as choice points deleted. Moreover when the program has progressed by n elements in the list, $n/2$ cells of the copy stack can be collected. This is why more and more choices are created on the following steps by the program, since there is more space to work after each garbage collection. It is important to note that it would not have been the case if the initial list were part of the query.

```
?- ptest(1800,L).
```

Table 2.

	Standard wam	With dynamic revision
	Copy, Trail	Copy, Trail, Bkill, Tos
gc1	0,0 -> abort	136, 34, (34), 1632
gc2	-	272, 68, (68), 3264
gc3	-	544, 136, (136), 6528
gc4	-	1088, 272, (272), 13056
	-	solution

Let us now consider the program *qsort* written in pure Prolog. This program will be able to recover space in all stacks with dynamic revision:

```
qsort(L0,L1) :- qsort(L0, L1,[]).

qsort([],L,L).
qsort([X|L0],R0,R2) :-
   partition(X,L0,L1,L2),
```

```
    qsort(L1,R0,[X|R1]),
    qsort(L2,R1,R2).

partition(_,[],[],[]).
partition(X, [Y|L], [Y|L1],    L2 ) :-
    X >= Y,
    partition(X,L,L1,L2).
partition(X, [Y|L],    L1 , [Y|L2]) :-
    X < Y,
    partition(X,L,L1,L2).
```

The second clause of *partition* is compiled in Prolog II in the code:

```
L_partition2:
    retry_me_else L_partition3
    get_list A2
    unify_variable A5
    unify_variable A2
    get_list A3
    unify_value A5
    unify_variable A3
    compare A1 ge A5
    execute_1 3, L_partition/4
```

Here we should note that the "in line" expansion of the predefined rule '>=' means we do not need the *allocate* instruction in *partition/4* and only have to use temporary variables. If the garbage collector is activated, all the choice points are discarded by dynamic revision, and a substantial amount of space is recovered both in the copy stack, the trail, and the stack of environments and choice points.

```
qtest(N,L1) :- alist(N,L), qsort(L,L1).
```

The following test is carried out on a list having the form [n, (n-1)/2, n-2, (n-3)/2, n-4, ...] which will create choice points on half the number of calls to the *partition* procedure. The test is launched with a copy stack of 30Kb, and an environment stack of 1Mb to cope with deep left recursion in this program. The fields have the same meaning as above (Bkill column is therefore the number of choice points deleted by dynamic revision at each gc activation).

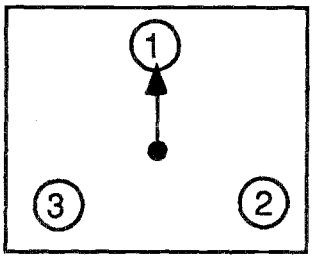
```
?- qtest(200,L).
```

Table 3.

	Standard wam	Wam + dynamic revision
	Copy, Trail	Copy, Trail, Bkill, Tos
gc1	0, 0 -> ovf	3272, 1589, (1506), 984000 (-1088)
gc2	-	3272, 1648, (1630), 1081552 (-7488)
gc3	-	3272, 1652, (1628), 1184080 (-2624)
gc4	-	3272, 1657, (1626), 1288572 (- 832)
gc5	-	3272, 1830, (1615), 1396668 (-1024)
		solution

One can notice the small reduction of Tos (for example 1088 bytes for 1506 choice points deleted corresponding to 1506*(32+4*8)=96,384 bytes). This comes from the fact that most times, in standard Wam, there are very few holes in the environment stack and therefore it is not garbage collected in present implementations (holes can be caused by environment trimming and deep cuts, followed by the creation of a choice point before deallocation of the cut environment). In contrast to standard wam, dynamic revision creates a lot of holes which are not recovered, meaning that it would be worthwhile to also garbage collect the environment stack.

The third example is a program written with the *dif* constraint. By means of a relation which gives the possible successor of each state, this program defines the list of possible transitions from one given state, such that return to the directly previous state never occurs (meaning that whenever the switch cycles in one direction, it must keep that direction). In the given query, all the choice points can be discarded, except the one created for the first call on the ***transition*** clause:




```
transition(1,3).
transition(1,2).
transition(2,1).
transition(2,3).
transition(3,1).
transition(3,2).

node_list(0, []).
node_list(1, [Current]).
node_list(2, [Previous,Current]) :-
   transition(Previous, Current).
node_list(N, [Previous,Current,Next|L]) :-
   N > 2,
   transition(Previous, Current),
   dif(Previous,Next),      /* install constraint */
   N1 is N-1,
   node_list(N1, [Current,Next|L]).
```

The first example is launched with a copy stack of 60 Kb. As in the previous example, the top environment is responsible for the small reduction of the Tos during dynamic revision. However it holds roughly the same value when garbage collection is activated, meaning that part of the killed choice point space may be re-used when the environment is deallocated. Anyway the space collected in the two other stacks is very substantial with dynamic revision.

```
?- nodelist(2000,[1|L]).
```

Table 4.

	Standard wam	Wam + dynamic revision
	Copy, Trail	Copy, Trail, Bkill, Tos
gc1	3972, 248	6177, 1571, (441), 963016 (- 0)
gc2	2044, 120	5054, 1311, (375), 1000408 (-56)
gc3	1050, 62	4135, 1070, (306), 949516 (- 0)
gc4	544, 32	first solution
gc5	281, 17	
gc6	144, 8	
gc7	72, 4	
gc8	43, 3	
...	...	
gc12	overflow	

In the following measurement, the *transition* procedure has been modified in such a way that choice points remain for transition 3: a dead end transition to 5 is added so that the goal *dif(1,X), transition(3,X)* now has two solutions.

```
transition(1,3).
transition(1,2).
transition(2,1).
transition(2,3).
transition(3,1).
transition(3,2).
transition(3,5).
```

Here are the results obtained in this case:

Table 5.

	Standard wam	With dynamic revision
	Copy, Trail	Copy, Trail, Bkill, Tos
gc1	2867, 27	5072, 1350, (441), 985116 (- 0)
gc2	1040, 0	3379, 921, (307), 1031216 (- 0)
gc3	380, 0	2253, 615, (205), 1061916 (- 0)
gc4	136, 0	1505, 411, (137), 1082416 (- 0)
gc5		

2.6 Some time measurements

We will compare the cost of garbage collection alone and dynamic revision plus garbage collection in two extreme programs running with a 60 Kb copy stack:

Table 6.

	Nodelist 1000	Compile
gc alone	109	9
revision + gc	18 + 20	2 + 9
gc/(rev+gc)	2.87	0.82

The first one is the *nodelist* program. The goal *nodelist(1000,[1|L])* leads to a solution after one call to the garbage collector in either case. With the dynamic revision algorithm, all choice points but one are deleted before garbage collection is applied.

The second one is the compilation of a big term by the Prolog II compiler, which also leads to a solution after one call to the garbage collector in either case. In this example, no choice point can be recovered by the dynamic revision algorithm.

3 Conclusion

We have described a procedure for dynamic revision of choice points during garbage collection. This procedure allows more space to be recovered in all the wam stacks i.e. the copy stack, the trail stack and the environment and choice point stack. This is done by performing an anticipated shallow backtracking on all current choice points. It turns out that the time spent in the revision procedure is often returned with interest by the garbage collector.

4 Bibliography

[Bekkers et al. 86]
Bekkers Y., Canet B., Ridoux O., Ungaro L., 1986, A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages, *Proc. of the third Symposium on Logic Programming Conference*, IEEE.

[Carlsson 87]
Carlsson M., 1987, "Freeze, Indexing, and Other Implementation Issues in the WAM", *Proc. of the Fourth International Conference on Logic Programming*, MIT Press, Lassez J.L. ed..

[Colmerauer 82b]
Colmerauer A., 1982, Prolog, Bases théoriques et développements actuels, dans *TSI*, vol. 2, n°4 (AFCET-Bordas), août 1983, avec H. Kanoui et M. Van Caneghem.

[Colmerauer 90]
Colmerauer A., 1990, An Introduction to Prolog III, *Communications of the ACM*, july 90, vol 33, n 7, pp 70-90.

[Dincbas et al. 88]
Dincbas M., Van Hentenrick P., Simonis H., Aggoun A., Graf T. and Berthier F., 1988, The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, vol 1, Ohmsha Pubishers, Tokyo, pp 693-702.

[Jaffar 87]
Jaffar J. and Michaylov S., 1987, Methodology and Implementation of a CLP System, *Proc. 4th International Conference on Logic Programming* (Melbourne 87), MIT Press, Cambridge, Mass., pp 196-218.

[Hickey et Mudambi 89]
Hickey T., Mudambi S., 1987, Global Compilation of Prolog, *Journal of Logic Programming*, vol 7, N 3, nov 1989, pp 193-230..

[Le Huitouze et Ridoux 86]
Le Huitouze S., Ridoux O., 1986, Une expérience de réalisation du Gel et du Dif dans MALI, In *Actes du Séminaire 1986 - Programmation en Logique*, Trégastel, France, CNET, Mai 1986.

[Morris 78]
Morris F.L., 1978, A Time and Space Efficient Garbage Compaction Algorithm, *Communications of the ACM*, Vol 21, No 8.

[Naish 85]
Naish L., 1985, *Negation and Control in Prolog*, Ph. D. Thesis, Dept of Computer Science, University of Melbourne.

[Pique 90a]
Pique J.F., 1990, Compilation d'un prolog II modulaire, *Habilitation à diriger des recherches*, Université Aix-Marseille II, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy.

[Pique 90b]
Pique J.F., 1990, Definition of a Prolog Machine with Interrupts, *New Computing Techniques in Physics Research*, D. Perret-Gallix & W. Wojcik ed., Editions du CNRS, Paris 1990, pp 369-379.

[Touraïvane 88]
Touraïvane, 1988, Récupération de mémoire dans Prolog III, *Thèse de Doctorat en Informatique et Mathématiques*, Université d'Aix-Marseille II, Faculté des sciences de Luminy.

[Turk 86]
Turk A.K., 1986, Compiler Optimizations for the Wam, *Proceedings of the Third International Conference on Logic Programming*, London, Springer-Verlag, Shapiro E. ed., pp 657-662.

[Van Caneghem 86]
Van Caneghem M., 1986, *L'anatomie de Prolog II*, InterEditions, Paris.

[Warren 77]
Warren D.H.D., 1977, *Implementing Prolog: Compiling Predicate Logic Programs*, Dept. of A.I. Research Reports 39 & 40, Edinburgh, May 77.

[Warren 83]
Warren D.H.D., 1983, *An Abstract Prolog Instruction Set*, Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, Calif.

Ecological Memory Management in a Continuation Passing Prolog Engine

Paul Tarau

Université de Moncton,
Moncton N.B., Canada, E1A 3E9,
Email: tarau@info.umoncton.ca

Abstract. Starting from a simple 'ecological' metaphor, we introduce a new memory management scheme (*heap-lifting*) implemented in BinProlog, a continuation passing style variant of WAM. We discuss copying garbage collection mechanisms based on *heap-lifting* and an OR-parallel execution model. We point out some surprising similarities with related work on functional languages and the difficulties that arise in the context of nondeterministic execution. Finally, we describe the full implementation of two builtins: a recursive `copy_term` and a very fast heap-lifting based `findall` and we evaluate their impact on the performances of BinProlog.
Keywords: *WAM, Prolog run time system, continuation passing style compilation of Prolog, fast builtins for Prolog, copying garbage collection.*

1 Introduction

We suppose the reader is familiar with the WAM (see [14]) and at least one of its runtime incarnations. Our BinProlog engine[1] is a variant of the WAM, specialized for efficient execution of *binary programs*[2]. Binarization by continuation passing introduced in [12] is the logic programming equivalent of CPS compilation for functional languages (see [3] and [4]). We use it as a preprocessing step, working on a clause by clause basis. A clause like `c(A) :- a(A),e(A,B),b(B)` becomes `c(A,Cont) :- a(A,e(A,B,b(B,Cont)))` where `Cont` is a new variable representing the *continuation* that is recursively passed between calls. Efficient WAM-support deals with metavariables resulting from the transformation in the case of unit clauses (for example the clause `a(X)` becomes `a(X,Cont):-Cont`). We refer the reader to [11] and [10] for a description of our compiler and our abstract machine.

The result of binarization is that we give up WAM's environments (the AND-stack) and we put on the heap the *continuation*, recursively embedded in the last arguments of our binary programs. As a consequence, the heap consumption of the program goes up, although in some special cases, partial evaluation at source level can deal with the problem (see [8]), showing that a heap-only approach is not necessarily worse.

Simplicity of implementation and a small and clean run-time system have to compensate for the more intensive heap-consumption, to be competitive in absolute terms with well-engineered standard WAM implementations.

[1] available by ftp from 139.103.16.2, 215KLIPS on Sparcstation 2

[2] programs with clauses having only one literal in the body

The high heap consumption was the starting point of our optimization effort. The general view is that the WAM is a very space efficient engine. To raise reasonable doubt about that, we suggest to compare the space consumption of the WAM on the naive reverse benchmark, $O(N^2)$, with the size of useful data (the reversed list) that is produced, $O(N)$. One can argue that naive reverse is not a well-written Prolog program and after all the WAM is much more efficient than, for instance, engines without last call optimization or interpreters. For most of the programs, however, when compared with their theoretical lower limit (i.e the size of the *computed answer*) one must agree that the space complexity of WAM computations is almost always higher. Obviously, an easy way to restore the equilibrium at some stage is to copy the (possibly partially instantiated) answers and discard the space used for computations.

Traditionally, Prolog garbage collectors are mark-and-sweep because they want to preserve chronological order of heap and stack segments as required for backtracking, although it is well known that their performance is proportional to the total size of the heap instead of the size of useful data, as is the case with copying algorithms. Worst, this useful data is rather sparse in typical Prolog applications. In the case of BinProlog, the extra heap consumption of binary programs is one more reason to pay attention to *copying techniques*[3].

2 Towards an ecological Prolog engine

An ideal memory manager is *'ecological'*. We want it to have a *'self-purifying'* engine that recuperates space not as a deliberate *'garbage-collection'* operation but as a natural *'way of life'* i.e something done inside the normal, useful activities the engine performs.

2.1 The rain-forest metaphor

In a rain-forest, a natural garbage-collection process takes place. Evaporated water originating from the forest form clouds, and then condensed water falls back as rain[4]. Some heat (a Sun) is also needed to make things work.

In a Prolog engine terms are created on the heap. Although WAM's environment stack does some limited and well intentioned garbage *prevention* (i.e. environment trimming), often garbage *collection* is needed for large practical programs. The possibility of garbage prevention is reduced even more in BinProlog where continuations go also on the heap.

Following the rain-forest metaphor, our objective is to set up a natural and automatic memory recuperation cycle. Basically the heap is split in a small lower half (the *rain forest*) and a large upper half (the *clouds*). The key idea is to fool Prolog's execution mechanism to work temporarily in the upper half (*evaporation*) and then let useful data 'fall back' in the lower half of the heap (*rain*) in a condensed

[3] GC timings differ a lot for few and much garbage. However, by keeping the relative size of useful data small with respect to the available memory, copying GC can be made fairly efficient, as shown in [4].

[4] well, in Canada it is mostly *snow*, but that's only an implementation detail

form (compaction by copying). The trick is very simple: as structure creation on the heap is always done around the H pointer while everything else stays in BinProlog's registers, all we have to do is temporarily set H to a location at the beginning of the upper half of the heap and then let the engine work as usual. The figure 1 shows this *heap lifting* technique and the position of the H pointer at various stages.

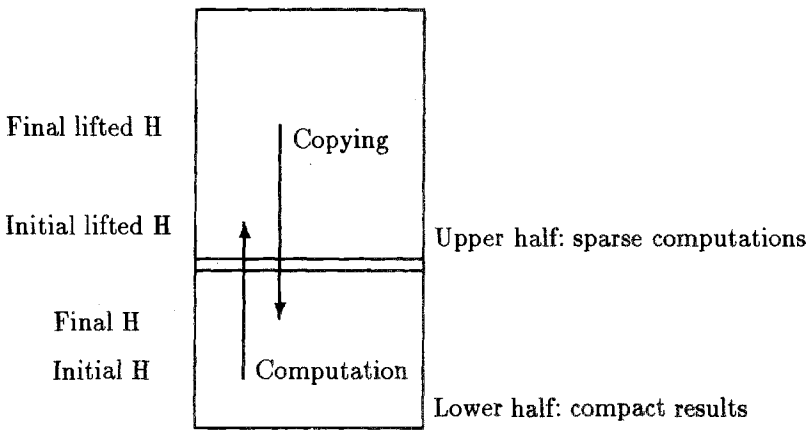


Fig. 1. Heap-lifting

The metaphor ends when applied to *recursive* uses of the mechanism, but the concrete implementation deals properly with this problem by saving the necessary information on a stack and by ensuring a reasonable number of embedded heap-splits.

3 Fast heap-based builtins

Let us start with a detailed description of a heap-based implementation for two very useful Prolog primitives that will give an insight on how the principle can be used in practice. They also have the potential to significantly speed-up the overall performances of Prolog systems that decide to convert from classical assert-based implementations to a heap-based technique.

3.1 Copy_term

Our copy_term primitive is implemented recursively. Variable cells in the source-term are forced to point *upward* to their new copies, working as temporary value-dictionaries. This ensures that variables seen as *equivalence classes* are transferred to the copy. As they are trailed in this process, we simply unwind the trail at the end and free them from their temporary values.

The algorithm is very compact. The C-code is given in appendix A.

There's an interesting similarity between this copying algorithm and the *forwarding* technique used in the garbage collector of Appel [3]. The New-Jersey SML

garbage collector implements Cheney's [7] elegant, non-recursive algorithm to move terms from a *fromspace* to a *tospace*. The basic idea is to use forwarded terms as a breadth-first queue for the copying process. One can be attempted to adopt it for copy_term in Prolog and get rid of recursion that has (in the worst case) a depth proportional to the size of the heap. Actually a variant of this algorithm is used in [5] for a parallel memory manager.

However, recursive calls can be more efficient on the average, at least on a Sparcstation as they will span over register windows, while the non recursive algorithm will do memory accesses. On the other hand, if the term is deeper than the number of available register windows we pay back those gains, at least in part. The Fujitsu chip in the Sparcstation 1 implements 7 register windows. When a call is made, 8 output registers of the caller simply overlap 8 input registers of the callee. However, as far as recursion is bounded, one has to keep in mind that if a non-recursive procedure works on at most 24 registers, a recursive one using all the register windows can span its work over 120 registers. Notice that the number of register windows is expected to grow on future Sparc implementations. Although Cheney's algorithm is obviously more space efficient and also preserves the shared nature of subterms, some empirical study on the tradeoffs involved in typical uses of copy_term and primitives that rely on it (like findall) is needed.

3.2 Findall

We implemented both findall/3 and findall/4. The latter, (suggested by a public domain version of R.A. O'Keefe) appends the answers to an existing list. We used our fast copy_term to do it. Basically we execute the goal in the upper half[5]. Findall is written as a failure driven loop in Prolog, using 3 builtins implemented in C.

```
findall_workhorse(X,G,_):-
  lift_heap,
  G,
  findall_store_heap(X).
findall_workhorse(_,_,Xs):-
  findall_load_heap(Xs).

findall(X,G,Xs,End):-findall_workhorse(X,G,[End|Xs]).

findall(X,G,Xs):-findall_workhorse(X,G,[[]|Xs]).
```

The builtin *lift_heap* simply cuts the heap in half and temporarily assigns to H the value of the middle of the heap. The previous value of H and the previous HEAP_MARGIN used for overflow check are stacked. This allows for embedded calls to *findall* to work properly.

Then the goal G is executed in the upper half of the heap.

[5] Actually the ratio between the lower and the upper half is 1:4, 1:8 or even a higher power of 2. This must be in principle the statistically expected ratio between the size of the answer and the size of the computation that produces it.

The builtin *findall_store_heap* pushes a copy of the answer X, made by *copy_term* to a an open ended list located in the lower half of the heap and starting from the initial position of H, that grows with each new answer. Then it forces backtracking.

The failure driven prolog-loop ensures that in the case of a finite generation of answers, they can all be collected by the builtin *findall_load_heap* starting from the initial position of H and forming an open ended list. The space used is exactly the sum of the sizes of the answers plus the size of the list cells used as glue. The builtin *findall_load_heap* puts on the heap and returns a cons-cell containing the end of the list and the list itself. It also removes from the stack H and HEAP_MARGIN, used inside copy_term to check heap overflow. For findall/4 we return also the end of the open ended list of answers, while for for findall/3 we close it with an empty list. The total size of the Prolog and C code is about half of the usual assert based (correct) implementation of findall.

The C-code of the 3 builtins is given in Appendix B. They are part of the main switch of our WAM-loop. The implementation can be further accelerated by moving it completely to C, but we left it in Prolog to allow tracing and to be able to use the same builtins for other primitives.

Remark that at the end, all the upper half of the heap is freed. This means that an amount of space proportional to the size of the computation is freed within time proportional to the size of the answer.

One reason why findall is so simple and fast is that we have no constraints on the relative position of the heap and the OR-stack. But what to do in the case of general WAM? When executing in the upper half, we suppose that new objects are always higher than old objects - impossible, as the AND stack is even higher! However, by modifying the run-time checks to detect where a variable is located, one can adapt this algorithm to standard WAM, with some additional cost. A better solution is to allocate a temporary space higher than the AND-stack and execute there, successively each recursive call to *findall*. This implies however copying twice. We think that we have here a a good example of implementation simplicity that is lost in the case of standard WAM.

3.3 Performance evaluation

The table 1 compares the performances of our compiler with C-emulated and native code Sicstus Prolog 2.1, all running on a Sparcstation IPC. The Sicstus compiler is bootstrapped so that builtins are compiled to native code (the fastest possible configuration). The reader can get the benchmarks by ftp from the BinProlog distribution (139.103.16.2). PERMS(8) is a nondeterministic permutation generator, DET-ALLPERMS(8) is a deterministic all permutation program written in pure Prolog, FINDALL-PERMS(8) is a findall-based all-permutations program and BFIRST-META is a breadth-first prolog meta-interpreter. The 3 permutation benchmarks give an idea about the overhead of accumulating solutions (DET-ALLPERMS) and the overhead of findall in both emulated and native Sicstus and emulated BinProlog. As one can see on the PERMS(8) and DET-ALLPERMS(8) benchmarks BinProlog and emulated Sicstus are quite close on pure Prolog programs. Therefore, the difference between DET-ALLPERMS(8) and FINDALL-PERMS(8) comes from the implementation of findall. Together with the BFIRST-META benchmark, which also

uses findall, this shows how efficient our *heap-lifting* technique can be, when compared with a fairly optimized assert-based approach.

Benchmark program	Sicstus 2.1 Emulated	BIN-Prolog Emulated	Sicstus 2.1 Native
PERMS(8)	1.350 sec	1.200 sec	0.639 sec
DET-ALLPERMS(8)	2.699 sec	3.040 sec	0.820 sec
FINDALL-PERMS(8)	13.089 sec	6.060 sec	11.219 sec
BFIRST-META	1.230 sec	0.420 sec	1.070 sec

Table 1. Performances

Timing is given by the `statistics(runtime,_)` predicate i.e. without garbage collection or system time. However, on a 8 Megabytes Sparcstation IPC with 32 Megabytes of swapping space and OpenWindows, BinProlog took 44 seconds of real time (as given by the *rusage* Unix primitive) to perform all the permutation benchmarks, while Sicstus took 8 minutes (native) and more than 10 minutes (emulated) because of the overall complexity of its memory management. Quintus Prolog had a similar behaviour on a 16 Megabytes sun4 machine due to a huge number of stack shifts (more than 2 minutes). As these figures indicate, keeping memory management as simple and predictable as possible pays off.

4 Copying GC in the presence of backtracking

4.1 Related work on functional languages

In a heap-intensive engine like BinProlog, as it is also the case with modern ML-engines (see [3]), mark-and-sweep garbage collection with time complexity proportional to the size of the heap is too expensive. This suggest a memory management scheme for BinProlog that is based on a copying garbage collector. It is inspired by the idea of *resource-driven failure* (RDF), described in [11], where the reader can also find a working prototype in Prolog. RDF is triggered when some resource limitation (stacks, CPU-time, etc.) occurs. Basically, we save (a compact copy) of the current resolvent and its partial answer substitution, we force backtracking and we work temporarily on another branch of the OR-tree. Later, when resources become available we restart the computation from the saved copy.

As we pointed out before, binarization based compilation of logic programming languages is very similar to the Continuation Passing Style compilation of functional programs described in [4] and [3]. Although our design presented in [11] and [10] was developed independently based on a very natural program transformation, it is worthwhile to stretch some similarities that are consequences of the same continuation passing approach. The key idea of [3] is that heap based allocation and deallocation by copying garbage-collection has a better amortized cost than stack based allocation and deallocation, especially when a generational collector is used.

4.2 What's different with Prolog

This is also (almost) true in the case of binary logic programs. As far as there's no backtracking involved, Appel's technique can be used to partially garbage-collect in a very efficient way a deterministic, deep branch of a WAM-implemented SLD-tree. However, in the presence of choice points, either we preserve chronological ordering with a traditional mark-and-sweep garbage collection, or we ensure a form of OR-parallel evaluation that has the same operational semantics as standard Prolog execution. Let us describe this second alternative in more precise terms.

When evaluating binary definite programs using SLD-resolution, each step can be seen as an unfolding step, starting from a clause of the form $G \leftarrow G$, where G is the original (atomic) goal of the program. Instead of the usual resolvent we are dealing with a *conditional answer* of the form $(G \leftarrow B)\theta$, where G is the original goal, B is the body of the last clause used in the resolution process and θ is the composition of the substitutions used in resolution steps so far.

Whether resolution is terminated by a unit clause or it is in progress, $(G \leftarrow B)\theta$ is a logical consequence of the program that keeps all the information computed so far by the SLD-derivation. In particular, it is possible to restart the derivation at any time from (a compacted copy) of $(G \leftarrow B)\theta$.

In terms of the engine, this means starting a new Prolog process with a copy of the current resolvent[6], while the parent searches for another answer, efficiently recovering its space by backtracking. More generally, overflow of other resources (CPU-time, stack, trail) can trigger a similar action.

A possible implementation, given BinProlog's very small code overhead is to create a new UNIX process. The presence of portable lightweight processes and parallel forthcoming Unix kernels make this choice very appealing. Another possibility is to implement a process queue inside BinProlog. This allows to be as lazy as possible about allocating the new data areas Heap, Trail and OR-stack while the standard Unix-based technique would start from a full copy. This choice has also the advantage to be portable to non-Unix machines.

By giving high priority or even preference to the parent we obtain a *fair* OR-parallel evaluation mechanism[7] similar to iterative-deepening but more efficient as no repeated computation is involved. This makes very attempting an uncompromising approach that sacrifices Prolog execution order. Under this OR-parallel execution model, each process efficiently recovers space by backtracking within fixed data areas, while spanning a new process that continues the aborted computation starting with minimal resources.

A possible scheduling mechanism, suitable for machines with relatively few but very fast CPUs (like the multiprocessor versions of the Sparcstation 10) is based on the following principle:

> *Processes that will free a resource must have higher priority than those which ask for a resource.*

[6] that happens to be simply the instantiated body of the last matching binary clause, accessible from the argument registers before the next EXECUTE instruction

[7] fairness is implemented by ensuring higher or exclusive priority for the parents

As a consequence, the backtracking parent who's memory needs are diminishing has priority over the forward executing child (who will soon ask for more as it comes often from a memory consuming branch of the search tree that still may grow). Hence work in an almost deterministic parent will die off very quickly so that its resources can be reused. Garbage collected children will wait in small heaps with empty trail and empty OR-stack until they are scheduled. Then data areas can grow dynamically.

An interesting reference point is the Muse model described in [1] and [2]. The main similarities are multiple parallel engines and the presence of a form of differential heap-copying. The main differences relative to Muse in our proposal are:

- process creation is only triggered by need i.e by a resource limitation
- a fair evaluation mechanism replaces Prolog's depth-first evaluation order
- efficient garbage collection is for free

In our case, OR-parallelism comes as a byproduct of resource management. The intended architecture for our proposal is basically a Sparc-like RISC with one or a few CPUs, as we do not necessarily want to maximise OR-parallelism, unless we are constrained to do so by resource limitations. On the other hand, in Muse, as in the Aurora model (see [6]), the main goal is to extract as much OR-parallelism as possible, the intended architecture being a shared memory switch-based multiprocessor.

4.3 Preserving Prolog execution order

Let us see what's happening if we want to preserve Prolog execution order. One problem is CUT. To solve it we have to suspend the parent until any child containing a CUT that can affect the parent finishes. The same principle can be applied to deal with I/O and side-effects. It leads to standard Prolog execution order. Similar techniques have been used in the OR-parallel engines of [6], [1] and [2].

Let us see what happens if we want to use *resource-driven failure* as a garbage collector for a *sequential* Prolog engine. The simplest is to execute the child process in the upper half of the heap, as in our implementation of findall. The same heap-splitting technique can be applied to deal with smaller and smaller recursive activations. The neat effect is that of a *copying, multi-generational* garbage collector.

By taking a closer look to possible implementations of an RDF based garbage collector, the following possibilities arise:

- a programmer controlled RDF
- RDF triggered on predefined heap marks
- RDF by compiler generated annotations

In the case of a programmer controlled approach, a simple predicate

```
gc_call(Goal):-
     findall(Goal,Goal,Instances),
     member(Goal,Instances).
```

found in the Craft of Prolog (see [9]), p.85) combined with our fast heap-based findall can be used. We actually annotated in a few minutes our memory-hungry real-time automatic Tetris player to port it to BinProlog. The program uses an energy minimization algorithm to find a best fit for a falling bloc of an irregular shape to an irregular ground made up of previously fallen blocs. Before `gc_call` the program run out of heap in less than a second. After annotation with `gc_call` the program was running for hours[8]. Execution on Prologs with mark-and-sweep garbage collectors was quite unpleasant as the real-time effect of the falling blocs was lost due to unexpected interruptions.

A more automatic approach is to use heap-marks. When the H pointer reaches such a limit, the next EXECUTE instruction can wrap the current goal G inside a `gc_call`, executing `gc_call(G)` instead of G. As heap margin is checked anyway by our EXECUTE instruction there's no additional cost involved. As continuations are first-order objects in BinProlog, we can do this work only on the continuation (actually an argument of the goal). The figure 2 shows a Prolog search tree with gc_call margins and nodes subject to gc_call.

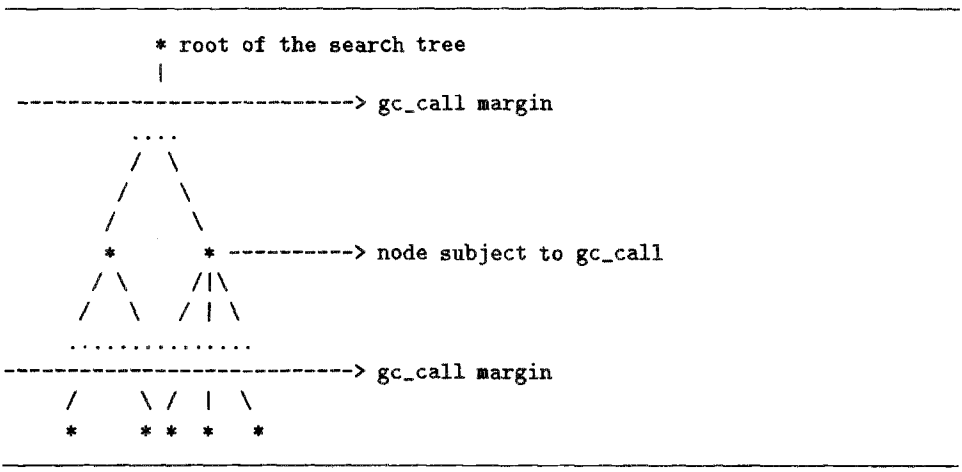


Fig. 2. Gc_call margins

The heap marks where the split is enforced can be set by the programmer or can be computed by the system heuristically, based on execution profiling data. The formula we are currently using for findall has a predetermined upper/lower region ratio of about 4 to 1:

```
#define HEAP_MIDDLE() \
    (*HEAP_BASE+(wam[HeapStk].end-*HEAP_BASE)>>2))
```

Remark that `HEAP_BASE` is actually a pointer to a stack as calls can be embedded, while `wam[HeapStk].end` is the actual end of the heap.

[8] and playing the game much better than its author

Finally, compiler generated `gc_call` annotations are possible. A preprocessor that does *abstract interpretation* can spot out potential `gc_call` points as memory needs for execution of a given goal can be statically approximated reasonably well, at least in the case of the very simple execution model of BinProlog. We are currently working on the mathematics of such estimations. This approach has the advantage of being portable to every Prolog having a heap-based fast findall, provided that it does not create garbage itself as *assert* based implementations often do in the dynamic code-space. Our heap-lifting technique, for example, ensures that execution-space is fully recovered after copying, with no additional cost.

5 Conclusion

A key difference between memory management in Prolog and functional languages comes from its OR-tree execution model. Basically branches of the tree do not need to communicate or to be present in the storage simultaneously. This is a natural advantage that seems not exploited at its potential by current Prolog garbage-collection technology that seems mostly inspired from Lisp memory management principles except perhaps in elaborated but relatively high overhead memory managers like the one presented in [5]. In this context, an interesting future work is to combine an independent parallel memory manager like the MALI system of [5] with the OR-parallel execution model for BinProlog.

In [13] a few garbage collection 'devils' lurking around in object-oriented systems (like the *indigestion devil* = clustering of long lived objects) are described. Similar things happen in the Prolog world too, quite often. As we described it in the section on performance evaluation of our heap-based findall, even top-quality commercial systems like Quintus and Sicstus 2.1 are not free from unexpected memory management related behaviour. By keeping BinProlog as small as possible (49K C emulator, 20K Prolog compiler) we hope that the simplicity and the proven soundness of our resource driven failure based proposal (with its sequential and OR-parallel implementations) will give robust and predictable Prolog systems while using Prolog's obvious natural strength: its backtracking mechanism.

We have proposed here an *'ecological'* approach to memory management. The concept is probably more important than the actual implementation: the key idea is to set up execution mechanisms that are able, while doing their natural, useful activities to achieve memory management objectives. One such example is our heap-lifting technique for implementing findall. This contrasts with the traditional view of memory management as a dedicated, resource consuming activity.

Acknowledgements. This work is supported by NSERC (grant OGP0107411) and the FESR of the Université de Moncton. We thank the anonymous referees for their constructive criticism, interesting comments and suggestions.

References

1. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog model and its performance. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, Cambridge, Massachusetts London, England, 1990. MIT Press.

2. K. A. M. Ali and R. Karlsson. Scheduling Or-Parallelism in Muse. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 807–821, Cambridge, Massachusetts London, England, 1991. MIT Press.
3. A. Appel. A runtime system. *Lisp and Symbolic Computation*, (3):343–380, 1990.
4. A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
5. Y. Bekkers and L. Ungaro. Two real-time garbage collectors for a prolog system. In *Proceedings of the Logic Programming Conference'91*, pages 137–149. ICOT, Tokyo, Sept. 1991.
6. M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* Phd thesis, SICS, 1990.
7. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of ACM*, 11(13):677–678, Nov. 1970.
8. B. Demoen. On the transformation of a prolog program to a more efficient binary program. Technical Report 130, K.U.Leuven, Dec. 1990.
9. R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.
10. P. Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Proceedings of the Russian Conference of Logic Programming.* St-Petersbourg, Sept. 1991.
11. P. Tarau. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, Sept. 1991.
12. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.
13. D. Ungar and F. Jackson. Outwitting gc devils: A hybrid incremental garbage collector. *OOPSLA'91 Garbage Collection Workshop Position Paper*, 1991.
14. D. H. D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, Oct. 1983.

A Code for copy_term in BinProlog 1.24


```
Some MACROS and functions:

- DEREF2 gets the last pointer in a reference chain and its value
- SETREF,GETREF sets and gets the value of a variable
- GETARITY extracts the arity form a cell
- VAR, INTEGER: type testing macros
- CHECK: overflow checking macro
- unwind_trail(To,From) - resets trailed variables
- NEXT - sets program counter P and jumps
- CONT - moves the continuation to register 1 and jumps
- UNIFAIL unify or signal failure

#define SAVED_H (*(A-2))
#define SAVED_TR ((term *)*(A-1))
#define SAVED_P ((instr)*A)
```

```
#define TR_TOP wam[TrailStk].top

#define TRAIL_IT(V) \
  {CHECK(TR_TOP,TrailStk,"trail overflow"); *TR_TOP++=(V);}

#define TRAIL_IF(V) \
  if((V)<SAVED_H) TRAIL_IT(V)

term recursive_copy_term(h,t,ct,A)
  register term h,t,ct,*A;
{ register cell val_t;
  DEREF2(t,val_t);
  if(VAR(val_t)) /* deals with variables */
    { SETREF(ct,ct);
      SETREF(t,ct);
      TRAIL_IF(t); /* if older than SAVED_H */
    }
  else if(INTEGER(val_t) || !GETARITY(val_t))
    SETREF(ct,val_t);  /* deals with atomic objects */
  else
    {
      SETREF(ct,h);
      SETREF(h,val_t);
      ct=h++; h+=(val_t=GETARITY(val_t));
      CHECK(h,HeapStk,"heap overflow in copy_term");
      while(val_t--) /* compound terms */
        h=recursive_copy_term(h,++t,++ct,A);
    }
  return h;
}

term copy_term(h,t,A)
  register term h,t,*A;
{ term ct,
  bakHB=SAVED_H,
  *bakTR=TR_TOP;
  SAVED_H=h; /* to TRAIL only what's before h */
  SETREF(h,h); /* makes a new variable to begin the copy */
  ct=h++;

    h=recursive_copy_term(h,t,ct,A); /* do the copy */

  TR_TOP=unwind_trail(TR_TOP,bakTR); /* reset the trailed vars */
  SAVED_H=bakHB; /* restore it as it was */
  return h;
}
```

B C-code for findall: 3 builtins


```
case LIFT_HEAP:
  /* assert: we are in the "lower" half of the heap */
  /* this is recursively true, for more than 1 split */
  PUSH_HEAP_MARKS(); /* remember where we are, for embedding */
  *HEAP_BASE = *HEAP_MARK = H; /* H in lower half */
  H=HEAP_MIDDLE(); /* set starting H in upper half */
NEXT(1); /* continue with next instruction */

case FINDALL_STORE_HEAP:
  /* assert: the answer is in the upper half, ready for copy */
  H=*HEAP_MARK; /* get old H in lower half */
  MAKE_LIST();  /* makes cons-cell for an answer */
  acc=H++; SETREF(acc,acc+2); acc=H++;
  (term)wam[HeapStk].margin=HEAP_MIDDLE(); /* sets new margin */

  H=copy_term(H,regs+1,A); /* copies the answer to lower half */

  wam[HeapStk].margin=wam[HeapStk].end; /* resets heap-margin */
  SETREF(acc,H);
  *HEAP_MARK=H; /* set new H in lower half */
FAILURE() /* backtracks */

case FINDALL_LOAD_HEAP:
  /* assert: all found answers are on a list in the lower half*/
  H=acc=*HEAP_MARK; /* resets H to lower half and sets acc */
  /* assert: the final list of answers is in acc */
  /* (as if an ORACLE had put it there) */
  /* everything done in the upper half can be forgotten safely */
  PUSHVAL(acc);
  MAKE_LIST(); /* makes a cons-cell */
  PUSHVAL(acc++); /* puts the (open) end of the list on it */
  PUSHVAL( *HEAP_BASE); /* puts the head of the list of answers*/
  POP_HEAP_MARKS(); /* pops the stack of embedded findalls */
  UNIFAIL(regs[1],acc) /* unify with the list of answers */
CONT(2); /* moves the continuation from regs[2] to regs[1] */
```

Replication-Based Incremental Copying Collection

Scott Nettles[1], James O'Toole[2], David Pierce[3], Nicholas Haines[4]

Abstract

We introduce a new *replication-based* copying garbage collection technique. We have implemented one simple variation of this method to provide incremental garbage collection on stock hardware with no special operating system or virtual memory support. The performance of the prototype implementation is excellent: major garbage collection pauses are completely eliminated with only a slight increase in minor collection pause times.

Unlike the standard copying algorithm, the replication-based method does not destroy the original replica when a copy is created. Instead, multiple copies may exist, and various standard strategies for maintaining consistency may be applied. In our implementation for Standard ML of New Jersey, the mutator continues to use the from-space replicas until the collector has achieved a consistent replica of all live data in to-space.

We present a design for a concurrent garbage collector using the replication-based technique. We also expect replication-based gc methods to be useful in providing services for persistence and distribution, and briefly discuss these possibilities.

Keywords: replication, garbage collection, incremental collection, concurrent collection, real-time garbage collection

1 Introduction

Copying garbage collection (GC) is an important memory management technique, but its application has been largely limited to situations that can tolerate GC pauses. There have been numerous schemes for incremental or concurrent copying collectors that are

Authors' affiliations: [1]nettles@cs.cmu.edu, [3]dp30@andrew.cmu.edu, [4]nickh@cs.cmu.edu, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.
[2]otoole@lcs.mit.edu, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597 and by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

"real time," i.e. that limit GC pauses to small bounded intervals. Real-time collectors interleave garbage collection with program execution, thus spreading out the copying work so that the individual interruptions are unobtrusive. These incremental collectors fall into one of two groups: those that require special hardware [6], and those that use virtual memory protection [2].

The disadvantage of techniques which use special hardware is that they are not portable. Techniques which use other operating system support such as the ability to control the virtual memory system are often not portable, and can be prohibitively costly due to the cost of trap handling or similar operations. We propose a new technique for implementing incremental and concurrent copying collectors that requires no special support from either hardware or operating system. In addition, it promises to be useful for other algorithms that use copying to provide features such as persistent data and distributed computing.

We first introduce our general approach, based on nondestructive copying or replication. Next we outline our experimental implementation and present preliminary performance measurements which demonstrate its excellent real-time behavior. Finally we discuss the application of the replication-based technique to concurrent collection, and suggest other applications.

2 The General Method

Copying collection works by copying all of the valid data from one region (from-space) to another (to-space), leaving the garbage behind. We assume the reader is familiar with the basic technique of copying collection as well as the notion of generational collection. The key operations of copying collection are as follows:

- *Copy* an object from from-space into to-space, leaving a forwarding pointer in the original from-space object.
- *Forward* a from-space pointer into to-space, if necessary copying the object it references, and redirecting the pointer to the to-space copy.
- *Scan* a to-space object, forwarding all of the object's pointers.

The mutator can perform the following operations on objects: read a field, write a field, and compare pointers for equality. Incremental GC requires that these operations be interleavable with the operations of the garbage collector outlined above. (Concurrent GC has much stricter requirements, discussed in section 5 below.)

Since the standard copying technique overwrites from-space objects with forwarding pointers in the *Copy* operation, most incremental collectors require that the mutator use only the to-space copy of an object. To maintain this invariant, the collection algorithm must rely on low-level hardware support. (E.g. hardware support for following forwarding pointers or trapping all attempts to access the unscanned portions of to-space.)

In contrast, our technique simply *replicates* the from-space object in to-space. A forwarding pointer is placed in a special word reserved at the head of the from-space object. Since the original object is not destroyed by the copying operation, any use of the object may continue to reference the original object. However, because multiple copies of an object may exist, read and write operations must adhere to one of several consistency protocols.

If reads are permitted to access either copy, write operations must modify both to-space and from-space replicas. Also, pointer-based equality tests must follow the forwarding pointers in order to ensure that only to-space (or only from-space) pointers are compared. In more sophisticated systems, where copying is used for purposes other than GC and there may be more than two replicas of an object, the mutator must modify *all* replicas (for this purpose we can make the forwarding chain circular by having a 'reversing pointer' in the newest replica). In this system, read operations can be freely interleaved with any of the GC operations, but under some consistency protocols the write operations may require synchronization with the collector, and care may be required to ensure that the mutator does not write from-space pointers into previously scanned to-space replicas.

This general protocol of reading any copy and writing all copies is a standard one used for maintaining replicated data, so we use the term "replication-based copying". Another possibility is to have write operations modify only the newest version of an object, in which case the read operations for *mutable* objects must always read the newest version. In section 5, we discuss this possibility, which may be preferable for concurrent applications.

Note that these operations are distinct from that of updating the 'root set', that set of pointers directly visible to the mutator (registers, the stack, etc.). At some point in the GC process, these pointers must be updated. In a standard incremental collector, this is done immediately after the 'flip' by a simple 'forward' operation to start the GC. With a replication-based algorithm, it is possible to delay this step until just before the flip, after copying all live data into to-space. By using this technique, the collector can ensure that the mutator uses only from-space objects. In this case, there is no need for the collector to synchronize with the mutator except very briefly at flip time. Notice that this variation is not fully general, as it does not provide for more sophisticated uses of copying.

The advantage of the above technique is that it allows for incremental collection with no special hardware or OS support, but what are the disadvantages? First, it requires one extra word per object for the forwarding pointer. Fortunately, this extra word can often be absorbed into other object header words which are already present. The second disadvantage is that the consistency protocol may make writes (and possibly reads of mutable objects) more expensive. For some languages this would be unsatisfactory because mutations are common. However, for applicative languages like SML, in which side effects are less frequent and mutable objects are clearly distinguished by a type system, this runtime cost is probably not a problem. The third disadvantage is that of copying latent garbage, but this is an inevitable cost of any incremental method, and all such garbage is discarded by the next collection. The final disadvantage is that tests of pointer equality become more expensive. This may be a serious disadvantage for Lisp family languages where the use of eq is common. It is probably less important for SML, because equality testing is already expensive, and not as frequently used.

3 Implementation

We have built a prototype implementation of a replication-based incremental collector for SML/NJ (version 66). In order to quickly test the utility of the replication-based method, we chose to implement a simple variation of the general replication algorithm.

In this variation, the mutator uses only the from-space replicas. Therefore, the mutator need not adhere to a consistency protocol, and so only one small change to the SML/NJ compiler was required. The rest of the implementation work required modifications to the standard SML/NJ garbage collector.

SML/NJ uses a simple generational copying collector [1], with two generations known as new-space and old-space. The new-space is used for newly allocated data, and the old-space contains data which has survived at least one collection. When the new-space fills, a 'minor' collection is performed, copying data from the new-space to the old-space. The compiler keeps a record (the 'store list') of all writes to mutable objects so that references from the old-space into the new-space can be found during minor collection. When the old-space fills, a 'major' copying collection is performed. Minor collections are typically short and non-disruptive, but major collections are often lengthy.

Our implementation leaves minor collections as they are, but makes the major collections incremental, doing some portion of the major collection at each minor collection. There are several reasons for this choice. First, it avoids having the allocator allocate the forwarding word; instead it is added when objects are copied from new to old. This avoids a change to the compiler backend's allocation primitives. Second, since the GC is in control during a minor collection, it is convenient and cheap to do incremental work at that time. By limiting the amount of incremental work done at each minor collection, we can keep pauses brief, within a factor of, say, three times as long as for a minor collection alone.

We use the strategy, described above, of only updating the root set when the GC is complete. The mutator can therefore only see from-space objects. We use the store list during each GC increment to update to-space versions and rescan them if necessary. The SML/NJ compiler version 66 keeps a log of all mutations which store pointers, for use by the generational collection algorithm. We modified the mutation log to include all mutations, so that the incremental collector can update to-space. This avoided the need to modify the compiler to add a write-all-replicas protocol.

In order to ensure that the garbage collector terminates, we must guarantee that all live data will be replicated in to-space before from-space overflows with new data copied by the minor collections. We want to restrict the amount of GC work done in each increment, but still ensure that a 'flip' takes place before from-space is full. Otherwise, when from-space fills, the incremental collector will have to perform a large amount of remaining gc work, which will be tantamount to a major garbage collection pause.

In the prototype implementation, we guarantee that this will not happen by requiring the incremental collector to copy more objects into to-space than were added to from-space by the minor collection. Therefore, the duration of the incremental collector's pauses can be controlled by adjusting the size of the new-space and the amount of additional incremental copying done.

4 Measurements

The initial performance measurements for our prototype implementation are shown in table 1. The table describes the garbage collector pauses which occurred during a single test case. The test case compiled a significant part of the SML/NJ compiler, and was run without paging activity on a DECstation 5000/200 equipped with 64 Mb of main

	#minor pauses	mean pause	modal pause	max. pause	90% below	#major pauses	mean pause	max. pause	total GC
orig	5422	17ms	15ms	734ms	45ms	48	2.2s	5.0s	201s
incr	5422	57ms	46ms	499ms	93ms	—	—	—	312s

Table 1: Pause timings for stop-and-copy vs. incremental collectors.

memory. The incremental collector completely eliminates the major collection pauses of 2 to 5 seconds with which every SML/NJ user is aggravatedly familiar.

The minor pauses measured for the original collector represent the delay caused by a collection of old-space into new-space. The minor pause time for the incremental collector includes the generational collection of old-space into new-space and also the work done by the incremental algorithm transporting objects in the from-space (old-space) to the to-space.

The statistical distribution of the minor pause times are both unimodal, with pronounced modes at at a pause time of less than 50ms, but with a long tail to several hundred milliseconds. Our collector increases the mode, but its performance appears to be interactive enough to remain acceptable to users.

The measured mean pause time for our collector is 57 milliseconds. We expect to reduce that figure to 50ms or less by varying the control parameters of our implementation. Reducing the size of the new-space and the fraction of incremental work done will shorten these pauses. Because our collector is incremental, we can also cut short the incremental collection activity if it becomes too lengthy.

The total garbage collection time is increased by more than 50% relative to version 66 of the SML/NJ. We anticipate being able to reduce this to approximately 10% by simple optimizations of our existing code (we believe most of this increase is due to the fact that the prototype implementation performs a 'flip' operation twice as often as the standard algorithm. There is no mutator time overhead in the current implementation.

	#objects copied	total size	overhead bytes	% heap
all objects	27M	344Mb	108Mb	24%
mutable only	1.76M	18Mb	7Mb	2%

Table 2: Space overhead of forwarding words for incremental collector.

Table 2 shows the total space overhead of our system. The total size measurements given in the table do not include the overhead for forwarding words, and the percentage figure measures the amount of overhead bytes as a percentage of the total heap size, including overhead. The prototype implementation uses a separate forwarding word for

every object, which results in a very high space overhead of 24% because a majority of objects are two-word records ('cons cells') with a header word. However, we can reduce the space overhead by storing the forwarding pointer and the header information in the same word. In this scheme, a replicated object has header information on only the newest copy. Any operation which needs the header information must follow forwarding pointers to locate the newest copy of the object. In the write-newest protocol, this optimization can be applied to all objects, eliminating the space overhead entirely.

However, in the write-all consistency protocol, even the newest replicas of mutable objects require 'backwarding pointers', so this optimization cannot be applied to them. In this case the space overhead would be reduced to just 2% of the heap, as shown in the table. Certain operations such as `size` would need to follow the forwarding pointer chain, as well as other low-level run-time operations such as tag checks.

5 Concurrent Collection

The same technique is applicable to a concurrent system, in which the collector and the mutator run in parallel, as separate threads of a single process. This is only an advantage in multi-processor systems, when the collector may be running on one processor while the mutator (or mutators) is running on the others—in single-processor systems one is merely sacrificing control over when the collector runs, which is pointless.

In a concurrent system, not only must the semantic operations of the collector and mutator be independent, as discussed above, but the individual machine instructions of each must be interleavable. This is a much stronger condition, but it is not hard to satisfy in a concurrent version of the incremental collector described above.

First consider whether running our prototype incremental collector concurrently with the mutator would produce read/write conflicts. The mutator only reads or writes from-space replicas. The collector reads from-space replicas, but writes only to-space replicas. The collector also writes the forwarding words of from-space replicas, which the mutator does not access. Thus the collector will not interfere with the mutator. If the forwarding word and the header word are merged, then the collector and the mutator could conflict while accessing this word. However, as long as the collector can atomically update the header word to install the forwarding pointer, there is no danger. The mutator will either read the from-space replica's header word before it is overwritten, or follow the forwarding pointer to the to-space replica.

Now consider whether the mutator will interfere with the collector. It can only interfere by writing a word the collector is reading. But at worst this would cause the collector to copy the wrong value to to-space and at some point this mistake would be corrected in the process of updating to-space to reflect mutator writes. Thus the mutator does not interfere with the collector.

Almost all of the synchronization needed to make our prototype incremental collector concurrent is already present in the incremental collector, because the effects of mutator stores are communicated to the collector indirectly through the store list. Implementing a concurrent collector is simply a matter of managing the handoff of the current roots and the store list, and synchronizing to forward the root pointer set when the collection terminates.

6 Related Works

Real-time incremental or concurrent garbage collection has been the goal of many research projects in the past. Recent work includes that by Ellis, Li, and Appel [2], which exemplifies the use of the virtual-memory system to control the GC behavior, and Halstead [5], using hardware improvements. The first real-time copying collector, by Baker [3] requires special hardware, and paved the way for many other such systems. Some existing algorithms work on stock hardware without operating systems support, such as those by Brooks [4] and later North [8], but none of these show such small time and space overheads as our technique.

7 Future Work

Since the overhead for this new technique appears to be acceptable, we believe it will be useful when applied to several other interesting GC-related algorithms. These other algorithms can all make use of copying to achieve some useful end other than collecting garbage, and may be able to share some runtime and/or storage costs with the garbage collector.

One such algorithm is used to implement persistent storage. One of us has implemented a persistent storage system based on copying objects from the heap into a persistent heap [7]. A major performance bottleneck is the need to scan the entire heap for pointers to objects which have been copied. Nondestructive copying will eliminate this scan.

We are also interested in using copying to implement mechanisms for distributed computing, such as those required by object repositories. In these distributing computing systems, data which will be replicated at a remote machine is copied into a message buffer, linearizing it for transmission purposes. Again nondestructive copying will greatly lessen the overhead of such copies. Also, we anticipate a simple interface between the local GC described here and the global (distributed) GC required in such a system.

A final possibility is the technique of delayed hash consing. Here the system tries to detect if two (immutable) objects are identical. If they are then they can be merged. This merge can be implemented by nondestructively adding a forwarding pointer from one object to the other. This technique may greatly reduce the amount of heap space needed.

We are extending our implementation in these directions and exploring some ideas for "opportunistic" GC [9], in which the timing of garbage collections is chosen to minimize disruptiveness. We are investigating triggering GC within the user-interaction loop, immediately before prompting for input, and after long waits for input. As a start, we are adding some very simple code to disable the incremental technique when the mutator is compute-bound, reverting to the more efficient stop-and-copy collection, the pauses of which will not be noticed during the compute delay.

8 Conclusions

We have introduced a promising new copying GC technique, replication-based copying. This technique is especially well suited to languages like SML where mutations are rare.

We have implemented a simple incremental GC for SML/NJ based on this technique and have obtained preliminary data showing our idea to be workable. We are continuing work to make related algorithms equally practical.

Acknowledgments: Scott Nettles and James O'Toole would like to thank DEC's Systems Research Center for support as summer interns, during which time this idea was originally conceived. Scott Nettles and David Pierce would like to thank Peter Lee for support with the implementation. Thanks also to John Reppy for his suggestion to merge the forwarding pointer and header word. Greg Morrisett provided many hours of helpful conversation. Thanks to Penny Anderson, Mark Sheldon, Ellen Siegel and the Venari group for proofreading.

References

[1] A. Appel. Simple generational garbage collection and fast allocation. *Software–Practice and Experience*, 19(2):171–183, February 1989.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[3] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 256–262, 1984.

[5] Robert H. Halstead, Jr. Implementation of multilisp: LISP on a multiprocessor. In *ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

[6] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.

[7] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. Technical Report CMU-CS-91-173, Carnegie Mellon University, August 1991.

[8] S. C. North and J.H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 113–133. Springer-Verlag, 1987.

[9] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *Proceedings of ACM SIGPLAN 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1989.

Atomic Incremental Garbage Collection

Elliot K. Kolodner[1] and William E. Weihl[2]

[1] IBM Science and Technology, Technion City, Haifa 32000, Israel.
email: kolodner@haifasc3.ibm.vnet.com
[2] MIT Lab. for Computer Science, 545 Technology Square, Cambridge MA 02139, USA.
email: weihl@lcs.mit.edu

Abstract. A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. These features enhance reliability and simplify programming by preventing errors due to explicit deallocation, by masking failures and concurrency using transactions, and by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management is useful for programming languages for reliable distributed computing, programming languages with persistent storage, and object-oriented database systems.

Many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. We present garbage collection and recovery algorithms for a stable heap implementation that meet these goals and are appropriate for stock hardware. The collector is incremental: it does not attempt to collect the whole heap at once. The collector is also atomic: it is coordinated with the recovery system to prevent problems when it moves and modifies objects. The time for recovery is independent of heap size, even if a failure occurs during garbage collection.

