Zpravodaj Československého sdružení uživatelů TeXu

Hans Hagen
ConTeXt Performance

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 28 (2018), No. 1-4, 59–78

Persistent URL: http://dml.cz/dmlcz/150107

ConTEXt Performance

Hans Hagen

The processing speed of a TEX engine is affected by a number of factors, such as the format, macros, scripting, fonts, microtypographic extensions, SyncTEX, and command-line redirection. The article discusses the individual factors from the perspective of a ConTEXt user. The article also measures the overhead of ConTEXt MkII and MkIV, the impact of command-line redirection on the speed of ConTEXt MkIV, the impact of fonts on the speed of typesetting with ConTEXt MkIV, and the speed of typesetting with ConTEXt MkII and MkIV.

Keywords: Lua, LuaTEX, LuajitTEX, ConTEXt MkII, ConTEXt MkIV

Introduction

This article is about performance. Although it concerns LuaTEX this text is only meant for ConTEXt users. This is not because they ever complain about performance, on the contrary, I never received a complain from them. No, it's because it gives them some ammunition against the occasionally occurring nagging about the speed of LuaTEX (somewhere on the web or at some meetings). My experience is that, in most such cases, those complaining have no clue what they're talking about, so effectively we could just ignore them, but let's, for the sake of our users, waste some words on the issue.

What performance

So, what exactly does performance refer to? If you use ConTEXt, there are probably only two things that matter:

- How long does one run take?
- How many runs do I need?

Processing speed is reported at the end of a run in terms of seconds spent on the run, but also in pages per second. The runtime is made up of three components:

- start-up time,
- processing pages, and
- finishing the document.

The startup time is rather constant. Let's take my 2013 Dell Precision with i7-3840QM as reference. A simple

```
\starttext
\stoptext
```

doi: 10.5300/2018-1-4/59

document reports 0.4 seconds but, as we wrap the run in an `mtxrun` management run, we have an additional 0.3 overhead (auxiliary file handling, PDF viewer management, etc). This includes loading the Latin Modern font. With LuajitTeX, these times are below 0.3 and 0.2 seconds. It might look like a lot of overhead, but it feels snappy in an edit-preview runs. One can try this:

```
\stoptext
```

which bring down the time to about 0.2 seconds for both engines, but it doesn't do anything useful in practice.

Finishing a document is not that demanding, because most gets flushed as we go. The more (large) fonts we use, the longer it takes to finish a document, but, on the average, that time is not worth noticing. The main runtime contribution comes from processing the pages.

Okay, this is not always true. For instance, if we process a 400 page book from 2500 small XML files with multiple graphics per page, there is a little overhead in loading the files and in constructing the XML tree as well as in inserting the graphics, but in such cases one expects a few seconds longer runtime. The METAFUN manual has some 450 pages with over 2500 runtime-generated METAPOST graphics. It has color, uses quite some fonts, has lots of font switches (verbatim, too), but, still, one run takes only 18 seconds in stock LuaTeX and less than 15 seconds with LuajitTeX. Keep these numbers in mind if a non-ConTeXt user barks against the performance tree that his few-page mediocre document takes 10 seconds to compile: the content, styling, quality of macros and whatever one can come up with all play a role. Personally, I find any rate between 10 and 30 pages per second acceptable, and, if I get the lower rate, then I normally know pretty well that the job is demanding in all kind of aspects.

Over time, the ConTeXt–LuaTeX combination, in spite of the fact that more functionality has been added, has not become slower. In fact, some subsystems have been sped up. For instance, font handling is very sensitive to adding functionality. However, each version so far performed a bit better. Whenever some neat new trickery was added, at the same time improvements were made thanks to more insight in the matter. In practice, we're not talking of changes in speed by large factors but more by small percentages. I'm pretty sure that most ConTeXt users never noticed. Recently, a 15–30% speed up (in font handling) was realized (for more complex fonts), but only when you use such complex fonts and pages full of text will you see a positive impact on the whole run.

There is one important factor I didn't mention yet: the efficiency of the console. You can best check that by making a format (`context --make en`). When that is done by piping the messages to a file, it takes 3.2 seconds on my laptop and about the same when done from the editor (SCITE), maybe because the LuaTeX run and the log pane run on a different thread. When I use the standard console, it takes 3.8 seconds in Windows 10 Creative update (in older versions it took 4.3 seconds and slightly less when using a console wrapper). The

powershell takes 3.2 seconds, which is the same as piping to a file. Interesting is that in Bash on Windows, it takes 2.8 seconds and 2.6 seconds when piped to a file. Normal runs are somewhat slower, but it looks like the 64 bit Linux binary is somewhat faster than the 64 bit mingw version.[1] Anyway, it demonstrates that when someone yells a number, you need to ask what the conditions were.

