ConTEXt: Just-in-time LuaTEX

Hans Hagen

1 Introduction

Reading occasional announcements about LuaJIT,[1] one starts wondering if just-in-time ("jit") compilation can speed up LuaTEX. As a side track of the SwigLib project and after some discussion, Luigi Scarso decided to compile a version of LuaTEX that had the jit compiler as the Lua engine. That's when our journey into jit began.

We started with Linux 32-bit as this is what Luigi used at that time. Some quick first tests indicated that the LuaJIT compiler made ConTEXt MkIV run faster but not that much. Because LuaJIT claims to be much faster than stock Lua, Luigi then played a bit with `ffi`, i.e. mixing C code and Lua, especially data structures. There is indeed quite some speed to gain here; unfortunately, we would have to mess up the ConTEXt code base so much that one might wonder why Lua was used in the first place. I could confirm these observations in a Xubuntu virtual machine in VMware running under 32-bit Windows 8. So, we decided to conduct some more experiments.

A next step was to create a 64-bit binary because the servers at Pragma are KVM virtual machines running a 64-bit OpenSuse 12.1 and 12.2. It took a bit of effort to get a jit version compiled because Luigi didn't want to mess up the regular codebase too much. This time we observed a speedup of about 40% on some runs so we decided to move on to Windows to see if we could observe a similar effect there. And indeed, when we adapted Akira Kakuto's Windows setup a bit we could compile a version for Windows using the native Microsoft compiler. On my laptop a similar speedup was observed, although by then we saw that in practice a 25% speedup was about what we could expect. A bonus is that making formats and identifying fonts is also faster.

So, in that stage, we could safely conclude that LuaTEX combined with LuaJIT made sense if you want a somewhat faster version. But where does the speedup come from? The easiest way to see if jitting has effect is to turn it on and off.

```
jit.on()
jit.off()
```

To our surprise ConTEXt runs are not much influenced by turning the jitter on or off.[2] This means that the improvement comes from other places:

- The virtual machine is a different one, and targets the platforms that it runs on. This means that regular bytecode also runs faster.
- The garbage collector is the one from Lua 5.2, so that can make a difference. It looks like memory consumption is somewhat lower.
- Some standard library functions are recognized and supported in a more efficient way. Think of `math.sin`.
- Some built-in functions like `type` are probably dealt with in a more efficient way.

The third item is an important one. We don't use that many standard functions. For instance, if we need to go from characters to bytes and vice versa, we have to do that for UTF so we use some dedicated functions or LPEG. If in ConTEXt we parse strings, we often use LPEG instead of string functions anyway. And if we still do use string functions, for instance when dealing with simple strings, it only happens a few times.

The more demanding ConTEXt code deals with node lists, which means frequent calls to core LuaTEX functions. Alas, jitting doesn't help much there unless we start messing with `ffi` which is not on the agenda.[3]

2 Benchmarks

Let's look at some of the benchmarks. The first one uses MetaPost and because we want to see if calculations are faster, we draw a path with a special pen so that some transformations have to be done in the code that generates the PDF output. We only show the Windows and 64-bit Linux tests here. The 32-bit tests are consistent with those on Windows so we didn't add those timings here (also because in the meantime Luigi's machine broke down and he moved on to 64 bits).

```
\setupbodyfont[dejavu] % benchmark-1.tex
\starttext
\dontcomplain
\startluacode
  if jit then
    jit.on()
    jit.off()
  end
\stopluacode
```

[1] LuaJIT is written by Mike Pall and more information about it and the technology it uses is at `http://luajit.org`, a site also worth visiting for its clean design.

[2] We also tweaked some of the fine-tuning parameters of LuaJIT but didn't notice any differences. In due time more tests will be done.

[3] If we want to improve these mechanisms it makes much more sense to make more helpers. However, profiling has shown us that the most demanding code is already quite optimized.

```
\startluacode
  statistics.starttiming()
\stopluacode

\dorecurse {10} {
  \dorecurse{1000} {
    \dontleavehmode
    \startMPcode
      for i = 1,100 :
        draw fullcircle
          scaled 10pt withpen pencircle
          xscaled 2 yscaled 4 rotated 20 ;
      endfor ;
    \stopMPcode
    \enspace
  }
  \page
}

\startluacode
  statistics.stoptiming()
  context(statistics.elapsedtime())
\stopluacode
\stoptext
```

The following times are measured in seconds. They are averages of 5 runs. There is a significant speedup but jitting doesn't do much.

