Performance again

Introduction

In a MAPS article of 2019 I tried to answer the question ‘Is TEX really slow?’. A while after it was published on the Dutch TEX mailing list a user posted a comment stating that in his experience the LUATEX engine in combination with LATEX was terribly slow: one page per second for a Japanese text. It was also slower than PDFTEX with English, but for Japanese it was close to unusable. The alternative, using a Japanese TEX engine was no option due to lack of support for certain images.

In order to check this claim I ran a test in ConTEXt. Even on my 8 year old laptop I could get 45 pages per second for full page Japanese texts (6 paragraphs with each 300 characters per page): 167 pages took just less than 4 seconds. Typesetting Japanese involves specific spacing and line break handling. So, naturally the question arises: why the difference. Frans Goddijn wondered if I could explain a bit more about that, so here we go.

In the mentioned article I already have explained what factors play a role and the macro package is one of them. It is hard to say to what extent inefficient macros or a complex layout influence the runtime, but my experience is that it is pretty hard to get speeds as low as 1 page per second. On an average complex document like the LUATEX manual (lots of verbatim and tables, but nothing else demanding apart from color being used and a unique MetaPost graphic per page) I get at least a comfortable 20 pages per second.

I can imagine that for a TEX user who sees other programs on a computer do complex things fast, the performance of TEX is puzzling. But, where for instance rendering videos can benefit from specific features of (video) processors, multiple cores, or just aggressive optimization by compilers of (nested) loops and manipulation of arrays of bytes, this is not the case for TEX. This program processes all in sequence, there is not much repetition that can be optimized, it cannot exploit the processor in special ways and the compiler can not do that many optimizations.

I can’t answer why a LATEX run is slower than a ConTEXt run. Actually, one persistent story has always been that ConTEXt was slow in comparison. But maybe it helps to know a bit what happens deep down in TEX and how macro code can play a role in performance. When doing that I will simplify things a bit.

Text and nodes

The TEX machinery takes input and turns that into some representation that can be turned into a visual representation ending up as PDF. So say that we have this:

```
hello
```

In a regular programming language this is a string with five characters. When the string is manipulated it is basically still a sequence of bytes in memory. In TEX, if this is meant as text, at some point the internal representation is a so called node list:

```
[h] -> [e] -> [l] -> [l] -> [o]
```

In traditional TEX these are actually character nodes. They have a few properties, like what font the character is from and what the character code is (0 up to 255). At some point TEX will turn that list into a glyph list. Say that we have this:

```
efficient
```

This will eventually become seven nodes:

```
[e] -> [ffi] -> [c] -> [i] -> [e] -> [n] -> [t]
```

The ffi ligature is a glyph node which actually also keeps information about this one character being made from three.

In LUATEX it is different, and this is one of the reasons for it being slower. We stick to the first example:

```
[h] <-> [e] <-> [l] <-> [l] <-> [o]
```

So, instead of pointing to the next node, we also point back to the previous: we have a double linked list. This means that all over the program we need to maintain these extra links too. They are not used by TEX itself, but handy at the LUA end. But, instead of only having the font as property there is much more. The TEX program can deal with multiple languages at the same time and this relates to hyphenation. In traditional TEX there are language nodes that indicate a switch to another language. But in LUATEX that property is kept with each glyph node. Actually, even specific language properties like the hyphen min, hyphen max and the choice if uppercase should be hyphenated are kept with these nodes. Spaces are turned into glue nodes, and these nodes are also larger than in regular TEX engines.

So, in LUATEX, when a character goes from the input into a node, a more complex data structure has to be set up and the larger data structure also takes more memory. That in turn means that caching (close to the CPU) gets influenced. Add to that the fact that we operate on 32 bit character values, which also comes with higher memory demands.

We mentioned that a traditional engine goes from one state of node list into another (the ligature building). Actually this is an integrated process: a lot happens on the fly. If something is put into a \hbox no hyphenation takes place, only ligature building and kerning. When a paragraph is typeset, hyphenation happens on demand, in places where it makes sense.