At a ConTeXt meeting, there has been a presentation about possible speedups of a run by using, for instance, a separate syntax checker to prevent a useless run. However, the use case concerned a document that took a minute on the machine used, while the same document took a few seconds on mine. At the same meeting, we also did a comparison of speed for a LaTeX run using PdfTeX and the same document migrated to ConTeXt MkIV using LuaTeX (Harald Königs xml torture and compatibility test). Contrary to what one might expect, the ConTeXt run was significantly faster; the resulting document was a few gigabytes in size.

Bottlenecks

I will discuss a few potential bottlenecks next. A complex integrated system like ConTeXt has lots of components and some can be quite demanding. However, when something is not used, it has no (or hardly any) impact on performance. Even when we spend a lot of time in Lua, that is not the reason for a slowdown. Sometimes using Lua results in a speedup, sometimes it doesn't matter. Complex mechanisms like natural tables, for instance, will not suddenly become less complex. So, let's focus on the "aspects" that come up in those complaints: fonts and Lua. Because I only use ConTeXt and occasionally test with the plain TeX version that we provide, I will not explore the potential impact of using truckloads of packages, styles, and such, which I'm sure of plays a role, but one neglected in my discussion.

Fonts

According to the principles of LuaTeX, we process (OpenType) fonts using Lua. That way, we have complete control over any aspect of font handling, and can, as expected in TeX systems, provide users with what they need, now and in the future. In fact, if we didn't have that freedom in ConTeXt, I would probably have already quit using TeX a decade ago and found myself some other (programming) niche.

[1] Long ago, we found that LuaTeX is very sensitive to for instance the cpu cache, so maybe there are some differences due to optimization flags and/or the fact that bash runs in one thread, and all file io takes place in the main Windows instance. Who knows.

After a font has been loaded, part of the data gets passed to the TEX engine, so that it can do its work. For instance, in order to be able to typeset a paragraph, TEX needs to know the dimensions of glyphs. Once a font has been loaded (that is, the binary blob) it's fetched from a cache the next time. Initial loading (and preparation) takes some time, depending on the complexity and the size of the font. Loading from cache is close to instantaneous. After loading, the dimensions are passed to TEX, but all data remains accessible for any desired usage. The OpenType feature processor, for instance, uses that data, and ConTEXt, for sure, needs that data (quickly accessible) for different purposes, too.

When a font is used in a so-called base mode, we let TEX do the ligaturing and kerning. This is possible with simple fonts and features. If you have a critical workflow, you might enable base mode, which can be done per font instance. Processing in node mode takes some time, but how much depends on the font and script. Normally, there is no difference between ConTEXt and generic usage. In ConTEXt, we also have dynamic features, and the impact on performance depends on usage. In addition to base and node, we also have plug mode, but that is only used for testing and therefore not advertised.

Every `\hbox` and every paragraph goes through the font handler. Because we support mixed modes, some analysis takes place, and because we do more in ConTEXt, the generic analyzer is more lightweight, which again can mean that a generic run is not slower than a similar ConTEXt one.

Interesting is that added functionality for variable and/or color fonts had no impact on performance. Runtime-added user features can have some impact, but, when defined well, it can be neglected. I bet that when you add additional node list handling yourself, its impact on performance will be larger. But, in the end, what counts is that the job gets done and the more you demand the higher the price you pay.

Lua

The second possible bottleneck when using LuaTEX can be in using Lua code. However, using that is laughable as an argument for slow runs. For instance, ConTEXt MkIV can easily spend half its time in Lua, and that is not making it any slower than MkII using PdfTEX doing equally complex things. For instance, the embedded metapost library makes MkIV way faster than MkII, and the built-in xml processing capabilities in MkIV can easily beat MkII xml handling, apart from the fact that it can do more, like filtering by path and expression. In fact, files that take, say, half a minute in MkIV, could as well have taken 15 minutes or more in MkII (and imagine multiple runs then).

So, for ConTEXt, using Lua to achieve its objectives is mandatory. The combination of TEX, metapost and Lua is pretty powerful! Each of these

components is really fast. If TEX is your bottleneck, review your macros! When LUA seems to be the bad, go over your code and make it better. Much of the LUA code I see flying around doesn't look that efficient, which is okay, because the interpreter is really fast, but don't blame LUA beforehand, blame your coding (style) first. When METAPOST is the bottleneck, well, sometimes not much can be done about it, but when you know that language well enough, you can often make it perform better.