1 Introduction

A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. Automatic storage management, used in modern programming languages, enhances reliability by preventing errors due to explicit deallocation (e.g., dangling references and storage leaks). Transactions, used in database and distributed systems, provide fault-tolerance by masking failures that occur while they are running. A uniform storage model simplifies programming by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management will make

This paper reports on research done by the authors at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

The research was supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation.

Keywords: database, distributed systems, garbage collection, object-oriented, persistence, recovery, transactions.

it easier to write reliable programs and could be useful in programming languages for reliable distributed computing [10, 24], programming languages with persistent storage [1, 2], and object-oriented database systems [8, 26, 38, 40].

In earlier research [18, 19] we designed algorithms suitable for the implementation of small stable heaps. However, many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. This paper presents a garbage collection algorithm suitable for the large stable heaps necessary to support these applications.

A recovery system provides fault-tolerance for transactions; it manages information that ensures that the effects of successful transactions persist across failures and that unsuccessful transactions have no effect. A garbage collector typically moves and modifies objects as it collects storage that is no longer in use. It moves objects to improve locality of reference and reduce fragmentation; it modifies them in order to speed its work and reduce the amount of additional storage it requires. In a stable heap the movement and modification of objects by the garbage collector may interfere with the work of the recovery system; yet, the recovery system must be able to recover the objects modified by the collector and find the moved ones when a failure occurs. The collector must also have access to the recovery information; objects may be reachable from this information that the collector would not otherwise retain. A collection algorithm that solves these problems and is coordinated correctly with the recovery system is called an *atomic garbage collector*.

In our earlier work we introduced this notion of atomic garbage collection and presented an algorithm for it. We based our earlier atomic collector on a stop-the-world copying collector; it suspends work on all transactions while it collects and traverses the stable heap. These pauses grow longer as the heap grows larger. Similarly, the recovery system used in our earlier work requires a traversal of the whole stable object graph after a crash. For an application with a large stable graph, this traversal delays recovery and reduces the availability of the application. The exact heap size at which the pauses for garbage collection or the time for recovery become intolerable depends on the application, its response time requirements, and its availability constraints, as well as on hardware characteristics such as processor speed.

Our current research builds on our previous research: we have designed an integrated atomic garbage collector and recovery system appropriate for a large stable heap on stock hardware. The collector is incremental; i.e., it does not attempt to collect the whole heap in one pause. The time for recovery is independent of heap size even if a failure occurs during garbage collection, and can be shortened using a checkpointing mechanism. The design is for a conventional processor with virtual memory; no other special hardware is assumed.

This paper describes our algorithm for atomic incremental garbage collection and its interaction with the recovery system. The first author's dissertation [20] describes the recovery system in detail. In it we also show how to avoid the costs of atomic garbage collection for volatile objects by dividing the heap into stable and volatile areas. Storage management in the volatile area is provided cheaply by a normal garbage collector; the more expensive atomic garbage collector is used only in the stable area.

We have implemented a stable heap prototype to show the feasibility of our algorithms. The current implementation of Argus [25] serves as the basis for the prototype; we replaced its existing storage management and recovery algorithms.

There has been other work dealing with the problem of providing fault-tolerant heap storage, but none of the solutions has been satisfactory. Some work (e.g., [7, 33]) provides persistence but not transactions, so it offers less functionality. PS-algol [2] uses a stop-the-world garbage collector and does not permit garbage collection to occur while transactions are in progress. It also uses a less general transaction model and a recovery system that imposes a high run-time overhead. Earlier work on Argus [29] uses a normal garbage collector, but treats all crashes as media failures; as a result, recovery from a system failure is relatively slow, particularly for large heaps. Our work grew out of an attempt to design a faster recovery system for Argus.

Detlefs's work [11] is the closest to ours; he has published an algorithm he calls concurrent atomic garbage collection. In his algorithm the pauses for garbage collection and the time for recovery are independent of heap size, but the pauses are too long. Each pause requires multiple synchronous writes to disk; furthermore, these writes are random. Our algorithm is better integrated with the recovery system and does not require any synchronous writes to disk.

Here is an overview of the structure of this paper. Section 2 presents our model of a stable heap. Section 3 describes an approach to recovery called repeating history [27]. Using the approach, modifications to objects in the heap follow a write-ahead log protocol. The protocol ensures that the modifications are repeatable after a failure. Section 4 describes our algorithm for atomic incremental garbage collection. It shows how to use the write-ahead log protocol to make the steps of an incremental copying collector repeatable. It also discusses the interactions between the collector and a recovery system. Section 5 discusses the applicability of our algorithm to other recovery systems. Section 6 concludes with an evaluation of the algorithm.

2 Stable Heaps

We abstracted our model of a stable heap from the model of computation used by Argus, a programming language for reliable distributed computing, for local computation at each node in a distributed system. The stable heap model is also appropriate for object-oriented database systems and other programming languages with persistent storage. We begin this section by describing the model. Then we discuss the hardware and operating systems for which our design is appropriate.

2.1 System Model

In our model, computations on shared state run as atomic transactions [15], and storage is organized as a heap. Transactions provide concurrency control and fault tolerance; they are *serializable* and *total*. Serializability means that when transactions are executed concurrently, the effect will be as if they were run sequentially in some order. Totality means that a transaction is all-or-nothing; i.e., either it completes entirely and *commits*, or it *aborts* and is guaranteed to have no effect.

A transaction consists of a series of short low-level recoverable actions: a *read* action reads a single object, an *update* action modifies a single object, and an *allocate* action creates a new object. These actions synchronize through logical mutual exclusion locks on objects. In practice these mutual exclusion locks may be of a coarser granularity. For example, in Argus most read and update actions are indivisible. Indivisibility is enforced by allowing context switches only at low-level action boundaries.

Objects shared among transactions must be *atomic*. Atomic objects provide the synchronization and recovery mechanisms necessary to ensure that transactions are serializable and total. Atomic objects can be mutable or immutable. Immutable objects are always atomic because their values never change. For the purposes of this paper we assume that the heap synchronizes access to mutable atomic objects using standard read/write locking (i.e., shared for read, exclusive for write), and we describe appropriate recovery mechanisms. Using the built-in types, a programmer can build objects of user-defined atomic types [35] that exhibit greater concurrency than the built-in atomic types with the aid of lazy nested top-level transactions [17, 32] and multi-level concurrency control [37].

A heap consists of a set of root objects and all the objects accessible from them. Objects vary in size and may contain pointers to other objects. In a stable heap, some programmer-specified roots are stable; the rest are volatile. The stable roots are global. The *stable state* is the part of the heap that must survive crashes; it consists of all objects accessible from the stable roots. The objects in the stable state must be atomic. The *volatile state* does not necessarily survive crashes; it consists of all objects that are accessible from the volatile roots, but are not part of the stable state, e.g., objects local to a procedure invocation, objects created by a transaction that has not yet completed, and global objects that do not have to survive crashes.

The programmer sees one heap containing both stable and volatile objects. He can store pointers to stable objects in volatile objects, and can cause volatile objects to become stable by storing pointers to them in an object that is already stable. (A volatile object actually becomes stable when a transaction that makes it accessible from a stable object commits.) Transactions share a single address space that contains both shared global objects and objects local to a single transaction; the programmer does not need to move objects between secondary storage and a transaction's local memory, or distinguish between local and global objects.

For the purposes of this paper we assume that all of the roots of the heap are stable. The first author's dissertation [20] shows how to deal with volatile state.

2.2 Implementation Platform

Our design is for conventional hardware – stock uniprocessors with virtual memory No special-purpose hardware to support recovery or garbage collection is assumed.

The design requires an operating system that allows a program some control over the virtual memory system. Primitives are needed to control when a page of virtual memory can be written to the backing store and to set protections on pages. The ability to preserve the backing store for virtual memory after a crash is also required. Mach [30] satisfies these requirements; for the prototype we ported Argus to run under it.

3 Recovery and Failure Model

Below we describe the storage architecture for a stable heap, the failure model, recovery, and optimizations to recovery.

3.1 Storage Architecture

A recovery system provides fault-tolerance by controlling the movement of data between the levels of a storage hierarchy. In a typical database there are four components in the hierarchy: (1) main memory, (2) disk, (3) log, and (4) archive. We assume a similar hierarchy for the design of our algorithms.

A database keeps its data on disk, which is non-volatile, and uses main memory, which is volatile, as a cache or buffer pool. A buffer manager decides which pages to keep in the cache; it reads pages from disk into main memory and writes modified pages back to disk. The recovery system may constrain the buffer manager by *pinning* a page in main memory; a pinned page may not be written back to disk until recovery unpins it. For a stable heap, the main memory and disk together implement a one-level store or virtual memory.

The log is a sequential file, usually kept on a stable storage device, to which the recovery system writes information that it needs in order to redo the effects of a committed transaction or undo the effects of an aborted transaction. A stable storage device [21] is often implemented using a pair of disks; with very high probability, it avoids the loss of information due to failure. The recovery system does not directly write to the log on stable storage; rather, it spools information to a log buffer. When a buffer fills, recovery writes it to disk asynchronously and begins spooling to the next buffer. A well designed recovery system synchronously writes a buffer to stable storage, or *forces* the log, only at transaction commit when it must ensure that the effects of the transaction survive failure.[3] In this paper when we say *write to the log*, we mean spool to the log buffer. If we want to describe a synchronous write, we use the phrase *force the log*. To distinguish the part of the log on stable storage from the part in the log buffer, we call the former the *stable log* and the latter the *volatile log*. When we use the word log without qualification, we mean the whole log, both its stable and volatile parts.

The archive is an out-of-date copy of the database; it may be on disk or some cheaper non-volatile medium such as magnetic tape.

3.2 Failure Model

A recovery system deals with three kinds of failure: (1) transaction, (2) system, and (3) media. A transaction fails when it aborts; the recovery system may use information in main memory, on the disk, or in the log to ensure that the transaction has no effect.

A system failure can be caused by software (e.g., inconsistent data structures in the operating system) or hardware (e.g., power failure). When the system fails main

[3] A high performance transaction system will use *group commit* [14] instead of forcing the log for every transaction; this allows the buffer to fill before writing it to stable storage, and commits many transactions at the same time.

memory is lost, but the disk and stable log survive. A system failure also aborts transactions that are active when it occurs. The recovery system uses information in the stable log and on the disk to recover the state of the heap. The recovered heap reflects the cumulative effects of all the transactions that committed before the failure, and none of the effects of aborted transactions. We also call a system failure a crash.

A media failure occurs when a page or several pages of the disk get corrupted. The recovery system uses the log together with the archive to recover the pages.

3.3 Recovery

Given the storage architecture and failure model described above, we describe a way to do recovery, called repeating history, due to Mohan, et. al. [27]. We chose repeating history because it is simple compared to previous recovery algorithms [4, 16, 23], and easy to optimize.

The key to repeating history is the *write-ahead log protocol*, which we also call the *redo protocol*. The recovery system follows the protocol for all modifications to objects:

1. It pins the page on which the object resides in main memory. The buffer manager may not write the pinned page to disk.
2. It modifies the object on the page.
3. It spools a record containing redo information to the log buffer. The record contains the address at which the modification occurred and the new value.
4. At this point, the modification is complete and the protocol returns to its invoker.
5. After the redo record is in the stable log, the page is unpinned. The buffer manager is then free to write the unpinned page back to disk.

The write-ahead log protocol ensures the following property: if a modification is on disk, the redo record describing the modification is in the stable log. The *repeating history invariant*, which simplifies recovery after a crash, follows directly from this property:

Invariant 3.1 (Repeating History) *The disk state that would be produced by applying the stable part of the redo log to the disk (i.e., carrying out each of the redo actions in the order they are recorded in the log) is a state that actually occurred at some previous point in the computation.*[4]

The action of redoing the log is called *repeating history*.

To deal with transaction abort, the recovery system includes undo information together with the redo information in the record it writes during the write-ahead log protocol. The undo information may be *logical*, the name of an operation and arguments to the operation, or *physical*, the previous state for the part of the object that is modified. Usually logical information takes up less space in the log than

[4] Formally, an invariant is a predicate on state; at first glance this statement of the repeating history invariant may not appear to be such a predicate. However, we can formalize it by defining the redo function determined by the log, and defining a history variable that captures the sequence of states through which the computation passes.

physical information. To abort a transaction, the recovery system undoes the transaction's updates in reverse order. Undoing an update is a modification so it follows the write-ahead log protocol and writes a redo record describing the undo. A redo record written by undo is called a *compensation log record* or CLR. There is no undo information in a CLR; undo never has to be undone.

The repeating history invariant simplifies recovery after a crash. Recovery begins by repeating history, i.e., applying the redo information in the stable log to the disk. According to the invariant, this brings the database to a state from which it is valid to abort the transactions that were active before the crash.[5] Then recovery completes by using its normal method for transaction abort to abort the active transactions.

The repeating history invariant is also useful for our atomic garbage collector. In the design of our collector, we depend on the invariant to bring the heap to a state from which the collector can complete its work after a crash.

3.4 Optimizations

It is easy to optimize a recovery system based on repeating history, and to understand why the optimizations work. Logical undo is one such optimization. We briefly describe two other optimizations below; both shorten recovery times after a system failure.

First, the buffer manager writes a *page-fetch record* to the log each time it fetches a page from disk into main memory, and an *end-write record* just after an updated page of main memory reaches disk. These records contain the number of the page that was read or written. Using these records, the recovery system can deduce a superset of the pages that were dirty at the time of a system failure. When it repeats history, it only has to apply redo records to the pages in this set.

Second, the recovery system checkpoints at regular intervals to keep the time for recovery short. To checkpoint, it stops the system in a low-level quiescent state, a state for which no transaction is in the middle of the write-ahead log protocol (steps 1 – 4 of the protocol discussed in Sect. 3.3). Then it constructs and writes a *checkpoint record* to the log. The record contains a list of the dirty pages at the time of the checkpoint and for each page the log address of its last page-fetch record. Using this information after a system failure, recovery deduces a point in the log from which it starts repeating history. These checkpoints are cheap; they do not require any synchronous writes, and they halt the system for very brief periods.

4 Atomic Incremental Garbage Collection

The principal requirement for our atomic garbage collector is that it be suitable for a large heap. There are two implications: (1) the pauses associated with garbage collection must be short enough to support interactive response times, and (2) the

[5] Recovery is a bit more complicated for an update to an object that spans multiple pages. For such an update the write-ahead log protocol pins all updated pages until individual redo records describing the change to each page are in the stable log. After a crash, recovery applies the redo records for a multi-page update only if all of its records are in the stable log.

collector must interact well with virtual memory. Two general techniques have been used to shorten garbage collection pauses: (1) incremental garbage collection [3], and (2) dividing the heap into independently collectible areas [5]. Steps of an incremental collector are interleaved with normal program steps such that the pause due to each incremental step is small. An incremental collector is also called real-time if there is a bound on the longest possible pause. Many incremental collectors have been based on Baker's algorithm [3], which is a copying collector.

A good division of the heap into independently collectible areas leaves few inter-area references and places objects with similar lifetime characteristics into the same area. For programs without persistent storage, one automatic way of dividing the heap without programmer intervention is based on the age of objects; this is called generational collection [22, 28, 34]. Generational collection depends on an observed behavior of program heaps that new objects are more likely to become garbage than old objects; it concentrates its work on the areas containing the youngest objects, where the most storage will be reclaimed for the least amount of effort. Generational collection might also be appropriate for persistent heaps; but this can be determined only by studying real workloads.

In this paper we discuss incremental collection. In the first author's dissertation [20] we apply both techniques to shorten garbage collection pauses: we show how to divide the heap into a stable area and a volatile area, and we use our atomic incremental collector to collect the stable area.

For large heaps implemented in virtual memory, an important purpose of garbage collection is to reorganize the heap to provide good paging performance. Reorganizing the heap requires a collector that can move objects, e.g., a copying collector. Copying collectors can increase locality of reference and reduce paging by moving objects that are referenced together to the same page [9, 28, 39].[6]

The other requirement for our algorithm is that it work well on stock hardware. Without hardware assists, Baker's incremental garbage collector is expensive–it requires a comparison on every heap reference. A variant of Baker's collector [6] substitutes a memory indirection for the comparison, but is still too expensive. Ellis, Li and Appel [13] have shown how the virtual memory hardware on stock hardware can be used to facilitate incremental copying garbage collection with lower overhead. Zorn [41] has suggested a similar technique, but it has a higher overhead.

We base our atomic incremental garbage collector on the collector of Ellis, Li and Appel (hereafter attributed to Ellis). Before describing our collector, we review Ellis's collector, and we show how a copying collector interferes with recovery. After describing our collector, we discuss its other interactions with recovery and its performance.

4.1 Incremental Collection

Ellis's collector is based on Baker's algorithm [3]. As in other copying collectors, Baker divides memory into from-space and to-space. In one collection cycle, the

[6] The actual performance of these specific techniques (i.e., a measure of how much they actually increase locality) requires further investigation. Better ways to increase locality may be discovered, but they will also require copying.

collector copies the objects accessible from the roots from from-space to to-space. As each object is copied, a forwarding pointer is inserted in its from-space copy. Forwarding pointers preserve sharing in the object graph.

In Baker's algorithm the program doing useful computation (often called the mutator [12]) and the collector run as coroutines subject to a synchronization constraint that we discuss below. The mutator calls the collector to do some work each time it allocates a new object. When the garbage collector runs, it either *scans* a fixed number of locations in to-space (i.e., it converts from-space pointers in those locations to to-space pointers, copying objects if necessary) or it *flips*. At a flip to-space becomes from-space, a new to-space is allocated, and the collector copies the root objects to the new to-space. Baker's algorithm can be extended in the obvious way to allow the collector and multiple mutators to run in separate threads.

Read Barrier. Synchronization between the mutator and the collector depends on the invariant that the mutator, which in the case of a transaction system includes transactions, never sees a pointer into from-space. This invariant is established at a flip: the root objects (i.e., those objects referenced by a register, a stack, or an own variable) are copied to to-space, and the corresponding registers, stack locations and own variables are updated to point to the to-space copies.

The invariant is enforced during the collection by the so-called "read barrier". The read barrier prevents the program from seeing pointers into from-space. Baker's implementation of the read barrier requires a comparison on every reference to the heap.

Ellis suggests a cheap implementation of the read barrier. After the root set is copied to to-space at a flip, the collector uses the virtual memory hardware to protect the unscanned pages of to-space against both reads and writes. When the program tries to access an unscanned page, the collector fields the resulting trap and scans the page, translating all from-space addresses on the page to the corresponding to-space addresses. Scanned pages do not contain from-space addresses; the program never accesses an unscanned page, so it never sees a from-space address.

Ellis's approach is cheap; it adds little to the overall garbage collection time since there will be at most one trap per page of to-space. However, Ellis's collector might not be as incremental as we would like. The distribution of read barrier traps will be skewed to be very frequent just after a flip and each trap requires that a whole page be scanned. Ellis suggests techniques for shortening the flip time and for bounding the time taken to scan a single page. Nevertheless, the pauses for garbage collection just after a flip might be long and frequent, thereby defeating the purpose of incremental collection. The seriousness of this problem depends on the mutator's rate of access to the heap and locality of reference. Since we expect the rate of access to a persistent heap to be low, we do not believe it will be a problem for us. We are building a prototype that will enable us to measure the length and frequency of the pauses attributable to the read barrier.

Scanning An Arbitrary Page. Ellis's algorithm requires the capability of scanning an arbitrary page of to-space. This is not a problem if every memory location is

tagged to indicate whether or not it contains a pointer. However, tagging is expensive. Instead a heap implementation may construct objects such that the first cell contains a descriptor giving the object's low-level type, the object's length, and the positions of pointers in the objects. Since objects may cross page boundaries, the collector needs an additional mechanism to scan an arbitrary page. For that purpose it creates a data structure as it copies objects called the Last Object Table [31]. This table is an array indexed by to-space page number; the entry for a page contains the location of the last object on that page. To scan an arbitrary page, the collector uses the Last Object Table to find the last object on the previous page. Then it uses the object descriptors to parse the objects on the page, and find and scan their pointers.[7]

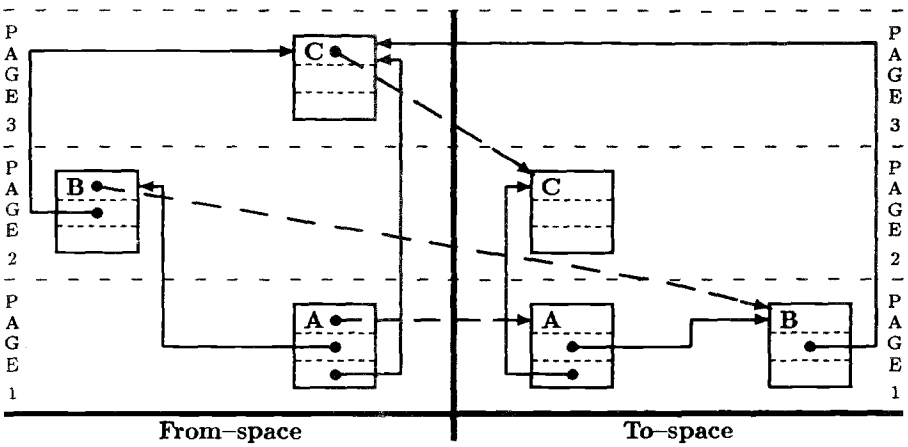
4.2 Why Copying GC is not Atomic

Copying garbage collection is not atomic for several reasons. First, a copying collector modifies from-space by inserting forwarding pointers in objects, and to-space by copying and scanning objects. We assume that the object does not have an extra cell to hold the forwarding pointer; thus, the pointer overwrites a cell of the object. Second, the collector moves objects, changing their addresses. We discuss these two problems below; they were first described in a report on our previous research [18, 19]. We describe solutions to the problems in Sect. 4.3. There are also other interactions between the collector and the recovery system; we describe them in Sect. 4.4.

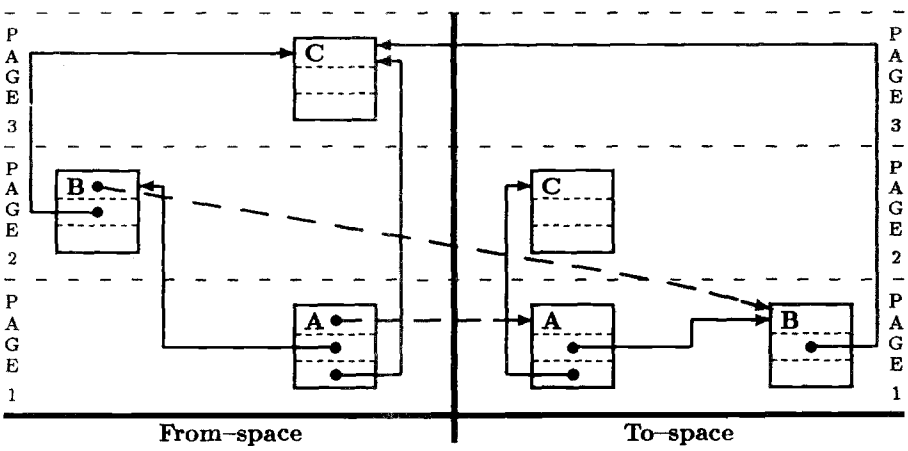
Modifications to Objects. As a copying collector modifies from-space and to-space, the pages of virtual memory are paged into volatile main memory and out to disk by the buffer manager. A crash can easily leave the contents of the disk in an inconsistent state. The following two examples show the kinds of problems that can occur and that must be avoided by an atomic collector.

The first example, illustrated in Fig. 1, shows that forwarding pointers can be lost. If an object is copied, but its forwarding pointer does not survive the crash, then recovering on the basis of information already copied to to-space would not preserve the sharing in the graph of accessible objects. Suppose the copying of the object graph in Fig. 1.a were interrupted by a crash after objects A, B, and C had been copied to to-space, but before the pointer to object C from object B had been replaced by a to-space pointer. Figure 1.a shows virtual memory just before the crash. Figure 1.b shows a possible state of the disk after the crash. The crash occurred after pages 1 and 2 of from-space and pages 1 and 2 of to-space had been written to disk, but before page 3 of from-space had been written. The forwarding pointer for object C has been lost, even though the object has already been copied to to-space.

[7] In place of the Last Object Table, Ellis suggests using a crossing map, a bitmap that contains a bit for each page of to-space. The bit is set if an object crosses the boundary at the beginning of a page. To make sure that there are pages for which the bit is not set, the allocator may waste space at the end of some pages. Using the Last Object Table, the first object on an arbitrary page can be found faster, and the space taken by the table is likely less than the wasted space in Ellis's scheme.



1.a: Virtual Memory Before a Crash

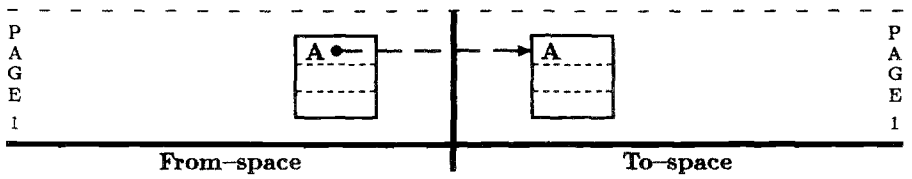


1.b: Disk Just After Crash

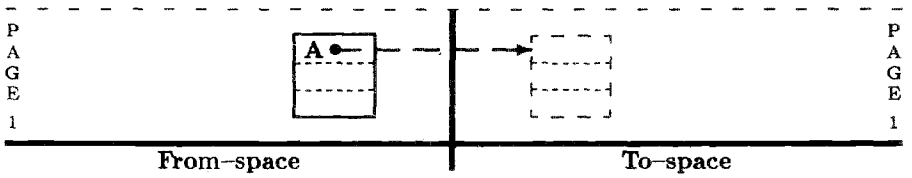
Fig. 1. Lost Forwarding Pointer

The second example, illustrated in Fig. 2, shows that the contents of the cell overwritten by the forwarding pointer can be lost. Figure 2.a shows an object copied from from-space to to-space; a forwarding pointer was placed in the from-space copy. The forwarding pointer overwrites a cell of the from-space copy. The page of from-space on which the old object version resides is then written to disk. Figure 2.b shows what happens if the system crashes before the new version in to-space reaches the disk. The disk will not contain a valid version of the object after the crash. A cell of the object has been overwritten with a forwarding pointer, and is not available on the backing store for from-space. Neither is it available on the backing store for to-space.

The example in Fig. 2 also shows that not all forwarding pointers are valid after



2.a: Virtual Memory Just Before Crash



2.b: Backing Store After Crash

Fig. 2. Lost Object Descriptor

a crash; in Fig. 2.b there is no object at the forwarding address. If the collector were to be restarted after a crash, object A would never be recopied to to-space.

Approach. In our approach the atomic garbage collector runs below the level of user transactions and provides for its own recovery. (It cannot run as a series of user level transactions because that could lead to deadlock.) A user transaction consists of low-level read, update and allocate actions. To allow atomic garbage collection, we add two additional types of low-level actions: *copy* actions and *scan* actions. A copy action copies a single object from from-space to to-space. A scan action scans a single page of to-space. In Sect. 4.3 we show how to make these new actions recoverable using the redo protocol and in Sect. 4.4 how to handle their synchronization with read, update, and allocate actions.

Movement of Objects. A copying garbage collector moves objects. However, the recovery system records values for objects in the log and these values contain the names of other objects. This leads to a naming problem: what names should the recovery system use in its log to refer to objects? There are two requirements on the solution: (1) the identity of an object must be preserved across garbage collections, and (2) the recovery system needs to find objects on disk after a failure.

Approach. There are two approaches to solving the naming problem: (1) the unique object identifier (UID) approach and (2) the virtual address approach. When an object value is written to the log using the UID approach, the pointers in it to other objects are replaced by the UIDs of those objects. For fast recovery from system crashes, the UID approach requires that a map of UIDs to virtual addresses also be available (either in the log or in virtual memory) so that objects on disk can be found.

In contrast, when an object value is written to the log using the virtual address approach, pointers in it to other objects remain unchanged. For recovery from media failure, the virtual address approach requires that translation information relating object addresses before a garbage collection to addresses after a collection be written to the log. Depending on the recovery algorithms, some of this translation information may also be required for system crashes.

4.3 Making Copying Garbage Collection Atomic

After a failure during garbage collection, it is sufficient that the system be able to complete the interrupted garbage collection. This requires that the recovery system be able to restore the heap to a state from which the garbage collection can be completed. This is exactly what the redo protocol does; its repeating history invariant guarantees that the state reached by applying the redo log to the disk is an actual state from the history of the system.

A copying collection consists of copy and scan actions. We show how to apply the redo protocol to make these actions recoverable.

Copy Action. A copy action makes two modifications to memory: it inserts a forwarding pointer in an object in from-space, and it copies that object to to-space. Thus a naive application of the redo protocol would pin at least two pages (the from-space page of the forwarding pointer and the to-space page of the copied object) and it would write two redo records to the log. The redo record describing the modification to to-space would contain the value of the object, so that the whole object graph would be written to the log during the course of a collection.

To optimize the copy action we require that the recovery system be able to recover from-space after a crash to its state at the last flip, except for the from-space cells overwritten by forwarding pointers. The recovery system described in Sect. 3 satisfies this requirement. In addition we ensure that the cells overwritten by forwarding pointers are recoverable from the redo information we write to the log for the copy. Since transactions do not modify from-space after a flip except for the forwarding pointers, we can redo a copy by re-copying the object from from-space to to-space and taking the cell overwritten by the forwarding pointer from the log. Below we describe this optimized copy action; it requires only one page to be pinned and writes small records to the log. Then we argue its correctness.

Here is the optimized copy action.

1. Pin the from-space page of the object cell that will be overwritten by the forwarding pointer.
2. Copy the object to to-space and insert a forwarding pointer in its from-space copy.
3. Spool a *copy record* to the log. The copy record contains the from-space address of the object, the to-space address of its copy, and the contents of the cell overwritten by the forwarding pointer.
4. The copy action is over and the collector can continue.
5. When the copy record is physically in the log, unpin the from-space page.

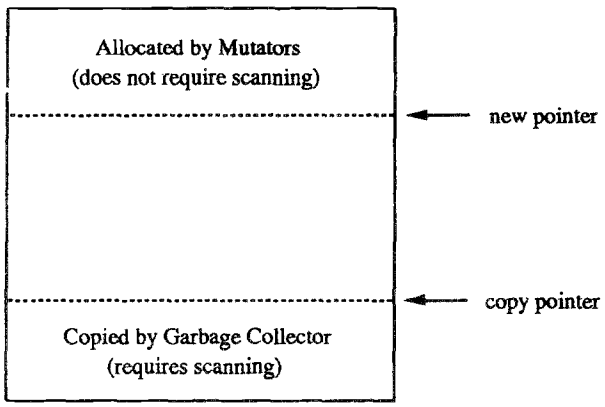


Fig. 4. Layout of To-space

to the argument: (1) the forwarding pointer in from-space is recovered correctly, and (2) the object in to-space is recovered correctly. The first part is trivial because the redo protocol guarantees correct recovery of the forwarding pointer.

To argue the second part, we rely on the repeating history invariant. After the flip, the only modification to the from-space copy of an object is the insertion of a forwarding pointer by a copy action of the collector. Thus, if we apply the redo log to the disk (i.e., repeat history) up until the point in the log where the flip occurred, the from-space object is recovered to the same state as at the time of the flip except for the cell containing the forwarding pointer. As we continue to repeat history from the time of the flip until the copy record, the from-space object does not change. Thus, when we redo the copy, the correct value of the object is in from-space except for the cell overwritten by the forwarding pointer. We recover the value for this cell from the copy record.

Because of the dependence on from-space for the value of an object that needs to be recopied, disk storage for the from-space object cannot be discarded until the to-space copy has reached disk. This means that the disk storage for from-space must be kept until both (1) the garbage collection is complete, i.e., all accessible objects have been copied to to-space, and (2) every to-space copy has reached disk. The discussion of the scan action will show that this condition must be made even stronger.

Scan Action. A scan action scans a unit of to-space, looks for pointers into from-space, and converts them into to-space pointers. The unit could be a single location, an object, or a page. The page is the best unit for two reasons: (1) it is the unit updated on disk atomically, so it is the natural unit of recovery, and (2) it is the unit of synchronization for Ellis's incremental garbage collection algorithm. Choosing a unit larger than a page would make the collector less incremental.

A scan action is made recoverable using the redo protocol. The redo protocol prevents the problem of lost forwarding pointers mentioned in Sect. 4.2, i.e., it prevents a to-space pointer to a copied object from reaching disk until the forwarding pointer

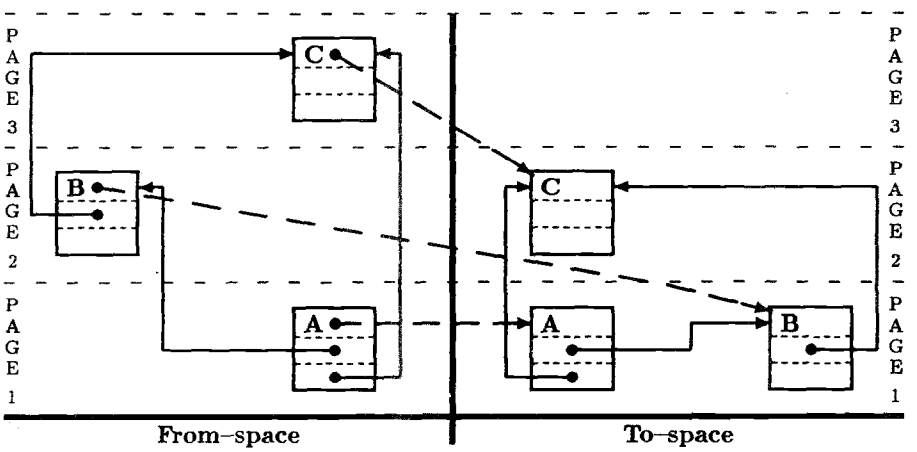
for the object is recoverable. However, a naive application of the redo protocol writes a redo record containing the entire scanned page and would lead to the entire object graph being written to the log, as in the case of the unoptimized copy action.

We optimize the scan action in two stages. First we replace its redo record by a scan record that contains just the address of the scanned page. Then we eliminate the scan record altogether.

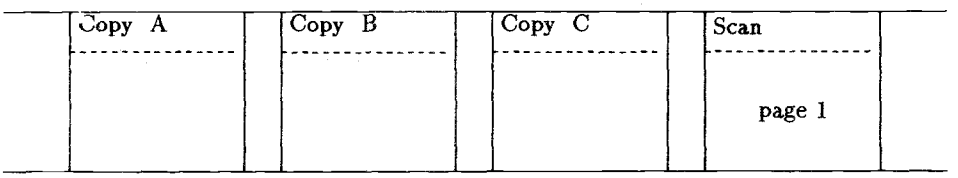
First Optimization. Here is the optimized scan action.

1. Pin page to be scanned.
2. Scan page and update its from-space pointers to to-space pointers, copying objects as necessary using copy actions.
3. Spool a scan record containing the address of the page.
4. The scan action is over and the collector can continue.
5. Unpin the page.

Figure 5 shows the effects of a scan action. Figure 5.a shows virtual memory after page 1 has been scanned. Figure 5.b shows the scan record for page 1 in the log. Notice that copy records for objects A, B, and C precede the scan record.



5.a: Virtual Memory



5.b: Log Showing Scan Record

Fig. 5. The Scan Action

The address of the scanned page is sufficient to make the scan action repeatable. By the time the scan record has been written to the log, there are copy records in the log for all objects referenced by pointers on the scanned page. That means that when the redo log is applied to the disk, all of these objects will be copied to the same place in to-space, and their forwarding pointers will be the same as when they were first copied. Therefore when the page is re-scanned, all of its pointers will be assigned the same values that they were assigned during the first scan.

Since the repeatability of the scan action depends on forwarding pointers in from-space, from-space cannot be discarded until both the garbage collection is complete (every page of to-space has been scanned), and every scanned page has reached disk. The writing of scanned pages to disk does not need to occur synchronously; rather, the garbage collector informs the buffer page manager as it scans each page. Then the buffer page manager can schedule the writes according to its own policies. The buffer page manager must also provide an operation that allows the collector to check if all of the scanned pages have been written.

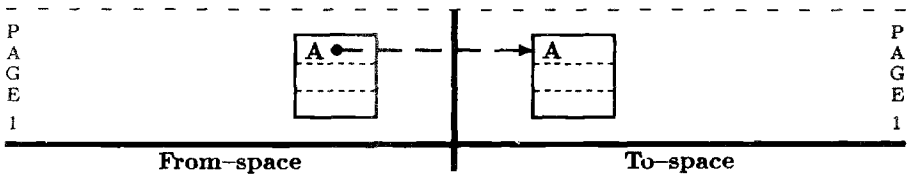
Second Optimization. As a further optimization, we eliminate the spooling of the scan record in Step 3 above. In the absence of a scan record, we unpin the scanned page when copy records for all of the objects copied during the scan are in the log. In addition the buffer page manager includes an indication that the page has been scanned in the next end-write record it writes for the page.

For repeatability of the scan we depend on the copy records for objects copied during the scan, the redo record for the first update to an object on the scanned page, and the end-write record. The read barrier ensures that the redo record for the update action will not be written to the log until the page of the updated object has been scanned, and the scan action ensures that the scanned page will not reach disk before the copy records on which it depends are in the log. Thus, to repeat history we repeat the scan of the page just before redoing the first update to the page. If there is an end-write record for a scanned page, we avoid repeating the scan because the end-write record indicates that the page reached disk.

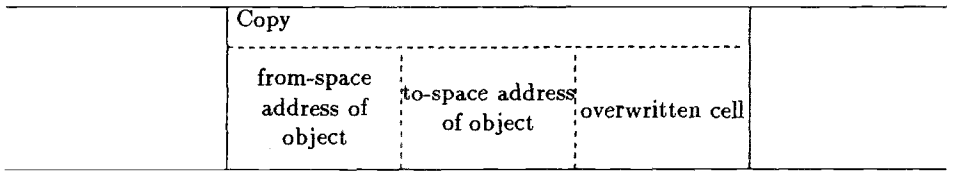
Scanning An Arbitrary Page. To allow the scanning of an arbitrary page of to-space after a crash, the Last Object Table must also be recoverable. Fortunately, the copy records contain sufficient information to recover the table. The collector keeps the table in the stable heap. When repeating history the copy records act as redo records for the table.

Movement of Objects. The information written to the log to solve the modification problem solves the naming problem caused by the movement of objects. Each copy record contains a from-space, to-space address pair for an object. There is one copy record for each accessible object. This is precisely the translation information required by the virtual address approach to solving the naming problem.

Size of Copy Records. Assume that a pointer takes up four bytes. That means that the overhead for a copy record is greater than 12 bytes, four bytes for each component plus the extra overhead to delineate the record and indicate its type.



3.a: Virtual Memory



3.b: Log Showing Copy Record

Fig. 3. The Copy Action

Figure 3 illustrates the copy action.

First we show that the pinning optimization is correct, i.e., that only a single page has to be pinned. It is easy to ensure that the cell in from-space overwritten by the forwarding pointer resides entirely on a single page — always overwrite the first cell of the object and during allocation make sure that the first cell of an object never crosses page boundaries. Thus, only one from-space page needs to be pinned even if the object spans more than one page.

The to-space pages to which an object is copied do not need to be pinned. The redo protocol pins pages to prevent partial modifications from reaching disk. However, we can mask partial modifications to to-space by the copy action without pinning pages. Following Baker we assume that the collector copies objects contiguously to the low part of to-space and allocates by adding to a *copy pointer*; the mutator allocates objects contiguously in the high part of to-space and allocates by subtracting from a *new pointer* (illustrated in Fig. 4). Since the collector writes copy records to the log in the order in which it copies objects, the copy pointer can be recovered after a crash by looking at the last copy record in the stable log. Any partial modifications to to-space by copy actions whose copy records did not reach the log are at addresses greater than the copy pointer. Thus, these modifications will never be observed.

Next we show how to repeat the copy action using the information in the copy record.

1. Using the from-space address of the object and the to-space address of the object from the copy record, reinsert the forwarding pointer in from-space.
2. Re-copy the object from the from-space address in the record to the to-space address.
3. Take the contents of the cell overwritten by the forwarding pointer from the copy record and write it to its place in the to-space copy of the object.

Finally we argue that the repeat of the copy action is correct. There are two parts

This is high for small objects such as a cons cell in Lisp or a variant in Argus (both are eight bytes). The importance of this issue depends on two factors: (1) the frequency of small objects compared to larger objects, and (2) the overhead of writing to the log. If there are few small objects, the total space taken up by copy records in the log will be minor. If the overhead for writing is low, the extra overhead for copy records will be tolerable.

A further optimization would reduce the log space taken by copy records: a *block copy action* copies a group of objects consecutively to to-space and writes a single *block copy record* to the log. To satisfy the redo protocol, the action pins all of the from-space pages modified by the objects' forwarding pointers until the record is in the stable log. The record contains the number of objects copied, the from-space address and old contents of the cell overwritten by the forwarding pointer for each object copied, and the to-space address of the first object copied. By re-copying the objects in the same order after a crash, recovery can deduce the to-space addresses of the remaining objects. A natural unit of blocking would be all of the objects copied during the scan of a single to-space page.

4.4 Other Interactions With Recovery

In the previous sections we described the modification and movement problems and our solutions to them. Here we describe three other interactions between garbage collection and recovery: (1) synchronization between the garbage collector, the transaction system and recovery, (2) roots in the recovery information, and (3) fast recovery for a system failure during garbage collection.

Synchronization. Because the collector moves and modifies objects it must be synchronized with the transaction system. Since it writes recovery information to the log, it also has to be synchronized with the recovery system. This synchronization has to be cheap. First we describe synchronization with transactions; then we describe synchronization with the recovery system.

Synchronization With Transactions. Each transaction is a sequence of elementary actions that read, update, and allocate individual objects. Our approach to synchronization with transactions is to require that a flip occur in an action-quiescent state. The system is action-quiescent when no elementary action is in progress. Since the elementary actions are short, a flip is not significantly delayed by waiting for an action-quiescent state. Given that a flip occurs in an action-quiescent state, the read barrier ensures that a copying or scanning action of the atomic incremental garbage collector only observes an object in an action-consistent state.

The correctness of the above approach can be seen by viewing the system as a multi-level transaction system with two levels [36]. At the high level, there are user transactions; at the low level, there are elementary actions: update, read, allocate, copy, and scan. A user transaction is made up of a sequence of read, update, and allocate actions. The garbage collector uses copy and scan actions.

At the level of transactions, a transaction obtains read and write locks that it holds for its duration. These locks ensure serializability at the transaction level.

At the level of actions, a synchronization mechanism ensures that only one action accesses a given object at a time, so that an action always observes a consistent object state. Since the garbage collection actions do not change the abstract values of objects, they are invisible to the transaction level: an object can be copied or scanned even while a transaction holds a write lock.

In the approach described above, synchronization between garbage collection actions and the other low-level actions is implemented cheaply by the read barrier. At a flip the collector obtains a "lock" for every stable object at once by protecting the unscanned pages of to-space. As each page of to-space is scanned, the "locks" for the objects in it are released by changing its protection. Restricting flips to occur in an action-quiescent state ensures that the collector obtains the "locks" at an appropriate time.

Synchronization With the Recovery System. Because the collector writes to the log, we must ensure that a garbage collection trap cannot occur inside a procedure of the recovery system when it is in the middle of writing a record to the log. If it did, the log would not be readable after a crash. A general solution for this problem is to have the recovery system construct an entire record in memory outside of the heap and then copy the record to the log buffer which is also outside of the heap. Clearly no trap can occur while the record is being copied.

Roots for Garbage Collection in Recovery Information. Because the collector runs while transactions are active, it must be careful to account for the modifications made by active transactions to ensure that no objects are lost. For example, suppose a stable object A contains a pointer to object B, and that B is not accessible from any other object. Now suppose that a transaction T modifies A to point instead to some object C. If T aborts, the pointer to B should be restored, while if T commits, the pointer to C should be installed permanently, and B becomes garbage. Suppose a collection takes place after T has modified A, but before it commits or aborts. If T modified A directly, and the collector does not look at undo information, the storage for B will be reclaimed. If T then aborts, there is no way to restore the heap to its original state.

To solve this problem, the collector must use the redo and undo information maintained by the recovery system for active transactions in determining which objects are accessible. An object must be considered accessible if (1) it is directly accessible from the stable root; (2) it is directly accessible from undo or redo information for an active transaction that has modified some other accessible object; or (3) it is accessible from some other accessible object.

Since our system updates objects directly, the latest redo information for an object is reflected in the state of the object in the heap, so the collector can ignore redo information. However, the collector cannot ignore undo information in the log records for active transactions. Reading the log during collection to find the records for active transactions can be expensive. We show how to avoid this expense in the first author's dissertation [20].

Fast Recovery Even if a Crash Occurs During Garbage Collection. Even if a crash occurs in the middle of a collection, the time for recovery should be short

and independent of heap size. Fast recovery depends on the checkpoints and other optimizations of the recovery system.

The copy and scan actions of the atomic incremental garbage collector have been made atomic using the redo protocol. Therefore the optimizations that work for repeating history and the redo protocol, checkpoints and end-write records, continue to work for the atomic garbage collector. By increasing the frequency of checkpoints, and the frequency at which dirty pages are written back to disk, the time for recovery can be shortened.

In particular, if there is an end-write record in the log for a scanned page, that page does not have to be rescanned during recovery. If there is a copy record for an object in the log and there is also an end-write record for the page of from-space holding the object's forwarding pointer, the forwarding pointer does not have to be reinserted during recovery. If there is a copy record for an object in the log and there is also an end-write record for the page(s) of to-space to which the object has been copied, the object does not have to be recopied during recovery.

5 Other Recovery Systems

In this paper we showed how to make an incremental collector atomic by dividing its work into copy and scan actions. We made the copy and scan actions recoverable using the redo protocol. In Section 3 we described an approach to recovery based on repeating history. Our collector was designed with this recovery system in mind, but the collector can also be incorporated into other recovery systems. We summarize the constraints imposed by our collector on the recovery system by stating some of the invariants that it must maintain.

1. From-space can be recovered to its state at the time of the last flip (using the disk and the log), with the exception of the cells that were overwritten by forwarding pointers.
2. The cell that gets overwritten by a forwarding pointer is on the disk for from-space or it is in a copy record in the log.
3. The disk backing store for from-space is available until the garbage collection is complete and every scanned page has reached disk.

In the first author's dissertation [20] we provide a complete description of a recovery system that meets these constraints.

6 Conclusion

A major goal in the design of any computer system is to achieve the required functionality at the minimum cost. For recovery systems, this means that we would like to pay a minimal run-time cost in order to ensure that the state can be recovered quickly after a crash. The repeating history recovery algorithm and its accompanying redo protocol have been designed with this goal in mind.

To keep the run-time cost of recovery low, designers of recovery systems

1. avoid random synchronous writes to disk by writing to a log,

2. avoid synchronous writes to the log except for transaction commit,
3. and minimize writing to the log.

By basing the atomic garbage collector on the redo protocol, we avoided synchronous I/O to disk and the log. Then we optimized the protocol for the copy and scan steps to minimize writing to the log.

Our atomic garbage collector never delays itself or transactions because it is waiting for the completion of a synchronous write to disk or to the log. In comparison, Detlefs [11] focused on minimizing writing to the log; as a result his concurrent atomic garbage collector requires both random synchronous writes to the disk and synchronous writes to the log.

We have completed the implementation of a stable heap prototype. In the future we plan to measure the prototype. Unfortunately, the Argus implementation is itself a prototype and has few users. Thus, we would also like to incorporate our atomic incremental collector into an object-oriented database (OODB). An OODB will provide a better platform for testing our algorithm.

References

1. A. Albano, L. Cardelli, and R. Orsini. A Strongly Typed Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.
2. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.
3. Henry Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.
4. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, Ma., 1987.
5. Peter B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1977.
6. Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, 1977.
7. Alfred Brown and John Rosenberg. Persistent Object Stores: An Implementation Technique. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice/ The Fourth International Workshop on Persistent Object Systems*, pages 199–212. Morgan-Kaufmann Publishers, San Mateo, California, 1990.
8. M. Carey, D. DeWitt, J. Richardson, and E. Sheikta. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Databases*, August 1986.
9. Robert Courts. Improving Locality of Reference in a Garbage-Collecting Memory Management System. *Communications of the ACM*, 31(9):1128–1138, September 1988.
10. David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, 21(12), December 1988.
11. David L. Detlefs. Concurrent, Atomic Garbage Collection. Technical Report CMU-CS-90-177, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., October 1990.

12. Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.
13. John R. Ellis, Kai Li, and Andrew W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Technical Report 25, Systems Research Center, Digital Equipment Corporation, Palo Alto, Ca., February 1988.
14. D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *Database Engineering*, 8(2):63–70, June 1985.
15. James N. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems–An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, New York, 1978.
16. Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.
17. Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987.
18. Elliot Kolodner, Barbara Liskov, and William Weihl. Atomic Garbage Collection: Managing a Stable Heap. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 15–25, June 1989.
19. Elliot K. Kolodner. Recovery Using Virtual Memory. Technical Report MIT/LCS/TR-404, Laboratory for Computer Science, MIT, Cambridge, Ma., July 1987.
20. Elliot K. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. Technical Report MIT/LCS/TR-534, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1992.
21. Butler. W. Lampson. *Atomic Transactions*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, New York, 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System.*
22. Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.
23. B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. Notes on Distributed Databases. Technical Report RJ2571, IBM Research Laboratory, San Jose, Ca., July 1979.
24. Barbara Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1984.
25. Barbara Liskov, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987.
26. David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, pages 472–482, November 1986.
27. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. Technical Report RJ6649, IBM Almaden Research Center, San Jose, Ca., January 1989.
28. David Moon. Garbage Collection in a Large Lisp System. In *Proc. of the 1984 Symposium on Lisp and Functional Programming*, pages 235–246, 1984.
29. Brian Oki, Barbara Liskov, and Robert Scheifler. Reliable Object Storage to Support Atomic Actions. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 147–159, December 1985.

30. Richard F. Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, August 1986.
31. Mark Reinhold. Personal communication.
32. Alfred Z. Spector, J. J. Bloch, Dean S. Daniels, R. P. Draves, Daniel Duchamp, Jeffrey L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot Project. *Database Engineering*, 9(4), December 1986.
33. Satish M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In U. Dayal and K. Dittrich, editors, *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, CA, September 1986.
34. David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.
35. William Weihl and Barbara Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.
36. Gerhard Weikum. A Theoretical Foundation of Multi-Level Concurrency Control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 31–42, Cambridge, Ma., March 1986.
37. Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1), 1991.
38. Daniel Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb. An Object-Oriented Database System to Support an Integrated Programming Environment. Submitted for publication, 1988.
39. Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, June 1991.
40. Stanley Zdonik and Peter Wegner. Language and methodology for object-oriented database environments. In *Proceedings of the 19th Annual Hawaiian Conference on Systems Science*, January 1986.
41. Benjamin G. Zorn. Comparative Performance Evaluation of Garbage Collection Algorithms. Technical Report UCB/CSD 89/544, Computer Science Division (EECS), University of California, Berkeley, California, December 1989.

This article was processed using the LaTeX macro package with LLNCS style

Incremental Collection of Mature Objects*

Richard L. Hudson[1] and J. Eliot B. Moss[2]

[1] University Computing Services
University of Massachusetts
Amherst, MA 01003, USA
hudson@cs.umass.edu
[2] Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
moss@cs.umass.edu

Abstract. We present a garbage collection algorithm that extends generational scavenging to collect large older generations (*mature objects*) non-disruptively. The algorithm's approach is to process bounded-size pieces of mature object space at each collection; the subtleties lie in guaranteeing that it eventually collects any and all garbage. The algorithm does not assume any special hardware or operating system support, e.g., for forwarding pointers or protection traps. The algorithm copies objects, so it naturally supports compaction and reclustering.
Keywords: clustering, compaction, copying collection, garbage collection, garbage collector toolkits, generation scavenging, incremental collection, mature objects, non-disruptive collection.

1 Introduction

Generational garbage collection is very effective at reducing total garbage collection time. The majority of the collections are also non-disruptive. However, as good as generational collectors are, they can still be disruptive when the larger, older generations need to be collected. To collect these older objects in a non-disruptive manner, we present an algorithm that has the following properties:

Incremental: The maximum number of bytes moved at each incremental collection is small.
Compaction and Clustering: The algorithm supports compaction and reclustering of objects via copying.
Efficient Implementation: The algorithm can be implemented on stock hardware and does not rely on operating system features such as protected pages.

Bishop [Bishop, 1977] discussed how related objects should be placed in the same area and references into these areas should be handled by a level of indirection so that each area could be collected independently of other areas. Our algorithm borrows heavily from his conceptual work describing areas that hold related objects. Our contribution is to show how this can be used to create a collector with the above characteristics.

* This project is supported by National Science Foundation Grant CCR-8658074 and by Digital Equipment Corporation and Apple Computer.

In Section 2 we discuss the problem of collecting older generations holding mature objects and review some recent work in the area. In Section 3 we give an overview of the collector. Section 4 presents our algorithm, while Section 5 shows an example of how the algorithm works. Section 6 describes how to extend the algorithm to handle a potential problem. Finally, Section 7 discusses some future extensions to support additional techniques that mitigate the disruptive behavior of garbage collection.

2 The Problem and Some History

This paper addresses the problem of collecting older objects incrementally, in the context of a copying, scavenging collector. We insist on copying in order to support compaction and clustering (e.g., hierarchical decomposition [Wilson *et al.*, 1991]). Because copying collectors move objects, we assume the safety property that all pointers (and pointer derived quantities) can be found and updated appropriately (see, e.g., [Diwan *et al.*, 1992]).

Pieces of the problem of doing garbage collection non-disruptively have been worked on for years. In this section, we will review some of these attempts and discuss some of their drawbacks.

Baker [Baker, 1978] discussed the problem of constructing a non-disruptive garbage collector. His solution was a modification of a stop-and-copy algorithm first discussed by Fenichel and Yochelson [Fenichel and Yochelson, 1969]. Baker used a read barrier that trapped all reads of old objects and then copied the objects or updated the pointers to moved objects. White [White, 1980] suggested that collecting unreachable objects was not as much of a problem as improving the locality of reference of live objects, and proposed a scheme that improved locality of reference of running programs but that collected unreachable objects off-line. Both Baker and White assumed special pointer forwarding hardware support for their algorithms.

Lieberman and Hewitt [Lieberman and Hewitt, 1983], Moon [Moon, 1984], and Ungar [Ungar, 1984] all presented algorithms that reduced the running time required by most garbage collections by focusing attention on the youngest and most volatile generations of objects. Lieberman and Hewitt relied on special hardware and a Baker-style algorithm to achieve incremental performance. Moon also relied on the Lisp machine hardware to provide a read barrier. Ungar was concerned only with young objects, and collected older objects "off-line". This work made the time to perform most garbage collections reasonable and the majority of collections non-disruptive. The drawback was the large cost and disruption when large old generations needed to be collected.

Appel, Ellis, and Li [Appel *et al.*, 1988] suggested collecting on stock hardware by using read-protected or no-access pages in older generations: when a page is touched, it is scanned and all pointers to moved objects are updated. For efficiency their algorithm depends on two properties. First, the algorithm requires a fast protection fault reflection mechanism. Providing such a fast mechanism may require modifying the operating system. Second, the algorithm requires high locality of reference in the application being run. Without this property, the scanning resulting from touching several pages shortly after a collection would make collection effectively disruptive.

Boehm [Boehm *et al.*, 1991] showed how collectors could be made "mostly parallel" in the trace phase of a collector. His algorithm also relies on using the page trap hardware and operating system support to do bookkeeping during the mark phase. This choice reduces the amount of mutator cooperation needed.

The Lisp Machine [Weinreb and Moon, 1981] demonstrated how linked lists could be compacted using cdr-coding. Wilson [Wilson *et al.*, 1991] showed how hierarchical decomposition could also be used to compress data, in addition to improving locality of reference. Wilson's scheme is similar to Moon's "approximately depth-first" algorithm [Moon, 1984] and demonstrates the gains in locality that can be made by reclustering items based on their reachability path characteristics. Unfortunately, mutation of objects requires periodic reclustering. To allow this reclustering and compaction, the objects need to be moved and pointers to the moved objects updated appropriately.

