LuaJITTEX

Luigi Scarso

Abstract

Here we introduce LuaJITTEX, an implementation of LuaTEX that uses LuaJIT 2.0 instead of Lua 5.1 .

1 Introduction

On Thursday, November 8, 2012 the long-awaited release of LuaJIT 2.0 finally happened, after 11 beta releases over three years. It happened a month after EuroTEX 2012 & 6th ConTEXt meeting, where I and Hans Hagen discussed the possibility of building a set of *bindings* to shared libraries of general interest for LuaTEX. A binding is an object module that acts as a bridge between a specific library and the Lua interpreter of LuaTEX. Its role is to expose the library to the point of view of Lua(TEX), thus easing the job of the programmer. The module is specific to the library, and its source code must be created in some way. A manual binding is feasible only for a small library; with a large library, it's better to make use of dedicated tools. Each tool has its pros and cons, but we have found that SWIG [11] can satisfy our needs, having used it before (see [18] and [19]). Its syntax is quite similar to C and it can parse the header files of a library and automatically produce the source code of the binding. Usually the compilation of the module is also straightforward.

In a TEX project we would like to satisfy the requirements of several platforms, each one with its own toolchain. A candidate library is not always available for a target platform, or complete support for a toolchain may be lacking (we need at least a compiler, assembler and linker). A library can use some "dirty tricks" that are hard to translate into a binding module and extensive testing can become prohibitive. Last but not least, during the meeting we also considered the consequences on the upcoming transition from Lua 5.1 to Lua 5.2 in LuaTEX.

When we explore some ideas, sometimes we fall into what looks like an unsolvable problem, and it's a good strategy to temporarily change focus and start a completely different activity and return after a while to the original problem with a fresh point of view. As it happens, I had such a problem with the binding of a function with a variable argument list (in an apparently absurd attempt to bind the libc library of Microsoft Windows 7), and the new release of LuaJIT offered a good reason to momentarily drop this task and start to see if it was possible to replace the Lua interpreter with a LuaJIT one. But before I go on, it's necessary to understand what exactly LuaJIT is and why such a substitution may be interesting. I will first discuss the TEX and Lua interpreters.

2 TEX, Lua, LuaJIT

2.1 TEX

LuaTEX is the union of two interpreters, one of the Lua language and one for the TEX language. The main actor is the TEX interpreter [12]: the input processor scans each input line of the source producing a pair `(character-code, category-code)`. LuaTEX currently has 2^{21} `character-code`s and 16 `category-code`s. Starting from a pair, a character token, a control sequence token, or a parameter token is formed; there are currently around 350 subtypes of tokens. With an abuse of terminology, we can call this subtype an *operation code* (opcode), and hence an opcode fits into two bytes. A token is then executed by mean of a `jump_table` using a *function pointer*: `(jump_table[opcode])()` calls the function that implements `opcode`. The statement `while (1)` means that this task continues until the variable `main_control_state` has the value `goto_return`, that signals to exit from the main loop and end the program:

```
void main_control(void)
{
  main_control_state = goto_next;
  init_main_control();
  if (equiv(every_job_loc) != null)
     begin_token_list(equiv(every_job_loc),
                      every_job_text);
  while (1) {
     if (main_control_state == goto_skip_token)
       main_control_state = goto_next;
     else
       get_x_token();

     if (interrupt != 0 && OK_to_interrupt) {
       back_input();
       check_interrupt();
       continue;
     }
     if (int_par(tracing_commands_code)> 0)
       show_cur_cmd_chr();

     (jump_table[(abs(mode)+cur_cmd)])();
     if (main_control_state == goto_return) {
       return;
     }
  }
  return;
}
```

This kind of interpreter is called a Syntax-Directed Interpreter because it mimics what we do when we trace the code manually. It is well suited for a DSL (Domain Specific Language, see [16]) as is

