advanced

Optimizing TEX code

some words on speed and space

Hans Hagen
PRAGMA ADE
Ridderstraat 27
8061GH Hasselt NL
pragma@wxs.nl

abstract

Macros can be collected in macro packages. These packages can be stored in a form that permits fast loading. Although TEX is already pretty fast, for demanding applications it makes sense to speed up TEX to the max. Switching to e-TEX and beyond is one way to achieve this, another way can be found in optimizing the macro code by means of a dedicated program. Currently the combination of both can speed up TEX runs by at least 10%.

keywords

speeding up, formats, optimizing code, e-TEX, ConTEXt

Wat is TEX?

TEX is a typographic computer language with a strange character. To mention one: typographical programs and text to be typeset can be mixed. Therefore the border between programs and text is not always that clear. Take

```
1 2 3 \hbox spread 1em{\hss4\hss} 5 6 7
```

which shows up as

1 2 3 4 5 6 7

This piece of code does not really contain a program, but the next example does:

```
\setbox0=\hbox{12}\dimen0=\wd0

1 2 3 \hbox to \dimen0{\hss4\hss} 5 6 7
```

As long as one digit is used, the result is the same as in the previous example, because a digit has a width of .5em. It makes sense to isolate textual input and programming code, like in:

```
\def\TwoDigitsWide#1%
  {\hbox to 1em{\hss#1\hss}}

1 2 3 \TwoDigitsWide{4} 5 6 7
```

The `\def`'d things are called macros. Quite often macros are collected into so called macro packages. That way we save ourselves the time of retyping commonly used macros and at the same time force consistency.

So, TEX is a programming language and a program written in such a language should either be compiled into some low level machine code or interpreted at runtime. Again, TEX is a strange breed, because it does a bit of both: TEX the programming language is processed by TEX the program. While read in from a file, characters and sequences of characters are translated into an internal representation and when not directly typeset, they are stored in appropriate data structures.

In our example, the definition (`\def\Two...`) is stored for later use, and called upon while `1 2 ..` is typeset.

Running TEX

On many computer systems users invoke their prefered macro package by typing in its name at the command line and often they think that `tex`, `latex` or `context` is simply a program. This situation becomes even more vague when one encounters `pdftex` and `pdflatex`. (Be suspicious if you ever encounter `pdfcontext`, because by nature CONTEXT is not aware of any specific brand of TEX, except E-TEX!)

What actually happens is that TEX the program is launched and instantly starts reading in the macro package one asked for. So, it's still TEX that one's running, no matter in what way it is invoked! On many systems saying:

```
tex      <filename>
latex    <filename>
context  <filename>
```

is just the same as:

```
tex &plain   <filename>
tex &latex   <filename>
tex &context <filename>
```

which is the 'official' way of calling TEX with some kind of macro package.

Formats

The macros normally are not stored in ASCII format, but in a so called memory dump, a sort of precompiled format. One can generate such a format by saying:

```
tex --ini plain.tex \dump
tex --ini latex.ltx
tex --ini context.tex
```

When run in ini (also called virgin) mode, TEX reads in the macro package and when finished, it dumps its memory on disk. Such a dump normally has the suffix `fmt` and can be reloaded fast: `&plain` refers to the format. It does not hurt to know this! Try it, if only to be able to generate an updated format.

So, to summarize, we have TEX the programming language, TEX the program, and all kinds of macro packages, that can be packed into memory dumps for fast loading.

When generating an PDFTEX format, or E-TEX or PDF-E-TEX format, a slightly other method applies: PDFTEX has it own configuration information, while E-TEX only has additional functionality when the filename is prefixed by a `*`. To ease the life of CONTEXT users, there is TEXEXEC. This PERL script wraps all brands of TEX in a shell, so there we say:

```
texexec --make en
```

to generate a format with an english user interface. Running CONTEXT is done by saying something like:

```
texexec myfile
texexec --pdf myfile
```

So, when you're using CONTEXT, don't worry if all this formatting talk is beyond you: TEXEXEC is there to guide you. So far for this interlude.

