SRC Technical Note

2000-006

November 2000

Selected 2000 SRC Summer Intern Reports

Compiled by

Cathy Miller, MLIS

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://www.research.compaq.com/SRC/

This document features informal reports by interns who spent the summer of 2000 working with researchers at Compaq Systems Research Center (SRC). The interns were graduate students in computer science or electrial engineering Ph.D. programs. Each worked for about three months at SRC, collaborating on a project with members of the research staff. The primary goal of this technical note is to describe the summer research projects. However, the interns were encouraged to write their reports in whatever format or style they preferred, so that non-technical observations (such as background and impressions arising from their stay) could also be included.

Cloudburst:
A Compressing, Log-Structured Logical Disk for Flash Memory

Gretta Bartels
University of Washington

1 Introduction

This summer, I worked with Tim Mann on building Cloudburst, a device driver that enables persistent file storage on handheld computers with flash memory, such as the Itsy. Cloudburst is a compressing, log-structured logical disk. Currently, handheld computers with flash generally have read-only file systems, or read-write file systems with no compression.

2 Flash Memory

Flash memory has characteristics that make it different from most other forms of persistent storage. To its advantage, reading and writing flash is nearly as fast as reading and writing DRAM. However, flash can not generally be rewritten. Each bit in a flash memory has an erase state -- say, 1. When the flash is erased, all of the bits in the flash are set to 1. Each individual bit may be flipped to 0 at any time. But to flip back from 0 to 1, an entire sector of the flash must be erased simultaneously. Sectors are large, often 256 KB or 512 KB, and it takes a long time to erase a sector, nearly a second.

For this reason, rewriting small blocks of data in-place on flash memory is very inefficient. For example, suppose we are trying to store an array of 1024 512-byte blocks in a sector of flash. Suppose we need to update one of the blocks, and that some of the bits in the block will flip from 0 to 1 when we do the update. To do the update, we'll need to:

1. Read the entire half-MB sector into memory from flash
2. Update the block in memory
3. Erase the sector
4. Rewrite the entire sector from memory to flash

Besides being time- and memory-consuming, this approach also leaves a sizable window of opportunity for data loss, should the system lose power. Furthermore, each sector of flash may only be erased about 100,000 times. After that, it no longer guarantees data integrity.

3 Cloudburst Rationale

We designed Cloudburst to be:

- log structured, because writing blocks in-place on flash is very inefficient
- compressing, because flash is generally quite small, and file systems expand to fill all available space
- a logical disk, because implementing complete file systems is tricky and unnecessary. By working at a lower level, we can support many different file systems.

The logical disk abstraction is a layer between the file system and the storage medium. To the file system, the logical (or virtual) disk acts like a normal, magnetic disk: it reads and writes blocks of a fixed size. However, under the covers, the logical disk can store the data on the storage medium however it chooses.

In the case of Cloudburst, we store the data in a log, writing blocks sequentially on the flash. Because there is no minimum write granularity on the flash, we can compress the blocks as we write them to the log without losing space due to fragmentation.

4 Related Work

Many different combinations of compression, log-structure, logical disks, and flash-friendliness have been tried before. To the best of our knowledge, the combination of all four has never been tried.

5 Acknowledgments

I would like to thank Tim Mann, my host, for being so helpful, available, and knowledgeable about the task at hand. Thanks also to Mike Burrows for explaining various compression algorithms and coming up with so many great ideas for Cloudburst's cleaner algorithms.

6 About the Author

I'm a third-year student in the PhD program in computer science at the University of Washington. Hank Levy is my advisor at UW. In my most recent project before coming to SRC, I designed and implemented a highly scalable failure detection and reporting protocol for large clusters in a LAN environment.

Towards the Vantage Project: Camera calibration and structure recovery from a single image

Jonathan Deutscher
Oxford University

The Vantage Project proposes to deploy many cameras around SRC and to track people as they move from the field of view of one camera to another. It should be possible to use this information to analyse the movements of people and identify individuals.

This tracking (in particular the hand-off between cameras as a person moves between their fields of view) will be easier if we can make inferences about the real location of a person in the world based on that person's position in a camera's image plane. It would also be helpful to know the 3D structure of the scene to guide the tracking and enforce the constraint that the person is walking on the floor, not up the wall.

The problem can be divided into three areas:

Camera Calibration

A video camera can be approximated as a projection from the 3D world onto a 2D image plane, and a calibrated camera is one for which that projection matrix is known. Current methods for automatic calibration and structure recovery require stereo images, hand-registration of features or the observation of known objects. We used a method that uses the Manhattan assumption (that most of the lines in a scene are aligned along three perpendicular axes) to automatically recover the camera calibration from a single image.

Image Segmentation

Once the camera has been calibrated we want to segment the pixels in the image into different regions that correspond to some kind of structure in the world. Using our Manhattan assumption we can assume that most of the surfaces in the scene are planar and that they are separated by extended lines in one of the three primary directions. Once the camera is calibrated we can detect these extented lines in the image and use them to define an initial set of regions. We then reduce this set by merging the most similar neighbouring regions until a minimum region difference is reached.

Structure Recovery

Once the image has been segmented we assume that each segment represents a planar surface in the world. We begin by heuristically identifying the floor region and assuming that the camera has been installed roughly upright we can compute the orientation of this floor region. We then compute the world coordinates of every pixel in the floor segment by performing simple line plane intersections. Using these world coordinates we can discern the boundaries of the floor which should assist greatly in tracking people for the Vantage Project.

Annotation Inference Techniques

Michael Y. Levin
University of Pennsylvania
milevin@cis.upenn.edu

Cormac Flanagan
Compaq SRC
cormac.flanagan@compaq.com

K. Rustan M. Leino
Compaq SRC
rustan.leino@compaq.com

1 Introduction

Houdini is a static program checking tool that helps uncover potential run-time errors in Java programs. Among other kinds of errors, it can detect array bound overflows, null dereferences, and division by zero. Houdini is based on an earlier program checker developed at SRC called ESC/Java. While ESC/Java expects the methods in its input programs to be annotated with a sufficient number of pre and post conditions, Houdini drops this requirement and attempts to perform useful analysis even when given unannotated input programs.

Initially, Houdini was prototyped using a shell script that iteratively modified the text of the input program and repeatedly called ESC/Java. This prototype was useful in verifying the Houdini approach, but was quite inefficient. Our project goals were to redesign Houdini's architecture to be more independent of ESC/Java and to develop Houdini-specific optimization techniques. The rest of this document describes the original implementation of Houdini and several optimizations introduced by our project.

2 Initial Houdini Prototype

The initial version of Houdini works in two stages. First, it guesses a set of preconditions and postconditions for every method in the input program and inserts them directly in the program. Then, it uses ESC/Java to figure out which of the guessed annotations are incorrect. ESC/Java works by converting each annotated method into a formula called verification condition and sending it to the theorem prover Simplify which, in turn, returns a list of incorrect annotations. Houdini then removes these incorrect annotations from the program. Since removing one annotation may cause subsequent annotations to become invalid, Houdini repeats this process until it settles on an appropriate set of annotations. At this point Houdini invokes ESC/Java one final time in order to check the program with respect to the inferred set of annotations. The diagram in figure 1 shows the architecture of this initial version of Houdini.