TEX in this case, but usually a DSL is not so complex as TEX (which is also Turing-complete). The main part of this kind of interpreter is usually a big `switch-case` statement, where each opcode has its own `case`. The C standards do not specify how to implement a `switch` statement, but usually a compiler can use `jump_table` only if each label is equal to the preceding label plus one (or if the compiler is able to bring values of the labels to an equivalent case, see [1], section 7.12 Branches and switch statements); if the values are far from each other, a compiler must implement it as a kind of binary search among `if-then-else` like statements, and this has a bad impact on the branch predictor of the CPU. For example, let's consider this `switch` fragment of C code:

```
switch (OPCODE) {
 case 0: func_000(); break;
 case 1: func_001(); break;
 case 2: func_002(); break;
 default:            break;
}
```

A compiler, maybe with some kind of optimization enabled, can generate machine code corresponding to the following pseudo-code (not C):

```
static address jump_table[] =
    {case_0,case_1,case_2,end };
if (index > 2)      goto end;
goto jump_table[index];
case_0: func_000(); goto end;
case_1: func_001(); goto end;
case_2: func_b();
end:
```

which is more efficient than multiple `if-then-else`. Note that this `jump_table` is not the same as we've seen for TEX: there we use a label to run a function in the compiled code that refers to a `jump_table` being a function pointer table defined earlier in the source code, and once compiled it adds overhead due to the call of the selected function. On the other hand, the function pointer method definitely avoids the `if-then-else` like statements (because it's a choice made by the programmer, not the compiler) and makes the code more compact and more manageable. More on this later.

It is known that in order to speedup the loading of a large set of macros, TEX (or, more exactly, iniTEX, a special version of TEX) can also *dump* the macros into a kind of memory-compiled format (simply called *format*), which can be loaded at runtime. A format depends on the current release of the interpreter and the machine on which the interpreter runs: a format cannot generally be exchanged between different releases and formats cannot always be exchanged between different machines even with the same release — but this currently fails only if a format uses floating point values, because floating point numbers related to glue are stored in the format and hence will generally not be readable across platforms. (See [2]: LATEX, for example, doesn't use glue values in the format and hence the result `.fmt` is portable, thankfully.)

This is not seen as a penalty; because TEX is used as *document compiler* speed is a concern, and a format can give a speedup of several orders of magnitude, so it's typically built when TEX is installed. TEX users are generally more interested in durability/portability of their document source code, not the format. A remarkable exception is ConTEXt Mark IV, but its users find it natural to rebuild the format on every update of the code — which happens quite often, because it's still evolving. Dumping a format is also an uncommon characteristic for a DSL language.

2.2 Lua

The Lua interpreter is designed in a different way: it first translates the source code into another form and then executes this form. The translation is called *compiling into bytecode* because it's similar to the task of a compiler, which translates source code (like a C program) into machine code. While a compiler usually translates a source program into an intermediate representation which is optimized and then translated again into a machine code, a Lua interpreter directly translates the source into bytecode — but even in this case some optimization is possible [15].

Like TEX, Lua can dump a module into a kind of "format" (called the bytecompiled version of the module) and this "format" can be exchanged between different machines with the same architecture and the same interpreter (i.e. the same major and minor version number). The reason for this is that the Lua interpreter has a kind of "software CPU" called *Virtual Machine* (VM), which is implemented in ANSI C and it is the same for all the same release of the interpreter. The name "bytecode" is not casual: each opcode of the instructions of the VM fits into one byte (while the size of an instruction is 32 bit) and a VM is nothing else than a bytecode interpreter (after all we can also see a physical CPU as a machine-code interpreter).

A bytecode interpreter is usually the best choice if we want to implement a general programming language and we also want a fast and portable interpreter (see [16], chapter 10). The reason is that the design of a general programming language is more

complex than a simple DSL, but the theory of compilers is a powerful tool that can help enormously — we have only to avoid producing machine code. In fact any specific CPU is its own machine, and to try to adapt the compiler to each CPU is in conflict with the portability across different architectures. A better solution is to design a byte-compiler for a VM — this task is common for all platforms, gaining in portability — and implement the VM with a high level and widely available language like C. A VM is usually simpler than a physical CPU, so the byte-compiler can be optimized for performance — the translation must be fast; this means that the code can be complex and hence its design and implementation can require more effort compared to a DSL. This is why the bytecode is also important: we can use a cache to avoid re-parsing the source language. Of course a VM must be also fast, otherwise the interpreter of the general language is slow.

