LuaTEX and ConTEXt: Where we stand

Hans Hagen

Abstract

We consider the release of LuaTEX 0.50 to be a very important one, both for LuaTEX and for MkIV, so here I will reflect on the state around this release. I will do this from the perspective of processing documents because usability is an important measure.

1 Where do we stand?

There are several reasons why LuaTEX 0.50 is an important release, both for LuaTEX and for MkIV. Let's start with LuaTEX.

- Apart from a couple of bug fixes, the current version is pretty usable and stable. Details of what we've reached so far have been presented previously.
- The code base has been converted from Pascal to C code, and as a result the source tree has become simpler (being CWEB compliant happens around 0.60). This transition also opens up the possibility to start looking into some of the more tricky internals, like page building.
- Most of the front end has been opened up and the new backend code is getting into shape. As the backend was partly already done in C code the moment has come to do a real cleanup. Keep in mind that we started with pdfTEX and that much of its extra functionality is rather interwoven with traditional TEX code.

If we look at ConTEXt, we've also reached a crucial point in the upgrade.

- The code base is now divided into MkII and MkIV. This permits us not only to reimplement bits and pieces (something that was already in progress) but also to clean up the code (only MkIV).
- If you kept up with the development you already know the kind of tasks we can (and do) delegate to Lua. Just to mention a few: file handling, font loading and OpenType processing, casing and some spacing issues, everything related to graphics and MetaPost, language support, color and other attributes, input regimes, XML, multipass data, etc.
- Recently all backend related code was moved to Lua and the code dealing with hyperlinks, widgets and alike is now mostly moved away from TEX. The related cleanup was possible because we no longer have to deal with a mix of DVI drivers too.
- Everything related to structure (which includes numbering and multi-pass data like tables of contents and registers) is now delegated to Lua. We move around way more information and will extend these mechanisms in the near future.

2 Performance testing

Tracing on Taco's machine has shown that when processing the LuaTEX reference manual the engine spends about 10% of the time on getting tokens, 15% on macro expansion, and some 50% on Lua (callback interfacing included). Especially the time spent by Lua differs per document and garbage collections seems to be a bottleneck here. So, let's wrap up how LuaTEX performs around the time of 0.50. We use three documents for testing (intermediate) LuaTEX binaries: the reference manual, the history document 'mk', and the revised Metafun manual.

The reference manual has a MetaPost graphic on each page which is positioned using the ConTEXt background layering mechanism. This mechanism is active only when backgrounds are defined and has some performance consequences for the page builder. However, most time is spent on constructing the tables (tabulate) and because these can contain paragraphs that can run over multiple pages, constructing a table takes a few analysis passes per table plus some so-called vsplitting. We load some fonts (including narrow variants) but for the rest this document is not that complex. Of course colors are used as well as hyperlinks. The report at the end of these runs is shown in figure 1.

The runtime is influenced by the fact that some startup time and font loading takes place. The more pages your document has, the less the runtime is influenced by this.

More demanding is the 'mk' document (figure 2). Here we have many fonts, including some really huge CJK and Arabic ones (and these are loaded at several sizes and with different features). The reported font load time is large but this is partly due to the fact that on my machine for some reason passing the tables to TEX involved a lot of pagefaults (we think that the cpu cache is the culprit). Older versions of LuaTEX didn't have that performance penalty, so probably half of the reported font loading time is kind of wasted.

The hnode processing time refers mostly to OpenType font processing and attribute processing time has to do with backend issues (like injecting color directives). The more features you enable, the larger these numbers get. The MetaPost font loading refers to the punk font instances.

```
input load time           - 0.109 seconds
stored bytecode data      - 184 modules, 45 tables, 229 chunks
node list callback tasks  - 4 unique tasks, 4 created, 20980 calls
cleaned up reserved nodes - 29 nodes, 10 lists of 1427
node memory usage         - 19 glue_spec, 2 dir
h-node processing time    - 0.312 seconds including kernel
attribute processing time - 1.154 seconds
used backend              - pdf (backend for directly generating pdf output)
loaded patterns           - en:us:pat:exc:2
jobdata time              - 0.078 seconds saving, 0.047 seconds loading
callbacks                 - direct: 86692, indirect: 13364, total: 100056
interactive elements      - 178 references, 356 destinations
v-node processing time    - 0.062 seconds
loaded fonts              - 43 files: ....
fonts load time           - 1.030 seconds
metapost processing time  - 0.281 seconds, loading: 0.016 seconds,
                            execution: 0.156 seconds, n: 161
result saved in file      - luatexref-t.pdf
luatex banner             - this is luatex, version beta-0.42.0
control sequences         - 31880 of 147189
current memory usage      - 106 MB (ctx: 108 MB)
runtime                   - 12.433 seconds, 164 processed pages,
                            164 shipped pages, 13.191 pages/second
```