	traditional	jit on	jit off
Windows 8	26.0	20.6	20.8
Linux 64	34.2	14.9	14.1

Our second example uses multiple fonts in a paragraph and adds color as well. Although well optimized, font-related code involves node list parsing and a bit of calculation. Color again deals with node lists and the backend code involves calculations but not that many. The traditional run on Linux is somewhat odd, but might have to do with the fact that the MetaPost library suffers from the 64 bits. It is at least an indication that optimizations make less sense if there is a different dominant weak spot. We have to look into this some time.

```
\setupbodyfont[dejavu] % benchmark-2.tex
\starttext \dontcomplain
\startluacode
  if jit then
    jit.on()
    jit.off()
  end
\stopluacode
\startluacode
  statistics.starttiming()
\stopluacode

\dorecurse {1000} {
  {\bf \red   \input tufte } \blank
  {\it \green \input tufte } \blank
  {\tf \blue  \input tufte } \page
}

\startluacode
  statistics.stoptiming()
  context(statistics.elapsedtime())
\stopluacode
\stoptext
```

Again jitting has no real benefits here, but the overall gain in speed is quite nice. It could be that the garbage collector plays a role here.

	traditional	jit on	jit off
Windows 8	54.6	36.0	35.9
Linux 64	46.5	32.0	31.7

This benchmark writes quite a lot of data to the console, which can have impact on performance as TEX flushes on a per-character basis. When one runs TEX as a service this has less impact because in that case the output goes into the void. There is a lot of file reading going on here, but normally the operating system will cache data, so after a first run this effect disappears.[4]

The third benchmark is one that we often use for testing regression in speed of the ConTEXt core code. It measures the overhead in the page builder without special tricks being used, like backgrounds. The document has some 1000 pages.

```
\setupbodyfont[dejavu] % benchmark-3.tex
\starttext \dontcomplain
\startluacode
  if jit then
    jit.on()
    jit.off()
  end
\stopluacode
\startluacode
  statistics.starttiming()
\stopluacode

\dorecurse {1000} {
  test \page
}

\startluacode
  statistics.stoptiming()
  context(statistics.elapsedtime())
\stopluacode
\stoptext
```

These numbers are already quite okay for the normal version but the speedup of the LuaJIT version is consistent with the expectations we have by now.

[4] On Windows it makes sense to use `console2` because due to some clever buffering tricks it has a much better performance than the default console.

	traditional	jit on	jit off
Windows 8	4.5	3.6	3.6
Linux 64	4.8	3.9	4.0

The fourth benchmark uses some structuring, which involved Lua tables and housekeeping, an itemize, which involves numbering and conversions, and a table mechanism that uses more Lua than TEX.

```
\setupbodyfont[dejavu] % benchmark-4.tex
\starttext \dontcomplain
\startluacode
  if jit then
    jit.on()
    jit.off()
  end
\stopluacode
\startluacode
  statistics.starttiming()
\stopluacode

\startbuffer
  \margintext{test} test test

  \startitemize[a]
    \startitem test \stopitem
    \startitem test \stopitem
    \startitem test \stopitem
    \startitem test \stopitem
  \stopitemize

  \startxtable
    \startxrow
      \startxcell test \stopxcell
      \startxcell test \stopxcell
      \startxcell test \stopxcell
    \stopxrow
    \startxrow
      \startxcell test \stopxcell
      \startxcell test \stopxcell
      \startxcell test \stopxcell
    \stopxrow
  \stopxtable
\stopbuffer

\dorecurse {25} {
  \startchapter[title=Test #1]
    \dorecurse {25} {
      \startsection[title=Test #1]
        \getbuffer
      \stopsection
    }
  \stopchapter
}
\page

\startluacode
    statistics.stoptiming()
    context(statistics.elapsedtime())
\stopluacode \stoptext
```

Here it looks like jit slows down the process, but of course we shouldn't take the last digit too seriously.

	traditional	jit on	jit off
Windows 8	20.9	16.8	16.5
Linux 64	20.4	16.0	16.1

Again, this example does a bit of logging, but not that much reading from file as buffers are kept in memory.

