Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1159-1176
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05

The Implementation of Lua 5.0

Roberto Ierusalimschy
(Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil
roberto@inf.puc-rio.br)

Luiz Henrique de Figueiredo
(IMPA–Instituto de Matemática Pura e Aplicada, Rio de Janeiro, Brazil
lhf@impa.br)

Waldemar Celes
(Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil
celes@inf.puc-rio.br)

Abstract: We discuss the main novelties of the implementation of Lua 5.0: its register-based virtual machine, the new algorithm for optimizing tables used as arrays, the implementation of closures, and the addition of coroutines.
Key Words: compilers, virtual machines, hash tables, closures, coroutines
Category: D.3.4, D.3.3, D.3.2, E.2

1 Introduction

Lua was born in an academic laboratory as a tool for in-house software development but somehow was adopted by several industrial projects around the world and is now widely used in the game industry.[1]

How do we account for this widespread use of Lua? We believe that the answer lies in our design and implementation goals: to provide an embeddable scripting language that is simple, efficient, portable, and lightweight. These have been our main goals since the birth of Lua in 1993 and they have been respected during its evolution. (For a history of the development of Lua up to just before the release of Lua 5.0, see [12].) These features, plus the fact that Lua has been designed from the start to be embedded into larger applications, account for its early adoption by the industry.[2]

Widespread use generates demand for language features. Several features of Lua have been motivated by industrial needs and user feedback. Important examples are the introduction of coroutines in Lua 5.0 and the implementation

[1] An informal poll conducted in September 2003 by `gamedev.net`, an important site for game programmers, showed Lua as the most popular scripting language for game development. For details, see `http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163`.

[2] The adoption of a liberal MIT-like license also helped.

of incremental garbage collection in the upcoming Lua 5.1. Both features are specially important for games.

In this paper, we discuss the main novelties of the implementation of Lua 5.0, compared to Lua 4.0:

Register-based virtual machine: Traditionally, most virtual machines intended for actual execution are stack based, a trend that started with Pascal's P-machine [15] and continues today with Java's JVM and Microsoft's .Net environment. Currently, however, there has been a growing interest in register-based virtual machines (for instance, the planned new virtual machine for Perl 6 (Parrot) will be register based [17]). As far as we know, the virtual machine of Lua 5.0 is the first register-based virtual machine to have a wide use. This virtual machine is presented in Section 7.

New algorithm for optimizing tables used as arrays: Unlike other scripting languages, Lua does not offer an array type. Instead, Lua programmers use regular tables with integer indices to implement arrays. Lua 5.0 uses a new algorithm that detects whether tables are being used as arrays and automatically stores the values associated to numeric indices in an actual array, instead of adding them to the hash table. This algorithm is discussed in Section 4.

The implementation of closures: Lua 5.0 supports first-class functions with lexical scoping. This mechanism poses a well-known difficulty for languages that use an array-based stack to store activation records. Lua uses a novel approach to function closures that keeps local variables in the (array-based) stack and only moves them to the heap if they go out of scope while being referred by nested functions. The implementation of closures is discussed in Section 5.

The addition of coroutines: Lua 5.0 introduced coroutines in the language. Although the implementation of coroutines is more or less traditional, we present a short overview in Section 6 for completeness.

The other sections complement or give background to this discussion. In Section 2 we present an overview of Lua's design goals and how those goals have driven its implementation. In Section 3 we describe how Lua represents its values. Although the representation itself has no novelties, we need this material for the other sections. Finally, in Section 8 we present a small benchmark and draw some conclusions.

2 An Overview of Lua's Design and Implementation

As mentioned in the introduction, the goals in our implementation of Lua are:

Simplicity: We seek the simplest language we can afford and the simplest C code that implements this language. This implies a simple syntax with a small number of language constructs, not far from the tradition.

Efficiency: We seek fast compilation and fast execution of Lua programs. This implies a fast, smart, one-pass compiler and a fast virtual machine.

Portability: We want Lua to run on as many platforms as possible. We want to be able to compile the Lua core unmodified everywhere and to run Lua programs unmodified on every platform that has a suitable Lua interpreter. This implies a clean ANSI C implementation with special attention to portability issues, such as avoiding dark corners of C and its libraries, and ensuring that it also compiles cleanly as C++. We seek warning-free compilations.

Embeddability: Lua is an extension language; it is designed to provide scripting facilities to larger programs. This and the other goals imply the existence of a C API that is simple and powerful, but which relies mostly on built-in C types.

Low embedding cost: We want it to be easy to add Lua to an application without bloating it. This implies tight C code and a small Lua core, with extensions being added as user libraries.