There are two ways to implement a VM: simulating a *stack* (stack-based VM) and simulating a *register machine* (register-based VM). A register-based VM is similar to a real piece of hardware, because it uses simulated general-purpose registers, but has no practical limits on their number as a real CPU does. A stack-based VM doesn't have to figure out which register to use for which value, because instructions have implicit operands. Stack-based VMs are thus easy to implement, but register-based implementations better optimize the use of the registers of the physical CPU, which is the fastest memory available (300 times faster than DRAM), but is also very limited in size (typically not more than 1000 bytes, vs. a typical 4 GBytes of DRAM). Lua is the first widely used language to have a register-based VM ([15], section Introduction).

Let's see for example how `a=1;b=2;c=a+b` is translated in bytecode by `luac`, the Lua bytecode compiler (text after `;` is a comment):

```
SETTABUP 0 -1 -2 ; _ENV "a" 1
SETTABUP 0 -3 -4 ; _ENV "b" 2
GETTABUP 0 0 -1  ; _ENV "a"
GETTABUP 1 0 -3  ; _ENV "b"
ADD      0 0 1
SETTABUP 0 -5 0  ; _ENV "c"
RETURN   0 1
```

Expanding the meaning of the opcodes we have:

```
SETTABUP 0 -1 -2 ; UpValue[0][RK(-1)] := RK(-2)
SETTABUP 0 -3 -4 ; UpValue[0][RK(-3)] := RK(-4)
GETTABUP 0 0 -1  ; R(0) := UpValue[0][RK(-1)]
GETTABUP 1 0 -3  ; R(1) := UpValue[0][RK(-3)]
ADD      0 0 1   ; R(0) := RK(0) + RK(1)
SETTABUP 0 -5 0  ; UpValue[0][RK(-5)] := RK(0)
RETURN   0 1     ; return R(0),R(-1)
```

`UpValue[0]` is the current environment, while `R(.)` is a register and `RK(.)` is a register or a constant: the access can be relative. So,

```
UpValue[0][RK(-1)] := RK(-2)
```

means "in the current environment, set `RK[-1]` (i.e. `"a"`) to `RK[-2]` (i.e. `1`)."

Each of these instructions is executed by the bytecode interpreter, which is, perhaps a bit surprisingly, a big `switch-case` loop (here we show a fragment):

```
while (1) {
...
 switch (op) {
  case OPR_AND: {
    luaK_goiftrue(fs, v);
    break;
  }
  case OPR_OR: {
    luaK_goiffalse(fs, v);
    break;
  }
  case OPR_CONCAT: {
    luaK_exp2nextreg(fs, v);
    break;
  }
  case OPR_ADD:
  case OPR_SUB:
  case OPR_MUL:
  case OPR_DIV:
  case OPR_MOD:
  case OPR_POW: {
    if (!isnumeral(v)) luaK_exp2RK(fs, v);
    break;
  }
  ...
 }/* end switch */
 ...
}/* end while */
```

Currently most bytecode interpreters use the *threading model* technique, where *each instruction is the address of the case target code*. This is similar to, but not the same as, what we have seen for the TEX interpreter.

To explain exactly what this means, let's first remember that C has a `goto` statement that transfers the program flow to a point marked by a label, i.e.

```
goto somelabel;
...
somelabel:
/* some  code */
```

The address marked by `somelabel` is fixed at compile time, but some compilers (notably GCC) allow us to store the address of `somelabel` into an array to be used with a `goto`, using the label as a value so that the `goto` *is computed at runtime*:

```
static void *array[] = { &&somelabel };
...
/* equivalent to goto somelabel */
goto *array[0];
...
somelabel:
/* code */
```

Such labels as values are valid only within a function: computed goto cannot be used to jump to code in a different function. (Computed goto for GCC is described in [3].) Hence in this case labels are not truly first-class values, i.e. values that can be dynamically created, destroyed or passed as an argument. In contrast, in Lua all types (nil, boolean, number, string, table, function, userdata, thread) are first-class values and version 5.2 of Lua adds the `goto` statement too.

