# 2 Lua Performance Tips

Roberto Ierusalimschy

In Lua, as in any other programming language, we should always follow the two maxims of program optimization:

**Rule #1:** *Don't do it.*

**Rule #2:** *Don't do it yet.* (for experts only)

Those rules are particularly relevant when programming in Lua. Lua is famous for its performance, and it deserves its reputation among scripting languages.

Nevertheless, we all know that performance is a key ingredient of programming. It is not by chance that problems with exponential time complexity are called *intractable*. A too late result is a useless result. So, every good programmer should always balance the costs from spending resources to optimize a piece of code against the gains of saving resources when running that code.

The first question regarding optimization a good programmer always asks is: "Does the program needs to be optimized?" If the answer is positive (but only then), the second question should be: "Where?"

To answer both questions we need some instrumentation. We should not try to optimize software without proper measurements. The difference between experienced programmers and novices is *not* that experienced programmers are better at spotting where a program may be wasting its time: The difference is that experienced programmers know they are not good at that task.

A few years ago, Noemi Rodriguez and I developed a prototype for a CORBA ORB (Object Request Broker) in Lua, which later evolved into OiL (*Orb in Lua*). As a first prototype, the implementation aimed at simplicity. To avoid the need for extra C libraries, the prototype serialized integers using a few arithmetic operations to isolate each byte (conversion to base 256). It did not support floating-point values. Because CORBA handles strings as sequences of characters, our ORB first converted Lua strings into a sequence (that is, a Lua table) of characters and then handled the result like any other sequence.

When we finished that first prototype, we compared its performance against a professional ORB implemented in C++. We expected our ORB to be somewhat slower, as it was implemented in Lua, but we were disappointed by how much slower it was. At first, we just laid the blame on Lua. Later, we suspected that the culprit could be all those operations needed to serialize each number. So, we decided to run the program under a profiler. We used a very simple profiler, not unlike the one described in Chapter 23 of *Programming in Lua*. The profiler results shocked us. Against our gut feelings, the serialization of numbers had no measurable impact on the performance, among other reasons because there were not that many numbers to serialize. The serialization of strings, however, was responsible for a huge part of the total time. Practically every CORBA message has several strings, even when we are not manipulating strings explicitly: object references, method names, and some other internal values are all coded as strings. And the serialization of each string was an expensive operation, because it needed to create a new table, fill it with each individual character, and then serialize the resulting sequence, which involved serializing each character one by one. Once we reimplemented the serialization of strings as a special case (instead of using the generic code for sequences), we got a respectable speed up. With just a few extra lines of code, the performance of your implementation was comparable to the C++ implementation.[1]

So, we should always measure when optimizing a program for performance. Measure before, to know where to optimize. And measure after, to know whether the "optimization" actually improved our code.

Once you decide that you really must optimize your Lua code, this text may help you about how to optimize it, mainly by showing what is slow and what is fast in Lua. I will not discuss here general techniques for optimization, such as better algorithms. Of course you should know and use those techniques, but there are several other places where you can learn them. In this article I will discuss only techniques that are particular to Lua. Along the article, I will constantly measure the time and space performance of small programs. Unless stated otherwise, I do all measures on a Pentium IV 2.9 GHz with 1 GB of main memory, running Ubuntu 7.10, Lua 5.1.1. Frequently I give actual measures (e.g., 7 seconds), but what is relevant is the relationship between different measures. When I say that a program is "$X\%$ times faster" than another it means that it runs in $X\%$ less time. (A program 100% faster would take no time to run.) When I say that a program is "$X\%$ times slower" than another I mean that the other is $X\%$ faster. (A program 50% slower means that it takes twice the time.)

[1]Of course our implementation was still slower, but not by an order of magnitude.

## Basic facts

Before running any code, Lua translates (precompiles) the source into an internal format. This format is a sequence of instructions for a virtual machine, similar to machine code for a real CPU. This internal format is then interpreted by C code that is essentially a *while* loop with a large *switch* inside, one case for each instruction.

