Performance

Hans Hagen

1. 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 complaint 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 meeting). 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.

2. 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 out of three components:

- start-up time
- processing pages
- 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
```

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 much overhead but in an edit-preview runs it feels snappy. One can try this:

```
\stoptext
```

which bring down the time to about 0.2 seconds for both engines but as it doesn't do anything useful that is no 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 constructing the XML tree as well as in inserting the graphics but in such cases one expects a few seconds more 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 users 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 plays 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 for 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 you will 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 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 where.
At a ConTEXt meeting there has been a presentation about possible speed-up of a run for instance by using 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

[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 in the main windows instance. Who knows.

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.

3. 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 slow-down. 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 the 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 to be expected in TEX systems, provide users what they need, now and in the future. In fact, if we didn't had that freedom in ConTEXt I'd probably already quit using TEX a decade ago and found myself some other (programming) niche.

After a font is 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) the next time it's fetched from a cache. Initial loading (and preparation) takes some time, depending on the complexity or 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 (fast accessible) for different purposes too.

When a font is used in 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 light weight, 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 is 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

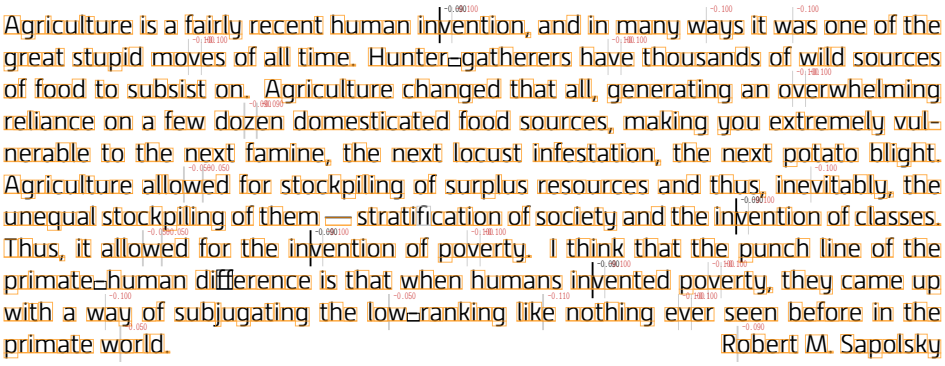
The second possible bottleneck when using LuaTEX can be in using Lua code. However, using that as argument for slow runs is laughable. For instance ConTEXt MkIV can easily spend half its time in Lua and that is not making it any slower than MKII using pdfTEX doing equally complex things. For instance 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 mandate. 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.

4. 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 given in 3 decimals but often (within a set of runs) and as the system load is normally 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 normally one has only 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]
There is a bit difference between the font sets but a safe average is 150 milli seconds and this is rather constant over runs.
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:
Using them again later on takes no time:
Before we start timing the font handler, first a few baseline benchmarks are shown. When no font is applied and nothing else is done with the node list we get:
A simple monospaced, no features applied, run takes a bit more:
Now we show a one font typesetting run. As the two benchmarks before, we just typeset a text in a `\hbox`, so no par builder interference happens. We use the `sapolsky`

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

sample text and typeset it 100 times 4 (either of not with font switches).
Much more runtime is needed when we typeset with four font switches. The garamond is most demanding. Actually we're not doing 4 fonts there because it 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. The only comment I can make about that is that it also depends on the strategies of the font designer. Combining lookups saves space and time so complexity of a font is not always a good predictor for performance hits.
If we typeset paragraphs we get this:
We're talking of some 275 pages here.
There is of course overhead in handling paragraphs and pages:
Before I discuss these numbers in more details two more benchmarks are shown. The next table concerns a paragraph with only a few (bold) words.
The following table has paragraphs with a few mono spaced words typeset using `\type`. 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 and syntax highlighting and such.

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

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them — stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

Figure 1:

So what is a realistic example? That is hard to say. Unfortunately no one 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 professional server 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, that the setup is pretty bad. Personally I'd not waste time on such a complaint.

Figure 2:

Figure 3:

Figure 4:

Figure 5:

5. 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, again there is a penalty. Now, if you use XƎTEX 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 XƎTEX 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 XƎTEX wins, sometimes 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 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 loose other functionality which in turn would 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 XƎTEX is also 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 no issue. And, if you want more speed you can 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?

If the answer is that 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 speculate where performance hits might come from. When we're talking fonts, ConTEXt can actually a bit slower than the generic (or LATEX) variant because we can kick in more functionality. Also, when

[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.

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 complained so far.
If ConTEXt is not used, one probably also uses a large TEX installations. 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 honestly 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 is 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 runtime don't complain about other components. The same is true for enabling SYNCTEX: if you are willing to add more than 10% runtime for that, don't wither about the same amount for font handling.[4]

How efficient is the styling that you use?

Probably the most easily overseen optimization is in switching fonts and color. 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 bit 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 significant different runtime one can wonder what happens indeed.

[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.

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 the 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 of for 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 him or her to stick to tool used previously. 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 large. Tell them to spend time on learning and maybe adapting and 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, maybe if only to hide their ignorance or incompetence. Who knows.

6. 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 later 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	50	500	2500
pdftex	0.43	0.77	2.33
xetex	0.85	2.66	10.79
luatex	0.94	2.50	9.44
luajittex	0.68	1.69	6.34

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	50	500	2500
pdftex	0.58	2.10	8.97
xetex	1.47	8.66	42.50
luatex	1.59	8.26	38.11
luajittex	1.12	5.57	25.48

The third example does a few 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	50	500	2500
pdftex	0.59	2.20	9.52
xetex	1.49	8.88	43.85
luatex	1.64	8.91	41.26
luajittex	1.15	5.91	27.15

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 than 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	50	500	2500
pdftex	0.61	2.36	10.33
xetex	1.53	9.25	45.59
luatex	1.65	8.91	41.32
luajittex	1.15	5.93	27.34

In these measurements the accuracy is a few decimals but a pattern is visible. As expected pdfTEX wins on simple documents but starts loosing 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	50	500	2500
pdftex	0.46	1.05	3.72
xetex	0.73	1.80	6.56
luatex	0.84	1.44	4.07
luajittex	0.61	1.10	3.33

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	50	500	2500
pdftex	0.36	0.36	0.36
xetex	0.57	0.57	0.59
luatex	0.74	0.74	0.74
luajittex	0.53	0.53	0.54

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 a lot is 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 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 coming years so at some point we're stuck with what we have.

7. 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 frame rate, low color game with a modern one. You need to have up to date hardware and even then, the writer of such programs need to make sure it runs efficient, simply because hardware no longer scales like it did decades ago. You need to look at the larger picture.