3 Guarded Verification Conditions

One inefficiency of the above approach is that it involves repeated generation of verification conditions from the same methods. Instead we can create a special kind of verification condition, called a *guarded verification condition (GVC)*, just once when Houdini is started. We introduce a unique guard variable for each annotation guessed by Houdini. The GVC of a method has the property that when we substitute true for a guard variable in the GVC, the resulting formula is equivalent to the verification condition produced ESC/Java when the corresponding annotation is present in the program. In addition, when we substitute

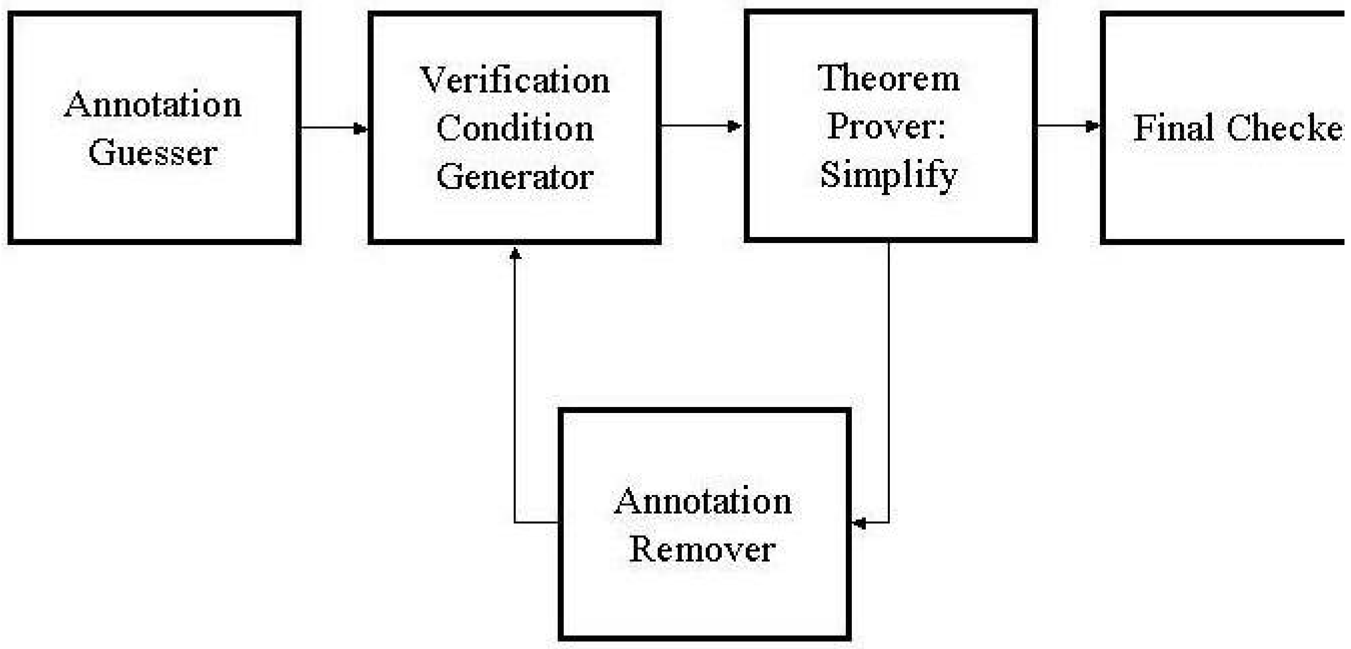

Figure 1: Original Houdini Architecture

false for a guard variable, the resulting formula is equivalent to the verification condition generated when the corresponding annotation is not present in the program.

To implement the above idea, we first modified ESC/Java to produce guarded verification condition. The new Houdini implementation maintains a truth value assignment for each guard variable depending on whether the corresponding annotations has been refuted so far. At each iteration, the implementation

- substitutes for the guard variables according to the truth assignment
- performs basic simplifications on the obtained formula
- and sends the optimized verification condition directly to Simplify

The new architecture uses ESC/Java only in its initial stage of generating GVCs and in the final stage of checking the program with respect to the inferred set of annotations. The rest of Houdini is autonomous. Eliminating the need in parsing Java methods and generating verification conditions repeatedly resulted in a substantial performance increase. The new design is described by the diagram in figure 2.

4 Optimizations

Our implementation of Houdini employs many other optimizations. A lot of them are too detailed to be introduced in this short document. We focus only on two main optimizations in this section.

4.1 Fine Grained Optimization

The basic simplifications mentioned in the previous section are local; they do not exploit the knowledge about verification conditions gained by Houdini over several iterations of substituting and sending them to Simplify. For example, suppose at some point after substituting for guards a verification condition has the

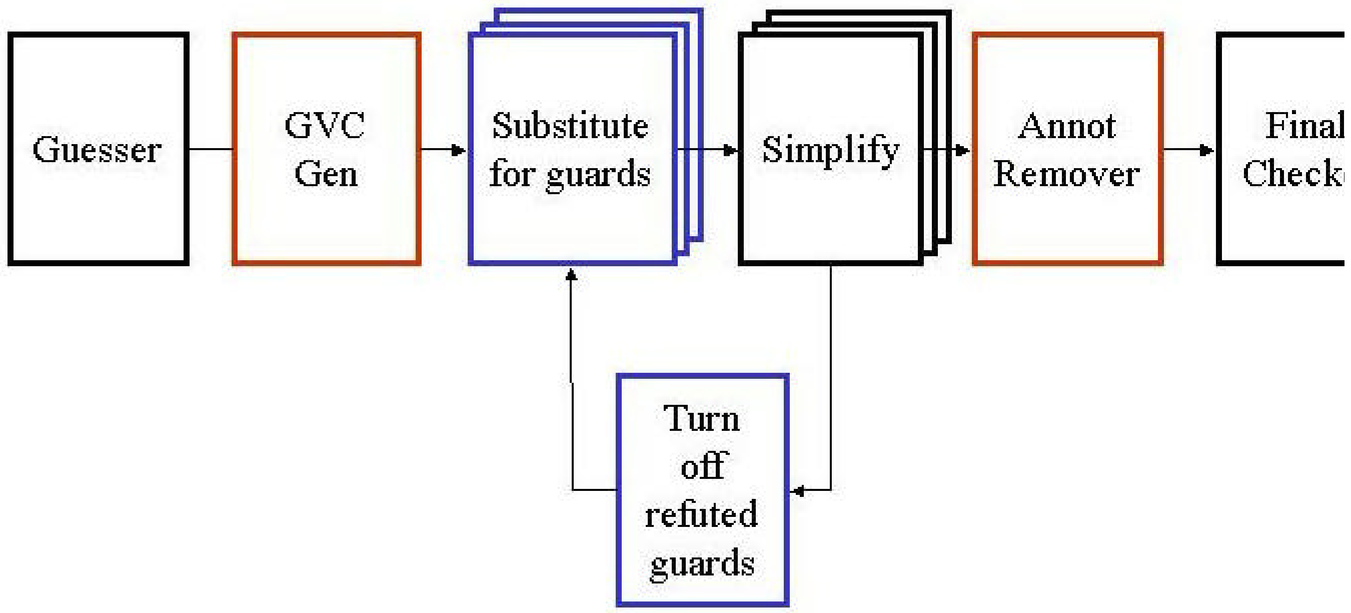

Figure 2: GVC Architecture

form P ^ Q and turns out to be valid. If at a later iteration after substitution, we find that the verification condition for the same method is of the form P ^ Q', we can simplify it to just Q' since we know that P is valid. *Fine grained optimization* is a generalization of this basic idea.