Perhaps you have already read somewhere that, since version 5.0, Lua uses a register-based virtual machine. The "registers" of this virtual machine do not correspond to real registers in the CPU, because this correspondence would be not portable and quite limited in the number of registers available. Instead, Lua uses a stack (implemented as an array plus some indices) to accommodate its registers. Each active function has an *activation record*, which is a stack slice wherein the function stores its registers. So, each function has its own registers[2]. Each function may use up to 250 registers, because each instruction has only 8 bits to refer to a register.

Given that large number of registers, the Lua precompiler is able to store all local variables in registers. The result is that access to local variables is very fast in Lua. For instance, if `a` and `b` are local variables, a Lua statement like `a = a + b` generates one single instruction: `ADD 0 0 1` (assuming that `a` and `b` are in registers 0 and 1, respectively). For comparison, if both `a` and `b` were globals, the code for that addition would be like this:

```
GETGLOBAL       0 0     ; a
GETGLOBAL       1 1     ; b
ADD             0 0 1
SETGLOBAL       0 0     ; a
```

So, it is easy to justify one of the most important rules to improve the performance of Lua programs: *use locals!*

If you need to squeeze performance out of your program, there are several places where you can use locals besides the obvious ones. For instance, if you call a function within a long loop, you can assign the function to a local variable. For instance, the code

```
for i = 1, 1000000 do
  local x = math.sin(i)
end
```

runs 30% slower than this one:

```
local sin = math.sin
for i = 1, 1000000 do
  local x = sin(i)
end
```

[2]This is similar to the *register windows* found in some CPUs.

Access to external locals (that is, variables that are local to an enclosing function) is not as fast as access to local variables, but it is still faster than access to globals. Consider the next fragment:

```
function foo (x)
  for i = 1, 1000000 do
    x = x + math.sin(i)
  end
  return x
end

print(foo(10))
```

We can optimize it by declaring `sin` once, outside function `foo`:

```
local sin = math.sin
function foo (x)
  for i = 1, 1000000 do
    x = x + sin(i)
  end
  return x
end

print(foo(10))
```

This second code runs 30% faster than the original one.

Although the Lua compiler is quite efficient when compared with compilers for other languages, compilation is a heavy task. So, you should avoid compiling code in your program (e.g., function `loadstring`) whenever possible. Unless you must run code that is really dynamic, such as code entered by an end user, you seldom need to compile dynamic code.

As an example, consider the next code, which creates a table with functions to return constant values from 1 to 100000:

```
local lim = 10000
local a = {}
for i = 1, lim do
  a[i] = loadstring(string.format("return %d", i))
end
print(a[10]())   --> 10
```

This code runs in 1.4 seconds.

With closures, we avoid the dynamic compilation. The next code creates the same 100000 functions in $\frac{1}{10}$ of the time (0.14 seconds):

```
function fk (k)
  return function () return k end
end
```

```
local lim = 100000
local a = {}
for i = 1, lim do  a[i] = fk(i)  end
print(a[10]())   --> 10
```

## About tables

Usually, you do not need to know anything about how Lua implement tables to use them. Actually, Lua goes to great lengths to make sure that implementation details do not surface to the user. However, these details show themselves through the performance of table operations. So, to optimize programs that use tables (that is, practically any Lua program), it is good to know a little about how Lua implements tables.

The implementation of tables in Lua involves some clever algorithms. Every table in Lua has two parts: the *array* part and the *hash* part. The array part stores entries with integer keys in the range 1 to $n$, for some particular $n$. (We will discuss how this $n$ is computed in a moment.) All other entries (including integer keys outside that range) go to the hash part.

As the name implies, the hash part uses a hash algorithm to store and find its keys. It uses what is called an *open address* table, which means that all entries are stored in the hash array itself. A hash function gives the primary index of a key; if there is a collision (that is, if two keys are hashed to the same position), the keys are linked in a list, with each element occupying one array entry.

When Lua needs to insert a new key into a table and the hash array is full, Lua does a *rehash*. The first step in the rehash is to decide the sizes of the new array part and the new hash part. So, Lua traverses all entries, counting and classifying them, and then chooses as the size of the array part the largest power of 2 such that more than half the elements of the array part are filled. The hash size is then the smallest power of 2 that can accommodate all the remaining entries (that is, those that did not fit into the array part).