For the record: every additional mechanism that kicks in, like character spacing (the ugly one), case treatments, special word and line trickery, marginal stuff, graphics, line numbering, underlining, referencing, and a few dozen more will add a bit to the processing time. In that case, in CONTEXT, the font related runtime gets pretty well obscured by other things happening, just that you know.

Some timing

Next, I will show some timings related to fonts. For this, I use stock LUATEX (second column) as well as LUAJITTEX (last column), which, of course, performs much better. The timings are rounded to three decimal places, but, as the system load is usually only consistent in a set of test runs, the last two decimals only matter in relative comparison. So, for comparing runs over time, round to the first decimal. Let's start with loading a bodyfont. This happens once per document, and one usually only has one bodyfont active. Loading involves definitions as well as setting up math, so a couple of fonts are actually loaded even if they're not used later on. A setup normally involves a serif, sans, mono and math setup (in CONTEXT).[2]

bodyfont		
modern	0.023 s	0.019 s
pagella	0.127 s	0.079 s
termes	0.128 s	0.087 s
cambria	0.180 s	0.123 s
dejavu	0.140 s	0.092 s
ebgaramond	0.142 s	0.093 s
lucidaot	0.146 s	0.120 s

There is a bit of a difference between the font sets, but a safe average is 150 milliseconds, and this is rather constant over runs.

[2]The timing for Latin Modern is so low, because that font is loaded already.

An actual font switch can result in loading a font, but this is a one-time overhead. Loading four variants (regular, bold, italic and bold italic) roughly takes the following time:

bodyfont switch and 4 style changes (first time)		
modern	0.028 s	0.028 s
pagella	0.035 s	0.031 s
termes	0.036 s	0.069 s
cambria	0.052 s	0.047 s
dejavu	0.091 s	0.069 s
ebgaramond	0.022 s	0.016 s
lucidaot	0.017 s	0.031 s

Using them again later on takes no time:

bodyfont switch and 4 style changes (follow-up)		
modern	0.000 s	0.000 s
pagella	0.001 s	0.000 s
termes	0.000 s	0.001 s
cambria	0.000 s	0.000 s
dejavu	0.001 s	0.000 s
ebgaramond	0.000 s	0.000 s
lucidaot	0.000 s	0.000 s

Before we start timing the font handler, a few baseline benchmarks are shown. When no font is applied and nothing else is done with the node list, we get:

100 hboxes with 4 texts and no font handling		
baseline	0.142 s	2.343 s

A simple monospaced no-features-applied run takes a bit more:

100 hboxes with 4 texts and no features		
baseline	0.275 s	0.220 s

Now, we show a one-font typesetting run. As with the two benchmarks before, we just typeset a text in a `\hbox`, so no par builder interference happens. We use the `sapolsky` sample text and typeset it 100 times 4, first without font switches.

100 hboxes with 4 texts using one font		
modern	0.933 s	0.591 s
pagella	1.027 s	0.660 s
termes	1.032 s	0.604 s
cambria	1.483 s	0.862 s
dejavu	1.009 s	0.581 s
ebgaramond	3.240 s	1.774 s
lucidaot	0.699 s	0.444 s

Much more runtime is needed when we typeset with four font switches. Ebgaramond is the most demanding. Actually, we're not doing 4 fonts there because ebgaramond has no bold, so the numbers are a bit lower than expected for this example. One reason for it being demanding is that it has lots of (contextual) lookups. Combining lookups saves space and time, so complexity of a font is not always a good predictor for performance hits.

100 hbox with 4 texts using 4 font switches		
modern	1.611 s	0.946 s
pagella	1.697 s	0.975 s
termes	1.727 s	1.038 s
cambria	2.815 s	1.626 s
dejavu	1.946 s	1.087 s
ebgaramond	5.445 s	2.899 s
lucidaot	1.288 s	0.746 s

If we typeset paragraphs, we get the following:

100 times 4 texts on pages (Figure 1)		
modern	1.377 s	0.904 s
pagella	1.523 s	0.961 s
termes	1.453 s	0.898 s
cambria	1.901 s	1.138 s
dejavu	1.437 s	0.917 s
ebgaramond	3.714 s	2.133 s
lucidaot	1.117 s	0.767 s

We're talking of some 275 pages here.

100 times 4 texts on pages using 4 styles (Figure 2)		
modern	2.074 s	1.307 s
pagella	2.155 s	1.338 s
termes	2.153 s	1.373 s
cambria	3.349 s	2.012 s
dejavu	2.408 s	1.453 s
ebgaramond	4.368 s	2.512 s
lucidaot	1.682 s	1.056 s

There is, of course, overhead in handling paragraphs and pages:

100 times 4 texts on pages with no features (Figure 3)		
baseline	0.825 s	0.559 s

Before I discuss these numbers in more detail, two more benchmarks are shown. The next table concerns a paragraph with only a few (bold) words.