Lang [Lang and Dupont, 1987] showed how the incremental compaction of a large heap can be done using a hybrid mark/sweep and copying collector. The algorithm copies as much of the heap as there is contiguous free space during each collection thus compacting some portion of the heap. The remaining live objects are not copied. Dead objects in areas where live objects were not evacuated are marked and placed on a free list along with the large evacuated area. This process incrementally continues until the entire heap is compacted. This algorithm requires that all live objects be inspected and possible updated during each pass of the collector. Such romping through memory becomes disruptive as the heap grows large enough to affect the cache and virtual memory mechanisms.

Wilson [Wilson and Moher, 1989] tries to make his collector non-disruptive using temporal opportunism, a technique that tries to hide long garbage collection by piggy-backing onto long computations or onto long interactive pauses. Hayes [Hayes, 1991] suggested key object opportunism, which monitors key objects. When a key object become unreachable, one attempts to collect the objects associated with it. By using the key object as an indicator of when a group of objects become unreachable, the collector focuses its attention on a group of objects that are likely to be unreachable. While temporal opportunism uses hints about *when* to do collections, key object opportunism adds hints about *where* to do collections.

Bishop [Bishop, 1977] presented a garbage collection algorithm that divided the heap into multiple areas. Users specified the area in which each object was allocated. These areas were designed to be garbage collected individually. By collecting the areas independently, the collections would not interfere with processes that did not use the area being collected. In order to allow independent collection, each area kept track of pointers both into the area and out of the area. Referencing an object in another area was accomplished using a level of indirection.

Bishop pointed out that related areas could be collected at the same time. He handled multiple area cycles of garbage either by collecting all areas involved in the cycle at the same time, or by using copying to consolidate the cycle of objects into one area. In his thesis, Bishop presented an inductive proof to show that his technique of moving objects guarantees that all unreachable objects are collected.

Bishop did not bound the size of an area or provide ways to collect individual areas incrementally. In addition, his use of levels of indirection to communicate between areas was a source of inefficiency.

Our mature object space algorithm does not require special hardware or special operating system support, and it is not disruptive. It insures that all reachable objects are collected, that they are moved in a manner consistent with compaction and clustering algorithms, and that they are available immediately after each collection. Our algorithm also limits area size, provides ways to collect individual areas incrementally, and eliminates levels of indirection between areas.

3 Overview of the Garbage Collector

To collect young objects, we designed a garbage collection toolkit that supports generational scavenging techniques.[3] Our algorithm for collecting mature objects is an extension of this toolkit. We now offer an overview of the toolkit as a basis for explaining the extensions.

3.1 The Toolkit Concept

The toolkit divides the responsibility for, and support of, garbage collection into two parts: a language-independent part, supplied by the toolkit, and a language (implementation) specific part, nominally supplied by the language implementor. The language-independent part consists mostly of the data structures and code for managing multiple generations and for allocating heap objects. The language implementor must supply the following capabilities: the ability to locate at scavenge-time all *root pointers* (those pointers outside the scavenged generations that refer to objects in the scavenged generations), and the ability to locate all pointers within a heap object, given a pointer to the object. The toolkit includes a library of routines that an implementor can use to locate inter-generational pointers; it is the implementor's responsibility to locate roots lying in the stack(s), registers, and any other areas outside the heap.

3.2 The Structure of the Heap

The toolkit defines the structure of the heap and supplies the necessary allocation routines. The heap consists of a number of *generations*. Generations are numbered 0, 1, 2, ..., in order of increasing age. In any given collection, a selected generation and all younger generations will be scavenged. The total number of generations may vary over time.

Each generation consists of a number of *steps*. Steps segregate objects by age and/or type within a generation, and during scavenging all surviving (reachable) objects in a given step are copied to some other step. This *promotion step* may belong to the same or a different generation, and by adjusting the promotion steps before scavenging, one can introduce new steps, combine existing steps, etc. The number of steps in a generation may vary over time.

A primary function of steps is to eliminate the need for storing or maintaining any age information in individual objects. This reduces storage and time costs, but also gives the collector age information without imposing any requirements on object formats (which are entirely the responsibility of the language implementor).

While the meaning of steps is somewhat arbitrary, we impose a convention that the lowest numbered step in a generation has the youngest objects in that generation, etc. Further, we number the steps 0, 1, 2, ..., such that every step in the system has a unique number. For example generation 0 might have steps 0 and 1, generation 1 might have steps 2 through 4, and so on. A simple promotion policy is to promote survivors of step k to step $k + 1$. In that case, the number of steps in a generation determines the number of scavenges (of that generation) necessary to promote objects to the next generation.

Each step consists of a number of *blocks*. A block is 2^n bytes, aligned on a 2^n byte boundary for some value of n chosen when the system is built. A typical block size might be 64K bytes. The number of blocks in a step may vary over time. While the blocks of a step

[3] For a more detailed discussion of the toolkit see [Hudson *et al.*, 1991].

are usually not contiguous, a *nursery* may be set up to consist of a number of contiguous blocks, so that one might more readily use a page trap (rather than an explicit limit check) to detect nursery overflow and trigger a scavenge.

Blocks have four primary advantages. First, they allow sizes of steps and generations to change easily since the storage of a step need not be contiguous. Second, they allow speedy determination of the generation, step, and promotion step of an object: the address of the object is simply shifted right by n bits and indexes a block table containing the needed information. Third, blocks match naturally with page trapping or card marking schemes (both of which the toolkit supports). Fourth, they reduce the storage needed under some circumstances when compared with copying collectors that use semi-spaces. If b bytes are present in a generation before a scavenge and the survivors consume a bytes, then a semi-space scheme uses $2 \times b$ bytes whereas our scheme uses $b + a$ bytes (modulo rounding resulting from the block size). The degree of advantage depends on the survival rate a/b, but may be significant in some applications.

Blocks do introduce a problem, however. They cannot handle objects larger than the block size. To handle such objects we provide a *large object space* (LOS), as suggested in [Ungar and Jackson, 1988]. In fact, it is probably a good idea to put into LOS any object that consumes a significant fraction of a block; we used the heuristic threshold of 1/8 of a block. Further, as also discussed in [Ungar and Jackson, 1988], any object that contains few pointers and that exceeds some threshold in size should be stored in LOS to avoid the overhead of copying. LOS uses free list allocation based on splay trees [Sleator and Tarjan, 1983, Sleator and Tarjan, 1985, Jones, 1986] and, once allocated, an LOS object is never moved. However, LOS objects still belong to a step, which is indicated by threading the objects onto a doubly linked list rooted in the step data structure. When a LOS object is promoted, we simply unchain it from one list and chain it into another. When scavenging is complete, any LOS objects remaining on a scavenged step's LOS list are freed.

While the generation, step, and block, of a non-LOS object can be determined using the simple shift and index technique, LOS may combine objects from different steps and generations in the same block. Therefore, we store a back reference from a LOS object's header to its containing step. It is relatively easy to determine the step given a pointer to the base of an LOS object, but determining the step given a pointer into the middle of the object requires locating the object header, which is supported but involves additional work.

3.3 Phases of a Scavenge

A scavenge consists of two phases. First, the root set for the scavenge is determined based on the remembered sets, as well as the stack, register, and global variable contents. All objects directly reachable from the roots are copied into new space, and the roots updated to point to the copied object. All objects reachable from the new space objects are then copied over using a non-recursive Cheney scan [Cheney, 1970].[4] As each object is copied, a forwarding pointer is left in the old copy, so that other references to the object can be updated as they are encountered. Since the toolkit makes no assumptions about object format, language implementors can define the details of the forwarding pointer format. The toolkit does

[4] The toolkit might be adapted to support mark-sweep or other approaches to collection, but currently it provides only copying collection. Also, it would not be hard to incorporate suggestions such as hierarchical clustering [Wilson *et al.*, 1991].

determine automatically where to allocate the new copy of the object, given the object's size (which must be determined by language-specific code).

Before a scavenge begins, the toolkit, following a dynamically modifiable plan supplied by the language implementor, determines the generations to be scavenged and creates new steps accordingly. It also sets up all the promotion step references. After a scavenge, all the old steps of the scavenged generations are deleted and their blocks become available for allocation.

These scavenge techniques work well for small heaps. In large heaps, however, scavenging older and older generations along with all younger generations becomes disruptive. In order to avoid this disruption we limit the number of generations in the heap. Any object that lives through several scavenges is moved into *mature object space* (MOS) where it is collected using our non-disruptive algorithm.

4 The Mature Object Space Algorithm

We now describe mature object space, its structure and its collection algorithm. The components used to implement this algorithm are the same as those used to implement the garbage collection toolkit for young objects. In particular, the blocks, the remembered sets, and the scanning and copying mechanisms are the same for both mature object space and generational space.

First, we will describe how mature space is divided into areas. Second, we will discuss the remembered set mechanisms used to track pointers between mature space areas. Third, we will present the rules that determine where mature objects are placed. Fourth, we will show how collection of an area results in objects being moved so that any unreachable object is eventually isolated and collected.

4.1 The Structure of Mature Object Space

Mature object space is divided into *areas*, just as young object space is divided into generations. The structure of an area is similar to a generation in that all pointers into an area can be found at scavenge time (i.e., each area has a remembered set). An area consists of one or more blocks. These blocks are the same as the blocks used in young object space and share the same bookkeeping functions, including quick determination of the area in which the block resides, and during a collection, determination of the area to which an object should be copied. In addition, blocks support determination of whether a pointer should be recorded in a remembered set. Unlike generations in the heap, age information is no longer interesting, so areas do not have steps.

Unlike Bishop's areas, our areas are sized so that each individual area can be collected quickly. The collector works on one area at a time. The problem with a straightforward implementation of Bishop's algorithm with limited area size is that a multiple area circular structure might not be collected because local information is not sufficient to determine if an object is globally unreachable. Hence, just as in Bishop's approach, we must migrate a multi-area cycle of garbage into a single area in order to reclaim it. However, the limit on area size makes this impossible if the linked structure is larger than can fit into a single area. Since Bishop did not restrict the size of areas, he did not have this problem. Hence, the key contribution of our algorithm is insuring that we reclaim large structures of garbage while still imposing the limit on area size, so that collections will be non-disruptive.

To further describe the structure of MOS, we first introduce some terminology. Pointers to mature objects from outside mature object space are *root pointers*. Root pointers reside in young object space, large object space, on the stack, in registers, and in static areas. Objects immediately reachable from roots are *leaders*. Objects that are not immediately reachable from roots, but still reachable from objects in mature object space, are *followers*.

We will use a train metaphor to describe the algorithm. An area can be thought of as a railroad *car*. The cars are used to bound the amount of work that is done during each invocation of the collector. A group of cars holding a linked structure of objects can be thought of as a *train*. Trains are used to group large related objects so that they can be managed as a unit.

4.2 Roots and Remembered Sets

Each area has an associated remembered set, which allows us to find all pointers from outside the area that refer to objects in the area. However, since we will scavenge an area only when all young spaces are also scavenged, and since all scavenges process all roots (stack(s), registers, static areas), remembered sets for areas need only track references from other MOS areas. The remembered set for a *train* is simply the union of the remembered sets of its cars, less any intra-train references.

A more subtle remembered set property comes from the fact that the algorithm processes areas in round-robin order. To understand this, suppose we assign each area a sequence number, and when an area is scavenged, it is assigned the next highest number. Then a remembered set need only record references from higher numbered to lower numbered areas. When an area is collected, its number will be the lowest, and hence we will be able to find all the references from other areas into the collected area.

We gain two advantages from handling the remembered sets this way. First, we reduce the total volume of remembered set information. If pointers are evenly distributed in terms of the direction they point, the remembered sets would be half as big, but it is not clear that the algorithm leads to such distributions, so the magnitude of this benefit is unclear. Second, and perhaps more importantly, we do not have to update *other* area's remembered sets when an area is scavenged. This is because none of the scavenged area's information could possibly be recorded in the other area's remembered sets, since such entries would record pointers from lower-numbered areas to higher-numbered ones, which our directionality rule specifically does not record.

The toolkit leaves the structure of the remembered sets up to the language implementor. The toolkit does, however, provide several alternative implementations including remembering slots, objects, cards, or pages. See [Hosking *et al.*, 1992] for performance studies comparing the available techniques.

4.3 Collecting an Area in Mature Object Space

As previously mentioned, we process areas in round robin order, collecting one area (or car) upon each scavenge of MOS (which implies that all young generations are also scavenged at the same time).

There is a check that is always done before collecting a car: if there are no root pointers to the train whose car is about to be collected, then we check the train's remembered set. If the remembered set is empty, then the entire train can be reclaimed with no further effort. We

can readily enhance the remembered set bookkeeping to make the check efficient (though possibly inaccurate for one round robin cycle of scavenging): record with each car the number of extra-train references to objects in that car, and also keep a sum across all the cars (easily updated as cars are collected (removed) or added).

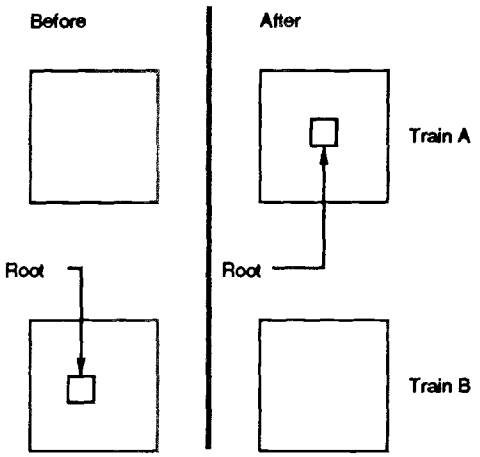


Fig. 1. Leaders in Train B are moved into another train.

When a car is collected we refer to it as a *from* car. Each reachable object in a *from* car has an associated *to* car which is determined by how the object is reached. First, we copy any objects in the car referred to by roots into either a new train or some *other* train (Figure 1). Which train we choose is a policy decision that does not affect the correctness of the algorithm. Next, we scan the copied objects and copy over, in typical copy collector style, all other objects in the *from* car reachable from objects in the other train.

At the same time we move objects being promoted from young generations into trains holding references to them, or if they are referred to by roots, into any train. Since the young generations are bounded in size, the volume of promoted objects is also bounded, so we can bound the disruption caused by promotions[5].

At this point the *from* car may still contain reachable follower objects, but they must be reachable only from other cars. Using the *from* car's remembered set, we locate all references from outside the train to objects still in the *from* car, and move them to the train containing the reference. See Figure 2.

The only remaining reachable objects in the *from* car are reachable from other cars in the same train. These objects are moved into the last car of the train as illustrated in Figure 3. This leaves only unreachable objects in the car. The space for these objects is then recycled.

This is similar to Bishop's approach. If the train to which we want to move an object is full, we add a car to that train and copy the object there. In any case, an object that is reachable from outside the train being collected is moved to some other train (thus collapsing garbage into fewer trains and eventually a single train), or (if unreachable) is reclaimed.

[5] Setting the size of young generations is a policy decision. The size can be limited by collecting young generations more often or by promoting more objects into mature space during each collection.

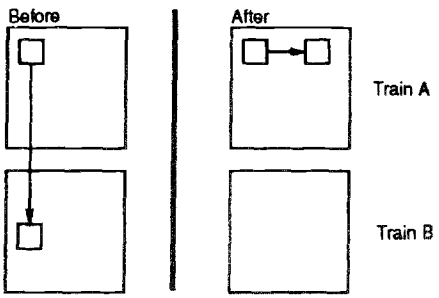


Fig. 2. Followers in Train B reachable from another train are moved there.

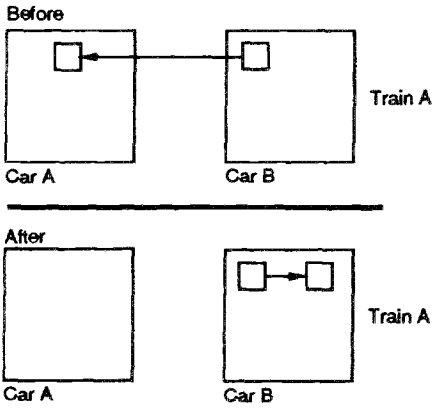


Fig. 3. Other followers are moved to the last car.

Of course, we scan moved objects, and evacuate any remaining reachable objects from the car. We collect cars in the order they were linked onto the train. Since we check to see if an object is referred to by another train *before* we check references inside the train, objects referred to directly from other trains will always moved out of this train. This is important since a train might contain a multi-area cycle of objects that "belongs in" another train, i.e., is not reachable from the leaders of this train.

The objects that are referenced during any one invocation of the algorithm are just those objects that are involved with the car where we are currently focused, either as a member of the car or by pointing into the car. This locality is known ahead of time so the algorithm will be able to provide the operating system hints about what locations it will need during the next cycle of the collector.

Since the algorithm periodically copies all reachable objects in mature space, it reclusters live objects at no additional cost. During the copying, we can apply sophisticated compaction and clustering techniques such as those described by Wilson [Wilson *et al.*, 1991]. In addition, this algorithm avoids the fragmentation that can occur with mark and sweep collectors.

4.4 Why the Collector Works

Having presented the collection algorithm, we now argue that it will eventually collect all unreachable objects, even large cycles of garbage. Suppose we have some garbage that threads through a number of trains. As we process the lowest numbered train, car by car, one of three things will happen to each object: it will be detected as garbage and reclaimed; it will be moved to another train; or it will be moved to another car of the same train (but not a car of the *new* train). The last case will not repeat indefinitely, since eventually we reach a situation where we have reclaimed or moved to other trains all objects reachable from roots or other trains, and the remembered set of the current train will be empty. By induction, then, in one round robin pass of the trains, the garbage structure will be compacted into a single train. Again, we see that as we process, car by car, eventually the train's remembered set will be empty and the garbage then reclaimed.

Each train pass may require objects to be copied several times to other cars in the train, but each pass through the cars in a train will reduce the number of objects since any object referenced from outside the train will be moved. By induction each pass through the cars on a train will either reduce the size of the train or reclaim the entire train.

One way to conceptualize the algorithm is as pulling different threads or chains of objects apart, until garbage is isolated and then reclaimed. Of course, smaller garbage structures are reclaimed sooner and with less copying, but the point is that the algorithm is guaranteed to reclaim garbage in a train or evacuate it into another train within $O(n^2)$ car collections, where n is the number of cars in the train. Since pieces of garbage structures can not be copied back into a train from which they were evacuated, the algorithm takes at most one pass through the trains to collect a garbage structure[6] while retaining the desired non-disruptive, incremental behavior.

5 An Example

The next several figures illustrate a simple example of how the algorithm works. For simplicity we will assume the maximum number of objects that can fit in a car is 3. This means that any given invocation of the collector will move at most three objects.

In Figure 4 we show three data structures. One structure, consisting of objects R, S, and T, is reachable from a root. The structure consisting of object A and B is circular garbage spanning two trains. The other structure, consisting of objects C, D, E and F, forms a large circular structure of garbage that can not fit into one car. We will show how the structure C-D-E-F is isolated and freed and how A-B is consolidated and freed.

We start by applying the rules to train B. Is any object in the train reachable from outside of train B? Both object A and object R are reachable so we focus our attention on car 1. Leader R is evacuated to another train. The choice of which train is a policy decision. Here we chose train A instead of creating a new train. Next follower B is reachable from train A so it is evacuated to train A. Object C is only reachable from train B so C is moved into the last car in train B. The space used for car 1 is now recycled. Figure 5 shows the state of the trains after the first invocation of the algorithm.

[6] It might require two passes through the objects if the train remembered set information is managed as previously discussed.

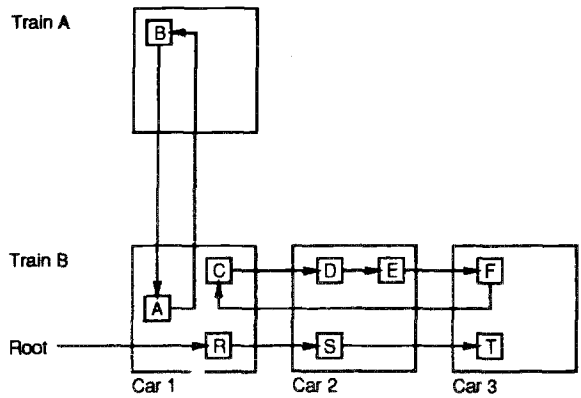


Fig. 4. The starting configuration

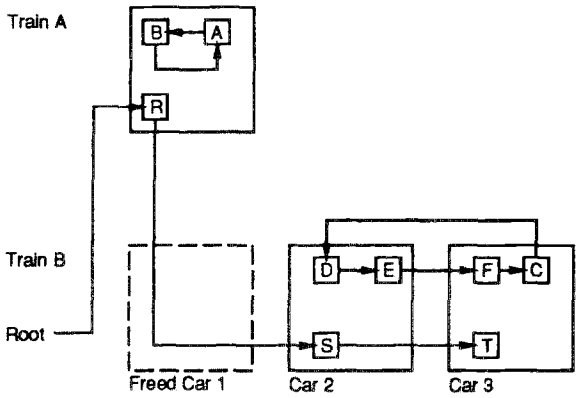


Fig. 5. Evacuate leader R, group A-B and copy C.

The second invocation of the collector focuses on car 2 in Figure 5. No objects are referenced by roots, so we look for objects in car 2 referenced from outside train B. Object S is referenced from train A, so we move S into train A. Since all cars in train A are full, we need to add another car to make room for S. Finally, we look for objects referenced from other cars in the train. Object D is moved into the last car of the train. Car 3 is full so we create a new car to make room for object D. The scanning of object D finds object E in car 2. E is evacuated into car 4. This gives us the state found in Figure 6. Notice how the live R-S-T structure is being extracted from the dead C-D-E-F structure.

On the next invocation of the algorithm we note that train B is still referenced so we focus on car 3. Again no objects are referenced by roots. Follower T is referenced by train A so it is moved into train A. Object T is scanned but contains no references into car 3. Next we consider references from within the train B. Object F is so referenced so it is moved into car 4. The scan of object F finds a reference to C. Since car 4 is full a new car is attached to the end of the train and C is moved into it. At this point (shown in Figure 7) structure R-S-T has been separated from structure C-D-E-F like pulling spaghetti out onto a fork.

The next invocation of the algorithm notes that train B has no references into it, so the

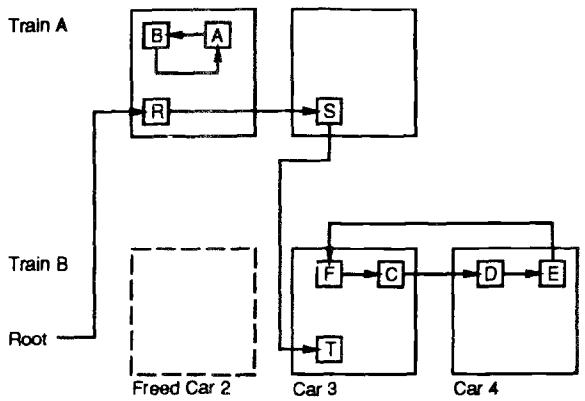


Fig. 6. Evacuate follower S and copy D and E.

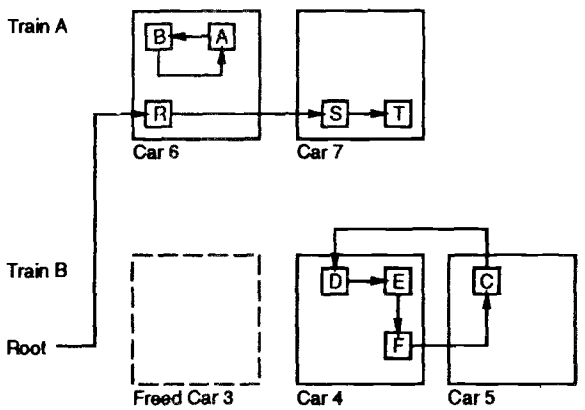


Fig. 7. Large cyclic dead structures are isolated into one train.

entire train is recycled immediately. Note that we isolated the structure C-D-E-F into one train where it can be reclaimed even though it is larger than any area we incrementally considered. This leaves us with only train A.

In Figure 8, we note that train A has a reference from outside the train so we focus on car 6. Since object R is reachable from a root we move it to another train. In this case we create a new train C and move R into it. Structure A-B, which used to form a circular list that spanned multiple trains, is now isolated and is recycled.

Figure 9 does not show recycled train B but does show the new train C that holds object R. We now consider car 7. Object S is moved into the train C and object S is scanned for references into car 7. Object T is found and moved into train C. Car 7 can now be recycled.

In Figure 10, what remains is a train with three neatly clustered live objects. These were the only three reachable objects present at the beginning of the example. The algorithm successfully grouped all unreachable objects into unreachable trains where they could be freed without disruption.

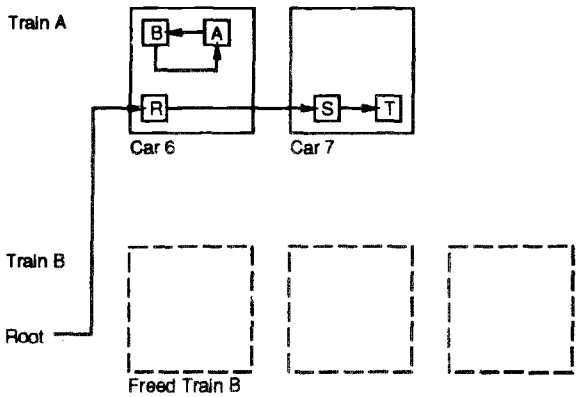


Fig. 8. Trains with no references can be freed.

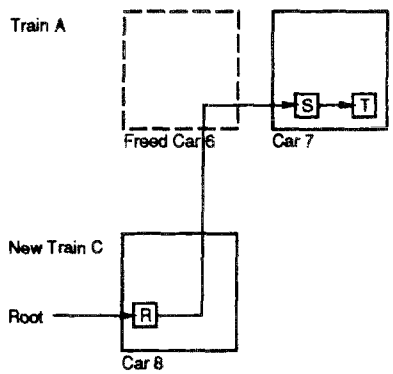


Fig. 9. Evacuate R so cycle A-B can now be freed.

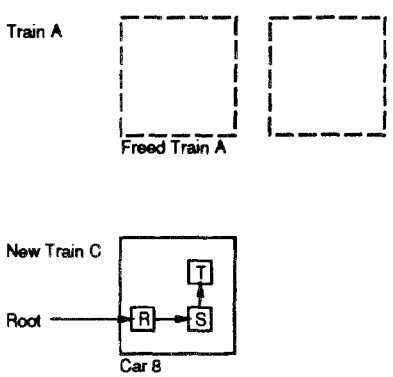


Fig. 10. Live structure R-S-T is clustered into one train.

6 Popular objects

There is one possible way in which our algorithm as presented might be disruptive. Call an object *popular* if there are many references to it. To copy a popular object, we must process a large remembered set and update many pointers.[7] In fact, we cannot bound the number of references to an object, so we cannot bound the work involved in moving an object.

Our solution is not to move such objects, analogously to the treatment of large objects. We can detect popular objects (or popular cars, anyway) by considering the size of their remembered set. If the remembered set size exceeds some threshold, we simply retain the whole area, logically (but not physically) copying it and having it start a new train (if it is an engine) or join the newest train (if it is a boxcar). With some cleverness we might be be able to clear out some objects, but it may not be worthwhile. The remembered set is discarded and will be rebuilt over time as we cycle through all the other areas. We need only take care that the threshold that determines popular versus non-popular areas is high enough that we can still collect highly linked cyclic garbage. Thus, the threshold should be no smaller than the number of pointers that fit in one area. We have yet to work out the details and correctness argument.

7 Future Work

We can add Wilson's temporal opportunism to our algorithm with no problem. Hayes's key object opportunism is more problematic since we assumed round-robin processing of the areas. To process areas in arbitrary order, we would have to remember all inter-area references (instead of just those pointing in one "direction") and we would have to deal with updating remembered sets. We might avoid updating remembered sets by including a "time-stamp" with remembered set entries, which would allow us to detect and ignore stale entries, rather than having to remove them immediately. The costs and benefits are unclear.

We envision a distributed version of the mature space algorithm. Though it falls outside the scope of this paper, we intend to develop a version where each node in a distributed system holds multiple complete trains. The algorithm does not change. If node A holds a structure S without a leader, then S will be migrated to some train in node B that holds a reference to S. If no node B is willing to accept the structure then either the structure will be discarded or node B and node A would have to agree on some sort of "rent" so node A could afford to retain S. Such a rental agreement would be equivalent to introducing a root in node A referencing S.

The MOS approach also seems promising for collect large persistent heaps for persistent and database languages. Some details would need to be worked out to insure that the algorithms makes as few secondary storage accesses as possible. It will probably pay to be opportunistic and do whatever processing one can on parts of the heap that are brought into main memory by normal application activity, as well as to exploit temporal opportunism to make more progress during periods of light load, etc.

[7] Large objects could also be a problem, but we can put them in large object space just as we do for the young generations.

8 Conclusions

We have described what we believe is the first efficient non-disruptive copy collection algorithm for mature objects. The algorithm is incremental, supports fast allocation, and supports compaction and clustering via copying. We believe this algorithm goes a long way towards making garbage collection palatable for a variety of languages and long running applications.

9 Acknowledgements

We appreciate Tony Hosking's work on implementing the toolkit discussed here. Amer Diwan and David Moon provided extensive comments on drafts of the paper. Other colleagues also read and critiqued the paper. Finally, we thank Barry Hayes for challenging us to implement key opportunism; it was thinking about that problem that led to our invention of the algorithm described here.

References

[Appel *et al.*, 1988] Andrew W. Appel, John R. Ellis, and Kai Li. Realtime concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), *ACM SIGPLAN Not. 23*, 7 (July 1988), pp. 11–20.

[Baker, 1978] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (April 1978), 280–294.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Boehm *et al.*, 1991] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In [OOPSLA, 1991], pp. 157–164.

[Cheney, 1970] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (November 1970), 677–678.

[Diwan *et al.*, 1992] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), SIGPLAN, ACM Press, pp. 273–282.

[Fenichel and Yochelson, 1969] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM 12*, 11 (November 1969), 611–612.

[Hayes, 1991] Barry Hayes. Using key object opportunism to collect old objects. In [OOPSLA, 1991], pp. 33–46.

[Hosking *et al.*, 1992] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, October 1992). To appear.

[Hudson *et al.*, 1991] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991. Submitted for publication.

[Jones, 1986] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM 29*, 4 (April 1986), 300–311.

[Lang and Dupont, 1987] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. *SIGPLAN '87 – Symposium on Interpreters and Interpretive Techniques* (1987), 253–263.

[Lieberman and Hewitt, 1983] Henry Lieberman and Carl Hewitt. A real-time garbage collection based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[Moon, 1984] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, TX, August 1984), pp. 235–246.

[OOPSLA, 1991] *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, October 1991), *ACM SIGPLAN Not. 26*, 11 (November 1991).

[Sleator and Tarjan, 1983] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory* (Boston, Massachusetts, April 1983), pp. 235–245.

[Sleator and Tarjan, 1985] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July 1985).

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, April 1984), *ACM SIGPLAN Not. 19*, 5 (May 1984), pp. 157–167.

[Ungar and Jackson, 1988] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Diego, California, September 1988), *ACM SIGPLAN Not. 23*, 11 (November 1988), pp. 1–17.

[Weinreb and Moon, 1981] Daniel Weinreb and David Moon. *Lisp Machine Manual*, third ed. Massachusetts Institute of Technology, 1981.

[White, 1980] Jon L. White. Address/memory management for a gigantic Lisp environment or, GC considered harmful. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Stanford, California, August 1980), ACM, pp. 119–127.

[Wilson *et al.*, 1991] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Canada, June 1991), *ACM SIGPLAN Not. 26*, 6 (June 1991), pp. 177–191.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, October 1989), *ACM SIGPLAN Not. 24*, 10 (October 1989), pp. 23–35.

Object Type Directed Garbage Collection To Improve Locality

Michael S. Lam[1], Paul R. Wilson[2], and Thomas G. Moher[1]

[1] University of Illinois at Chicago
[2] University of Texas at Austin

Abstract. Most garbage collected systems have excessive need for RAM to achieve reasonable performance without too much paging. The reason for such poor locality is the way data are organized in the heap. Conventional organization approaches such as breadth-first ordering do not always bring objects in the same active working set together. When such co-active objects are distributed throughout the heap (on different memory pages), high paging costs will result from accessing objects during execution. To alleviate such poor ordering, researchers have tried many different approaches: depth-first ordering, dynamic reorganization, object creation ordering, and hierarchical decomposition. Each of these approaches has its associated costs, effectiveness, and limitations. This paper presents a new ordering approach to improve locality. By paying a little attention to object type and format, effective heuristics can be derived to group co-active objects together. To investigate this idea, a number of such object type directed grouping techniques are incorporated into a Scheme-48 system. Page fault reduction of up to an order of magnitude was observed.

1 Introduction

Garbage collection was first introduced by McCarthy to reclaim heap (dynamically allocated) storage automatically for the language Lisp [9]. This first proposed mark-and-sweep scheme would have very poor locality in a virtual memory environment. Because the majority of the heap-allocated data die young, the surviving objects take up only a small percentage of the entire heap space, but they tend to be distributed throughout the heap. In a virtual memory operating system, the heap is divided into virtual memory pages, and most of such pages have to be brought into RAM in order to access the live objects. As a result, the paging costs of such a system remain that of the entire heap size, while only 10 percent of the heap may actually contain live data.

To alleviate this excessive paging cost, compacting can be done, as proposed in the first stop-and-copy algorithm by Fenichel and Yochelson [7]. In "stop-and-copy" collection, the heap is divided into two semi-spaces. Allocation is done in one space until that space is filled. Then garbage collection is invoked to copy all surviving objects in the filled space to the other space. Since only a small percentage of objects remain alive, this copying process can usually compact the surviving objects into a smaller area at the beginning of the other space. More room is made for future allocation, and many fewer pages are needed to be brought in to access all of the live objects.

However, even the compacted surviving data can be quite sizable. For heap-based languages like Lisp and Smalltalk, surviving data means not only the data generated through execution, but the system image of predefined library functions, typically including the compiler, editor, browser, and debugger. Because these functions remain alive throughout execution of every program, they have very different life-spans from program-generated data, and should be isolated from program-generated data to avoid creating short-lived data among them. Otherwise, program-generated data can become garbage very soon and pollute the static quality of the system image. The garbage spreads the system functions apart and more pages are needed to cover the same amount of functions. To achieve this isolation, generational garbage collection can be used [8, 15]. By dividing the heap into different generations, program-generated data are created in the younger generation, while the static system image can be placed in the older generation with a minimal set of pages.

Even with compaction and generational collection, locality of reference for most heap-based systems still remains unsatisfactory. Caudill reported that for Tektronix Smalltalk, good performance is achieved only when most of the system is paged in [3]. Apparently, there is some flaw in garbage collection that does not interact well with modern virtual memory systems. To address the problem, this paper discusses the flaw in the next section, then examines previously proposed solutions, and finally presents our new approach to the locality problem.

2 The Problem and Previous Solutions

In seeking to improve locality of reference, many researchers have identified the problem as the way heap data are rearranged during collection. Traditionally, the Cheney algorithm has been the standard in implementing copying collectors because of its simplicity [5]. The Cheney algorithm imposes a breadth-first ordering when copying objects across semi-spaces.

To illustrate how breadth-first traversal can go wrong, consider a heap with a root set of 10 linked lists. A breadth-first traversal will first copy the 10 list heads to the new space. Then the 10 heads are scanned, and because they lead to the second nodes, the second nodes of all 10 lists are brought over next. This copying of the list, level by level, keeps repeating until all nodes are copied. Mixing these 10 individual lists is generally a bad idea because elements within the same list are more likely to be referenced together. Evenly distributing these lists means spreading the elements within the same list apart, perhaps on different pages (figure 1).

A more sensible ordering is to copy one complete list at a time (figure 2). If the head of one linked list is touched, the next most likely touched node is the second one, because linked lists are traversed sequentially. Hence, the second node is, in a sense, more "related" to the linked list head than anything else. To minimize paging, we can lay out such related objects (elements of the same list) sequentially by a depth-first traversal.

Many researchers have realized the problem of breadth-first traversal and have tried improving locality by changing the garbage collector to use depth-first traversal [10, 13, 6]. Unfortunately, the improvements achieved were disappointingly small. The most recently reported number by Courts was an improvement of 10 to 15 percent [Cour88].

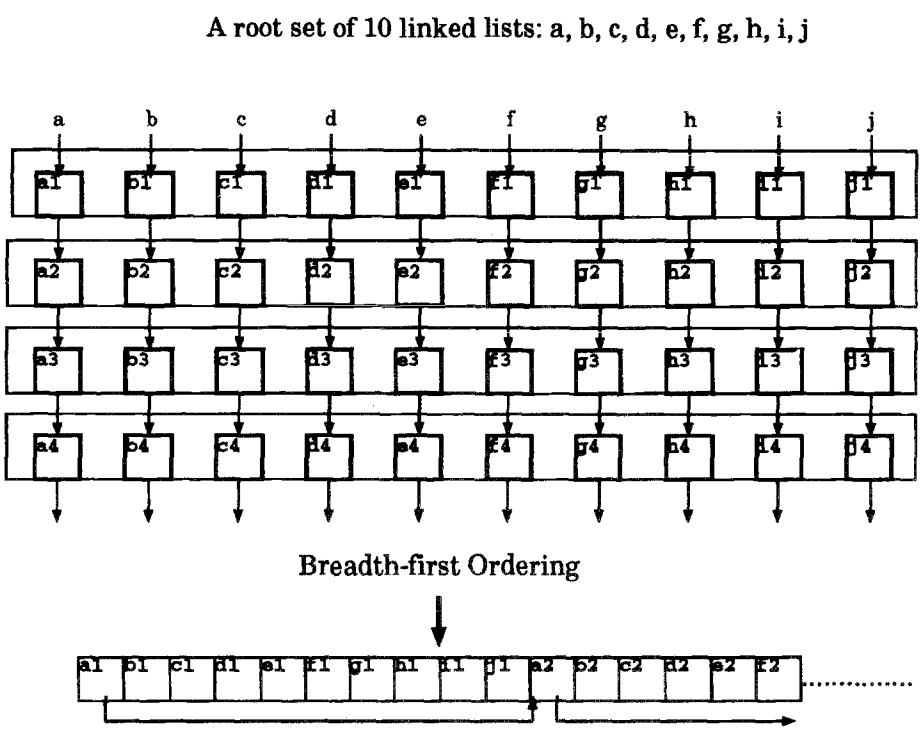


Fig. 1. Breadth-first Ordering of 10 lists

Why is the improvement from depth-first so insignificant? The main reason is that linked lists are not the most common objects in a system image. The majority of most system images are library functions. To achieve good locality, we need an organization scheme that groups related functions together.

One major attempt at organizing heaps with a focus on function grouping was reported by David Andre [1]. His most notable technique was using the *creation order* (order of presentation to the compiler) of function code to organize system image.

Organizing the system according to creation order makes sense because the creation order tends to bring related functions close to each other. In any sizable software development effort, different files are used to contain different groups of related functions. Consider a compiler divided into four phases: scanning, parsing, optimizing, and code generating; the scanning functions are normally placed in files different from those for parsing, optimizing, and code generating. If we organize the object code the way functions are laid out in files, we can preserve the grouping of related functions. Then when the compiler is in the scanning phase, the needed paged-in scanning functions will be grouped in a minimal set of pages. When the scanning phase is over, the set of parsing related functions are brought in. Again, they will be grouped in another minimal set of pages. Since programmers by default tend to write related functions in order, preserving the functions' compilation order in the object code is generally a win.

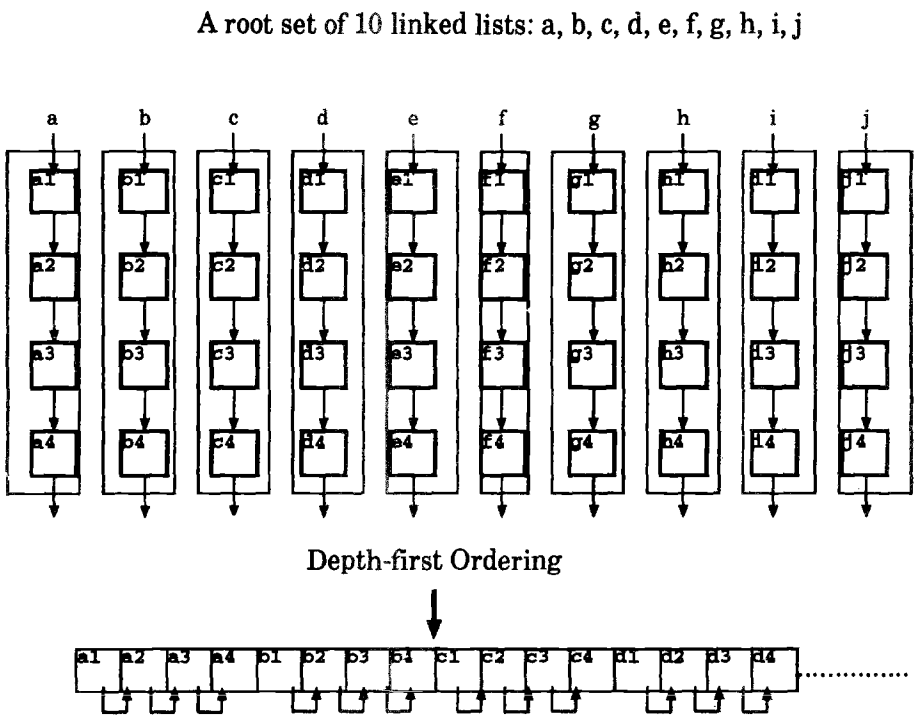


Fig. 2. Depth-first Ordering of 10 lists

We elaborated on Andre's ideas and implemented our version of function creation ordering garbage collection [17]. The resulted locality improvement was about an order of magnitude in page fault reduction. In our previous work, we have made extensive use of four benchmarks: the Scheme-48 compiler, the Conform program, the Boyer-Moore Theorem Prover, and the Zebu parser generator.

The Scheme-48 compiler consists of several thousand lines of Scheme-48 code. Its front end performs input reading, macro expanding, and source transformations. Its back end generates static link objects (called closures), literal frames (called templates) , and byte code object (called code vectors). The Conform program consists of several hundred lines of code, implementing programming language type conformance. The Boyer-Moore benchmark, consisting several hundred lines of code, is a a rewrite-based theorem prover written by Bob Boyer, as part of Gabriel's suite of Lisp benchmarks. The Zebu benchmark is a yacc-like parser generator that consists of several thousand lines of code. It takes SLR and LALR(1) grammars in list forms and generates a parsing-action table. All these programs execute millions of interpreted byte code instructions, allocate megabytes of data, and perform non-trivial work. We believe they are good representatives of Lisp programs because they cover a wide range of Lisp operations.

In addition to the function treatment, we added another technique called *hierarchical decomposition* [17] to improve the locality of the remaining non-function data structures. We chose to try a new traversal approach rather than staying with

depth-first traversal because depth-first may not be the most optimal. Although the previous example demonstrated that depth-first traversal is better at grouping linked lists than breadth-first, linked lists are not the most common data structure. We believe that most data structures are tree-like; unfortunately, depth-first ordering tends to group a tree into vertical slices (figure 3).

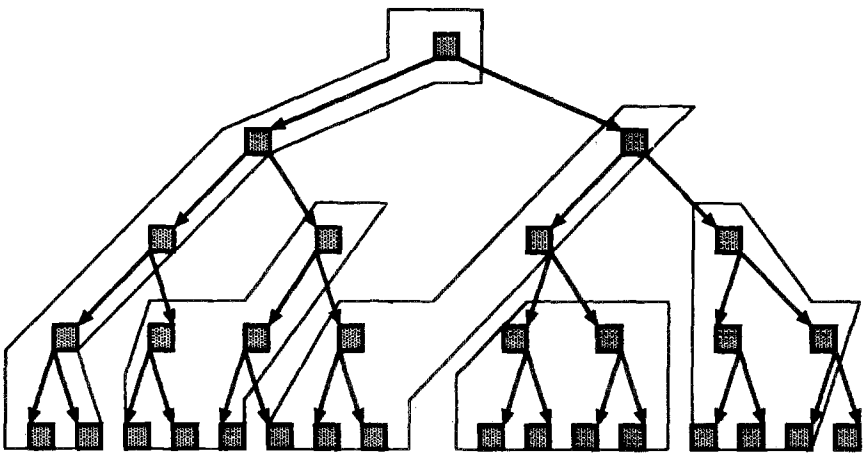


Fig. 3. Depth-first Ordering a Tree

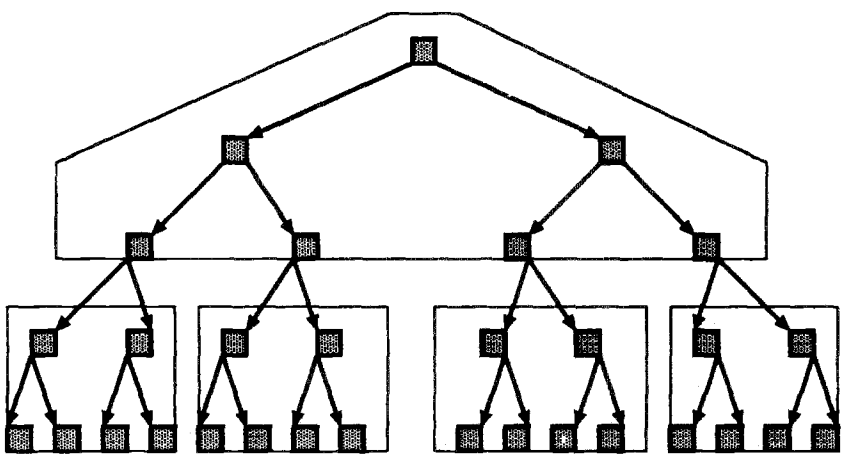


Fig. 4. Hierarchical Decomposition Ordering a Tree

Our hypothesis is that tree structures should best be grouped in subtrees (figure 4). That is, whenever we start with a node, we group it with its closest descendents up to a few levels. This grouping should be better because it covers both children of

a node, while depth-first grouping usually covers only one. We also believe such hierarchical decomposition will even be better for B-trees, where tree nodes have more than two children. To enable such grouping, we have developed our own hierarchical decomposition traversal algorithm (details discussed in [17]).

With hierarchical decomposition, the locality improvement was even greater than creation ordering alone. However, our system image consisted of too much code, not enough data, and did not demonstrate enough of hierarchical decomposition's effectiveness. Hence, the OO1 (Object Operations version 1) database benchmark, also known as the "Sun Benchmark," published by Cattell of Sun Microsystems [2], was implemented to further evaluate the effectiveness of hierarchical decomposition. This benchmark basically creates a tree via standard binary tree balancing routines, and creates data records at the leaf level of the tree. Such leaf-level records also contain pointer fields to other leaf level records, resulting in a network throughout the leaves (see figure 5). This database is huge, over five megabytes, 14 times the size of our normal system image.

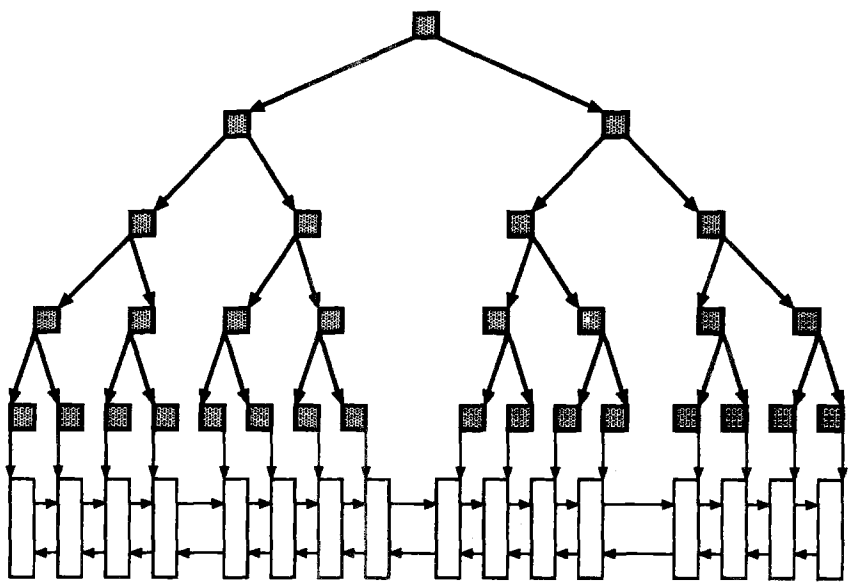


Fig. 5. OO1 Database

Unfortunately, upon testing the hierarchical decomposition scheme with the OO1 benchmark, the result was very disappointing: even a breadth-first ordered system had better locality than a hierarchically decomposed system.

To explain such inconsistent results, the OO1 benchmark was modified such that the pointer fields of the leaf level records were nullified. This extracted away the leaf-level network from the structure and turned it into a pure tree. With such modified structure, hierarchical decomposition again yielded better locality. Hence, it is not that hierarchical decomposition failed to organize a tree, but rather that the data structure was not a pure tree to begin with. This led us to the observation that the

effectiveness of a traversal algorithm depends heavily on how the data structure is connected.

Since data structures are created and used so differently, it is understandable that a fixed traversal cannot yield the optimal grouping for all cases. It is more likely that each traversal works best with a certain class of data structure. This explains why the improvements of depth-first over breadth-first have been so inconsistent, and why hierarchical decomposition does better in some cases, and worse in others.

3 Object Type Directed Function Grouping

To achieve locality, related objects should be grouped together. The most successful example of such was Courts' dynamically reorganizing garbage collector, on the TI Explorer [6]. With hardware assists, this scheme can relocate objects on-the-fly. It achieves excellent locality because objects are relocated and grouped exactly as they accessed. With today's inexpensive RISC technology, few can afford the luxury of Lisp machines. Still, a good grouping of related objects should be possible on stock hardware.

In heap-based languages like Lisp or Smalltalk, data tags are often used to classify objects in memory. Most languages, however, rarely use the tag information during garbage collection. For instance, in Lisp garbage collection, whether an object is a vector or cons cell is irrelevant. The collector is still going to copy and scan it the same way. In practice, the object's type gives a good indication about how the object is accessed. The access pattern can in turn be an excellent heuristic in grouping related objects together.

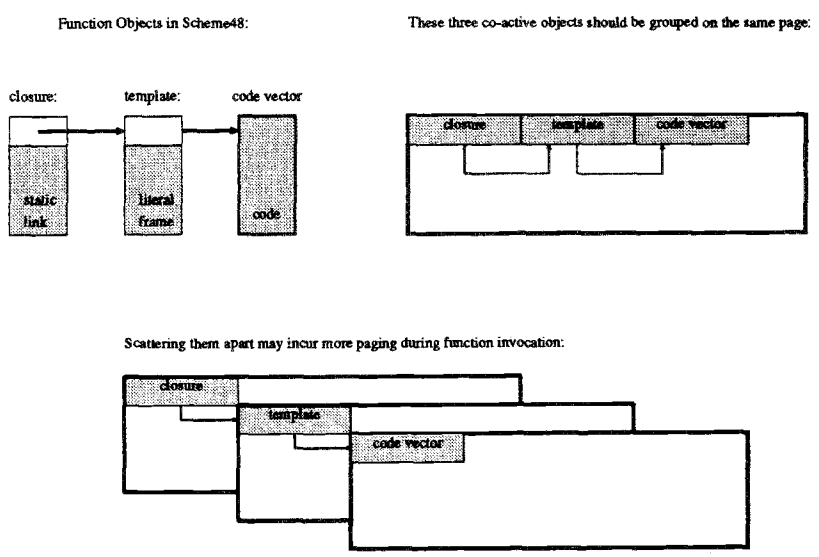


Fig. 6. Putting Scheme-48 Function Objects Together

In Scheme-48, a function is composed of a closure object (static links), a template (literal frame), and a code vector. When a function is invoked, the closure is accessed, the template is accessed through a pointer field of the closure, and then the code vector is accessed through a pointer field of the template. Since all three objects are used during function invocation, it makes sense to group them on the same memory page so paging cost can be minimized. To perform such a grouping, the garbage collector already has all the necessary information. While copying an object across semi-spaces, the data tag can be examined. Should the tag indicate a closure object, the pointer to the template could be scanned and the template copied. Likewise, the pointer to the code vector could be scanned and the code vector copied (figure 6).

While the above grouping is logical and easy to perform, common breadth-first traversal usually fails to bring these three together. Depth-first traversal, on the other hand, will scan all pointer fields in the closure object next, and may bring too many objects close to the closure before getting to the template and code vector. To yield a good grouping, the most effective approach is to understand the internal format of all objects, and explore the pointers that lead to the most related objects. However, spending too much effort here can incur a large overhead for the garbage collector. Hence, only a few common objects with common access patterns should be targeted for optimization.

Because code is the most frequently accessed object in the heap, the most productive optimization should be done on grouping related functions. In Scheme-48, a function call is implemented by loading the function address from the template (literal frame) of the executing function, then branching to that address. Thus, all of the potentially callable functions can be found through the template. One way to group the related functions is to scan through the template and copy the reachable functions close to the currently scanned function. Because these reachable functions in turn call other functions, the template scanning and function copying process has to repeat until no more functions can be reached, or until all reachable functions have already been copied.

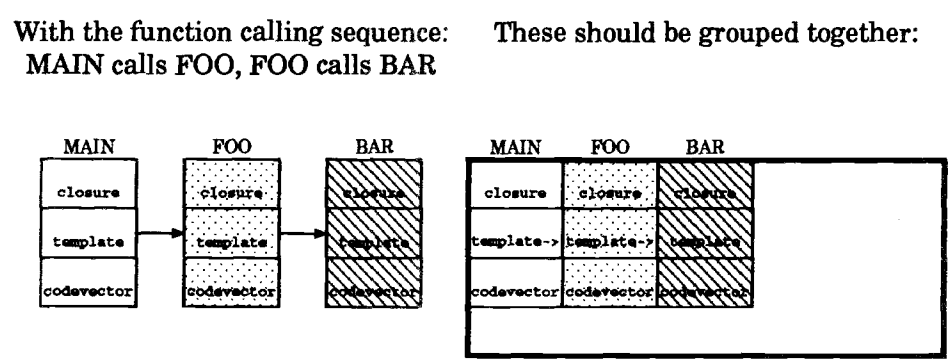


Fig. 7. Grouping Functions in a Calling Sequence

The function calling sequence is tree-like; execution starts from one root function, this root function calls a set of second-level functions, and the second-level functions

in turn call more functions. Such function calling behavior means calling functions should be grouped with the called functions. Then paging can be minimized between function calls.

4 Object Type Directed Data Structure Grouping

In addition to function objects, commonly used data types can also be used as grouping heuristics. In Lisp, the most commonly used data type is the *cons* cell, and its most commonly accessed patterns are depth-first trees and association lists (not cdring down lists) [12]. Using this information, a garbage collector can be made to recognize trees or association list, then perform different traversal algorithms as appropriate. To guess the data structure that a cons cell heads is not really that hard; a few levels of pointer traversal usually suffices. For instance, the cons cell in an association list typically leads to a symbol in the car part, and a value in the cdr part (this is based on programming experience, and Shaw's measurement that most car fields lead to symbols when excluding other cons cells and NIL values [Shaw86]). As for trees, both car and cdr parts lead to many other cons cells.

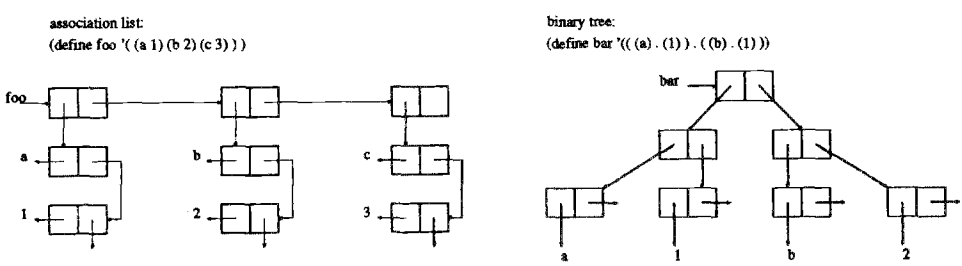


Fig. 8. Common Data Structures

Upon determining the form of data structures, the traversal can be done accordingly, using, for instance, hierarchical decomposition for trees, and depth-first traversal for association lists (this decision has not yet been proven optimal). In practice, one data structure often leads to another data structure (for instance, the leaf-level records of the OO1 database, figure 5). The specific traversal for the currently traversed data structure may not be optimal for another one. So some provision should be made to change the currently used traversal when crossing a data structure "boundary."

Again, object types can be used here. For instance, an association list is typically made up of cons cells. While traversing the list, if any non-cons cell object has been reached, it is a good indication that the boundary of the association list has been reached, and that the non-cons cell object may not be appropriate to be grouped with the cons cells. Actually, this decision depends on the kind of objects encountered. For pointer-containing objects, they may be roots that lead to another structure of a different nature. In this case, they deserve a deeper look, and perhaps a different traversal treatment.