These goals are somewhat conflicting. For instance, Lua is frequently used as a data-description language, for storing and loading configuration files and sometimes quite large databases (Lua programs with a few megabytes are not uncommon). This implies that we need a fast Lua compiler. On the other hand, we want Lua programs to execute fast. This implies a smart compiler, one that generates good code for the virtual machine. So, the implementation of the Lua compiler has to balance between these two requirements. However, the compiler cannot be too large; otherwise it would bloat the whole package. Currently the compiler accounts for approximately 30% of the size of the Lua core. For memory-limited applications, such as embedded systems, it is possible to embed Lua without the compiler. Lua programs are then precompiled off-line and loaded at run time by a tiny module (which is also fast because it loads binary files).

Lua uses a hand-written scanner and a hand-written recursive descent parser. Until version 3.0, Lua used a parser produced by yacc [13], which proved a valuable tool when the language's syntax was less stable. However, the hand-written parser is smaller, more efficient, more portable, and fully reentrant. It also provides better error messages.

The Lua compiler uses no intermediate representation. It emits instructions for the virtual machine "on the fly" as it parses a program. Nevertheless, it does perform some optimizations. For instance, it delays the generation of code for base expressions like variables and constants. When it parses such expressions, it generates no code; instead, it uses a simple structure to represent them. Therefore, it is very easy to check whether an operand for a given instruction is a constant or a local variable and use those values directly in the instruction, thus avoiding unnecessary and costly moves (see Section 3).

To be portable across many different C compilers and platforms, Lua cannot use several tricks commonly used by interpreters, such as direct threaded code [8, 16]. Instead, it uses a standard `while–switch` dispatch loop. Also, at places the C code seems unduly complicated, but the complication is there to ensure portability. The portability of Lua's implementation has increased steadily throughout the years, as Lua got compiled under many different C compilers in many different platforms (including several 64-bit platforms and some 16-bit platforms).

We consider that we have achieved our design and implementation goals. Lua is a very portable language: it runs on any machine with an ANSI C compiler, from embedded systems to mainframes. Lua is really lightweight: for instance, on Linux its stand-alone interpreter, complete with all standard libraries, takes less than 150 Kbytes; the core is less than 100 Kbytes. Lua is efficient: independent benchmarks [2, 4] show Lua as one of the fastest language implementations in the realm of scripting languages (i.e., interpreted and dynamically-typed languages). We also consider Lua a simple language, being syntactically similar to Pascal and semantically similar to Scheme, but this is subjective.

3 The Representation of Values

Lua is a dynamically-typed language: types are attached to values rather than to variables. Lua has eight basic types: *nil*, *boolean*, *number*, *string*, *table*, *function*, *userdata*, and *thread*. Nil is a marker type having only one value, also called nil. Boolean values are the usual `true` and `false`. Numbers are double-precision floating-point numbers, corresponding to the type `double` in C, but it is easy to compile Lua using `float` or `long` instead. (Several games consoles and smaller machines lack hardware support for `double`.) Strings are arrays of bytes with an explicit size, and so can contain arbitrary binary data, including embedded zeros. Tables are associative arrays, which can be indexed by any value (except `nil`) and can hold any value. Functions are either Lua functions or C functions written according to a protocol for interfacing with the Lua virtual machine. Userdata are essentially pointers to user memory blocks, and come in two flavors: *heavy*, whose blocks are allocated by Lua and are subject to garbage collection,

```
typedef struct {
  int t;
  Value v;
} TObject;
```

```
typedef union {
  GCObject *gc;
  void *p;
  lua_Number n;
  int b;
} Value;
```

Figure 1: Lua values are represented as tagged unions

and *light*, whose blocks are allocated and freed by the user. Finally, threads represent coroutines. Values of all types are first-class values: we can store them in global variables, local variables and table fields, pass them as arguments to functions, return them from functions, etc.

Lua represents values as *tagged unions*, that is, as pairs (t, v), where t is an integer tag identifying the type of the value v, which is a union of C types implementing Lua types. Nil has a single value. Booleans and numbers are implemented as 'unboxed' values: v represents values of those types directly in the union. This implies that the union must have enough space for a `double`. Strings, tables, functions, threads, and userdata values are implemented by reference: v contains pointers to structures that implement those values. Those structures share a common head, which keeps information needed for garbage collection. The rest of the structure is specific to each type.

Figure 1 shows a glimpse of the actual implementation of Lua values. `TObject` is the main structure in this implementation: it represents the tagged unions (t, v) described above. `Value` is the union that implements the values. Values of type nil are not explicitly represented in the union because the tag is enough to identify them. The field `n` is used for numbers (by default, `lua_Number` is `double`). The field `b` is used for booleans. The field `p` is used for light userdata. The field `gc` is used for the other values (strings, tables, functions, heavy userdata, and threads), which are those subject to garbage collection.