100 times 1 text on pages with bold font switches (Figure 4)		
modern	0.409 s	0.263 s
pagella	0.445 s	0.281 s
termes	0.432 s	0.300 s
cambria	0.606 s	0.368 s
dejavu	0.465 s	0.295 s
ebgaramond	0.922 s	0.530 s
lucidaot	0.345 s	0.220 s

The next table concerns a paragraph with a few monospaced words using `\type`.

100 times 1 text on pages with word verbatim switches (Figure 5)		
modern	0.380 s	0.255 s
pagella	0.396 s	0.266 s
termes	0.384 s	0.278 s
cambria	0.535 s	0.355 s
dejavu	0.366 s	0.247 s
ebgaramond	0.939 s	0.533 s
lucidaot	0.322 s	0.216 s

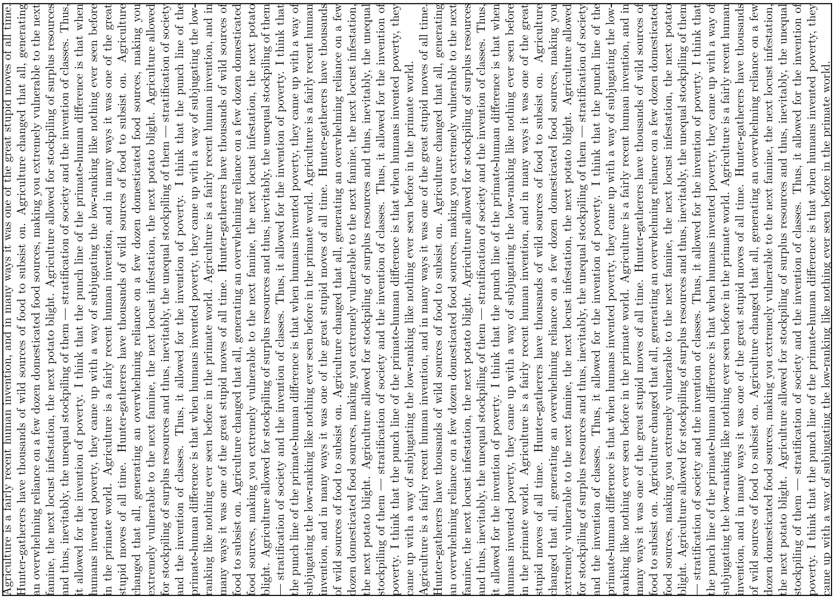


Figure 1 100 times 4 texts on pages

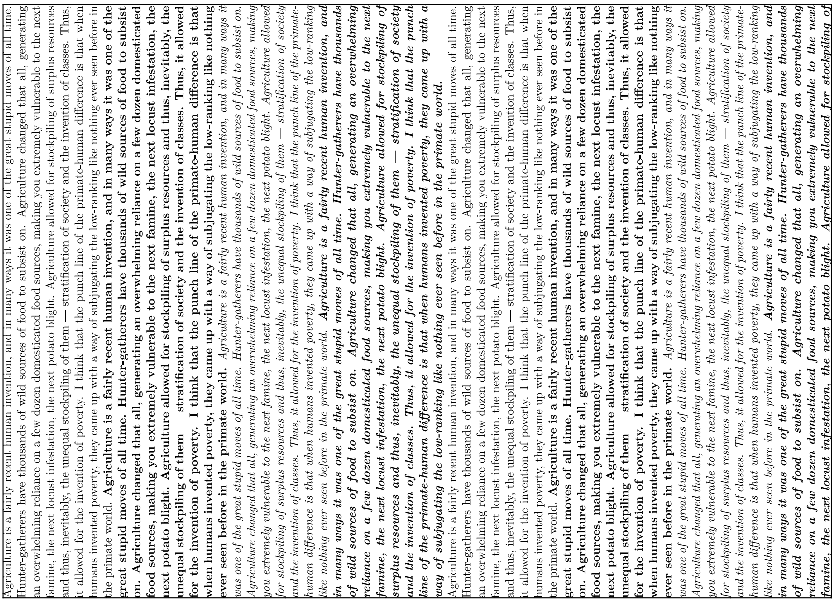


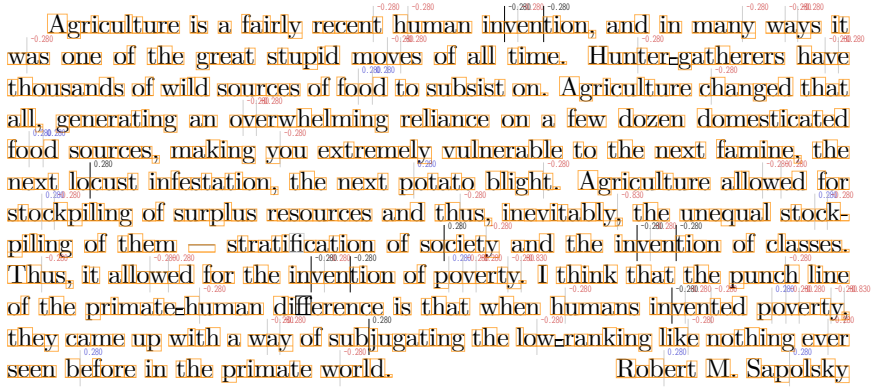
Figure 2 100 times 4 texts on pages using 4 styles

Figure 3 100 times 4 texts on pages with no features

Figure 4 100 times 1 text on pages with bold switches

Figure 5 100 times 1 text on pages with word verbatim switches

When a node list (hbox or paragraph) is processed, each glyph is looked at. One important property of LuaTeX (compared to PdfTeX) is that it hyphenates the whole text, not only the most feasible spots. For the `sapolsky` snippet, this results in 200 potential breakpoints registered in an equal number of discretionary nodes. The snippet has 688 characters grouped into 125 words and, because it's an English quote, we're not hampered with composed characters or complex script handling. And, when we mention 100 runs, then we actually mean 400 ones when font switching and bodyfonts are compared.



In order to get substitutions and positioning right, we need not only to consult streams of glyphs but also combinations with preceding pre or replace, or trailing post and replace texts. When a font has a bit more complex substitutions, as ebgaramond has, multiple (sometimes hundreds of) passes over the list are made. This is why the more complex a font is, the more runtime is involved.

Another factor, one you could easily deduce from the benchmarks, is intermediate font switches. Even a few such switches (in the last benchmarks) already result in a runtime penalty. The four switch benchmarks show an impressive increase of runtime, but it's good to know that such a situation seldom happens. It's also important not to confuse, for instance, a verbatim snippet with a bold one. The bold one is indeed leading to a pass over the list, but verbatim is normally skipped, because it uses a font that needs no processing. That verbatim or bold have the same penalty is mainly due to the fact that verbatim itself is costly: the text is picked up using a different catcode regime and travels through TEX and LUA before it finally gets typeset. This relates to special treatments of spacing, syntax highlighting, and such.