In this way it could be possible to replace the `switch-case` statement storing the bytecoded instruction of the program with an array `instruction` and counting the next instruction with a *program counter* `pc`, as in the following pseudo-code:

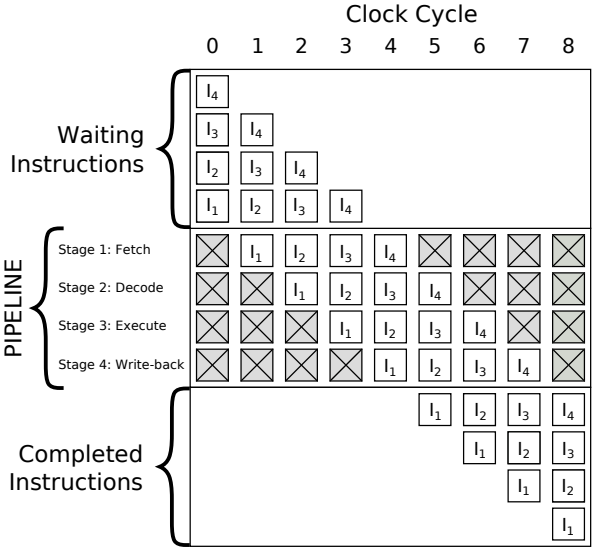
```
static void* dispatch_table[] = {
        ...
        &&OPR_AND,
        &&OPR_OR,
        &&OPR_CONCAT,
        &&OPR_ADD,
        ...};

#define DISPATCH() \
  goto *dispatch_table[instruction[pc++]]

int pc = 0;
while (1) {
  ...
  OPR_AND:
    luaK_goiftrue(fs, v);
    DISPATCH();
  }
  OPR_OR: {
    luaK_goiffalse(fs, v);
    DISPATCH();
  }
  OPR_CONCAT: {
    luaK_exp2nextreg(fs, v);
    DISPATCH();
  }
  OPR_ADD: {
    if (!isnumeral(v)) luaK_exp2RK(fs, v);
    DISPATCH();
  }
  ...
}/* end while */
```

There is also another important benefit: computed goto helps the branch predictor of the CPU. To understand the problem, let's consider a real CPU as an interpreter of machine code. At this level, it's still possible to divide the execution of a single instruction into atomic stages: let's call them *fetch* (read an instruction from memory) *decode*, *execute* and *write-back* (write the result into memory) and let's suppose that all the stages take the same time (which is not true in modern CPUs). A clock is mandatory to synchronize the stages, and a simple method (called Single-Cycle) is fetch - decode - execute - write the first instruction (4 cycles), fetch - decode - execute - write the second (again 4 cycles) and so on. If we have 4 instructions then after 16 cycles the overall execution is done, if we suppose that none of the instructions use jumps to other instructions.

But, if each stage is independent the CPU can do a better job: after the fetch of the first instruction, it can start its decode stage and simultaneously the fetch stage of the second instruction, as explained in this picture from [4]:



The CPU can thus complete 4 instructions after 8 cycles, doubling the *throughput*, even if each instruction still takes 4 cycles. This method is called *pipelining*. Modern CPUs can have more than 4 stages: more stages means more cycles, but also simpler circuitry and hence the chance to use a faster clock; but especially more stages mean high throughput. In fact in this case the distance T_{pipeline} (in CPU cycles) between I_1 and I_2 is 1 cycle, while with the Single-Cycle we have $T_{\text{Single-Cycle}} = 4$ cycles and in general for an N-stage pipeline we have at best $T_{\text{pipeline}} = T_{\text{Single-Cycle}}/N$.