One consequence of using tagged unions to represent Lua values is that copying values is a little expensive: on a 32-bit machine with 64-bit doubles, the size of a `TObject` is 12 bytes (or 16 bytes, if doubles are aligned on 8-byte boundaries) and so copying a value requires copying 3 (or 4) machine words. However, it is difficult to implement a better representation for values in ANSI C. Several dynamically-typed languages (e.g., the original implementation of Smalltalk80 [9]) use spare bits in each pointer to store the value's type tag. This trick works in most machines because, due to alignment, the last two or three bits of a pointer are always zero, and therefore can be used for other purposes. However, this technique is neither portable nor implementable in ANSI C.

The C standard does not even ensures that a pointer fits in any integral type and so there is no standard way to perform bit manipulation over pointers.

Another option to reduce the size of a value would be to keep the explicit tag, but to avoid putting a double in the union. For instance, all numbers could be represented as heap-allocated objects, just like strings. (Python uses this technique, except that it preallocates some small integer values.) However, that representation would make the language quite slow. Alternatively, integer values could be represented as unboxed values, directly inside the union, while floating-point values would go to the heap. That solution would greatly increase the complexity of the implementation of all arithmetic operations.

Like earlier interpreted languages, such as Snobol [11] and Icon [10], Lua *internalizes* strings using a hash table: it keeps a single copy of each string with no duplications. Moreover, strings are immutable: once internalized, a string cannot be changed. Hash values for strings are computed by a simple expression that mixes bitwise and arithmetic operations, thus shuffling all bits involved. Hash values are saved when the string is internalized to allow fast string comparison and table indexing. The hash function does not look at all bytes of the string if the string is too long. This allows fast hashing of long strings. Avoiding loss of performance when handling long strings is important because they are common in Lua. For instance, it is usual to process files in Lua by reading them completely into memory into a single long string.

4 Tables

Tables are the main — in fact, the *only* — data-structuring mechanism in Lua. Tables play a key role not only in the language but also in its implementation. Effort spent on a good implementation of tables is rewarded in the language because tables are used for several internal tasks, with no qualms about performance. This helps to keep the implementation small. Conversely, the absence of any other data-structuring mechanism places a pressure on an efficient implementation of tables.

Tables in Lua are *associative arrays*, that is, they can be indexed by any value (except `nil`) and can hold values of any type. Moreover, they are dynamic in the sense that they may grow when data is added to them (by assigning a value to a hitherto non-existent field) and shrink when data is removed from them (by assigning `nil` to a field).

Unlike many other scripting languages, Lua does not have an array type. Arrays are represented as tables with integer keys. The use of tables for arrays bring benefits to the language. The main one is simplicity: Lua does not need two different sets of operators to manipulate tables and arrays. Moreover, programmers do not have to choose between the two representations. The implementation of sparse arrays is trivial in Lua. In Perl, for instance, you can run

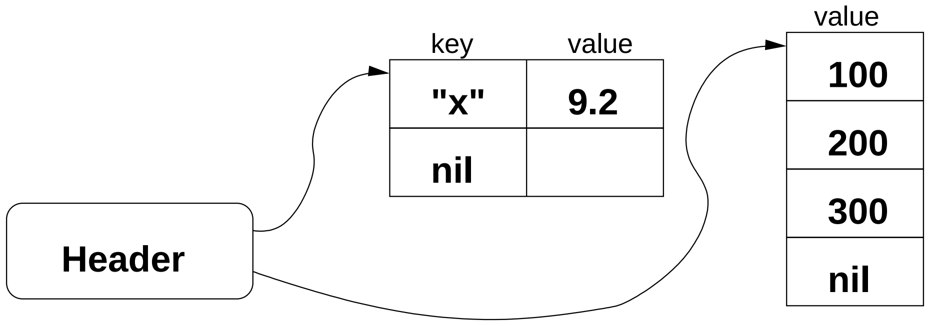

Figure 2: A Lua table.

out of memory if you try to run the program `$a[1000000000]=1;`, as it triggers the creation of an array with one billion elements. The equivalent Lua program, `a={[1000000000]=1}`, creates a table with a single entry.

Until Lua 4.0, tables were implemented strictly as hash tables: all pairs were explicitly stored. Lua 5.0 brought a new algorithm to optimize the use of tables as arrays: it optimizes pairs with integer keys by not storing the keys and storing the values in an actual array. More precisely, in Lua 5.0, tables are implemented as hybrid data structures: they contain a hash part and an array part. Figure 2 shows a possible configuration for a table with the pairs `"x"` $\rightarrow 9.3$, $1 \rightarrow 100$, $2 \rightarrow 200$, $3 \rightarrow 300$. Note the array part on the right: it does not store the integer keys. This division is made only at a low implementation level; access to table fields is transparent, even to the virtual machine. Tables automatically and dynamically adapt their two parts according to their contents: the array part tries to store the values corresponding to integer keys from 1 to some limit n. Values corresponding to non-integer keys or to integer keys outside the array range are stored in the hash part.

