Profiling TEX input files

Martin Ruckert

Abstract

A profiler is a tool used by programmers to analyze the runtime behavior of the code they write. The profiler can map the CPU time of a program to specific files and lines, or it can map the time to individual procedures. This information is necessary if a programmer wants to optimize the code for speed.

No such tool has been available to date to programmers who write macro packages for TEX. This paper presents `texprof` and `texprofile`, two programs working together to profile TEX input files.

1 Who needs a profiler?

The TEX profiler is a tool for programmers writing TEX macros. This does not mean that an author who occasionally writes a TEX macro should use or even needs to use this tool. Optimizing a macro for speed should be done only if the macro is used *very* often. To get a feeling for what "*very* often" means, consider the following: Under reasonable assumptions (200 watt peak power consumption of your PC and 500g of CO_2 emission per kWh electric power use) one second of CPU time results in 28mg of CO_2 emission. Again under reasonable assumptions (2370g of CO_2 emission per liter fuel and 6l fuel consumption per km) driving 200km results in 28kg of CO_2 emission. That means that you need to save millions of seconds in CPU time before it has any substantial impact on your CO_2 footprint or your budget.

So for the occasional macro writer there are better opportunities to invest time and intelligence than optimizing macros for speed. But of course there are macro packages for TEX that have millions of users that use these macros in multiple runs every day, and if you are the programmer of such a package, you might be interested to know if there are opportunities for optimization, where these opportunities are hiding in your code, and how much you might gain when optimizing this code. Maybe even more important, a profiler can tell you where not to look for optimizations, and — after the optimization — if the changes to your code had the desired effect. As a general rule, you should never optimize code for speed without using a profiler.

2 How does the TEX profiler work?

2.1 Mapping commands to files and lines

Every TEX engine is an interpreter that executes the built-in commands of TEX, like creating a horizontal box, incrementing a count register, or adding a character to the current paragraph. The TEX profiler, called `texprof`, is such an engine with extensions to map every command to a file and a line in that file. If `texprof` reads such a command from an input file, it can determine the file name and the line number from the data structures that every TEX engine maintains to display good error messages. If on the other hand, such a command was part of a format file, the file name and the line number is not known. We will see below how using a format file can be avoided.

But even if we avoid using a format file, many commands are not read directly from an input file, instead they come from expanding a macro. When a macro is defined, TEX stores the commands that belong to the macro body in a hash table, and when the macro is used, TEX retrieves the commands from the hash table and inserts them into the input. Since the file and line are known when a macro is defined, this information can be stored in the hash table and retrieved along with the commands when a macro is expanded. The same mechanism comes into play when TEX reads ahead, for example when scanning a keyword, and discovers that the commands seen must be pushed back into the input for later processing: the commands take their file and line numbers with them.

There are a few rare cases where this mechanism does not work. For example TEX inserts a pair of curly braces around an output routine to make sure that commands executed in an output routine do not have unexpected global effects. These extra commands are marked as coming from the "system" file in line zero.

There are other "line numbers" in the "system" pseudo-file that the profiler will use. TEX invokes system procedures like breaking a paragraph into lines or writing a page to the DVI file that can be quite time consuming. It would be misleading if the time spent in these routines would be mapped to whatever command happened to be executed at the end of a paragraph or caused the page builder to eject a page. So `texprof` associates these times with the "system" pseudo-file and uses the line number to indicate the responsible procedure in TEX.

2.2 Mapping execution times to commands

Most of TEX's commands, but not all, are executed in TEX's `main_control` procedure. There we find a loop that reads a command and then executes the command by branching to its code to execute it using the so-named `big_switch` (the label in the Pascal code). `texprof` looks up the current time at

doi.org/10.47397/tb/45-2/tb140ruckert-profiler

the start of each iteration of this loop. The time is taken from a hardware clock using a low-level routine provided by the operating system. The time taken from the clock is the start time of a new time interval and simultaneously the end time of the previous time interval. After reading the start time, `texprof` continues with the normal processing of TEX and reads the next command from its input stack. Once the command is known, `texprof` takes note of the command, its file, its line (and the macro it comes from — but let's focus for the moment on commands; macros will be explained later). Then normal processing continues and the command is executed using the `big_switch` which ends with a jump to the beginning of the loop. There `texprof` will again look up the time, compute the time difference and record command, file, line, and time difference in a large array.