Back to the picture: a problem arises if execution of I_1 has as a consequence a jump to I_4, skipping I_2 and I_3 — and this is known only at the execution

stage. Given that the CPU knows that I_1 is a type of jump, a simple solution is to avoid using any stage of the pipeline until I_1 has ended its execute stage; in this case I_1 will end at the 5^{th} cycle and I_4 will end at the 9^{th} cycle — and we have the same performance of a Single-Cycle, because in this case with a Single-Cycle the CPU fetches I_4 at the 5^{th} cycle. This is called a *(control or branch) hazard*.

To reduce the performance impact of this, modern CPUs have specialized hardware that predicts, with a conditional jump, the next instruction to fetch. This *branch prediction* can be a fixed rule ("never take the second choice") or a *dynamic branch prediction*, which is usually based on a branch history table: a small amount of memory indexed by the lower portion of the address of the branch instruction, which contains some bits that say whether that branch was recently taken or not. (For the sake of simplicity, we are not distinguishing between branch predictors (has to decide if a branch condition will fail or not) and branch path predictors (which address to jump to), because often both are on the same circuitry.)

If the branch predictor makes the right choice, the throughput increase is saved; otherwise it has to clean the pipeline and fetch the correct instruction — and this is bad for performance. Nowadays, with an adequate algorithm, it's possible to have from 99% to 82% of correct predictions. For a comprehensive treatment of these subjects see [14] and [17].

The key point in the above is that the prediction is based on the current branch instruction. We have seen that, in the best situation, a `switch-case` is implemented with the `switch` condition used as offset in a look-up table: in the TEX pseudo-code above the crucial line is '`goto jump_table[index];`'. The branch predictor sees this line as a branch instruction and, starting from `index`, it has to choose between *all* the following cases (the branches): it has one base address and n equally-spread branches to choose from, and modern CPUs cannot manage large n efficiently. The line '`if (index > 2) goto end;`' which is mandatory for the `switch` also adds overhead.

With a computed goto the key line is

```
goto *dispatch_table[instruction[pc++]]
```

Each individual `case` becomes the address used by the branch predictor; the branch predictor has n different base addresses and statistically the next instruction is not equally spread between all n choices and hence the address to jump is better predicted.

The difference between these two can be significant: following [5], a branch predictor mispredicts 81%–98% with `switch` and 57%–63% with the threaded model.

So, why doesn't the Lua implementation use the computed goto? The reason is that this extension is not ANSI C, the language chosen to implement Lua. This is clearly explained in [15]: the threading model would compromise the portability of Lua. For example, the Microsoft C compiler doesn't support labels as values, and the Intel ICC compiler supports them under Linux but not under Microsoft Windows.

In the end, this is a good choice at least for TEX, given that also TEX aims to be portable: and if it were a true bottleneck, it should be possible to re-factor the C source of Lua with macros and conditional compilation to choose at compile time the type of the interpreter. And, finally, maybe there will be ANSI C compatible interpreters with lower error rates based on completely different models (after all, a failure rate of 57% is surely high enough to justify further research).

So far we have seen that TEX and Lua use different kinds of interpreter (direct-syntax vs. bytecode), both optimal for their purposes; both have bytecode output, with almost the same issues on portability, though not so relevant for the TEX users (and TEX is slightly better), both use the best choice of main loop of the interpreter, compatible with the portability goal and manageability of the code.

Let's look now at LuaJIT.

2.3 LuaJIT

LuaJIT, by Mike Pall [6], drops the requirement of portability on as many platforms as possible, and changes important parts of the Lua interpreter, keeping compatibility with Lua 5.1 plus other constructs like `goto`. First, LuaJIT still has a bytecode interpreter, but it is written in *assembly language*. It's clear that this immediately leads to the conclusion that there can be (and in fact are) some platforms that are not supported, but we postpone this topic for later.