When a table needs to grow, Lua recomputes the sizes for its hash part and its array part. Either part may be empty. The computed size of the array part is the largest n such that at least half the slots between 1 and n are in use (to avoid wasting space with sparse arrays) and there is at least one used slot between $n/2+1$ and n (to avoid a size n when $n/2$ would do). After computing the new sizes, Lua creates the new parts and re-inserts the elements from the old parts into the new ones. As an example, suppose that `a` is an empty table; both its array part and hash part have size zero. If we execute `a[1]=v`, the table needs to grow to accommodate the new key. Lua will choose $n = 1$ for the size of the new array part (with a single entry $1 \rightarrow v$). The hash part will remain empty.

This hybrid scheme has two advantages. First, access to values with integer keys is faster because no hashing is needed. Second, and more important, the array part takes roughly half the memory it would take if it were stored in the hash part, because the keys are implicit in the array part but explicit in the hash part. As a consequence, if a table is being used as an array, it performs as an array, as long as its integer keys are dense. Moreover, no memory or time penalty is paid for the hash part, because it does not even exist. The converse holds: if the table is being used as an associative array, and not as an array, then the array part is likely to be empty. These memory savings are important because it is common for a Lua program to create many small tables, for instance when tables are used to implement objects.

The hash part uses a mix of chained scatter table with Brent's variation [3]. A main invariant of these tables is that if an element is not in its main position (i.e., the original position given by its hash value), then the colliding element is in its own main position. In other words, there are collisions only when two elements have the same main position (i.e., the same hash values for that table size). There are no secondary collisions. Because of that, the load factor of these tables can be 100% without performance penalties.

5 Functions and Closures

When Lua compiles a function it generates a *prototype* containing the virtual machine instructions for the function, its constant values (numbers, literal strings, etc.), and some debug information. At run time, whenever Lua executes a `function...end` expression, it creates a new *closure*. Each closure has a reference to its corresponding prototype, a reference to its *environment* (a table wherein it looks for global variables), and an array of references to *upvalues*, which are used to access outer local variables.

The combination of lexical scoping with first-class functions creates a well-known difficulty for accessing outer local variables. Consider the example in Figure 3. When `add2` is called, its body accesses the outer local variable `x` (function parameters in Lua are local variables). However, by the time `add2` is called, the function `add` that created `add2` has already returned. If `x` was created in the stack, its stack slot would no longer exist.

Most procedural languages avoid this problem by restricting lexical scoping (e.g., Python), not providing first-class functions (e.g., Pascal), or both (e.g., C). Functional languages do not have those restrictions. Research in non-pure functional languages like Scheme and ML has created a vast body of knowledge about compilation techniques for closures (e.g., [19, 1, 21]).[3] However, those works do

[3] The techniques used in pure functional languages, such as Haskell, are usually not applicable to procedural languages.

not try to limit the complexity of the compiler. For instance, just the control-flow analysis of Bigloo, an optimizer Scheme compiler [20], is more than ten times larger than the whole Lua implementation: The source for module `Cfa` of Bigloo 2.6f has 106,350 lines, versus 10,155 lines for the core of Lua 5.0. As explained in Section 2, Lua needs something simpler.

Lua uses a structure called an *upvalue* to implement closures. Any outer local variable is accessed indirectly through an upvalue. The upvalue originally points to the stack slot wherein the variable lives (Figure 4, left). When the variable goes out of scope, it migrates into a slot inside the upvalue itself (Figure 4, right). Because access is indirect through a pointer in the upvalue, this migration is transparent to any code that reads or writes the variable. Unlike its inner functions, the function that declares the variable accesses it as it accesses its own local variables: directly in the stack.

Mutable state is shared correctly among closures by creating at most one upvalue per variable and reusing it as needed. To ensure this uniqueness, Lua keeps a linked list with all open upvalues (that is, those that still point to the stack) of a stack (the *pending vars* list in Figure 4). When Lua creates a new closure, it goes through all its outer local variables. For each one, if it can find an open upvalue in the list, it reuses that upvalue. Otherwise, Lua creates a new upvalue and links it in the list. Notice that the list search typically probes only a few nodes, because the list contains at most one entry for each local variable that is used by a nested function. Once a closed upvalue is no longer referred by any closure, it is eventually garbage collected.