Tokens

TEX thinks in terms of tokens. Let's for this moment forget about the sometimes pretty confusing way TEX reads and interprets the characters, and consider the next example:

```
\expandafter\def\csname name\endcsname{...}
 1          2    3   4 5 6 7    8      9 11 13
```

Once read in memory, this example is represented in 13 tokens. The names of macros as well as the `{`, `}` and .'s each become one internal quantity. The space after `\csname` ends the control sequence and does not count. I will not go into details on the amount of memory each token takes, but in general, one can say that TEX tries to minimize the amount of memory used. A macro name is stored in the so called hash table (a lookup table) and after that only references to this table are used. In the TEX that I'm currently running, a reference to a macro name takes 8 bytes: 4 bytes for the index, a pointer to the next token, some space for an additional index (used by `\charsubdef`) and a rounding byte for efficient internal representation.

The main point of this short story is that once read in, TEX does not need to translate macro names, but simply uses pointers to reach the meaning of this macro. This is not only faster, it also takes less memory. And this is why it makes sense to use memory dumps instead of reading in a macro package each time. TEX loads faster and runs faster.

One should be aware of the fact that we're only talking of a clever way of storing data. Opposite to for instance high level programming languages like Pascal and C, no compilation is done. The meaning of the macro is stored as it is, even if its meaning is wrong in some way or another. Nearly nothing is checked and nothing is optimized. And this is one of the reasons why TEX, while being an interpreter, is so fast.

The main reason why I could give CONTEXT a multi lingual interface without having to recode those tens of thousands lines of code, is that I used a rather high level of abstraction. Keywords and values are stored as macros. This is a sort of lucky coincidence: when CONTEXT grew, TEX's were still pretty small, so I ran out of string memory (the total length of all strings used) before I ran out of hash memory (the number of macros). So, I was forced to use macros, which after all is not that bad, because it also forced and garantees consistent use of keywords.

```
\def\NameKey{name}
```

So, `name` is stored once, and can be accessed by `\NameKey`, not only many times without taking string memory, but also pretty fast. This is only true in CONTEXT deepest inners, because at the user level, one does not type in macroded keys, but verbose ones: `name=hans`.

```
1: \def\Name{Hans}
    1    2   3 4 5 6 7 8

2: \def\SurName{Hagen}
    1    2    3 4 5 6 7 8 9

3: \def\FullName{Hans Hagen}
    1    2    3 4 5 6 7 8 9 11 13
```

This is one way of storing names, but when one wants to save space, the next alternative is more efficient.

```
4: \def\FullName{\Name\space\SurName}
    1    2    3   4     5      6    7
```

Counting tokens is not the way to determine this, because several types of memory are involved: the hash table, the string pool with string pointers, main memory, etc. Therefore we only really save space when more than one refer-

ence is made.

Coding keywords in macros can also be faster, especially when passing large arguments, skipping branches in conditionals and while doing certain low level lookups.

Optimizing code

One can sqeeze quite some speed from clever coding macros and using memory as efficient as possible, but when one hits the frontiers of coding itself, other methods are needed.

I started experimenting with optimization when rewriting part of the system modules. I found out that in one occasion grouping speeds up, while in another rather similar situation doing the housekeeping myself was to be prefered. Potential slow–downers are: passing long arguments, all kind of tests, especially string comparisons, list processing, and rather massive catcode changes.

Original TEX is frozen. Named E-TEX, its successor offers some basic programming features that are meant to speed up TEX as well as provide more control to macro programmers. Being rather curious, I decided to change some low level code and look to what extend E-TEX would speed up a large package like CONTEXT. (Notice that speed was not the primary target of the E-TEX project.)

Although currently most of the critical parts of CONTEXT are rather well optimized —I'm still documenting, optimizing and sometimes recoding the source— I consider the measurements to be be rather representative. It is, by the way, always an interesting dilemma: do I code for speed or for readability. It's one of the reasons why font handling routines often look rather obscure: a pretty complex font mechanism demands dirty coding to get acceptable speed.