When Lua creates an empty table, both parts have size 0 and, therefore, there are no arrays allocated for them. Let us see what happens when we run the following code:

```
local a = {}
for i = 1, 3 do
  a[i] = true
end
```

It starts by creating an empty table `a`. In the first loop iteration, the assignment `a[1]=true` triggers a rehash; Lua then sets the size of the array part of the table to 1 and keeps the hash part empty. In the second loop iteration, the assignment `a[2]=true` triggers another rehash, so that now the array part of the table has size 2. Finally, the third iteration triggers yet another rehash, growing the size of the array part to 4.

A code like

```
a = {}
a.x = 1; a.y = 2; a.z = 3
```

does something similar, except that it grows the hash part of the table.

For large tables, this initial overhead is amortized over the entire creation: While a table with three elements needs three rehashings, a table with one million elements needs only twenty. But when you create thousands of small tables, the combined overhead can be significant.

Older versions of Lua created empty tables with some pre-allocated slots (four, if I remember correctly), to avoid this overhead when initializing small tables. However, this approach wastes memory. For instance, if you create millions of points (represented as tables with only two entries) and each one uses twice the memory it really needs, you may pay a high price. That is why currently Lua creates empty tables with no pre-allocated slots.

If you are programming in C, you can avoid those rehashings with the Lua API function `lua_createtable`. It receives two arguments after the omnipresent `lua_State`: the initial size of the array part and the initial size of the hash part of the new table.[3] By giving appropriate sizes to the new table, it is easy to avoid those initial rehashes. Beware, however, that Lua can only shrink a table when rehashing it. So, if your initial sizes are larger than needed, Lua may never correct your waste of space.

When programming in Lua, you may use constructors to avoid those initial rehashings. When you write `{true, true, true}`, Lua knows beforehand that the table will need three slots in its array part, so Lua creates the table with that size. Similarly, if you write `{x = 1, y = 2, z = 3}`, Lua will create a table with four slots in its hash part. As an example, the next loop runs in 2.0 seconds:

```
for i = 1, 1000000 do
  local a = {}
  a[1] = 1; a[2] = 2; a[3] = 3
end
```

If we create the tables with the right size, we reduce the run time to 0.7 seconds:

```
for i = 1, 1000000 do
  local a = {true, true, true}
  a[1] = 1; a[2] = 2; a[3] = 3
end
```

If you write something like `{[1] = true, [2] = true, [3] = true}`, however, Lua is not smart enough to detect that the given expressions (literal numbers, in this case) describe array indices, so it creates a table with four slots in its *hash part*, wasting memory and CPU time.

[3]Although the rehash algorithm always sets the array size to a power of two, the array size can be any value. The hash size, however, must be a power of two, so the second argument is always rounded to the smaller power of two not smaller than the original value.

The size of both parts of a table are recomputed only when the table rehashes, which happens only when the table is completely full and Lua needs to insert a new element. As a consequence, if you traverse a table erasing all its fields (that is, setting them all to nil), the table does not shrink. However, if you insert some new elements, then eventually the table will have to resize. Usually this is not a problem: if you keep erasing elements and inserting new ones (as is typical in many programs), the table size remains stable. However, you should not expect to recover memory by erasing the fields of a large table: It is better to free the table itself.

A dirty trick to force a rehash is to insert enough nil elements into the table. See the next example:

```
a = {}
lim = 10000000
for i = 1, lim do a[i] = i end            --  create a huge table
print(collectgarbage("count"))            --> 196626
for i = 1, lim do a[i] = nil end          --  erase all its elements
print(collectgarbage("count"))            --> 196626
for i = lim + 1, 2*lim do a[i] = nil end  --  create many nil elements
print(collectgarbage("count"))            --> 17
```

I do not recommend this trick except in exceptional circumstances: It is slow and there is no easy way to know how many elements are “enough”.