It is possible for a function to access an outer local variable that does not belong to its immediately enclosing function, but to an outer function. In that case, even by the time the closure is created, the variable may no longer exist in the stack. Lua solves this case by using *flat closures* [5]. With flat closures, whenever a function accesses an outer variable that is not local to its enclosing function, the variable also goes to the closure of the enclosing function. Thus, when a function is instantiated, all variables that go into its closure are either in the enclosing function's stack or in the enclosing function's closure.

```
function add (x)                  add2 = add(2)
  return function (y)             print(add2(5))
           return x+y
         end
end
```

Figure 3: Access to outer local variables

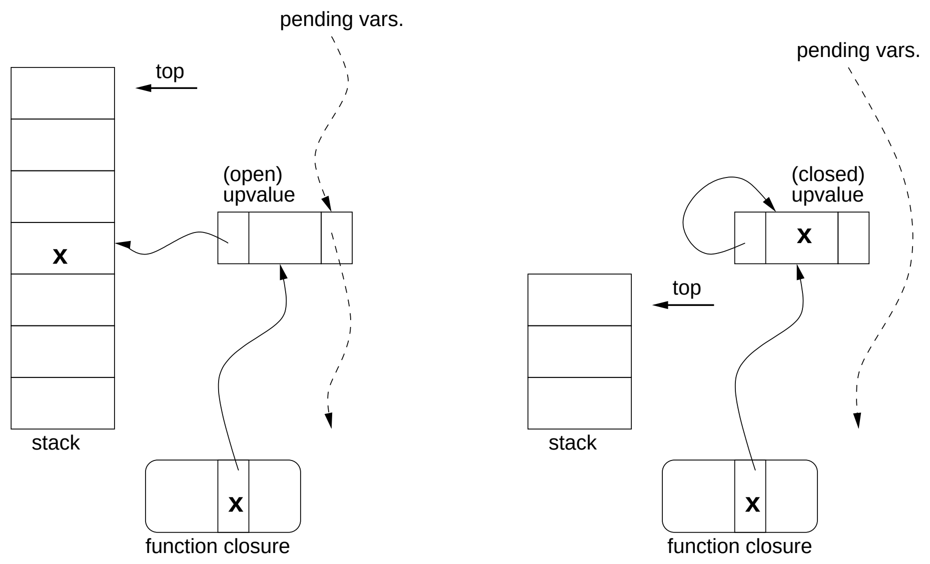

Figure 4: An upvalue before and after being "closed".

6 Threads and Coroutines

Since version 5.0, Lua implements *asymmetric coroutines* (also called *semi-symmetric coroutines* or *semi-coroutines*) [7]. Those coroutines are supported by three functions from the Lua standard library: `create`, `resume`, and `yield`. (These functions live in the `coroutine` namespace.) The `create` function receives a "main" function and creates a new coroutine with that function. It returns a value of type *thread* that represents that coroutine. (Like all values in Lua, coroutines are first-class values.) The `resume` function (re)starts the execution of a given coroutine, calling its main function. The `yield` function suspends the execution of the running coroutine and returns the control to the call that resumed that coroutine.

Conceptually, each coroutine has its own stack. (Concretely, each coroutine has two stacks, as we shall discuss in Section 7, but we can consider them as a single abstract stack.) Coroutines in Lua are *stackful*, in the sense that we can suspend a coroutine from inside any number of nested calls. The interpreter simply puts aside the entire stack for later use and continues running on another stack. A program can restart any suspended coroutine at will. The garbage collector collects stacks whose coroutines are no longer accessible.

The combination of stackfulness and first-class status makes coroutines, as implemented in Lua, is equivalent to one-shot continuations. As such, they allow programmers to implement several advanced control mechanisms, such as cooperative multithreading, generators, symmetric coroutines, backtracking, etc. [7].

A key point in the implementation of coroutines in Lua is that the interpreter cannot use its internal C stack to implement calls in the interpreted code. (The Python community calls an interpreter that follows that restriction a *stackless* interpreter [23].) When the main interpreter loop executes a call operation, it creates a new slot in the stack, adjusts several pointers, and continues the loop with the instructions of the called function. Similarly, a return operation removes the top stack slot, adjusts pointers, and continues the loop with the instructions of the calling function. Not by coincidence, that is exactly what a real CPU does to perform function calls.

When the interpreter executes a resume, however, it does a recursive call to the main interpreter function. This new invocation is responsible for executing the resumed coroutine, using the coroutine stack to perform calls and returns. When this new loop executes an yield, it returns to the previous interpreter invocation, leaving the coroutine stack with any pending calls. In other words, Lua uses the C stack to keep track of the stack of active coroutines at any given time. Each yield returns to the previous interpreter loop, which is the one that called the corresponding resume.

A source of difficulties in the implementation of coroutines in some languages is how to handle references to outer local variables. Because a function running in a coroutine may have been created in another coroutine, it may refer to variables in a different stack. This leads to what some authors call a *cactus* structure [18]. The use of flat closures, as we discussed in Section 5, avoids this problem altogether.

7 The Virtual Machine

Lua runs programs by first compiling them into instructions ("opcodes") for a virtual machine and then executing those instructions. For each function that Lua compiles it creates a *prototype*, which contains an array with the opcodes for the function and an array of Lua values (`TObjects`) with all constants (literal strings and numerals) used by the function.