5 Implementation Details

Our testbed is the Scheme-48 system, a Lisp dialect originally written by Jonathan Rees [11] (Scheme was invented by Steele and Sussman [14]). The system is divided into a high-level user interface including the reader, compiler, and command interpreter (written in Scheme), and a low-level run-time system of byte code interpreter, and garbage collector (written in C++). We use a generational garbage collection scheme in which the heap is divided into three generations. The youngest generation is used for creating new objects, the middle generation is used for promoted objects from the youngest generation (which typically live through the program execution), the oldest generation is used to store static objects such as the system image. Each generation consists of a pair of semi-spaces; a stop-and-copy scheme is used to collect surviving objects.

Our first step was to implement the type-directed function grouping. Since the essence of this scheme is to group all the transitively called functions in a calling sequence, finding the root functions to begin with is vital. We did not want this to incur too much collection overhead, so only three entry points in the Scheme-48 system are used as root functions: the main function, the reader, and the system table-making routine. These are selected because most of the system primitive operations go through them indirectly. In addition, the specific function (program) being run is also optimized. In Scheme-48, entering the function name at the command prompt provides enough information for the garbage collector to locate the function objects, so the function grouping can also be applied to any program before running. Generally, the more root functions to which we apply the function grouping optimization, the better the locality.

During garbage collection, the root functions are copied to the new space one at a time. After each function is copied, a scan is done on its pointer fields leading to other functions. As a result, all the reachable functions get copied transitively through the root function. The traversal order is basically depth-first, although not all pointers are explored.

Naturally, function literal frames have more pointer fields than just those that lead to other functions. Initially, these fields are skipped. To complete the garbage collection, these fields need to be scanned. There are basically two ways in going back to these pointers that have already been copied to the new space: remembering them in a data structure, or re-scanning. We chose re-scanning because of its simplicity, and because the total number of functions brought over due to the functions grouping is relatively small.

Apart from the re-scanning, the only other major cost of the function grouping traversal is checking the pointer fields. To determine whether a pointer leads to a function is easy: one indirection operation will lead to the closure object (the first of the three components of a function). Because a Scheme-48 closure object has its distinct data tag, one more compare operation suffices to indicate whether a pointer leads to a function.

The next step was to implement the data structure grouping. As mentioned in the section three, we cannot afford to invest too much overhead in analyzing object internal semantics. So only cons cells and vectors are targets for optimization.

For cons cells, we followed Shaw's statistics in assuming that they are mostly

accessed as association lists or trees [12]. While performing the normal breadth-first scanning phase of garbage collection, we added an extra object type check for cons cells. If a cons cell is being scanned, a few levels of lookahead are performed to see if it leads to a sequence of symbol/value pairs (i.e. it resembles an association list). If so, a local depth-first traversal is started with the current cons cell. Depth-first is used here because we want all cons cells of the association list to be grouped in sequential order, and we expect them to be traversed sequentially during execution.

If the cons cell data structure does not resemble an association list, we assume it is a tree. According to our previous hypothesis [17], our hierarchical decomposition traversal is more suitable in grouping trees. So a local hierarchical decomposition traversal is done here to create a tree grouping rooted at the currently scanned cons cell.

Vector objects are detected exactly at the same place as cons cells, during normal breadth-first scanning. Since vectors are used for so many different data structures, we made the simplifying assumption that a tree-like structure is the most common. Again, hierarchical decomposition is used.

To summarize our changes, we have basically augmented the standard breadth-first scanning segment:

```
free = to_space;        /* switch semi-spaces */
copy(root_set, &free); /* copy root set over */
scan = to_space;  /* start scanning from the beginning of to_space */
while (scan < free)
{
        if (IS_POINTER(scan))
                copy(scan,&free); /* copy the referent of scan */
        scan++;
}
```

by modifying it in the following way:

```
free = to_space;
copy(root_set, &free);

for (all root functions f) /* traverse and transitively copy related*/
        explore_related_functions_from(f);              /* functions*/

scan = to_space;
while (scan < free)
{
        if (IS_POINTER(scan))
                if (IS_CONS_CELL(scan))
                        if (LEAD_TO_ALIST(scan))/* association lists*/
                                                /* use depth-first  */
                                depth_first_group(scan);
                        else
                                hier_decompose(scan);/* trees, use  */
                                      /* hierarchical decomposition */
```

```
            else if (IS_VECTOR(scan))                    /* vectors */
                    hier_decompose(scan);    /* use hierarchical */
                                             /*    decomposition */
            else
                    copy(scan,&free); /* copy other kind of     */
        scan++;                       /* referents as before    */
}
```

The functions IS_POINTER, IS_CONS_CELL, IS_VECTOR, and LEAD_TO_ALIST all have constant costs. The other three: *explore_related_functions_from* , *depth_first_group* and *hier_decompose* have costs proportional to the numbers of transitively copied objects; still, these copying costs were needed for the original copying scheme anyway. We have just altered the copying order of some objects, and added re-scanning costs for such pre-ordered objects.

6 The Experiment

Virtual memory simulation was used to measure the effectiveness of the grouping techniques. The Scheme-48 run time system is instrumented such that memory reference traces can be collected during program execution. The collected reference traces are then passed to a virtual memory simulator to simulate the paging behavior under such memory references.

The virtual memory simulator maintains a LRU queue as in most operating systems. Given a memory address, the virtual memory simulator computes the corresponding page number. If this page number is not in the LRU queue, it means this page has never been paged in, so the page number is appended to the queue and its position (queue length) returned. If the page number is already in the queue, then its queue position is returned. Based on the queue positions, the page fault rate can be computed for any number of pages of various sizes.

In the simulation, only references to the oldest generation (which contains the system image) are collected. We neglected the younger generations because the grouping traversal did not apply to newly created data. Such data die rather quickly and are not usually worth the grouping effort. Techniques for dealing with newly created data have been addressed in [16, 18]. Actually, objects that get promoted to the middle generations may also benefit from the grouping techniques, but the effort is hard to justify because we cannot tell when the program will finish after investing the grouping effort on organizing the middle generation data.

7 Results

Our results are shown in Figures 9 - 22, as plots of page faults vs. memory size. A 4k page size was used in the simulation and then multiplied by the number of pages to get the memory size. Figures 9 - 16 present graphs for the four non-database benchmarks. Figures 17 - 22 show the graphs for the OO1 benchmark. The result for every benchmark is displayed in regular size, and in expanded form to illustrate the points of interest.

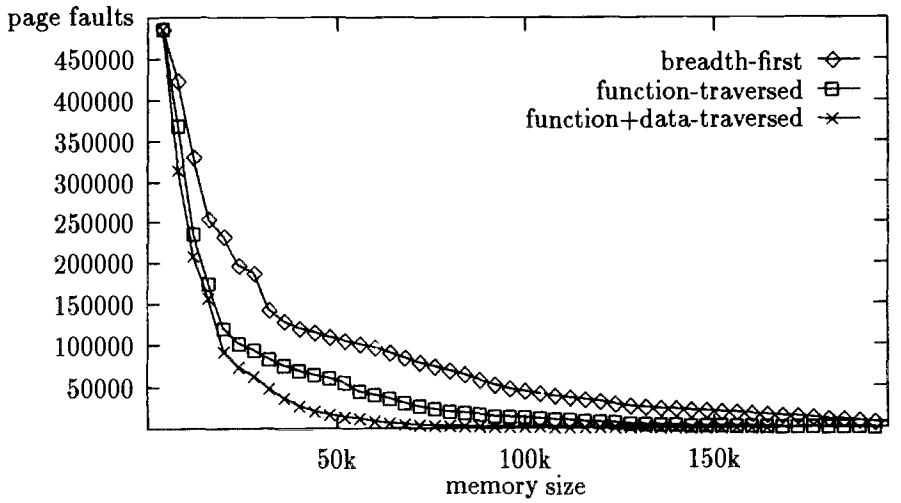


Fig. 9. The page faults from running the Scheme-48 Compiler for different memory sizes

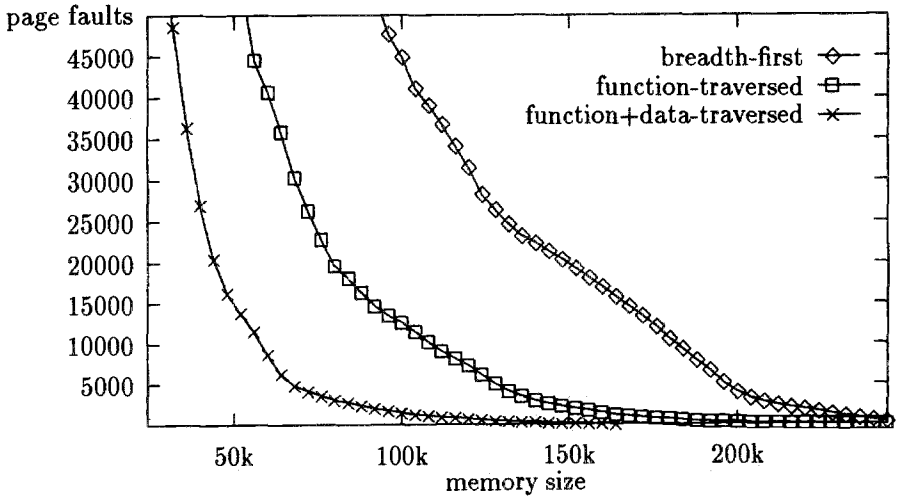


Fig. 10. Focusing on the closer page faults for running the Scheme-48 Compiler

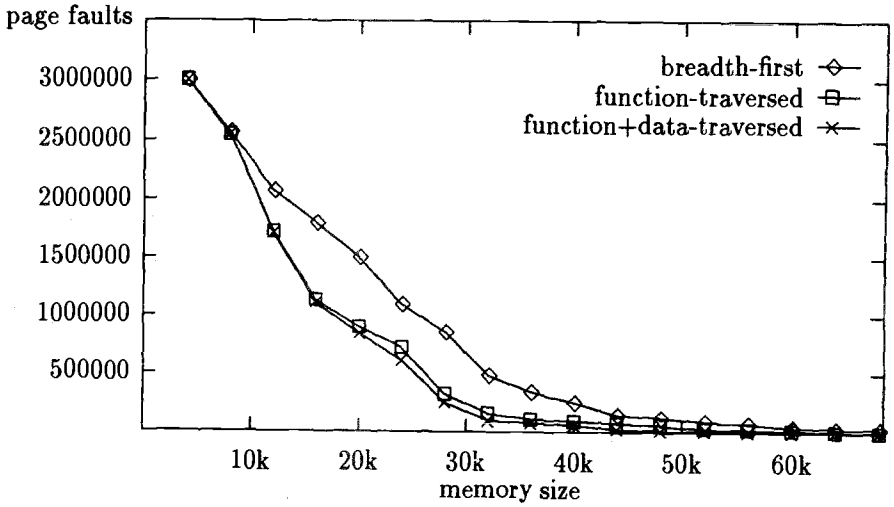


Fig. 11. The page faults from running the Boyer-Moore Theorem Prover for different memory sizes

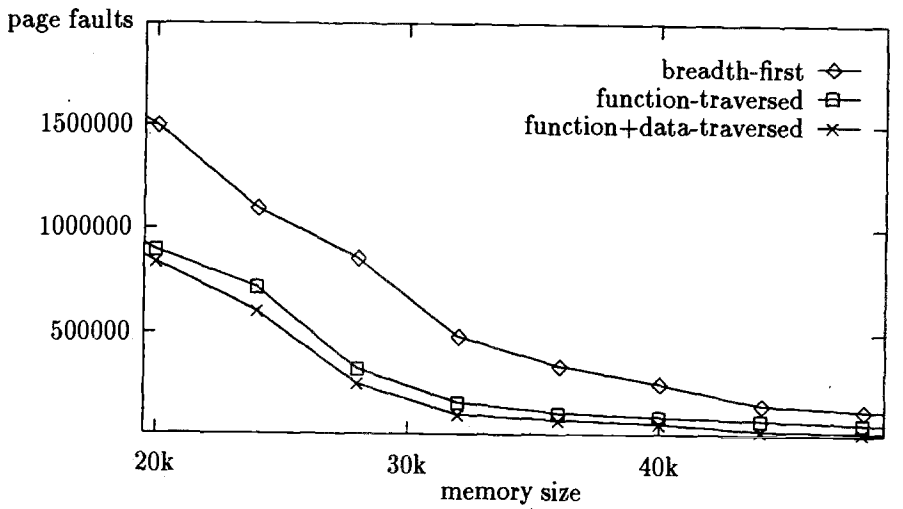


Fig. 12. Focusing on the closer page faults for running the Boyer-Moore Theorem Prover

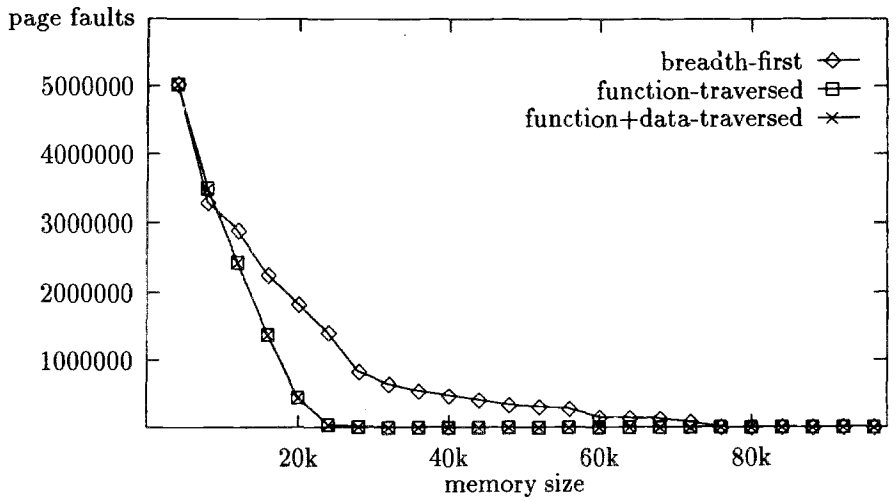


Fig. 13. The page faults from running the Conform Program for different memory sizes

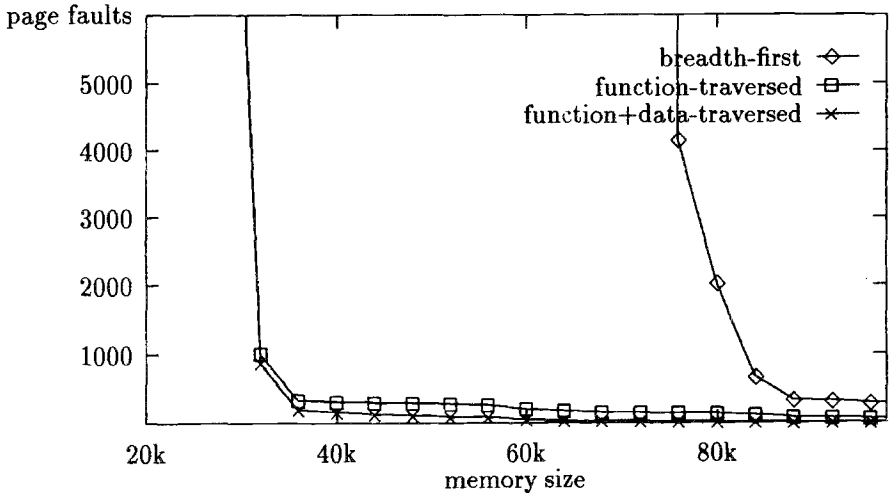


Fig. 14. Focusing on the closer page faults for running the Conform Program

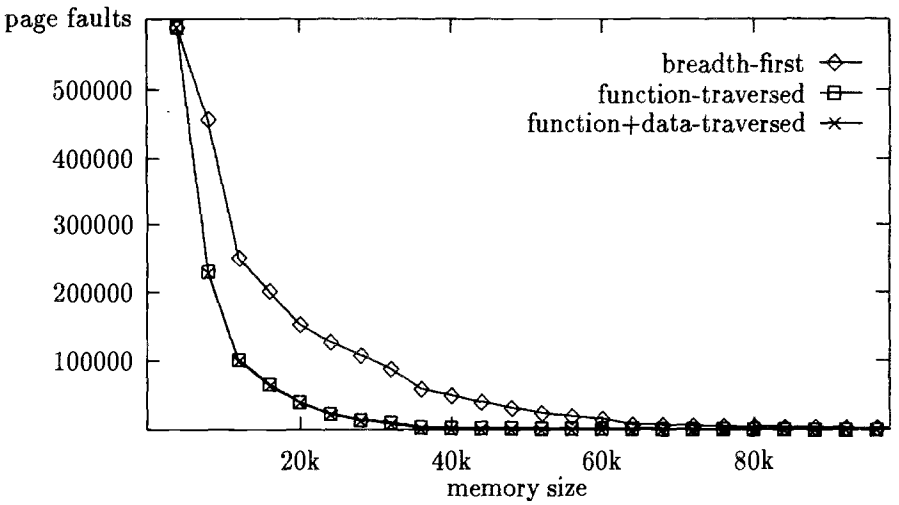


Fig. 15. The page faults from running the Zebu Parser Generator for different memory sizes

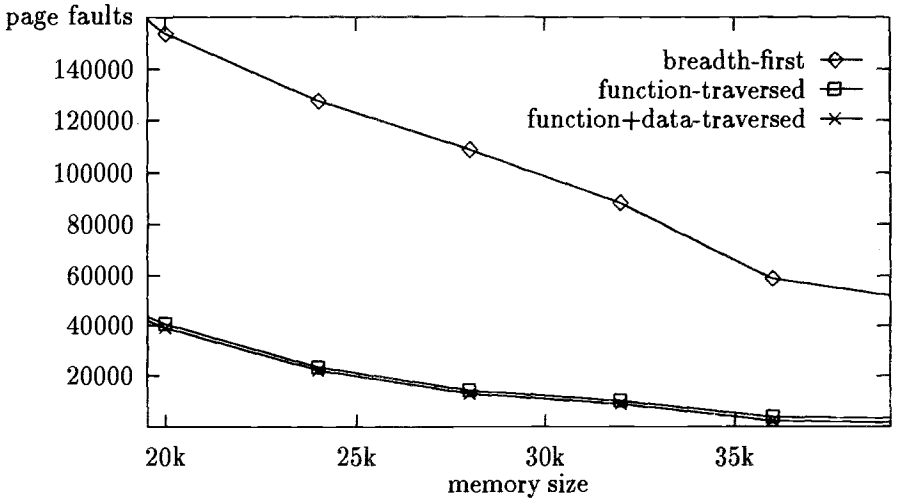


Fig. 16. Focusing on the closer page faults for running the Zebu Parser Generator

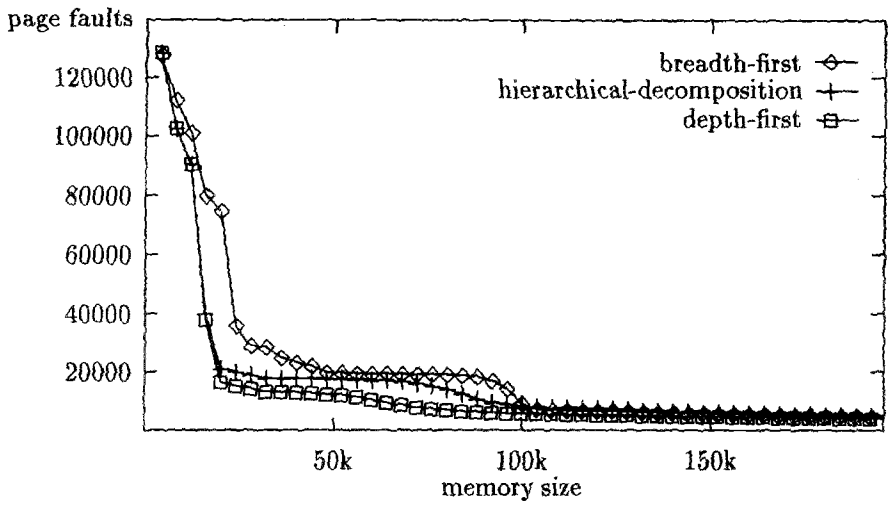


Fig. 17. The page faults from running the OO1 Database Lookup operation for different memory sizes

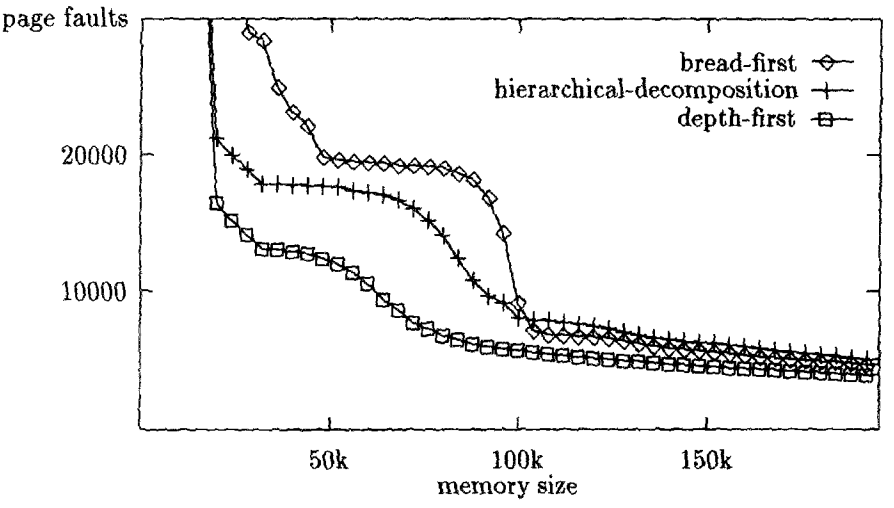


Fig. 18. Focusing on the closer page faults for the OO1 Database Lookup operation

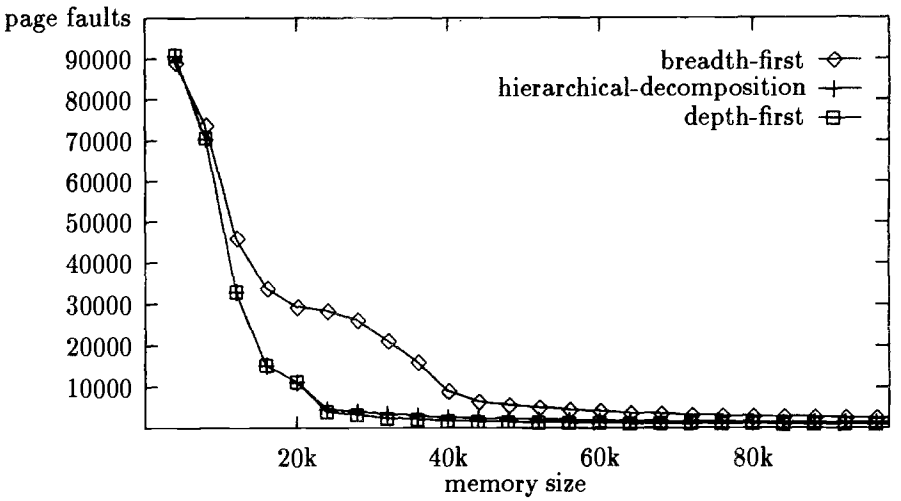


Fig. 19. The page faults from running the OO1 Database Traversal operation for different memory sizes

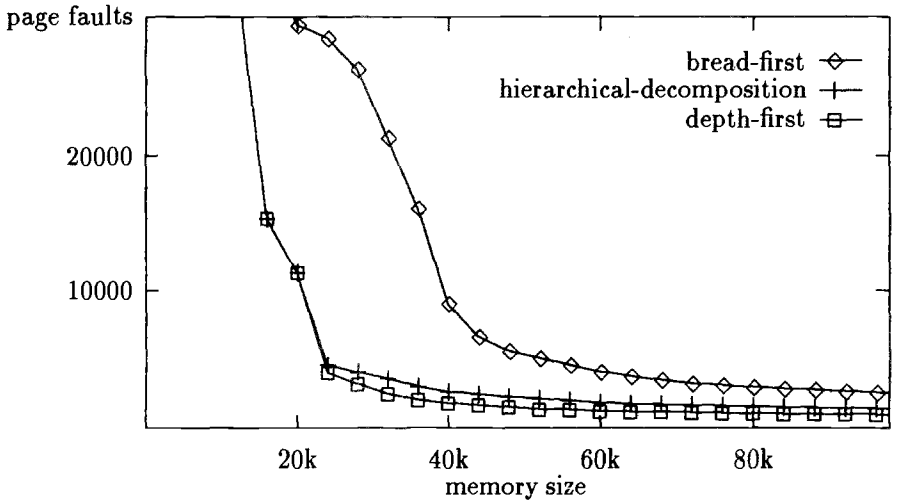


Fig. 20. Focusing on the closer page faults for the OO1 Database Traversal operation

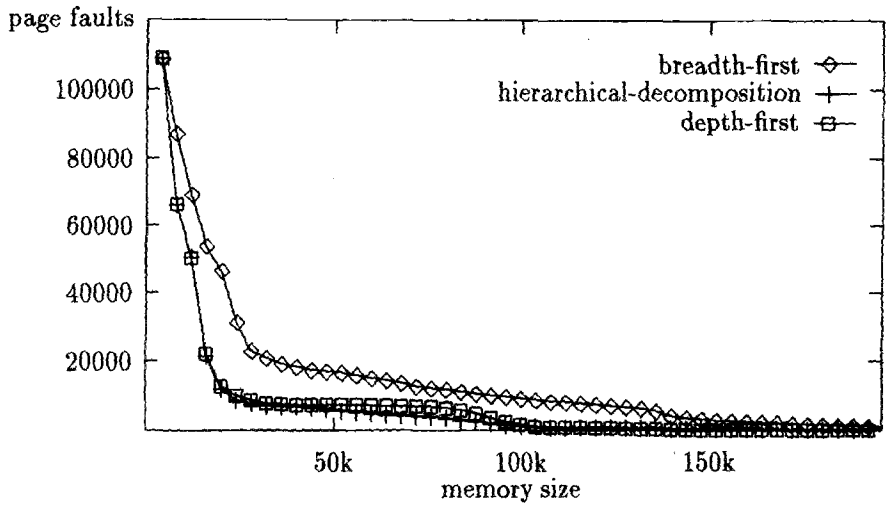


Fig. 21. The page faults from running the OO1 Database Insert operation for different memory sizes

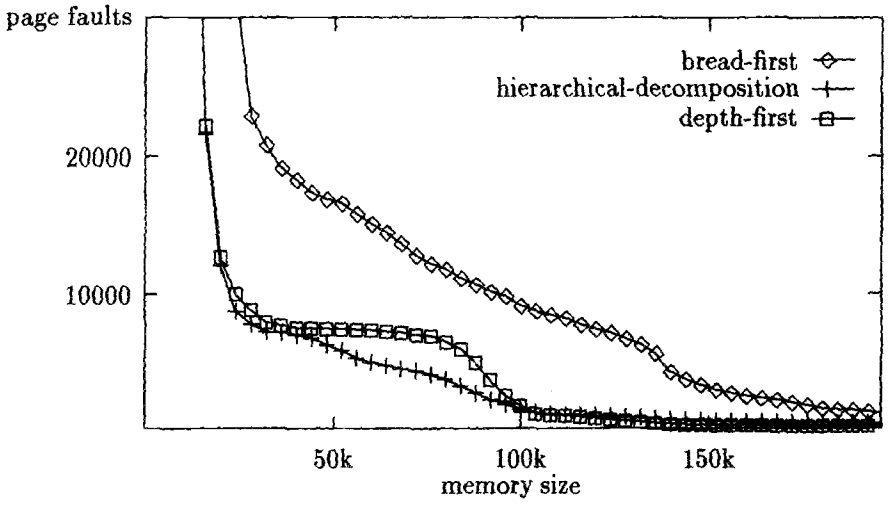


Fig. 22. Focusing on the closer page faults for the OO1 Database Insert operation

For the four non-database benchmarks, results were collected for both related-function grouping only and related-function grouping plus data structure grouping. For these four benchmarks, the majority of the system image is composed of code, so the effect of related function grouping was expected to be much more significant. The effectiveness of adding the data structure grouping depends on the degree to which association lists and trees were used in the benchmarks. Both the compiler and the Boyer theorem prover use a fair number of association lists, so the page fault differences are noticeable. In contrast, the Zebu parser generator and Conform program rarely use any system data structures, so these curves look identical in the regular graph (they are different on the expanded form).

When applying both grouping techniques, the drop in page faults is very sharp for every benchmark, especially for the two larger programs (compiler and Zebu). This improvement was found even with a very small amount of memory At 20k (five 4k pages of RAM), page faults are at least halved. The improvement increases to over an order of magnitude for a long range of realistic memory sizes.

Our function grouping is basically achieving the same gain as Andre's function creation ordering [1]. Both group related functions together by understanding some common function characteristics. We compared our related function grouping results with the creation ordering results from a previous paper (not shown in our graphs) and our new results are marginally better.

The OO1 database benchmark is run in three different operations: lookup, traversal, and insertion. Further details about these operations are described in [2].

Because the database is so much larger than the code for the three operations, the function grouping effect is negligible; only the curves for both groupings are displayed. The locality improvement comes mainly from data structure grouping. Because the database is basically a tree, the scanning algorithm selects a hierarchical decomposition strategy (and the object type analysis described in previous sections to include only tree nodes).

We were also curious about how our hierarchical decomposition compared with depth-first traversal. A depth-first organization of the database image was also created and measured (also using object type analysis to include only tree nodes). There is a definite drop in page faults while applying either hierarchical decomposition or depth-first traversal. So there is definite advantage in organizing tree data structures using schemes other than breadth-first ordering. The improvements, however, are not as compelling and consistent as those for the non-database benchmarks.

It should be pointed out that breadth-first ordering is actually doing not too bad to begin with. For all three operations, the drop is sharp up to around 20k of memory, then the drop becomes much more gradual. With limited RAM size, such performance is really not bad, considering the database size is five megabytes. One explanation of such performance is the absence of hash tables, whose destructive locality effects we discussed in our previous paper [17].

The lookup operation result surprised us the most because depth-first ordering outperformed hierarchical decomposition. This contradicts our hypothesis that hierarchical decomposition is best at organizing tree structures. Currently we are looking for more tree benchmarks and ways to tune hierarchical decomposition.

One last point about the database benchmark is that although the data structure is exactly the same for all three operations, different ways of accessing the data

structure make radical locality differences for different organization schemes. Here, depth-first ordering seems to be the best, yielding the most consistent results for organizing binary trees.

8 Conclusion

By examining readily available object type information, object access patterns can be inferred, which can guide the garbage collector in better organizing heap data to improve locality. This idea has been tested in a Scheme-48 system. Locality gain in related functions grouping is up to an order of magnitude, and gain in data structure grouping is up to 100 percent.

Before adopting such techniques, however, the first natural question is whether the extra overhead outweighs the locality gain. Since garbage collection rarely occurs in the system image, our relatively more expensive grouping traversal is needed only after system compilation, and its benefits will last through all subsequent use. With generational garbage collection, our techniques can be made even more applicable. For the youngest generation that consists of young dying objects, quick and simple breadth-first collection should be used. For older generations that contain promoted objects, our more expensive grouping can be afforded, because older generations get collected much less frequently, and the grouping effect will last for a long time.

We believe that grouping related functions is a very effective locality optimization because function objects can be easily identified and their access pattern is predictable. To group related functions, creation ordering is actually easier to implement. However, the creation order may not always be available. For Smalltalk, a predefined tree of classes already exists within the system, and more user-defined classes can be added to tree. In such an environment, our newly developed function traversal is more appropriate than creation ordering. The root of the Smalltalk class can be used as our root function, and a traversal on subclass pointers should transitively reach all the subclasses and group them quite close to each other. In practice, the polymorphic nature of class methods may make the grouping harder to apply, but customization techniques based on most common data types [ChUn89] should help finding a grouping in spite of the missing types.

Data structure grouping is essentially a much harder problem, because the shapes and connectedness of data structures vary so much. It is very hard (and perhaps unreasonable) to assume a typical operation on a complex data structure. Nevertheless, there are clearly identifiable flaws in common breadth-first ordering, providing a clear incentive to put more research effort into improving data structure locality.

9 Acknowledgements

We would like to thank Jon Solworth for providing access to the computer equipment needed in conducting this research, and to the referees who offered several helpful suggestions.

References

1. Andre, D.L.: "Paging in Lisp Programs," M.S. Thesis, University of Maryland, 1986.
2. Cattell, R.G.G.: *Object Data Management.* Reading, MA: Addison-Welsey.
3. Caudill, P., Wirfs-Brock, A.: "A third generation Smalltalk-80 implementation." *OOPSLA '86 Conference Proceedings*, pages 119-130, Portland, OR, September 1986.
4. Chambers, C., Ungar, D.: "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object Oriented Language." *Proc. SIGPLAN 89 Conference on Programming Language Design and Implementation*, June 1989.
5. Cheney, C.: "A nonrecursive list compacting algorithm," *Communications of the ACM*, 13(11):677-678, November 1970.
6. Courts, R.: "Improving locality of reference in a garbage-collecting memory management system." *Communications of the ACM*, 31(9):1128-1138, September 1988.
7. Fenichel, R., and Yochelson, J.: "A Lisp garbage-collector for virtual memory computer systems." *Communications of the ACM*, 12(11):611-612, November 1969.
8. Lieberman, H., and Hewitt, C.: "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, 26(6):419-429, June 1983.
9. McCarthy, J.: "Recursive functions of symbolic expressions and their computations by machine, part I.," *Communications of the ACM*, 3(4):184-195, April 1960.
10. Moon, D.: "Garbage collection in a large Lisp system," *1984 ACM Symposium on LISP and Functional Programming*, pages 235-246, Austin, Texas, August 1984.
11. Rees, J.A., Clinger, W., et al.: "Revised Report on the Algorithmic Language Scheme," *Sigplan Notice 21*, .21,12 pages 37-39, December 1986.
12. Shaw, R.: *Empirical analysis of a LISP System.* PhD thesis, Stanford University, February 1986.
13. Stamos, J.W.: "Static grouping of small objects to enhance performance of a paged virtual memory," *ACM Transactions on Programming Languages and Systems*, 2(2), May 1984, pp. 155-180.
14. Steele, G., Sussman, G.: *The revised report on Scheme, a dialect of Lisp.* MIT Artificial Intelligence Memo, January 1978.
15. Ungar, D.: "Generation scavenging: A non-disruptive high performance storage reclamation algorithm." *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157-167, April 1984.
16. Ungar, D.: "The Design and Evaluation of a High Performance Smalltalk System," Ph.D. Thesis, UC Berkeley, UCB/CSD 86/287, March 1986.
17. Wilson, P., Lam, M., and Moher, T.: "Effective Static-graph Reorganization to Improve Locality," *Proc. SIGPLAN 91 Conference on Programming Language Design and Implementation*, June 1991.
18. Wilson, P., Lam, M., and Moher, T.: "Caching Considerations for Generational Garbage Collection," *1992 ACM Conference on Lisp and Functional Programming*, June 1992.
19. Zorn, B.: "Comparative Performance Evaluation of Garbage Collection Algorithms," Ph.D. Thesis UC Berkeley EECS Dept., 1989

This article was processed using the LaTeX macro package with LLNCS style

Allocation Regions & Implementation Contracts

V. Delacour

[1] Xerox PARC, 3333 Coyote Hill rd, Palo Alto, CA 94304, USA
[2] INRIA, Domaine de Voluceau, 78153 Le Chesnay Cedex, France

Abstract. The purpose of this article is to advocate the use of two intermediate level abstractions named *allocation region* and *implementation contract* as the interface of language-independent memory management systems (MMS).
The allocation regions are both sets of objects and memory domains (sets of pages or sets of segments.) The implementation contracts represent symbolically the capabilities of the memory management system: the set of object formats and collection policies it supports. These two concepts give an intermediate level of abstraction to describe and compare most existing garbage collected MMS. We go further and advocate a full support for this level of abstraction through a flexible memory management interface: the allocation becomes a two step process in which (*i*) an allocation region is created with an attached implementation contract, and (*ii*) objects are allocated in the region. We show how to implement the abstraction, and how to take advantage of it.

1 Introduction

The performance of memory intensive programs in virtual memory is known to be related to a so-called "locality of references" quality, which in turn must be in some manner related to the way the objects are disposed in the memory space, because many memory accesses result from traversals of the data structures.

An important differences between most MMS with a copying garbage collector and most MMS with a non-copying garbage collector is that the former ones often pack all kinds of objects together in a non-structured heap [1, 3, 20, 16, 4, 22], whereas the latter ones are frequently used with a structured heap in which objects of different kinds are sorted in different memory domains [23, 8, 7]. So, any comparison of a pair of algorithms that fall respectively in these two categories compares not only different collection methods (copying vs non-copying), but also two different ways of organizing the memory space[3].

A careful study of the existing garbage collected MMS shows that the heap organization issues are in fact independent from those of garbage collection methods: nothing precludes building a Mark&Sweep (MS) collector for a non-structured heap, or (perhaps more interestingly) a copying collector for a structured heap. Moreover, we observed that most garbage collection algorithms would easily accommodate variable heap structures.

[3] Making it difficult to draw any conclusion concerning either the relative merits of the collection algorithms, or the relative merits of the heap organizations [24].

The two abstractions and the language independent MMS interface I propose in this article contribute to the field in several ways:

1. As a description language, they are general enough to describe and compare in a single framework very different MMS implementations.
2. As a target language they generalize the interfaces offered by the existing MMS in a way that opens new possibilities, such as the static or dynamic control of the heap structure, which is also a control over the VM usage: the proposed MMS interface gives ground to real-sized experiments in that direction.
3. The language-independent MM interface I propose in this article will help make different languages inter-operate by providing a single back-end to different type systems.
4. As a conceptual framework, the proposed concepts provide guidelines for the implementor, which may help make the implementation of the MMS both efficient and clear (this is admittedly a matter of personnal judgement).

1.1 Organization of the Paper

The next section (Sec. 2) presents the two abstractions of allocation regions and implementation contracts, with a proposed interface for language independent memory management systems. An example set of contracts is presented in Sec. 3. The following three sections of the paper proceed mainly from lower to higher level: Section 4 is mainly aimed at MMS implementors and shows how to implement the abstraction with different kinds of garbage collection policies, including the approximate depth-first, region-preserving copying policy I implemented as part of the K2 Implementation and Compilation Kit[4] (this is an example of copying collector for a structured heap.) Section 5 is aimed at language implementors (ie: clients of a MMS with a regions interface) and presents several basic strategies for structuring the heap by distributing the objects amongst regions. A prospective section (Sec. 6) cites research directions related to memory management, that could express results in terms of regions. In Sec. 7 I give credit to related previous work.

2 Allocation Regions and Implementation Contracts

2.1 Background

It is a fact that many MMS implementations with garbage collection actually deal with multiple object formats and multiple memory management policies in separate memory domains, but few or none of them take advantage of these capabilities to offer a flexible and general interface: they are either rigidly tied to a particular language, or in any advent use some fixed policy to segregate the objects in separate populations.

A common characteristic of the existing implementations is that, when multiple memory domains are used, one can associate a particular set of invariants and assertions to each of these domains. Typical such invariants and assertions include:

[4] The K2 Kit is made of two distinct parts: (*i*) a compiler for an intermediate-level language due to N. Séniak [19], and (*ii*) a MMS [10].

- every object in the domain has the *same size* (KCL's cell types [23], Boehm and Demers' garbage collector [7]),
- every object in the domain bears an *object descriptor* of some kind, giving the garbage collector an uniform way to make its decisions (SRC Modula-3 [17], KCL's cell types, most implementations which have a copying collector, most implementations, at least in one memory domain...),
- every object in the domain is *explorable* (or non-explorable) by the garbage collector (Boehm and Demers' garbage collector),
- every reachable object is *pointed by exactly one cell*, and bears a *back pointer* to that cell, (relocatable bodies in KCL, or in Le-Lisp15 [8]),
- every object in the domain is a *two words cell*, and each of the two words is a *tagged value* (Le-Lisp15's list cells area),
- the objects in the domain *must not be moved*, for they can be pointed from the outside (SRC Modula-3's non-collectable objects),
- etc.

I will call *implementation contract* the set of assertions attached to a memory domain.

2.2 Implementation Contracts

The implementation contract attached to a memory domain explicits the way the objects are laid out, and thus prescribes how the garbage collector is to explore the live data structures contained in the domain. Nothing precludes coding the contract directly with a GC exploration method. Depending on the particular type of garbage collector used, the way live objects are promoted (copy in the alternate semispace, shift-compaction, marking in a bitmap...) may be part of the contract, although this is not mandatory: one may easily think of garbage collection policies in which compaction does not occur at every collection cycle. On the other hand, the object format does belong to the implementation contract, because the garbage collector has to "understand" the object's layout to explore it correctly[5].

Conservative Collectors. We make here a special case of garbage collectors, only to show that, after all, they are no special case, for they can be described in terms of contracts. This section gives the peculiarities of parts of the implementation contracts in such collectors.

When the objects in a memory domain may be designated by ambiguous pointers, an identification method is usually part of the contract: given a (possible) pointer in

[5] Some may argue that in some contexts a garbage collector may need no context-independant information at all (eg: *tags*) to perform the exploration of the live graph of objects (for ex. [2]). This certainly *may* be, but remains to be seen. In practice also, such a claim can be made only within a particular type system, which does not suit our objectives, since we want to build a language-independent memory management interface.

the memory domain, the identification method tells (*i*) wether an allocated object is designated by the pointer, and if so (*ii*) the base address of the object.[6].

For example, if every object in a given memory domain has the same fixed size, and if the offset of the first object in the domain is known, a simple remainder operation gives the identification method. This method is used for example by H-J. Boehm and A. Demers [7], and by B. Schelter in AKCL's conservative garbage collector [18][7].

With conservative garbage collectors as with "regular" ones, stating explicitly the implementation contracts attached to the memory domains allows one to describe easily the strategies, and to compare the choices made by the implementors.

Discussion. It is worth noting that, whereas the set of object formats is in some manner "buried into" the implementation of a memory management system, the actual division of the memory space in separate domains is *mostly arbitrary*: adding an extra domain changes virtually nothing to the implementation problem, as long as no new contract is supported. This leads us to a straightforward generalization, namely the *allocation regions*.

2.3 Allocation regions

We define an *allocation region* to be a memory domain (set of pages, or of segments), to which is attached an *unique implementation contract*. Moreover, we will say that any allocation takes place in a region. Depending on the MMS capabilities, regions may be created at runtime, and destroyed at runtime (which is an unsafe operation, unless is it is the result of some automated deduction on the program).

2.4 Language-Independent Interfaces

The two abstractions described previously allow one to describe in simple terms existing implementations. A generalization yields a language-independent, simple and high-level interface for memory management systems. Given a set of implementation contracts supported by the system, the following two operations will be provided:

- creation of a region (with an attached contract),
- allocation of objects in a region.

Among the desirable extensions of this minimal interface are the following:

[6] In practice, different identification methods may be used, depending on some "degree of ambiguity" of the pointer; eg, a pointer located in an execution stack might be considered "more ambiguous" than a pointer located in an allocated object.

[7] Some conservative garbage collection algorithms do not even make use of an identification method: for example, one of J. Bartlett's first mostly copying GCs [4] used to "fix" and explore any object contained in a page that was designated by an ambiguous pointer (this approach is acceptable because in Bartlett's implementation only the stack may contain ambiguous pointers; in Boehm and Demers' one, any word in a live object is considered to be an ambiguous pointer, so the above approach would most likely lead to excessive retention.)

- destruction of a region,
- region predicates.

The interface is language independent because the contracts concern only implementation properties of the objects, regardless of higher-level considerations such as the type system, etc (this is why we call these abstractions *intermediate-level*.) On top of such an interface, one can easily either (*i*) offer a `malloc`-like facility, by automatically mapping the objects in regions – thus *hiding* the implementation differences between different populations of objects, or rather (*ii*) let this interface be the MMS interface, thus providing a *target language* to express implementation choices, typically done at the language level, but possibly also on a per-application basis.

The following section describes in more details a MMS interface in C.

2.5 The complete abstract regions interface

Figure 1 presents an example interface, as may be offered by a portable memory management system. The `malloc` friends are omitted. One goal is to provide as simple an interface as can be made. Inevitably, parts of this interface are implementation-dependent, because all memory management systems do not support the same set of object formats. I signal the places where some amount of implementation-dependent code may occur by a comment: `/* imp. dep. */`.

```
typedef struct
{ /* imp. dep. */ } region_t;

typedef enum
{ /* imp. dep. */ } contract_t;


region_t *create_region(contract_t c /*, imp. dep. */);

void *allocate(region_t *r /*, imp. dep. */);

void destroy_region(region_t *r);

region_t *region_of(object_t obj);
```

Fig. 1. A proposed C interface for a memory management system supporting regions.

Region Creation. When a region is created, an implementation contract must be associated to it. Typically, most of the implementation contracts will demand some extra parameter(s), such as the size of the objects in the region (or a range

of sizes), and possibly some boolean values expressing properties (one may think of the "pointer-free" property).[8]

Object Allocation. allocating an object in a region obviously asks for the region as a parameter. Some additional parameters might be needed, such as a size and/or a tag of some sort, depending on the contract, and thus on the implementation.

Destruction of a Region. this means deallocating all the objects in the region.

Region Retrieval. this function returns the region the object belongs to (may be used to implement type predicates).

In addition to the proposed interface, some extensions such as the static allocation of regions together with allocation macros may provide increased efficiency. A MMS implementation will typically offer a set of predefined implementation contracts, such as the ones we already find in existing implementations. The following section describes briefly the set of contracts offered by a MMS I'm currently developping.

3 Example Set of Contracts

This section describes briefly the set of contracts offered by a core Mark&Sweep (MS) GC designed to be used either within PCR [21] as a generational, mostly parallel GC (using H-J. Boehm's algorithm [6]), or independently as a simple Mark&Sweep collector.

First, as in [7], the system distinguishes between pointer-containing (composite) and pointer-free (atomic) objects. Composite objects are explored according to an *exploration mode* (Table 1). The first three modes both view the objects as unstructured intervals of memory; they differ by the way they recognize pointers in these intervals. The most liberal conservative mode (conservative2) accepts interior pointers, and is intended for the root objects (execution stacks in particular). These three modes ask very little from the client of the MMS: no particular knowledge of the internals of the system is required. The *structured* mode asks for a specialized exploration function, which should obey a precise protocol in order to preserve desirable properties of the exploration process (such as proper tail-recursion): these functions are meant to be automatically generated by a compiler (from provided templates), taking advantage of some knowledge on the objects structure.

Mode	*Description*
Tagged	Each word in the object is a tagged value (either an immediate value or a reference)
Conservative1	Each word is an ambiguous reference, interior pointers are not accepted
Conservative2	Interior pointers are accepted
structured	The object is explored by a specialized exploration function
pointer-free	No exploration (pseudo-mode)

Table 1. Exploration modes in a conservative MS collector.

[8] Some may argue that an object oriented hierarchy of region types would be the right way to go. That is probably right. Just waiting for a suitable, low-level language to happen.

Finally, each region contains either *small* objects, in which case a unique size is part of the contract, or *big* objects (of any size bigger than half a page.) The page size is a parameter of the implementation. It is expected that in practice any *structured* implementation type will fall in the small objects category for any reasonable page size; so the distinction between small and big objects should not be a hindrance. Table 2 summarizes the contracts offered by the MMS, and their parameters.

	Atomic	Composite			
		Tagged	Conservative1	Conservative2	Structured
Large	*no parameter*	*no parameter*	*no parameter*	*no parameter*	expl. func.
Small	size	size	size	size	size, expl. func.

Table 2. The contracts and their parameters in a conservative MS collector

Internally, the system makes a difference between power-of-two sizes and other small sizes. Also, a non-homogeneous region of small objects is used internally for allocating bit-maps (with a buddy system). Future extensions may include a public non-homogeneous mode for small objects. With the current set of contracts, non-homogeneous populations must be implemented with sets of regions. As in [7], the system offers a default `malloc`-like interface, using an array of predefined regions for various sizes of small objects, and another predefined region for the big objects; these predefined regions use the "conservative1" exploration mode. As in [7], a similar default interface for atomic objects is also offered (implemented with a second set of predefined regions).

4 Implementing the Abstraction

This section is mainly aimed at MMS implementors. I'll show here how to implement the abstraction in a garbage-collected system, assuming (for the sake of brevity) one wants to modify an existing implementation. I'll discuss successively the Mark&Sweep, relocatable bodies, and copying collection policies.

4.1 Mark&Sweep Collectors

The exploration part of a garbage collector does not directly deal with the way object are segregated in separate populations: it is only concerned with the object formats, and more generally by the implementation contract: (*i*) where to find the mark bit, given the address of the object – either in the object, or in a separate bit-map, (*ii*) how to enumerate the explorable slots of an object, etc. So, the main part of the garbage collector itself does not have to be modified at all because it already has support for multiple contracts and in practice it has no concern for regions.

The sweep phase must construct one free list per region, so the region is a parameter to the sweep functions (instead of the size only, as in [7] or [23]). Since the sweeping code is already capable of dealing with multiple object sizes, no particular difficulty arises. Lazy sweeping may be achieved by linking, for each region, the pages to be swept.

4.2 Relocatable Bodies

Separate memory domains for relocatable bodies are often used with Mark&Sweep garbage collectors, as an easy way to deal with the fragmentation induced by multiple objects sizes (KCL [23], Le-Lisp [8].)

Sticking to the implementation, low level point of view, it may be worthwhile considering the relocatable bodies as plain objects (and not as merely the *second* parts of objects.) Part of these objects' contract is that they are pointed once, and once only. Separate regions of relocatable bodies are implemented as easily as separate regions of Mark&Sweep cells. I will not describe that further.

4.3 Region Preserving, Copying Collectors

With copying garbage collectors, the regions (ie: the grouping of the objects) should be preserved by the copying process: one copying pointer per region is required.

The copy space associated to each region needs not be allocated before the copying process actually takes place, so the respective sizes of the regions may vary according to the program's needs. A convenient solution is to have a pool of free pages, and a separate count of available pages. Initially, all the pages in the pool are available, so the count is equal to the size of the pool. Each time a page is allocated to a region of copiable objects, two pages are discounted (instead of one), so that enough pages remain in the pool when the count reaches zero. At this moment either a garbage collection must be performed, or the heap must be expanded[9]. I adopt this approach in the portable garbage collector included in the K2 Implementation and Compilation Kit (next section.)

4.4 Approximate Depth-First Copying Collectors

Independently from P. Wilson [22], I devised and implemented in 1990-91 a more general adaptation of Moon's scheme [16] in a mostly copying garbage collector with ambiguous roots and full support for regions [10]. This garbage collector is one of the two parts of the K2 Implementation and Compilation Kit[10].

The region-preserving, approximate depth-first copying is implemented as follows: as described above (Sec 4.3), one copying pointer is required per region, so there may be one copying page per region of copiable objects. To implement Moon's approximate depth-first copying, one "local" scavenging pointer per region is used in addition to the global, "Cheney" scavenging pointer [9]. (To avoid scanning some locations twice, the last location reached by a local scavenging pointer in a copying page is saved when that page is replaced.)

4.5 Implementing Dynamic Regions

The simplest way to implement the regions is probably to use plain runtime objects (as opposed to diffuse informations in the code) identified by a pointer. The addresses

[9] Such a decision can naturally be made *before* the count reaches zero, according to some gc-triggering policy (but this is an out of scope issue.)

[10] The other part is a compiler for an intermediate language designed by N. Séniak [19].

of statically defined allocation regions may be included inline in some portions of the code (in allocation macros for example) for efficiency; some portions of the memory management system still have to cope with variable regions (and thus have to dereference the pointer), but the performance cost of the abstraction is in practice close to nought. Garbage collecting the dynamically created region objects poses no particular problem except that some portions of the garbage collector's code may keep a pointer on a region object for some time, so it may be better not to move the region objects. At the end of a GC, the regions which (*i*) do not contain any object, and (*ii*) are not pointed as objects themselves are dead and can be collected.

In K2's gc the regions are identified by a small number. A "page tag" packs in one word the region number and two attributes *zone* and *mode* representing both the implementation contract and (for copiable objects) the logical *from* or *to* space the page is in. Still, the GC accesses the regions objects through an array of pointers, to allow for the dynamic creation/deletion of regions. The argument for such a solution is that almost all the information needed by the gc to make (multi-way) decisions is packed in a very compact way, and retrieved in only one memory access. In retrospect however I think that using a pointer as a region identifier makes the implementation simpler and is worth the extra word in the page headers. My new (in progress) garbage collector uses this latter solution.

Having the abstract regions materialized as runtime objects has the advantage of inspectability, and makes the implementation more clear. The region objects will typically hold the free lists (Mark&Sweep regions), or the allocation pointers (copying regions), and a materialization of the contract attached to them : size and shape of the objects, some property bits, specialized exploration functions, etc. Some of these informations may be redundantly present more superficially (say, in page descriptors) for efficiency reasons. Using pseudo regions for the free pages (or segments), and for the "outside world" (holes in the address space, data-structures private to the gc) may help make the garbage collector simple and efficient, for most of its actions may be driven by some compact representation of the contract, in a so-called "object-oriented" manner.

5 Implementing with the Abstraction : Distributing the Objects Into Regions

This section now turns to the language implementor's point of view. The actual way objects are segregated into separate regions is quite naturally the result of some trade-offs. The present section reviews a few aspects of the question.

5.1 Implementation Choices

The very nature of certain kinds of objects commands partly the way they are implemented. For example, it is common in Lisp implementations to implement differently (and in different memory areas) character strings and list cells. Such a choice is easily expressed in terms of regions.

In quite the opposite manner, many implementations with a copying garbage collector pack all kinds of objects together, and thus do not segregate the objects

according to their implementation. As a consequence (to maintain the unique contract requirement), they use a very general encoding of the objects formats, often with a header word (SRC Modula-3, SML-NJ [1]), but not always (Lucid Common Lisp's list cells don't have a header word, but a distinguished pointer tag [20]). Using a header word in each object trades some space for the simplicity of the implementation.

With its regions mechanism K2's collector, as any other efficient implementation, allows one to use simultaneously very efficient, dedicated data formats (for list cells for example), and more general, flexible formats (with, say, interpreted headers); unlike other efficient memory management systems, K2's MMS and any MMS with full support for allocation regions is not related to any type system, and allows one to express the space/efficiency trade-offs in a high-level and flexible manner.

5.2 Mapping Types to Regions

An obvious way of segregating the objects in regions is to use a mapping from types to regions. There is no particular reason why different types should be implemented in different regions, or conversely why a single type (at the source language level) could not be implemented in several regions.

As pointed out in previously, there is a space/other trade-off when a region's contract supports multiple objects formats, for an object descriptor of some kind may be needed. Among the reasonable uses of such "inhomogeneous" regions, one may think of:

- ML types with multiple constructors (aka variant records),
- object oriented type subtrees (idem).

In a statically typed world, some automatic analyses may help determine a suitable mapping of types to regions, tailored for each particular program.

5.3 Mapping Sizes to Regions

Mapping sizes to regions is the other obvious way of segregating the objects, for memory areas (typically: pages) containing one-sized objects are easy to deal with efficiently at the (MMS) implementation level. KCL uses this mapping for its so-called cells types. Boehm and Demers also do (although they also distinguish between pointer-free and non pointer-free objects). Boehm and Weiser initially reported some fragmentation problems when many sizes were present in an application (because a single object of one size mobilizes an entire page). A suitable rounding method now remedies this inconvenience[11].

[11] The rounding method used (due to R. Atkinson) is very clever: it consists in rounding up to 3, (or 4 , or n) upper significant bits. The maximum loss of space with this method is quite acceptable 2^{1-n} (25% with 3 bits), and the number of sizes is greatly reduced: 33 sizes between 0 and 512 with 3 bits. As a bonus, there is no loss for 2^n first sizes!

5.4 Discussion

The trade-offs for distributing the objects amongst regions may be different in small heap and in large heaps. Large heap sizes may contain several thousands of pages, far exceeding both the number of object sizes and the number of types present in the running program. In such a situation, if no other information is available, the finest distribution based on both sizes and types may be a good one. In other words there's no particular reason to put the objects of one type in the same set of pages than the objects of another type, unless we know that (*i*) the two given types of objects are strongly related, and (*ii*) related objects have great chances to be together in memory. In smaller heaps however, a fine grained distribution may introduce too much fragmentation, that is, a less efficient usage of the memory space.

The above discussion as well as the preceding sections deals mostly the static distribution of the objects in regions. Keeping in mind our goal, which is to group the related objects, and conversely to separate unrelated objects, we can think of allocating the objects in different regions according to less ordinary policies:

- Dynamic criteria : replace temporarily the allocation region in which to allocate some kinds of objects. For example, in a compiler that would process one procedure at a time, one may wish to (partly) switch regions each time a new procedure is to be compiled.
- Allocation site : allocate some kinds of objects in different regions, according to their allocation site (the place in the source code where the allocation occurs).
- etc.