Occasionally, a command is "reswitched". That means, it is replaced in the `big_switch` by another command and the `big_switch` is used again to execute it. `texprof` ignores this replacement and will record the entire time together with the command, file, and line that was obtained at the beginning of the current iteration. This introduces some imprecision into the measurements but does not cause significant errors.

A much bigger effect on the association of time to file and line is caused by the existence of TEX's `main_loop`. The `big_switch` will jump to this loop whenever it encounters a character and it will stay in this loop as long as the commands are the typical commands found in plain text: characters, spaces, kerns, ligatures, font changes, and a few more such things. The complete time spent in this loop is then recorded with the command, file, and line that started the loop. So the time used to process an entire paragraph might be associated with the first letter of that paragraph. The decision to forgo a more precise attribution of time in this case is justified by the following considerations: First, such a paragraph is normally processed only once and the time it takes is usually not a significant fraction of the total run time. Second, using a profiler, we are usually not interested in the time spent on letters, spaces and other parts of plain text. After all, no author optimizes the text for the speed of processing it. And finally, this reduces considerably the number of time intervals that `texprof` needs to record.

TEX's `main_control` procedure exits when executing the "stop" command. This ends the recording of time intervals. As part of TEX's closing procedure, `texprof` will write all the data collected to a binary file. The name of this file is obtained by appending `.tprof` to the `\jobname`. All further processing of the collected data is not done by `texprof` but by a second program called `texprofile`. Its use will be illustrated below.

2.3 The problem of measuring time

Operating systems usually provide several clocks to choose from. `texprof` uses the common function `clock_gettime` to measure the CPU time of the current thread in nanoseconds. A measurement of nanoseconds seems like very precise information, but the actual precision is more on the order of a hundred microseconds, for several reasons. First, modern computers run many different processes at the same time and switch the available CPUs from one process to another. The time to switch processes includes of course the time needed to swap out and swap in the contents of memory caches. So when a process was recently swapped in, a load instruction from a location in main memory might take considerately longer if that memory location is no longer in the cache. This extra time is then associated with the TEX command that happens to be executing.

Second, modern CPUs use a technique called frequency scaling in order to reduce energy consumption. While you just type text in the editor, you laptop might run with a frequency below 1 GHz; shortly after you have started TEX, your operating system might notice that there is a lot of work to do and increase its clock frequency to 4 GHz. All commands that are executed after this change will need less time than the same commands before this change.

Third, in recent years manufacturers have begun to combine on one chip a few performance cores with a few efficiency cores. The former use sophisticated superscalar pipelines to execute multiple instructions per clock cycle; the latter use simple in-order execution which requires far less power and produces far less heat but might need multiple clock cycles per instruction. So if your operating system decides to move `texprof` from an efficiency core to a performance core in the middle of running, this has a drastic effect on the command times. The operating system might move it even back to the efficiency core if `texprof` starts to do file input/output which causes it to wait for the disk.

There are a few possibilities to mitigate these effects like computing the average over multiple runs or using synthetic times, but none of them is currently implemented.

2.4 Mapping computing times to macros

When TEX encounters an active character or a control sequence, it knows it has to execute a macro. It looks up the list of commands that form the body of the macro and pushes this list on its input stack. When TEX needs the next command from the input, it takes it from the topmost list on the input stack. And when TEX reaches the end of the topmost list, it pops it from the stack and continues to read commands from where it was before in the next lower list on the stack. The input stack is used not only for macro calls, but also for implicit calls to other routines. The page builder will, for example, push the output routine on the input stack when a new page is ready. Or at the beginning of a paragraph, the system will push the commands specified with `\everypar` on the stack. Entire files are pushed on the stack when you use `\input`. It is also very common that TEX reads ahead, for example to check for a possible keyword, and pushes unused tokens back on the input stack to be read again if the keyword was not found.