For ten years (since 1993, when Lua was first released), Lua used a stack-based virtual machine, in various incarnations. Since 2003, with the release of Lua 5.0, Lua uses a register-based virtual machine. This register-based machine also uses a stack, for allocating activation records, wherein the registers live. When Lua enters a function, it preallocates from the stack an activation record large enough to hold all the function registers. All local variables are allocated in registers. As a consequence, access to local variables is specially efficient.

Register-based code avoids several "push" and "pop" instructions that stack-based code needs to move values around the stack. Those instructions are particularly expensive in Lua, because they involve the copy of a tagged value, as discussed in Section 3. So, the register architecture both avoids excessive copying of values and reduces the total number of instructions per function. Davis et al. [6] argue in defense of register-based virtual machines and provide hard data on the improvement of Java bytecode. Some authors also defend register-based virtual machines based on their suitability for on-the-fly compilation (see [24], for instance).

There are two problems usually associated with register-based machines: code size and decoding overhead. An instruction in a register machine needs to specify its operands, and so it is typically larger than a corresponding instruction in a stack machine. (For instance, the size of an instruction in Lua's virtual machine is four bytes, while the size of an instruction in several typical stack machines, including the ones previously used by Lua, is one or two bytes.) On the other hand, register machines generate less opcodes than stack machines, so the total code size is not much larger.

Most instructions in a stack machine have implicit operands. The corresponding instructions in a register machine must decode their operands from the instruction. Such decoding adds overhead to the interpreter. There are several factors that ameliorate this overhead. First, stack machines also spend some time manipulating implicit operands (e.g., to increment or decrement the stack top). Second, because in a register machine all operands are inside the instruction and the instruction is a machine word, the operand decoding involves only cheap operations, such as logical operations. Moreover, instructions in stack machines frequently need multi-byte operands. For instance, in the Java VM, goto and branch instructions use a two-byte displacement. Due to alignment, the interpreter cannot fetch such operands at once (at least not with portable code, where it must always assume worst-case alignment restrictions). On a register machine, because the operands are inside the instruction, the interpreter does not have to fetch them independently.

There are 35 instructions in Lua's virtual machine. Most instructions were chosen to correspond directly to language constructs: arithmetic, table creation and indexing, function and method calls, setting variables and getting values. There is also a set of conventional jump instructions to implement control structures. Figure 5 shows the complete set, together with a brief summary of what each instruction does, using the following notation: `R(X)` means the `X`th register. `K(X)` means the `X`th constant. `RK(X)` means either `R(X)` or `K(X-`k`)`, depending on the value of `X` — it is `R(X)` for values of `X` smaller than k (a build parameter, typically 250). `G[X]` means the field `X` in the table of globals. `U[X]` means the `X`th upvalue. For a detailed discussion of Lua's virtual machine instructions, see [14, 22].

Registers are kept in the run-time stack, which is essentially an array. Thus,

```
MOVE        A B     R(A) := R(B)
LOADK       A Bx    R(A) := K(Bx)
LOADBOOL    A B C   R(A) := (Bool)B; if (C) PC++
LOADNIL     A B     R(A) := ... := R(B) := nil
GETUPVAL    A B     R(A) := U[B]
GETGLOBAL   A Bx    R(A) := G[K(Bx)]
GETTABLE    A B C   R(A) := R(B)[RK(C)]
SETGLOBAL   A Bx    G[K(Bx)] := R(A)
SETUPVAL    A B     U[B] := R(A)
SETTABLE    A B C   R(A)[RK(B)] := RK(C)
NEWTABLE    A B C   R(A) := {} (size = B,C)
SELF        A B C   R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD         A B C   R(A) := RK(B) + RK(C)
SUB         A B C   R(A) := RK(B) - RK(C)
MUL         A B C   R(A) := RK(B) * RK(C)
DIV         A B C   R(A) := RK(B) / RK(C)
POW         A B C   R(A) := RK(B) ^ RK(C)
UNM         A B     R(A) := -R(B)
NOT         A B     R(A) := not R(B)
CONCAT      A B C   R(A) := R(B) .. ... .. R(C)
JMP         sBx     PC += sBx
EQ          A B C   if ((RK(B) == RK(C)) ~= A) then PC++
LT          A B C   if ((RK(B) <  RK(C)) ~= A) then PC++
LE          A B C   if ((RK(B) <= RK(C)) ~= A) then PC++
TEST        A B C   if (R(B) <=> C) then R(A) := R(B) else PC++
CALL        A B C   R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
TAILCALL    A B C   return R(A)(R(A+1), ... ,R(A+B-1))
RETURN      A B     return R(A), ... ,R(A+B-2)      (see note)
FORLOOP     A sBx   R(A)+=R(A+2); if R(A) <?= R(A+1) then PC+= sBx
TFORLOOP    A C     R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
TFORPREP    A sBx   if type(R(A)) == table then R(A+1):=R(A), R(A):=next;
SETLIST     A Bx    R(A)[Bx-Bx%FPF+i] := R(A+i), 1 <= i <= Bx%FPF+1
SETLISTO    A Bx
CLOSE       A       close stack variables up to R(A)
CLOSURE     A Bx    R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))
```

Figure 5: The instructions in Lua's virtual machine

access to registers is fast. Constants and upvalues are stored in arrays and so access to them is also fast. The table of globals is an ordinary Lua table. It is accessed via hashing but with good performance, because it is indexed only with strings (corresponding to variable names), and strings pre-compute their hash values, as mentioned in Section 2.

