Foreign Function Interface in LuaTEX

Hans Hagen, Luigi Scarso

Abstract

We present a new Foreign Function Interface module for LuaTEX with the same API as `ffi` module. The paper shows some simple applications of a direct binding to a shared library and some tests done replacing the standard ConTEXt's Lua text shaper with Harfbuzz library's one. The results show that Harfbuzz performs better than Lua with some complex scripts like Arabic, while Lua is usually the best choice with a common Latin script. In that respect `jit` might come into the picture.

Sommario

Presentiamo un nuovo modulo Foreign Function Interface per LuaTEX con la stessa API di LuajitTEX. L'articolo illustra alcuni semplici casi di collegamento diretto con una libreria dinamica ed alcuni test fatti sostituendo il *text shaper* in Lua di ConTEXt con quello della libreria Harfbuzz. I risultati mostrano che Harfbuzz ha prestazioni migliori nel caso di alfabeti complessi come l'*arab*, mentre Lua è la scelta migliore per quelli latini. In questo contesto `jit` potrebbe avere un ruolo importante.

1 Introduction

The `libffi` library introduction (see https://github.com/libffi/libffi/tree/master/doc) gives the best explanation of what a 'Foreign Function Interface' is:

> "Compilers for high level languages generate code that follows certain conventions. These conventions are necessary, in part, for separate compilation to work. One such convention is the "calling convention". The calling convention is a set of assumptions made by the compiler about where function arguments will be found on entry to a function. A calling convention also specifies where the return value for a function is found. The calling convention is also sometimes called the "ABI" or "Application Binary Interface".
>
> Some programs may not know at the time of compilation what arguments are to be passed to a function. For instance, an interpreter may be told at run-time about the number and types of arguments used to call a given function. 'Libffi' can be used in such programs to provide a bridge from the interpreter program to compiled code.
>
> [...A] "libffi" library provides a portable, high level programming interface to various calling conventions. This allows a programmer to call any function specified by a call interface description at run time.
>
> FFI stands for Foreign Function Interface. A foreign function interface is the popular name for the interface that allows code written in one language to call code written in another language."

LuajitTEX provides the `ffi` module which offers in a very compact way almost the same functionality of `libffi` with the following remarkable differences:

- `ffi` supports less architecture/operating system pairs than `libffi`;
- `ffi` compiles and executes C declarations just in time, while `libffi` has no C parser.

Since version 1.0.3 (expected to appear in the mid of February 2017) even LuaTEX has a first implementation of `ffi` based on `luaffifb` (see https://github.com/facebook/luaffifb) that is still at an experimental stage. In the next sections we will show some applications of this module.

2 A first example

In the following code we will see how to directly use a C type. We declare a type `matrix` of size $N \times N$ as a linearly indexed array of `double`. Then we declare I as the identity matrix, $M : M[i,j] = (i+1)/(j+1)$, and perform $N = M \cdot I$, to check later that $N = M$.

```
\directluacode{
if  jit then
 --[==[ Luajittex ?  ]==]
 ffi = require("ffi")
 jit.on()
end
--[==[ ffi could be also disabled for this Arch/OS ]==]
if (type(ffi)=='table' and  ffi.os=='' and ffi.arch=='')
  then return
end
--[==[ ffi could be also fully disabled at runtime ]==]
if (type(ffi)=='table' and ffi.os==nil and ffi.arch==nil)
 then return
end
local C = ffi.C
```

```
--[==[ interface ]==]
ffi.cdef[[
  enum {N=256};
  typedef double matrix[N*N];
]]
local I = ffi.new('matrix')
for i=0,ffi.C.N-1 do I[i*C.N+i] = 1 end
local M = ffi.new('matrix')
for i=0,C.N-1 do
   for j=0,C.N-1 do M[j*C.N+i] = (i+1)/(j+1) end
end
local N = ffi.new('matrix')
for i=0, C.N-1 do
 for j=0, C.N-1 do
    N[j*C.N+i] = 0
    for k=0,C.N-1 do
     N[j*C.N+i] =
        M[j*C.N+k] * I[k*C.N+i]+N[j*C.N+i]
    end
 end
end
for i=0,C.N-1 do
   for j=0,C.N-1 do
     if M[j*C.N+i] ~= N[j*C.N+i] then
       print("TEST FAILED",
          M[j*C.N+i],N[j*C.N+i])
     end
    end
end
}
\end
```