In LUATEX these stages are split. A node list is *always* hyphenated. This step as well as ligature building and kerning are *three* separate steps. So, there's always more hyphenation going on than in a traditional TEX engine: we get more discretionary nodes and again these take more memory than before; also the more nodes we have, the more it will impact performance down the line. The reason for this is that each step can be intercepted and replaced by a LUA driven one. In practice, with modern OPENTYPE fonts that is what happens: these are dealt with (or at least managed in) LUA. For Japanese for sure the built-in ligature and kerning doesn't apply: the work is delegated and this comes at a price. Japanese needs no hyphenation but instead characters are treated with respect to their neighbors and glue nodes are injected when needed. This is something that LUA code is used for so here performance is determined by how well the plugged in code behaves. It can be inefficient but it can also be so clever that it just takes a bit of time to complete.

I didn't mention another property of nodes: attributes. Each node that has some meaning in the node list (glyphs, kerns, glue, penalties, discretionary, ..., these terms should ring bells for a TEX user) have a pointer to an attribute list. Often these are the same for neighboring nodes, but they can be different. If a macro package sets 10 attributes, then there will be lists of ten attributes nodes (plus some overhead) active. When values change, copies are made with the change applied. Grouping even complicates this a little more. This has an impact on performance. Not only need these lists be managed, when they are consulted at the LUA end (as they are meant as communication with that bit of the engine) these lists are interpreted. It all adds up to more runtime. There is nothing like that in traditional TEX, but there some more macro juggling to achieve the same effects can cause a performance hit.

Macros and tokens

When you define macro like this:

```
\def\MyMacro#1{\hbox{here: #1!}}
```

the TEX engine will parse this as follows (we keep it simple):

\def	primitive token
\MyMacro	user macro pointing to:
#1	argument list of length 1 and no delimiters
{	openbrace token
\hbox	hbox primitive token
h	letter token h
e	letter token e
r	letter token r
e	letter token e
:	other token :
	space token
#1	reference to argument
!	other token !
}	close brace token

The \def is eventually lost, and the meaning of the macro is stored as a linked list of tokens that get bound to the user macro \MyMacro. The details about how this list is stored internally can differ a bit per engine but the idea remains. If you compare tokens of a traditional TEX engine with LUATEX, the main difference is in the size: those in LUATEX take more memory and again that impacts performance.

Processing

Now, for a moment we step aside and look at a regular programming language, like PASCAL, the language TEX is written in, or C that is used for LUATEX. The high level definitions, using the syntax of the language, gets compiled into low level machine code: a sequence of instructions for the CPU. When doing so the compiler can try to optimize the code. When the program is executed all the CPU has to do is fetch the instructions, and execute them, which in turn can lead to fetching data from memory. Successive versions of CPU's have become more clever in handling this, predicting what might happen, (pre) fetching data from memory etc.

When you look at scripting languages, again a high level syntax is used but after interpretation it becomes compact so called byte-code: a sequence of instructions for a virtual machine that itself is a compiled program. The virtual machine fetches the bytes and acts upon them. It also deals with managing memory and variables. There is not much optimization going on there, certainly not when the language permits dynamically changing function calls and such. Here performance is not only influenced by the virtual machine but also by the quality of the original code (the scripts). In LUATEX we're talking Lua here, a scripting language that is actually considered to be pretty fast.

Sometimes byte-code can be compiled Just In Time into low level machine code but for LuaTEX that doesn't work out well. Much Lua code is executed only once or a few times so it simply doesn't pay off. Apart from that there are other limitations with this (in itself impressive) technology so I will not go into more detail.

So how does TEX work? It is important to realize that we have a mix of input and macros. The engine interprets that on the fly. A character enters the input and TEX has to look at it in the perspective of what it what it expects. Is is just a character? Is it part of a control sequence that started (normally) with a backslash? Does it have a special meaning, like triggering math mode? When a macro is defined, it gets stored as a linked list of tokens and when it gets called the engine has to expand that meaning. In the process some actions themselves kind of generate input. When that happens a new level of input is entered and further expansion takes place. Sometimes TEX looks ahead and when not satisfied, pushes something back into the input which again introduces a new level. A lot can happen when a macro gets expanded. If you want to see this, just add `\tracingall` at the top of your file: you will be surprised! You will not see how often tokens get pushed and popped but you can see how much got expanded and how often local changes get restored. By the way, here is something to think about:

```
\count4=123
\advance \count4 by 123
```

If this is in your running text, the scanner sees `\count` and then triggers the code that handles it. That code expects a register number, here that is the `4`. Then it checks if there is an optional `=` which means that it has to look ahead. In the second line it checks for the optional keyword `by`. This optional scanning has a side effect: when the next token is *not* an equal or keyword, it has to push back what it just read (we enter a new input level) and go forward. It then scans a number. That number ends with a space or `\relax` or something not being a number. Again, some push back onto the input can happen. In fact, say that instead of `4` we have a macro indicating the register number, intermediate expansion takes place. So, even these simple lines already involve a lot of action! Now, say that we have this

```
\scratchcounter 123
\scratchcounter =123
\advance\scratchcounter by 123
\advance\scratchcounter 123
```