4.2 Distributed Implementation

Finally, we notice that Simplify is the performance bottleneck of the system and that running Simplify on a verification condition is self-contained. Hence, it is natural to distribute this task over multiple processes. We achieve such distribution by running a central controller and multiple worker processes on different processors and exchanging data between the controller and workers over sockets. The distributed architecture is shown in figure 3.

5 Conclusion

The initial Houdini version successfully inferred annotations for a 37,000 line program taking more than 60 hours. The new version incorporating the above as well as many other optimizations performed the same task in a little over 1 hour. We ran the experiment in a 12 worker configuration and the parallelization gave us a factor of 7 speedup. Guarded verification conditions accounted for a factor of 2 improvement. Surprisingly, fine grained optimization did not result in any measurable performance gain. The rest of the improvement came from additional optimizations not discussed here.

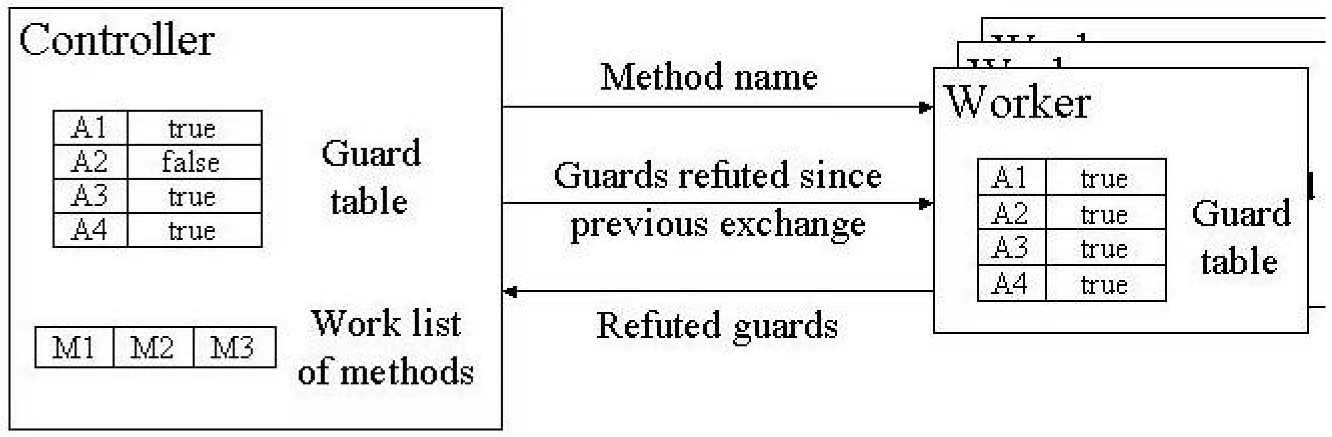

Figure 3 Distributed Architecture

Performance Evaluation of the Piranha Memory Hierarchy

Julien Sebot
Universite Paris-Sud

1 Introduction

This summer, I explored how to improve TLB performance and second level cache fill policy in Piranha. Piranha is an 8-CPU on-chip multiprocessor targeted at database applications. The goal for this project is to achieve 2 times the On-Line Transaction Processing (OLTP) performance in half the time and with one tenth of the engineering effort, when compared to contemporary processor design efforts.

The Piranha system will be fully synthetized in an ASIC process. The Piranha processing node will include 8 simple one-way alpha cores running at 400MHz, 32kB direct mapped first-level instruction and data caches, a shared 1MB 8-way set-associative second-level cache, 8-way set-associative, memory controllers, and an interconnect subsystem that connects processing nodes together.

2 Benchmarks

I used SimOS-Alpha and scaled-down TPC benchmarks for the performance evaluation. SimOS is a full-system simulation tool that models hardware in enough detail to boot an operating system. SimOS integrates several processor and memory systems simulators that I have used and improved. The TPC benchmarks we used are OLTP benchmarks called TPC-B and TPC-C. These are standard benchmarks used to model the activity of bank transactions and wholesale suppliers. For these programs, over 45% of the execution time is spent in the memory system in an architecture like Piranha.

3 Evaluation

One aspect of the memory system that has significant impact on performance is the address translation cache (TLB). The design constrains of the ASIC process to which Piranha is targeted prevents us from implementing a traditional fully-associative TLB. We have studied the impact of limited associativity on TLB performance, and concluuded that a 256-entry, 4-way set-associative TLB is 4% better than a 64-entry fully-associative TLB, for the scaled-down benchmarks at our disposal. This result is not definitive given that the TLB performance is affected by the scaled down nature of our benchmarks.

Another important area of system design that strongly effects memory system performance is the second-level cache. In Piranha, the combined first-level cache size is 512kB, and the second-level cache is 1MB. The Piranha team has chosen to implement a non-inclusive, shared, second-level cache (shared victim cache) to avoid wasting space in the second-level cache. We have determined that the performance impact of this choice on Piranha performance ranges between 5% to 9%, and that the performance gains over a standard inclusive policy becomes negligible for second-level cache sizes of 2MB or greater. The intuition behind these results is that, even when inclusion not enforced by hardware, in practice there will be many times in which a line will be present both in the L2 and one or many L1s. For larger caches, the penalty of enforcing inclusion decreases as the fraction of replicated (L1) data in the L2 is reduced.

In a non-inclusive cache hierarchy, the second-level cache is responsible for deciding when a L1 cache has to write back into the L2 (i.e., the L2 fill policy). We have evaluated the performance of Piranha's current fill policy with respect to two potentially "ideal" fill policies: one that is very eager and one that is as lazy as possible in sending write backs to the L2. Both eager and lazy policies are infeasible to implement since the amount of L2 state that needs to be inspected would cause extra delays in satisfying processor requests. Therefore the comparison aims only at determining how different our current (implementable) policy deviates from the ideal cases. The results show that the 3 policies never differ more than 3% in performance, which further corroborates the effectiveness of the current scheme being implemented in Piranha.