We should start wondering when jit does kick in. This is what the fifth benchmark does.

```
\starttext % benchmark-5.tex
\startluacode
  if jit then
    jit.on()
    jit.off()
  end

  local t = os.clock()
  local a = 0
  for i=1,10*1000*1000 do
    a = a + math.sin(i)
  end
  context(os.clock()-t)

  context.par()

  local t = os.clock()
  local sin = math.sin
  local a = 0
  for i = 1,10*1000*1000 do
    a = a + sin(i)
  end
  context(os.clock()-t)
\stopluacode
\stoptext
```

Here we see jit having an effect! First of all the LuaJIT versions are now 4 times faster. Making the `sin` a `local` function does not make much of a difference because the math functions are optimized anyway. See how we're still faster when jit is disabled:

	traditional	jit on	jit off
Windows 8	1.97 / 1.54	0.46 / 0.45	0.73 / 0.61
Linux 64	1.62 / 1.27	0.41 / 0.42	0.67 / 0.52

Unfortunately this kind of calculation (in these amounts) doesn't happen that often but maybe some users can benefit.

3 Conclusions

So, does it make sense to complicate the LuaTEX build with LuaJIT? It does when speed matters, for instance when ConTEXt is run as a service. Some 25% gain in speed means less waiting time, better use of CPU cycles, less energy consumption, etc. On the other hand, computers are still becoming faster

and compared to those speed-ups the 25% is not that much. Also, as TEX deals with files, the advance of SSD disks and larger and faster memory helps too. Faster and larger CPU caches contributes too. On the other hand, multiple cores don't help that much on a system that only runs TEX. Interesting is that multi-core architectures tend to run at slower speeds than single cores where more heat can be dissipated and in that respect servers mostly running TEX are better off with fewer cores that can run at higher frequencies. But anyhow, 25% is still better than nothing and it makes my old laptop feel faster. It prolongs the lifetime of machines!

Now, say that we cannot speed up TEX itself that much, but that there is still something to gain at the Lua end — what can we reasonably expect? First of all we need to take into account that only part of the runtime is due to Lua. Say that this is 25% for a document of average complexity.

$$\text{runtime}_{\text{tex}} + \text{runtime}_{\text{lua}} = 100$$

We can consider the time needed by TEX to be constant; so if that is 75% of the total time (say 100 seconds) to begin with, we have:

$$75 + \text{runtime}_{\text{lua}} = 100$$

It will be clear that if we bring down the runtime to 80% (80 seconds) of the original we end up with:

$$75 + \text{runtime}_{\text{lua}} = 80$$

And the 25 seconds spent in Lua went down to 5, meaning that Lua processing got 5 times faster! It is also clear that getting much more out of Lua becomes hard. Of course we can squeeze more out of it, but TEX still needs its time. It is hard to measure how much time is actually spent in Lua. We do keep track of some times but it is not that accurate. These experiments and the gain in speed indicate that we probably spend more time in Lua than we first guessed. If you look in the ConTEXt source it's not that hard to imagine that indeed we might well spend 50% or more of our time in Lua and/or in transferring control between TEX and Lua. So, in the end there still might be something to gain.

Let's take benchmark 4 as an example. At some point we measured for a regular LuaTEX 0.74 run 27.0 seconds and for a LuaJITTEX run 23.3 seconds. If we assume that the LuaJIT virtual machine is twice as fast as the normal one, some juggling with numbers makes us conclude that TEX takes some 19.6 seconds of this. An interesting border case is `\directlua`: we sometimes pass quite a lot of data and that gets tokenized first (a TEX activity) and the resulting token list is converted into a string (also a TEX activity) and then converted to bytecode (a Lua task) and when okay executed by Lua. The time involved in conversion to byte code is probably the same for stock Lua and LuaJIT.

In the LuaTEX case, 30% of the runtime for benchmark 4 is on Lua's tab, and in LuaJITTEX it's 15%. We can try to bring down the Lua part even more, but it makes more sense to gain something at the TEX end. There macro expansion can be improved (read: ConTEXt core code) but that is already rather optimized.

Just for the sake of completeness Luigi compiled a stock LuaTEX binary for 64-bit Linux with the `-o3` option (which forces more inlining of functions as well as a different switch mechanism). We did a few tests and this is the result:

	LuaTEX 0.74	`-o2`	`-o3`
`benchmark-1`		15.5	15.0
`benchmark-2`		35.8	34.0
`benchmark-3`		4.0	3.9
`benchmark-4`		16.0	15.8

This time we used `--batch` and `--silent` to eliminate terminal output. So, if you really want to squeeze out the maximum performance you need to compile with `-o3`, use LuaJITTEX (with the faster virtual machine) but disable jit (disabled by default anyway).