Most of the information `texprof` needs to keep track of macros can be found on TEX's input stack — but not all of it. Notably, the input stack will not contain information about the correct nesting level of the macros. The reason for this is a clever optimization that TEX uses on its input stack called last call optimization: Before a new macro body is pushed on the input stack, TEX checks repeatedly if the topmost list on the input stack is already empty. And if so, it will remove the empty list from the stack. Only after all empty lists have been removed the body of the new macro is pushed. Using this optimization, a loop that is implemented by a recursive macro, calling itself as the last action of the macro body, can run without overflowing the stack. Unfortunately this technique will remove a macro from the input stack while its last sub-macro is still running; and it will push new macros at a lower nesting level than their "true" nesting level.

So `texprof` adds information about the "true" current macro nesting level to TEX's input stack and maintains a separate stack that contains information about all macros up to the true nesting level: the name as well as the file and line number of the macros definition. This stack provides information about the true begin and end of a macro call which is recorded together with the timing information of executed commands.

3 Analyzing profiling data

The raw data that `texprof` writes to the output file is just a long list of thousands of "command file line time" records interspersed with records that reflect the changing of the macro stack. Extracting useful information from this data is the job of the `texprofile` program.

To explain the use of `texprof` and `texprofile`, it is best to use examples. For the first example, I was looking for a large document with a focus on text. Searching the internet for such an example, I found a TEX version of the bible (`github.com/vermiculus/bible`). With a few changes I made it use the "plain" TEX format so that it makes only a limited use of macros. Most macros are taken from plain TEX, but there are also some user defined macros. Running `texprof -prof bible` will create `bible.tprof` with a size of 17 Mbyte. Running `tprof bible` without further command line options will print the following summary:

Total time measured:	728.92 ms
Total number of samples:	2157642
Average time per sample:	337.00 ns
Total number of files:	69
Total number of macros:	1097
Maximum stack nesting depth:	7

You can use command line options to specify which data tables `texprofile` should display and how it should display the information.

3.1 The top ten lines

Given the `-T` option, for example, `texprofile` will traverse the data, add up the times for each file and line combination separately, then sort the results, and display the ten lines with the highest cumulative times: the "Top Ten" lines. The output is shown in Fig. 1.

The first line in the table is attributed to the "system" pseudo-file. The entry shows the accumulated time for an important system routine using the line number to identify the specific routine like producing the output DVI file (shipout), or a bit further down breaking a paragraph into lines (linebrk), or breaking the document into pages (buildpg). These times do not depend on the use of macros but simply on the size of the document.

From the next line, we can see that line 29 of `bible.tex` is responsible for 17.63% of the total run time and therefore is a good candidate for optimization, which we will try to do in the next section. The line by itself is quite fast, on average only 2.85 μs are spent on this line, but the line is used very often: 54649 times.

The fact that the remaining six lines all contibute less than 1% to the overall runtime means that we need not consider any of them for optimization.

file	line	percent	absolute	count	average	file
system	shipout	17.68%	156.05 ms	1130	138.09 μs	system
5	29	17.63%	155.65 ms	54649	2.85 μs	bible.tex
system	linebrk	15.21%	134.26 ms	25777	5.21 μs	system
system	buildpg	1.69%	14.89 ms	55190	269.00 ns	system
5	56	0.86%	7.61 ms	4750	1.60 μs	bible.tex
5	15	0.62%	5.43 ms	6183	878.00 ns	bible.tex
3	555	0.47%	4.17 ms	8549	487.00 ns	plain.tex
3	1204	0.28%	2.44 ms	3390	719.00 ns	plain.tex
3	1201	0.26%	2.33 ms	2260	1.03 μs	plain.tex
3	1203	0.25%	2.20 ms	2258	973.00 ns	plain.tex