Cooperative Backup System

Sameh Elnikety
Rice University

1 Introduction

I am a second year Ph.D. student in the department of Computer Science at Rice University. I am working with Willy Zwaenepoel, my advisor, on distributed services. During my internship at SRC during summer 2000, I worked with Mark Lillibridge, my host, and Mike Burrows on building a prototype of a cooperative backup system.

2 Cooperative Backup System

In this system, each machine stores its backup data remotely among a group of other peer machines, and in return it stores equivalent amounts of data from its partners in its local file system. This form of cooperation and distribution gives several benefits and poses several interesting challenges. Let us first consider the benefits. As the partners are independent and geographically distributed, they have independent failure modes, which is important for a backup system and corresponds to taking the backup tapes off-site. In addition, this cooperation makes the system very cost effective as the partners do not have to pay a fee to a third party for the backup service and there is no need to purchase new equipment. As for the challenges in the design of the system, the partners do not trust each other and it is possible that some partners are down at any moment. Therefore, we had to use several techniques to ensure confidentiality, robustness, integrity, and cooperation.

As the backup data might be sensitive and is stored remotely, we used secret key cryptography to encrypt the data. In particular, we used IDEA to encrypt the backup data before sending it to the partners during a backup operation and to decrypt the data during a restore operation. To ensure that the partners could not modify the data unnoticed, we used a cryptographic hash function to make the data blocks self-checking. So, when a machine is retrieving its data during a restore operation, it could check the cryptographic hash values in the data blocks to make sure that it is the same data that was backed up during the last backup operation. We used erasure codes to add redundancy to the data, so it is possible to do a backup or a restore even if a few partners are down. The possible security attacks against the system were novel and guarding against them was the hardest design issue. We used challenges to ensure that the partners of a given machine are keeping its backup data. To challenge a partner, a machine requests a randomly chosen block of data from that partner and checks it is the right block. Also, we had to impose several rules to ensure the cooperation of the partners and to prevent any partner from gaining any benefit if it does not apply the rules. For example, to prevent an attacker from exploiting one machine after the other, we imposed a commitment cost whenever a machine acquires a new partner. The machine pays the commitment cost by being forced to store its partners' data for a certain period of time without any guarantee that it can restore its own data. We ended our study with profiling the prototype system. We found out that the costs of encryption, computing the secure hash, and applying erasure codes are much less than what we had expected.

3 SRC

I found SRC to be an exciting environment, full of very friendly, cooperative and brilliant people. The research taking place at SRC is very interesting and there is a prevailing mutual spirit of cooperation and integration among the different research groups. I had the chance to work closely with many smart people at SRC. Also, SRC is in downtown Palo Alto, which is a very lively place and I enjoyed a lot of fine restaurants and shopping places. In addition, I had the chance to visit other research labs in the Silicon Valley (e.g., Xerox PARC, HP Labs, IBM Almaden and Sun Labs).

Detecting and Correcting LAN Pathologies

Neil Spring
University of Washington

1 Introduction

Switched Ethernets are popular for their ability to isolate traffic between different pairs of hosts for performance and security. The increase in aggregate bandwidth allows switched networks to scale larger than broadcast networks using hubs. This larger scale makes switched networks more vulnerable to common network pathologies. The pathologies we address in this paper include:

1) broadcast storms
2) ARP fights

Both pathologies exist in traditional, shared broadcast media, but are more relevant in switched networks because of their large scale.

2 Pathologies

Broadcast storms occur when a buggy or malevolent host emits a continuous stream of broadcast packets. Broadcast packets cannot be switched and must traverse each link in the network. This allows a single host to execute a denial-of-service attack on all other hosts on the same subnet.

ARP fights occur when two hosts with different MAC (layer 2 hardware) addresses conflict for the same IP address. ARP is the protocol used to map IP addresses to MAC addresses for transport of IP traffic over a local network. It is not suited to resolving conflicting responses. This often happens due to misconfiguration or buggy DHCP implementations, and could be used by a misbehaving host as part of a man-in-the-middle attack.

We have developed a tool, named Vincent, to determine the source of broadcast storms and disable the offending network segment. The tool understands the switched network topology using a standard SNMP interface and minimal information. Another tool monitors the broadcast traffic associated with ARP requests, and verifies the stability and lack of conflict in the IP to MAC address mapping. Both tools alert the system administrator via electronic mail, and, in the case of broadcast storms, take action.

3 Topology Discovery

Understanding the switched network topology is important for two reasons. First, it helps in deciding which port to disable when misbehavior occurs. Second, it allows our tool to help in physically locating misbehaving hosts.

Our approach uses the forwarding database (FDB) of each switch. The FDB maps each MAC address to a switch output port, analogous to the forwarding table of an IP router. Since switch topology is restricted to that of a tree, either by physical connection or by use of the spanning tree protocol, topology discovery is the process of recovering that tree.

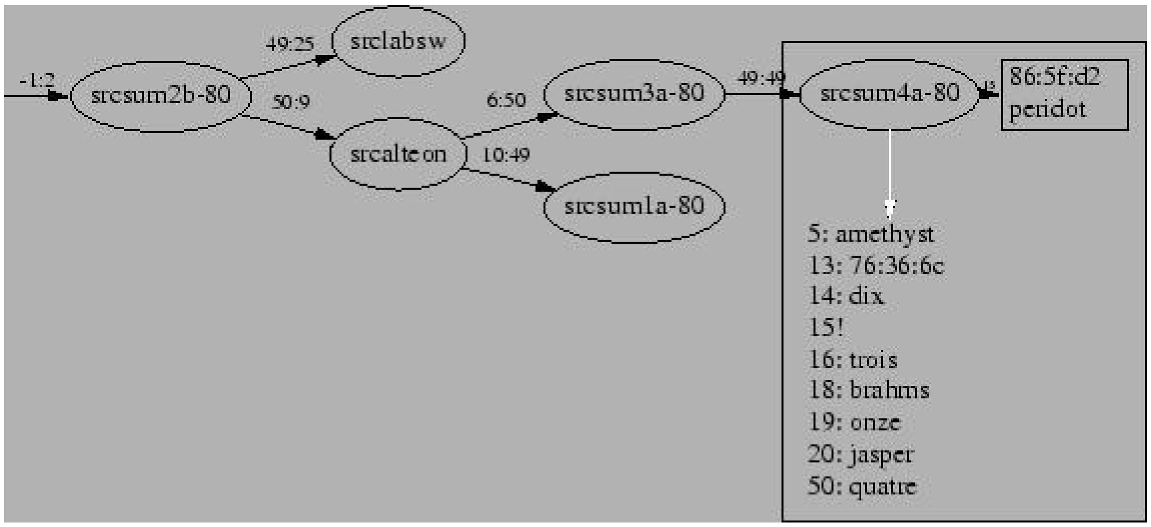

Figure 1: Sample, abridged, topology.

4 Broadcast Storm Detection

Broadcast storms are detected and localized in two ways. First, a packet sniffer in non-promiscuous mode can see all broadcast traffic. Seeing more than x packets in t seconds implies that a broadcast storm is underway. It is likely that the packet is correctly addressed, and that knowing the source mac address and the network topology will point to a particular switch port to be disabled.