The way that LuaJIT builds the VM is a bit complex: first, a `buildvm` program is built for the given platform, then `buildvm` uses a `dasc` file (a mix of C and assembly) that describes the physical CPU and emits a `lj_vm.s` assembly file (the VM) that is finally compiled. For an `x86_64` CPU, the `vm_x86.dasc` file looks like this:

```
/* Generate the code for
   a single instruction.  */
static void build_ins(BuildCtx *ctx,
                      BCOp op, int defop)
{
  int vk = 0;
```

```
  |// Note: aligning all instructions
  |//  does not pay off.
  |=>defop:

  switch (op) {
  /* -- Comparison ops --- */
  /* Remember: all ops branch for a true
     comparison, fall through otherwise. */
  |.macro jmp_comp, lt, ge, le, gt, target
  ||switch (op) {
  ||case BC_ISLT:
  |   lt target
  ||break;
  ||case BC_ISGE:
  |   ge target
  ||break;
  ||case BC_ISLE:
  |   le target
  ||break;
  ||case BC_ISGT:
  |   gt target
  ||break;
  ||default: break;  /* Shut up GCC. */
  ||}
  |.endmacro

  case BC_ISLT: case BC_ISGE:
  case BC_ISLE: case BC_ISGT:
    |  // RA = src1, RD = src2,
    |  // JMP with RD = target
    |  ins_AD
    |.if DUALNUM
    |  checkint RA, >7
    |  checkint RD, >8
    |  mov RB, dword [BASE+RA*8]
    |  add PC, 4
    |  cmp RB, dword [BASE+RD*8]
    |  jmp_comp jge, jl, jg, jle, >9
    |6:
    |  movzx RD, PC_RD
    |  branch
 ...
 } /* end main switch */
}
```

It seems that the main loop is still a `switch` statement but LuaJIT under the hood uses a threading model — the same computed goto we saw above. This is possible because assembly language does not have the limitations of the C language, but of course the price to pay is maintaining several different assembly language sources of the same program.

It's important to stress a couple of things: the optimal use of registers (LuaJIT keeps all important variables of the state in registers, and this kind of optimization is hard to achieve with a C compiler, at least for an `x86` CPU) and the size in bytes. An assembly program once compiled is usually smaller than the C counterpart, which means it has a better chance of fitting into cache memory (which is at least two times faster than DRAM). For example, in a compiled version of LuaJITTEX for an `x86_64` CPU the VM is 28560 bytes and it's common to find laptop computers with an L1 cache of 128 KiB with an access time 100 times faster than DRAM. Note that the bytecode is still portable between different LuaJIT VMs (sharing the same version), but it's not compatible with the Lua VM. The interpreter alone is claimed to be from 2 to 4 times faster than the Lua interpreter [7].

The second important feature is that the VM supports translation of the bytecode into machine code at *run time*. This is called *Just In Time compilation* (hence the name LuaJIT), and it uses a *trace* compiler: a compiler that keeps track of frequently used "flat" sequences of bytecodes and only translates the "hot" ones the first time, reusing the machine-code subsequently. The VM and the trace compiler cooperate very closely, but we can describe the operations in four phases (see [7], section 'How a trace compiler works'):

1) *interpretation*: the VM interprets the bytecodes and collects statistics, so that if some code path (i.e. a sequence of Lua statements) reaches a given threshold its *trace* (the relative linear sequence of bytecodes) is considered "hot" and the VM goes on to the next phase;

2) *interpretation and recording*: while continuing the interpretation of bytecode, the VM records the associate actions and translates them into an intermediate representation called *static-single assignment* (SSA), in which each variable is assigned exactly once;

3) *trace compilation*: if the recording is ok (for example all bytecode of the trace can be translated into a SSA), the SSA is optimized and translated into machine code;

4) *trace execution*: the compiled code is executed and *reused* if possible.

It's important to note that even during the trace compilation phase some runtime conditions (e.g. a bound check that fails) can halt execution of the compiled code and return to the standard way of bytecode interpretation, with a loss of performance. Of course we cannot forget the fact that we can have the benefits of a compiled language (high speed of execution) with the benefits of an interpreted one (high speed of development) — the key point of the JIT method.