First some words on the checks. As it is now, `ffi` is enabled if and only if `--shell-escape` is true and `--shell-restricted` is false (the `cnf` files are also considered) because `ffi` is inherently insecure (as we will discuss in greater detail later). Currently not all of the arch/OS pairs supported by LuajitTex are supported by `ffi`—practically only Intel x86_64 on Linux and OSX and Windows. However, support on Windows has some limitations due to a difference in underlying libraries. For normal usage this is not a problem. There is also a slight difference between LuaTEX, where `ffi` is globally available, and LuajitTEX, where it's only available on demand. The usual pattern is parsing a C *interface* and then using the declared types. In this example it's clear that the goal is to measure the time of pure computation and indeed `luajittex --luaonly` shows that the '`ffi`' version is about 1.8 times faster than the pure Lua one (on an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz) with `jit.on()`) but the same code with `luatex --luaonly` is about 7.6 times *slower*. This is a first indication that there is still a lot of work to be done on the jit side.

The second example (courtesy of A. Kakuto) is more interesting: it shows how to call a function from the C run-time (also known as `libc` in Unix):

```
% courtesy of A. Kakuto
\documentclass{article}
\usepackage{luacode}
\begin{document}
\begin{luacode*}
--[==[ the usual ffi checks here .. ]==]
ffi.cdef[[
  typedef const char* LPCSTR;
  int system(LPCSTR lpcstr);
]]
local msvcr100 = ffi.load("msvcr100")
msvcr100.system("echo abc")
\end{luacode*}
\end{document}
```

Several remarks: first, the run-time is operating system-specific, so the code is not portable. Second, this is a way to bypass the protections against the execution of untrusted programs, so it's now clear the reason why `ffi` is only enabled with a full `shell-escape`, and even in this case it's possible to disable it by putting at the very beginning of the format input file the following Lua code:

```
ffi=require[[ffi]];
for k,_ in pairs(ffi) do
 if k~='gc' then
  ffi[k]=nil
 end;
end;
ffi=nil;
```

The following, more interesting example ends the section: the computation of the Fast Fourier Transform (FFT) of a sampled sequence. The library used is the `libfftw3.so` from http://www.fftw.org/, also available for Windows as `libfftw3-3.dll`:

```
if not(ffi) then
 ffi = require("ffi")
 jit.on()
end
local PI = math.pi
local sin= math.sin
local function sinc(t)
 if t==0 then
  return 1
 else
  return sin(PI*t)/(PI*t)
 end
end
-- For windows:
-- local fftw3 = ffi.load("./libfftw3-3.dll")
local fftw3 = ffi.load("./libfftw3.so")
ffi.cdef([[
  enum {
   FFTW_ESTIMATE = (1 << 6)
  };
```

```
  typedef struct fftw_plan_s *fftw_plan;
  typedef double complex  fftw_complex;
  void *fftw_malloc(size_t);
  fftw_plan fftw_plan_dft_r2c_1d(int n, double *in,
               fftw_complex *out, unsigned flags);
  void fftw_execute(const fftw_plan p);
  void fftw_destroy_plan(fftw_plan p);
  void fftw_free(void *p);
]])
local N = 1024*1024
local data_in  = ffi.new("double[?]", N)
local data_out = ffi.new("fftw_complex[?]",N/2+1)
local p,t
for i=0, N-1 do
  t = i/2
  data_in[i] = sinc(t)
end
p= fftw3.fftw_plan_dft_r2c_1d(N, data_in, data_out,
                              ffi.C.FFTW_ESTIMATE)
fftw3.fftw_execute(p)
local string_format = string.format
local real,imag,m,f
local mag = io.open('mag.txt','w')
local ph  = io.open('phase.txt','w')
for i=0, N/2 do
 real,imag = data_out[i].re,data_out[i].im
 m= math.sqrt(real^2+imag^2)/(PI/2)
 f= math.atan2(imag,real)
 mag:write(string_format("%f\n", m))
 ph:write(string_format("%f\n", f))
end
fftw3.fftw_destroy_plan(p)
```