Also, keep in mind that the page examples are quite unreal. We use a layout with no margins, just text from edge to edge.

So, what is a realistic example? That is hard to say. Unfortunately, no one has ever asked us to typeset novels. They are rather brain dead-products for a machinery, so they process fast. On the mentioned laptop, 350 word pages in Dejavu fonts can be processed at a rate of 75 pages per second with LUATEX and over 100 pages per second with LUAJITTEX. On a more modern laptop or a professional server, the performance is of course better. And, for automated flows, batch mode is your friend. The rate is not much worse for a document in a language with a bit more complex character handling, take accents or ligatures. Of course, PDFTEX is faster on such a dumb document, but kick in some more functionality, and the advantage quickly disappears. So, if someone complains that LUATEX needs 10 or more seconds for a simple few page document ... you can bet that when the fonts are seen as reason, then the setup is pretty bad. Personally I would not waste time on such a complaint.

Valid questions

Here are some reasonable questions that you can ask when someone complains to you about the slowness of LUATEX:

What engines do you compare?
If you come from PDFTEX, you come from an 8-bit world: input and font handling are based on bytes, and hyphenation is integrated into the par builder.

If you use UTF-8 in PdfTeX, the input is decoded by TeX macros, which carries a speed penalty. Because in the wide engines macro names can also be UTF sequences, construction of macro names is less efficient too.

When you try to use wide fonts, there is, again, a penalty. Now, if you use XeTeX or LuaTeX, your input is UTF-8, which becomes something 32-bit internally. Fonts are wide, so more resources are needed, apart from these fonts being larger and in need of more processing due to feature handling. Where XeTeX uses a library, LuaTeX uses its own handler. Does that have a consequence for performance? Yes and no. First of all, it depends on how much time is spent on fonts at all, but even then, the difference is not that large. Sometimes XeTeX wins, sometimes it's LuaTeX. One thing is clear: LuaTeX is more flexible as we can roll out our own solutions and therefore do more advanced font magic. For ConTeXt, it doesn't matter as we use LuaTeX exclusively, and we rely on the flexible font handler, also for future extensions. If really needed, you can kick in a library-based handler but it's (currently) not distributed as we lose other functionality, which would, in turn, result in complaints about that fact (apart from conflicting with the strive for independence).