The instructions in Lua's virtual machine take 32 bits divided into three or four fields, as shown in Figure 6. The `OP` field identifies the instruction and takes 6 bits. The other fields represent operands. Field `A` is always present and takes 8 bits. Fields `B` and `C` take 9 bits each. They can be combined into an 18-bit field: `Bx` (unsigned) and `sBx` (signed).

Most instructions use a three-address format, where `A` points to the register

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

OP	A	B	C
OP	A	Bx	
OP	A	sBx	

Figure 6: Instruction layout

```
function max (a,b)
  local m = a           1 MOVE     2 0 0   ; R(2) = R(0)
  if b > a then         2 LT       0 0 1   ; R(0) < R(1) ?
    m = b               3 JMP      1       ; to 5 (4+1)
  end                   4 MOVE     2 1 0   ; R(2) = R(1)
  return m              5 RETURN   2 2 0   ; return R(2)
end                     6 RETURN   0 1 0   ; return
```

Figure 7: Bytecode for a Lua function

that will hold the result and `B` and `C` point to the operands, which can be either a register or a constant (using the representation `RK(X)` explained above). With this format, several typical operations in Lua can be coded in a single instruction. For instance, the increment of a local variable, such as `a = a + 1`, is coded as `ADD` x x y, where x represents the register holding the local variable and y represents the constant 1. An assignment like `a = b.f`, when both `a` and `b` are local variables, is also coded as the single instruction `GETTABLE` x y z, where x is the register for `a`, y is the register for `b`, and z is the index of the string constant `"f"`. (In Lua, the syntax `b.f` is syntactic sugar for `b["f"]`, that is, `b` indexed by the string `"f"`.)

Branch instructions pose a difficulty because they need to specify two operands to be compared plus a jump offset. Packing all this data inside a single instruction would limit jump offsets to 256 (assuming a signed 9-bit field). The solution adopted in Lua is that, conceptually, a test instruction simply skips the next instruction when the test fails; this next instruction is a regular jump, which uses an 18-bit offset. Actually, because a test instruction is always followed by a jump instruction, the interpreter executes both instructions together. That is, when executing a test instruction that succeeds, the interpreter immediately fetches the next instruction and does the jump, instead of doing it in the next dispatch cycle. Figure 7 shows an example of Lua code and the corresponding bytecode. Note the structure of the conditional and jump instructions just described.

Figure 8 shows a small sample of the optimizations performed by the Lua

```
local a,t,i              1: LOADNIL  0 2 0
a=a+i                    2: ADD      0 0 2
a=a+1                    3: ADD      0 0 250 ; 1
a=t[i]                   4: GETTABLE 0 1 2
```

Figure 8: Register-based opcode (Lua 5.0)

```
local a,t,i              1: PUSHNIL    3
a=a+i                    2: GETLOCAL   0     ; a
                         3: GETLOCAL   2     ; i
                         4: ADD
                         5: SETLOCAL   0     ; a
a=a+1                    6: GETLOCAL   0     ; a
                         7: ADDI       1
                         8: SETLOCAL   0     ; a
a=t[i]                   9: GETLOCAL   1     ; t
                        10: GETINDEXED 2     ; i
                        11: SETLOCAL   0     ; a
```

Figure 9: Stack-based opcode (Lua 4.0)

compiler. Figure 9 shows the same code compiled for Lua 4.0, which used a stack-based virtual machine with 49 instructions. Note how the switch to a register-based virtual machine allowed the generation of much shorter code. Each executable statement in this example compiles to a single instruction in Lua 5.0, but needs three or four instructions in Lua 4.0.

For function calls, Lua uses a kind of *register window*. It evaluates the call arguments in successive registers, starting with the first unused register. When it performs the call, those registers become part of the activation record of the called function, which therefore can access its parameters as regular local variables. When this function returns, those registers are put back into the activation record of the caller.

Lua uses two parallel stacks for function calls. (Actually, each coroutine has its own pair of stacks, as we discussed in Section 6.) One stack has one entry for each active function. This entry stores the function being called, the return address when the function does a call, and a base index, which points to the activation record of the function. The other stack is simply a large array of Lua values that keeps those activation records. Each activation record keeps all temporary values of the function (parameters, local variables, etc.). Actually, we can see each entry in the second stack as a variable-size part of a corresponding entry in the first stack.