It should be noted that switching regions, either at places or at times carries no semantics from the client's point of view (unless he makes use of predicates on the regions). Unless some regions are explicitly destroyed, switching regions does not affect the correctness of a program: in that respect, it can be compared to pragmas in compiled code: pragmas can help (or hinder), but should not affect the semantics.

The grouping policies described above apply to individual programs, rather than to language implementations; They should probably be used as the result of some automatic program transformation, rather than explicitly. The next section presents briefly two directions of research for such automatic program transformations.

6 Regions and Automatic Program Analysis

This section is mainly prospective, and expectedly far incomplete. Its purpose is to hint research directions on "how to improve the memory usage automatically". The two approaches I describe below would be much less promising if the results of the analyses could not be expressed in terms of regions. The region abstractions provide the natural back-end to take advantage of such analyses.

6.1 Inference of Effects

Some automatic program analyses, initially aimed at automatic parallelization (for example by Gifford and al. [14]), try to determine a partition of the set of objects

into separate and "non-interacting" populations. By allocating these populations in different sets of pages (ie: in different regions), we can reduce the dilution of the active data within the working set, and thus reduce the size of the working set. For practical purposes, only an approximation of this analysis is probably desirable, because the "non-interacting" requirement may be too strong.

6.2 Lifetime Analysis

Other analyses try to determine automatically the lifetime of objects or sets of objects [11], and may already produce enough information to insert some deallocation statements at places in some programs. However, the explicit deallocation of individual objects is not proven to be more efficient than the automatic reclamation provided by a garbage collector. On the other hand, it is often the case that a whole lot of temporary data structures are known to be dead a certain execution points. In such a situation, creating and deleting dynamically a dedicated set of regions for these objects would be the appropriate way of taking advantage of this knowledge.

6.3 Discussion

Both of the theoretical approaches described above are the matter of active research. They both attack a non-decidable problem, but it is a fact that most actual programs present many regularities that can be exploited automatically. This is especially true when strongly typed programming languages such as ML are used. As a consequence, these approaches should be considered promising and modern memory management systems should provide some kind of abstract region support to help exploit them.

7 Related Work

Although not present as such in existing implementations other than K2's garbage collector, the allocation region abstraction is strongly hinted in some previous work.

Both KCL's memory management system and Boehm&Weiser's article were undoubtedly valuable sources of inspiration: the excellent performances of KCL on memory-intensive programs more than demonstrate the importance of the memory management policy in the global performances of an implementation, and shows that some structuration of the memory space, with separate populations of objects is probably a good option. KCL's system is completely dedicated to Common Lisp and thus could not serve my purposes (building a portable, language independent memory management system). I was nevertheless impressed by its simplicity and its fitness: K2's memory management system may be viewed as a generalization of KCL's one. The same remark applies to H-J. Boehm and A. Demers' portable garbage collector, for their distinction between *atomic* and *composite* objects, as well as the use of homogeneous populations of objects lead in some manner to the notion of a language-independent implementation contract attached to an abstract region.

The whole idea of considering populations of objects as memory domains is probably quite ancient. For example, Bishop [5] uses *areas* as the basic unit for partial

garbage collection. A recent paper by D. Hanson [12] describes the benefit of allocating through a notion of abstract population (which is there called *arena*). Hanson shows the advantage of allocating in such arenas in applications where the lifetime (more exactly the death-time) of entire sets of objects may be predicted easily: he shows that in the `lcc` compiler, a large part of the allocated objects are known to be dead when the compilation of a function is finished: by allocating them in a dedicated *arena* he is able to deallocate them all at once. Also, it must be noted that the Cedar language [15] does include a notion of zones, close to both our regions and Hanson's arenas.

The concept of a language-independent memory management toolkit has been made public by R. Hudson &al. [13], with an approach almost orthogonal to ours: the interface offered by this toolkit focuses on generations management issues, whereas all the memory space may be viewed as a single allocation region.

8 Conclusion

The concepts we have presented in this paper give an intermediate level of abstraction to describe and compare most existing garbage collected MMS. When these concepts are also supported by a MMS through an abstract regions interface, this level of abstraction becomes as well a level of control: the client of the memory management systems may control either statically or dynamically the way objects are grouped in memory.

We have presented in this paper how these concepts are only a slight generalization of existing techniques, and how they could be implemented in existing or in new memory management systems. We have presented how to use them in a language implementation, and how promising theoretical approaches on memory management could be put to use through the abstract interface we have proposed.

9 Acknowledgements

This paper has greatly benefited from comments by H-J. Boehm.

References

1. A. Appel. A runtime system. Technical report, Princeton University, February 1989. (DRAFT).
2. A. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2, 6 1989.
3. A. Appel. Simple generational garbage collection and fast allocation. *Software — Practice and Experience*, 19(2), February 1989.
4. J. F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, Digital Western Research Laboratory, Palo Alto, Cal., February 1988.
5. P. B. Bishop. *Computer systems with a very large address space and garbage collection.* PhD thesis, MIT, Cambridge, MA, May 1977.
6. H-J. Boehm. Mostly parallel garbage collection. In *SIGPLAN'91*, pages 157–163. Xerox PARC, 1991.

7. H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9), September 1988.
8. J. Chailloux and al. *Le-Lisp de l'INRIA, version 15.2. Le manuel de référence.* INRIA, Le Chesnay, 1986.
9. C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.
10. V. Delacour. Gestion mémoire automatique pour langages de programmation de haut niveau. Thèse de l'Université Paris 6, LIX-INRIA, Paris, juin 1991.
11. A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Annual ACM Symposium on Principles of Programming Languages*, San Francisco, January 1990. ACM.
12. D. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software — Practice and Experience*, 20(1), January 1990.
13. R. Hudson, J. Eliot Moss, A. Diwan, and C. Weight. A language-independent garbage collector toolkit. Technical Report 91-47, University of Massachusetts, Amherts, 1991.
14. P. Jouvelot and D. Gifford. Parallel functional programming: the FX project. In M. Cosnard and al, editors, *Parallel and distributed algorithms*, pages 257–267. North-Holland, 1989.
15. B. Lampson. A description of the Cedar language: a Cedar language manual. Technical Report 15, Xerox PARC, Palo Alto, CA, 1983.
16. D. A. Moon. Garbage collection in a large Lisp system. In *Symposium on Lisp and Functionnal Programming*, Austin, Texas, 1984. ACM.
17. E. Muller and B. Kalsow. SRC Modula-3, version 2.05. Technical report, DEC, Palo Alto, CA, 1992. Included in the public distribution of SRC Modula-3.
18. W. Schelter. AKCL : Austin Kyoto Common Lisp. University of Texas, Austin, December 1987. *first release (akcl-1).*
19. N. Séniak. Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels. Thèse de l'Université Paris 6, LIX-INRIA, Paris, octobre 1991.
20. P. G. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Bachelor's thesis, MIT, Cambridge, MA, 1988.
21. M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime approach to interoperability. In *ACM 13th symposium on Operating Systems Principles*, December 1989.
22. Paul R. Wilson. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *SIGPLAN'91*, pages 177–191, june 1991.
23. T. Yuasa and M. Hagiya. Kyoto Common Lisp report. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1986.
24. B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, UCB, Berkeley, CA, 1989.

This article was processed using the LaTeX macro package with LLNCS style

A concurrent generational garbage collector for a parallel graph reducer

Niklas Röjemo
Department of Computer Science
Chalmers University of Technology
412 96 Göteborg, SWEDEN
rojemo@cs.chalmers.se

This paper describes a garbage collector for an implementation of a lazy functional language using parallel graph reduction. The garbage collector is an extension to the Appel-Ellis-Li garbage collector. The extension consists of two parts:

Firstly, generations has been added: this often decreases the garbage collection time with nearly 20%. This shows that generational garbage collection is useful even for a lazy functional language implemented with graph reduction (which performs a lot of updates).

Secondly, the Appel-Ellis-Li garbage collector has been changed so that it can collect garbage processes (which is essential if we do speculative evaluation).

Keywords: garbage collection, graph reduction, concurrent, generations

Introduction

Normally, most garbage collectors introduce pauses in the normal execution of the program. We want these pauses to be both rare and brief; i.e., we want the collector to reclaim as much memory as possible with a minimum of work. This is achieved if the garbage collector concentrates on nodes that are likely to be garbage. Another way to decrease the annoyance of the pause is to collect garbage without stopping the evaluation. The garbage collector described in this paper uses both tactics.

Speculative evaluation is a nice programming concept which makes it easier for the programmer to use a parallel machine, by using free processes to evaluate expressions that might be useful. We do not want to burden the programmer with the task of detecting, and killing, speculative processes that do not contribute to the final result of the program. The garbage collector described here will detect, and kill,

these garbage processes when collecting garbage.

Generational garbage collection, i.e., the idea to concentrate garbage collection on the nodes that are most likely to be garbage, is not new. It is used in [Lib83] [Moo84] [Ung86] [App89] among others to great success. They all use the observation that young nodes are more likely to be garbage than old nodes. By separating young nodes and old nodes in different heap spaces it is possible for the garbage collector to concentrate on the nodes that are most likely to be garbage. The garbage collector does not need to work on the old nodes, except when there is not enough garbage available to recycle among the new nodes (which happens rarely). There is a hidden assumption here that nodes do not point at nodes younger than themselves as this would force the garbage collector to scan the old nodes for pointers to young nodes when deciding which young nodes that are garbage. This assumption is mostly true for object oriented languages, LISP and strict functional languages, the few exceptions to the assumption that exist in an implementation can be handled with a small overhead, which is paid when nodes are updated.

We are interested in the implementation of lazy functional languages using graph reduction [PJ87] and parallel graph reduction [AJ89]. Unfortunately, there is a serious mismatch between graph reduction and generational garbage collection: a fundamental operation in a graph reducer is to update nodes representing unevaluated expressions, and this is done frequently, which is a problem as the overhead of checking pointers is paid for every update.

This paper describes a generational garbage collector that can work with an implementation of a lazy functional language using graph reduction. With some modifications, described below, generational garbage collection is useful also with graph reduction, according to our measurements. The trick is to be selective when tenuring a node (promoting it to an older generation).

Collecting garbage processes has been done before in [Hud82] [Hud83], among others. Hudak gives an algorithm for concurrent garbage collection that can collect garbage processes. He does however rely on cooperation between the mutators[1] and the garbage collector. The code for evaluation must therefore contain extra code for garbage collection which might slow-down the evaluation even when no garbage collector is running. Augustsson [Aug90] does not need any cooperation but instead stops the evaluation during garbage collection.

---

1. The user's processes are called mutators because they change all the data that the garbage collector has cleaned up.

The garbage collector described in this paper uses the Appel-Ellis-Li garbage collector [AEL88] as its base, but it can collect garbage processes with a small loss of concurrency between mutators and garbage collector. The possibility to include generational garbage collection in the Appel-Ellis-Li garbage collector is mentioned in [AEL88] as future work, and have been implemented in the collector described in this paper.

Generational garbage collection

A graph reducer (for a lazy language) normally creates many short lived objects in the heap. These objects, for example function applications which are evaluated shortly after creation, are what the garbage collector should concentrate on. There may also exist long lived objects that never change, e.g., a table of reserved words in a compiler, which we do not want to spend any garbage collection time on.

A way to distinguish these two types of object is to allocate all new objects in one heap and then during garbage collection move old nodes to another heap. This concept can be taken further by using more than two heaps and moving objects to successively older heaps if they survive garbage collection in younger heaps. The implementation in this paper uses only two generations as it simplifies the garbage collector.

After we have decided how many generations to use, the problem of how to decide when to move a node to an older generation appears. The normal method is to count the number of garbage collections that the node has survived. The age when a node is tenured varies from zero (i.e., a node is old if it survives any garbage collection [App89]) and upwards. To prevent nodes that were created just before garbage collection to be tenured our implementation uses an age of one (i.e., the second time the garbage collector encounters a node it is considered old). As the implementation uses a large heap, normally 4-8 Mbytes of combined heap for the new and the old generation, survival of two garbage collections indicates that the node is quite stable.

Unfortunately some pointers among the old nodes will point at younger nodes. These pointers must be followed when the garbage collector decides which nodes that are garbage. A table of pointers, called the exception table, to these troublesome pointers are therefore maintained, so that the garbage collector can find them with-

out scanning all old nodes. A slight variation of this method is used in [Ung86], another method is to use an indirect table [Lib83]. If it is possible to update a pointer in an old node then a test if the new pointer points at a young node is necessary, in which case the address of the new pointer is entered in the exception table. This test can be done in software if updates are rare, otherwise this is a big problem.

A graph reducer evaluates a program by successively re-writing the graph. This causes a fundamental mismatch between generational garbage collection and graph reduction, as graph reduction often updates pointers in previously allocated nodes. These new pointer values might point at young nodes and can therefore put a heavy load on the exception table. The overhead induced by checking every pointer when updating a node is also expensive, as in addition to a simple store the code has to check if the position stored in is old, and in that case check if the node pointed at is young, in which case the updated address must be added to the exception table. The solution proposed here is to only tenure nodes that are known never to be updated. This condition restricts the possibility to use generations, but our measurements clearly indicate that there are enough nodes that will never be updated to make generations useful. All constructor nodes belong to this group, as only what they point at can be changed by the mutators, never the pointers themselves.

In fact, the only nodes that can be updated during normal evaluation are nodes that represent un-evaluated applications with all their arguments available. By never tenuring these nodes the need to update old nodes disappears. This removes the need to check pointers when updating nodes, which makes generational garbage collection feasible for a graph reducer.

There will still be pointers in the old heap that points at young nodes, but these pointers are all created when the garbage collector moves young nodes to the old heap. The penalty for checking these pointers and, if necessary, add them to the exception table is not paid during normal evaluation, and does therefore not make a big impact on the total execution time. The number of these "trouble" pointers never exceeded 10% of the pointers in the old heap in any of the measured evaluations. This means that approximately one word out of 20 in the old heap contains a pointer that points at a young node. (The average node consists of four words where two words are pointers.)

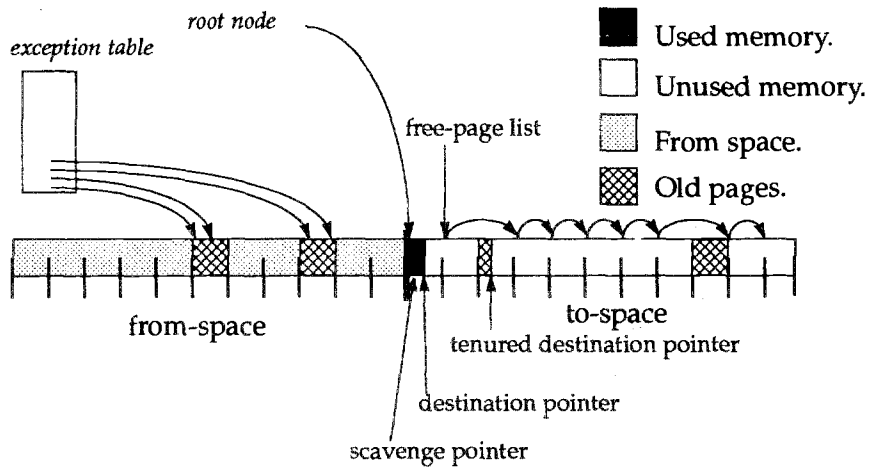


Figure 1 *The heap just after a minor garbage collection when the garbage collector has copied the root-node and all young nodes accessible from the old nodes to to-space.*

Implementation

The garbage collection works as the Appel-Ellis-Li garbage collector [AEL88] but uses two destination pointers instead of one (Figure 1), so that young and old nodes can be separated during garbage collection. The additional destination pointer, called tenured destination pointer, are used when a node that should be tenured is found in from-space. The node is copied to the position given by the tenured destination pointer but is treated as usual otherwise. This will create pages that only contains tenured nodes. The garbage collector does not need to look at these pages at the next garbage collection as they are old and therefore have probably not turned into garbage. The only thing necessary is to change the "trouble" pointers, but these are easy to find because of the exception table. The mutators can therefore access the old pages nearly immediately after the flip of to-space and from-space. When too many pages are occupied by old nodes a major collection is done where all pages are garbage collected. Currently the limit is when 30% of the available heap is used for old nodes. The decision to use 30% is a compromise; a large old heap decreases the number of major collections, but increases the danger that not enough free memory is available when a major collection is needed. None of our test programs had any problem with the 30% limit. In fact, they still worked with a limit of 40%, but this did not give any significant speed-up.

Collecting garbage mutators

The reason collecting garbage mutators is necessary in this implementation is that the languages it is implemented for allows speculative evaluation. Speculative evaluation makes it easier to gain a higher utilisation on a parallel machine, as spare processors can be used to evaluate expressions that might be useful. The decision if such a mutator is garbage, i.e., does not contribute to the final result of the program, is related to the decision if a node is garbage. The garbage collector therefore seems to be the ideal candidate to find, and kill, these garbage-mutators.

It is easy to collect garbage mutators if the garbage collector stops all mutators during garbage collection and then decides which to restart after it knows which parts of the graph that survived the garbage collection. This does however not use the possibility to hide garbage collection by allowing the mutators to continue working during garbage collection.

The garbage collector described in this paper must also stop the mutators at the start of the garbage collection, but this garbage collector can resume a mutator as soon as the garbage collector reaches any node in the heap that the mutator is working on. Sharing can make this difficult, but a way around this problem is described at the end of the next section.

Implementation

The garbage collector described in this paper is implemented for the <v,G>-machine [AJ89], a parallel graph reducer for a shared memory machine. The work of a mutator is represented by a linked list of frame-nodes in the heap (Figure 2). Every frame node represents a function application, and includes pointers to all arguments needed. New mutators are created by starting speculative evaluations on e.g., function arguments that might be needed (Mutator B and C in Figure 2). These speculative evaluations terminate when they have evaluated their top node to head-normal form, or if the garbage collector kills them. Every mutator has a corresponding process-node in the heap which, among other thing, contains pointers to the top node and the node the mutator is currently working on.

A garbage collection (both a minor one and a major one) starts by stopping all mutators, which is done in the same way as in the previous garbage collector, and continues approximately as the Appel-Ellis-Li garbage collector does. The differ-

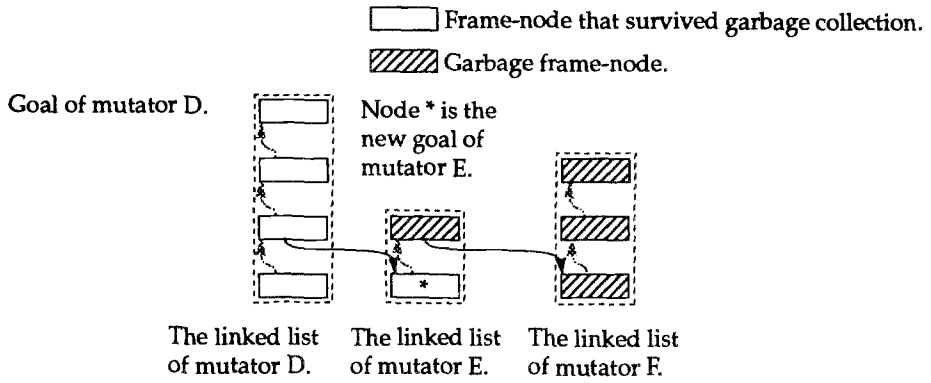


Figure 3 *Mutator D is resumed unchanged after garbage collection, mutator E is resumed but will stop after evaluating node * to weak-head normal form. Mutator F is killed by the garbage collector.*

ence is that our collector does not treat the registers of the mutators as root-pointers, and can therefore not resume the mutators as early as the Appel-Ellis-Li garbage collector can. Mutators are instead resumed, and their process-nodes are moved to to-space, as soon as the garbage collector finds any node in their linked list of frame-nodes. If the found node is the goal of the mutator then the mutator can be resumed immediately. The problem is if, due to sharing, the garbage collector finds another node in the list (e.g., the *-node in Figure 3). It is not possible at this moment to decide exactly how much of the linked list that mutator E works on that is not garbage, as the collector may later find a pointer that points at the goal node. It is however possible to say that at least all frame-nodes in the list up to, and including, the *-node is useful. The garbage collector therefore creates a mutator that has the *-

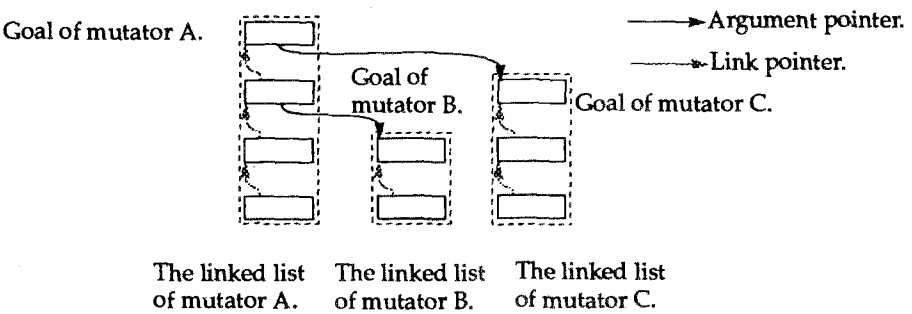


Figure 2 *A snap-shot of the linked stack with three mutators. Argument pointers which are not pointing at frame nodes under evaluation are not shown.*

node as its goal node and resume this mutator. If the collector later finds that the goal of mutator E is not garbage then it creates a mutator which has the same goal as E but waits for the result of the evaluation of the *-node is available. This waiting mechanism was already available in the <v,G>-machine where it was used to prevent many mutators from evaluating the same frame-node, which is a waste of resources.

All mutators that have not been resumed when the garbage collection has finished is killed. To kill a mutator means that its process-node is not moved to to-space.

Performance

All measurements are from a Sequent Symmetry, a shared memory multi-processor machine with 16 Intel 386 processors running DYNIX V3.0.17.9.

A problem with testing garbage collectors is that their performance depends heavily on the behaviour of the test program. The results included in this paper is summarized in Figure 4.

The figure shows that generational garbage collection is approximately as useful as concurrent garbage collection for these programs. This is very interesting as concurrent garbage collectors needs a parallel computer to achieve its speed-up, but generational collection works on any single-cpu computer.

Another piece of information in the table is that generational and concurrent garbage collection sometimes help each other (see gff and logic). One reason is that generational collection gives the resumed mutators earlier access to some nodes, as the old nodes do not need to be garbage collected before they are available to the mutators.

A description of the test programs now follows with more detailed graphs about how time varies due to different choice of page sizes. The reason for testing with different page sizes is that small pages are better for the concurrency, as pages will be available faster for the mutators after the start of a garbage collection, but are more expensive during allocation, as we need to grab a new page more often. Larger pages are cheaper when allocating, but increase the time before the garbage collector can release the page to the mutators.

The table shows: $1-\frac{E_{old}-E}{G_{old}}$

where:

E_{old} *the execution time when the previous garbage collector is used.*

G_{old} *the previous garbage collection time.*

E *the execution time with the new garbage collector with concurrent and/ or generational garbage collection enabled.*

	concurrent only	generational only	both theory	both meas.
logic	74%	94%	70%	64%
graph	78%	80%	63%	66%
gff	100%	82%	82%	75%
kwic	59%	56%	32%	49%

Figure 4 *The values shows the new garbage collection time as a percentage of the old garbage collection time. The theoretical value for both is calculated by:*
*concurrent only * generational only*
The best possible value is 0% which means that the garbage collection time is hidden by the normal evaluation.

logic proves a tautology in statement logic using natural deduction and breadth-first searching. The program creates an enormous tree which consist of a lot of old constructor nodes. It needs 4 garbage collection when using a heap of 8 Mb. Logic slows down if small pages are used, but this behaviour disappear when generational garbage collection is used. The reason is that the non-generational collector copies all nodes back and forth between the two spaces, but the generational collector soon consider the nodes old and do not move them any more. This is, despite the few garbage collections, a big win.

graph compiles a simple functional program into G-code. It needs 9 garbage collections when using a heap of 4 Mb. There is a nice behaviour for old nodes with only 645 pointers to young nodes among 410kb of old nodes. This is

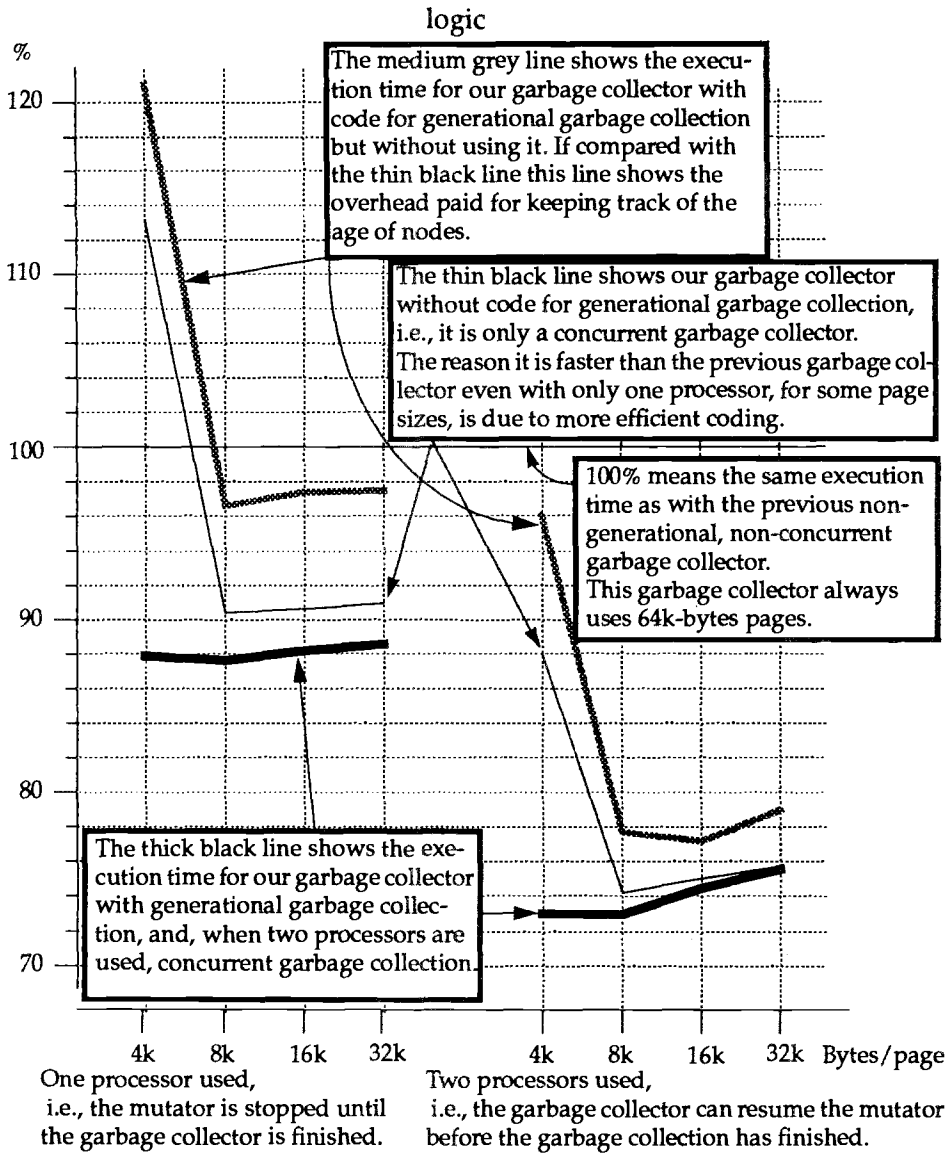


Figure 5 This graph shows the execution time of logic using the new garbage collector compared with the previous non-generational, non-concurrent garbage collector.

very few compared to the approximate 50000 pointers that exist in 410kb of constructor nodes. Graph was not possible to run with the previous garbage collector, when the measurements for Figure 6 was done, due to a bug in the current version of the LML-compiler for the <v,G>-machine.

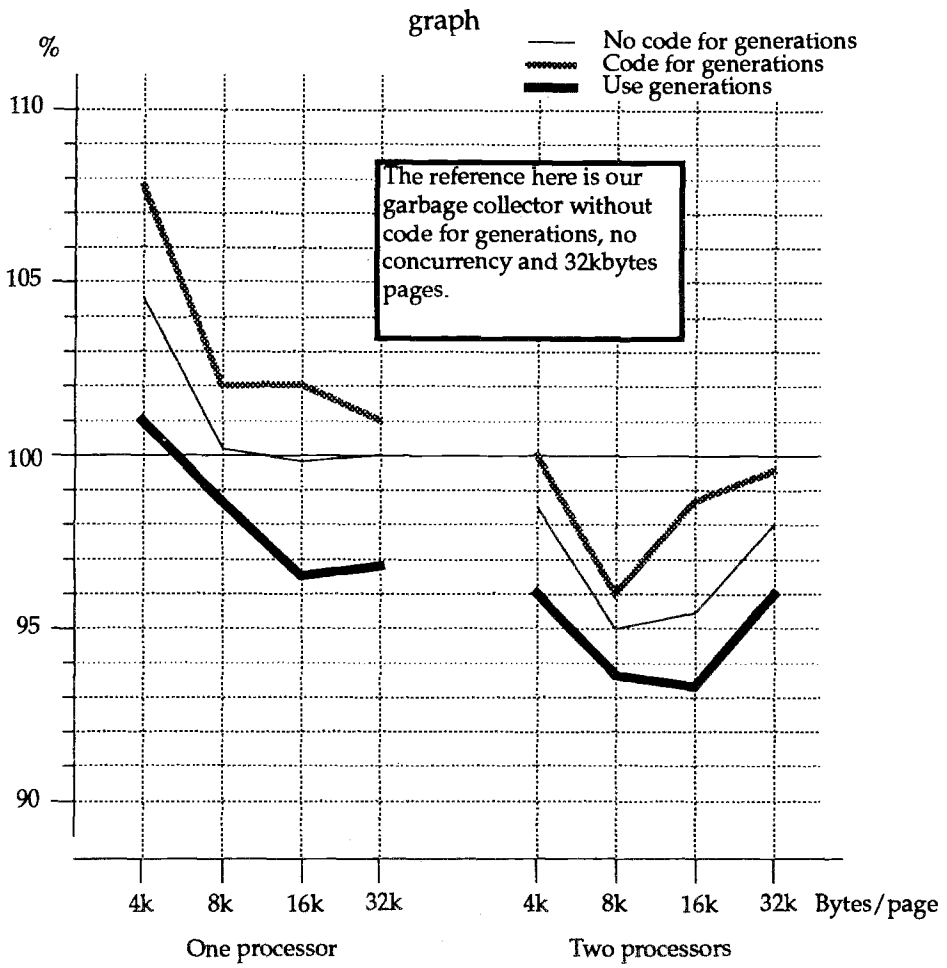


Figure 6 *This graph shows the execution time of graph using different choices for our garbage collector compared to our garbage collector with no code for generations and 32kb page size.*

gff solves a graph colouring problem. Gff needs 24 garbage collections when using a heap of 6 Mb. The generation behaviour in this program is not very good, as 40kB of old nodes contain 663 pointers to young nodes. There are also very few useful nodes after garbage collection (only around 60kb counting both old and young nodes Gff was not possible to run with the previous garbage collector, when measurements for Figure 7 was done, due to a bug in the current version of the LML-compiler for the <v,G>-machine.)

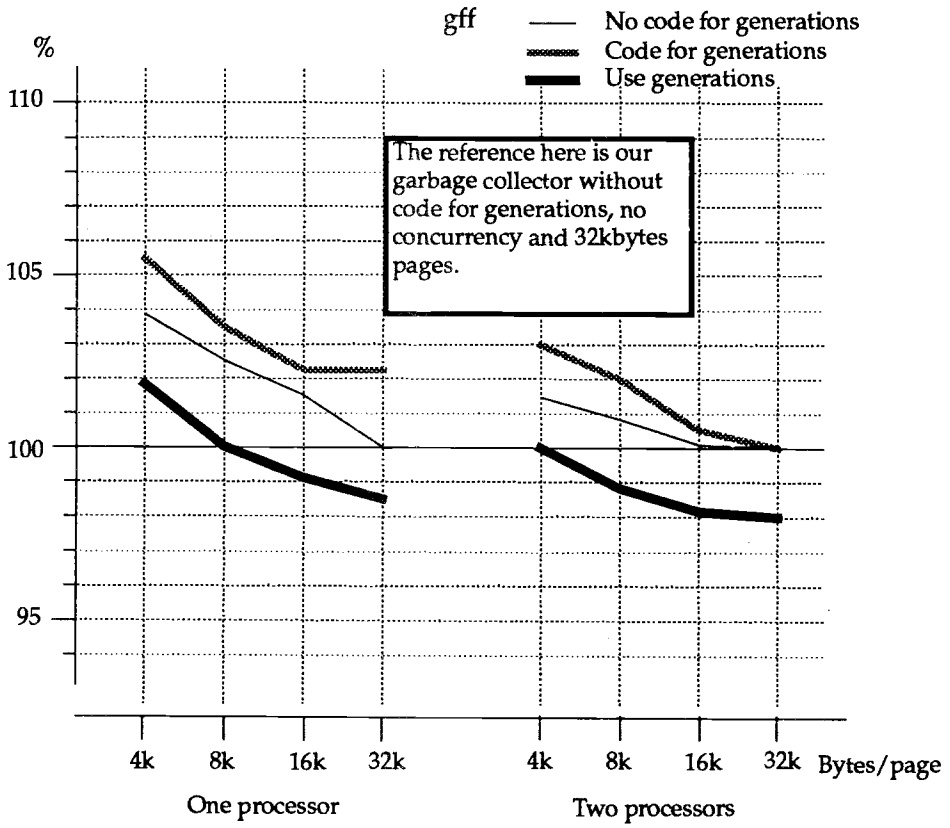


Figure 7 *This graph shows the execution time of gff using different choices for our garbage collector compared to our garbage collector with no code for generations and 32kb page size.*

kwic creates a list of keywords in context.It needs 18 garbage collections using a heap of 6 Mb. Kwic works on words, represented as list of characters, which are very good to tenure. The lazy semantics does however disturb the nice picture by leaving unevaluated frame nodes among the words which results in the highest observed number of pointers to young nodes, 3809 pointers in 260kb of old nodes. This number decreased with time as the unevaluated frame nodes became evaluated.

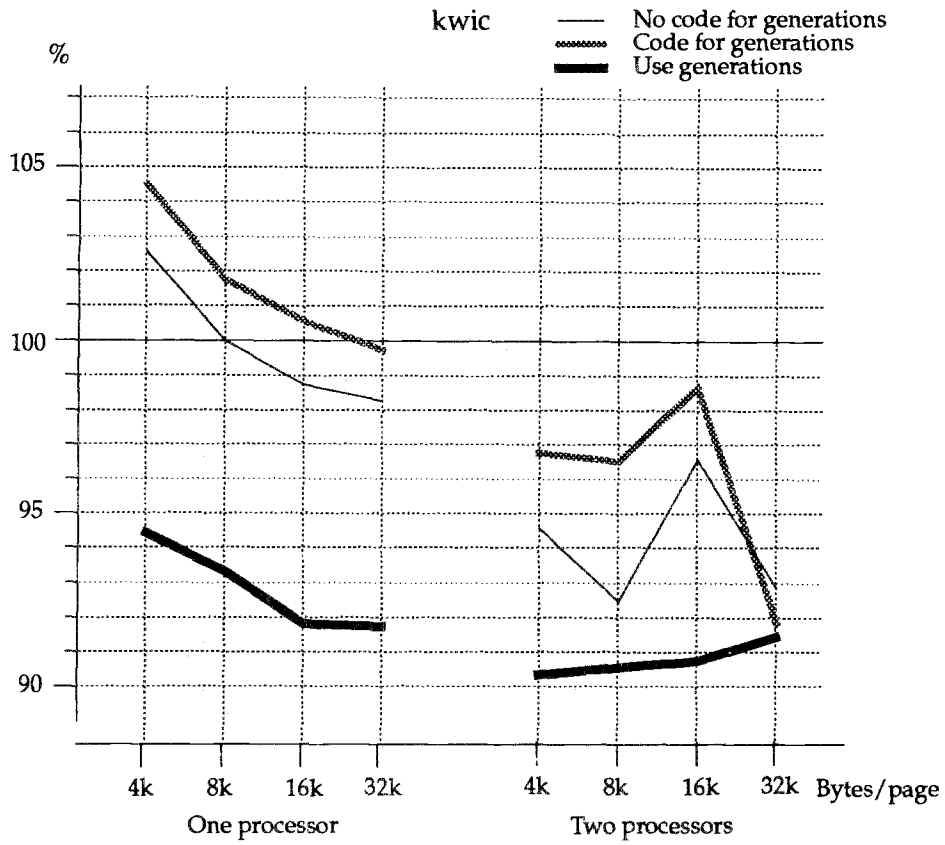


Figure 8 This graph shows the execution time of kwic using our garbage collector compared with the previous garbage collector.

Future work

Future work is to find a way to tenure frame-nodes, without paying the overhead to check all updates to them.One method to do this is to write protect old pages and when a mutator tries to write to the page the protection is changed to allow writes and the page is marked so that the garbage collector will scan the page during the next garbage collection. A problem is that frame-nodes might be updated with shorter nodes in which case it is easy to lose sync with the nodes when scavenging the page. This problem can be solved by inserting a fill-node if the new node is shorter than the frame-node it replaces, but this introduces an extra cost.

Another way to make it possible to tenure nodes that can be updated is to keep a pointer to all such nodes in the old pages. The garbage collector can remember all

nodes that can be updated which it moves to the old pages. This method is currently under investigation and looks promising so far.

Another change is to use the fact that if an old constructor node points at a young node (not a frame node in the current implementation) then this node can be prematurely tenured. The reason is that the young node can not be garbage unless the old constructor node is collected, so there is no reason for the garbage collector to work on the young node unless it also checks the old node.

References

[AEL88] Andrew W. Appel John R. Ellis Kai Li
Real-time Concurrent Collection on Stock Multiprocessors.
Proceedings of SIGPLAN 88

[App89] Andrew W. Appel
Simple Generational Garbage Collection and Fast Allocation
Software - Practice and experience, volume 19, February 171-183

[AJ89] Lennart Augustsson and Thomas Johnsson
Parallel Graph Reduction with the <v,G>-Machine.
Proceedings of the 1989 Conference on Functional Languages and Computer Architecture, 202-213, 1989

[Aug90] Lennart Augustsson
Garbage Collection in the <v,G>-machine or So much garbage, so little time.
PMG memo 73

[Hud82] Paul Hudak and Robert M. Keller
Garbage collection and task deletion in distributed applicative processing system.
ACM Symposium on LISP and Functional Programming, 167-178, 1982

[Hud83] Paul Hudak
Distributed graph marking.
Research report 268, Yale University, January 1983

[Lib83] Henry Liberman and Carl Hewit
A real-time garbage collector based on the lifetime of objects.
Communications of the ACM, 23(6):419-429, 1983

[Moo84] David A. Moon
Garbage collection in large LISP system.
ACM Symposium on LISP and Functional Programming, 235-246, 1984

[PJ87] Simon L. Peyton Jones
The Implementation of Functional Programming Languages
Prentice-Hall 1987

[Ung86] David Ungar
The Design and Evaluation of a High Performance Smalltalk System.
MIT Press 1986

Garbage Collection in Aurora: An overview

Patrick Weemeeuw and Bart Demoen

K. U. Leuven, Dept. of Computer Science, Celestijnenlaan 200 A, B-3001 Leuven.
E-mail: patrick@cs.kuleuven.ac.be, bimbart@cs.kuleuven.ac.be

Abstract. Aurora is an OR-parallel Prolog system whose implementation is based on the WAM, an efficient sequential implementation model. This paper discusses several issues related to parallel Garbage Collection (GC) in Aurora. The GC itself is a generalization of GC techniques used for sequential Prolog.
In order to make this paper self-contained, we focus on the general principles. More specifically, many optimizations are not discussed, insofar they are not directly related to the GC process.

1 Introduction

Prolog is a high level programming language with implicit parallelism as one of its most attractive features. Exploiting the OR-parallelism available in a Prolog program can speed up the computation considerably. However, the faster the Prolog system, the larger the problems are that users want to solve with it. Therefore, a garbage collector is still necessary in many cases. Moreover, due to the particular data structures used to represent the run time stacks in Aurora, holes may appear on the stacks, which tend to increase considerably the memory consumption of the system.

The Aurora system ([7]) is a prototype OR-parallel Prolog system with standard Prolog semantics, implemented on a shared memory multiprocessor. Its implementation is based on an abstract machine (the WAM [11]), which is an efficient sequential implementation model. The GC process we propose is a generalization of sequential GC techniques.

Garbage collection for such a system is complicated by several factors. Firstly, the data structures are more complex: each worker has, so to speak, its own view on the shared search tree (see further), and extra information about the state of the system is managed by a scheduler, which has to be kept consistent. Furthermore, since there are multiple processes working at the same time on the data structures, synchronization is necessary. This demands also for a parallel garbage collection algorithm. And finally, garbage collection is complicated by often conflicting efficiency considerations, for which it is not easy to find a suitable trade off.

This document is organized as follows. In section 2, we will illustrate how Prolog programs specify search trees, and how a particular sequential Prolog implementation model (the WAM) is used to explore this tree. In the next section, this is generalized to OR parallel execution, and some aspects related to memory management are discussed. Then a section follows about segmented GC in sequential Prolog.

Finally, in section 5, these principles of segmented GC are generalized for parallel GC in Aurora, and several related implementation aspects will be presented. Besides this, we will also discuss some higher level strategic considerations, such as when to perform GC, and for which parts of the search tree. Sections 2 and 4 can be skipped by readers already familiar with Prolog implementations and GC for such systems.

2 A short introduction to Prolog and its implementation

In this section, we show in an intuitive way how Prolog programs correspond to search trees, and how a Prolog engine is used to evaluate a program (i.e. to explore such a search tree).

Readers already familiar with Prolog and the WAM will note that many aspects that are not directly relevant, have been omitted (such as theoretical aspects, some language constructs (control operators, side effects), and some implementation optimizations (tail recursion, environment trimming)).

2.1 Prolog programs

A Prolog program consists of a set of predicate definitions, and a query. A predicate is a set of Horn clauses, each one specifying some alternative way to satisfy the relation the predicate stands for. E.g. the relation "ancestor" could be specified as: X is an ancestor of Y if X is a parent of Y, or if X is a parent of Z and Z is an ancestor of Y. This is expressed by the predicate for ancestor, that consists of two clauses:

ancestor(X,Y) ← parent(X,Y).
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y).

The variables occurring in a clause have a scope local to that clause. Clauses can be facts, e.g.:

parent(John, Bill).
parent(Mary, Bill).
parent(Pete, Mary).

which states that John is a parent of Bill, and so on.

A query has the form

← ancestor(Pete,X).

which states that Pete is an ancestor of some unknown X.

Generally speaking, executing a Prolog program means trying to match the query with the predicates in the program, in order to find out whether the expression stated in the query holds, and if so, for what values of the variables. Ignoring many theoretical and technical aspects, a Prolog program can then be seen as a tree, with the query at the top, and with branches for all the possible ways to satisfy the query. The nodes show the successive transformations of the original query (called goals) each time it is resolved with a matching predicate. Many branches are dead ended, but some of them may lead to a solution. We call this tree the search tree.

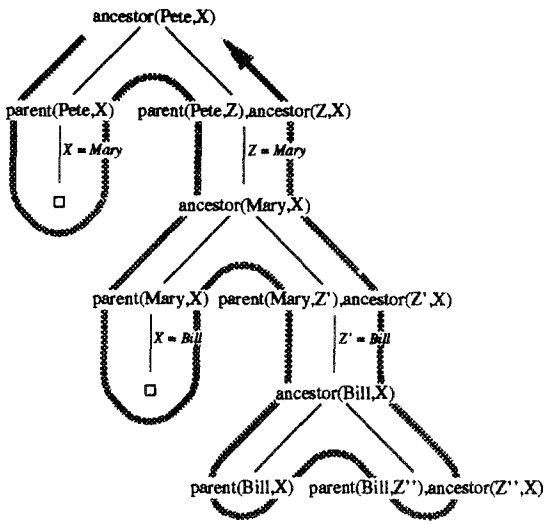


Fig. 1. Search tree—the gray arrow shows the evaluation order

In figure 1, we show the search tree for the above program. Variable bindings are shown beside the branches. A small square indicates a successful branch. The other tip nodes represent goals where the leftmost subgoal cannot be further resolved, which guarantees that the complete goal can not be solved.

Goals are matched with clauses using a process called unification. This means that all variables occurring in the clause are renamed such that they are unique, and then the head of the clause is matched with the goal literal: every two entities in corresponding positions are constrained to be equal to each other. One particular consequence of this is that variables can be assigned to only once on each path from the root to a tip node in the search tree.

2.2 Prolog engines

A Prolog engine is a process that, given a query and a set of predicates, explores the corresponding search tree and reports all successful branches, if any. This is done in a depth first, left to right manner (see figure 1). We will first explain how one branch is explored, and then how the machinery is extended to explore the whole search tree.

Exploring one branch

For each clause executed, an environment frame is created. An environment can be compared to a stack frame for executing a procedure in an imperative language. It holds the variables local to that clause, a reference to the parent environment and a reference to the next instruction to be executed in the parent environment (the so called continuation pointer). The environment is deallocated when the corresponding

clause has completed its execution. The clause "ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)." can then be interpreted in a procedural way as: to execute ancestor/2, first call the procedure parent(X,Z), and then call ancestor(Z,Y) recursively. X and Y are parameters of the procedure, through which variables in ancestor environments can be accessed. Z is a local variable.

There is also a heap to hold the global bindings, which have to survive the lifetime of the environment, and structures, which do not fit in the slot provided for a variable in the environment. The heap never shrinks while exploring one branch.

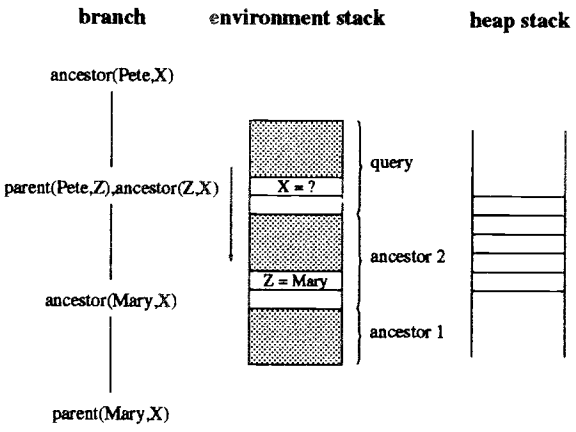


Fig. 2. Exploring one branch

Figure 2 shows the environment stack for the second branch at the moment that the query has been reduced to parent(Mary,X). We have so to speak three nested procedure calls (query, ancestor_2 and ancestor_1—ancestor_1 stands for the first clause of the predicate ancestor/2) and are about to call the clause parent(Mary,Bill). The header of each environment frame is shaded with gray. The variable X of the query is still unbound. The environment ancestor_1 has no local variables.

Backtracking

In order to explore the whole search tree, the engine is equipped with machinery to restore a previous state (i.e. to backtrack). Each time a tip node in the search tree is reached, the state can be restored to the one corresponding to the nearest fork point in the tree, from where execution continues by taking the next branch.

Therefore, a choice point exists for each node on the current branch with yet untried alternatives: it points to the next alternative clause to be taken, and saves the abstract machine registers and the top of stack pointers at the moment of its creation. A choicepoint is created when calling a predicate with more than one matching alternative; the choicepoint is removed when trying the last alternative. This way, the choicepointstack represents the yet unexplored part of the search tree.

Parts of the stacks created between two consecutive choicepoints are called segments. Choicepoints are allocated on a separate stack.

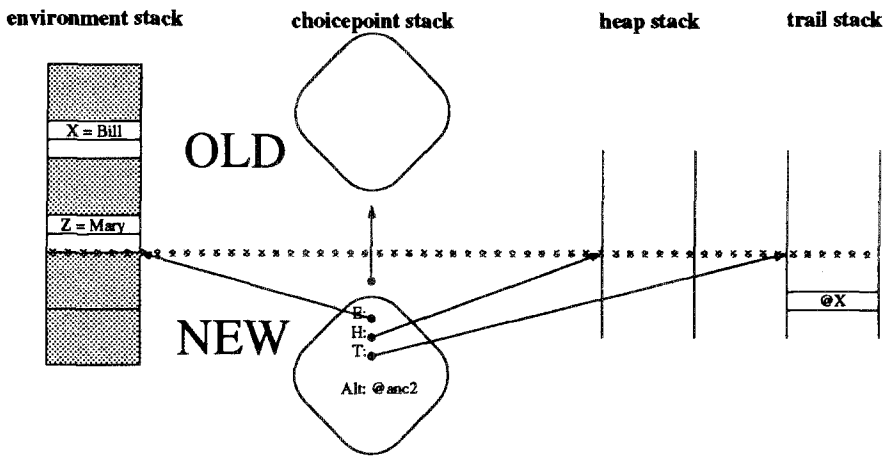


Fig. 3. Trailing a variable binding (@X is the address of variable X)

On backtracking, the parts of the stacks created since the last choicepoint are discarded (i.e. the top of stack pointers are reset to the value saved in the choicepoint—the implementation guarantees that no dangling references are possible). However, all changes made to the older part of the stacks since the creation of the choicepoint, have also to be undone. Therefore, every binding to a variable that is older than the most recent choicepoint—these bindings are called *conditional*—is recorded on the trail stack. On backtracking, this trail is rolled back to its previous size as well, at the same time unbinding all variables whose address is recorded in the trail segment.

This is illustrated in figure 3, where X is bound by the second clause for parent/2 (second branch in the search tree). The address of X has been pushed on the trail stack. The variable Z is unconditionally bound, as the choicepoint created for the parent/2 call has been deleted before trying parent_3, and therefore the variable is not recorded on the trail.

Note that a choice point also protects the environment stack. E.g. the environment for the clause "a ← b, c, d." can not be deallocated after d has finished its execution as long as there are any untried alternatives for b, c or d (omitting tail recursion optimization).

We call the ensemble of environment stack, choicepoint stack, heap and trail stack, which record the current state of the computation, the execution stack in short.

3 OR-parallel Prolog

OR-parallel execution of a Prolog program means that several alternatives are explored at the same time. In Aurora, multiple Prolog engines (called workers) explore in parallel each a part of the search tree (see figure 4). The upper part of the branches being explored is shared among the processors; the lower part is private to each worker. The advantage of this approach is that each worker can work efficiently on its own private branches, much the same way as in sequential Prolog.

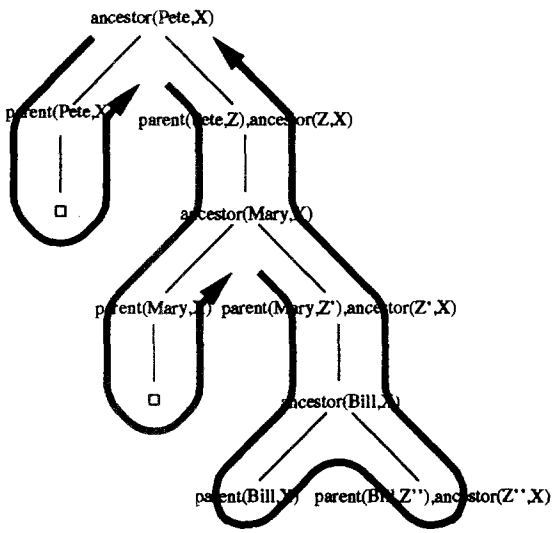


Fig. 4. OR-parallel exploration of the search tree

Stacks are used to represent the branch of the search tree currently being explored (environment and heap), as well as the still unexplored part of the search space (choicepoints and trail). The stacks are organized in a tree-like structure, representing the relationship between multiple active branches. We will refer to this tree structure as the execution tree, in analogy with the term execution stack. Each path from the top to a tip in the execution tree corresponds to the stack group (i.e. the ensemble of choicepoint, local, global and trail stacks) of one worker in sequential execution. The tips grow and shrink as execution proceeds, while the top part is shared by all the workers. The scheduler keeps track of the shape of the tree, by maintaining the parent-child relationship between public choicepoints. This is illustrated in figure 5. The upper part shows the search tree, and the position of workers W1 and W2. The lower part shows the execution stacks of W1 and W2. The segments shaded in gray correspond to the common part in the search tree, and are shared by both workers. The data structure for a choicepoint is extended to contain information about the layout of the tree, and is now called a node. Nodes may still exist when the enclosed choicepoint is logically dead (i.e. it represents no untaken alternatives any more), to maintain connectivity of the execution tree.

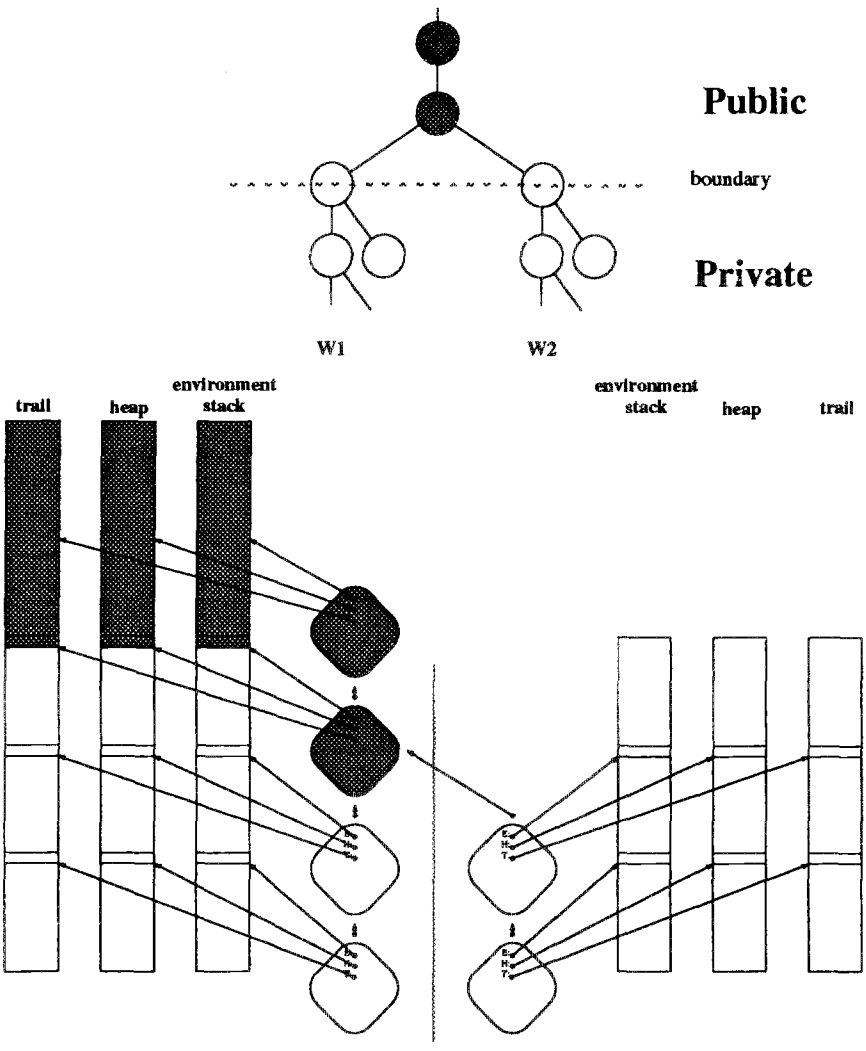


Fig. 5. Execution tree

The execution tree is divided in a public part (accessible to all workers) and a private part (accessible for only one worker). The public part consists of non fork and fork nodes (i.e. choicepoints with only one and more than one child choicepoint respectively), connected into a tree. The private part consists of linear chains of nodes, as in sequential Prolog.

3.1 The SRI model

As conditional bindings (i.e. bindings to variables that have to be undone on backtracking, or, alternatively, which can have a different value depending on which

branch at a lower level choicepoint is taken) can differ for distinct workers, one must allow workers to maintain their own versions of conditional bindings in the shared part of the tree. For Aurora, the SRI model [13, 12] is used to keep track of private versions of such variables. Each conditional variable is represented by an index to a Binding Array entry. In the Binding Array, the worker stores its (private) copy of the binding (see figure 6).