Figure 1: Timing reports for the LuaTEX reference manual.

```
input load time           - 0.125 seconds
stored bytecode data      - 184 modules, 45 tables, 229 chunks
node list callback tasks  - 4 unique tasks, 4 created, 24295 calls
cleaned up reserved nodes - 116 nodes, 29 lists of 1411
node memory usage         - 21 attribute, 23 glue_spec, 7 attribute_list,
                            7 local_par, 2 dir
h-node processing time    - 1.763 seconds including kernel
attribute processing time - 2.231 seconds
used backend              - pdf (backend for directly generating pdf output)
loaded patterns           - en:us:pat:exc:2 en-gb:gb:pat:exc:3 nl:nl:pat:exc:4
language load time        - 0.094 seconds, n=4
jobdata time              - 0.062 seconds saving, 0.031 seconds loading
callbacks                 - direct: 98199, indirect: 20257, total: 118456
xml load time             - 0.000 seconds, lpath calls: 46, cached calls: 31
v-node processing time    - 0.234 seconds
loaded fonts              - 69 files: ....
fonts load time           - 28.205 seconds
metapost processing time  - 0.421 seconds, loading: 0.016 seconds,
                            execution: 0.203 seconds, n: 65
graphics processing time  - 0.125 seconds including tex, n=7
result saved in file      - mk.pdf
metapost font generation  - 0 glyphs, 0.000 seconds runtime
metapost font loading     - 0.187 seconds, 40 instances,
                            213.904 instances/second
luatex banner             - this is luatex, version beta-0.42.0
control sequences         - 34449 of 147189
current memory usage      - 454 MB (ctx: 465 MB)
runtime                   - 50.326 seconds, 316 processed pages,
                            316 shipped pages, 6.279 pages/second
```

Figure 2: Timing reports for the 'mk' document.

Looking at the Metafun manual one might expect that one needs even more time per page but this is not true. We use OpenType fonts in base mode as we don't use fancy font features (base mode uses traditional TEX methods). Most interesting here is the time involved in processing MetaPost graphics. There are a lot of them (1772) and in addition we have 7 calls to independent ConTEXt runs that take one third of the total runtime. About half of the runtime involves graphics. See figure 3.

By now it will be clear that processing a document takes a bit of time. However, keep in mind

```
input load time           - 0.109 seconds
stored bytecode data      - 184 modules, 45 tables, 229 chunks
node list callback tasks  - 4 unique tasks, 4 created, 33510 calls
cleaned up reserved nodes - 39 nodes, 93 lists of 1432
node memory usage         - 249 attribute, 19 glue_spec, 82 attribute_list,
                            85 local_par, 2 dir
h-node processing time    - 0.562 seconds including kernel
attribute processing time - 2.512 seconds
used backend              - pdf (backend for directly generating pdf output)
loaded patterns           - en:us:pat:exc:2
jobdata time              - 0.094 seconds saving, 0.031 seconds loading
callbacks                 - direct: 143950, indirect: 28492, total: 172442
interactive elements      - 214 references, 371 destinations
v-node processing time    - 0.250 seconds
loaded fonts              - 45 files: l.....
fonts load time           - 1.794 seconds
metapost processing time  - 5.585 seconds, loading: 0.047 seconds,
                            execution: 2.371 seconds, n: 1772,
                            external: 15.475 seconds (7 calls)
mps conversion time       - 0.000 seconds, 1 conversions
graphics processing time  - 0.499 seconds including tex, n=74
result saved in file      - metafun.pdf
luatex banner             - this is luatex, version beta-0.42.0
control sequences         - 32587 of 147189
current memory usage      - 113 MB (ctx: 115 MB)
runtime                   - 43.368 seconds, 362 processed pages,
                            362 shipped pages, 8.347 pages/second
```

Figure 3: Timing reports for the Metafun manual.

that these documents are a bit atypical. Although ... the average ConTEXt document probably uses color (including color spaces that involve resource management), and has multiple layers, which involves some testing of the about 30 areas that make up the page. And there is the user interface that comes with a price.

3 Fonts and performance

It might be good to say a bit more about fonts. In ConTEXt we use symbolic names and often a chain of them, so the abstract `SerifBold` resolves to `MyNiceFontSerif-Bold` which in turn resolves to `mnfs-bold.otf`. As XETEX introduced lookup by internal (or system) fontname instead of filename, MkII also provides that method but MkIV adds some heuristics to it. Users can specify font sizes in traditional TEX units but also relative to the body font. All this involves a bit of expansion (resolving the chain) and parsing (of the specification). At each of the levels of name abstraction we can have associated parameters, like features, fallbacks and more. Although these mechanisms are quite optimized this still comes at a performance price.

Also, in the default MkIV font setup we use a couple more font variants (as they are available in Latin Modern). We've kept definitions sort of dynamic so you can change them and combine them in many ways. Definitions are collected in typescripts which are filtered. We support multiple mixed font sets which takes a bit of time to define but switching is generally fast. Compared to MkII the model lacks the (font) encoding and case handling code (here we gain speed) but it now offers fallback fonts (replaced ranges within fonts) and dynamic OpenType font feature switching. When used we might lose a bit of processing speed although fewer definitions are needed which gets us some back. The font subsystem is anyway a factor in the performance, if only because more complex scripts or font features demand extensive node list parsing.