We have no reason to abandon stock Lua. Also, because during these experiments we were still using Lua 5.1 we started wondering what the move to 5.2 would bring. Such a move forward also means that ConTEXt MkIV will not depend on specific LuaJIT features, although it is aware of it (this is needed because we store bytecodes). But we will definitely explore the possibilities and see where we can benefit. In that respect there will be a way to enable and disable jitting. So, users have the choice to use either stock LuaTEX or the jit-aware version but we default to the regular binary.

As we use stock Lua as benchmark, we will use the `bit32` library, while LuaJIT has its own bit library. Some functions can be aliased so that is no big deal. In ConTEXt we use wrappers anyway. More problematic is that we want to move on to Lua 5.2 and not all 5.2 features are supported (yet) in LuaJIT. So, if LuaJIT is mandatory in a workflow, then users had better make sure that the Lua code is compatible. We don't expect too many problems in ConTEXt MkIV.

4 About speed

It is worth mentioning that the Lua version in LuaTEX has a patch for converting floats into strings. Instead of some `INF#` result we just return zero, simply because TEX is integer-based and intercepting

incredibly small numbers is too cumbersome. We had to apply the same patch in the jit version.

The benchmarks only indicate a trend. In a real document much more happens than in the above tests. So what are measurements worth? Say that we compile *The TEXbook*. This grandparent of all documents coded in TEX is rather plainly coded (using of course plain TEX) and compiles pretty fast. Processing does not suffer from complex expansions, there is no color, hardly any text manipulation, it's all 8 bit, the pagebuilder is straightforward as is all spacing. Although on my old machine I can get ConTEXt to run at over 200 pages per second, this quickly drops to 10% of that speed when we add some color, backgrounds, headers and footers, font switches, etc.

So, running documents like *The TEXbook* for comparing the speed of, say, pdfTEX, XETEX, LuaTEX and now LuaJITTEX makes no sense. The first one is still eight bit, the rest are Unicode. Also, *The TEXbook* uses traditional fonts with traditional features so effectively that it doesn't rely on anything that the new engines provide, not even ε-TEX extensions. On the other hand, a recent document uses advanced fonts, properties like color and/or transparencies, hyperlinks, backgrounds, complex cover pages or chapter openings, embeds graphics, etc. Such a document might not even process in pdfTEX or XETEX, and if it does, it's still comparing different technologies: eight bit input and fast fonts in pdfTEX, frozen Unicode and wide font support in XETEX, instead of additional trickery and control, written in Lua. So, when we investigate speed, we need to take into account what (font and input) technologies are used as well as what complicating layout and rendering features play a role. In practice speed only matters in an edit-view cycle and services where users wait for some result.

It's rather hard to find a recent document that can be used to compare these engines. The best we could come up with was the rendering of the user interface documentation. The last column is the time in seconds, the others are the command line invocation.

```
texexec --engine=pdftex    --global x-set-12.mkii   5.9
texexec --engine=xetex     --global x-set-12.mkii   6.2
context --engine=luatex    --global x-set-12.mkiv   6.2
context --engine=luajittex --global x-set-12.mkiv   4.6
```

Keep in mind that `texexec` is a Ruby script and uses `kpsewhich` while `context` uses Lua and its own (TDS-compatible) file manager. But still, it is interesting to see that there is not that much difference if we keep jit out of the picture. This is because in MkIV we have somewhat more clever XML processing, although earlier measurements have demonstrated that in this case not that much speedup can be assigned to that.

And so recent versions of MkIV already keep up rather well with the older eight bit world. We do way more in MkIV and the interfacing macros are nicer but potentially somewhat slower. Some mechanisms might be more efficient because of using Lua, but some actually have more overhead because we keep track of more data. Font feature processing is done in Lua, but somehow can keep up with the libraries used in XETEX, or at least is not that significant a difference, although I can think of more demanding tasks. Of course in LuaTEX we can go beyond what libraries provide.

No matter what one takes into account, performance is not that much worse in LuaTEX, and if we enable jit and so remove some of the traditional Lua virtual machine overhead, we're even better off. Of course we need to add a disclaimer here: don't force us to prove that the relative speed ratios are the same for all cases. In fact, it being so hard to measure and compare, performance can be considered to be something taken for granted as there is not that much we can do about getting nicer numbers, apart from maybe parallelizing which brings other complexities into the picture. On our servers, a few other virtual machines running TEX services kicking in at the same time, using CPU cycles, network bandwidth (as all data lives someplace else) and asking for disk access have much more impact than the 25% we gain. Of course if all processes run faster then we've gained something.