The last important fact is the support of the *Foreign Function Interface* (FFI) via the Lua module `ffi`. Briefly, this module allows two things:

1) pure Lua code can call external C functions (i.e. functions in external libraries such as `.*dll` and `*.so`); there is a special namespace `ffi.C` that permits using, at least on POSIX systems and Microsoft Windows, the symbols from the current system C library.

2) it's possible *to use C data structures from pure Lua code*: they are compiled to machine code at runtime by the JIT compiler. An example shown at `http://luajit.org/ext_ffi.html` is eloquent: replacing a Lua table with a C `struct` on `x86_64` has a speedup of 110x (i.e. 110 times faster than Lua) and the memory consumption is 64 times less. Apart from the C preprocessor, LuaJIT with the `ffi` module is hence similar (but less powerful) to a C interpreter like `cling` [8] (an interpreter for C++).

3 Building LuaJITTEX and first results

3.1 Building LuaJITTEX

Building LuaJITTEX was a bit of a complicated task, because LuaJIT has its own system to detect the host CPU and build the VM in assembly language, and this system doesn't fit well with the way LuaTEX builds its binaries. After a few tries, we eventually decide to modify the layout of the source code of the original LuaTEX, moving the LuaJIT source to the same level of other support libraries like `png`, `cairo`, `zlib`, `xpdf` and using the original build system of LuaJIT. This seemed reasonable, given that LuaTEX is moving in the same direction (i.e. also move the stock Lua 5.2 to the level of the support libraries), so integration in the future can be easier than now. Some C files (less than ten) also needed to be adapted, but overall, after the change of the layout the integration was quite easy.

We knew that an important point was building LuaJITTEX for several platform with different compilers and checking the performance with a significant Lua code base. Initially the first version was only for Linux 32-bit, then the support for 64-bit was added; after that, we checked the `mingw` 32-bit version, using the same compiler of LuaTEX, but cross-compiling under Linux. After the `mingw` version we started to work on the source of `luatex.exe` from `http://www.w32tex.org/` by Akira Kakuto, which is known to compile with the Microsoft compiler for `x86`. We were able to adapt that source code to LuaJIT as well and compile it with the MS compiler VC 2008 Express edition, again under Linux with Wine.

Compilations in hand, we started a period of testing using the Lua code base of ConTEXt Mark IV. That ended around mid-December 2012, and eventually the LuaJITTEX project was created at `http://foundry.supelec.fr/gf/project/luajittex`; the first release was on Christmas 2012. On 31 December the first version of `luajittex.exe` made by Prof. Kakuto was on the `w32tex` server. Later, in January 2013 we fixed the binary for Mac OS X 64-bit and added support for the compilation with the `clang` compiler. Of course as always testing is welcome: as stated in [9], the choice of compiler can influence the performance, and we need more feedback on this.

3.2 First impressions

Extensive tests were done on the Lua code base of ConTEXt Mark IV, and [13] (in this issue of *TUGboat*) reports numerical results. The first tests show that there was an improvement of speed of about 25%, and, if we decompose into the TEX time and the Lua time, we have measured effectively a 2x speedup of the Lua interpreter. Turning the JIT compilation itself on and off didn't change the results significantly; in fact, with JIT on, LuaJITTEX is a bit slower than with JIT off. Very likely the reason is that few functions of the Lua standard libraries are JIT-compiled (see [10]) and when the JIT compiler sees a Not Yet Implemented (NYI) instruction, it has to jump from the trace compilation phase to the interpretation phase, and this has a cost. And of course, when nothing can be JIT-compiled the analysis is useless overhead.

Given that Mark IV uses the standard libraries and does lots of node-list manipulations it's not a surprise that there is a performance penalty: there is not much to JIT. Thus, the speedup essentially comes from the new VM written in assembly language.

The full power of JIT can be seen with pure Lua or with the math functions; we have also made some quick tests on using the `ffi` module and registered 10x speedups. Of course the price to pay is the loss of garbage collection: using `ffi` we must pay attention to the memory management and how the garbage collector works. We have not checked the calling of external libraries.