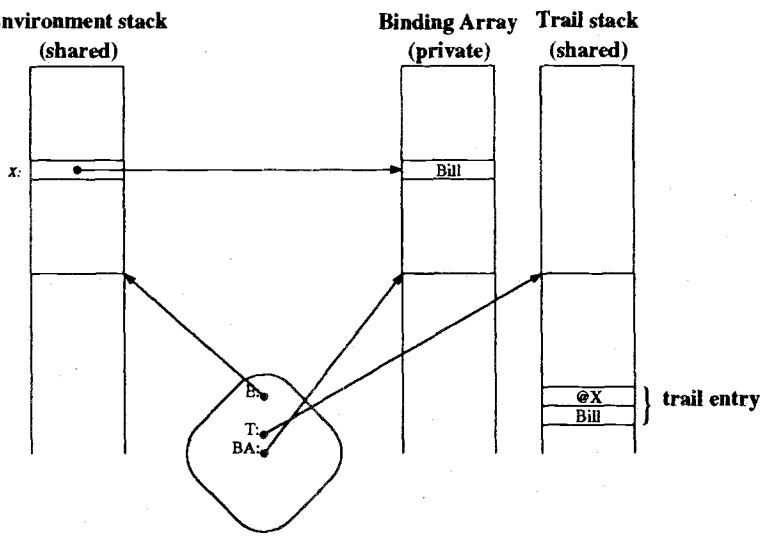


Fig. 6. A conditional binding

There are two Binding Arrays for each processor: a local and a global one, the former containing conditional bindings of the local (environment) stack, the latter containing the conditional bindings of the global stack (heap).

Maintaining the SRI model demands special tags, a more elaborate unification algorithm, an extended trail (see below) and some more information in choicepoints and environments (the indices of the Binding Array entries used up to that moment). The main advantage of the SRI model is the constant bounded overhead during sequential execution (typically 25%).

Because workers have to be able to move around in the search tree, the trail is extended: besides the addresses of the conditionally bound cells, it contains the contents too. The trail stack mirrors the contents of the Binding Arrays exactly (see figure 6): each worker sharing the trail segment in which the binding for X is recorded, will have the binding 'Bill' in its Binding Array. Thus a worker can install and de-install its binding arrays when it is moving around in the shared part of the search tree.

3.2 Scheduling

A worker can be in engine mode or in scheduler mode. In engine mode, it explores one branch private to itself, much like in sequential Prolog. In scheduler mode, the necessary cooperation between the processes is established: i.e. matching idle workers with available work, maintaining the public part of the search tree, and the synchronization for the execution of side effects.

Initially, there is only one task consisting of the whole search space, explored by one worker. The boundary between public and private nodes resides at the topmost node. As there are idle workers available, the busy worker is asked to make work available by lowering its boundary between public and private nodes. The idle workers can then take untried alternatives from the choicepoints made public. This way, tasks are split up and divided among the workers.

The Aurora System provides several schedulers, each with its own heuristics for maximizing parallelism and minimizing overhead[3, 4]. The interface between the worker and the scheduler part is specified in [10].

When a worker has to perform a side effect that might have an effect on other workers, it has to be leftmost in the search tree: otherwise the effect could differ from the standard Prolog semantics. If the side effect cannot yet be carried out, the current work is suspended by creating a dummy choicepoint at the tip branch with only one alternative (corresponding to the current continuation) to allow resumption at a later time during the execution. Suspension may also be triggered on behalf of the scheduler, in order to make the worker move to more interesting work.

3.3 Memory management

Each worker creates always new data structures on its own stacks. The worker that allocated the segment is called the *owning worker*. Physical stack space reclamation is always done by the owning worker.

Four kinds of segments can be discerned: local, ghost, remote and public segments.

Local segments are private segments on the top of the own stacks. For such segments, we have the same reclamation mechanism as in sequential Prolog. Each time a next alternative of a local node is taken, a "roll back" operation is performed to the situation immediately after the creation of the choicepoint, discarding the most recent segment. When the last alternative is taken, the choicepoint itself is no longer needed and can be deallocated.

Ghost segments are segments that logically do not exist any more, but can not yet be deallocated from the stacks because other segments are allocated on top of them. Their creation is illustrated in figure 7: W1 finishes its current task, and moves, by lack of work on its own branch to the next alternative of node o, which is q. W1 cannot deallocate the segments/nodes c, d and e from its stacks, since W2 shares them. It allocates the segment q on top of segment e. When W2 then finishes its task and moves to the right part of the tree, nodes c, d and e are no longer needed, but cannot be deallocated. They have become ghost nodes.

The chance that segments become ghost is related to the number of task switches. The number of task switches is related to the number of workers, the number of

suspensions and the shape of the search tree. In Aurora, large chunks of unused memory space can appear.

A ghost segment can only be reclaimed when a new task is started by the owning worker, and there are no non ghost segments allocated on top of it.

Remote segments are private segments allocated on the stack of another worker, or on the own stacks, with segments of another task allocated on top of them. Such situations arise when workers have to suspend their work, and to take work somewhere else in the tree. When a worker resumes the suspended branch, it must consider the segments as remote.

For remote segments, no immediate reclamation is possible, since the owning worker may have already created new segments on the stack group. The segment group is only marked as reclaimable, and becomes a ghost segment group.

For *public segments*, the scheduler decides when these are no longer accessible. This requires that the corresponding node has no alternatives left and that below the node no worker is or may become active (i.e. the node has no child nodes any more). The segments then become ghost segments.

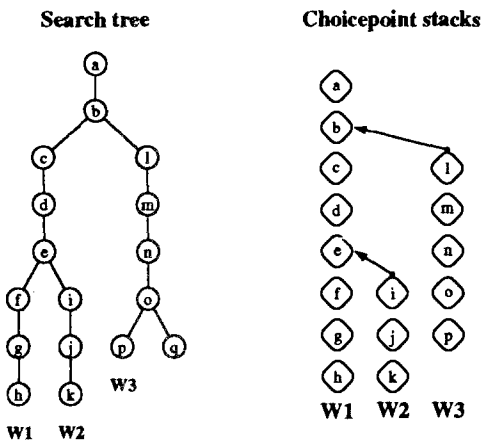


Fig. 7. The creation of ghost segments

On a logical level, each stack can be seen as a contiguous stretch of memory with a reference to the top of the stack. On a lower level however, each stack is implemented by a doubly linked list of memory blocks. Execution proceeds as if the stack consists of one contiguous piece of memory, but at certain points, a check occurs whether there is still enough memory available beyond the top of stack pointer to avoid overflow, and, if necessary, a new block is allocated and linked to the previous one. This imposes some less efficient memory usage, but offers much more flexibility.

4 GC for sequential Prolog

Garbage collection consists of three phases: a marking phase, a sweep and a compaction phase. We restrict the discussion to GC of the heap.

During the marking phase, all objects (potentially) still needed in the current branch or from any frozen state, are identified.

To identify all objects reachable from the current (active) branch, the argument registers and the variables in the environment chain starting from the current environment, are taken as a starting point for the marking. All the references found are followed, and the reachable cells on the heap are marked and taken into account to find further references. In [2], a clever algorithm is presented to mark in an efficient way all objects reachable from one cell.

To find all objects reachable from a frozen state, one has to consider that the argument registers are saved in the choicepoint itself, as well as a reference to the top of the environment chain as it existed at that point of execution. Marking proceeds in much the same way as for the active branch.

Everything reachable from the trail is also reachable from some choicepoint, and does not have to be taken as a starting point for the marking.

During the compaction phase, all reachable objects on the heap are shifted towards one end of the heap, making all unmarked cells that were scattered over the heap available again at the other end. At the same time, references are updated to reflect the new location of the objects.

The algorithm of Morris[8], which is based on a pointer reversal technique, is often used for the compaction phase. It preserves the order of the cells, which is necessary to be able to deallocate segments on backtracking.

Each cell has to be copied to its new location, and references to the cell have to be updated. Therefore all cells pointing towards a cell are temporarily linked in a data structure called a *relocation chain* . This is actually a pointer chain, starting at the cell originally pointed to (the head), and continuing over all the cells pointing originally to the head. The last cell contains the original contents of the head. This structure is disambiguated by an appropriate annotation.

In the algorithm of Morris, a cell can not be contained in a relocation chain and be the head of another one at the same time—a situation that may arise due to reference chains. Therefore, the compaction proceeds in two phases: an upward and a downward one.

During the upward phase, the heap is scanned from the top towards the bottom. For each marked cell, its relocation chain, if present, is updated first. The relocation chain contains at this moment all the upward references to this cell. We know the new location of the current cell—it can be derived from the total number of marked cells and the number of cells already treated during the upward phase—and each cell in the relocation chain is updated with a reference to the new location. The original contents of the current cell is restored at the same time.

Then the contents of the cell is inspected, and if this is a reference to another cell higher on the stack, the cell is included on its turn in the relocation chain of the cell it points to.

During the downward phase, the heap is scanned in the opposite direction. For each marked cell, the cells in its relocation chain are updated with the new position.

Then the contents of the cell is copied to the new location, while resetting the mark bit. And finally, the cell is included in the relocation chain if it contains a downward reference to another cell, so that the relocation chain of that cell will contain all downward references.

This way, all cells are moved, and all internal pointers are updated accordingly.

The sweep phase between the marking and the compaction phase is necessary to make sure that *all* references will be updated during the compaction. Therefore, all references from registers, trail and environments towards heap cells are included into the relocation chains before the compaction phase. Also the top of heap pointers saved in the choicepoints must be taken care of.

4.1 Segmented garbage collection

In segmented GC[9], one restricts GC to those segments that have not yet been collected before. Indeed, a choicepoint represents a frozen configuration, and everything reachable in such frozen segments remains reachable until the last alternative is taken from the corresponding choicepoint. Hence it is sufficient to collect each segment only once.

In segmented GC, a reference is kept to the youngest choicepoint already collected. This reference separates the new segments from the old, already collected ones. It moves up while backtracking over it and downwards when garbage collection is done.

Marking then stops on this border: references into the old part are not followed. The new trail segments contain all conditional bindings made while the new segments were constructed; therefor, every potential reference from the old into the new part can be found by scanning the new trail segments (see 3). Each such reference is included in the relocation chain of the cell it points to, such that it will be updated during the compaction phase. Compaction is then also done for the new parts of the heap only.

5 GC for Aurora

In this section, we propose a garbage collection scheme for the Aurora System. Let's first present the underlying principles.

1. Garbage collection is an expensive operation not directly contributing to the solution of the query. Therefore we try to postpone it as long as possible. Since memory reclamation is done much more efficiently by backtracking, we try to maximize this possibility by deferring GC.
2. We take advantage of sequential optimizations, which should fit in nicely since Aurora is based on an extension of the sequential model.
3. We want to avoid that workers have to wait because there is a garbage collection in progress. Either they should join the garbage collection process, or be allowed to proceed doing useful work, the latter being preferable.
4. We make all the data globally accessible, to allow for parallel execution. Therefore we de-install the Binding Arrays, and take the trail information into account

instead. After the garbage collection, the binding array is reinstalled. This involves some extra overhead when entering and leaving the garbage collection mode, but on the other hand, this is compensated since these cells have not to be treated during the marking and the compaction. As a consequence, there are no downward intersegment references.

We used a concurrent approach: while certain parts of the execution tree are under GC, some workers (but not all) may continue normal execution in the other parts. Execution can always continue on a branch that does not have any segment under GC. However, the scheduler must ensure that a worker, while it is looking for a new task, cannot move to a position in the execution tree that is under GC.

GC is always done on subtrees of the execution tree. These subtrees correspond to one or more tasks, which are allocated on the stacks interspersed with other tasks (and ghost segments, which represent ex-tasks). When we perform GC for a task, a gap appears on the stack near the end of the task. The memory occupied by this gap will only be effectively reclaimed (by unlinking the blocks from the doubly linked list) insofar as it consists of a number of complete memory blocks. The upper bound for the memory lost per task is then the size of a memory block: this is the maximum size of the gap after a task that is not directly reclaimable.

This inability to reclaim all memory immediately is a consequence of the Aurora memory model. If one wants to reclaim a gap at the end of a segment immediately, one has no choice but to relocate all tasks allocated on top of the segment, which is inefficient and induces a lot more synchronization constraints.

5.1 Generalization of segmented GC for execution trees

Each worker maintains a reference to the choicepoint on its branch that closes the segment group most recently compacted (the mark). The mark moves up on backtracking over the node, and down when performing GC, exactly in the same way as for sequential implementations, but now we have exactly one such reference on each path from the top to a tip segment in the execution tree.

When a worker decides to initiate GC, this should at least include the active branch up to the oldest uncollected segment, as this is the basis of segmented garbage collection. However, all branches sharing this topmost segment have to be updated with references to the new location of the cells in this segment. Therefore, we select the whole subtree, with the top node immediately below the marked node and belonging to the current active branch for garbage collection. Multiple subtrees can be compacted concurrently; each of these is uniquely defined by its topnode.

In sequential segmented garbage collection, a choicepoint is removed when its last alternative is selected, and the corresponding segment is extended and becomes subject to GC again. This is not the case in the public part: although the choicepoint no longer logically exists, there remains a dead node in the execution tree. We still treat this node for the GC as closing a segment group which is garbage collected. This may result in some unreclaimed garbage in such segments. However, this is negligible compared to the memory not immediately reclaimable between tasks.

5.2 Scheduler aspects

When a low memory condition arises, the scheduler decides which subtrees have to be selected. For the first GC invocation, there is no choice but to collect the whole execution tree. For the subsequent GC activations, the scheduler makes its selection based on the expected amount of garbage in the subtree, the speculativeness of the subtree and the position of the subtree. Note that when GC is postponed till memory is completely exhausted, there is no choice but to include the subtree where the worker is located that runs out of memory, as its execution cannot proceed until some memory block has been freed.

The scheduler roughly estimates the amount of garbage by counting the number of uncollected segments, and may give preference to big subtrees to offset the synchronization and initialization overhead. It could also try to postpone the GC of suspended speculative branches, as these may be cut away (by side effects in other branches), which reclaims memory at no cost. Finally, the scheduler may try in some situations to avoid GC in the leftmost branch of the execution tree, as every slow down in its execution may cause suspension of other branches (which must be leftmost in the execution tree in order perform some side effect), which would increase memory consumption. As all these aspects are closely related to the general scheduling policy, it is up to the scheduler implementor to decide which criteria are used.

Each worker active in one of the selected subtrees, suspends its work on request of the scheduler, and joins the GC process. As all data to be collected is globally accessible, every idle worker joins the GC as well, because it is better to speed up the GC process than to wait until new work becomes available in the tree.

The scheduler further isolates the subtree under GC for any other worker that continues normal execution, to prohibit that it accesses the temporarily inconsistent data when it looks for new work.

5.3 Sequences and series

Each subtree that has to be compacted is internally divided in a set of related sequences (see figure 8). A sequence consists of a number of segments that can be handled as a unit, both for synchronization and relocation. It corresponds to a number of nodes belonging to the same task and consecutively allocated on the same stack (with possibly some ghost nodes between them). All the nodes in a sequence are non fork nodes, except possibly the last one. The segments occurring in the same sequence can, after compaction, consecutively be allocated on the same stack.

The operations to be done on these sequences depend on each other; e.g. to compact the segments of a sequence in the upward direction, the upward compaction of its child sequences has to be completed first. Hence, analogously to nodes, sequences stand in a parent-child relationship to each other. An important property is that inter-sequence references only occur between sequences, one being an ancestor of the other.

On each sequence, a number of operations has to be applied in order. Each such "job" (consisting of an operation and a sequence) ready for execution, is matched with the next available worker in the pool of workers, for execution.

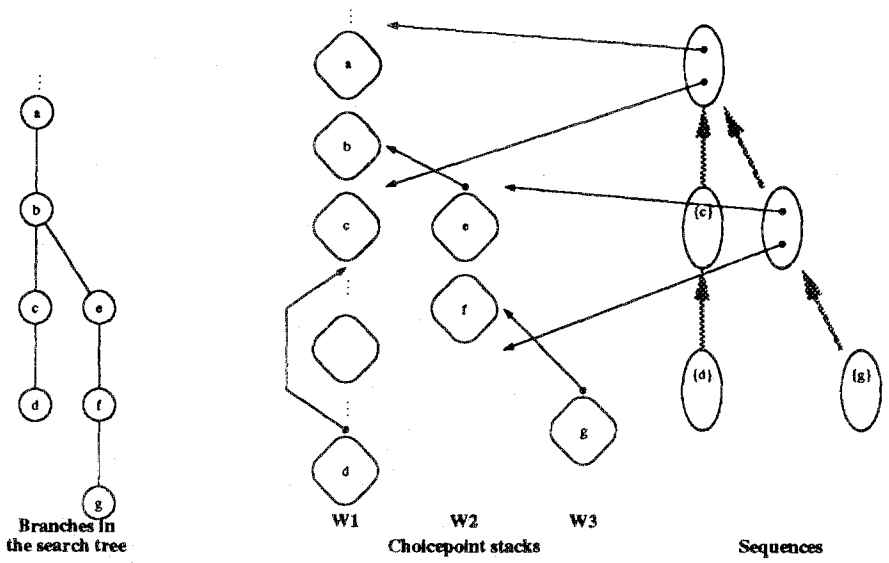


Fig. 8. Sequences

We call a set of consecutive sequences possibly separated by ghost segments, a *series*. The complete stack then consists of one or more series, interleaved with segments not involved in the GC, which have a fixed location. A series encompasses the sequences that depend on each other for the address calculations and the actual copying of the cells. In figure 8, the sequences {a,b} and {c} belong to the same series, but not {d}, as there are other segments with a fixed location in between.

5.4 The marking phase

Each subtree can be marked starting from the tip sequences, and gradually proceeding upwards. Sequences "at the same depth" can be marked in parallel. If we do not apply "early reset of variables" (an optimization described in [2]), we can even mark in parallel from different choicepoints on the same branch.

The main difference with sequential marking is that the mark bit has to be tested and set atomically. When two processors try to set the mark bit at the same time, only one of them will see it as previously not set, allowing us to maintain the correct count of marked cells per cell and per subtree.

Some provision must be made to maintain the correct count of marked cells for each individual worker. By aligning the memory blocks, every reference can easily be mapped to its memory block header, where the worker ID can be found.

5.5 The compaction phase

For the compaction, we use an adapted version of the algorithm of Morris, presented below.

The compaction phase proceeds as follows. There is an upward and a downward phase for each subtree. For each segment that is treated during the upward phase, the following conditions are satisfied:

1. the segments in the subtree below the segment have already been compacted in the upward direction. Therefore, the relocation chains of the cells in the segment are complete regarding the segments under it.
2. The worker that handles the segment is the only one having access to the segment. However, while inserting a cell in the relocation chain of a cell in an older segment (reachable by an upward pointer), provisions have to be made for non exclusive access to the higher segment. This can easily be done by an atomic exchange instruction, since all we need to do is (a) get a copy of the contents of the cell and (b) replace the contents with an annotated pointer to the cell in the segment that contained the reference.

This way, workers start compacting upwards with the tip segments, and proceed gradually towards the top of each subtree.

After the top segment of a subtree has been compacted in the upward direction, the downward phase starts. As there are no downward intersegment pointers, this phase can be started for each segment as soon as the destination of the cells is known to be free (i.e. it can be overwritten). The downward phase has only to take care of downward references within one segment, and of the actual copying of the cells.

The global Binding Array is compacted by a scan in downward direction over the global stack. This pass can be integrated with the downward compaction phase. Since the saved indices in the choicepoints have to be updated too, the scan is done segment by segment, updating the corresponding choicepoint after each segment. During the scan, an index is maintained to the next available global binding array cell. Each time a binding array reference is met, this reference is updated to refer to the next cell, while incrementing the index. All unreachable conditional variables are already eliminated from the global stack at this moment, and all bindings that became unconditional do not contain a binding array reference any more. When a worker later reinstalls its binding arrays, it will find binding array indices in the conditional variables, such that these refer to a compacted binding array segment.

5.6 Dependencies between the GC operations

We can now list the preconditions for each operation on a sequence:

- mark
  - The sequence has no child or all the children sequences have been marked already.
- sweep
  - The subtree has completely been marked.
- compact upwards

- The subtree has completely been swept;
- The sequence has no children or they are all compacted upwards;
- The destination of the youngest cell is known.

compact downwards

- The subtree has completely been compacted upwards;
- The sequence has no parent, or its parent has been compacted downwards;
- The destination of the youngest cell is free—i.e. the previous sequence in the series has completed this phase.

The destination of the youngest cell of a sequence can be computed as follows. We know the start address for each series, which is fixed. Hence the destination of the youngest cell of a tip sequence is always known, as the sum of the marked cells in the sequences is known for each worker and each subtree from the marking phase. The destination of the youngest cell of a non-child sequence can be derived with the extra information of the number of marked cells in its descendant sequences, obtained during their upward compaction.

Note that there exists always a partial order satisfying the requirements listed above: no deadlock will occur. Indeed, the only possible conflict is between two sequences for which the parent-child relationship requires an order of execution opposite to the order imposed by the predecessor-successor relationship holding between the two sequences. This would only be possible for the downwards compaction. However, the previously allocated segment can never be the child of the current segment.

6 Related work

In [1], K. A. M. Ali presented an incremental GC scheme for WAM based OR-parallel Prolog systems. His scheme is based on the observation that segmented GC can be done as in sequential Prolog in the private part of the tree. By requiring that each segment is already collected before it is made public, normal garbage is almost eliminated in the public part. Ali also presented some details about the collection of the global Binding Array and the trail.

The advantage of Ali's approach is its simplicity. There are however a number of disadvantages. First, it does not solve the recuperation of the ghost segments. It also introduces a significant delay for making nodes public, which is undesirable when there is a lack of work. Then GC is triggered when nodes are made public, which is unrelated to the memory usage. Consider for example a very broad, shallow subtree: garbage collection will often be triggered unnecessarily. And finally, cells initially not garbage in the public part may become garbage later, although this is unlikely.

In [5, 6], Dorochevsky presented and implemented a garbage collector for the ElipSys System. The situation for ElipSys is somewhat different from Aurora, since ElipSys also supports distributed architectures, where it is undesirable for a worker to write in the stack space of an other one.

The main advantage of his approach is that it is simple, retains all sequential optimizations and is efficient. However, it is not a general approach, and for Aurora it would have the same disadvantages as [1].

The paper gives some interesting figures as well about the amount of garbage in the Binding Arrays (typically 70–90%).

7 Conclusion

In this paper we presented the design decisions and most important implementation aspects of a garbage collector for Aurora. The garbage collector is a generalisation of well known techniques for sequential Prolog implementations. Hence, most of the optimizations for the sequential case can be applied, and the GC process itself is rather efficient, the major overhead being caused by the synchronization requirements. GC for private branches is neatly integrated in the general scheme, and causes no synchronization overhead.

The garbage collector requires only a minimal extension of the already existing Engine-Scheduler interface.

The major drawback of the implementation is that not all garbage can be reclaimed (a gap remains after each task), but this aspect is inherent to the Aurora design.

8 Acknowledgments

The interaction with the scheduler was greatly simplified by discussions with Mats Carlsson and Péter Szeredi.

This research is supported by ESPRIT Project 2471 (PEPMA) and the RFO-AI-02 project, sponsored by DPWB of the Belgian government.

References

1. K. A. M. Ali. Incremental garbage collection for or-parallel Prolog based on WAM. Gigalips Workshop, April 20–21, 1989.
2. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.
3. A. Beaumont, S. Muthu Raman, P. Szeredi, and D. H. D. Warren. Flexible scheduling of or-parallelism in Aurora: The Bristol scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991.
4. R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling or-parallelism: an Argonne perspective. In K. A. Bowen, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605. MIT Press, 1988.
5. M. Dorochevsky. Garbage collection in the or-parallel logic programming system ElipSys. ECRC, Technical Report DPS-85, 1991.
6. M. Dorochevsky, K. Schuerman, A. Véron, and J. Xu. Constraint handling, garbage collection and execution model issues in ElipSys. In A. Beaumont and G. Gupta, editors, *Parallel Execution of Logic Programs, Proceedings of the ICLP'91 Pre-Conference Workshop*, Lecture Notes in Computer Science 569, pages 17–28, June 1991.
7. E. Lusk, E. R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830, 1988.
8. F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, 1978.

9. E. Pittomvils, M. Bruynooghe, and Y. Willems. Towards a real time garbage collector for Prolog. In *Proceedings of the Second Symposium on Logic Programming*, 1985.
10. P. Szeredi and M. Carlsson. The engine-scheduler interface in the Aurora or-parallel Prolog system. Technical report, University of Bristol, Computer Science Department, April 1990. TR 90-09.
11. D. H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, Artificial Intelligence Center, August 1983.
12. D. H. D. Warren. Or-parallel execution models of Prolog. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 243–259, 1987.
13. D. H. D. Warren. The SRI model for or parallel execution of Prolog - abstract design and implementation. In *Proceedings of the Symposium on Logic Programming*, pages 92–102, 1987.

This article was processed using the LaTeX macro package with LLNCS style

Collections and Garbage Collection

Simon C. Merrall and Julian A. Padget

Bath University, School of Mathematical Sciences*, Bath BA2 7AY, United Kingdom
Email: {sm,jap}@maths.bath.ac.uk

Abstract. We present here a data parallel dialect of lisp, Plural EuLisp, which is a relatively low-level abstract model of massively parallel processing. It is not as rich as languages like Connection Machine Lisp and Paralation Lisp but encompasses ideas integral to at least Paralation Lisp. However its low-level nature makes the explanation of the underlying processor/memory management mechanisms easier as the low level structures are closer to the objects in Plural EuLisp. We describe how memory and processors are allocated and garbage collected, with particular interest in heterogeneous data parallel objects – which in general have been considered too expensive to be supported seriously.

Keywords: Data Parallelism, Garbage Collection, Heterogeneous Collections, Lisp, Processor/Memory Management, SIMD.

1 Introduction

In 1985 the Thinking Machines Connection Machine, a SIMD array processor with 4K of processing elements (PEs) scalable to 64K, represented a major step in massively parallel processing. Detailed descriptions of the operations and architecture of the hardware have been published[7], in contrast relatively little has been published on the implementation of the languages for the machine.

The three key functional languages are *Lisp, Connection Machine Lisp and Paralation Lisp, all of which are implementations of Common Lisp with data parallel extensions. The languages are documented in great detail[13] [9] and various discussions of their semantics have been presented [11] [10] but no real details of their implementations are available [1] [12].

At Bath we are interested in data parallel implementations of various symbolic applications and so we require a functional language for our own massively parallel computer, a MasPar MP-1011 (see Appendix A). We have been developing Paralation Lisp style extensions to EuLisp, our local, parallel dialect of lisp. A kernel of data parallel lisp primitives implemented in *mpl* (see Appendix B) are encapsulated in a EuLisp module to give a core language called Plural EuLisp. We consider it to be intermediate between *Lisp and Paralation Lisp as it exhibits aspects of both languages. The kernel has also been used for implementations of Connection Machine

* This work has been partially supported through the British Council ARC Programme, a Science and Engineering Research Council (SERC) Studentship, SERC grant GR/G31048, International Computers Limited (SERC CASE award)

Lisp and Paralation Lisp. We are currently experimenting with these languages, trying to identify useful modifications and extensions, or perhaps the functional requirements for a new language.

Here we give a description of Plural EuLisp and briefly compare it to Paralation Lisp. We then describe the memory/processor management system we have devised, discussing its advantages and possible improvements that could be made in its operation and efficiency. We describe the garbage collector and finish by discussing the handling of front-end back-end references and how the power of the MasPar can be applied to their resolution when garbage collecting.

2 Plural EuLisp

EuLisp is a parallel dialect of Lisp developed at Bath and in conjunction with academic and industrial researchers around Europe. The distinguishing features of the language are modules for separate compilation, threads for multi-tasking and a fully integrated object system based on classes and generic functions. A more detailed description can be found in [8].

We have extended EuLisp to allow use of the MasPar by adding a new module called `plural`. This is a set of primitives which operate on a lisp object on each member of a specified processor set. The primitives are too low-level to write programs with and so they are encapsulated in another module called **eubang**, this provides a new class called `plural` and a set of operators on that class.

An object of class `plural` has the appearance of a vector, but each element is allocated on a separate processor. Two plurals are said to be *conformant* if they are allocated on the same set of processors. Plurals in the same conformant set can be operated on in parallel, for example we can cons each pair of elements together to create a plural of cons-pairs. This gives us two key ideas:

- Allocating processor sets.
- Allocating objects on these processor sets.

The function `make-plural` encapsulates both of these ideas. If given an integral argument, `n`, it allocates a plural with `n` elements on the first suitable set of PEs it identifies. If given a plural as an argument it allocates a plural conformant to the argument, that is, on the same set of processors. In both cases each element of the new plural contains ()—the empty list. The contents of a plural can be referenced using the function `plural-ref` the behaviour of which is similar to that of `vector-ref` and has a corresponding updator function.

```
(setq p (make-plural 5))

=> #P(() () () () ())

((setter plural-ref) p 1 '(1 a))

=> #P(() (1 a) () () ())
```

Conversion methods for lists and vectors are supplied for convenience, the functions `list-to-plural` and `vector-to-plural` work as would be expected, creating a

new plural in which the values of the elements are taken from the elements of the argument list/vector. Both functions can take an optional plural argument to which the result will be conformant—truncating or padding as necessary.

The primitive operations are either supplied as additional functions, distinguished from their serial counterparts by the **-s** suffix (indicating plural!) or as additional methods to existing generic functions.

```
(setq q (list-to-plural '(55 44 33 22 11) p))

=> #P(55 44 33 22 11)

(cons-s p q)

=> #P((55) (44 1 a) (33) (22) (11))

(+ q q)

=> #P(110 88 66 44 22)
```

In order to be able to write parallel code we require two more special functions. The first of these is **bang** which takes two arguments: the first is a singular value and the second is a plural. The result is a new plural conformant to the second argument each element of which has been initialised with the first argument. The second is the parallel conditional form called **if-s**. The arguments are three expressions which must deliver (conformant) plural values. The first expression is evaluated to give a plural of values which are interpreted as booleans in the Lisp sense, that is either **()**, for false, or non-**()**, for true. These values are then used to modify the active set, evaluating the consequent on those processing elements for which the condition plural is true and the alternative expression on the remainder. The result of evaluating each arm of an **if-s** is a plural and these are then merged to form the result of the conditional expression. If none of the processors are active for a form then it will not be executed; this is important when defining recursive functions.

```
(defun sum-list-s (list-s)
  (if-s (eq-s list-s (bang () list-s))
        (bang 0 list-s)
    (+ (car-s list-s) (sum-list-s (cdr-s list-s)))))
```

The example above shows how it is, in principle, straightforward to define parallel versions of functions, and this represents the control aspect of the massively parallel abstract machine model. The other aspect is communications and the mechanism supplied is modeled closely on that in Paralation Lisp.

Inter-processor communication is abstracted by the class, **mappings**, which are created by the function **match**. A mapping describes how to create a plural in one conformant set from the elements of a plural in another conformant set, the sets can but do not have to be different. The programmer specifies which elements of the source contribute to each element of the destination by **matching** a plural from each, **eq** is used to decide which elements correspond to each other. Having created a map, a plural conformant to the source can be **moved** down it to create a plural in the destination set. **Move** uses a specified combinator, some appropriate binary function, to resolve collisions where more than one source element maps to a destination element. Finally to resolve the case where a destination element receives no value from the source plural **move** is passed a default value.

```
(setq from (list-to-plural '(nowhere first first second nowhere)))

=> #P(nowhere first first second nowhere)

(setq map (match (list-to-plural '(first second third)) from))

=> #<mapping>

(move (list-to-plural '(a b c d e) from) map cons-s 'empty)

=> #P((b . c) d empty)
```

This then introduces Plural EuLisp, a more formal specification is given in Appendix C. We consider it to be intermediate between *Lisp and Paralation Lisp as its parallel operators are explicitly named and are all "flat". That is, there is no concept of recursive parallelism as it is not possible to create a plural containing plurals. This restriction is not present in Paralation Lisp (or Connection Machine Lisp).

2.1 A Comparison to Paralation Lisp

A paralation is a set of (virtual) processors, a field belonging to a paralation has a value for each element of that paralation. The paralation constructor, **make-paralation** creates a special field in the new paralation, the *index* field, which enumerates the elements from $0, n-1$ and returns this as its result. Thus we can define **make-paralation** in Plural EuLisp as follows.

```
(defun make-paralation (size)
  (labels ((list-n-to (n)
             (if (= n size) ()
               (cons n (list-n-to (+ size 1))))))
    (list-to-plural (list-n-to 0))))
```

Fields belonging to the same paralation can be operated on in parallel. This is similar to the idea of conformant sets of plurals. However the parallel operations are specified by using **elwise** which applies a given lisp form to each element of the parameter fields. Thus the plural primitive **cons-s** could be defined using **elwise**:

```
(defun cons-s (a b) (elwise (a b) (cons a b)))
```

As already mentioned, the style of communication in Plural EuLisp is taken directly from Paralation Lisp. The only difference is that whereas in Plural EuLisp we must give a parallel combining function, for example **cons-s**, in Paralation Lisp we need only give the singular form and the parallel form is derived from it.

```
(setq from (make-paralation 5))

=> #F(0 1 2 3 4)

(setq map (match (make-paralation 3)
                 (elwise (from) (list-ref '(() 0 0 1 ()) from))))
```

```
=> #<mapping>

(setq data (elwise (from) (list-ref '(a b c d e) from)))

=> #F(a b c d e)

(move data map cons 'none)

=> #F(none (b . c) d none)
```

We do not intend to give a full description of the paralation model, merely to show that Plural EuLisp represents a kernel that can be used to build higher level abstractions like Paralation Lisp. It could be argued that since the problem of implementing such languages should not be too difficult, indeed details of such implementations have been published [2], that Plural EuLisp has no real value in itself. This is true if considered as a new language, but here we are interested in memory and processor management and so feel that the Plural EuLisp abstract machine model, being lower-level than the Paralation model simplifies explanations of the underlying mechanisms.

3 Storage Allocation

A novel feature of the allocation scheme that has been implemented for Plural EuLisp is that different sized data structures can be allocated in parallel on different processing elements. It is necessary to be able to do this if we are to support heterogeneous collections. Some sources suggest that this facility is not required and that, for example, a field of structures can be best represented by a structure of fields. To our mind heterogeneous collections and particularly collections of structures should be supported because:

- They are in keeping with the spirit of Lisp.
- Operations on structures are, in general, slot references, and a generic slot reference on a SIMD architecture can be implemented using local indirect addressing (LIA), which is a particular property of the MasPar.

3.1 Parallel Lisp Objects

Each processing element contains a small garbage-collected heap. We have adopted a 16-bit addressing system in which an address is an index into an array of 16-bit words which serves as the heap. With this system we can address upto 128K of memory, which is twice the size of MasPar's latest memory option and is rather more space efficient than 32-bit addresses. When systems which have more local memory than can be addressed like this become available we can change to a 32-bit address system as space conservation will no longer be such a pressing issue.

A Parallel Lisp Object (PLO) is a set of lisp objects where each object is allocated from the heap of a different PE. A PLO need not be homogeneous and as a result of this the objects will, in general, be at a different locations on each PE. This also

means that the heaps will become exhausted at different rates. Consequently, to specify a PLO we need an address for each PE—a 16-bit plural (see Appendix B) variable fulfills this requirement. The contents of the location indexed on each PE contains the object's heap header. The sixteen bits are split up as follows:

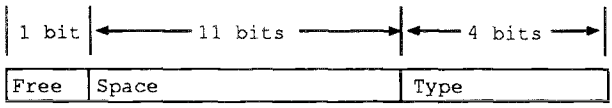


Fig. 1. Organisation of heap header

Attempting to cram the heap header into 16-bits causes some limitations within the system. No more than 16 types and a maximum object size of 1024 slots (2048 bytes). However if we assume each element of a vector of size 1024 contains an integer, which takes 2 bytes of header and 4 bytes of data (possibly more depending on alignment) the total is $\approx$ 8K. This is a sizable proportion of all the memory available on a 16K machine.

The data component of the object is allocated in the next `Space` locations. The heap allocation function `mp_alloc` uses a table, `type_info_table`, which specifies the size and alignment requirements of each type so that different objects can be allocated on different PEs in a single parallel operation.

3.2 Contexts

Within Paralation Lisp the number of physical processors is hidden from the programmer by the paralation construct. This can be thought of as a handle on a set of virtual processors.

Plurals provide a similar mechanism with the *Conformant Sets* having the same role as paralations. The function `make-plural` can be used to request a new conformant set of any size ($n > 0$). We can classify these requests by n as follows:

1. ($n =$ `nproc`[1]), this is a trivial case.
2. ($n <$ `nproc`), this is the most interesting case since we want unallocated processors to be available for other paralations.
3. ($n >$ `nproc`), we can consider this as a combination of cases 1 and 2.

We use the concept of a *context* to identify a processor set and allow us to handle case 2. Each element of the processor array executes the globally broadcast instruction stream conditionally on its activity bit. So by making only those processors within a context set active we can make sets of PEs independent. By considering the array as a sequential string of processors we can identify a context by a start processor number and a length. Figure 2 shows how a structure allocated on the Array Control Unit (ACU, see Appendix B) specifies a set of processors on the Data Parallel Unit (DPU).

[1] The value of the mpl global variable nproc is the size of the processor array.

We hope to modify the system to treat the array as a rectangle rather than a string, so that we allocate rectangular blocks. This will improve locality within a context and make use of the nearest neighbour communication net meaningful. This will be useful when implementing *shaped* paralations (and plurals) *c.f.* Sabot [10], chapter 9.

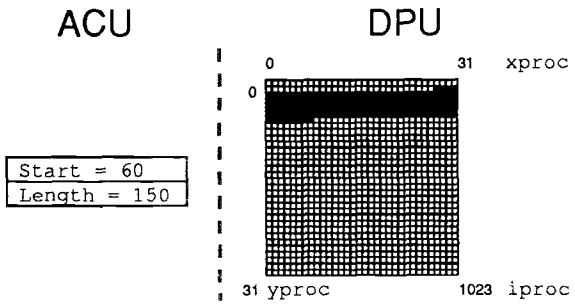


Fig. 2. ACU structure specifying a context's processor set.

To execute functions within a context we deactivate all those processors not within the context before calling the function. So for the context `C` we simply activate each processor whose processor id (denoted by `iproc`[2]) satisfies the expression:

$$(\texttt{C.start} < \texttt{iproc} < (\texttt{C.start} + \texttt{C.length}))$$

With contexts we are able to operate on portions of the array and we have moved away from *Lisp style global array operations.

3.3 Plurals

We can now see the basic form a plural will take. A Parallel Lisp Object combined with a context, with conformant plurals sharing the same context. Before tying these ideas up in a single concept we need to consider:

1. Methods of allocating contexts.
2. Giving the front end a handle on a PLO.

The Plural Space

Both of these problems are reconciled in a memory/processor management scheme which uses a portion of the heap space called the plural space. A single offset specifies a location on each PE which contains the address of an object on that PE, i.e. a PLO.

Figure 3 shows how the plural space allows us to represent a Parallel Lisp Object by a single value on the front end. Everything in the plural space is an address,

[2] **iproc** is an mpl global plural containing each PE's number.

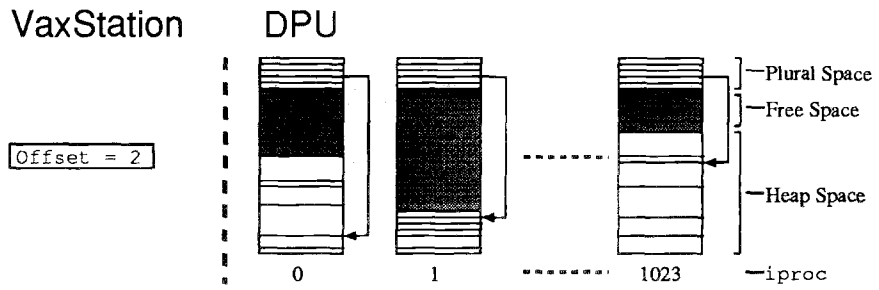


Fig. 3. Front End Integer specifying a Parallel Lisp Object.

this makes management easier and there is no need for headers. The plural space is located at the high end of memory and grows towards the low end, that is towards the heap space. With simple Parallel Lisp Objects this is a very easy mechanism to manage. Contexts however complicate the issue since they make it possible for the plural space to be used at different rates on different PEs. As contexts can overlap this means we cannot simply allocate the next free plural space location on each PE since they would not necessarily be the same. The basic mechanism described here is similar to that used for *xecs* in Connection Machine Lisp [12]. However here fragmentation can occur where as in CM Lisp it could not.

Contexts make it necessary to allocate plural space on only a portion of the array. When we do this we want the plural space on the remaining PEs to be available for later allocation. To allocate plural space for a PLO within a context of n elements we must identify a contiguous set of n processors and an offset i such that the location i in the plural space is free for each PE in the set. This will give us a new context each time, if we also specify which PE the processor set must start at we will be able to allocate plural space for an existing context.

The plural space is the same size on each PE. When the plural space is extended (initially the plural space has zero length) each new location is initialised with management data. The high bit is set as a free flag, since we have a maximum of somewhere around 7000 locations this bit will be clear if the location contains an address (i.e. is not free). If the location is free the value in the low 15 bits is the number of contiguous processors above (i.e. with higher `iproc`) for which the offset is free.

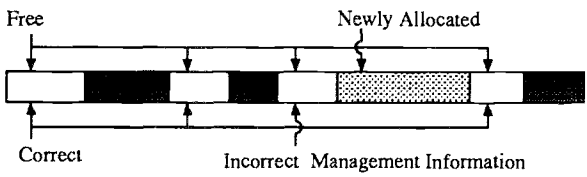
The algorithm below identifies an *offset* and a contiguous processor set of *size* elements starting at *start_PE* satisfying the conditions described above.

```
start_PE = -1
for each offset in plural_space
    if free?(plural_space[offset])
      potential_PEs = space(plural_space[offset]) > size
    fi
    if (potential_PEs) start_PE = reduceMin(iproc); fi
    if (start_PE ≠ -1) break; fi
rof
return (start_PE, offset)
```

If this fails to identify a processor set, the plural space is extended and the set allocated from the new space. In the worst case, this algorithm runs in time proportional to the size of the plural space; but this can be improved by starting from the last offset allocated for a context, after garbage collection these can be set to the top of the plural space again. If all the processors are active at the start of the algorithm then *any* processor set will be identified. By modifying the active set so that only one PE is active the algorithm will automatically identify a processor set starting at that PE.

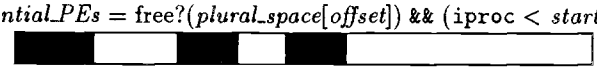
Having identified a set of processors and an offset into the plural space the contents of the locations indexed by *offset* need to be modified in order to show they are no longer free.

Below is a possible allocation pattern for the offset in question across the entire array. We notice that the plural space management data in the free block immediately preceding the newly allocated block is now incorrect.



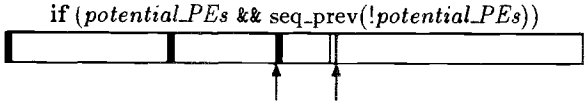
This only happens when a block is allocated from the middle of a free block. To identify this block, i.e. which PEs it starts and finishes on, we first identify all those processors which precede the newly allocated block and for which this location in the plural space is free. Below we indicate the processors in this set:

potential_PEs = free?(*plural_space*[*offset*]) && (iproc < *start_PE*)



We then activate all those *potential_PEs* whose immediate predecessor is not a *potential_PE*. This identifies the first PE of every free block preceding the newly allocated block. **seq_prev** is one of a pair of macros which allow the processor array to be treated as a sequential string of processors.

if (*potential_PEs* && seq_prev(!*potential_PEs*))



from_PE = reduceMax(iproc) *to_PE* = *start_PE* - 1

reduceMax(iproc) will return an integer identifying the highest of the active PEs. Thus we have identified the PE set to be renumbered (*from_PE* - *to_PE*). The code to initialise a strip of the plural space can be used for the renumber operation by applying it to only a segment of the processor array. Similar methods are employed when freeing plural space, this is more complicated though since we need to merge contiguous free space.

Plural Organisation

We are now able to allocate contexts and PLOs using a single simple interface. Having identified a contexts processor set we allocate an ACU structure which specifies that set. So we can see that a plural is specified by a 32-bit context address and a 16-bit plural space offset.

Conformant plurals share the same context, they also share the same context stack, which is used by **if-s**. Since both the context and the context stack are concerned with modifying the active set it seems reasonable to combine the two concepts.

The context stack for each element of a context is represented by a list of () and non-() values. To associate the context stacks with the context we create a plural to contain them and store the offset of this plural in the context structure:

```
typedef struct MP_Context_ {

natural start;
natural length;
natural offset;

} MP_Context;
```

Now when executing functions within a given context, we first activate only those processors which are in the context's processor set. We then take the top of the context stack for each member of the processor set and modify the active set further depending on this value. A front end handle now consists of an offset, context address pair which also gives us the internal context of the conformant set to which the plural belongs. Two plurals are conformant if the address of their context handles are equal, this makes it easy for the front end to check a set of arguments are conformant before applying a parallel primitive.

4 Garbage Collection

We have decided not to support inter-processor references, i.e. all the structures referenced on a PE are allocated on the same PE. As there will be a rich selection of communication primitives which copy objects between processors we do not feel this to be too limiting a restriction. Hopefully a communication link between PEs will serve as well as an inter-processor reference for most tasks. Because the heaps are independent the processor array can be garbage collected in data parallel, the fact that the heaps are different on each PE being the only real obstruction to an efficient solution.

As the plural and heap spaces share the same address space, memory is exhausted when the two regions are about to clash. A compacting garbage collector seems the most appropriate choice as the plural space must always have room to grow into, this also simplifies the allocation process (see Section 3). A possible GC technique is Stop and Copy in which the active structures are copied to an alternative memory space, where they form a contiguous region. However this requires sacrificing half our available memory for the "alternative" space. For this reason a Mark and Sweep

collector, where the active memory is marked and then compacted, seems the better choice.

The active memory on a PE is identified by propagating a mark through the heap space starting from each heap address in the plural space. These addresses in the plural space correspond to references from the enclosing environment on the host. When a front end handle on a collection is collected the memory used in the plural space is freed. We do not go into the details here but point out that this process is separate from the PE heap garbage collection. The pseudo-code below describes the key points of the mark-phase.

```
current_offset = first_plural_offset
while (current_offset < last_plural_offset)
   mark(plural_space[current_offset++])
   current_offset
elihw
```

```
mark(heap_objects)
   free(heap_objects) = false
   if (!atomic?(heap_objects))
      index = 0
      while (index < length(heap_objects))
         mark(slot_ref(heap_objects,index++))
      elihw
   fi
```

Having identified the active memory, compaction is a highly data parallel process and should run in time proportional to the largest number of objects (both active and inactive) in any one heap. The compaction pseudo-code given here could possibly be improved by attempting to skip over contiguous blocks of free objects rather than simply examining the next object on the PE each time. How well this would improve the system behaviour is difficult to predict and would be highly dependent on the allocation patterns.

```
current_heap_objects = bottom_of_heap
new_heap_space = bottom_of_heap
while (current_heap_objects < heap_space)
      if (!free?(current_heap_objects))
         copy(new_heap_space,current_heap_objects)
         new_heap_space += size(current_heap_objects)
      fi
current_heap_objects += size(current_heap_objects)
elihw
```

There are a few more updates we must make so that the heap is consistent, this involves building a map of the heap so that we can resolve pointers to moved objects in both the heap and the plural space, this is a costly process but a data parallel version should not affect its behaviour adversely! [5] [6].

As mentioned above the compaction phase is a highly data parallel operation, however this is not true of the mark phase. The recursive mark process will behave optimally only when identical structures are being marked on each PE. We give an example to show how the behaviour is less satisfactory when the structures being marked differ between PEs. Consider two PEs with a cons cell on each, both containing vectors of a 100 and 10 elements, but in different orders. Initially the cons cell will be marked in parallel and a recursive call will be made to mark a vector. After 10 iterations one of the PEs will have finished and will then have to remain

inactive for the remaining 90 iterations it takes for the second PE to finish. After returning the process will be repeated for the second vector on each PE. If we ignore the contents of the vectors we see that the process of marking a total of 110 vector slots on each PE takes 200 mark vector slot iterations on both PEs.

In a SIMD processor array the globally broadcast instruction stream can only be executed by the active PEs, if we can increase the number of participating processors we should improve the system's performance. The processors become inactive because of the recursive calls which, by extending the parallel stacks, deactivate the processors not making that call. Further because the PEs must all return at the same time some may have to wait until this condition is satisfied. If we can implement an iterative solution these situations should be eliminated and we will have achieved the desired affect.

We have here an example of a general problem in SIMD programming, the broadcasted instructions are applicable to all elements but PEs become inactive while waiting for the worst case PE to complete. In some cases it is possible to make use of the local indirect addressing[3] available on the MasPar to improve the situation. Here we transform the code into an iterative version with explicit stack management, this means the operation will be applied to the top of the stack rather than the highest stack level as happens in recursion. This will require using more memory for the stacks but will also use less C stack. In this special case we can use pointer reversal to traverse the environment and eliminate the need for a stack. Below is pseudo-code outlining versions using pointer reversal and explicit stacks respectively. Both are iterative and at the beginning of each iteration only the PEs which are still marking will be active.

```
mark( heap_objects )
  prev_objects = last_objects
  while (!last_objects?(heap_objects)
    if (atomic?(heap_objects) ||
          visited?(heap_objects))
      clear_flags(heap_objects)
      free(heap_objects) = false
      unshuffle(prev_objects, heap_objects)
    else
      shuffle(prev_objects, heap_objects)
    fi
  elihw
```

Pointer Reversal Version

```
mark( heap_objects )
  push(to_mark_stack, heap_objects)
  while (!empty?(to_mark_stack))
    heap_objects = pop(to_mark_stack)
    free(heap_objects) = false
    if (!atomic?(things_to_mark))
      copy_push(to_mark_stack,
                first(heap_objects),
                length(heap_objects))
    fi
  elihw
```

Explicit Stack Version

This shows how if a heterogeneous set of heaps are independent, they can be effectively garbage collected in data parallel. Well defined GC techniques are directly applicable to the compaction of the data parallel heaps and we expect the compaction phase to take time proportional to the maximum number of objects on any PE. Implementing data parallel versions of the algorithms would be much harder without the local indirect addressing available on the MasPar. The mark phase is less straight forward but again we can take advantage of LIA to mark the heap in time proportional to the maximum number of active objects on a PE. A direct data

[3] This is where the broadcast instruction is applied to a different location on each PE.

parallel conversion of a sequential marking algorithm would take time proportional to the sum of the size of each structure differing between PEs. This is because where the structures on PEs diverge the mark time is the sum of their seperate mark times. This also gives us a possible method for dealing with generic operations on sets with different size and content in general [14]. In the next section we consider a special case of inter-processor references.

5 Collecting in a Distributed Environment

Moving data between processors, be it from the host to the processor array or between processing elements, is a copying operation. These copies are new lisp objects in their own right and with the exception of symbols, are not `eq` to the original. This will not pose a problem for many applications but functional programmers may at times want to make use of the `eq` behaviour of the objects they are manipulating. It would be impractical for all objects on the processor array to have the correct `eq` behaviour, so we introduce a special class of processor array object, an *fe-object*, which is a handle of an object on the front end. These can be moved between PEs while preserving their `eq` properties. Fe-objects will also allow singular objects to be dereferenced in parallel; the efficient implementation of such an operation is beyond the scope of this paper. We do not suggest how such objects and their operations would be supported at the language level but feel they should be available as:

- They are in keeping with the spirit of Lisp.
- Recursive parallel structures fit naturally into this scheme.

Currently an fe-object is a unique key under which the object is placed in a table on the front end. This could possibly be improved by storing the address of the object but this would require modifications to the front end garbage collector. Placing the object in the table also prevents it from being garbage collected by the host if only the processor array references it. Reclaiming all the free memory requires a global garbage collector which spans both the host and the array. It is attractive to allow the two components to still GC independently, using the asynchronous execution model available on the MasPar the host and array could perform local GC during idle time. Global GC is further complicated as circular references between the array and the host are now possible.

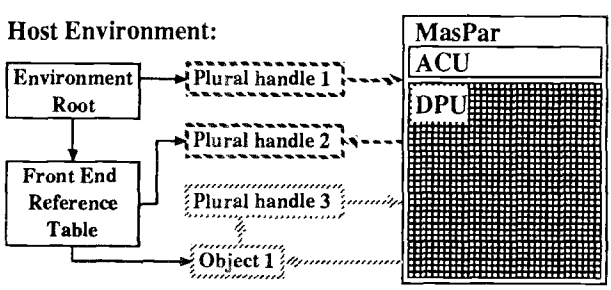


Fig. 4. Examples of BE-FE structures

Figure 4 shows some possible back-end front-end structures, plural handle 2 is legitimately held in the *fe-reference* table since plural 1, which is still in the front end environment points to it. Whereas object 1, and hence plural handle 3, are held in the table only by virtue of a circular set of pointers and so should be removed from the table so that they can be collected.

We use a method similar to that given by [3] to resolve front-end/back-end references but using the fe-reference table makes it possible to stabilise in a constant number of iterations. The table allows us to follow all the possible references to the back-end (possible in the sense some may no longer be present in the front-end environment). To do this we need to be able to find all the plural handles on the host and determine which are in the fe-reference table, the simple garbage collector will require little modification to give us this information. These references are followed on the MasPar and a list of front-end references is generated for each of these. Only these objects will be used to create a new fe-reference table, effectively freeing those no longer referenced by the processor array. To identify any circular structures we first search on the MasPar for any plurals containing plurals, this will be a router intensive operation but for a large number of front end to back end references and vice versa should prove worthwhile. We now have a concise representation of the references between the host and the MasPar in terms of the plurals they involve. We can propagate a mark through this "map" to identify which plurals in the fe-reference table should be retained. Any unmarked plurals are freed on the processor array, and the fe-objects they contain can be eliminated before the new fe-reference table is constructed, or they can be collected on subsequent garbage collections.

The techniques are, perhaps, not novel, but present an interesting use of a processor array in the resolution of complex structures. The problem of identifying circular structures in the network of inter-processor references is essentially a graph problem. Data parallel machines are well suited to such problems [4] as a PE can be used for each node of the graph and the arcs between nodes are handled by the router mechanisms. Here we have applied this power to the management of a high level data parallel language.

6 Summary

We have described here various techniques used in implementing a functional data parallel language with emphasis on efficient memory/processor management. Of special interest perhaps is the support for heterogeneous collections. This is made possible by the local indirect addressing available on the MasPar, which also permits us to effectively garbage collect the resulting heterogeneous heaps in data parallel. This gives us a Lisp system with around 600 Mega-bytes of memory (for a 16K MasPar with 64K memory option) with 16384 garbage collectors operating on it. Finally processor arrays, with their excellent PE to PE communications, are well suited to solving graph oriented problems. We make use of this feature to resolve inter-processor references when garbage collecting the environment distributed between the host and MasPar.

7 Appendices

A MasPar MP-1: Technical Summary

The MasPar MP-1 is a massively parallel SIMD machine with 1024 processors scalable to 16384. The system comprises five major subsystems:

The Array Control Unit (ACU) controls the processor array by broadcasting all PE instructions. It is also capable of independent program execution.

The Processor Element Array (PE Array) executes the instruction stream broadcast by the ACU on each PE, conditional on the activity status. Each PE has 16K of local memory which can be expanded to 64K. The CPU consists of a 4-bit ALU and 192 bytes of scratch RAM.

Communication Mechanisms include:

- The 8-way X network for communication with neighbouring PEs.
- The global router, which gives random PE-to-PE communication via a hierarchical crossbar.
- Two global busses, one for broadcasting data and instructions from the ACU and one for consolidating the status responses of all the PEs to the ACU via a logical OR-tree.

The Unix Subsystem provides UNIX services to the data-parallel system, e.g. job management.

The I/O Subsystem supports high speed communication between the host and parallel subsystem.

B MasPar Programming Language: MPL

Mpl is used to program the ACU and parallel array, it is based on K&R and C and in the tradition of C it gives the finest control over the processor array. There are three basic additions to K&R C:

- A variable can be declared as `plural` indicating it should be instantiated on all PEs otherwise it is instantiated on the ACU.
- Existing C control structures are extended to handle plural arguments, in which case the activity status of the processor array is modified before broadcasting the associated code.
- Language syntax has been added to support the use of the X-net and the global router.