If the packet sniffer fails at determining the source, possibly because of incorrectly formatted packets or because the misbehaving host has not been seen on the network before, the per-port broadcast ingress packet counters can be used to trace broadcast packets to their source. This is, however, a less timely detector, since retrieving these counters from the switch is a somewhat heavyweight operation, and thus cannot be executed often.

5 Broadcast Storm Resolution

After enough packets have been seen over a 1 second interval to warrant action, the source's port is disabled. The port will be reenabled after the passage of an interval, which doubles each time the source port is disabled.

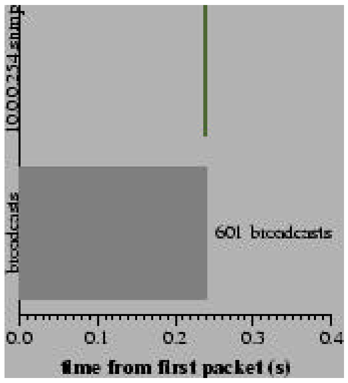

Figure 2: Shutoff of broadcast traffic

6 ARP Fight Detection

The packet sniffer watches for ARP requests, which are, by nature, broadcast to all hosts. Vincent enqueues both the source and destination IP addresses from the ARP request for verification. This queue is used to reduce the overall traffic load imposed on the network by Vincent.

Every second, Vincent chooses the next IP, and sends an ARP request. If two conflicting responses are received, then two machines have decided to use the same IP address. Vincent then notifies the administrator by electronic mail.

7 ARP Fight Resolution

It is possible for Vincent to choose one of the conflicting hosts to stay in the network and disable others. However, the best policy is not clear. Should the host having the IP address for the longest time be entitled to continue using it? Probably, but it might be an address allocated by DHCP, for which the host (for any number of reasons) has not properly renewed the lease. Without a better understanding of what the DHCP server intends, our response is limited to notifying the administrator.

There are some interesting possibilities in this domain.

First, the ISC DHCP server includes a flat text file containing its IP to MAC address mappings. Any that contradict this list would be disabled. Ideally, the DHCP protocol would include provision for such verification.

Second, some institutions keep a list of MAC addresses that are allowed to obtain an IP address via DHCP. This list includes the user who owns the machine, which would make it particularly easy to notify the parties involved.

8 Conclusion

Our tool, Vincent, is able to recover the topology of a switched ethernet using commonly available information from SNMP. This feature alone makes it particularly useful to network administrators interested in planning improvements to the network infrastructure.

With this understanding of host location, it becomes possible to disable the ports of these switches that connect to misbehaving hosts. By disconnecting these hosts, it is possible to preserve connectivity between correctly-behaving hosts. Vincent disables some misbehaving hosts, and reports misbehavior to the network administrator.

Link Compression in the Connectivity Server

Rajiv Wickremesinghe (Duke University)
with Raymie Stata, Janet Wiener, and others.
Compaq Systems Research Center
Summer, 2000.

1 Introduction

There is considerable interest in analyzing the manner in which web-pages are linked together. The link-structure of the web-graph is part of the page ranking criteria used by some popular web-search sites. It has also been used in research on the evolution and form of the web. The Connectivity Server (CS) provides fast, random access to this information. In order to accomplish this, it has to compress the graph information to fit in memory. The current version (CS2) can store link information for approximately 200 million pages (about a 3-week web crawl) in 8GB of RAM.

During the summer, we assessed different methods for improving the compression in CS2. Implementing several alternatives, we produced CS3, which improves compression by a factor of two, allowing us to double the number of links we are able to store in a given memory size.

2 The Connectivity Server

The CS densely allocates positive numeric IDs to all the web pages in the database. Link information is returned as lists of IDs forming an adjacency list for an ID being queried.ID order approximates alphabetical order of the URLs identifying pages, with the exception that pages are first separated into groups corresponding to having `many', `intermediate' or `few' links. This latter division ensures that well-referenced pages have small and proximate IDs (which gives better encoding), and also improves index compression (we are able to make assertions about the number of links of a page). CS2 also provides other indexing and lookup functionality which is not affected by the changes we made.

There are two models we can use to reduce the amount of data required to store the web-graph. The first model makes predictions about the set of links on a page based on other pages' links. This gives us inter-row compression. The second model makes predictions about individual links on a page, given other links on the same page. This gives us intra-row compression. One of the main contributions of CS3 is the addition of inter-row compression to CS2, which already included intra-row compresion. CS3 also improves the overall compression by using escaped Huffman codes to encode the data, in place of a length and value code.

3 Realization

We formulated several compression schemes based on different models and evaluated them by predicting the compression for some of our datasets. We chose the scheme sketched below because it seemed the most flexible (possible to adjust the speed vs. compression tradeoff easily) and efficient (fast).

For each set of links on a page to be compressed:

- Examine the links in a `collection' of different pages, and pick the one most similar to the current page. (see below).
- Encode a reference to that page, and the differences, using Huffman codes combined with a differential technique.

There are many ways of selecting a collection of pages as candidates for comparison, above. We observed that the pages in close proximity by ID form a good (and hard to beat) collection for this purpose. They can also be accessed effciently when scanning the database. ID order is effective because pages with similar URLs have many common links (eg: menubars, homepage and index links etc.). We found that, on average, half the links on a page were common to other pages with nearby IDs. Our goal, then, was to fully utilize this redundency to improve compression.

There are several parameters that can be easily changed in the above scheme. These include the size of the window to examine, and the length of chains to allow. Increasing the size of the window results in better compression, though with dimishing effect; we experimented with values in the range 2-32. Allowing longer chains improves compression, but adversely affects speed.

Depending on the database chosen, we obtained 44%-54% compression as compared to the original compression scheme in CS2. We also observed that the new compression does not cause any significant slowdown of the program. We are now able to store link information for approximately 400 million pages in 8 GB of RAM.

Last modified: Mon Jan 8 13:29:53 2001 by rajiv.

A Hardware Compiler for Data-streaming Reconfigurable Architectures

Grigorios Magklis
University of Rochester

1 Introduction

I'm currently working towards my Ph.D. at the University of Rochester, working with professor Michael Scott, my advisor. My current research is on Complexity Adaptive Processing. This summer I worked with Laurent Moll, on building a compiler for the Sepia board.

2 Sepia

Sepia consists of a reconfigurable device (Xilinx FPGA) that has a PCI connection to the host machine and a network interface to connect with other machines or Sepia boards. It is currently being used to merge partially rendered images to a final image. In the current configuration each board accepts two streams of data (from the local machine and from the network) representing parts of an image, and outputs a data steam that is a combination of the two inputs. The combining function can be anything from alpha-channel blending to z-buffer comparison.

3 Hardware Configuration

The process of configuring the h/w involves many steps. First the algorithm is described and tested in some high level language, like C. Then the appropriate hardware is designed by hand. After this the hardware is described in some hardware description language (HDL) and is tested again. Finally the FPGA code is compiled and "downloaded" to the board.