Even if it's unlikely that users will do demanding runtime calculations when typesetting a document, computation is quite fast: for 1MByte of samples the timings are approximately 1.2 s / 2.1 s with LuajitTEX (`jit.on()` / `jit.off()`) and 4.6 s for LuaTEX. The choice of the library is not accidental: the values of the FFT are `double complex`, but this type is not defined in the Microsoft C run-time so this code doesn't work; on the other hand, *it is* defined in the Windows MinGW `libgcc`, so the code works with MinGW. The solution could be `typedef double fftw_complex[2];` which is also accepted by `FFTW`, but this gives a segmentation fault in LuaTEX (while in LuajitTEX is valid: again, something to fix for the 2018 TEX Live release).

The next section shows the probably most important application of `ffi`: the usage of the Harfbuzz library (`hb`) as text shaper (a text shaper is the function that converts Unicode text into glyph indices and positions). One of the author of this paper is the leading developer of ConTEXt. Based on research by Kai Eigner he has extended the format to enable a third party text shaper as a *plug-in*; it's hence possible to perform an effective comparison between the ConTEXt's native Lua text shaper and `hb` (for the record: this will be a module based option but it is not recommended for regular use as it has drawbacks). What follows is a verbatim excerpt from the documentation of this new functionality, the *Plug mode*, with the warning that `ffi` it's still at an experimental stage and some details may change.

3 Plug mode

During a past NTG meeting, Kai Eigner and Ivo Geradts demonstrated how to use the Harfbuzz (`hb`) library to process OpenType fonts. The main reason for playing with that was twofold: it would provide a way to compare the Lua-based font machinery against other methods and it could give a better performance for complex fonts and/or scripts.

One of the guiding principles of LuaTEX development is to provide no hard-coded solution. For that reason we opened up the internals so that advanced users may provide solutions written in pure Lua or can cooperate with libraries via Lua code as well. Hard-coded solutions make no sense as there is usually more than a possible solution to a specific problem, depending on one's need. Although development is closely related to ConTEXt, the development of the LuaTEX engine is independent and we try to be macro package-agnostic. Starting from a very early development stage we made the ConTEXt font handler compatible with other macro packages, though users might want to use a lighter one for special purposes. So we also kept the standard TEX font handler and called it `base` mode in ConTEXt, while the Lua handler is `node` mode because it operates on the node list. We will later refer to these modes by their names instead of their programming languages. Supporting `hb` as a way to compare the Lua-based font machinery against other methods is not a strong reason: we already have XƎTEX as a milestone and, just in case we want to do a comparison against the standard, we have MS-Word.

The second reason (the look for better performance with complex fonts and/or scripts) could be significant for users because at least in a case the Lua variant performs better than the standard. Some fonts use many lookup steps or are even inefficient in using the available features. Anyway, up to now I haven't heard ConTEXt users complaining about speed. In fact, font handling became much faster during the latest few years, though probably no one noticed it. When using alternatives to the built-in methods, it is highly probable to lose some of the functionalities built into the current font system and/or its interactions with other mechanisms.