For what it's worth: processing this text takes some 2.3 seconds on my laptop for regular LuaTEX and 1.8 seconds with LuaJITTEX, including the extra overhead of restarting. As this is a rather average example it fits earlier measurements.

Processing a font manual (work in progress) takes LuaJITTEX 15 seconds for 112 pages compared to 18.4 seconds for LuaTEX. The not yet finished manual loads 20 different fonts (each with multiple instances), uses colors, has some MetaPost graphics and does some font juggling. The gain in speed sounds familiar.

5 The future

At the 2012 Lua conference Roberto Ierusalimschy mentioned that the virtual machine of LuaJIT is about twice as fast due to it being partly done in assembler while the regular machinery is written in standard C code and keeps portability in mind.

He also presented some plans for future versions of Lua. There will be some lightweight helpers for

UTF. Our experiences so far are that only a handful of functions are actually needed: byte to character conversions and vice versa, iterators for UTF characters and UTF values and maybe a simple substring function is probably enough. Currently LuaTEX has some extra string iterators and it will provide the converters as well.

There is a good chance that LPEG will become a standard library (which it already is in LuaTEX), which is also nice. It's interesting that, especially on longer sequences, LPEG can beat the string matchers and replacers, although when in a substitution no match and therefore no replacements happen, the regular gsub wins. We're talking small numbers here, in daily usage LPEG is about as efficient as you can wish. In ConTEXt we have a `lpeg.UR` and `lpeg.US` and it would be nice to have these as native UTF related methods, but I must admit that I seldom need them.

This and other extensions coming to the language also have some impact on a jit version: the current LuaJIT is already not entirely compatible with Lua 5.2 so you need to keep that into account if you want to use this version of LuaTEX. So, unless LuaJIT follows the mainstream development, as ConTEXt MkIV user you should not depend on it. But at the moment it's nice to have this choice.

The yet experimental code will end up in the main LuaTEX repository in time before the TEX Live 2013 code freeze. In order to make it easier to run both versions alongside, we have added the Lua 5.2 built-in library `bit32` to LuaJITTEX. We found out that it's too much trouble to add that library to Lua 5.1 but LuaTEX has moved on to 5.2 anyway.

6 Running

So, as we will definitely stick to stock Lua, one might wonder if it makes sense to officially support jitting in ConTEXt. First of all, LuaTEX is not influenced that much by the low level changes in the API between 5.1 and 5.2. Also LuaJIT does support the most important new 5.2 features, so at the moment we're mostly okay. We expect that eventually LuaJIT will catch up but if not, we are not in big trouble: the performance of stock Lua is quite okay and above all, it's portable![5] For the moment you can consider LuaJITTEX to be an experiment and research tool, but we will do our best to keep it production ready.

So how do we choose between the two engines? After some experimenting with alternative startup scenarios and dedicated caches, the following solution was reached:

```
context --engine=luajittex ...
```

The usual preamble line also works:

```
% engine=luajittex
```

As the main infrastructure uses the `luatex` and related binaries, this will result in a relaunch: the `context` script will be restarted using `luajittex`. This is a simple solution and the overhead is rather minimal, especially compared to the somewhat faster run. Alternatively you can copy `luajittex` over `luatex` but that is more drastic. Keep in mind that `luatex` is the benchmark for development of ConTEXt, so the jit aware version might fall behind sometimes.

Yet another approach is adapting the configuration file, or better, provide (or adapt) your own `texmfcnf.lua` in for instance `texmf-local/web2c` path:

```
return {
  type    = "configuration",
  version = "1.2.3",
  date    = "2012-12-12",
  time    = "12:12:12",
  comment = "Local overloads",
  author  = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
  content = {
    directives = {
      ["system.engine"] = "luajittex",
    },
  },
}
```

This has the same effect as always providing `--engine=luajittex` but only makes sense in well controlled situations as you might easily forget that it's the default. Of course one could have that file and just comment out the directive unless in test mode.

Because the bytecode of LuaJIT differs from the one used by Lua itself we have a dedicated format as well as dedicated bytecode compiled resources (for instance `tmb` instead of `tmc`). For most users this is not something they should bother about as it happens automatically.

Based on experiments, by default we have disabled jit so we only benefit from the faster virtual machine. Future versions of ConTEXt might provide some control over that but first we want to conduct more experiments.