The goal of this project is to simplify this process by allowing the user to specify the algorithm in some high level language and have the final FPGA code generated automatically by the compiler.

4 The Compiler

Our compiler understands a very simplified C-like language. This language does not have features like pointers and arrays due to the problems they present with efficiently mapping them to hardware resources. It also does not have other features like complex types (`struct`, `union`) and support for functions (except for a `main` function), because there was not enough time to implement them. These features can be easily added in the future though. Loops are also not allowed (`for`, `while`, `do`) because of the nature of the computation: the hardware is operating on streams of data and has to generate a result on every clock cycle.

When designing the compiler we decided to follow traditional compiler techniques. The only difference is in the intermediate language instruction set. Instead of representing an ISA, it represents the available hardware resources. The first stages of the compiler are similar to a common C compiler. It first generates a parsing tree and then it builds the intermediate representation code (IR) in single static assignment form. The only interesting difference is that the generated code is one large basic block. This is possible because the IR contains a "select" instruction – similar to the "`?:`" C operator.

The next step is optimizing the IR. In this step we also follow traditional compiler techniques, only that now we are trying to optimize for code size, not speed. The optimizations performed, in order, are: copy and constant propagation, constant expression evaluation, expression simplification and unreachable code elimination.

The final step of the compilation is to produce location information for each primitive (IR instruction). This is the most important step since it is going to decide the performance, i.e. clock frequency, of the resulting hardware. Unfortunately I didn't have time to implement this step. The "scheduling problem" as it is called, is an NP-hard problem and so far people have used a number of heuristics to solve it. We are thinking to use the datapath information we get from the IR in order to generate placement information. This can be done by first identifying which IR instructions belong to each pipeline stage of the resulting hardware and then try to schedule (i.e. generate location information) for each stage separately. This has the property of reducing the problem dimensions from two to one.

Related field analysis in the Swift compiler

Aneesh Aggarwal
University of Maryland

I spent my internship at SRC working on aspects of the Swift compiler with Keith Randall. A paper on this work is forthcoming. Here is the abstract from that paper:

We present an extension of field analysis called "related field analysis" which is a general technique for proving relationships between two or more fields of an object. We demonstrate the feasibility and applicability of related field analysis by applying it to the problem of removing array bounds checks. For array bounds check removal, we define a pair of related fields to be an integer field and an array field for which the integer field has a known relationship to the length of the array. This related field information can then be used to remove array bounds checks from accesses to the array field. Our results show that related field analysis can remove an average of 50% of the dynamic array bounds checks on a wide range of applications.

We describe the implementation of related field analysis in the Swift optimizing compiler for Java, as well as the optimizations that exploit the results of related field analysis.

Mining the Web for Site Structure

Chris Homan
Rochester University

September 1, 2000

1 Overview

Our goal was to discover interesting structural properties on the Web, using a Connectivity Server 2 database built from a Mercator crawl. We were particularly interested in using structure to group Webpages into Websites. Such groupings are crucial to the following applications:

- Link-based authority measures (for determining which links are biased).
- Spam detection.
- Measuring the size of the web.

Many factors make this problem hard. First, there is no well-defined definition of "Website." Clearly, the hostname of a URL is not good enough, as many organizations have more than one, and others, like Geocities, contain pages from many independent authors. Websites like universities have departmental sites that are nested inside the university-wide site. There are also some very exceptional organizations like FujiXerox that overlap with two otherwise distinct organizations (in this case Fuji and Xerox). Another problem is that many distinct types of organizations exist on the Web. No simply defined organizing principle necessarily applies to all of them.

In light of this complex but potentially very rewarding problem, our approach was experimental: we wrote code to measure various properties of the Web graph, hoping that they would reveal some as yet unseen structure. Since hostname seems to at least partially correspond to the notion of Website, many of our measures focused on grouping by hostname. Since we believe that every Website has a homepage, some of our measures were intended to reveal homepages. The actual properties we measured included:

- the number of local and remote links per host

• the number of strongly connected components per host (to support the hypothesis that multisite hosts like Geocities have a large number of strongly connected components),

• the size of the link neighborhood of each page (to support the hypothesis that homepages have large neighborhoods)

• The number of links per host relative to the size of the host.

From these measurements, we made several interesting observations. The observations include:

• Even though in total there are more inlinks than outlinks on the Web, most hosts have more outlinks than inlinks. This means that hosts with fewer pages tend to have more outlinks than inlinks, and that hosts with a very large number of pages tend to have more inlinks than outlinks.

• Exceptions to the above observation include www.geocities.com, which has a large number of pages but more outlinks than inlinks. The simplest explanation for this is that it hosts many distinct authors who each control relatively few pages. Since these authors are independent of each other, there is little reason for them to link within www.geocities.com.

• A surprising number of hosts are nearly cliques (i.e. almost every possible link exists).

• A surprising number of hosts have about as many links as pages.

• Regarding the distribution of local strongly connected components, for hosts of every size, there were a large number of connected components that were close to the exact size of the host, a large number of very small strongly connected components, but very few intermediate-sized connected components. A possible explanation is that many hosts really do consist of a single strongly connected component, but the crawl simply didn't connect them. The small components would then be residuals of an incomplete crawl. One way to test this hypothesis would be to pick some example hosts, try crawling them completely, and match the resulting graphs against the components found in the original crawl.

In the next phase of our research, we compared several methods for grouping hosts by combinations of name and connectivity features (this was the subject of my intern talk). In particular, name-based grouping had been used by previous applications, although its accuracy was not well-known. Our belief was that adding connectivity features would increase the accuracy of name-based techniques. We ran four experiments:

- grouping by the “significant part ”of the hostname,
- grouping by the A, B, and C sections of the IP address,
- grouping by both hostname and IP address, and
- grouping by any two of hostname,IP address, or Conn, where Conn is a measure of the relative connectivity between two hosts.

Based on a sampling of the groupings produced, each metric found about the same number of true groupings, but the number of false positives varied greatly from metric to metric. In general, over all samples, the IP-alone grouping produced far more false than true positives. This was surprising as IP is a popular way to group hosts. The “any 2 ”grouping was generally robust, but on occasion yielded a catastrophic number of false positives. The experiment should be repeated with much larger samples.

2 Conclusion

Our experimental approach was necessarily broad and unfocused, but we see ways in which future research in this area can be more structured, based on what we have learned. We believe it is worthwhile to first consider a standard for evaluating the structure we discover. We mention three such standards.

Application-based structure. There are many motivating applications for this research. One way to measure the effectiveness of a grouping technique is to measure the performance of test applications that depend on some sort of accurate grouping, such as spam detection or authority. The downside of this approach is that it is not necessarily any easier to evaluate the performance of the test applications.

Example-based structure. Choose several specific example sites and try to determine what structural properties distinguish them. Proceed by running experiments to determine if these properties generalize to similar sites or if new types of example sites should be introduced.