Just kicking in some alternative machinery is not the whole story. We still need to deal with the way TEX sees text: a sequence of glyph nodes, mixed with discretionary nodes for languages that hyphenate, glue, kern, boxes, math, and more. Discretionary nodes are those text elements most difficult to manage. In contextual analysis, as well as positioning, we need to process up to three extras: the text preceding the node, the text following the node and the text to replace the current with, along with the occasional links to the preceding and/or following nodes. In case the font has features and the user activates one or more of them, we have to process again all of those subsequences, keeping an eye on spaces, as they can be involved in lookups, and on glyphs injection or deletion, that can add or remove one or more attributes.

Kai and Ivo are plain TEX users so they use a font definition and switching environment that is quite different from ConTEXt. In a usual ConTEXt-run the time spent on font processing is measurable but it's not the main bottleneck because other time consuming operations get executed. Sometimes, the load on the font subsystem can be higher because we provide additional features normally not found in OpenType. Add to that a more dynamic font model, and it will be clear that comparing performance between situations that use different macro packages is not that trivial (and relevant).

Another reason why we follow a Lua route is that we support (run time generated) virtual fonts, we are able to kick in additional features, we can let the font mechanism cooperate with other functionalities, and so on. And the current mechanisms are likely to acquire more trickery in the near future. We also need access to various font pieces of information. Since we had to figure out a lot of these OpenType things a decade ago, when standards were fuzzy, we allowed some tracing and visualisation.

After his presentation, Kai wrote an article and that was the moment that I looked into the code and tried to replicate his experiments. Since we're talking of libraries, it's clear that the topic is all but trivial, especially because I'm on another platform than he is: Windows instead of OSX. The first thing that I have done was rewriting the code that connects the library to TEX in a more suitable way for ConTEXt. Mixing with existing modes (base or node mode) makes no sense and it is asking for unwanted interferences, so I preferred to write a new plug mode. A sort of general text filtering mechanism was derived from the original code so that we can plug in whatever we want. After all, stability is not the strongest point of contemporary software development so when we depend on a library, we need to be prepared to switch to other (library based) solutions too (for instance, if I understood correctly, XeTEX switched a few times).

After redoing the code the next step was to get the library running and that somehow failed due to some expected functions not being supported. At that time I thought it was a matter of interfacing. But, I could get around it by piping into the command line tools that come with the library and that was good enough for testing, although of course it was deadly slow. After that I just quit and moved on to something else.

Just before the publication of Kai's articles I tried the old code again and, surprise, after some messing around, the library worked. On my system the one shipped with Inkscape is used which is okay as it frees me from bothering about installations. I must admit that we have no real reason in ConTEXt for using fonts libraries but the interesting part was that it allowed me to play with the so called `ffi` interface of LuajitTEX. And, since that generates a nasty dependency, after a while Luigi Scarso and I managed to get a similar library working in stock LuaTEX (as that is the reference). So, I decided to give it a second try and in the process rewrote the interfacing code. After all, there is no reason for libraries not to be well-written and to have an optimised interface where possible.

Now, after a decade of writing Lua code, I dare to claim that I know a bit how to write relatively fast code so I was surprised to see that where Kai claimed that the library was faster than the Lua code, I saw that it really depends on the font. Sometimes the library approach is actually slower, which is not what I might expect. But, remember that complex fonts and scripts are a reason to use a library. What does 'complex' mean?

Most Latin fonts are not complex: ligatures and kerns and maybe a little bit of contextual analysis. Here the Lua variant is the clear winner. It runs up to ten times faster than the competitors. For more complex Latin fonts, like EBGaramond, which resolves ligatures in a different way, the library almost catches up, though the Lua handler keeps running faster. Keep in mind that we need to juggle discretionary nodes in any case. One difference between both methods is that the Lua handler runs over the whole lists (although it has to jump over fonts not being processed then) while the library gets snippets. However, tests show that the overhead involved in that is close to zero and can be neglected. Moreover, already long ago we have seen that when we compare MkIV LuaTEX and MkII XeTEX, the Lua based font handler is not that slow at all, which makes sense because the problem doesn't change and, maybe more important, Lua is a pretty fast language. If one or the other approach is less that two times faster, the gain will probably go unnoticed in real runs. In my experience a few bad choices in macro or style writing is more harmful than a bit slower font machinery. Indeed, if you add one more node processing step, you will not notice a measurable slow down. By the way, one reason why font handling has been sped up over the years is that our workflows sometimes have a high load and, for instance, remotely processing a set of 5 documents has to be fast. Also, in an edit workflow you want the runtime to be a bit comfortable.