Below is a segment of mpl code showing: declaration of parallel variables (line 2), parallel expressions (line 4) and X-net (line 5) and router (line 6) communication constructs:

```
(1)  {
(2)   plural int x, z;
(3)   x = 1;
(4)   z = x + iproc;
(5)   x = xnetNE[1].x + xnetSW[1].x;
(6)   z = router[x%1024].x;
(7)   }
```

C Plural EuLisp Semantic Definition

We provide a brief description of the kernel of Plural EuLisp by means of the signatures of the essential operations:

```
bang       : T → plural(T) → plural(T)
make-plural: integer → plural(T)
match      : plural(integer) → plural(integer) → map
move       : plural(T)→map→(T→T→T)→T→plural(T)
```

$$\frac{\rho \vdash E_1 \rightarrow \alpha \in \text{singular} \quad \rho \vdash E_2 \rightarrow \beta \in \text{plural(T)} \quad \gamma = \{\gamma_i = \alpha,\ \beta_i \in \beta\}}{\rho \vdash (\text{bang } E_1\ E_2) \rightarrow \gamma}$$

$$\frac{\rho \vdash E_1 \rightarrow \alpha, \alpha \in \text{integer} \quad \beta = \{\beta_i = (),\ i = 0 \ldots \alpha\}}{\rho \vdash (\texttt{make-plural}\ E_1) \rightarrow \beta}$$

$$\frac{\rho \vdash E_1 \rightarrow \alpha \in \text{plural(T)} \quad \beta = \{E_2 \rightarrow \beta_i,\ \alpha_i \neq ()\} \cup \{E_3 \rightarrow \beta_i,\ \alpha_i = ()\}}{\rho \vdash (\texttt{if-s}\ E_1\ E_2\ E_3) \rightarrow \beta}$$

$$\frac{\rho \vdash E_1 \rightarrow \alpha,\ \alpha \in \text{plural(integer)} \quad \rho \vdash E_2 \rightarrow \beta,\ \beta \in \text{plural(integer)} \quad \gamma = makeMap\{(i,j),\ \alpha_i = \beta_j,\ \alpha_i \in \alpha,\ \beta_j \in \beta\}}{\rho \vdash (\texttt{match}\ E_1\ E_2) \rightarrow \gamma}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \rightarrow \alpha,\ \alpha \in \text{plural(T)} \quad \rho \vdash E_2 \rightarrow \beta,\ \beta \in \text{map} \\ \rho \vdash E_3 \rightarrow \gamma,\ \gamma \in T \times T \rightarrow T \quad \rho \vdash E_4 \rightarrow \delta,\ \delta \in \text{singular} \\ \epsilon = map\ (reduce\gamma)\ \{\{\epsilon_j = \alpha_i \cup \epsilon_j,\ (i,j) = \beta_i, \\ \alpha_i \in \alpha\} \cup \{\epsilon_k = \delta,\ k \neq j,\ (i,j) = \beta_i\}\}\end{array}}{\rho \vdash (\texttt{move}\ E_1\ E_2\ E_3\ E_4) \rightarrow \epsilon}$$

References

1. Bale, A. *Implementing Lisp on the ICL Distributed Array Processor.* Queen Mary College, Dept. of Computer Science, 1986.
2. Blelloch, G. E. and Sabot, G. W. *Compiling Collection-Oriented languages onto Massively Parallel Computers*, volume 8, pages 119–134. Journal of Parallel and Distributed Computing, 1990.
3. Lang *et al. Gargbage Collecting the World.* ACM Symposium on Principles of Programming Languages, New York, 1992.
4. Evett, M. and Hendler, J. *Achieving Computationally Effective Knowledge Representation via Massively Parallel Lisp Implementation.* Europal Workshop for High Performance and Parallel Computing in Lisp, Nov 1990.
5. Fitch, J. P. and Norman, A. C. *A Note on Compacting Garbage Collection*, volume 10, pages 31–34. The Computer Journal, July 1976.

6. Haddon, B. K. and Waite, Q. M. *A Compacting Procedure for variable-length storage elements*, volume 10, page 162. The Computer Journal, 1967.
7. Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
8. Padget, J. A. and Nuyens, G. *The EuLisp Definition.* to be published by the Commission of the European Communities, 1992.
9. Sabot, G. W. *Paralation Lisp Reference Manual.* Thinking Machines Corp., 1988. Tech. Report PL87-11.
10. Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming.* MIT Press, Cambridge, MA, 1988.
11. Steele, G. L., Jr., and Hillis, W. D. *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing*, pages 279–297. ACM Conference on Lisp and Functional Programming, 1986.
12. Steele, G. L., Jr., and Wholey, S. *Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming.* International Conference on SuperComputing, 1987. TMC Tech. Report PL87-6.
13. Thinking Machines Corporation. **Lisp Reference Manual*, 1988.
14. Tomboulian, S. and Pappas, M. *Indirect Addressing and Load Balancing for Faster Solution to Mandelbrot Set on SIMD architectures.* MasPar Corporation Tech. Report, October 1990.

This article was processed using the LaTeX macro package with LLNCS style

Memory Management and Garbage Collection of an Extended Common Lisp System for Massively Parallel SIMD Architecture

Taiichi Yuasa[1]

Toyohashi University of Technology, Toyohashi 441, Japan

Abstract. We have developed an extended Common Lisp language and system, called TUPLE, for massively parallel SIMD (Single Instruction stream, Multiple Data stream) architecture. The system is an extension of Common Lisp with features for SIMD parallel computation.
Unlike other Lisp languages on SIMD architecture, TUPLE supports the programming model that there are a huge number of subset Common Lisp systems running in parallel. For this purpose, each processing element (PE) of the target machine has its own heap in its local memory. In addition, there is a full-set Common Lisp system with which the user interacts to develop and execute parallel programs. The result is that there are huge number of heaps with pointers across heaps.
This paper briefly introduces the TUPLE language and system, and then describes the memory management and garbage collection of the TUPLE system. In particular, we focus on the current implementation of TUPLE on the SIMD machine MasPar MP-1 with at least 1024 PEs.

1 Introduction

TUPLE (Toyohashi University Parallel Lisp Environment) is an extension of Common Lisp [6] with functions for massively parallel computation. The TUPLE system is based on KCL (Kyoto Common Lisp [9]), a full-set Common Lisp system developed by the group including the author. The system is currently running on the MasPar MP-1, a SIMD massively parallel computer with at least 1024 PEs. As the original KCL is written partly in C and partly in Common Lisp, the TUPLE system on the MasPar is written partly in MPL [15], the extended C language on the MasPar, and partly in TUPLE itself.

So far, several Lisp languages have been proposed for SIMD architectures. These languages provide new data structures that can be handled in parallel. Examples are: *xappings* in Connection Machine Lisp [7, 8, 12], *paralations* in Paralation Lisp [4, 5], *plurals* in Plural Eulisp [2] and *pvars* in *Lisp [14, 13]. These languages share a same computation model that the front-end processor dominates the entire control flow and the PEs are used for parallel execution of operations on the extended data structures. TUPLE adopts a different approach. Its computation model is that there are a huge number of Lisp systems called *PE subsystems* running in parallel. These PE subsystems executes programs in a subset Common Lisp. In addition, there is a full-set Common Lisp system, called the *front-end system*, with which the user interacts to develop and execute parallel Lisp programs. With this model, the user

can write most part of his or her parallel programs in the same way as in ordinary sequential Lisp languages.

In order to realize this computation model of TUPLE, each PE subsystem must at least have the ability for symbolic computation and list processing, as well as the facility to transfer Lisp data objects to/from other PE subsystems and the front-end system. To fulfill these requirements in an efficient way, the following decisions are made for the TUPLE implementation.

- Each PE has its own heap where cons cells are allocated.
- Each PE can reference any Lisp objects in the heaps of other PEs or the front-end.
- Pointers to a single data object are represented uniquely among all PEs and the front-end.

The result is that there are a huge number of heaps (which are relatively small except for the front-end heap) with pointers across heaps. As well as other parallel architectures, communication among the PEs and the front-end is expensive in SIMD architecture. Thus garbage collection is an important issue in implementing TUPLE efficiently.

This paper reports the memory management and garbage collection of the TUPLE system. In particular, we focus on the current implementation of TUPLE on the MasPar MP-1. Since the language and the computation model of TUPLE is quite unique, we first introduce TUPLE through examples in Sections 2, 3, and 4, particularly focusing on SIMD parallel list processing. Then we overview the implementation of TUPLE on the MasPar in Section 5. We then report the main issues of this paper, data representation and garbage collection of TUPLE, in Sections 6 and 7, respectively. For the details of TUPLE, refer to [10]. For the performance measurements of the TUPLE system on the MasPar, refer to [11].

2 A Simple Example

This section introduces the language and system of TUPLE, through a simple example function **abs** that computes the absolute value of the given argument. In ordinary Common Lisp, this function can be defined as follows.

```
(defun abs (x)
  (if (>= x 0) x (- x)))
```

That is, if the argument is greater than or equal to zero, then the function simply returns the argument. Otherwise, the function returns the negative of the argument. By replacing **defun** with **defpefun**, the similar function will be defined in the PE subsystems.

```
(defpefun abs (x)
  (if (>= x 0) x (- x)))
```

When this ***PE function*** is invoked, all PEs receive independent values, one value per PE. Then those PEs that received non-negative numbers return the arguments. The other PEs return the negatives of the arguments.

TUPLE runs on SIMD architecture, where no two PEs can execute different instructions at the same time. What actually happens is the following. When the PE function **abs** is invoked, those PEs that do not satisfy the condition becomes inactive while the other PEs (i.e., those PEs that satisfy the condition) evaluate the *then* clause. Then the activity of each PE is reversed and the previously inactive PEs evaluate the *else* clause, while the previously active PEs are inactive.

Below is an example interaction between the user and the TUPLE system. The top-level of TUPLE is similar to that of ordinary Common Lisp systems. The user can input any Common Lisp form at the top-level. Then the form is (sequentially) evaluated and the result is displayed.

```
% tuple
TUPLE (Massively Parallel KCL)

>(defun abs (x)
   (if (>= x 0) x (- x)))
ABS
>(abs -3)
3
```

In order to start a parallel computation, the user has to supply a form in the extended language of TUPLE.

```
>(defpefun abs (x)
   (if (>= x 0) x (- x)))
ABS
>(ppe penumber)
#P(0 1 2 3 ...)
>(ppe (abs (- penumber 2)))
#P(2 1 0 1 ...)
```

In this example, the user uses the **ppe** form that passes a *PE form* to PE subsystems for evaluation and displays the results. This form is mainly used at the top-level of TUPLE to supply a top-level form to the PE subsystems.

The **penumber** in the above example is a built-in constant in PE subsystems which holds the processor number for each PE. The "first" processor has **0** as the value of **penumber**, the "second" has **1**, and so on. The second **ppe** form computes the absolute value of **penumber** – 2 by calling the PE function **abs**. Thus, the first processor, for instance, returns 2 as the value.

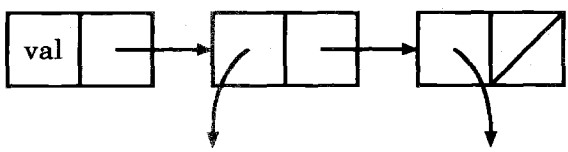
Note that TUPLE uses distinct name spaces for ordinary sequential functions and PE functions. In the example, **abs** names both the sequential function and the PE function. This is because a PE function is defined in the PE subsystems, whereas a sequential function is defined in the full-set Common Lisp system on the front-end with which the user interacts.

The **ppe** form in the example displays the values returned by the PEs, but it does *not* return the values. Actually, it returns "no value" in the terminology of Common Lisp. The so-called "parallel values" or "plural values" are not first-class objects in TUPLE. In order to obtain a single value from the values of PEs, the user has to use

a *reduction* operation. TUPLE supports a variety of reduction operations, such as the one to choose the value returned by a specified PE, the one to randomly choose a non-`nil` value among the returned values, and the one to sum up all the returned values.

3 Parallel List Processing

As a typical example of parallel list processing, we will show how binary search trees can be handled in TUPLE. In Lisp, each node of a binary search tree can be represented as a list of three elements.



The first element is the value of the node, the second and the third elements are respectively the left and the right subtrees of the node. Node values in the left subtree are all less than the current node value, and node values in the right subtree are all greater than the current node value. Ordinary binary search function can then be defined so that it recursively descends the given binary search tree, to find the given item in $\log n$ time, with n being the number of nodes in the tree.

In order to parallelize the binary search function, we assume that the entire binary search tree is represented by disjoint *PE trees*, one per PE. Each PE tree of a PE is itself a binary search tree that is constructed with cons cells in the PE subsystem of the PE. If any pair of two PE trees are disjoint (i.e., have no common node value), then we can regard the whole collection of the PE trees as a large binary search tree. We will show later how such PE trees can be constructed in TUPLE.

The parallel version of the binary search function can be defined as follows.

```
(defpefun binary-search (tree item)
  (if (null tree)
      nil
      (exif (= (car tree) item)
            t
            (binary-search
              (if (> (car tree) item)
                  (cadr tree)
                  (caddr tree))
              item))))
```

The point here is that, when one of the PEs finds the item in its PE tree, the other PEs need not go further. Rather, we would like to stop computation as soon as a PE finds the item. Since this kinds of processor synchronization is common to many parallel algorithms, we introduce a new construct **`exif`** (exclusive if). The **`exif`** form

(**`exif`** *condition then-clause else-clause*)

is similar to the ordinary **if** form, but if some PEs satisfy the *condition*, then the other PEs do not evaluate the *else-clause*. The parallel binary search function above returns immediately if the current node value for some PE is equal to the item, in which case that PE returns the true value **t** and the rest of the PEs return the false value **nil**. By computing the logical *OR* of all values returned by the function, we can determine whether the item was found in one of the PE trees. Or, by asking which PE returned the true value, we can determine in which PE tree the item is registered. TUPLE supports reduction operations for these purposes.

In order to construct PE trees, we use the following PE function, which inserts an item into one of the PE trees.

```
(defpefun bs-add (place n)
  (cond ((some-pe (null (car place)))
         (exif (binary-search (car place) n)
               nil
               (when (= (some-penumber
                          (null (car place)))
                        penumber)
                 (rplaca place
                   (list n nil nil)))))
        ((some-pe (= (caar place) n))
         nil)
        (t (bs-add
             (if (> (caar place) n)
                 (cdar place)
                 (cddar place))
             n))))
```

To simplify the algorithm, we pass to the function the place holders of the current subtrees. Each place holder is a cons cell whose **car** part points to the current subtree. By replacing the **car** part of a place holder with a pointer to a new node, we can easily expand a tree. Initially, each PE has an empty tree. This initialization can be done by the following top-level form.

```
>(defpevar pe-tree (list nil))
```

This form defines a *PE variable* named **pe-tree**, whose initial value is the place holder for the empty tree for each PE. By invoking the PE function **bs-add**, the specified item will be inserted into one of the PE trees. For example,

```
>(ppe pe-tree)
#P((NIL) (NIL) (NIL) (NIL) ...)
>(ppe (bs-add pe-tree 503))
#P(T NIL NIL NIL ...)
>(ppe pe-tree)
#P(((503 NIL NIL)) (NIL) (NIL) (NIL) ...)
```

By repeatedly invoking the function, we can construct PE trees in the PE local memories.

The difficulty here is that we have to decide to which PE tree the specified item is to be inserted. Perhaps the best choice will be to choose a PE that first reaches a leaf. With this simple choice, the PE trees are kept balanced among PEs. To do this, the PE function **bs-add** uses two built-in predicates **some-pe** and **some-penumber**. The predicate **some-pe** returns the true value to all PEs if its argument is true for some PE, and **some-penumber** returns the processor number of such a PE to all PEs. By comparing the value of **some-penumber** with the processor number of each PE, we can choose one of the PEs that satisfy a given condition. In the definition of **bs-add**, if some PE reaches a leaf, then the other PEs just search their subtrees. If some PE finds the item, then **bs-add** returns the false value, meaning that the item is already in some of the PE trees. Otherwise, we choose one of the PEs that has already reached a leaf and expand the leaf with a new node.

4 Other Parallel Features

In addition to parallel list processing, the PE subsystems support parallel computation on fixnums, short-floats, characters, and vectors (one-dimensional arrays). Among these, fixnums, short-floats, and characters are handled exactly in the same manner as in the front-end Common Lisp system.

Vectors in PE subsystems are defined in the PE subsystems by the following top-level form.

```
(defpevector v 10 nil)
```

This form creats a vector of length 10 in each PE subsystem and initializes the vectors so that all elements are **nil**. The created vector becomes the value of the PE variable named **v**. PE vectors are first-class objects as ordinary Common Lisp vectors, and PE functions similar to Common Lisp functions on vectors are available on PE vectors. For example, the form

```
(vset v i x)
```

replaces the **i**th element of the PE vector **v** with the value of the PE variable **x**. This operation is performed by the PE subsystems in parallel. Note that the vector index **i** and the new value **x** may be different among PE subsystems.

In addition to these "parallel objects", each PE subsystem can reference any objects in the front-end Common Lisp system. However, since communication between the front-end and PEs is essentially sequential in most SIMD machines, operations to obtain information in front-end objects are executed sequentially, one PE at a time. Thus the only parallel operation on front-end objects are the equality comparison.

Another important feature is the communication facility among PE subsystems. Since the actual communication network among PEs depends on the target machine, **TUPLE** supports only two kinds of PE communications: mesh communications and global communications. By a mesh communication, each PE can receive an object from its north, east, west, or south neighbor. By a global communication, each PE can receive an object from any specified PE. These communications are supported by most SIMD machines and thus can be implemented directly by the hardware communication networks.

TUPLE is designed so that each PE subsystem be as close to the front-end Common Lisp system as possible. Below is the list of special forms and built-in macros in the PE subsystems, that have counterparts in Common Lisp. They work almost the same way as the corresponding Common Lisp forms.

```
and       function   prog1
case      if         prog2
cond      labels     progn
declare   let        psetq
do        let*       quote
do*       locally    setq
dolist    loop       unless
dotimes   macrolet   when
flet      or
```

Because of the SIMD nature, some parallel versions of Common Lisp functions require *uniqueness* on their arguments. For example, the Common Lisp function `funcall` is used to dynamically specify the function to invoke. The function to invoke is given as the first argument to `funcall`. The PE function `funcall` works in the same way, but in SIMD architecture, only one function can be invoked at a time. Thus the PE function `funcall` requires its first arguments to be identical among all PEs. That is, when the following form is evaluated in PE subsystems,

```
(funcall f x y)
```

all values of `f` in the PE subsystems must be identical.

5 Implementation Overview

In this and the following sections, we will report implementation of TUPLE on the MasPar MP-1. The MasPar MP-1 is a SIMD machine with at least 1024 PEs. This machine consists of two parts: the front-end UNIX workstation and the back-end called the data parallel unit (DPU). The back-end consists of the array control unit (ACU), which broadcasts instructions to PEs, and the PE array, where PEs are aligned in a two-dimensional array. A program on the MasPar consists of front-end functions and ACU functions. Parallel computation begins by invoking an ACU function from a front-end function. The memory size in each component is relatively small. The size of the data memory in the ACU is 128 Kbytes and the size of the memory in each PE is 16Kbytes. Virtual memory is not supported on these memories.

Before the current version, we implemented a prototype version of TUPLE [1](see Figure 1). In that version, all PE functions (both built-in and user-defined) were stored in the front-end memory. The parallel evaluator also resided in the front-end. In the prototype version, therefore, several subroutines were implemented on the ACU, which are invoked by the parallel evaluator in the front-end. Since most of the subroutines were responsible for small jobs such as popping up the PE stack, communication between the front-end and the back-end took place frequently. Unfortunately, this communication is very slow and accordingly the performance of the prototype version was not satisfactory.

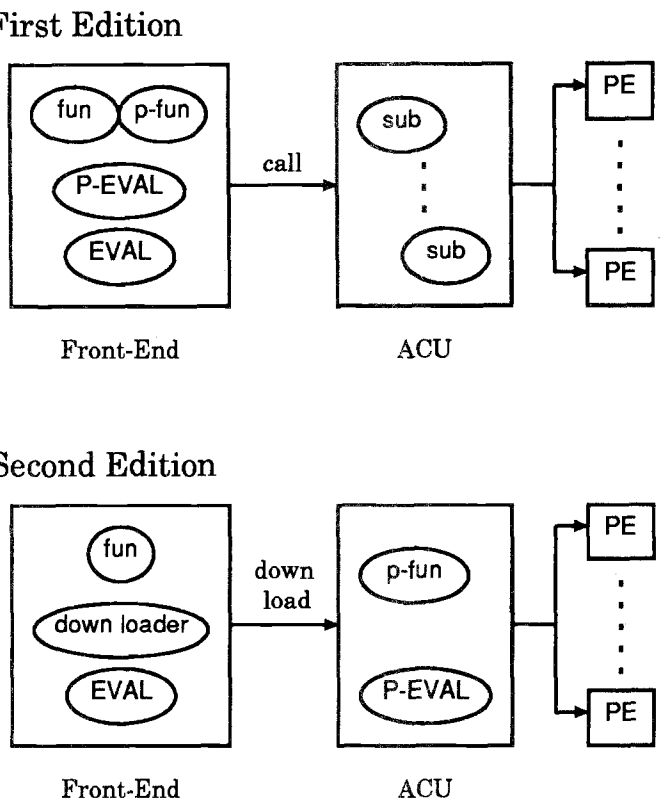


Fig. 1. Two versions of TUPLE

To improve the performance, the current version of TUPLE stores all PE functions in the ACU and the parallel evaluator runs in the ACU. When the user defines a new PE function, the *downloader* in the front-end puts the function definition into the ACU memory. Some front-end forms such as **ppe** downloads PE forms into the ACU memory before passing control to the parallel evaluator. Thus, in the current version, once triggered by the front-end, the entire parallel computation is performed solely in the ACU and no communication takes place between the front-end and the back-end.

To sum up, the current implementation of TUPLE on the MasPar uses three kinds of heaps:

- the font-end heap where ordinary Common Lisp objects are allocated
- the PE heaps where PE cons cells are allocated
- the ACU heap where those objects common to all PE subsystems, such as PE function objects (including built-in functions, user-defined functions, and function closures) and PE vector headers, are allocated

Any object in one of these heaps can be referenced from any component of the MasPar system. For example, an object in the front-end heap may be referenced

from the ACU (as part of a user-defined function) and PEs (by broadcasting). Also, a cons cell in a PE heap may be referenced from the front-end (by reductions), the ACU, and the other PEs (by PE communications).

6 Data Representation and Allocation

Figure 2 illustrates data representation of TUPLE. This representation is common to all components of the MasPar system. By having the same representation, we can avoid data conversion in communications among the components. The first four formats are those used by the original KCL. As seen from the Figure, the two least significant bits are used to distinguish these four formats. Since the third significant bit of a character object is always 0, we extended the data representation of KCL so that pointers to the ACU and PE heaps are distinguished by the bit pattern 110 at the three least significant bits.

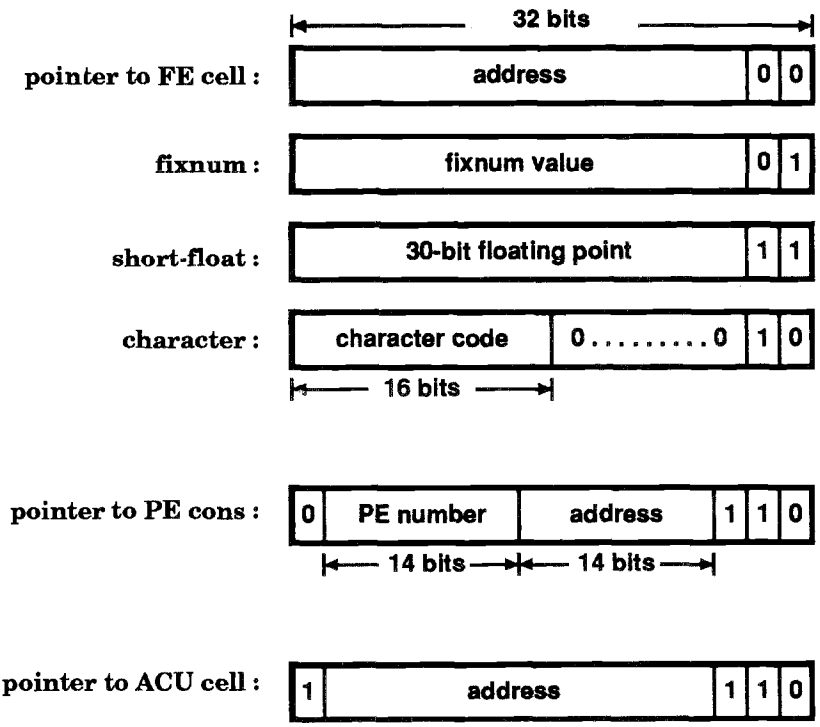


Fig. 2. Object representation of TUPLE

Figure 3 shows the data area of each PE that TUPLE handles directly. The run-time stack of MPL is not shown in the Figure. The first words of the memory area are used to allocate built-in constants `nil`, `t`, and `penumber`. Next to these words is the PE global area, where user-defined global PE variables, constants, and vectors are allocated. This global area expands dynamically as the user defines PE variables

etc. Next is the PE stack area where local PE variables are allocated and temporary values (including arguments to PE functions) are stored.

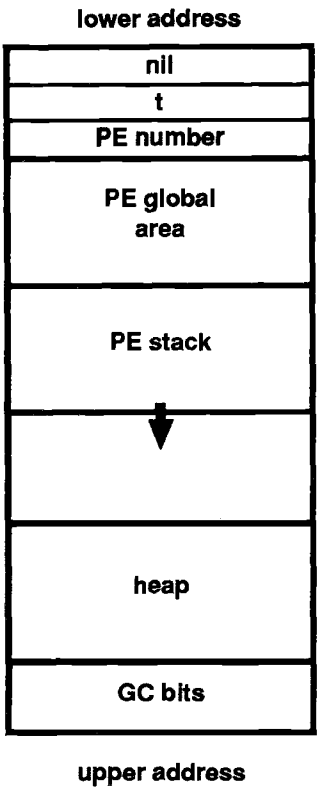


Fig. 3. The PE data area

Then there is a heap area where PE cons cells are allocated. Mark bits for PE cons cells are separately stored in the last part of the data area. These mark bits are used by the parallel garbage collector of TUPLE to reclaim unused PE cells (see the next section). The total size of the data area is 8 Kbytes for the MasPar system that has 16 Kbytes of local memory per PE. Half of the data area is used as the heap. Since each cons cell occupies 8 bytes (i.e., 2 words), 512 cells are available per PE.

Cells in the ACU heap are homogeneous and each cell occupies four words. The first word of a cell is used as the data tag and the GC mark bit. Use of the remaining three words depends on the tag. For the ACU cell for a PE vector, these words contain the name, the length, and the address of the PE vector in each PE global area. Each user-defined global function is represented by an ACU cell whose three words contain the name, the lambda list, and the body of the function.

Global PE variable bindings and global PE function bindings are represented by ACU cells called *ACU symbol cells*. Each ACU symbol cell corresponds to an ordinary front-end symbol, and no two ACU cells correspond to a same front-end

symbol. When a global PE variable is defined (typically by **defpevar**) or when a global PE function is defined (typically by **defpefun**), a new ACU symbol cell is created that corresponds to the name of the PE variable or the PE function if and only if there exists no such ACU symbol cell. The downloader converts all references to global PE variables and global PE functions in a PE form, to pointers to the corresponding ACU symbol cells.

Each ACU symbol cell contains the address of the global PE variable in the PE global area, the PE function object (pointer to an ACU cell), and the pointer to the corresponding front-end symbol. The pointer to the front-end symbol cell is mainly used in error messages in case unbound variables and undefined functions are detected. On the other hand, each front-end symbol cell contains a pointer to the corresponding ACU symbol cell, if one exists. This pointer is used mainly by the downloader for conversions from front-end symbols to back-end symbols.

As of the front-end cells, cells other than symbol cells are represented exactly in the same way as in the original KCL. Front-end symbol cells are extended so that they can contain information on parallel computation such as the ACU routine that handles a PE special form, and the pointer to the corresponding ACU symbol cell. Thus modification of the front-end system was surprisingly small.

7 Garbage Collection

As already seen, the implementation of TUPLE on the MasPar has the following unique features in relation with garbage collection.

1. It has multiple heaps: the front-end heap, the ACU heap, and a large number of PE heaps.
2. Each cell in a heap may be referenced from any component of the MasPar system.
3. Communication between the front-end and the back-end is relatively slow.
4. PEs can execute instructions in parallel.

The second feature implies that garbage collection cannot be done separately for each heap and that communications are inevitable between the front-end and the back-end during garbage collection. The third feature requires some mechanism to reduce communications during garbage collection. The last feature encourages us to develop parallel algorithm for garbage collection on PE heaps.

The original KCL uses the conventional mark-and-sweep algorithm. Free cells of the same size are linked together to form a free list. Some objects such as arrays are represented by a fixed-length *header cell* which has a pointer to the *body* of the object. For an array, for instance, array elements are stored in the body and the header cell contains various information on the array, such as the dimensions. Bodies are allocated in a special area called the *relocatable area* and are relocated during the sweep phase to make a large free space in the relocatable area. This implementation is closely related with the fact that KCL is written in the C language. Since bodies are always referenced via the header cells, we do not need to change the values of C variables that hold KCL objects, even when bodies are relocated. Note that the address of a C variable depends on the C compiler. Thus, this implementation of the original KCL increases the portability of the system.

Since TUPLE is written in MPL, a data-parallel extension of the C language, we essentially use the mark-and-sweep garbage collector for back-end heaps. Remember that all cells in a back-end heap are homogeneous. This includes that we need only one free list for each heap, and that no relocation is necessary once the size of the heap is fixed.

The garbage collector of TUPLE is invoked when one of the free lists becomes empty or when the relocatable area of the front-end becomes full. As in KCL, the garbage collector consists of two phases:

1. the mark phase, when all cells in use are marked, and
2. the sweep phase, when each non-marked cell (i.e., garbage cell) is linked to a free list and each body whose header cell is marked is relocated.

The sweep phase can be executed for each component of the MasPar, independently of the other components, for the following reasons.

- Each garbage cell in a component is linked to a free list in the same component.
- There is no pointer that points to the body in the front-end directly from the back-end. Even when a body is relocated in the front-end, no back-end pointers need to be changed.

Therefore, we use the sweep phase routine of the original KCL without changes. The sweep phase routine for ACU cells is obvious. It scans the entire ACU heap and simply links non-marked cells to the free list of the ACU. The similar routine works for cells in each PE heap, and this routine can be executed in parallel by all PEs, without overhead such as PE synchronization. The global area of each PE may contain vector bodies and thus we need to compactify the area during the sweep phase. We will explain how the global area is compactified, in a later subsection.

The mark phase, on the other hand, is not so easy because it requires communications between the front-end and the back-end, and we had to exploit an efficient algorithm, which we report in the following subsections. We first explain the marking algorithm for PE cells, which is the most sophisticated part of the mark phase, and then we explain the entire mark phase of the current implementation of TUPLE on the MasPar.

7.1 Marking PE Cells

In order to mark PE cells efficiently in parallel, we use a special bit called the request bit for each PE cell, as well as the ordinary mark bit. The request bits are used to remember those PE cells whose *car* and *cdr* fields are to be taken care of by the PE marking routine. Rather than recursively traversing pointers to PE cells, the PE marking routine repeatedly scans the PE heaps and sets on the request bits of those cells that are pointed to from within PE heaps. Here is the algorithm.

```
more := true;
while more do
    more := false;
    for i from 0 to M − 1 do
        if i.request then
```

```
            if not i.mark then
                mark_object(i.car);
                mark_object(i.cdr);
                i.mark := true;
                more := true;
            endif
            i.request := false;
        endif
    endfor
endwhile
```

where M is the number of cells in each PE heap, *i.request* and *i.mark* are the request bit and the mark bit, respectively, of the ith cell, and *i.car* and *i.cdr* are the *car* and *cdr* fields of the ith cell, respectively. Note that this algorithm is intended to be executed by all PEs in parallel. It is straightforward to implement this algorithm in a SIMD parallel language such as MPL. We will show later how the request bits are initialized.

This algorithm requires no extra stack space and thus is suitable for marking on machines with very small amount of memory. Although the algorithm requires one request bit per PE cell, the size of memory required for the entire request bits of a PE is only $M/64$ words on 32-bit architectures. About the run-time efficiency of the algorithm, the algorithm requires M^2 time in the worst case. If every cell i $(0 < i < M)$ points to the cell $i - 1$, then the body of the **while** statement can mark only one cell per iteration. On the other hand, the ordinary recursive marking algorithm requires M time in the worst case. Remember, however, that our target is an SIMD architecture. Even if each PE can finish the marking in time M, the entire execution may require MN time in the worst case, with N being the total number of PEs, because of different shape of structures among PEs. In the current implementation of TUPLE on the MasPar with 1024 PEs, $M = 512$ and $N = 1024$. Thus the above algorithm is superior to the parallel version of the ordinary recursive algorithm, in the worst case.

The parallel subroutine *mark_object* receives an object x for each PE and behaves as follows.

1. If x is a pointer to a PE cell, then set on the request bit of the cell.
2. If x is a pointer to a front-end cell, then save it into the *FE buffer*. We will explain below how this case is handled.
3. If x is a pointer to an ACU cell, then call the ACU marking routine.

Note that the first two cases can be handled by PEs in parallel, but the last case cannot because the ACU marking routine can handle one pointer at a time.

In the second case above, we use the buffer to temporarily store pointers to front-end cells, in order to reduce communications between the front-end and the back-end. This buffer consists of a single word per PE, and therefore the entire buffer can store as many pointers as the total number of PEs. When the entire buffer becomes full, all pointers in the buffer are block-transferred to the front-end and the front-end marking routine is invoked.

If a PE receives a pointer p to a front-end cell as the argument to *mark_object*, then it first tries to save p into its own buffer word w. However, the buffer word

may already be occupied by some pointer that was saved by a previous call of *mark_object*. Even in that case, there may remain some free words in the buffer. Even if the entire buffer is full, we can still have chance to find space in the buffer because some pointers in the buffer may be duplicated. These considerations suggest the possibility to defer control transfer to the front-end.

The algorithm below will be executed when there are more than one *occupied PEs*, that is, those PEs that received a pointer p to a front-end cell as the argument to *mark_object* and whose buffer word w is already occupied. In the algorithm, N denotes the total number of PEs. We assume that PEs are numbered 1 to N and the ith PE is denoted as P_i. By *free PEs*, we mean those PEs that received an object other than a pointer to a front-end cell as the argument to *mark_object* and whose buffer word w is not occupied yet.

1. Give a unique ordinal number *ord* to each occupied PE. Let N_s be the largest ordinal number, i.e., the total number of occupied PEs. Let s_i denote the PE number of the occupied PE that is given the ordinal number i.
2. Give a unique ordinal number *ord* to each free PE. Let N_d be the largest ordinal number, i.e., the total number of free PEs. Let d_i denote the PE number of the free PE with the ordinal number i.
3. For each occupied PE P_{s_i} such that $1 \leq i \leq min(N_s, N_d)$, save the pointer p into the buffer word w of the ith free PE P_{d_i}. This step consists of the following substeps.
   (a) For each occupied PE P_{s_i} such that $1 \leq i \leq min(N_s, N_d)$,
   $tmp \odot ord \Leftarrow p$
   (b) For each free PE P_{d_i} such that $1 \leq i \leq min(N_s, N_d)$,
   $dest \odot ord \Leftarrow d_i$
   (c) For each P_i such that $1 \leq i \leq min(N_s, N_d)$,
   $w \odot dest \Leftarrow tmp$
   Here, "$x \odot y \Leftarrow z$" means to assign the value of z to the variable x of P_y. This operation includes PE communication from the source PE to P_y.
4. If $N_d < N_s$, remove duplicated pointers in the entire buffer (which is full now) as follows.
   (a) Sort the pointers in the buffer so that the pointer value of the buffer word w of P_{i-1} becomes less than or equal to that of P_i for all i such that $1 < i \leq N$.
   (b) For each P_i $(1 < i \leq N)$, if the pointer value in its buffer word w is equal to that of P_{i-1}, then clear w.
   Then repeat Steps 1 to 3 once more. After that, if $N_d < N_s$ again, then block-transfer the entire buffer to the front-end and invoke the marking routine of the front-end. On return from the front-end marking routine,
   (a) Clear the entire buffer.
   (b) For each occupied P_{s_i} $(N_d < i \leq N_s)$,
   $w := p$

Operations in the algorithm are implemented efficiently on SIMD architecture such as the MasPar. We can use well-known $\log N$ time parallel algorithms for giving ordinal numbers in Steps 1 and 2 and for sorting the entire buffer in Step 4 (refer, for example, to [3]). In the MasPar, these algorithms are implemented as system

libraries. Step 3 requires PE communications three times. These PE communications are performed through the global PE communication network such as hyper cube, and thus are highly efficient on many modern SIMD machines, including the MasPar.

7.2 The Mark Phase

To simplify the algorithm, TUPLE always starts garbage collection with the top-level marking routine of the front-end. This routine scans the root locations in the front-end memory such as stack entries, and traverses pointers while marking all front-end cells it encounters. When it encounters a pointer to a cell in a back-end heap, it temporarily stores the pointer in a buffer and keeps going. There are two such buffers in the front-end: the *ACU buffer* for pointers to ACU cells and the *PE buffer* for pointers to PE cells. The buffers are used to reduce overhead of the communication between the front-end and the back-end.

When the ACU buffer becomes full, pointers in the buffer are block-transferred to the ACU and the marking routine of the ACU is invoked. On the other hand, when the PE buffer becomes full, the *requesting routine* is invoked, which sets on the request bits of those PE cells that are pointed to by the pointers in the PE buffer. When these routines return, execution of the top-level marking routine of the front-end resumes.

Note that the PE marking routine described in the previous subsection is never invoked during the top-level marking routine of the front-end. It is invoked only after all root locations in the front-end memory have been scanned. Thus all PE cells that are referenced from the front-end have been set their request bits on, at the first invocation of the PE marking routine. This reduces the number of repetition of the outermost loop of the PE marking routine.

The ACU marking routine traverses pointers and marks all ACU cells it encounters. When it encounters a pointer to a PE cell, it simply sets on the request bit of the PE cell. Again, this is because we would like to defer the call to the PE marking routine so that as many PE cells as possible have been set on their request bits when the PE marking routine is invoked next time.

When the ACU marking routine encounters a pointer to a front-end cell, it does not invoke the front-end marking routine immediately. Rather, it saves the pointer into the FE buffer that is used to save pointers from PEs to the front-end. By using that buffer, rather than another buffer in the ACU memory, we have the chance to remove duplicated pointers to the front-end efficiently in parallel in the way described in the previous subsection.

7.3 Compaction of the PE Global Areas

The global area of each PE may contain (global) PE variables and PE vectors. (In this subsection, we treat PE constants as global PE variables, since the difference is inessential in the context of garbage collection.) Since these may become garbage, we have the chance to compactify the global area and leave space for more global PE variables and PE vectors. In addition, compaction of the global area makes more space for the PE stack.

The following design decisions of TUPLE makes it simple to compactify the PE global area.

- The words of PE local memories at the same address are used for the same purpose. They may constitute a single PE variable, or they may constitute elements of a PE vector with the same index.
- Each PE variable is referenced only via an ACU symbol cell. Thus if the ACU symbol cell becomes garbage, the PE variable also becomes garbage. When a PE variable is relocated, we need to replace only the pointer in the ACU cell.
- Each PE vector is also referenced only via a header cell in the ACU. Thus if the header cell becomes garbage, the entire PE vector becomes garbage. When a PE vector is relocated, we need to replace only the pointer in the ACU header cell.

In order to keep track of the ACU cell that points to each word in the PE global area, TUPLE has a table of backward pointers, in the ACU memory. Note that the size of the table is equal to the maximum size of the global area in each PE, which is 512 words for the current implementation on the MasPar with 16 Kbytes of local memory per PE. This size is negligible when compared with the total size of the ACU data memory. Note also that the table need to be updated only during the sweep phase and when a new global PE variables or PE vector is being defined. Thus the maintenance of the table causes no run-time overhead.

As the result of compaction, the PE global area shrinks toward the lower address, and there becomes an open space between the global area and the stack area. This space is used as the stack area once control returns to the top-level of TUPLE. That is, the pointer that indicates the bottom of the stack is reset to the first free word when the execution of the current top-level form is finished.

7.4 Performance Measurements

So far, we have not yet made enough experiments to measure the performance of this garbage collection algorithm. However, our experiences indicates the garbage collector of the current version is more than twice as fast as the garbage collector in the first prototype version. This is quite satisfactory for the following reasons.

- The first version did not have the ACU heap. Thus there was no pointer from the front-end to the ACU. In addition, the sweep phase did not need to take care of the the ACU memory.
- The first version used the same structure of the PE stacks and the PE heaps, as the current version.
- The first version used the same parallel algorithm for sweeping the PE heaps as the current version. Thus the time for the sweep phase was shorter (since it did not take care of the ACU memory) in the first version.

These facts suggest that the marking routine of the current version is much (not just twice) faster than the sequential, non-buffering marking routine of the first version.

Acknowledgements

Takashi Okazawa implemented the back-end part of the prototype version of TUPLE. Yoshitaka Nagano and Katsumi Hatanaka have been implementing the current version in cooperation with the author. Taichi Yasumoto designed and implemented the mark phase parallel algorithm jointly with the author. Toshiro Kijima joined the design of the parallel algorithm and gave many useful suggestions based on his experiences of designing and implementing his extended C language for SIMD parallel computation. The project of TUPLE is supported partly by Sumitomo Metal Industries., Ltd. and partly by Digital Equipment Corporation.

The author is grateful for valuable comments from the referees.

References

1. Okazawa, T.: Design and Implementation of a Common Lisp System Extended for Massively Parallel SIMD Computer. Master's thesis (in Japanese), Toyohashi Univ. of Tech. (1992)
2. Padget, J.: Data-Parallel Symbolic Processing. Proceedings of the DPRI symposium, Boston (1992)
3. Quinn, M.: Designing Efficient Algorithms for Parallel Computers. McGraw-Hill (1987)
4. Sabot G.: Introduction to Paralation Lisp. Technical Report PL87-1, Thinking Machines Corporation (1987)
5. Sabot G.: The Paralation Model: Architecture Independent Parallel Programming. MIT Press (1988)
6. Steele, G.: Common Lisp the Language. Digital Press (1984)
7. Steele, G., Hillis, D.: Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. Proc. 1986 ACM Conf. on Lisp and Functional Programming (1986)
8. Wholey, S., Steele, G.: Connection Machine Lisp: a dialect of Common Lisp for data parallel programming. Proc. Second International Conf. on Supercomputing (1987)
9. Yuasa, T.: Design and Implementation of Kyoto Common Lisp. Journal of Information Processing, Vol.13, No.3 (1990)
10. Yuasa, T.: TUPLE - An Extension of KCL for Massively Parallel SIMD Architecture - Draft for the Second Edition. available from the author (1992)
11. Yuasa, T.: TUPLE: An Extended Common Lisp for Massively Parallel SIMD Architecture. Proceedings of the DPRI symposium, Boston (1992)
12. Connection Machine Lisp Reference Manual. Thinking Machines Corporation (1987)
13. Introduction to Data Level Parallelism. Technical Report PR86-14, Thinking Machines Corporation (1986)
14. *Lisp Reference Manual. Thinking Machines Corporation (1988)
15. MasPar Parallel Application Language (MPL) User Guide. MasPar Computer Corporation (1991)

This article was processed using the LaTeX macro package with LLNCS style

NREVERSAL of Fortune[1] — The Thermodynamics of Garbage Collection

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, California 91436
U.S.A.
(818) 501-4956 (818) 986-1360 (FAX)

Abstract

The need to *reverse* a computation arises in many contexts—debugging, editor undoing, optimistic concurrency undoing, speculative computation undoing, trace scheduling, exception handling undoing, database recovery, optimistic discrete event simulations, subjunctive computing, etc. The need to *analyze* a reversed computation arises in the context of static analysis—liveness analysis, strictness analysis, type inference, etc. Traditional means for restoring a computation to a previous state involve checkpoints; checkpoints require time to copy, as well as space to store, the copied material. Traditional reverse abstract interpretation produces relatively poor information due to its inability to guess the previous values of assigned-to variables.

We propose an abstract computer model and a programming language—Ψ-Lisp—whose primitive operations are injective and hence reversible, thus allowing arbitrary undoing without the overheads of checkpointing. Such a computer can be built from reversible conservative logic circuits, with the serendipitous advantage of dissipating far less heat than traditional Boolean AND/OR/NOT circuits. Unlike functional languages, which have one "state" for all times, Ψ-Lisp has at all times one "state", with unique predecessor and successor states.

Compiling into a reversible pseudocode can have benefits even when targeting a traditional computer. Certain optimizations, e.g., update-in-place, and compile-time garbage collection may be more easily performed, because the information may be elicited without the difficult and time-consuming iterative abstract interpretation required for most non-reversible models.

In a reversible machine, garbage collection for recycling storage can always be performed by a reversed (sub)computation. While this "collection is reversed mutation" insight does not reduce space requirements when used for the computation as a whole, it does save space when used to recycle at finer scales. This insight also provides an explanation for the fundamental importance of the push-down stack both for recognizing palindromes and for managing storage.

Reversible computers are related to *Prolog*, *linear logic* and *chemical abstract machines*.

---

[1] Apologies to Alan Dershowitz.

Introduction

Those behind cried "Forward!"
And those before cried "Back!"
T.B. Macaulay, *Lays of Ancient Rome — Horatius* (1842)

A physics revolution is brewing in computer science because many of the abstract models traditionally used have failed to provide deep insight into parallel and distributed computation. Discrete-time serial automata cannot faithfully model a relativistic world in which communication is more expensive than computation. Standard Boolean AND/OR/NOT logic found in all modern computers generates too much heat for use in high-performance 3D logic circuits. Parallel imperative programs have proved to be a nightmare to debug. Lattice-based compile-time analysis has reached its limits, yet significant problems in "aliasing/sharing", "strictness/laziness" and "resource estimation" remain.

Physicists, on the other hand, routinely decide deep questions about physical systems—e.g., they can talk intelligently about events that happened 15 billion years ago. Computer scientists retort that computer programs are more complex than physical systems. If this is true, then computer scientists should be embarrassed, considering the fact that computers and computer software are "cultural" objects—they are purely a product of man's imagination, and may be changed as quickly as a man can change his mind. Could God be a better hacker than man?[2]

Computation has heretofor been based on *writing* metaphors, either the *chalkboard* metaphor—e.g., the von Neumann model—or the *pen-and-paper* metaphor—e.g., the functional/logical model, rather than the *mechanical* metaphor of physics. The use of writing metaphors is curious, since a large fraction of computation is devoted to the simulation of physical systems!

The property that makes the writing metaphor so attractive is that *reading* has very little cost compared with *writing*; something can be read many times without alteration, and this property can be used to *broadcast* information for inexpensive reuse at a number of different times or locations. This property allows physical simulations to be instrumented in discreet ways not allowed by Heisenberg's principle.

Mechanical systems do not have the advantage of inexpensive copying, because mechanical objects can be in only one place at one time—they are conserved. While information can be recorded in an arrangement of mechanical objects and this arrangement can be copied into a similar arrangement of similar objects, the expense of this copying can be calculated and may be quite large. The mechanical metaphor would thus seem to be at a hopeless disadvantage compared with the writing metaphor for general purpose computing.

We have enjoyed the advantages of cheap copying so long, however, that we have difficulty perceiving its penalties. Its most apparent disadvantage is that software companies have a difficult time getting paid for their software—a significant irritation, but not important enough to dramatically change the future of computing. Illegal cloning, however, demonstrates a major problem with cheap copying—which of the copies is the "real" one, and how can multiple

---

[2]E.g., none of the Ten Commandments is concerned with von Neumann's Machine, nor does Moses mention ever seeing von Neumann on the mountain where the Commandments were obtained.

copies be kept consistent? "Object-oriented computing", a current major software trend, has at its core an assumption of *object identity*, which is the ability of a software object to have an identity which distinguishes it from other objects. In physical terms, the identity of an object must be *conserved*, meaning that it has a particular location which is different from the locations of other objects. According to Alan Kay [Kay77], it is no accident that object-oriented programming began with the *Simula* language for (physical) simulation.

A major problem in the programming of software simulations is in assuring that the simulations are faithful to the "real" system—i.e., whether the software model obeys the same conservation laws as the "real" system. If the real system is a physical system, then the conservation laws are the laws of physics—e.g., the conservation of mass, energy, etc., while if the real system is an economic one, then the conservation laws are the laws of economics—e.g., the conservation of money. Much of the complexity of modern software systems can be traced to these requirements—e.g., file backup and recovery, transaction models, type systems (including those for physical units), etc. For example, if one physically removes a physical file from a physical filing cabinet, it no longer resides in the filing cabinet, so that there is no possibility of conflict. The reading of a *software* file, on the other hand, does *not* remove the file, so we must go to extra trouble to avoid conflicts with writers and other readers. In short, we must use an lot of computing "machinery" to simulate an innately conservative "mechanical" system.

Current computer *languages* are based on ideas from 3 models—the von Neumann random-access memory, Church's lambda calculus, and Boolean logic networks. The serial nature of von Neumann RAM's shows up in conditionals, goto's, and assignments; this model has been attacked by advocates of the functional/applicative/logic languages based loosely on a write-once policy, which allows for more parallel execution while preserving Church-Rosser determinism. Boolean circuits can be claimed by both the functional/logical and the RAM communities, depending upon whether they are purely combinational or have feedback. All of these models, however, assume 1) that the *duplication* of information is free; and 2) that the *destruction* of information is free.[3] A RAM reads a cell many times between writes, knowing that the value will always be that of the last write; the S combinator cheerfully copies; and Boolean circuit models may allow indefinite fanout. A RAM write wipes out the previous memory contents; the K combinator knowingly kills; and Boolean AND's and OR's are not invertible.

Physical circuits have never lived up to these ideals. Physical storage (cores, DRAM's) must be refreshed, fan-out is limited and too much heat is generated. Yet the physical systems of which real computers are composed do not have these problems. All microscopic physical processes are inherently *conservative*, which means that in addition to conserving mass, energy, momentum, etc., they also *conserve information*. The fundamental theorem of mechanics states that "phase space is incompressible" [Penrose89], which means that two separate "states" cannot converge, nor can a single state diverge. Of course, separate points which are initially "close" in phase space may become widely dispersed; this behavior has been termed *chaotic*, even though chaotic mechanical systems still conserve information. It is this theorem that produces the limits on a Carnot heat engine, because for a Carnot engine to produce *work*, it must somehow *summarize* the information which describes the randomness in a high

[3]The "garbage collector" can erase a "write-once" memory in functional systems without contradiction, because the memory has previously become inaccessible.

temperature reservoir using a smaller amount of energy in the lower temperature reservoir. It is not the excess heat *energy* that must be exhausted into the lower temperature reservoir, but the excess *information*! Since the reliability of encoding this information is inversely proportional to the temperature, one can encode this information using less energy only when the exhaust temperature is lower.[4]

Electrical engineers spend a great deal of their time modelling traditional non-conservative two-state Boolean logic with reversible physical processes, which are most uncooperative. Thus, one must switch hundreds or thousands of electrons, and give up hundreds or thousands of "kT's" to make sure that a single bit is transmitted reliably. If we want to utilize these devices to compute conservative computations, we must then spend hundreds or thousands of devices to reliably simulate conservatism. In other words, we have given up factors of 10^6-10^9 in time and/or energy dissipation for nothing! Yet functional/logic programming, database programming and many other computations (e.g., FFT's) either are, or can readily be reorganized to be, conservative! Biological information processing, for example, dissipates far less heat/information. The processes used to copy DNA and transcribe RNA are mostly conservative, or else the heat dissipated during cell mitosis in an embryonic chicken would produce a hard-boiled egg!

If heat dissipation were the only problem with traditional computer models, the hardware circuits underlying individual functional units such as multipliers and caches could be re-implemented using conservative logic and thereby eliminate 80-90% of the wasted heat. We would then not need to change the abstraction seen by the programmer or compiler; after all, computer engineers have been hiding the physical truth from programmers for over 50 years.

Yet it is actually at the *highest* levels of computer usage where the traditional computer models are wearing the thinnest. One finds it very difficult to ensure that an accounting system "conserves money", or to ensure that a sort program "conserves records", or that a file system "preserves information". It is difficult to program a compiler to optimize the use of registers in the face of arbitrary control structures, or for parallel process to automatically "back down" during an optimistic concurrency control. Static analysis has failed to routinely produce important information, such as that needed for "escape analysis", "strictness analysis" and "aliasing/sharing analysis". For example, Milner's elegant type inference algorithm is routinely used to statically decorate ML programs with type information, which is useful for partial correctness, but type information produces at most an order of magnitude performance improvement over a dynamically typed system. Milner's algorithm can also be used to produce deeper structure sharing information [Baker90UC], but we conjecture that this usage will bring out the algorithm's latent exponential behavior. In short, static analysis techniques can produce interesting information only when their computational complexity is hopelessly exponential.

As a result of these considerations, we advocate the use of models of computation which have more structure because they obey more laws and restrictions. Strongly typed and functional systems are restrictive in their own way; we advocate restrictions which inherently conserve information—i.e., copying is expensive, and information destruction is impossible. Since

---

[4]Black holes seem to "eat" information; perhaps black holes are the "cooling fans" for the Biosphere of the universe.

information is conserved, these programs are *reversible*; reversibility becomes a property as important as the determinacy guaranteed by the Church-Rosser theorem. Because deciding reversibility for bulk computations is generally unsolvable, we guarantee it instead by constructing computations from reversible primitives which are reversibly composed.

While many computations are obviously reversible—e.g., FFT's—a skeptic might wonder how often this is true. For example, a sorting algorithm apparently throws away at least n•logn bits of information in the process of sorting n records. However, if the input records are given "serial numbers", then the original order can be retrieved by resorting on the serial numbers. Since the concatenation of serial numbers to the input records is conservative, we find that it is the *erasure* of these serial numbers at the end of sorting that inhibits reversibility, and not the sorting operation itself. The Newton iterative square root algorithm $x_{i+1}=(x_i+N/x_i)/2$ exemplifies a large class of computations in which the output appears to be independent of the input, since the Newton iteration produces the square root independent of the initial approximation. Nevertheless, if this algorithm is run a fixed number of iterations on infinite precision rational numbers, and if $x_0 \geq \text{sqrt}(N)$, it is *reversible*![5] Newton's iteration essentially encodes the initial conditions into lower and lower order bits, until these bits become insignificant. It is the *erasure* of these low order bits through rounding and/or truncation that dissipates the information needed for reversibility. In a sense, Newton's iteration converges only because Newton's inverted iteration produces chaos.

Many arithmetic operations such as increment and negate are invertible. Multiplication, however, opens up the possibility of multiplying a number by zero and losing all information about the number. In this instance, we must have a "multiplication by zero" exception which is analogous to "division by zero". Although multiplication is mathematically commutative, we need check only one argument for zero, because we will then know the other argument was zero in the case of a zero product.

Time warp [Jefferson85] was the first general scheme for reversing an arbitrary distributed computation. However, the primitive message-passing actors upon which time warp is based are traditional state machines; therefore, they can only be "rolled back" if they have saved copies of previous states. If, on the other hand, these primitive actors are all themselves reversible, then most of the clutter can be removed from this elegant scheme.

It is no accident that there is a significant overlap between those working on garbage collection and those working on "undoing" computations for purposes such as nondeterministic search, debugging, backup/recovery and concurrency control. However, the approach previously taken has been to utilize garbage collection to help manage the data structures used for "undoing". Our approach is exactly the opposite—we propose using "undoing" to perform garbage collection. In other words, the concept of reversibility is too important to be left to garbage collection and storage management. Garbage collection by undoing is more powerful than traditional garbage collection, because it can be used even for Actor systems in which the garbage may be quite active and thus hard to catch [Baker77].

---

[5]Curiously, inverting Newton's square root iteration itself requires taking a square root. However, the inputs for these roots will always be *rational perfect squares* if the initial forward approximation is rational. If the initial approximation is chosen so that x_0^2-N is *not* a perfect square, then reversal can use this property as its stopping criterion, and the algorithm becomes reversible even without the requirement for a fixed number of iterations.

Garbage Collection

Garbage collection is a favorite topic of researchers; the number of published papers on this topic approaches 1,000. Yet, the necessity for "garbage collection" in a symbolic processing system has troubled researchers from the beginning. That the symbolic computations required in Artificial Intelligence (AI) applications generate "garbage" which must be recycled, has long been a fundamental assumption; AI applications tend to "hypothesize and test", so the structures created during hypothesizing must somehow be recycled when the test is not successful.

Despite its apparent popularity, garbage collection has never been a subfield of computer science in its own right, because it is always seen as a semantics-preserving *transparent optimization* in a programming language implementation. Since it is by definition "invisible" to the programmer, the topic itself continues to be "swept under the rug". Unfortunately, the rug has developed a huge mound beneath it, because we cannot characterize on theoretical grounds the costs and benefits of various garbage collection algorithms, but must rely on relatively crude and *ad hoc* measurements. For example, we have no theoretical model which can tell us that "generational" garbage collection is inherently better than non-generational garbage collection, even though a number of measurements seem to bear this out.