When testing for speed, it can be tempting to put some critical code in a loop, and execute this code for instance 50.000 times. Such experiments demonstrate that individual pieces of code can be rewritten in new E-TEX primitives to run much faster. Bringing down runtime from 10 seconds to 5 seconds is fine, but in practice those fragments are not executed that many times, so the gain in normal production runs is minimal. When the protection mechanisms that CONTEXT uses for savely testing keywords are rewritten in a for E-TEX more natural way, there is even a speed penalty!

I therefore tend to conclude that it does not make much sense to recode a large macro package in E-TEX (version 2) for the sake of speed alone. This is mainly due to the fact that the critical components of CONTEXT are already coded rather efficient. Later on I will show that in one area, E-TEX beats TEX pretty well. Keep in mind that we are discussing speed and space. Much of E-TEX's new functionality goes beyond that and concerns better typography. This gives us another reason to use E-TEX.

An interesting observation is that using the new primitives `\protected` and `\ifcsname` saves about 500 hash entries in the current version of CONTEXT. However, making the specific pieces of macro code usefull for normal TEX and E-TEX, costs about 500 entries in normal TEX. Alas, that's the price original TEX users have to pay for progress.

Being parameter driven, CONTEXT does a lot of string and list processing and unfortunately TEX lacks low level support for this. When I discussed this with Taco Hoekwater, we came to the conclusion that some more straightforward support for string and list handling could speed up CONTEXT considerably, and Taco decided to extend TEX the program. That way we can present the E-TEX team with well defined and tested functionality for future versions.

When testing some first versions of Taco's binaries, I wondered if it would make sense to optimize TEX macro code in another way. To understand what I mean, I refer to a few pages back, where I introduced those hash entries and pointers. In writing macros, I try to be as clear as possible, so instead of

```
\csnamehello\endcsname
    1   23456      7
```

I code

```
\getvalue{hello}
    1    2345678
```

and not

```
\doifelse\Alpha{Beta}
    1        2   345678
```

but, at the cost of 16 bytes overhead and a two more 'lookups':

```
\doifelse{\Alpha}{Beta}
    1     2    3   456789
```

Did you notice the difference in the number of tokens used? Using a syntax highlight editor,

```
\dimen0=0pt
   1   23456
```

just looks better and more readable than

```
\dimena\z@
   1    2
```

Currently, and this is of course due to the fact that we are dealing with macros, we are also more talking of translating than of compiling. Some first experiments with an optimizer written in PERL were promising, but fearing unwanted side effects I decided to let this rest for a while.

The optimizer

And then Han The Thanh asked me to test his first version of PDF-E-TEX. This merge of two rather important developments in the TEX world: E-TEX and PDFTEX, drove me into making CONTEXT more permanently E-TEX aware as it was already pretty PDFTEX aware. While testing, I also picked up the optimization thread.

Using the timing build into TEXEXEC I found out that on a document of average complexity, the interactive MAPS bibliography, with 700 simple pages and 50 pages of cross linked indexes and lists, rewriting some core macros to use E-TEX functionality saves about 5% run time and the optimizer gives us an additional 5%: not impressive, but useful when one considers that I quite often run jobs that take an hour or more. And, apart from more efficient coding, I expect to gain another 10% in due time.

Before I will mention some characteristics of the CONTEXT optimizer, I need to give E-TEX some more credit. When I first timed the test run, I found out that E-TEX took 65% of the time the normal run needed. This proved to be due to the fact that outside E-TEX one has to fake multiple marks, and this fake can slow down a run with many color changes on a page considerably (due to lots of list manipulations). When using normal TEX, CONTEXT spends half of the run time on the 50 pages mentioned before. Because marks are used for keeping track of color, and because these pages have colored hyperlinks in multiple columns, a speed penalty is paid. Unless one heavily uses color in rather complicated documents, one will probably never notice.

When color does not cross pages, which did not really happen in this document, CONTEXT can be told to switch to local color mode. In local mode, E-TEX's gain in speed is reduced to the mentioned 5%.