There is no doubt that PdfTeX is faster, but, for ConTeXt, it's an obsolete engine. The hard-coded-solutions engine XeTeX is not feasible for ConTeXt either. So, in practice, ConTeXt users have no choice: LuaTeX is used, but users of other macro packages can use the alternatives if they are not satisfied with performance. The fact that ConTeXt users don't complain about speed is a clear signal that this is a no-issue. And, if you want more speed, you can always use LuajitTeX.[3] In the last section, the different engines will be compared in more detail.

Just that you know, when we do the four-switches example in plain TeX on my laptop, I get a rate of 40 pages per second, and, for one font, 180 pages per second. There is, of course, a bit more going on in ConTeXt in page building and so, but the difference between plain and ConTeXt is not that large.

What macro package is used?

When plain TeX is used, a follow up question is: what variant? The ConTeXt distribution ships with `luatex-plain`, and that is our benchmark. If there really is a bottleneck, it is worth exploring, but keep in mind that, in order to be plain, not that much can be done. The LuaTeX part is just an example of an implementation. We already discussed ConTeXt, and for LaTeX, I don't want to

[3] In plug mode, we can actually test a library and experiments have shown that performance on the average is much worse, but it can be a bit better for complex scripts, although a gain gets unnoticed in normal documents. So, one can decide to use a library but at the cost of much other functionality that ConTeXt offers, so we don't support it.

speculate where performance hits might come from. When we're talking fonts, ConTeXt can actually be a bit slower than the generic (or LaTeX) variant, because we can kick in more functionality. Also, when you compare macro packages, keep in mind that, when node list processing code is added in that package, the impact depends on interaction with other functionality and depends on the efficiency of the code. You can't compare mechanisms or draw general conclusions when you don't know what else is done!

What do you load?
Most ConTeXt modules are small and load fast. Of course, there can be exceptions when we rely on third party code; for instance, loading tikz takes a bit of time. It makes no sense to look for ways to speed that system up, because it is maintained elsewhere. There can probably be gained a bit, but, again, no user has complained so far.

If ConTeXt is not used, one probably also uses a large TeX installation. File lookup in ConTeXt is done differently, and can be faster. Even loading can be more efficient in ConTeXt, but it's hard to generalize that conclusion. If one complains about loading fonts being an issue, just try to measure how much time is spent on loading other code.

Did you patch macros?
Not everyone is a TeXpert. So, coming up with macros that are expanded many times and/or have inefficient user interfacing, can have some impact. If someone complains about one subsystem being slow, then honesty demands to complain about other subsystems as well. You get what you ask for.

How efficient is the code that you use?
Writing super-efficient code only makes sense when it's used frequently. In ConTeXt, most code is reasonably efficient. It can be that in one document, fonts are responsible for most runtime, but in another document, table construction can be more demanding while yet another document puts some stress on interactive features. When hz or protrusion is enabled, then you run substantially slower anyway, so when you are willing to sacrifice 10% or more of runtime, don't complain about other components. The same is true for enabling SyncTeX: if you are willing to add more than 10% of runtime for that, don't wither about the same amount for font handling.[4]

[4] In ConTeXt, we use a SyncTeX alternative that is somewhat faster, but it remains a fact that enabling more and more functionality will make the penalty of, for instance, font processing relatively small.

How efficient is the styling that you use?
Probably the most easily overlooked optimization is in switching fonts and colors. Although in ConTeXt, font switching is fast, I have no clue about it in other macro packages. But in a style, you can decide to use inefficient (massive) font switches. The effects can easily be tested by commenting out bits and pieces. For instance, sometimes you need to do a full bodyfont switch when changing a style, like assigning `\small\bf` to the `style` key in `\setuphead`, but often using e.g. `\tfd` is much more efficient and works quite as well. Just try it.

Are fonts really the bottleneck?
We already mentioned that one can look in the wrong direction. Maybe, once someone is convinced that fonts are the culprit, it gets hard to look at the real issue. If a similar job in different macro packages has a significantly different runtime, one can wonder what happens indeed.

It is good to keep in mind that the amount of text is often not as large as you think. It's easy to do a test with hundreds of paragraphs of text, but, in practice, we have whitespace, section titles, half empty pages, floats, itemize and similar constructs, etc. Often, we don't mix many fonts in the running text either. So, in the end, a real document is your best test.

If you use Lua, is that code any good?
You can gain from the faster virtual machine of LuajitTeX. Don't expect wonders from the jitting as that only pays off in long runs with the same code used over and over again. If the gain is high, you can even wonder how well-written your Lua code is anyway.