Can you predict what is more efficient? If this doesn't happen a lot performance wise there is no real difference because TEX is pretty fast in doing this, but given what we said before, adding the equal sign and `by` *could* actually be faster because there is no pushing back onto the input stack involved. It probably makes no sense to take this into account when writing macros but just keep in mind that performance is in the details.

This model of expansion is very different from compiled code or byte-code. To some extent you can consider a list of tokens that make up a macro to be byte-code, but instead of a sequence of bytes it is a linked list. That itself has a penalty in performance. Depending on how macros expand, the engine can be hopping all over the token memory following that list. That means that quite likely the data that gets accessed is not in your CPU cache and as a result performance cannot benefit from it apart of course from the expanding machinery itself, but that one is not a simple loop messing around with variables: it accesses code all over the place! Text gets hyphenated, fonts get applied, material gets boxed, paragraphs constructed, pages built. We're not moving a blob of bits around (as in a video) but we're constantly manipulating small amounts of memory scattered around memory space.

Now, where a traditional TEX engine works on 8 bit characters and smaller tokens, the 32 bit LuaTEX works on larger chunks. Although macro names are stored as single symbolic units, there are times when its real name is used, for instance when the `\csname` primitive is used. At that time, the real name is used and there are plenty cases where temporary string variables are allocated and filled. Compare:

```
\def\foo{\hello}
```

Here the macro `\foo` has just a one token reference to `\hello` because that's how a macro reference gets stored. But in

```
\def\foo{\csname hello\endcsname}
```

we have two plus five tokens to access what effectively is `\hello`. Each character token has to be converted to a byte into the assembled string. Now it must be said that in practice this is still pretty fast but when we have longer names and especially when we have UTF8 characters in there it can come at a price. It really depends on how your macro package works and sometimes you just pay the price of progress. Buying a faster machine is then the solution because often we're not talking of extreme performance loss here. And modern CPU's can juggle bytes quite efficiently. Actually, when we go to 64 bit architectures, LuaTEX's data structures fit quite well to that. As a side note: when you run a 32 bit binary on a 64 bit architecture there can even be a price being paid for that when you use LuaTEX. Just move on!

Management

Before we can even reach the point that some content becomes typeset, much can happen: the engine has to start up. It is quite common that a macro package uses a memory dump so that macros are not to be parsed each run. In traditional engines hyphenation patterns are stored in the memory dump as well. And some macro packages can put fonts in it. All kind of details, like upper- and lowercase codes can get stored too. In LuaTeX fonts and patterns are normally kept out of the dump. That dump itself is much larger already because we have 32 bit characters instead of 8 bit so more memory is used. There are also new concepts, like catcode tables that take space. Math is 32 bit too, so more codes related to math are stored. Actually the format is so much larger that LuaTeX compresses it. Anyway, it has an impact on startup time. It is not that much, but when you measure differences on a one page document the overhead in getting LuaTeX up and running will definitely impact the measurement.

The same is true for the backend. A traditional engine uses (normally) Type1 fonts and LuaTeX relies on OpenType. So, the backend has to do more work. The impact is normally only visible when the document is finalized. There can be a slightly larger hickup after the last page. So, when you measure one page performance, it again pollutes the page per second performance.

Summary

So, to come back to the observation that LuaTeX is slower that pdfTeX. At least for ConTeXt we can safely conclude that indeed pdfTeX is faster when we talk about a standard English document, with TeX ascii input, where we can do with traditional small fonts, with only some kerning and simple ligatures. But as soon as we deal with for instance xml, have different languages and scripts, have more demanding layouts, use color and images, and maybe even features that we were not aware of and therefore didn't require in former times the LuaTeX engine (and for ConTeXt it's LuaMetaTeX follow up) performs way better than pdfTeX. So, there is no simple answer and explanation for the fact that the observed slow LaTeX run on Japanese text, apart from that we can say: look at the whole picture: we have more complex tokens, nodes, scripts and languages, fonts, macros, demands on the machinery, etc. Maybe it is just the price you are paying for that.

Hans Hagen
Hasselt NL
Februari 2020