Some details

Back to the optimization. Apart from a few critical files, 100 modules that are part of the distribution are optimized in four converging passes (doing it in one pass is much slower). In the process, over 7.000 lines out of 80.000 lines of macro code are optimized, in many occasions, more optimizations per line. All changes are logged and some statistics are kept. Dubious and potential dangerous situations are skipped. Of course, E-TEX optimizations are optional. Don't confuse this optimization with rewriting core macros.

Some lines are skipped. Think of macro headings and calls that use delimiters like `\box`. Optimizing macro headings is 'not done' anyway, but the next substitutions are quite legal:

- removing redundant equal signs
- using `\empty` instead of `{}`
- using `\z@` instead of 0pt
- changing 2, 4, 6 and 8 into constants in box primitives
- changing 2, 4, 6 and 8 registers into predefined ones
- using macro constants where possible
- substituting TEX keywords by macros
- changing `\getvalue` cum suis into `\csname`
- optimizing all kind of `\doif...` macros
- applying E-TEX's `\ifcsname` and `\ifdefined`
- removing redundant `{}` in arguments
- removing redundant TEX keywords

As said, sometimes substituting can be dangerous. Changing for instance

```
\expandafter\ifx\csname...\endcsname\relax
```

into E-TEX's sequence:

```
\unless\ifcsname...\endcsname
```

can lead to unwanted side effects. The pure TEX alternative creates a hash entry, that defaults to `\relax`, which is why we test for `\relax`. We just consider `\relax` to represent undefined. The second one does not create an entry. Consider for instance that at a certain moment, for instance in the process of font switching,

```
\csname...\endcsname
```

is expanded. This leads to ... being set to `\relax`. Now, when in pure TEX, we test for existance, as expected we get reported back that the control sequence does not exist. In E-TEX however, `\ifcsname` is unrelated to `\relax`, and therefore, E-TEX reports that the control sequence does exist. Even if you don't completely understand what I'm talking about, you can imagine that macros that until now work perfectly ok, fail under E-TEX.

The size of the format file (the memory dump) when optimized is currently about 75 KB less than the non–optimized file (about 2.8 MB) and some quick tests show that another similar saving is possible. It's not that much a problem to save far more bytes, but sometimes speed goes over space:

1: `\setbox0=\hbox{something}`
   1 2 3 4 5 6 7 8 9 11 13 15

2: `\setbox0\hbox{something}`
   1 2 3 4 5 6 7 8 9 11 13

3: `\setbox\boxa\hbox{something}`
   1 2 3 4 5 6 7 8 9 11 13

4: `\setboxa\hbox{something}`
   1 2 3 4 5 6 7 8 9 11 13

5: `\sethboxboxa{something}`
   1 2 3 4 5 6 7 8 9 11

The third alternative, with \boxa being \chardef'd to zero is the fastest. The last two alternatives save memory (8 bytes per macro name). It does not take much fantasy to see that the third alternative involves the two tokens \setbox and \boxa, while the fourth one takes only one (\setboxa). When resolved, this indirect reference itself takes two, therefore totalling up to three.

```
\def\setboxa{\setbox\boxa}
```

Some gain in speed in E-TEX is due to optimizing existing TEX code. I did not compare the results to original TEX, but especially saving and restoring and in some situations more clever \aftergroup handling has proved to be a welcome extension.

Conclusion

What can we conclude. First that 15 years of TEX has proven that using memory dumps makes sense. Next that for gaining some speed, developments like E-TEX make sense too. For me however the most interesting conclusion is that some sort of preprocessing makes much sense too. Because this optimization is in many respects dependant of the way the macro package is written, it is not possible to bring this into E-TEX the program. Anyhow, given this 10% speed gain, makes me very optimistic about Taco's estimations on string and list processing. It can also be a step towards a higher level of typographic programming in TEX, where more readable definitions are compiled into their lower level TEX equivalents.

Literature

- The E-TEX manual, version 2, February 1998, Peter Breitenlohner.
- E-TEX, a 100% compatible successor to TEX, Philip Taylor, EuroTEX 1995 proceedings.
- Examples: especially the CONTEXT modules syst-gen, supp-mrk, font-ini.
- Unpublished e-mails, Taco Hoekwater.