8 Conclusion

In this paper we have presented the most innovative aspects of the implementation of Lua 5.0: its register-based virtual machine, the new algorithm for optimizing tables used as arrays, and the implementation of closures.

To our knowledge, Lua is the first language in wide use to adopt a register-based virtual machine. The optimization for tables allows a table to be partially implemented as an array when it is used that way (that is, when it has enough keys in a range $1 \ldots n$). Its implementation of closures is also unique, combining the use of an array-based stack with lexically scoped first-order functions, without complex control-flow analysis.

program	Lua 4.0	Lua 5'	Lua 5.0
sum (2e7)	1.23	0.54 (44%)	0.54 (44%)
fibo (30)	0.95	0.68 (72%)	0.69 (73%)
ack (8)	1.00	0.86 (86%)	0.88 (88%)
random (1e6)	1.04	0.96 (92%)	0.96 (92%)
sieve (100)	0.93	0.82 (88%)	0.57 (61%)
heapsort (5e4)	1.08	1.05 (97%)	0.70 (65%)
matrix (50)	0.84	0.82 (98%)	0.59 (70%)

Figure 10: Benchmarks (times in seconds; percentages are relative to Lua 4.0)

The table in Figure 10 shows some simple performance comparisons between the old implementation and the new one. The tests were run on an Intel Pentium IV machine with 512 Mbytes running Linux 2.6, with Lua compiled with gcc 3.3. Lua 4.0 uses neither the register-based virtual machine (its machine is stack based) nor the table–array optimization. Lua 5' is Lua 5.0 without table–array optimization, tail calls, and dynamic stacks (related to coroutines); Lua 5' is essentially Lua 4.0 with the new register-based virtual machine.

We took all test cases from *The Great Computer Language Shootout* [2], except the first one (*sum*), which is a simple loop to add all integers from 1 to n. This first test spends most of its time in the virtual machine; it shows that the new virtual machine can be more than twice as fast as the old one. The other tests spend more time in other tasks (function calls, table/array access, etc.), so the gain in the virtual machine has a smaller effect on the total time. In the tests that use arrays (*sieve*, *heapsort*, and *matrix*), the combination of the new virtual machine with the new optimization for arrays can reduce the running time up to 40%.

The complete code of Lua 5.0 is available for browsing at Lua's web site: `http://www.lua.org/source/5.0/`.

Acknowledgments

Edgar Toernig provided a key insight into the implementation of closures. Thatcher Ulrich made a previous implementation of coroutines in Lua 4.0 that worked as a proof-of-concept for our implementation in Lua 5.0. The authors are partially supported by grants from CNPq (grants 302608/2002-8, 300392/2003-6, and 401109/2003-8), FINEP (CT-INFO 01/2003), and Microsoft Research (2nd Rotor RFP).

References

1. A. W. Appel. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.
2. D. Bagley. The great computer language shootout. `http://www.bagley.org/~doug/shootout/`.
3. R. P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.
4. A. Calpini. The great Win32 computer language shootout. `http://dada.perl.it/shootout/`.
5. L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217, 1984.
6. B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49. ACM Press, 2003.
7. A. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.
8. M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, Nov. 2003.
9. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.
10. R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, 1986.
11. R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL 4 Programming Language.* Prentice-Hall, 1971.
12. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: A history of Lua. In *Proceedings of V Brazilian Symposium on Programming Languages*, pages B–14–B–28, 2001.
13. S. C. Johnson. YACC: Yet another compiler compiler. CS TR 32, Bell Labs, July 1975.
14. K.-H. Man. A no-frills introduction to Lua 5 VM instructions. `http://luaforge.net/docman/?group_id=83`.
15. S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter.* Ellis Horwood, 1982.

16. I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, Montreal, Canada, June 1998.
17. A. Randal, D. Sugalski, and L. Toetsch. *Perl 6 and Parrot Essentials*. O'Reilly, second edition, 2004.
18. M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.
19. M. Serrano. Control flow analysis: a functional language compilation paradigm. In *10th ACM Symposium on Applied Computing*, pages 118–122, Nashville, TN, Feb. 1995.
20. M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *2nd Static Analysis Symposium*, pages 366–381, Glasgow, Scotland, Sept. 1995. LNCS 983.
21. Z. Shao and A. W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, pages 150–161, June 1994.
22. Z. A. Shaw. The Lua virtual machine. `http://www.zedshaw.com/writings/luas-lvm-instructions/luas_lvm_instructions/luas_lvm_instructions.html`.
23. C. Tismer. Continuations and stackless Python. In *Proceedings of the 8th International Python Conference*, Arlington, VA, 2000.
24. P. Winterbottom and R. Pike. The design of the Inferno virtual machine. `http://www.herpolhode.com/rob/hotchips.html`.