Processing *The TEXbook* with LuaTEX on Taco's machine takes some 3.5 seconds in pdfTEX and 5.5 seconds in LuaTEX. This is because LuaTEX internally is Unicode and has a larger memory space. The few seconds more runtime are consistent with this. One of the reasons that *The TEXbook* processes fast is that the font system is not that complex and has hardly any overhead, and an efficient output routine is used. The format file is small and the macro set is optimal for the task. The coding is rather low level so to say (no layers of interfacing). Anyway, 100 pages per second is not bad at all and we don't come close with ConTEXt and the kind of documents that we produce there.

4 Engine performance comparisons

This made me curious as to how fast really dumb documents could be processed. It does not make sense to compare plain TEX and ConTEXt because

they do different things. Instead I decided to look at differences in engines and compare runs with different numbers of pages. That way we get an idea of how startup time influences overall performance. We look at pdfTEX, which is basically an 8-bit system, XETEX, which uses external libraries and is Unicode, and LuaTEX which is also Unicode, but stays closer to traditional TEX but has to check for callbacks.

In our measurement we use a really simple test document as we only want to see how the baseline performs. As not much content is processed, we focus on loading (startup), the output routine and page building, and some basic PDF generation. After all, it's often a quick and dirty test that gives users their first impression. When looking at the times you need to keep in mind that XETEX pipes to DVIPDFM*x* and can benefit from multiple cpu cores. All systems have different memory management and garbage collection might influence performance (as demonstrated in an earlier chapter of the 'mk' document we can trace in detail how the runtime is distributed). As terminal output is a significant slowdown for TEX we run in batchmode. The test is as follows:

```
\starttext
    \dorecurse{2000}{test\page}
\stoptext
```

On my laptop (Dell M90 with 2.3Ghz T76000 Core 2 and 4MB memory running Vista) I get the following results. The test script ran each test set 5 times and we show the fastest run so we kind of avoid interference with other processes that take time. In practice runtime differs quite a bit for similar runs, depending on the system load. The time is in seconds and between parentheses the number of pages per seconds is mentioned.

engine	30	300	2000	10000
xetex	1.81 (16)	2.45 (122)	6.97 (286)	29.20 (342)
pdftex	1.28 (23)	2.07 (144)	6.96 (287)	30.94 (323)
luatex	1.48 (20)	2.36 (127)	7.85 (254)	34.34 (291)

The next table shows the same test but this time on a 2.5Ghz E5420 quad core server with 16GB memory running Linux, but with 6 virtual machines idling in the background. All binaries are 64 bit.

engine	30	300	2000	10000
xetex	0.92 (32)	1.89 (158)	8.74 (228)	42.19 (237)
pdftex	0.49 (61)	1.14 (262)	5.23 (382)	24.66 (405)
luatex	1.07 (27)	1.99 (150)	8.32 (240)	38.22 (261)

A test demonstrated that for LuaTEX the 30 and 300 page runs take 70% more runtime with 32 bit binaries (recent binaries for these engines are available on the ConTEXt wiki `contextgarden.net`).

When you compare both tables it will be clear that it is non-trivial to come to conclusions about performances. But one thing is clear: LuaTEX with ConTEXt MkIV is not performing that badly compared to its cousins. The Unicode engines perform about the same and pdfTEX beats them significantly. Okay, I have to admit that in the meantime some cleanup of code in MkIV has happened and the LuaTEX runs benefit from this, but on the other hand, the other engines are not hindered by callbacks. As I expect to use MkII less frequently optimizing the older code makes no sense.

5 Futures

There is not much chance of LuaTEX itself becoming faster, although a few days before writing this Taco managed to speed up font inclusion in the backend code significantly (we're talking about half a second to a second for the three documents used here). On the contrary, when we open up more mechanisms and have upgraded backend code it might actually be a bit slower. On the other hand, I expect to be able to clean up some more ConTEXt code, although we already got rid of some subsystems (like the rather flexible (mixed) font encoding, where each language could have multiple hyphenation patters, etc.). Also, although initial loading of math fonts might take a bit more time (as long as we use virtual Latin Modern math), font switching is more efficient now due to fewer families. But speedups in the ConTEXt code might be compensated for by more advanced mechanisms that call out to Lua. You will be surprised by how much speed can be improved by proper document encoding and proper styles. I can try to gain a couple more pages per second by more efficient code, but a user's style that does an inefficient massive font switch for some 10 words per page easily compensates for that.

When processing the present 10 page document in an editor (Scite) it takes some 2.7 seconds between hitting the processing key and the result showing up in Acrobat. I can live with that, especially when I keep in mind that my next computer will be faster.

This is where we stand now. The three reports shown before give you an impression of the impact of LuaTEX on ConTEXt. To what extent is this reflected in the code base? Eventually most MkII files (with the `mkii` suffix) and MkIV files (with suffix `mkiv`) will differ and the number of files with the `tex` suffix will be fewer. Because they are and will be mostly downward compatible, styles and modules will be shared as much as possible.

◇ Hans Hagen
Pragma ADE
`http://pragma-ade.com`