Whether a computational object is "garbage" or not depends upon one's point of view. From the point of view of the computation itself—i.e., the "mutator"—a garbage object becomes inaccessible and cannot influence the future course of the computation. From the point of view of the system—"mutator" plus "collector", however, the object is still accessible, and will be reclaimed by the collector.

Since it is the mutator which "loses" objects, and the collector which "finds" them, we have another characterization of the mutator/collector interaction. What do we mean by "losing an object"? Clearly, the object is not completely lost, because the collector can find it; however, the mutator has "lost track" of it. More precisely, the mutator destroys its *information* about locating the object, and it is up to the collector to *regenerate* this information. Intuitively, the mutator is a "randomizing" influence, while the collector is an "ordering" influence. Interestingly enough, the mutator and the collector must both be either dissipative, or both conservative; one cannot be dissipative while the other remains conservative. If the collector is conservative and the mutator non-conservative, then the collector must somewhere dissipate the information it computes about which objects are garbage, since it creates a free-list with zero entropy/information. Similarly, if the mutator is conservative, then it doesn't produce garbage, and the collector is trivially conservative.

The efficiency of a generational garbage collector comes from its ability to recover storage before the mutator dissipates the information. It can do this because it can localize easily recovered garbage to a small fraction of the address space. If the newest generation consists of 1/256'th of the cells, and if half of these cells are garbage, then the *temperature* of this generation is the number of cells divided by the amount of information $\approx (N/256)/(N/256) \approx 1°$. The generational collector will have an input "temperature" of $\approx 1°$ and an exhaust "temperature" of $\approx 0.5°$. A non-generational collector collecting the same amount of garbage would have an input "temperature" of $\approx (N)/(.0204N) \approx 49°$, and an exhaust "temperature" of 511/512'th of that. Thus, a generational collector would be operating at a temperature ratio of 2:1, while a

non-generational collector would be operating at a temperature ratio of 1.002:1. Since the efficiency of a "Carnot" GC is proportional to the ratio of input to exhaust temperatures, it is easy to see why the generational GC is more efficient. (Of course, no reasonable system would operate a non-generational collector with such a small temperature differential.) Thus, the effectiveness of a generational collector depends upon reclaiming a higher fraction of cells than a full GC—i.e., a "dying" cell is statistically more likely to be young than old. If this is not true, then generational GC will not be effective.

Reference Counting and Functional Programming

The lambda calculus and combinators can be implemented using reference counting to collect "garbage", because these models do not require directed cycles. Many implementations do use directed cycles, however, for "efficiency" in implementing the Y combinator used to express recursion. However, even with these Y combinator loops, reference counting can still be used to collect garbage, because the Y loops are well-structured [Peyton-Jones87].

Functional languages cannot "overwrite" a storage location, because assignment is prohibited. This property has been called the "write-once" property of functional languages, and this property is shared with "logic languages"—e.g., Prolog. Since storage is never overwritten, the entire history of the computation is captured, and it would seem that we are very close to Charles Bennett's original thermodynamically efficient Turing Machine [Bennett73], which kept track on a separate tape of all of its actions. However, standard functional language implementations do not preserve the information regarding which branches were taken.

Interestingly, functional language implementations depend upon garbage collection to implement all of the optimizations which require "side-effects". For example, MLNJ [Appel90] does frame allocation on the garbage-collected heap, so that all garbage normally recycled by "stack allocation" is performed by the garbage collector. Furthermore, the use of shallow binding for the implementation of functional arrays [Baker91SB] allows the garbage collector to perform the in-place "assignment" normally expected in an imperative language implementation.

Since functional programming is a "good thing", and since reference counting is sufficient for these languages, it would seem that little more can be said. However, evidence is starting to mount that the lambda calculus and combinators, which are pure models of *substitution* and *copying*, may not be the best models for the highest performance parallel computers. Although one reason for this is that *copying* may be a very expensive operation for a physical computer, the major reason for the current interest in models which perform less copying—e.g., "linear logic"—is the fact that a system which copies in an unrestricted fashion also loses track of things, and requires a garbage collector.

Conservative Automata and Reversible Computation

A number of researchers have been studying the ultimate limits to computation from a physical perspective in order to guide engineers in designing high performance computers. In particular, these researchers have attacked a major problem for these computers—the generation of heat. As computers become faster and smaller, the removal of heat becomes a very difficult problem. One of the limitations on the number of active devices on a chip, for example, is the ability of the chip package to remove the heat fast enough to keep the chip from immolating itself.

It was long conjectured that heat was an essential byproduct of computation, but Bennett obliterated this conjecture by demonstrating a thermodynamically reversible computer [Bennett73]. Because it is thermodynamically reversible, any excess energy given off during one portion of the computation would be reabsorbed during another portion of the computation, leaving the entire computation energy and entropy neutral. Bennett's results show that computation in and of itself does not generate waste heat, but any *erasure* of information must necessarily generate waste heat.

Fredkin and his collaborators [Toffoli80] [Margolus88] are refining Bennett's work to show models for reversible logic which could be used on a thermodynamically-efficient logic chip. They have exhibited techniques for logic design using *conservative logic*, which conserves entropy by remaining reversible. They go on to show physics-like conservation principles, as well as logical analogues to *energy*, *entropy* and *temperature*.

We were intrigued by their simulation of traditional AND/OR/NOT Boolean logic circuits using reversible logic. In this simulation, each of the traditional gates is mapped into a configuration of reversible logic gates, but these gates have a number of "garbage" outputs, in addition to the simulated outputs. Bennett's key idea in making the emulation reversible is to mirror the mapped circuit with a reversed version of itself, with each "garbage output" being connected to the mirror image wire thus becoming a "garbage input" for the reversed circuit. So far, we have "much ado about nothing", since the output of the reversed computation is exactly the same as the original input! However, we can put additional reversible gates *in between* the mapped circuit and its reverse, which can "sense" the output to either affect some other computation, or to copy the final answer. Thus, just as the reversible computer collects its garbage bits with a reversed computation—after suitably summarizing the computation's result, we will utilize a reversed computation to recycle other kinds of resources—e.g., storage and processors.

Reversible Pointer Automata

A pointer automaton can be constructed from Fredkin's conservative logic using his universal simulation for non-conservative logic [Barton78] [Ressler81]. This simulation is not the most efficient use of logic or space, however. We describe a reversible pointer automaton which is a better model for a conservative higher level language.

Our reversible pointer automata will retain the finite state control and pointer registers, but with some restrictions. We need to make sure that each instructions can be "undone", and must therefore retain enough information about the previous state. All of the instructions are *exchanges*, conditional exchanges, and primitive arithmetic/logical operations. Arithmetic/logical instructions only come in forms that are invertible; e.g., replace the pair of registers which encode a double-precision integer dividend by an integer quotient and an integer remainder, with no change in the case of a zero divisor. Since the double-precision dividend can be reconstituted from the divisor, quotient and remainder, the instruction is invertible.

The following are primitive machine operations, where x,y,z,... denote distinct registers, x:y denotes a double-precision "register" constructed from the concatenation of x and y, and a,b,c,... denote distinct constants. Some operations have additional constraints which cause

them to "stick" in the same state. Arithmetic operations on "integers" are performed modulo 2^w, where w is the word size.

x <-> y; exchange x and y.
x <-> CAR(y); exchange x and CAR(y).
x <-> CDR(y); exchange x and CDR(y).
y := CONS(x,y) & x := nil; push x onto y and set x to nil.
if x=nil then swap(y,z); conditional exchange operation.
inc(x,y), semantics: x:=x+y; inverse is dec(x,y).
dec(x,y), semantics: x:=x-y; inverse is inc(x,y).
minus(x), semantics: x:=-x; inverse is minus(x).
mpy(x,y,z), 0≤x<z, semantics: x:y:=y*z+x; inverse is div(x,y,z).
div(x,y,z), 0≤x<z, semantics: x:=(x:y) mod z || y:=((x:y)-((x:y) mod z))/z; inv. is mpy(x,y,z).
rot(x,c) rotates the bit pattern in x by c bits; inverse is rot(x,-c)=rot(x,w-c).
xor(x,y), x,y distinct registers, has the semantics x:=x xor y; inverse is xor(x,y).
reverse(x) reverses the bits in register x; inverse is reverse(x).
macro for push(r1,r2): {car(free)<->r1; r2<->cdr(free); r2<->free;}
macro for pop(r2,r1):
 if consp(r2) and null(r1) then {r2<->free; r2<->cdr(free); car(free)<->r1;}
macro for push(r): {push(r,stack);}
macro for pop(r): {pop(stack,r);}

This machine language is carefully designed in such a way that all operations are invertible. This property requires restrictions on the topology of the program flowchart. We can "test" a value in a register, but the value must be preserved during, or restored after, the operations depending upon the test, so that when the program is reversed, we can test it again to reverse the appropriate arm of the conditional. Suitably structured programs can be statically checked for proper topology. Suzuki has proved [Suzuki82] that under these conditions 1) all reference counts are preserved, and 2) garbage cannot be created. Since all cons cells on the free list start out with a unity reference count, they always have a unity reference count—i.e., we have a Linear Lisp. Overall reversibility thus depends upon reversible primitives reversibly composed.

Programming Style

An operation P will have a "right" inverse when there exists an inverse operation P' which undoes the first operation—i.e., P•P'=I (the identity). For example, COPY and EQUAL are essentially inverses of one another, as are "destructuring bind" and "backquote". We may not be able to execute P' in an arbitrary state, however, because it may have conditions on it which cannot be satisfied. When P' is attempted in a situation in which its preconditions fail, we say that P' *sticks*, without doing anything else. Unlike most models, we do not map sticking states into an overall computational *failure*, because failure loses all information about what failed and how it failed, which is the whole point of reversible computation used for debugging.

The strangeness of the reversible programming style is due mainly to our lack of experience with it. Every subcomputation must be reversible, which means that no information is destroyed within the subcomputation. Loops in which the number of iterations is *a priori* fixed are trivial to reverse; other loops may require a counter to remember the number of iterations.

The hardest part of inverting an iterative program is to arrange that the loop iterations themselves preserve information. While a loop can always store its information in the form of a "trail" (analogous to the Prolog "trail"), the amount of storage required may grow quite rapidly. The

ability to summarize the information within a fixed number of "state variables" is the reversible computer's analogue to the "tail recursion optimization", which transforms recursive functional programs into iterative imperative programs.

A reversible computer can utilize the following optimization for loops. Most loop tests rarely succeed, and hence produce little "information"; they cannot be dispensed with, however, else the loop would never terminate. On a reversible computer the loop body can be iteratively doubled in size, and when it has exceeded the final value, the loop can be backed up to the correct point. This scheme involves only O(logn) tests instead of O(n), but the non-test work may double. As a result, it is useful only if the test is expensive relative to the body.

Input is handled by saving the input stream as in many other reversible and functional systems [Barton78] [Jefferson85]. Output could conceivably be "taken back" when reversed, as in pure *Time Warp* [Jefferson85], or saved until there is no further possibility of reversing, and then output, as in standard *Time Warp*.

Ψ-Lisp — Reversible Linear Lisp[6]

In a companion paper [Baker92LL], we introduced *Linear Lisp*, which is a variant of Lisp in which every cons cell has a unity reference count. In this section, we introduce a variant of Linear Lisp called Ψ-Lisp which looks very much like traditional Lisp, but all of its programs are reversible. The primitive operations of Ψ-Lisp are many of the operations discussed in previous sections. We make extensive use of swapping operations, as they are self inverses.

Lambda-expressions in Ψ-Lisp appear identical to those in traditional Lisp; they have a "lambda-list" of formal parameters, and a "body" consisting of a sequence of expressions. There are differences, however. The lambda-expression is considered to have a single "argument" which is a list "destructured" by the given lambda-list, in the manner of ML. All of the variable names appearing in the lambda-list must be distinct, and any reference to a free variable inside a Ψ-Lisp lambda-expression is the *only* reference to the variable to avoid violations of linearity.

The interpretation of a Ψ-Lisp lambda-expression is that the "variables" occurring in its lambda-list are "new" local variables distinct from any other variables which are bound to the respective portions of the argument list, and the cons cells from the argument list itself are returned to the free list during destructuring. The binding of these new variables is reversible, since each is a new variable which had no previous value, and the original argument list can be trivially recovered from the values of the distinct variables. The body of the lambda-expression consists of a list of Ψ-Lisp expressions, each of which is individually reversible, and which can be reversed by reversing the execution of each of the computations in the reversed list. Any changes to the values of the lambda-list variables during the execution of the body will be reversed during reversed execution, thus reconstituting their values.

It is an error if any lambda-list variable still has a value at the end of the execution of the lambda-expression body, because linearity promises that every variable will be accessed *exactly once* within the body, and any such access returns the variable to its unbound state. In other words, the argument list to a lambda-expression is completely consumed during its evaluation;

---

[6]Extra credit problem for the reader: why is it called Ψ-Lisp?

the list itself is consumed during binding, and each of the bound values is then consumed during the execution of the body. This property, along with the other properties of Ψ-Lisp, allows us to conclude that all of the information necessary for the reverse execution of the lambda expression is encoded into its returned value(s). Similarly, it is an error if any expression in the body—except for the last—returns a value; i.e., these expressions must all be operations with only side-effects, but not values. Of course, these side-effects are all completely local, since linearity forces each free variable to belong to only one closure; i.e., any "local" variables of an enclosing lambda which are referenced by a subsidiary lambda closure can no longer be accessed by the enclosing lambda.

A `let`-expression evaluates an expression and destructures it to bind several new distinct variables (destructuring means never having to say CAR or CDR again). The body of a `let` expression is a sequence of expressions, in which only the last can return a value. Like a lambda-expression, it is an error for a `let`-expression to terminate while its bound variables still have values. These rules allow the reverse execution of a `let`-expression in the same way they allow the reverse execution of a lambda-expression.

Nested expressions have an interesting interpretation. In general, a particular variable may "occur" only once. The values of sub-expressions are conceptually bound to new intermediate variables, so we could have decomposed nested expressions into an equivalent nest of `let`-expressions. The values of all of the intermediate variables are immediately consumed by the expression in which they are nested, so that they satisfy the restrictions on `let`-expressions. A nested expression is thus executed in a manner analogous to a dataflow architecture in which the values ("tokens") flow through the variables ("wires") and into the application nodes ("operators"); a variable without a value is analogous to a wire without a token. Multiple arguments may be evaluated in parallel [Baker92LL].

The most difficult and interesting case is that of the if-then-else expression. This expression has 4(!) sub-expressions—a Boolean test expression, a Boolean predicate, and two arms, a then-arm and an else-arm. If we were to execute the test expression in the normal fashion, then any arguments passed to it would be consumed, and it would be difficult to remember the direction of the evaluation during execution reversal. Therefore, we will evaluate the test expression somewhat differently from the expressions of the arms. We evaluate the test expression to determine the direction of the execution, and then *we undo the test expression to restore the values of any of its variables* before executing the appropriate arm. The variables referenced in the test expression must be referenced again in both arms, since otherwise they would not be consumed. During reverse execution, the Boolean predicate is applied to the value previously returned by the if-then-else expression to determine which arm to execute. In order to keep the programmer honest, this predicate is also applied to the value *to be returned*, to make sure that it returns true on the then arm, and false on the else arm. To allow for more traditional programming styles, we allow this boolean predicate to push some state on a hidden conditional "history" stack during forwards execution, which is popped during reverse execution to remember the direction of the branch. Since only a single bit of information is being saved for a conditional expression, and only by *some* conditional expressions, this bit stack can be implemented very efficiently. A clever optimizing compiler may be able to sometimes deduce the reverse predicate which doesn't use the bit stack; functions like these will operate with bounded history stacks and are analogous to tail recursion.

Here is a reversible version of the factorial function restricted to n>0 so as to be injective; the result is 1 if and only if the "then" arm was taken, which forces n=1.

```
(defun fact (n) (assert (and (integerp n) (> n 0)))
  (if (onep n) #'onep n (* n (fact (1- n)))))
```

Our interpretation so far is approximately consistent with an applicative/functional interpretation of the lambda-expression. We now give another, more operational, interpretation. Parameters are not passed by copy or by reference, but by "swap-in, swap-out" [Harms91]. When a variable is referenced as an argument in an expression, it is swapped out of the variable and into the argument list structure, so that a side-effect of computing the expression is to make all of the referenced variables unbound! This is why a variable should not appear more than once in an expression—succeeding occurrence will be unbound. Results are also returned by swapping, so that when a lambda-expression terminates, none of its local variables have any values.

Ψ-Lisp EVAL and APPLY look similar to their traditional Lisp counterparts, and their familiar appearance masks their metacircular ability to run backwards. There are some notable differences, however. Since EVAL must be reversible given only its computed value(s), EVAL returns 3 values: the expression, the environment and the computed value. APPLY is more symmetrical—it is given a function and an argument list, and returns the function and its value(s). Since many functions return multiple values, we make APPLY completely symmetrical—it consumes an argument list and returns a value list, which must be destructured by its recipient. We utilize these additional returned values to eliminate the need for the interpreter itself to save state, although the interpreted program itself may do so.

The code for a recursive function is incrementally copied during a recursive evaluation in a Y-like manner. Unfortunately, all of these copies become available at the end of the recursion and must be recycled. A more "efficient" scheme might keep multiple copies of the code on a separate "free list", so that they wouldn't have to be created during recursion. Tail recursive functions can not only reuse their stack frame, but their code, as well! Linear Lisp [Baker92LL] utilizes a more efficient scheme (similar to "copy on write") for managing copies.

Time and Space Complexity for a Reversible Garbage Collector

If we make the reasonable assumption (for a machine with a single level RAM memory) that the inverse execution of each instruction takes exactly the same amount of time as its forward execution, then the total time required is exactly double that of the non-collected computation, plus whatever time is required to copy the answer.

If the entire computation finishes before being reversed, then "garbage collection" has not helped at all. One can reverse portions of the computation at finer scales, however, and achieve significant storage savings. In this case, the "answer" which must be copied before the reversal is initiated is the summary of the computation "so far". In other words, we must "save" this intermediate state for later mutation (and reversal) at a larger scale. The saving of this intermediate state is equivalent to traditional marking!

The value returned from a subprogram is a good example. The "answer" can be copied, and a reversal initiated to collect any excess storage that the subprogram used, before the caller

resumes "forward" motion again. This reversal is equivalent to the popping of a classical stack frame. In fact, our "collection is reverse mutation" hypothesis can simulate most standard garbage collection optimizations.

Reverse Execution Paging

Researchers have found that garbage collection algorithms have much less reference locality than "normal" programs. Garbage collectors therefore page extensively, and many garbage collection improvements are aimed more at reducing the paging/caching costs than in reducing the number of instructions executed.

If we perform a "full" garbage collection, then every node and every link must be traced, requiring that every page/cache line be visited even if it has not otherwise been touched or modified since the last garbage collection. If we perform a "partial" garbage collection in a generational system, then optimizations exist to avoid visiting otherwise untouched or unmodified pages. On the other hand, even a generational garbage collector will have to spend some time tracing in the pages recently referenced or modified by the mutator.

If one believes the "garbage collection by reverse mutation" hypothesis, then the collector must visit the same pages/cache lines as the mutator, but does not bother with reversing a lot of extraneous computation not involved with pointers. As a result, the collector will tend to page "more heavily" than the forward computation, simply because the reverse of the operations not causing page faults are irrelevant to the task of garbage collection! Thus, although it may not visit any more pages than the forward computation, the collector appears to be less efficient, because it has less to do on each page.

If we know that collection is the reverse of mutation, we may be able to use this information to greatly improve the paging of the collector. We first note that the last-in, first-out (LIFO) behavior of a stack works extremely well with the least-recently-used (LRU) page replacement algorithm; even very small stack caches have high hit rates with LRU. It is also well known that the optimal paging algorithm (OPT) can be computed by submitting the reversed reference stream to an LRU algorithm. But we have already submitted our mutator program to the LRU algorithm during forwards execution, so we can compute the information necessary to utilize the OPT paging algorithm during the reversed execution (collection) phase!

Thus, while our garbage collection by reversed mutation would seem to exactly double the execution time of the mutator, we might reduce the collection time in the context of a paged implementation by utilizing the additional information generated during forward execution, and thereby achieve a running time only a fraction longer than an uncollected computation.[7]

Implications for Real Hardware

The biggest problem with an exchange-oriented architecture is the fact that it goes squarely against the grain of one of the most universally-held assumptions about computation—that copying by reference is free. Conversely, exchanges are cheaper than copies, even though

[7]For example, the TI Explorer GC "learns" locality by letting the running program copy incrementally; this scheme seems to provide locality superior to that from any uniform (depth-first or breadth-first) copying strategy.

"everyone knows" that exchanges take 3 copies, and that exchanges on a bus require atomic (i.e., slow) back-to-back bus cycles.

That exchanges are expensive seems to be an artifact of traditional Boolean—i.e., irreversible—logic. One can conceive of other types of logic in which the same connection could be used for signalling in both directions simultaneously—e.g., optical fiber. In fact, since all suitably small physical systems are reversible, it is actually more difficult to build irreversible/non-exchange architectures out of small components than to build reversible/exchange architectures.

Interestingly enough, one can find a precedent for exchange architecture in current *cache consistency* protocols for MIMD multiprocessors. In order to cut down on the bus/network traffic in the case that multiple writers are active, some cache line may be *owned* by a particular processor. If another processor attempts to write to this line, then the ownership of the line may be transferred to the other processor, in which case the first processor no longer owns it. While the information going in the direction opposite to that of moving cache line is essentially a "hole", it is not difficult to conceive of a more productive exchange protocol which would allow them to immediately synchronize, for example.

The restricted fan-in/fan-out of exchange architectures provides relief to the designer of a PRAM implementation, because the unlimited fan-in/fan-out of a CRCW PRAM architecture [Corman90] is very expensive to achieve.

What Makes the Free-List Free?

Joseph Halpern, *et al.* [Halpern84], asked the question "what makes the free-list free?" We believe that their paper could not properly answer the question, because it is a *thermodynamic* question, not a *semantic* question. Our answer is that the free-list is free because it is storage that is in the highest state of availability; in thermodynamic terms, it is *work*. What is work? Carnot's theorem tells us that work (available energy) can be produced from heat (energy+randomness), by removing the randomness into a cooler reservoir. The free-list always has exactly the same configuration—a semi-infinite sequence of cells (list of nil's)—i.e., the structure of the free-list is isomorphic to ω, the first infinite ordinal. Since $1+\omega=\omega$, allocating a cell from the free-list does not change its structure. Since the structure of the free-list is always the same, it has *zero* entropy/information. No other simple structure for the free-list would have as little entropy/information.

Conclusions and Previous Work

We have advocated the use of reversible models of computation, and we have shown how a particular model can be programmed to perform interesting computations. The property of reversibility is very strong, and should allow static analysis of programs to produce deeper information than is typically feasible with non-reversible models.

Hoare's, Dijkstra's and Pratt's logics all consider the operation of a program in reverse, although to our amazement, no one seems to have taken up the retrospectively obvious line of attack we here propose.[8] *Abstract interpretation* and *type inference* often consider the retrograde

[8]Linearity, even without reversibility, elegantly eliminates the *aliasing* problem which gives these logics fits!

execution of a program. Computer architects find that the reverse of reference strings are an important concept in the study of memory hierarchies; e.g., the optimal page replacement algorithm is the least-recently-used algorithm running on the reversed reference string.

Our Ψ-Lisp bears much resemblance to dataflow architectures. Its argument-consuming property neatly finesses the storage recovery problems addressed by [Inoue88].

The Deutsch-Schorre-Waite "pointer-reversing" list tracing algorithm [Knuth73] is a paradigm of reference-count-conserving programming. Binding tree "re-rooting" [Baker78b] is an even more sophisticated reference-count-conserving algorithm. Suzuki [Suzuki82] gives the fundamental properties of pointer exchange and rotate instructions. Common Lisp's `rotatef` operation comes directly from this paper.

The `unwind-protect` and `wind-unwind` operations of Common Lisp and Scheme, respectively, offer a form of "undoing" computation. In fact, some implementations of `wind-unwind` utilize the reversible "state space" tree re-rooting [Baker78SB].

The reverse execution abilities of *Prolog* were initially touted, but never developed, and modern Prologs cannot run predicates backwards (this is not the same as backtracking!). Subjunctive computing has been proposed as a valuable programming style [Zelkowitz73] [Nylin76] [Lafora84] [Heering85] [Leeman85] [Leeman86] [Strothotte87]. An enormous literature has developed around reversibility for debugging [Balzer69] [Grishman70] [Archer84] [LeBlanc87] [Feldman88] [Pan88] [Wilson89] [Tolmach90] [Agrawal91].

Hofstadter devotes a portion of his Pulitzer prize-winning book [Hafstadter79] to palindromes and "crab canons", which are musical pieces that simultaneously have same theme going in both the forward and reverse directions. One can conceive of the mutator process as playing a theme, while the collector process plays the retrograde theme.

Bill Gosper was an early investigator [Beeler72] into the power of the Digital Equipment Corporation's PDP-10 EXCH instruction, which we have appropriated for our reversible computer. Swapping has been given new life as a fundamental synchronization primitive in shared-memory multiprocessor architectures in the form of the "compare-and-swap" operation, which Herlihy has proven to be powerful and universal [Herlihy91].

Acknowledgements

We appreciate the discussions with Peter Deutsch, Richard Fateman, Richard Fujimoto, Bill Gosper, Robert Keller, Nori Suzuki, Tommaso Toffoli, and David Wise about these concepts.

References

Abadi, M. & Plotkin, G.D. "A Logical View of Composition and Refinement". *Proc. ACM POPL 18* (Jan. 1991),323-332.

Agrawal, H. *et al.* "An Execution-Backtracking Approach to Debugging". *IEEE Software 8*,3 (May 1991),21-26.

Appel, A.W. "Simple Generational Garbage Collection and Fast Allocation". *Soft. Prac. & Exper. 19*,2 (Feb. 1989), 171-183.

Appel, A.W. "A Runtime System". *Lisp & Symbolic Comput. 3*,4 (Nov. 1990),343-380.

Archer, J.E., *et al.* "User recovery and reversal in interactive systems". *ACM TOPLAS 6*,1 (Jan. 1984),1-19.

Bacon, David F., *et al.* "Optimistic Parallelization of Communicating Sequential Processes". *Proc. 3rd ACM Sigplan PPOPP*, Williamsburg, VA, April, 1991,155-166.

Baker, H.G. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978),565-569.

Baker, H.G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Progr.*, June 1990,218-226.

Baker, H.G. "The Nimble Type Inferencer for Common Lisp-84". Submitted to *ACM TOPLAS*, 1990.

Baker, H.G. "CONS Should not CONS its Arguments, or, A Lazy Alloc is a Smart Alloc". *ACM Sigplan Not. 27*,3 (March 1992),24-34.

Baker, H.G. "Equal Rights for Functional Objects". ACM *OOPS Messenger*, 1992, to appear.

Baker, H.G. "Cache-Conscious Copying Collectors". *OOPSLA'91 GC Workshop*, Oct. 1991.

Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Not.*, 1992, to appear.

Balzer, R.M. "EXDAMS: Extendable Debugging and Monitoring System". *Proc. AFIPS 1969 SJCC 34*, AFIPS Press, Montvale, NJ,567-580.

Barghouti, N.S. & Kaiser, G.E. "Concurrency Control in Advanced Database Applications". *ACM Comput. Surv. 23*,3 (Sept. 1991),269-317.

Barth, J. "Shifting garbage collection overhead to compile time". *CACM 20*,7 (July 1977),513-518.

Barth, Paul S., *et al.* "M-Structures: Extending a Parallel, Non-strict, Functional Language with State". *Proc. Funct. Progr. Langs. & Computer Arch.*, LNCS 523, Springer-Verlag, Aug. 1991,538-568.

Barton, Ed. Conservative Logic. 6.895 Term Paper, MIT, May, 1978.

Bawden, Alan. "Connection Graphs". *Proc. ACM Conf. Lisp & Funct. Progr.*, Camb., MA, Aug. 1986.

Beeler, M., Gosper, R.W, and Schroeppel, R. "HAKMEM". AI Memo 239, MIT AI Lab., Feb. 1972. Important items: 102, 103, 104, 149, 150, 161, 166, 172.

Benioff, Paul. "Quantum Mechanical Hamiltonian Models of Discrete Processes that Erase Their Own Histories: Application to Turing Machines". *Int'l. J. Theor. Phys. 21* (1982),177-201.

Bennett, Charles. "Logical Reversibility of Computation". *IBM J. Res. Develop. 6* (1973),525-532.

Bennett, Charles. "Thermodynamics of Computation". *Int'l. J. Theor. Phys. 21* (1982),905-940.

Bennett, Charles. "Notes on the History of Reversible Computation". *IBM J. Res. Develop. 32*,1 (1988),16-23.

Bennett, Charles. "Time/Space Trade-offs for Reversible Computation". *SIAM J. Computing 18*,4 (Aug. 1989).

Berry, G., and Boudol, G. "The Chemical Abstract Machine". *ACM POPL 17*, San Francisco, CA, Jan. 1990.

Chase, David. "Garbage Collection and Other Optimizations". PhD Thesis, Rice U., Nov. 1987.

Chen, W., and Udding, J.T. "Program Inversion: More than Fun!". *Sci. of Computer Progr. 15* (1990),1-13.

Chen, W. "A formal approach to program inversion". *Proc. ACM 18th Comp. Sci. Conf.*, Feb., 1990,398-403.

Cheney, C.J. "A Nonrecursive List Compacting Algorithm". *CACM 13*,11 (Nov. 1970),677-678.

Clarke, E.M. "Synthesis of resource invariants for concurrent programs". *ACM TOPLAS 2*,3 (July 1980).

Cohen, Jacques. "Non-Deterministic Algorithms". *Comput. Surveys 11*,2 (June 1979),79-94.

Corman, T.H., *et al. Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

Cousot, P., and Cousot, R. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". *Proc. ACM POPL 4* (1977),238-252.

Coveney, P.V. & Marcer, P.J. "Irreversibility and computation". *Specs. in Sci. & Tech. 14*,1 (1991?),51-55.

Deutsch, D. "Quantum Theory, the Church-Turing Hypothesis, and Universal Quantum Computers". *Proc. Roy. Soc.* (1985).

Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

Dobkin, D.P., and Munro, J.I. "Efficient Uses of the Past". *Proc. ACM FOCS 21* (1980),200-206.

Drescher, G.L. "Demystifying Quantum Mechanics: A Simple Universe with Quantum Uncertainty". *Complex Sys. 5* (1991),207-237.

Feldman, S., and Brown, C. "IGOR: A System for Program Debugging via Reversible Execution". *Proc. Sigplan/Sigops WS on Parl & Distr. Debugging*, May 1988,112-123.

Feynman, Richard P., *et al. The Feynman Lectures on Physics, Vol. I.* Addison-Wesley, Reading, MA, 1963.

Feynman, Richard P. "Quantum Mechanical Computers". *Founds. of Physics 16*,6 (1986),507-531.

Fisher, J. "Trace scheduling: A technique for global microcode compaction". *IEEE Tr.. Comps. C-30*,7 (July 1981),478-490.

Floyd, R.W. "Nondeterministic Algorithms". *J. ACM 14*,4 (Oct. 1967),636-644.

Fredkin, E., and Toffoli, T. "Conservative Logic". *Int'l. J. Theor. Physics 21*,3/4 (1982),219-253.

Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.

Grishman, R. "The debugging system AIDS". *AFIPS 1970 SJCC 41*, AFIPS Press, Montvale, NJ 1193-1202.

Halpern, J.Y., *et al.* "The Semantics of Local Storage, or What Makes the Free-List Free?". *ACM POPL 11*, 1984,245-257.

Harel, David. *First Order Dynamic Logic.* Springer-Verlag LNCS 68, 1979.

Harms, D.E., and Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Trans. SW Engrg. 17*,5 (May 1991),424-435.

Harrison, P.G. "Function Inversion". In Jones, N., *et al, eds. Proc. Workshop on Partial Evaluation and Mixed Computation*, Gammel Avernaes, Denmark, Oct. 1987, North-Holland, 1988.

Hederman, Lucy. "Compile Time Garbage Collection". MS Thesis, Rice U. Comp. Sci. Dept., Sept. 1988.

Heering, J., and Klint, P. "Towards monolingual programming environments". *ACM TOPLAS 7*,2 (April 1985),183-213.

Herlihy, Maurice. "Wait-Free Synchronization". *ACM TOPLAS 11*,1 (Jan. 1991),124-149.

Hofstadter, Douglas R. *Gödel, Escher, Bach: an Eternal Golden Braid*. Vintage Bks., Random House, NY, 1979.

Inoue, K., *et al.* "Analysis of functional programs to detect run-time garbage cells". *ACM TOPLAS 10*,4 (Oct. 1988),555-578.

Johnsson, T. "Lambda lifting: transforming programs to recursive equations". *Proc. FPCA*, Nancy, France, Springer LNCS 201, 1985,190-203.

Kay, A.C. "Microelectronics and the Personal Computer". *Sci. Amer. 237*,3 (Sept. 1977),230-244.

Keller, Robert M., *et al.* "An Architecture for a Loosely-Coupled Parallel Processor". Tech. Rep. UUCS-78-105, Oct. 1978,50p.

Kieburtz, Richard B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976),95-107.

Kieburtz, R. B. "The G-machine: a fast, graph-reduction evaluator". *Proc. IFIP FPCA*, Nancy, France, 1985.

Kieburtz, Richard B. "A RISC Architecture for Symbolic Computation". *Proc. ASPLOS II, Sigplan Not. 22*,10 (Oct. 1987),146-155.

Korth, H.F., *et al.* "Formal approach to recovery by compensating transactions". *Proc. 16th Int'l. Conf. on Very Large Databases*, 1990.

Kung, H.T. & Robinson, J.T. "On optimistic methods for concurrency control". *ACM Trans. on DB Sys. 6*,2 (June 1981).

Jefferson, David R. "Virtual Time". *ACM TOPLAS 7*,3 (July 1985),404-425.

Lafont, Yves. "The Linear Abstract Machine". *Theor. Computer Sci. 59* (1988),157-180.

Lafont, Yves. "Interaction Nets". *ACM POPL 17*, San Franciso, CA, Jan. 1990,95-108.

Lafont, Yves. "The Paradigm of Interaction (Short Version)". Unpubl. manuscript, July 12, 1991, 18p.

Lafora, F, & Soffa, M.L. "Reverse Execution in a Generalized Control Regime". *Comp. Lang. 9*,3/4 (1984), 183-192.

Landauer, R. "Dissipation and Noise Immunity in Computation and Communication". *Nature 335* (Oct. 1988),779-784.

LeBlanc, T.J., and Mellor-Crummey, J.M. "Debugging parallel programs with Instant Replay". *IEEE Tr. Comp. 36*,4 (April 1987),471-482.

Leeman, G.B. "Building undo/redo operations into the C language". *Proc. IEEE 15th Annual Int'l. Symp. on Fault-Tolerant Computing*, 1985,410-415.

Leeman, G.B. "A Formal Approach to Undo Operations in Programming Languages". *ACM TOPLAS 8*,1 (Jan. 1986),50-87.

Levy, E., *et al.* "An Optimistic Commit Protocol for Distributed Transaction Management". *Proc. ACM SIGMOD*, Denver, CO, May 1991,88-97.

Lewis, H.R., & Papadimitriou, C.H. "Symmetric Space-bounded Computation". *Theor. Comp. Sci. 19* (1982),161-187.

Lieberman, H., & Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*, 6 (June 1983),419-429.

Lindstrom, Gary. "Efficiency in Nondeterministic Control through Non-Forgetful Backtracking". Tech. Rep. UUCS-77-114, Oct. 1977,18p.

MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.

Manthey, M.J., & Moret, B.M.E. "The Computational Metaphor and Quantum Physics". *CACM 26*,2 (Feb. 1983),137-145.

Margolus, Norman. "Physics-Like Models of Computation". Elsevier North-Holland, *Physica 10D* (1984),81-95.

Margolus, Norman H. *Physics and Computation*. Ph.D. Thesis, MIT/LCS/TR-415, March 1988,188p.

Mattson, R.L., *et al.* "Evaluation Techniques for Storage Hierarchies". *IBM Sys. J. 9*,2 (1970),78-117.

McCarthy, John. "The Inversion of Functions defined by Turing Machines". In Shannon, C.E., and McCarthy, J., *eds. Automata Studies*, Princeton, 1956,177-181.

McDowell, C.E. & Helmbold, D.P. "Debugging concurrent programs". *ACM Comput. Surv. 21*,4 (Dec. 1989),593-622.

Miller, B.P. & Choi, J.-D. "A mechanism for efficient debugging of parallel programs". *Proc. ACM PLDI*, 1988,135-144.

Morita, K. "A Simple Construction Method of a Reversible Finite Automaton out of Fredkin Gates, and its Related Problem". *Trans. IEICE E 73*, 6 (1990),978-984.

Nylin, W.C.Jr., and Harvill, J.B. "Multiple Tense Computer Programming". *Sigplan Not. 11*,12 (Dec. 1976),74-93.

Pan, D.Z., and Linton, M.A. "Supporting reverse execution of parallel programs". *Proc. ACM Sigplan/Sigops WS on Par. & Distr. Debugging*, May 1988,112-123.

Penrose, R. *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*. Penguin Bks, London, 1989.

Peyton-Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice-Hall, NY, 1987.

Planck, Max. *Treatise on Thermodynamics*. Transl. Ogg, A., Dover Publ., NY, 1945.

Ressler, A.L. *The Design of a Conservative Logic Computer and a Graphical Editor Simulator*. M.S. Th., MIT, 1981, 128p.

de Roever, Willem P. "On Backtracking and Greatest Fixpoints". In Neuhold, Erich J., *Ed. Formal Description of Programming Concepts*, North-Holland, Amsterdam, 1978.

Romanenko, Alexander. "Inversion and Metacomputation". *ACM PEPM'91*, New Haven, CT, June 1991,12-22.

Rosenschein, Stanley J. "Plan Synthesis: A Logical Perspective". *Proc. IJCAI-81*, Vancouver, Canada, Aug. 1981, 331-337.

Ruggieri, C. & Murtagh, T. P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*,285-293.

Schorr, H., & Waite, W.M. "An efficient machine-independent procedure for garbage collection in various list structures". *CACM 10*,8 (Aug. 1967),501-506.

Shoman, Y., and McDermott, D.V. "Directed Relations and Inversion of Prolog Programs". *Proc. Conf. of 5th Gen. Comp. Sys.*, ICOT, 1984.

Sleator, D.D. & Tarjan, R.E. "Amortized Efficiency of List Update and Paging Rules". *CACM 28*,2 (Feb. 1985),202-208.

Smith, J.M., and Maguire, G.Q., Jr. "Transparent concurrent execution of mutually exclusive alternatives". *Proc. 9th Int'l. Conf. on Distr. Computer Sys.*, Newport Bch., CA, June 1989.

Strom, R.E., *et al.* "A recoverable object store". IBM Watson Research Ctr., 1988.

Strothotte, T.W., and Cormack, G.V. "Structured Program Lookahead". *Comput. Lang. 12*,2 (1987),95-108.

Suzuki, N. "Analysis of Pointer 'Rotation'". *CACM 25*,5 (May 1982)330-335.

Toffoli, T. "Reversible Computing". MIT/LCS/TM-151, Feb. 1980, 36p.

Toffoli, T. "Reversible Computing". In De Bakker & van Leeuwen, *eds. Automata, Languages and Programming*, Springer-Verlag (1980),632-644.

Toffoli, T. "Bicontinuous Extensions of Invertible Combinatorial Functions". *Math. Sys. Theor. 14* (1981),13-23.

Toffoli, T. "Physics and Computation". *Int'l. J. Theor. Phys. 21*, 3/4 (1982),165-175.

Tolmach, A.P., and Appel, A.W. "Debugging Standard ML without Reverse Engineering". *Proc. ACM Lisp & Funct. Progr. Conf.*, Nice, France, June 1990,1-12.

Turner, D. "A New Implementation Technique for Applicative Languages". *SW—Pract.&Exper. 9* (1979),31-49.

Vitter, J.S. "US&R: a new framework for redoing". *ACM Symp. on Pract. SW Dev. Envs.*, Pitts., PA, April 1984,168-176.

Wadler, P. "Views: A way for pattern matching to cohabit with data abstraction". *ACM POPL 14* (1987),307-313.

Wadler, P. "Is there a use for linear logic?". *Proc. ACM PEPM'91*, New Haven, June, 1991,255-273.

Wakeling, D. & Runciman, C. "Linearity and Laziness". *Proc. Funct. Progr. & Comp. Arch.*, Springer LNCS 523, 1991,215-240.

Wilson, P.R. & Moher, T.G. "Demonic memory for process histories". *Proc. Sigplan PLDI*, June 1989.

Zelkowitz, M.V. "Reversible Execution". *CACM 16*,9 (Sept. 1973),566-566.

Zurek, W.H., *ed. Complexity, Entropy and the Physics of Information*. Addison-Wesley, Redwood City, 1990.

Author Index

Abdullahi, Saleh E. 43
Agha, Gul 134
Baker, Henry G. 507
Bekkers, Yves 82
Berthomieu, Bernard 179
Chailloux, Emmanuel 218
Delacour, Vincent 426
Demoen, Bart 454
Duvvuru, Sreeram 264
Edelson, Daniel R. 299
Haines, Nicholas 357
Hansen, Lars 264
Hayes, Barry 277
Heck, Brian C. 248
Hudson, Richard L. 388
Jul, Eric 103
Juul, Niels Christian 103
Kolodner, Elliot K. 365
Lam, Michael S. 404
Langendoen, Koen G. 165
Le Sergent, Thierry 179
Mateu, Luis 230
Merrall, Simon C. 473
Miranda, Eliot E. 43
Moher, Thomas G. 404
Moss, J. Eliot B. 388
Muller, Henk L. 165
Nettles, Scott 357
O'Toole, James 357
Padget, Julian A. 473
Pierce, David 357
Pique, Jean-François 330
Plainfosse, David 116
Puaut, Isabelle 148
Ridoux, Olivier 82
Ringwood, Graem A. 43
Röjemo, Niklas 440
Samples, A. Dain 315
Sastry, A.V.S. 264
Seward, Julian 200
Shapiro, Marc 116
Sundararajan, Renga 264
Talcott, Carolyn 134
Tarau, Paul 344
Tick, Evan 264
Ungaro, Lucien 82
Venkatasubramanian, Nalini 134
Vree, Wim G. 165
Weemeeuw, Patrick 454
Weihl, William E. 365
Wilson, Paul R. 1, 404
Wise, David S. 248
Yuasa, Taiichi 490
Zhong, Xiaoxing 264

Lecture Notes in Computer Science

For information about Vols. 1–549
please contact your bookseller or Springer-Verlag

Vol. 550: A. van Lamsweerde, A. Fugetta (Eds.), ESEC '91. Proceedings, 1991. XII, 515 pages. 1991.

Vol. 551:S. Prehn, W. J. Toetenel (Eds.), VDM '91. Formal Software Development Methods. Volume 1. Proceedings, 1991. XIII, 699 pages. 1991.

Vol. 552: S. Prehn, W. J. Toetenel (Eds.), VDM '91. Formal Software Development Methods. Volume 2. Proceedings, 1991. XIV, 430 pages. 1991.

Vol. 553: H. Bieri, H. Noltemeier (Eds.), Computational Geometry - Methods, Algorithms and Applications '91. Proceedings, 1991. VIII, 320 pages. 1991.

Vol. 554: G. Grahne, The Problem of Incomplete Information in Relational Databases. VIII, 156 pages. 1991.

Vol. 555: H. Maurer (Ed.), New Results and New Trends in Computer Science. Proceedings, 1991. VIII, 403 pages. 1991.

Vol. 556: J.-M. Jacquet, Conclog: A Methodological Approach to Concurrent Logic Programming. XII, 781 pages. 1991.

Vol. 557: W. L. Hsu, R. C. T. Lee (Eds.), ISA '91 Algorithms. Proceedings, 1991. X, 396 pages. 1991.

Vol. 558: J. Hooman, Specification and Compositional Verification of Real-Time Systems. VIII, 235 pages. 1991.

Vol. 559: G. Butler, Fundamental Algorithms for Permutation Groups. XII, 238 pages. 1991.

Vol. 560: S. Biswas, K. V. Nori (Eds.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1991. X, 420 pages. 1991.

Vol. 561: C. Ding, G. Xiao, W. Shan, The Stability Theory of Stream Ciphers. IX, 187 pages. 1991.

Vol. 562: R. Breu, Algebraic Specification Techniques in Object Oriented Programming Environments. XI, 228 pages. 1991.

Vol. 563: A. Karshmer, J. Nehmer (Eds.), Operating Systems of the 90s and Beyond. Proceedings, 1991. X, 285 pages. 1991.

Vol. 564: I. Herman, The Use of Projective Geometry in Computer Graphics. VIII, 146 pages. 1992.

Vol. 565: J. D. Becker, I. Eisele, F. W. Mündemann (Eds.), Parallelism, Learning, Evolution. Proceedings, 1989. VIII, 525 pages. 1991. (Subseries LNAI).

Vol. 566: C. Delobel, M. Kifer, Y. Masunaga (Eds.), Deductive and Object-Oriented Databases. Proceedings, 1991. XV, 581 pages. 1991.

Vol. 567: H. Boley, M. M. Richter (Eds.), Processing Declarative Kowledge. Proceedings, 1991. XII, 427 pages. 1991. (Subseries LNAI).

Vol. 568: H.-J. Bürckert, A Resolution Principle for a Logic with Restricted Quantifiers. X, 116 pages. 1991. (Subseries LNAI).

Vol. 569: A. Beaumont, G. Gupta (Eds.), Parallel Execution of Logic Programs. Proceedings, 1991. VII, 195 pages. 1991.

Vol. 570: R. Berghammer, G. Schmidt (Eds.), Graph-Theoretic Concepts in Computer Science. Proceedings, 1991. VIII, 253 pages. 1992.

Vol. 571: J. Vytopil (Ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems. Proceedings, 1992. IX, 620 pages. 1991.

Vol. 572: K. U. Schulz (Ed.), Word Equations and Related Topics. Proceedings, 1990. VII, 256 pages. 1992.

Vol. 573: G. Cohen, S. N. Litsyn, A. Lobstein, G. Zémor (Eds.), Algebraic Coding. Proceedings, 1991. X, 158 pages. 1992.

Vol. 574: J. P. Banâtre, D. Le Métayer (Eds.), Research Directions in High-Level Parallel Programming Languages. Proceedings, 1991. VIII, 387 pages. 1992.

Vol. 575: K. G. Larsen, A. Skou (Eds.), Computer Aided Verification. Proceedings, 1991. X, 487 pages. 1992.

Vol. 576: J. Feigenbaum (Ed.), Advances in Cryptology - CRYPTO '91. Proceedings. X, 485 pages. 1992.

Vol. 577: A. Finkel, M. Jantzen (Eds.), STACS 92. Proceedings, 1992. XIV, 621 pages. 1992.

Vol. 578: Th. Beth, M. Frisch, G. J. Simmons (Eds.), Public-Key Cryptography: State of the Art and Future Directions. XI, 97 pages. 1992.

Vol. 579: S. Toueg, P. G. Spirakis, L. Kirousis (Eds.), Distributed Algorithms. Proceedings, 1991. X, 319 pages. 1992.

Vol. 580: A. Pirotte, C. Delobel, G. Gottlob (Eds.), Advances in Database Technology – EDBT '92. Proceedings. XII, 551 pages. 1992.

Vol. 581: J.-C. Raoult (Ed.), CAAP '92. Proceedings. VIII, 361 pages. 1992.

Vol. 582: B. Krieg-Brückner (Ed.), ESOP '92. Proceedings. VIII, 491 pages. 1992.

Vol. 583: I. Simon (Ed.), LATIN '92. Proceedings. IX, 545 pages. 1992.

Vol. 584: R. E. Zippel (Ed.), Computer Algebra and Parallelism. Proceedings, 1990. IX, 114 pages. 1992.

Vol. 585: F. Pichler, R. Moreno Díaz (Eds.), Computer Aided System Theory – EUROCAST '91. Proceedings. X, 761 pages. 1992.

Vol. 586: A. Cheese, Parallel Execution of Parlog. IX, 184 pages. 1992.

Vol. 587: R. Dale, E. Hovy, D. Rösner, O. Stock (Eds.), Aspects of Automated Natural Language Generation. Proceedings, 1992. VIII, 311 pages. 1992. (Subseries LNAI).

Vol. 588: G. Sandini (Ed.), Computer Vision – ECCV '92. Proceedings. XV, 909 pages. 1992.

Vol. 589: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1991. IX, 419 pages. 1992.

Vol. 590: B. Fronhöfer, G. Wrightson (Eds.), Parallelization in Inference Systems. Proceedings, 1990. VIII, 372 pages. 1992. (Subseries LNAI).

Vol. 591: H. P. Zima (Ed.), Parallel Computation. Proceedings, 1991. IX, 451 pages. 1992.

Vol. 592: A. Voronkov (Ed.), Logic Programming. Proceedings, 1991. IX, 514 pages. 1992. (Subseries LNAI).

Vol. 593: P. Loucopoulos (Ed.), Advanced Information Systems Engineering. Proceedings. XI, 650 pages. 1992.

Vol. 594: B. Monien, Th. Ottmann (Eds.), Data Structures and Efficient Algorithms. VIII, 389 pages. 1992.

Vol. 595: M. Levene, The Nested Universal Relation Database Model. X, 177 pages. 1992.

Vol. 596: L.-H. Eriksson, L. Hallnäs, P. Schroeder-Heister (Eds.), Extensions of Logic Programming. Proceedings, 1991. VII, 369 pages. 1992. (Subseries LNAI).

Vol. 597: H. W. Guesgen, J. Hertzberg, A Perspective of Constraint-Based Reasoning. VIII, 123 pages. 1992. (Subseries LNAI).

Vol. 598: S. Brookes, M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics. Proceedings, 1991. VIII, 506 pages. 1992.

Vol. 599: Th. Wetter, K.-D. Althoff, J. Boose, B. R. Gaines, M. Linster, F. Schmalhofer (Eds.), Current Developments in Knowledge Acquisition - EKAW '92. Proceedings. XIII, 444 pages. 1992. (Subseries LNAI).

Vol. 600: J. W. de Bakker, C. Huizing, W. P. de Roever, G. Rozenberg (Eds.), Real-Time: Theory in Practice. Proceedings, 1991. VIII, 723 pages. 1992.

Vol. 601: D. Dolev, Z. Galil, M. Rodeh (Eds.), Theory of Computing and Systems. Proceedings, 1992. VIII, 220 pages. 1992.

Vol. 602: I. Tomek (Ed.), Computer Assisted Learning. Proceedings, 1992. X, 615 pages. 1992.

Vol. 603: J. van Katwijk (Ed.), Ada: Moving Towards 2000. Proceedings, 1992. VIII, 324 pages. 1992.

Vol. 604: F. Belli, F.-J. Radermacher (Eds.), Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. Proceedings, 1992. XV, 702 pages. 1992. (Subseries LNAI).

Vol. 605: D. Etiemble, J.-C. Syre (Eds.), PARLE '92. Parallel Architectures and Languages Europe. Proceedings, 1992. XVII, 984 pages. 1992.

Vol. 606: D. E. Knuth, Axioms and Hulls. IX, 109 pages. 1992.

Vol. 607: D. Kapur (Ed.), Automated Deduction – CADE-11. Proceedings, 1992. XV, 793 pages. 1992. (Subseries LNAI).

Vol. 608: C. Frasson, G. Gauthier, G. I. McCalla (Eds.), Intelligent Tutoring Systems. Proceedings, 1992. XIV, 686 pages. 1992.

Vol. 609: G. Rozenberg (Ed.), Advances in Petri Nets 1992. VIII, 472 pages. 1992.

Vol. 610: F. von Martial, Coordinating Plans of Autonomous Agents. XII, 246 pages. 1992. (Subseries LNAI).

Vol. 611: M. P. Papazoglou, J. Zeleznikow (Eds.), The Next Generation of Information Systems: From Data to Knowledge. VIII, 310 pages. 1992. (Subseries LNAI).

Vol. 612: M. Tokoro, O. Nierstrasz, P. Wegner (Eds.), Object-Based Concurrent Computing. Proceedings, 1991. X, 265 pages. 1992.

Vol. 613: J. P. Myers, Jr., M. J. O'Donnell (Eds.), Constructivity in Computer Science. Proceedings, 1991. X, 247 pages. 1992.

Vol. 614: R. G. Herrtwich (Ed.), Network and Operating System Support for Digital Audio and Video. Proceedings, 1991. XII, 403 pages. 1992.

Vol. 615: O. Lehrmann Madsen (Ed.), ECOOP '92. European Conference on Object Oriented Programming. Proceedings. X, 426 pages. 1992.

Vol. 616: K. Jensen (Ed.), Application and Theory of Petri Nets 1992. Proceedings, 1992. VIII, 398 pages. 1992.

Vol. 617: V. Mařík, O. Štěpánková, R. Trappl (Eds.), Advanced Topics in Artificial Intelligence. Proceedings, 1992. IX, 484 pages. 1992. (Subseries LNAI).

Vol. 618: P. M. D. Gray, R. J. Lucas (Eds.), Advanced Database Systems. Proceedings, 1992. X, 260 pages. 1992.

Vol. 619: D. Pearce, H. Wansing (Eds.), Nonclassical Logics and Information Proceedings. Proceedings, 1990. VII, 171 pages. 1992. (Subseries LNAI).

Vol. 620: A. Nerode, M. Taitslin (Eds.), Logical Foundations of Computer Science – Tver '92. Proceedings. IX, 514 pages. 1992.

Vol. 621: O. Nurmi, E. Ukkonen (Eds.), Algorithm Theory – SWAT '92. Proceedings. VIII, 434 pages. 1992.

Vol. 622: F. Schmalhofer, G. Strube, Th. Wetter (Eds.), Contemporary Knowledge Engineering and Cognition. Proceedings, 1991. XII, 258 pages. 1992. (Subseries LNAI).

Vol. 623: W. Kuich (Ed.), Automata, Languages and Programming. Proceedings, 1992. XII, 721 pages. 1992.

Vol. 624: A. Voronkov (Ed.), Logic Programming and Automated Reasoning. Proceedings, 1992. XIV, 509 pages. 1992. (Subseries LNAI).

Vol. 625: W. Vogler, Modular Construction and Partial Order Semantics of Petri Nets. IX, 252 pages. 1992.

Vol. 626: E. Börger, G. Jäger, H. Kleine Büning, M. M. Richter (Eds.), Computer Science Logic. Proceedings, 1991. VIII, 428 pages. 1992.

Vol. 628: G. Vosselman, Relational Matching. IX, 190 pages. 1992.

Vol. 629: I. M. Havel, V. Koubek (Eds.), Mathematical Foundations of Computer Science 1992. Proceedings. IX, 521 pages. 1992.

Vol. 630: W. R. Cleaveland (Ed.), CONCUR '92. Proceedings. X, 580 pages. 1992.

Vol. 631: M. Bruynooghe, M. Wirsing (Eds.), Programming Language Implementation and Logic Programming. Proceedings, 1992. XI, 492 pages. 1992.

Vol. 632: H. Kirchner, G. Levi (Eds.), Algebraic and Logic Programming. Proceedings, 1992. IX, 457 pages. 1992.

Vol. 633: D. Pearce, G. Wagner (Eds.), Logics in AI. Proceedings. VIII, 410 pages. 1992. (Subseries LNAI).

Vol. 634: L. Bougé, M. Cosnard, Y. Robert, D. Trystram (Eds.), Parallel Processing: CONPAR 92 – VAPP V. Proceedings. XVII, 853 pages. 1992.

Vol. 635: J. C. Derniame (Ed.), Software Process Technology. Proceedings, 1992. VIII, 253 pages. 1992.

Vol. 636: G. Comyn, N. E. Fuchs, M. J. Ratcliffe (Eds.), Logic Programming in Action. Proceedings, 1992. X, 324 pages. 1992. (Subseries LNAI).

Vol. 637: Y. Bekkers, J. Cohen (Eds.), Memory Management. Proceedings, 1992. XI, 525 pages. 1992.

Vol. 639: A. U. Frank, I. Campari, U. Formentini (Eds.), Theories and Methods of Spatio-Temporal Reasoning in Geographic Space. Proceedings, 1992. XI, 431 pages. 1992.

Vol. 640: C. Sledge (Ed.), Software Engineering Education. Proceedings, 1992. X, 451 pages. 1992.