What if they don't believe you?
So, say that someone finds LuaTeX slow, what can be done about it? Just advice them to stick to their previously-used tool. Then, if arguments come that one also wants to use utf-8, OpenType fonts, a bit of metapost, and is looking forward to using Lua runtime, the only answer is: take it or leave it. You pay a price for progress, but, if you do your job well, the price is not that high. Tell them to spend time on learning and maybe adapting and to bark against their own tree before barking against those who took that step a decade ago. Most ConTeXt users took that step and someone still using LuaTeX after a decade can't be that stupid. It's always best to first wonder what one actually asks from LuaTeX, and if the benefit of having Lua on board has an advantage. If not, one can just use another engine.

Also think of this: when a job is slow, for me it's no problem to identify where the problem is. The question then is: can something be done about it?

Well, I happily keep the answer for myself. After all, some people always need room to complain, if only to hide their ignorance or incompetence. Who knows.

Comparing engines

The next comparison is to be taken with a grain of salt and concerns the state of affairs mid-2017. First of all, you cannot really compare MkII with MkIV: the latter has more functionality (or a more advanced implementation of functionality). And, as mentioned, you can also not really compare PdfTeX and the wide engines. Anyway, here are some (useless) tests. First, a bunch of loads. Keep in mind that different engines also deal differently with reading files. For instance, MkIV uses LuaTeX callbacks to normalize the input and has its own readers. There is a bit more overhead in starting up a LuaTeX run, and some functionality is enabled that is not present in MkII. The format is also larger, if only because we preload a lot of useful font, character and script related data.

```
\starttext
    \dorecurse {#1} {
        \input knuth
        \par
    }
\stoptext
```

When looking at the numbers, one should realize that the times include startup and job management by the runner scripts. We also run in batchmode to avoid logging to influence runtime. The average is calculated from 5 runs.

engine	**#1 = 50**	**#1 = 500**	**#1 = 2500**
pdftex	0.43 s	0.77 s	2.33 s
xetex	0.85 s	2.66 s	10.79 s
luatex	0.94 s	2.50 s	9.44 s
luajittex	0.68 s	1.69 s	6.34 s

The second example does a few switches in a paragraph:

```
\starttext
    \dorecurse {#1} {
        \tf \input knuth
        \bf \input knuth
        \it \input knuth
        \bs \input knuth
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.58 s	2.10 s	8.97 s
xetex	1.47 s	8.66 s	42.50 s
luatex	1.59 s	8.26 s	38.11 s
luajittex	1.12 s	5.57 s	25.48 s

The third example does more, resulting in multiple subranges per style:

```
\starttext
    \dorecurse {#1} {
        \tf \input knuth \it knuth
        \bf \input knuth \bs knuth
        \it \input knuth \tf knuth
        \bs \input knuth \bf knuth
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.59 s	2.20 s	9.52 s
xetex	1.49 s	8.88 s	43.85 s
luatex	1.64 s	8.91 s	41.26 s
luajittex	1.15 s	5.91 s	27.15 s

The last example adds some color. Enabling more functionality can have an impact on performance. In fact, as MkIV uses a lot of Lua and is also more advanced that MkII, one can expect a performance hit, but, in practice, the opposite happens, which can also be due to some fundamental differences deep down at the macro level.

```
\setupcolors[state=start] % default in MkIV
\starttext
    \dorecurse {#1} {
        {\red \tf \input knuth \green \it knuth}
        {\red \bf \input knuth \green \bs knuth}
        {\red \it \input knuth \green \tf knuth}
        {\red \bs \input knuth \green \bf knuth}
        \par
    }
\stoptext
```

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.61 s	2.36 s	10.33 s
xetex	1.53 s	9.25 s	45.59 s
luatex	1.65 s	8.91 s	41.32 s
luajittex	1.15 s	5.93 s	27.34 s

In these measurements, the accuracy is a few decimals, but a pattern is visible. As expected, PdfTeX wins on simple documents but starts losing when things get more complex. For these tests, I used 64-bit binaries. A 32-bit XeTeX with MkII performs the same as LuajitTeX with MkIV, but a 64-bit XeTeX is actually quite a bit slower. In that case, the mingw cross-compiled LuaTeX version does pretty well. A 64-bit PdfTeX is also slower (it looks) than a 32-bit version. So, in the end, there are more factors that play a role. Choosing between LuaTeX and LuajitTeX depends on how well the memory-limited LuajitTeX variant can handle your documents and fonts.