The overall impression is that LuaJITTEX is faster than LuaTEX, but not so overwhelmingly fast: in both, the TEX interpreter is still the dominant part. The memory footprint was slightly less in LuaJITTEX.

4 Conclusion

LuaJITTEX looks like the best way to write a high-performance Lua interpreter — it's hard to believe that another implementation could do better without multi-threading. We have consistently measured that the VM is 2 times faster than standard Lua (as in [7])

on an x86 CPU, which shows that LuaJIT is well-adapted to the LuaTEX code base. We also measured a 25% improvement on time, which is probably the best we can achieve modifying only the Lua side.

We think that the JIT compiler and FFI can achieve their full potential only if one starts by writing LuaJIT code from the very beginning; currently LuaJIT is not well-suited to a large standard Lua code base that uses the standard libraries.

Overall, we don't see LuaJITTEX as a potential replacement of LuaTEX, but rather as an engine that can have higher performance in particular situations, for example, an automatic workflow of simple typesetting tasks, especially in the hands of a developer with a good knowledge of the C language and memory management. In this situation, writing a format that is a mix of TEX, Lua and C, together with the ability of LuaJIT to make simple the task of the binding, can make LuaJITTEX a very effective tool.

On the other hand, we cannot hide the potential compatibility issue as LuaTEX moves on to Lua 5.2 and the resulting differences with LuaJIT 2.0 (which currently uses Lua 5.1 plus some constructs from Lua 5.2). We will try to keep LuaJITTEX and LuaTEX in sync as much as possible, but the preference is for LuaTEX, which is the main reference. Users with no particularly demanding tasks are strongly encouraged to use LuaTEX.

Finally, it was very instructive to learn how to set up a toolchain for different compilers, especially for the compilation of luatex.exe. We see this as preparation for the SwigLib project, where one of the challenges will be checking the binaries of the libraries for different platforms.

References

[1] http://www.agner.org/optimize/optimizing_cpp.pdf.

[2] http://tug.org/texinfohtml/web2c.html#Hardware-and-memory-dumps.

[3] http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html.

[4] http://en.wikipedia.org/wiki/Branch_predictor.

[5] http://ftp.complang.tuwien.ac.at/anton/lvas/sem06w/revucky.pdf.

[6] http://luajit.org.

[7] http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[8] http://root.cern.ch/drupal/content/cling.

[9] http://www.complang.tuwien.ac.at/anton/praktika-fertig/schroeder/thesis.pdf.

[10] http://wiki.luajit.org/NYI.

[11] David M. Beazley. SWIG — The Simplified Wrapper and Interface Generator. http://www.swig.org, 1996.

[12] Victor Eijkhout. *TEX by Topic. A TEXnician's Reference*. Addison-Wesley, London, 1991. http://www.eijkhout.net/tbt.

[13] Hans Hagen. ConTEXt: Just in Time LuaTEX. *TUGboat*, 34(1):72–78, 2013.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[15] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005. http://www.jucs.org/jucs_11_7/the_implementation_of_lua.

[16] Terence Parr. *Language Implementation Patterns. Create Your Own Domain-Specific and General Programming Language*. Pragmatic Bookshelf, first edition, 2010.

[17] David A. Patterson and John L. Hennessy. *Computer Organization and Design — The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 4th edition, 2012.

[18] Luigi Scarso. Extending ConTEXt MkIV with PARI/GP. *ArsTEXnica*, 11:65–74, April 2011. http://www.guitex.org/home/images/ArsTeXnica/AT011/AT11-scarso.pdf.

[19] Luigi Scarso. Extending ConTEXt MkIV with GraphicsMagick. *Proceedings of the 5th ConTEXt meeting*, Bassenge, Belgium, 2011. http://meeting.contextgarden.net/2011/talks/day1_05_luigi_graphicmagick/.

(*Links checked on 21 January 2013.*)

◇ Luigi Scarso
luigi dot scarso (at) gmail dot com