You may wonder why Lua does not shrink tables when we insert nils. First, to avoid testing what we are inserting into a table; a check for nil assignments would slow down all assignments. Second, and more important, to allow nil assignments when traversing a table. Consider the next loop:

```
for k, v in pairs(t) do
  if some_property(v) then
    t[k] = nil    -- erase that element
  end
end
```

If Lua rehashed the table after a nil assignment, it would havoc the traversal.

If you want to erase all elements from a table, a simple traversal is the correct way to do it:

```
for k in pairs(t) do
  t[k] = nil
end
```

A “smart” alternative would be this loop:

```
while true do
  local k = next(t)
  if not k then break end
  t[k] = nil
end
```

However, this loop is very slow for large tables. Function `next`, when called without a previous key, returns the “first” element of a table (in some random order). To do that, `next` traverses the table arrays from the beginning, looking for a non-nil element. As the loop sets the first elements to nil, `next` takes longer and longer to find the first non-nil element. As a result, the “smart” loop takes 20 seconds to erase a table with 100,000 elements; the traversal loop using `pairs` takes 0.04 seconds.

## About strings

As with tables, it is good to know how Lua implements strings to use them more efficiently.

The way Lua implements strings differs in two important ways from what is done in most other scripting languages. First, all strings in Lua are *internalized*; this means that Lua keeps a single copy of any string. Whenever a new string appears, Lua checks whether it already has a copy of that string and, if so, reuses that copy. Internalization makes operations like string comparison and table indexing very fast, but it slows down string creation.

Second, variables in Lua never hold strings, but only references to them. This implementation speeds up several string manipulations. For instance, in Perl, when you write something like `$x = $y`, where `$y` contains a string, the assignment copies the string contents from the `$y` buffer into the `$x` buffer. If the string is long, this becomes an expensive operation. In Lua, this assignment involves only copying a pointer to the actual string.

This implementation with references, however, slows down a particular form of string concatenation. In Perl, the operations `$s = $s . "x"` and `$s .= "x"` are quite different. In the first one, you get a copy of `$s` and adds `"x"` to its end. In the second one, the `"x"` is simply appended to the internal buffer kept by the `$s` variable. So, the second form is independent from the string size (assuming the buffer has space for the extra text). If you have these commands inside loops, their difference is the difference between a linear and a quadratic algorithm. For instance, the next loop takes almost five minutes to read a 5MByte file:

```
$x = "";
while (<>) {
  $x = $x . $_;
}
```

If we change `$x = $x . $_` to `$x .= $_`, this time goes down to 0.1 seconds!

Lua does not offer the second, faster option, because its variables do not have buffers associated to them. So, we must use an explicit buffer: a table with the string pieces does the job. The next loop reads that same 5MByte file in 0.28 seconds. Not as fast as Perl, but quite good.

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line
end
s = table.concat(t, "\n")
```

## Reduce, reuse, recycle

When dealing with Lua resources, we should apply the same three R's promoted for the Earth's resources.

Reduce is the simplest alternative. There are several ways to avoid the need for new objects. For instance, if your program uses too many tables, you may consider a change in its data representation. As a simple example, consider that your program manipulates polylines. The most natural representation for a polyline in Lua is as a list of points, like this:

```
polyline = { { x = 10.3, y = 98.5 },
             { x = 10.3, y = 18.3 },
             { x = 15.0, y = 98.5 },
               ...
           }
```

Although natural, this representation is not very economic for large polylines, as it needs a table for each single point. A first alternative is to change the records into arrays, which use less memory:

```
polyline = { { 10.3, 98.5 },
             { 10.3, 18.3 },
             { 15.0, 98.5 },
               ...
           }
```

For a polyline with one million points, this change reduces the use of memory from 95 KBytes to 65 KBytes. Of course, you pay a price in readability: `p[i].x` is easier to understand than `p[i][1]`.

A yet more economic alternative is to use one list for the x coordinates and another one for the y coordinates:

```
polyline = { x = { 10.3, 10.3, 15.0, ...},
             y = { 98.5, 18.3, 98.5, ...}
           }
```

The original `p[i].x` now is `p.x[i]`. With this representation, a one-million-point polyline uses only 24 KBytes of memory.

A good place to look for chances of reducing garbage creation is in loops. For instance, if a constant table is created inside a loop, you can move it out the loop, or even out of the function enclosing its creation. Compare:

```
function foo (...)
  for i = 1, n do
    local t = {1, 2, 3, "hi"}
    -- do something without changing ’t’
    ...
  end
end

local t = {1, 2, 3, "hi"}   -- create ’t’ once and for all
function foo (...)
  for i = 1, n do
    -- do something without changing ’t’
    ...
  end
end
```

The same trick may be used for closures, as long as you do not move them out of the scope of the variables they need. For instance, consider the following function:

```
function changenumbers (limit, delta)
  for line in io.lines() do
    line = string.gsub(line, "%d+", function (num)
              num = tonumber(num)
              if num >= limit then return tostring(num + delta) end
              -- else return nothing, keeping the original number
            end)
    io.write(line, "\n")
  end
end
```

We can avoid the creation of a new closure for each line by moving the inner function outside the loop:

```
function changenumbers (limit, delta)
  local function aux (num)
    num = tonumber(num)
    if num >= limit then return tostring(num + delta) end
  end
  for line in io.lines() do
    line = string.gsub(line, "%d+", aux)
    io.write(line, "\n")
  end
end
```

However, we cannot move `aux` outside function `changenumbers`, because there `aux` cannot access `limit` and `delta`.

For many kinds of string processing, we can reduce the need for new strings by working with indices over existing strings. For instance, the `string.find` function returns the position where it found the pattern, instead of the match. By returning indices, it avoids creating a new (sub)string for each successful match. When necessary, the programmer can get the match substring by calling `string.sub`.[4]

When we cannot avoid the use of new objects, we still may avoid creating these new objects through reuse. For strings reuse is not necessary, because Lua does the job for us: it always *internalizes* all strings it uses, therefore reusing them whenever possible. For tables, however, reuse may be quite effective. As a common case, let us return to the situation where we are creating new tables inside a loop. This time, however, the table contents are not constant. Nevertheless, frequently we still can reuse the same table in all iterations, simply changing its contents. Consider this chunk:

```
local t = {}
for i = 1970, 2000 do
  t[i] = os.time({year = i, month = 6, day = 14})
end
```

The next one is equivalent, but it reuses the table:

```
local t = {}
local aux = {year = nil, month = 6, day = 14}
for i = 1970, 2000 do
  aux.year = i
  t[i] = os.time(aux)
end
```

A particularly effective way to achieve reuse is through *memoizing*. The basic idea is quite simple: store the result of some computation for a given input so that, when the same input is given again, the program simply reuses that previous result.

LPeg, a new package for pattern matching in Lua, does an interesting use of memoizing. LPeg compiles each pattern into an internal representation, which is a “program” for a parsing machine that performs the matching. This compilation is quite expensive, when compared with matching itself. So, LPeg memoizes the results from its compilations to reuse them. A simple table associates the string describing a pattern to its corresponding internal representation.

A common problem with memoizing is that the cost in space to store previous results may outweigh the gains of reusing those results. To solve this problem in Lua, we can use a weak table to keep the results, so that unused results are eventually removed from the table.

In Lua, with higher-order functions, we can define a generic memoization function:

[4]It would be a good idea for the standard library to have a function to compare substrings, so that we could check specific values inside a string without having to extract that value from the string (thereby creating a new string).

```
function memoize (f)
  local mem = {}                       -- memoizing table
  setmetatable(mem, {__mode = "kv"})   -- make it weak
  return function (x)       -- new version of ’f’, with memoizing
    local r = mem[x]
    if r == nil then    -- no previous result?
      r = f(x)          -- calls original function
      mem[x] = r        -- store result for reuse
    end
    return r
  end
end
```

Given any function `f`, `memoize(f)` returns a new function that returns the same results as `f` but memoizes them. For instance, we can redefine `loadstring` with a memoizing version:

```
loadstring = memoize(loadstring)
```

We use this new function exactly like the old one, but if there are many repeated strings among those we are loading, we can have a substantial performance gain.

If your program creates and frees too many coroutines, recycling may be an option to improve its performance. The current API for coroutines does not offer direct support for reusing a coroutine, but we can circumvent this limitation. Consider the next coroutine:

```
co = coroutine.create(function (f)
       while f do
         f = coroutine.yield(f())
       end
     end
```

This coroutine accepts a job (a function to run), runs it, and when it finishes it waits for a next job.

Most recycling in Lua is done automatically by the garbage collector. Lua uses an incremental garbage collector. That means that the collector performs its task in small steps (incrementally) interleaved with the program execution. The pace of these steps is proportional to memory allocation: for each amount of memory allocated by Lua, the garbage collector does some proportional work. The faster the program consumes memory, the faster the collector tries to recycle it.

If we apply the principles of reduce and reuse to our program, usually the collector will not have too much work to do. But sometimes we cannot avoid the creation of large amounts of garbage and the collector may become too heavy. The garbage collector in Lua is tuned for average programs, so that it performs reasonably well in most applications. However, sometimes we can improve the performance of a program by better tunning the collector for that particular case.

We can control the garbage collector through function `collectgarbage`, in Lua, or `lua_gc`, in C. Both offer basically the same functionality, although with different interfaces. For this discussion I will use the Lua interface, but often this kind of manipulation is better done in C.

Function `collectgarbage` provides several functionalities: it may stop and restart the collector, force a full collection cycle, force a collection step, get the total memory in use by Lua, and change two parameters that affect the pace of the collector. All of them have their uses when tunning a memory-hungry program.

Stopping the collector "forever" may be an option for some kinds of batch programs, which create several data structures, produce some output based on those structures, and exit (e.g., compilers). For those programs, trying to collect garbage may be a waste of time, as there is little garbage to be collected and all memory will be released when the program finishes.

For non-batch programs, stopping the collector forever is not an option. Nevertheless, those programs may benefit from stopping the collector during some time-critical periods. If necessary, the program may get full control of the garbage collector by keeping it stopped at all times, only allowing it to run by explicitly forcing a step or a full collection. For instance, several event-driven platforms offer an option to set an *idle function*, which is called when there are no other events to handle. This is a perfect time to do garbage collection. (In Lua 5.1, each time you force some collection when the collector is stopped, it automatically restarts. So, to keep it stopped, you must call `collectgarbage("stop")` immediately after forcing some collection.)

Finally, as a last resort, you may try to change the collector parameters. The collector has two parameters controlling its pace. The first one, called *pause*, controls how long the collector waits between finishing a collection cycle and starting the next one. The second parameter, called *stepmul* (from *step multiplier*), controls how much work the collector does in each step. Roughly, smaller pauses and larger step multipliers increase the collector's speed.

The effect of these parameters on the overall performance of a program is hard to predict. A faster collector clearly wastes more CPU cycles per se; however, it may reduce the total memory in use by a program, which in turn may reduce paging. Only careful experimentation can give you the best values for those parameters.

## Final remarks

As we discussed in the introduction, optimization is a tricky business. There are several points to consider, starting with whether the program needs any optimization at all. If it has real performance problems, then we must focus on where and how to optimize it.

The techniques we discussed here are neither the only nor the most important ones. We focused here on techniques that are peculiar to Lua, as there are several sources for more general techniques.

Before we finish, I would like to mention two options that are at the borderline of improving performance of Lua programs, as both involve changes outside the scope of the Lua code. The first one is to use LuaJIT, a Lua just-in-time compiler developed by Mike Pall. He has been doing a superb job and LuaJIT is probably the fastest JIT for a dynamic language nowadays. The drawbacks are that it runs only on x86 architectures and that you need a non-standard Lua interpreter (LuaJIT) to run your programs. The advantage is that you can run your program 5 times faster with no changes at all to the code.

The second option is to move parts of your code to C. After all, one of Lua hallmarks is its ability to interface with C code. The important point in this case is to choose the correct level of granularity for the C code. On the one hand, if you move only very simple functions into C, the communication overhead between Lua and C may kill any gains from the improved performance of those functions. On the other hand, if you move too large functions into C, you loose flexibility.

Finally, keep in mind that those two options are somewhat incompatible. The more C code your program has, the less LuaJIT can optimize it.