From Lua 5.2 to 5.3

Hans Hagen

When we started with LuaTEX we used Lua 5.1 and then moved seamlessly to 5.2 when that became available. We didn't run into issues with this language version change because there were no fundamental differences that could not be easily dealt with. However, when Lua 5.3 was announced in 2015 we were not sure if we should make the move. The main reason was that we'd chosen Lua because of its clean design part of which meant that we had only one number type: double. In 5.3 on the other hand, deep down a number can be either an integer or a floating point quantity.

Internally TEX is mostly (up to) 32-bit integers so when we go from Lua to TEX we are forced to round numbers. Nonetheless, or perhaps because of this, one can expect some benefits in using integers in Lua. Performance-wise we didn't expect much, and memory consumption would be the same too. So the main question then was: can we get the same output and not run into trouble due to possible differences in serializing numbers? After all TEX is about stability. The serialization aspect is for instance important when we compare quantities and/or use numbers in hashes, so one must be careful.

Apart from this change in the number model (which comes with a few extra helpers), another interesting extension in 5.3 was that bit-wise operations are now part of the language. However, the lpeg library is still not part of stock Lua. There is also added some minimal utf8 support, but less than we provide in LuaTEX already. So, considering these changes, we were not in a big hurry to update. Also, it made sense to wait until this important number-related change became stable.

But, a few years later, we still had it on our agenda to test the new version of Lua, and after the ConTEXt 2017 meeting we decided to give it a try; here are some observations. A quick test involved just dropping in the new Lua code and seeing if with this we could still compile a ConTEXt format. Indeed that was no big deal but the test run failed because at some point a (for instance) `1` became a `1.0`. It turned out that serializing has some side effects, and with some ad hoc prints for tracing (in the LuaTEX source) I could figure out what was going on. How numbers are seen can (to some extent) be deduced from the `string.format` function, which is in Lua a combination of parsing, splitting and concatenation combined with piping to the C `sprintf` function:[1]

[1] Actually, at some point I decided to write my own formatter on top of `format` and I ended up with splitting as well. It's only now that I realize why this is working out so well (in terms of performance): simple format (single items) are passed more or less directly to `sprintf` and as Lua itself is fast, due to some caching, the overhead is small compared to the built-in splitter method. An advantage is that the ConTEXt formatter has many more options and is also extensible.

```
local a =  2   * (1/2) print(string.format("%s",  a),math.type(x))
local b =  2   * (1/2) print(string.format("%d",  b),math.type(x))
local c =  2           print(string.format("%d",  c),math.type(x))
```

```
local d = -2           print(string.format("%d",  d),math.type(x))
local e =  2   * (1/2) print(string.format("%i",  e),math.type(x))
local f =  2.1         print(string.format("%.0f",f),math.type(x))
local g =  2.0         print(string.format("%.0f",g),math.type(x))
local h =  2.1         print(string.format("%G",  h),math.type(x))
local i =  2.0         print(string.format("%G",  i),math.type(x))
local j =  2           print(string.format("%.0f",j),math.type(x))
local k = -2           print(string.format("%.0f",k),math.type(x))
```

This gives the following results:

a	2 * (1/2)	s	1.0	float
b	2 * (1/2)	d	1	float
c	2	d	2	integer
d	-2	d	2	integer
e	2 * (1/2)	i	1	float
f	2.1	.0f	2	float
g	2.0	.0f	2	float
h	2.1	G	2.1	float
i	2.0	G	2	float
j	2	.0f	2	integer
k	-2	.0f	2	integer

This demonstrates that we have to be careful when we need numbers represented as strings. In ConTEXt the places where we had to check for this was not that many: in fact, only some hashing related to font sizes had to be done using explicit rounding.

Another surprising side effect is the following. Instead of:

```
local n = 2^6
```

we now need to use:

```
local n = 0x40
```

or just:

```
local n = 64
```

because we don't want this to be serialized to `64.0` which is due to the fact that a power results in a float. One can wonder if this makes sense when we apply it to an integer.