Because in most of our recent styles we use OpenType fonts and (structural) features as well as recent metafun extensions only present in MkIV, we cannot compare engines using such documents. The mentioned performance of LuaTeX (or LuajitTeX) and MkIV on the metafun manual illustrate that, in most cases, this combination is a clear winner.

```
\starttext
    \dorecurse {#1} {
        \null \page
    }
\stoptext
```

This gives:

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.46 s	1.05 s	3.72 s
xetex	0.73 s	1.80 s	6.56 s
luatex	0.84 s	1.44 s	4.07 s
luajittex	0.61 s	1.10 s	3.33 s

That leaves the zero run:

```
\starttext
    \dorecurse {#1} {
        % nothing
    }
\stoptext
```

This gives the following numbers. In longer runs, the difference in overhead is negligible.

engine	#1 = 50	#1 = 500	#1 = 2500
pdftex	0.36 s	0.36 s	0.36 s
xetex	0.57 s	0.57 s	0.59 s
luatex	0.74 s	0.74 s	0.74 s
luajittex	0.53 s	0.53 s	0.54 s

It will be clear that when we use different fonts, the numbers will also be different. And, if you use a lot of runtime METAPOST graphics (for instance for backgrounds), the MkIV runs end up at the top. And, when we process XML, it will be clear that going back to MkII is no longer a realistic option. It must be noted that I occasionally manage to improve performance, but we've now reached a state where there is not that much to gain. Some functionality is hard to compare. For instance, in ConTeXt, we don't use much of the PDF backend features because we implement them all in Lua. In fact, even in MkII (already done in TeX), so in the end, the speed difference there is not large and often in favour of MkIV.

For the record, I mention that shipping out the about 1250 pages has some overhead too: about 2 seconds. Here, LuajitTeX is 20% more efficient, which is an indication of quite some Lua involvement. Loading the input files has an overhead of about half a second. Starting up LuaTeX takes more time than PdfTeX and XeTeX, but that disadvantage disappears with more pages. So, in the end, there are quite some factors that blur the measurements. In practice, what matters is convenience: does the runtime feel reasonable and, in most cases, it does.

If I would replace my laptop with a reasonable comparable alternative, then that one would be some 35% faster (single threads on processors don't gain much per year). I guess that this is about the same increase in performance that ConTeXt MkIV got in that period. I don't expect such a gain in the upcoming years, so, at some point, we're stuck with what we have.

Summary

So, how "slow" is LuaTeX really compared to the other engines? If we go back in time to when the first wide engines showed up, Omega was considered to be slow, although I never tested that myself. Then, when XeTeX showed up, there was not much talk about speed, just about the fact that we could use OpenType fonts and native UTF input. If you look at the numbers, for sure you can say that it was much slower than PdfTeX. So, how come that some people complain about LuaTeX being so slow, especially when we take into account that it's not that much slower than XeTeX, and that LuajitTeX is often faster than XeTeX? Also, computers have become faster. With the wide

engines, you get more functionality and that comes at a price. This was accepted for XeTeX and is also acceptable for LuaTeX. But the price is not that high if you take into account that hardware performs better: you just need to compare LuaTeX (and XeTeX) runtime with PdfTeX runtime 15 years ago.

As a comparison, look at games and video. Resolution became much higher as did color depth. Higher frame rates were in demand. Therefore, the hardware had to become faster, and it did, and, as a result, the user experience kept up. No user will say that a modern game is slower than an old one, because the old one does 500 frames per second compared to some 50 for the new game on the modern hardware. In a similar fashion, the demands for typesetting became higher: Unicode, OpenType, graphics, xml, advanced pdf, more complex (niche) typesetting, etc. This happened more or less in parallel with computers becoming more powerful. So, as with games, the user experience didn't degrade with demands. Comparing LuaTeX with PdfTeX is like comparing a low-res, low-framerate, low-color game with a modern one. You need to have up-to-date hardware and even then, the writer of such programs needs to make sure that they run efficiently, simply because hardware no longer scales like it did decades ago. You need to look at the bigger picture.

Rychlost ConTeXtu

Rychlost TeXového stroje je ovlivněna množstvím faktorů, jako je formát, makra, skripty, písma, mikrotypografická rozšíření, SyncTeX a přesměrování standardního chybového výstupu. Článek diskutuje jednotlivé faktory z pohledu uživatele ConTeXtu. Článek dále měří režii formátů ConTeXt MkII a MkIV, dopad přesměrování výstupu na rychlost ConTeXtu MkIV, dopad písem na rychlost sazby v ConTeXtu MkIV a rychlost sazby v ConTeXtu MkII a MkIV.

Klíčová slova: Lua, LuaTeX, LuajitTeX, ConTeXt MkII, ConTeXt MkIV

Hans Hagen, pragma@wxs.nl