Feedback-directed binary code specialization

Juan Navarro
Rice University
with
Sharon Smith, David Hunter
Alpha Technology Solutions

1. Introduction

The current approach to binary performance optimization is to apply all known or available optimizations at once. The problem is that there are complex -- sometimes negative -- interactions between individual optimizations, and deciding what subset to apply is a difficult task.

The use of profile-directed feedback to make that decision can help, because optimization is done in accordance to the actual program behavior. In addition, run-time information that is not available to the compiler can be used to discover new optimization opportunities.

Specialization or partial evaluation is an optimization technique that eliminates unnecessary generality from an application; it can benefit from profile-directed feedback to identify that excess generality.

The goal of this project is to study the potential of specialization in isolation of other optimizations. In the long term, the results are intended to help in realizing cost-benefit analysis of specialization, which allows for adaptation to specific workloads.

2. Spector

Spector is an off-line binary specializer that we built for this project. It optimizes entire functions for a particular value of one of its arguments.

The binary is first profiled to determine what the hot functions are. Then, hot functions are instrumented at the entry point to detect frequent values passed in the arguments. The instrumented program is run and the results are used to select what functions to specialize and for what argument and value.

To specialize a function, Spector builds the function's control-flow graph, propagates known values through the graph by interpreting each basic block's code, removes unreachable blocks, and deletes instructions mainly by means of constant folding. Guard code is inserted to fall back to the original function when not in the presence of the special case. If not enough instructions are deleted, the specialization for that function is discarded. A better approach would be to try to optimize for a different argument, but Spector currently does not do that.

Spector uses Atom to profile, instrument, and navigate the binary. Ideally, L'Atom (a Linux port and extension of Atom, that supports arbitrary modifications) would be used to rewrite the binary, but it was not available when Spector was built. The temporary solution was to use the application's source code (which will not be required once L'Atom is available) to produce assembly code. Since there is a one-to-one correspondence between machine instruction in the binary file and assembly instructions in the assembly listing, an ed script line is generated for each modification. Perl code is used to glue everything together.

Due to a bug in the compiler, this approach didn't work for some programs, especially large ones. When a workaround was found it was to late to try it.

3. Results

The table below shows the results for 6 small benchmark programs. The column "%" shows the percentage of cycles spent in the hottest specialized function. Column "Deleted" tells how many instructions where deleted from that function, out of the total number of instructions in the original version of the function.

Program	%	Deleted Speedup	(%)
fib	100.0	5/29	-11.1
hanoi	100.0	11/53	-1.5
linpack	87.0	6/57	0.0
sim	15.5	4/857	+1.0
fft	56.0	2/193	+1.8
dhrystone	10.8	7/53	+3.7

The specialized functions in fib and hanoi are recursive ones, and Spector specialized them for the base case. As a consequence, the overhead of the guard code is paid for nothing. This problem is easy to solve.

The noise in the experiments was negligible; therefore, a 1% speedup, as small as it is, does represent an improvement because it's clearly above the noise.

4. Conclusions and future work

More experimentation is required to better assess specialization's potential and discover formulas or heuristics to predict whether a given specialization would yield speedups or slowdowns. From the table above, the number of instructions deleted is clearly useless as a predictor.

Many improvements can be done, including the following:

- There is a long list of special cases that allow for more instruction deletions and are not currently being detected.

- Code modification introduces new optimization opportunities. A final pass with standard optimizations such as instruction rescheduling and dead code removal would most likely pay off.

- Polyvariant specialization: specialize for more that one argument per function or more than one value per argument.

- Specialize also the original code for the complement of the special case, since it will never be executed when the special case holds.

- Floating point operations were not considered at this time.

- Ignoring function boundaries and specializing for sets of basic blocks instead, would increase specialization opportunities.

Summer Internship at SRC

Shailesh Vaya
UCLA

1 Introduction

I am between the first and second year of my Masters leading to a PhD program at UCLA. The area of my research interest is Cryptography.

An interesting part of my Summer Internship at SRC was that my host gave me the flexibility to define a project/problem for myself to work on. This gave me a lot of time to talk to the researchers at SRC for the first few weeks. I spent the 2-3 weeks studying the problem of designing an Uncensorable Bulletin Board with Mark Lillibridge and prototyping a spam resistant mailing system with Mark Manasse (my host). However, we didn't implement either of these ideas.

I had the opportunity to work on two independent problems during the rest of the summer which I summarize in the next two sections.

2 On Existence of Incompressible functions

Dwork, Lotspeich, Naor ([2]) proposed a scheme for protecting Digital content from illegal distribution using a novel concept of "Digital Signets: Self Enforcing Protection of Digital Information". The main idea of the scheme was to make the decryption key to some encrypted digital information as large as the actual data, itself. A user is given a signet which is associated with his credit card number and using the pair the user can compute a decryption key to the encrypted digital data available publicly. Thus, a cheating user has two options: either part with his credit card number, or transmit a decryption key as large as the data itself! This claim, however, is not proved.

Dwork et. al. conjectured that the security of their scheme is based on the incompressiblity of the function:

$$f(u) = g_1^u \circ g_2^u \circ g_3^u \cdots \circ g_l^u$$

An incompressible function can be defined as follows. Consider two communicating parties. One party knows a short x and wants to communicate to another party the long value of $f(x)$ without revealing x. Roughly speaking, f is incompressible if no feasible computable short message from the sender to the receiver simultaneously achieves both goals of enabling the receiver to compute $f(x)$, and hiding x. The principal open problem suggested by [2] was to prove the existence/inexistence of incompressible functions based on standard cryptographic assumptions.

Working with Cynthia, I proved that Diffie-Hellman series is incompressible, under the Decisional Diffie-Hellman assumption. The main idea behind the proof is that if there exists polynomial time computable algorithm A for compressing the value of the series and polynomial time algorithm A' for retrieving the value of the series using this compressed value then there exists a polynomial time algorithm that can distinguish between a Diffie-Hellman Triplet and a triplet selected uniformly from the set of triplets in group Gp. The heart of the proof involved generating randomized Diffie-Hellman Series (or any series) using the ideas of Naor & Reingold, [1] from a Diffie-Hellman triplet (or a random triplet).

Although we made some progress it still remains an open problem to prove that the Signet Scheme is secure if the Diffie-Hellman series is incompressible. Another is to design an incompressible function that doesn't require a large publicly known data. A working manuscript of the proof may be available upon request from the authors (i.e., Cynthia & myself).

3 Resilient Deniable Authentication

I kept myself busy attending a workshop on Algorithmic Number Theory and CRYPTO-2000 for a couple of weeks and then we sought for a new problem to work on. In a joint work with Cynthia and Moni Naor (at Stanford) we studied Deniable Authentication protocols under Intrusive Adversaries. The motivation behind this work can be found while trying to construct a protocol for "Stock Brokering Without Trust" or "Private Retrieval of Medical Database".