At any rate, once we were able to process a file, two standard documents were chosen for a performance test. Some experiments with loops and casts had demonstrated that we could expect a small performance hit and indeed, this was the case. Processing the LuaTEX manual takes 10.7 seconds with 5.2 on my 5-year-old laptop and 11.6 seconds with 5.3. If we consider that ConTEXt spends about 50% of its time in Lua, then we find here a 20% performance penalty using the later version of Lua. Processing the MetaFun manual (which has lots of MetaPost images) went from less than 20 seconds (and LuajitTEX does it in 16 seconds) to up to more than 27 seconds. So there we lose more than 50% on the Lua end. When we observed these kinds of differences, Luigi and I immediately got into debugging mode, partly out of curiosity but also because consistent performance is always important to us.

As these results made no sense, we traced different sub-mechanisms and eventually it became clear that the reason behind the speed penalty was in fact that the core `string.format` function was behaving quite badly in the `mingw` cross-compiled binary, as can be seen by this test:

```
local t = os.clock()
for i=1,1000*1000 do
 -- local a = string.format("%.3f",1.23)
 -- local b = string.format("%i",123)
    local c = string.format("%s",123)
end
print(os.clock()-t)
```

	lua 5.3	**lua 5.2**	**texlua 5.3**	**texlua 5.2**
a	0.43	0.54	3.71 (0.47)	0.53
b	0.18	0.24	3.78 (0.17)	0.22
c	0.26	0.68	3.67 (0.29)	0.66

Both 5.2 binaries perform the same but the 5.3 Lua binary greatly outperforms the LuaTEX binary so we had to figure out why. After all, the integer optimization should bring some gain! It took us a while to figure out what was going wrong, and the numbers in parentheses are the results after fixing LuaTEX.

Because font internals are specified in integers one would expect a gain in running the command:

```
mtxrun --script font --reload force
```

and indeed that is the case. On my machine a scan results in 2561 registered fonts from 4906 read files and with 5.2 that takes 9.1 seconds while 5.3 needs a bit less: 8.6 seconds (with the bad cross-compiled format performance) and even less once that was fixed.

For a test:

```
\setupbodyfont[modern]     \tf \bf \it \bs
\setupbodyfont[pagella]    \tf \bf \it \bs
\setupbodyfont[dejavu]     \tf \bf \it \bs
\setupbodyfont[termes]     \tf \bf \it \bs
\setupbodyfont[cambria]    \tf \bf \it \bs
\starttext \stoptext
```

This code needs 30% more runtime using the newer version of Lua so the question is: how often do we call `string.format` there? A first run (when we wipe the font cache) needs some 715 000 calls while successive runs need 115 000 calls so the slow down definitely comes from the bad handling of `string.format`.

When we drop in a Lua or whatever other dependency update we don't want this kind of impact. In fact, when one uses external libraries that are or can be compiled under the TEX Live infrastructure and the impact would be so dramatic, this would be very bad advertising, especially when one considers the occasional complaint about LuaTEX being slower than other engines.

The good news is that eventually Luigi was able to nail down this issue and we got a binary that performed well. It looks like Lua 5.3.4 (cross)compiles badly under both gcc 5.3.0 and 6.3.0.

So in the end loading the fonts takes:

	caching	**running**
5.2 stock	8.3	1.2
5.3 bugged	12.6	2.1
5.3 fixed	6.3	1.0

So indeed after an initial scare it looks like 5.3 is able to speed up LuaTEX a bit, given that one integrates it in the right way! The use of a recent compiler is needed here, although one can wonder when another bad case will show up again. One can also wonder why such a slow down can mostly go unnoticed, because for sure LuaTEX is not the only compiled program integrating the Lua language.[2]

The next examples are some edge cases that show you need to be aware that (1) an integer has its limits, (2) that hexadecimal numbers are integers, and (3) that Lua 5.2 and LuaJIT can differ in small details:

	print(0xFFFFFFFFFFFFFFFF)	**print(0x7FFFFFFFFFFFFFFF)**
lua 52	`1.844674407371e+019`	`9.2233720368548e+018`
luajit	`1.844674407371e+19`	`9.2233720368548e+18`
lua 53	`-1`	`9223372036854775807`

2 We can only speculate that others do not pay such close attention to performance.

We see here that Lua 5.3 clearly represents some progress.