Unlike Latin, a pure Arabic text has (normally) no discretionary nodes and the library profits most of that. I have to pick up the thread with Idris about the potential use of discretionary nodes in Arabic typesetting. On the contrary, Latin text has not so many replacements and positioning and therefore the LUA variant gets the advantage. Some of the additional features that the LUA variant provides can of course be provided for the library variant by adding some list pre- and post-processing but then you quickly lose any gain a library provides.

Kai's prototype has some cheats for `right2left` and special scripts like Devanagari. As these tweaks mostly involve discretionary nodes, there is no real need for them: when we don't hyphenate no time is wasted anyway. I didn't test Devanagari but there is some preprocessing needed in the LUA variant (provided by Kai and Ivo) that I might rewrite from scratch once I understand what happens there. I expect the library to perform somewhat better there. Eventually, I might add support for some more scripts that demand special treatments but so far I had no requests for it.

So, how about non-Latin? An experiment with Arabic—using the frequently used arabtype font—shows that the library performs faster, but when we use a mixed Latin and Arabic document the differences become less significant.

How did we measure? The baseline measurement is the so-called `none` mode: nothing is done there. It's fast but still takes a bit of time as it is triggered by a general mode identifying pass. That pass determines what font processing modes are needed for a list. `Base` mode only makes sense for Latin and has some limitations. It's fast and basically its run time can be neglected. That's why, for instance, PDFTEX is faster than the other engines, but it doesn't do Unicode well. NODE mode is the fancy name for the LUA font handler. The listed modes are ordered according to increasing run time. If we compare `node` mode with `plug` mode (in our case using the `hb` library), we can subtract `none` mode. This gives a cleaner (more distinctive) comparison but not a real honest one because there's always the need for the identifying pass.

We also performed a test with and without hyphenation but in practice that makes no sense: only verbatim text is typeset that way and normally we typeset that in none mode anyway. On the other hand, mixing fonts does happen. All of the tests start with forced garbage collection in order to get rid of that variance. We also pack into horizontal boxes so that the par builder (with all kind of associated callbacks doesn't kick in, although the node mode should compensate that).

The timings for LUATEX are the following.

luatex latin:

modern	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.50	0.07	-0.82	0.38	0.08
context node	1.32	0.88	0.00	1.00	1.00
context none	0.43	0.00	-0.88	0.33	0.00
harfbuzz native	5.20	4.77	3.89	3.95	5.40

pagella	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.54	0.07	-0.85	0.39	0.08
context node	1.39	0.92	0.00	1.00	1.00
context none	0.47	0.00	-0.92	0.34	0.00
harfbuzz native	5.16	4.69	3.77	3.72	5.12

dejavu	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.49	0.07	-1.42	0.26	0.04
context node	1.91	1.48	0.00	1.00	1.00
context none	0.43	0.00	-1.48	0.22	0.00
harfbuzz native	5.25	4.82	3.34	2.75	3.25

cambria	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.47	0.05	-1.86	0.20	0.03
context node	2.34	1.91	0.00	1.00	1.00
context none	0.43	0.00	-1.91	0.18	0.00
harfbuzz native	4.73	4.30	2.39	2.02	2.25

ebgaramond	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.52	0.08	-3.44	0.13	0.02
context node	3.96	3.52	0.00	1.00	1.00
context none	0.44	0.00	-3.52	0.11	0.00
harfbuzz native	4.96	4.53	1.00	1.25	1.28