7 Addendum

These developments and experiments took place in November and December 2012. At the time of this writing we also made the move to Lua 5.2 in stock

[5] Stability and portability are important properties of TEX engines, which is yet another reason for using Lua. For those doing number crunching in a document, jit can come in handy.

LuaTEX; the first version to provide this was 0.74. Here are some measurements on Taco Hoekwater's 64-bit Linux machine:

	LuaTEX 0.70	LuaTEX 0.74	
benchmark-1	23.67	19.57	faster
benchmark-2	65.41	62.88	faster
benchmark-3	4.88	4.67	faster
benchmark-4	23.09	22.71	faster
benchmark-5	2.56/2.06	2.66/2.29	slower

There is a good chance that this is due to improvements of the garbage collector, virtual machine and string handling. It also looks like memory consumption is a bit less. Some speed optimizations in reading files have been removed (at least for now) and some patches to the `format` function (in the `string` namespace) that dealt with (for TEX) unfortunate number conversions have not been ported. The code base is somewhat cleaner and we expect to be able to split up the binary in a core program plus some libraries that are loaded on demand.[6] In general, we don't expect too many issues in the transition to Lua 5.2, and ConTEXt is already adapted to support LuaTEX with 5.2 as well as LuaJITTEX with an older version.

Running the same tests on a 32-bit Windows machine gives this:

	LuaTEX 0.70	LuaTEX 0.74	
benchmark-1	26.4	25.5	faster
benchmark-2	64.2	63.6	faster
benchmark-3	7.1	6.9	faster
benchmark-4	28.3	27.0	faster
benchmark-5	1.95/1.50	1.84/1.48	faster

The gain is less impressive but the machine is rather old and we can benefit less from modern CPU properties (cache, memory bandwidth, etc.). I tend to conclude that there is no significant improvement here but it also doesn't get worse. However we need to keep in mind that file I/O is less optimal in 0.74 so this might play a role. As usual, runtime is negatively influenced by the relatively slow speed of displaying messages on the console (even when we use `console2`).

A few days before the end of 2012, Akira Kakuto compiled native Windows binaries for both engines.

This time I decided to run a comparison inside the SciTE editor, that has very fast console output.[7]

	LuaTEX 0.74 (Lua 5.2)	LuaJITTEX 0.72 (Lua 5.1)	
benchmark-1	25.4	25.4	similar
benchmark-2	54.7	36.3	faster
benchmark-3	4.3	3.6	faster
benchmark-4	20.0	16.3	faster
benchmark-5	1.93/1.48	0.74/0.61	faster

Only the MetaPost library and conversion benchmark didn't show a speedup. The regular TEX tests 1–3 gain some 15–35%. Enabling jit (off by default) slowed down processing. For the sake of completeness I also timed LuaJITTEX on the console, so here you see the improvement of both engines.

	LuaTEX 0.70	LuaTEX 0.74	LuaJITTEX 0.72
benchmark-1	26.4	25.5	25.9
benchmark-2	64.2	63.6	45.5
benchmark-3	7.1	6.9	6.0
benchmark-4	28.3	27.0	23.3
benchmark-5	1.95/1.50	1.84/1.48	0.73/0.60

In this text, the term jit has come up a lot but you might rightfully wonder if the observations here relate to jit at all. For the moment I tend to conclude that the implementation of the virtual machine and garbage collection have more impact than the actual just-in-time compilation. More exploration of jit is needed to see if we can really benefit from that. Of course the fact that we use a bit less memory is also nice. In case you wonder why I bother about speed at all: we happen to run LuaTEX mostly as a (remote) service and generating a bunch of (related) documents takes a bit of time. Bringing the waiting down from 15 to 10 seconds might not sound impressive but it makes a difference when it is someone's job to generate these sets.

In summary: just before we entered 2013, we saw two rather fundamental updates of LuaTEX show up: an improved traditional one with Lua 5.2 as well as the somewhat faster LuaJITTEX with a mixture between 5.1 and 5.2. And in 2013 we will of course try to make them both even more attractive.

◇ Hans Hagen
`http://pragma-ade.com`

[6] Of course this poses some constraints on stability as components get decoupled, but this is one of the issues that we hope to deal with properly in the library project.

[7] Most of my personal TEX runs are from within SciTE, while most runs on the servers are in batch mode, so normally the overhead of the console is acceptable or even neglectable.