So, to summarize the migration, a quick test was relatively easy: move 5.3 into the code base, make slight adaptations to the internals (there were a few LuaTEX interfacing bits where explicit rounding was needed), run tests, and eventually fix some issues related to the Makefile (compatibility) and C obscurities (the very slow `sprintf`).[3]

Adapting ConTEXt was also not much work, but the test suite uncovered some nasty side effects. For instance, the valid 5.2 solution:

```
local s = string.format("02X",u/1024)
local s = string.char        (u/1024)
```

now has to become (works with both 5.2 and 5.3):

```
local s = string.format("02X",math.floor(u/1024))
local s = string.char        (math.floor(u/1024))
```

or (with 5.2 and emulated or real 5.3):

```
local s = string.format("02X",bit32.rshift(u,10))
local s = string.char        (bit32.rshift(u,10))
```

or (5.3 only):

```
local s = string.format("02X",u >> 10))
local s = string.char        (u >> 10)
```

or (5.3 only):

```
local s = string.format("02X",u//1024)
local s = string.char        (u//1024)
```

Unfortunately, adapting a conditional section like:

```
if LUAVERSION >= 5.3 then
    local s = string.format("02X",u >> 10))
    local s = string.char        (u >> 10)
else
    local s = string.format("02X",bit32.rshift(u,10))
    local s = string.char        (bit32.rshift(u,10))
end
```

[3] This demonstrates the importance of compilers, or rather how one writes code with respect to each compiler.

will fail because (of course) the 5.2 parser doesn't like the 5.3 syntax elements. In ConTEXt we have some experimental solutions for this but it is beyond the scope of this summary.

In the process of this update a few utf helpers were added to the string library so that we have a common set for both LuaJIT and Lua (the `utf8` library that was added to 5.3 is not very useful for LuaTEX). For now we also keep the `bit32` library on board, of course, we'll not mention all the details here.

When we consider a gain in speed of 5–10% with 5.3 that also means that the gain obtained using LuajitTEX compared to Lua 5.2 becomes less important. For instance in font processing both engines (Lua 5.3 and LuaJIT) now perform roughly to the same.

As I write this, we've just entered 2018 and after a few months of testing LuaTEX with Lua 5.3 we're confident that we can move the code to the experimental branch. This means that we will use this version in the ConTEXt distribution and likely will ship this as 1.10 in 2019 where Lua 5.3 becomes the default. The 2018 version of TEX Live will have 1.07 with Lua 5.2 while intermediate versions of the Lua 5.3 binary will end up on the ConTEXt garden, probably with number 1.08 and 1.09 (who knows what else we will add or change in the meantime).

Addendum

Around the 2018 meeting I also started what is to become the next major upgrade of ConTEXt, this time using a new engine LuaMetaTEX. In working on that I decided to try Lua 5.4 to see what consequences this new version would have for us. There are no real conceptual changes as were found with the number model in 5.3, so the tests didn't reveal any real issues. But as an additional step towards a bit cleaner distinction between strings and numbers, I decided to disable the automatic casting so that mixing strings and numbers in expression for instance is no longer permitted. If I remember correctly, there was only in one place I had to adapt the source (and we're talking about a pretty large Lua code base).

There is a new mechanism in Lua for freezing constants but I'm not yet sure if it makes much sense to use it. It's use goes along with some other restrictions, like the possibility to adapt loop counters inside the loop. Inside the body of a loop one could always adapt such a variable, which (I can imagine) can come in handy. I haven't checked the source code for that, but probably I don't do this anywhere.

Another new feature is an alternative garbage collector which seems to perform better when there are many variables with a short life spans. At least for now I have decided to default to this variant in future releases.

Overall the performance of Lua 5.4 is better than its predecessors which means that the gap between LuaTEX and LuajitTEX is closed or is closing. This is good because I have chosen not to support LuaJIT in LuaMetaTEX.