lucidaot	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.50	0.02	-0.52	0.49	0.04
context node	1.02	0.54	0.00	1.00	1.00
context none	0.48	0.00	-0.54	0.47	0.00
harfbuzz native	4.47	3.99	3.45	4.40	7.41

luatex arabic:

arabtype	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.47	0.01	-16.52	0.03	0.00
context node	16.99	16.53	0.00	1.00	1.00
context none	0.46	0.00	-16.53	0.03	0.00
harfbuzz native	6.88	6.42	-10.11	0.41	0.39

luatex mixed:

arabtype	t	$t - t_\mathrm{n}$	$t - t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.73	0.04	-8.02	0.08	0.00
context node	8.75	8.06	0.00	1.00	1.00
context none	0.69	0.00	-8.06	0.08	0.00
harfbuzz native	5.84	5.15	-2.91	0.67	0.64

The timings for LuajitTEX are of course overall better:

luajittex latin:

modern	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.42	0.05	-0.42	0.50	0.11
context node	0.83	0.46	0.00	1.00	1.00
context none	0.37	0.00	-0.46	0.44	0.00
harfbuzz native	2.56	2.19	1.72	3.07	4.71

pagella	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.46	0.05	-0.42	0.52	0.10
context node	0.87	0.46	0.00	1.00	1.00
context none	0.41	0.00	-0.46	0.47	0.00
harfbuzz native	2.48	2.08	1.61	2.85	4.48

dejavu	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.40	0.04	-0.70	0.37	0.06
context node	1.10	0.74	0.00	1.00	1.00
context none	0.36	0.00	-0.74	0.33	0.00
harfbuzz native	2.52	2.16	1.42	2.29	2.91

cambria	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.40	0.04	-0.92	0.30	0.04
context node	1.31	0.96	0.00	1.00	1.00
context none	0.36	0.00	-0.96	0.27	0.00
harfbuzz native	2.48	2.13	1.17	1.89	2.22

ebgaramond	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.43	0.05	-1.76	0.20	0.03
context node	2.19	1.82	0.00	1.00	1.00
context none	0.38	0.00	-1.82	0.17	0.00
harfbuzz native	2.22	1.85	0.03	1.01	1.02

lucidaot	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.42	0.01	-0.26	0.62	0.03
context node	0.67	0.27	0.00	1.00	1.00
context none	0.41	0.00	-0.27	0.61	0.00
harfbuzz native	2.28	1.87	1.61	3.39	7.06

luajittex Arabic:

arabtype	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.36	0.01	-7.52	0.05	0.00
context node	7.88	7.53	0.00	1.00	1.00
context none	0.35	0.00	-7.53	0.04	0.00
harfbuzz native	2.15	1.80	-5.73	0.27	0.24

luajittex mixed:

arabtype	t	$t-t_\mathrm{n}$	$t-t_\mathrm{d}$	t/t_d	$\frac{t-t_\mathrm{n}}{t_\mathrm{d}-t_\mathrm{n}}$
context base	0.61	0.03	-3.79	0.14	0.01
context node	4.40	3.83	0.00	1.00	1.00
context none	0.58	0.00	-3.83	0.13	0.00
harfbuzz native	2.54	1.97	-1.86	0.58	0.51

A few additional notes. Since a library is an abstraction, we need to make the best of it. In my case, I experienced a crash in `utf-32` mode. I could get around it but one advantage of using Lua is that it hardly crashes, (e.g., because as a scripting language it manages its memory well without user interference). My policy with libraries is just to wait till things get fixed and not to bother with the internals whys and hows.

Although ConTEXt will support the `plug` model, I won't officially use it (not even in documentation) so I can't support users in that. I didn't test the `plug` mode in real documents. Most documents I process contain Latin fonts or a mix; redefining feature sets or adapting styles for testing makes no sense. Now the question is: "can we just switch engine without looking at the way a font is defined?", the answer being: "not really, because (even with users not knowing about it) virtual fonts might be used or additional features be kicked in, so other mechanisms can make assumptions about how fonts are dealt with."