The resiliency of an authentication protocol is defined with respect to the VIEW of an adversary. For example, an adversary having the read access to the communication channel between the prover and the verifier is more powerful and has a larger VIEW compared to an adversary who would just believe a verifier based on the Transcript that the verifier gives to the prover. We classify that an adversary could be:

1. Online/Offline: An adversary is called Online if it concurrently interacts with the verifier while he gets some message ***m*** authenticated from the prover. If the adversary behaves in an "assign & collect" fashion he is called an Offline Adversary.

2. Eavesdropping: An adversary is called eavesdropping if it has the read access to the communication line between the prover and the verifier.

3. Coercing: An adversary is called Coercing if it can suggest the verifier to send some message ***m*** or use some string of random bits for composing the message etc. We proved a number of possiblity/impossiblity results regarding the existence of authentication protocols under various combinations of adversaries as listed above. A

working mansucript of the results is under preparation (vis-a-vis 09/18/00).

4 References

[1] M. Naor, O. Reingold Number Theoritic Construction of Pseudorandom Generators, FOCS'97.

[2] Digital Signets: Self Enforcing Protection of Digital Information, STOC'96.

Animating Proofs With Juno-xyt

Boris Dimitrov
boris@cs.caltech.edu
http://www.cs.caltech.edu/~boris

1 Introduction

Juno-2 is an extensible constraint-based drawing editor created by Greg Nelson and Allan Heydon [1], which was originally designed for producing high-quality still illustrations. Juno-2 permits arbitrary constraints to be specified using a declarative notation based on the theory of real numbers with a pairing function and equality. After the constraints have been specified, any degrees of freedom that remain in the drawing may be adjusted interactively with the mouse. Thus, Juno-2 makes it easy to illustrate and to explore virtually any geometric construction in the Euclidean plane.

Long before my internship, Allan and Greg had written in the Juno language [2] a module for playing animations [3]. Greg had animated a fragment of Archimedes's reasoning for the area of the circle. Most SRCers who had seen Greg's animation thought that it communicated superbly the crux of Archimedes's proof. My job this summer was to extend Juno-2 with features that would make it easier to produce such animations.

2 DimiTeX

One difficulty that Greg had encountered in animating Archimedes proof was that typesetting mathematical formulas with Juno-2 was a very tedious task. In order for Juno to display $?r^2$, Greg had to type a 30 line program that switched the current font to "Symbol" and back again as the letter *?* was displayed, and then raised the superscript "2" up by one-third of the current font's height.

Because Greek letters, mathematical symbols, superscripts and subscripts occur commonly in proofs, I implemented a subset of TeX's functionality in Juno. My implementation, which Greg named DimiTeX, mostly adheres to TeX's syntax; however, DimiTeX's escape character is the exclamation mark '!', because the backslash '\' was already taken by Juno. For example, here is what we would have to type in Juno-xyt for the formula $?r^2$ to appear at position a.

```
DimiTeX.Show(a, "$!pi r^2$")
```

Greg wrote the documentation for DimiTeX's syntax which is included in the Juno-xyt manual page.

Some proofs use symbols that are not available in the bundled PostScript and X fonts. With Juno, it is easy to draw a new symbol and to encapsulate the resulting closure as a DimiTeX glyph. This glyph can be bound to a DimiTeX control sequence, such as "!mynewsym". (Even animated drawings can be encapsulated as DimiTeX glyphs. That's how we created the DimiTeX "!circle{...}" macro, which slowly draws a circle around any formula.)

While dragging a compound formula across the screen, I observed a curious effect which Greg called "rounding flicker". Consider the distance between a superscript and a subscript in the same
formula. The real number with which DimiTeX represents this distance may not be divisible by the screen's pixel size, but the physical distance on the screen always is. The window system rounds DimiTeX's real number either up or down, depending on the formula's current position. As the formula is being dragged, its current position changes, and so does the physical distance between the superscript and the subscript on the screen. We did not fix the resulting flicker, because we had other fun things to do.

3 Animation

An animation can be fun even if it does not prove a theorem. Early in the summer, Greg and I created an animation showing how a calligrapher might write "Juno xyt". Juno-2 already had a calligraphic pen that could be easily applied to the current PostScript path. Still, given a PostScript path which consists of a few dozen Bezier segments, how would you draw that fraction of the path which is visible in the animation's current frame?

We first solved this problem for a single Bezier segment by using De Casteljau's construction. That is, we obtained Bezier control points for the visible fraction of our segment by nesting a number of simultaneous "sliders". (After this exercise, we created a graphical slider in the Juno user interface and connected it to the Juno variable "UISlider.val"). Then we encapsulated each segment of the PostScript path into a separate animation closure. Finally, we defined a new composition primitive called "Seq2" which right-reduced the list of single-segment animations into a single closure that animated the entire path.

Toward summer's end, we used the same "Seq2" to tie together the steps of a calculational "Feijen style" proof that the Fermat numbers $(2 \; exp \; 2^n)+1$ are pairwise relatively prime. Actually, only two of the animated slides in that proof were "Feijen style" --- more traditional "title slides" stated the results that were to be proved. We grouped the various slides into "scenes" so that slide transitions within a scene were made automatically by Juno, whereas "show mode" transitions from one scene to the next would occur only in response to user events. We also implemented a "Scene menu" in the user interface that allowed us to view the scenes in random order while working in "edit mode". Edit mode is the mode that Juno starts in; show mode is a mode for giving full-screen presentations that we added this summer.

4 Conclusion

Some of the extensions that we added this summer -- for example DimiTeX -- work just as well in Juno-2 as they do in Juno-xyt. Although this is also true of the last and the least trivial to implement addition that I will describe --- a facility for unfolding and refolding predicates and templates and for "narrowing" predicates --- we were not hard-pressed to make this addition to Juno until we started producing larger animations. That's because still illustrations seldom use the same predicates and the same templates instantiated over and over for the exact same parameter values, which would be required for automated editing operations to pay off. But that's just what happens in slide shows and animations --- they usually consist of many scenes all of which are derived from the same template. For example, all scenes in Greg's animation of Archimedes's proof display a caption at the exact same anchor point. What if, while editing scene N, Greg needed to move down that anchor point in order to make room for a larger caption? In Juno-2, he would have had to edit manually all other scenes in addition to editing scene N (please note that storing the anchor point in a global variable is not a valid option, because there may be non-trivial constraints relating the anchor point to other points in the same template). In Juno-xyt, Greg would get away by editing just the common template --- assuming that he has read the "Using Schemes" section of the Juno-xyt manual page.

5 References

1. Allan Heydon and Greg Nelson,
   The Juno-2 Constraint-Based Drawing Editor (SRC Research Report 131a)

http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-131a.html

2. Greg Nelson and Allan Heydon,
   Juno-2 Language Definition (SRC Technical Note 1997-009)

http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-009.html

3. Allan Heydon and Greg Nelson,
   Constraint-Based Animations (PostScript, 2 pp.)
   http://research.compaq.com/SRC/juno-2/papers/animations.ps