The usability of `plug` mode probably depends on the available workflow. We use ConTEXt in a few very specific workflows where interestingly we only use a small subset of its functionalities. Most of those workflows are user driven and tweaking fonts is popular and has resulted in all kind of mechanisms. Therefore it's unlikely that we will ever use it. If you process (in bursts) many documents in succession, each demanding a few runs, you don't want to sacrifice speed.

Of course timing can (and likely will) be different for plain TEX and LATEX usage. It depends on how mechanisms are hooked into the callbacks, what extra work is done or not done compared to ConTEXt. This means that my timings for ConTEXt will differ for sure from those of other packages. Timings are a snapshot anyway. And font processing is just one of the tasks to be executed. If you are not using ConTEXt you will probably use Kai's version because it is adapted to his use case and well tested.

A fundamental difference in the approach is that where the Lua variant operates on node lists only, the plug variant generates strings that get passed to a library (in the ConTEXt variant of `hb` support we use `utf-32` strings). It is interesting that a couple of years ago I considered using a similar method for Lua but eventually decided not to use it, first of all for performance reasons, but mostly because one still has to use some linked list model. I might pick up that idea as a variant but, since all this TEX related development doesn't really pay off and costs a lot of free time, it will probably never happen.

Using this mechanism (there might be variants in the future) allows the user to cook up special solutions. After all, that is what LuaTEX is about: the traditional core engine but with the ability to plug in your own code using Lua and this is just an example of it.

I'm not yet sure when the plugin mechanism will be in the ConTEXt distribution but it might happen once the `ffi` library is supported in LuaTEX.

At the end of this document we show the basics of the test setup, just in case you wonder what the numbers apply to.

Just to put things in perspective: the current (February 2017) MetaFun manual has 424 pages. It takes LuaTeX 18.3 seconds and LuajitTeX 14.4 seconds on my Dell 7600 laptop with 3840QM mobile i7 processor. It takes 6.1 (4.5) seconds of the total time to process 2170 ∓ graphics. Loading the 15 used fonts takes 0.25 (0.3) seconds including loading the outline of some. Font handling is part of the so-called hlist processing and takes around 1 (0.5) second and attribute backend processing takes 0.7 (0.3) seconds. One problem in these timings is that font processing often runs too fast to elapse it, especially when we have lots of small snippets. For example, short runs like titles and such simple texts take no time and verbatim needs no font processing. The difference in runtime between LuaTeX and LuajitTeX is significant so we can safely assume that we spend some more time on fonts than reported. Even if we add a few seconds, in this rather complete document, the time spent on fonts is still not that impressive, but a 5 times slower processing (we use mostly Pagella and Dejavu) would definitely add significantly to the total run time, especially if you need a few runs to get cross referencing etc. right.

4 Conclusion

Nonetheless, at this stage the LuaTeX `ffi` module does not offer the same performance, robustness and number of functions of the corresponding LuajitTeX module because it's still at an early stage of development.

Using native C types as a replacement of Lua types is not yet competitive as in LuajitTeX, and even in the latter environment the `jit` must be carefully managed to avoid to slow down the rest of the run.

The first results of using the direct binding to native shared libraries are encouraging and this can be useful in high specialised workflows. Nonetheless, interfacing with shared libraries also puts serious limits to the code portability and durability (i.e., the property of being easily processed by different releases of the same program) of the code.

Moreover, the main advantage over the corresponding SwigLib approach (see http://swiglib.foundry.supelec.fr) is to avoid the intermediate compilation step of the interface layer, saving another shared library and hence reducing the dependencies, though at the price of reducing the range of the supported architectures/operating systems and using a less sophisticated C parser than that implemented by SwigLib.

▷ Hans Hagen
Pragma ADE
The Netherlands
`http://luatex.org`

▷ Luigi Scarso
Padova, Italy
`luigi.scarso@gmail.com`
`http://luatex.org`