Figure 1: Running `tprof -T bible`

file	line	percent	absolute	count	average	file
system	shipout	18.35%	156.29 ms	1130	138.31 μs	system
system	linebrk	15.64%	133.23 ms	25777	5.17 μs	system
5	29	12.85%	109.48 ms	60839	1.80 μs	bible-opt.tex
3	666	1.95%	16.61 ms	55847	297.00 ns	plain.tex
system	buildpg	1.74%	14.85 ms	55190	269.00 ns	system
5	55	0.78%	6.67 ms	3552	1.88 μs	bible-opt.tex
5	15	0.63%	5.39 ms	6183	871.00 ns	bible-opt.tex
3	555	0.49%	4.18 ms	8549	489.00 ns	plain.tex
3	1204	0.29%	2.43 ms	3390	716.00 ns	plain.tex
3	1201	0.27%	2.29 ms	2260	1.01 μs	plain.tex

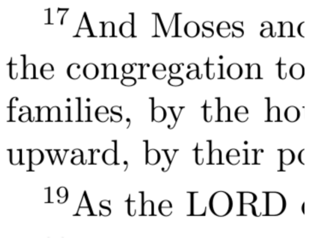
Figure 2: Running `tprof -T bible-opt`

3.2 Optimizing a macro

Line 29 of `bible.tex` defines the `\Verse` macro (formatted on two lines here for *TUGboat*):

```
\def\Verse{\global\advance\vcount
            by 1${}^{\the\vcount}$}
```

It is used to add the number of each verse, in small print and raised a bit above the baseline, to the beginning of every verse, like this:



^{17}And Moses an
the congregation to
families, by the ho
upward, by their p
^{19}As the LORD

We optimize this macro for speed: The `\global` prefix is not needed because the macro is used only on the global level. `by` is an optional keyword and can be left out. Any literal constant like “`1`” is stored in the macro body as a sequence of characters which is rescanned and converted to an integer every time the macro is called. It is more efficient to use one of TeX's registers instead. Last but not least, using math mode just to raise a number and use a small font is a lot of processing for a simple effect. Here is the optimized version:

```
\newcount\1 \1=1  \newdimen\3 \3=3.6pt
\def\Verse{\advance\vcount\1
           \leavevmode\raise\3
             \hbox{\sevenrm\the\vcount}}
```

It uses two registers for the necessary constants and requires a call to `\leavevmode` because `\raise` is not allowed in vertical mode.

The top ten lines after optimization are shown in Fig. 2. Line 29 of `bible.tex` dropped from second place with 17.63% down to third place with only 12.85%. But this is not the full story: New is line 666 of `plain.tex` on fourth place with a 1.95%. So we get an overall speed-up of almost 3% from 17.63% down to 14.80%.

If we want to know what caused the increased use of plain TeX, looking at the call graph can shed some light on it.

3.3 The call graph

The call graph gives us information on a higher level of abstraction than what we gain from looking at the top ten lines. Consider if a different layout distributed the macro in line 29 that we have considered over 10 lines. We would have had 10 entries with about 1.8% each and none of them would have

```
          time  loop  percent   count/total   macro
\Verse
     174.51 ms         24.64%        *        \Verse
     101.50 ms         58.16%      31011      \Verse
      73.01 ms         41.84%   31011/31011   \leavevmode

\output
     121.22 ms         17.11%        *        \output
       1.15 ms          0.95%      1130       \output
     120.07 ms         99.05%    1130/1130    \plainoutput

\plainoutput
     120.07 ms         16.95%        *        \plainoutput
     106.71 ms         88.87%      1130       \plainoutput
       6.99 ms          5.82%    1130/1130    \makeheadline
       2.76 ms          2.30%    1129/1130    \pagebody
       2.75 ms          2.29%    1130/1130    \makefootline
     859.36 μs          0.72%    1130/1130    \advancepageno

\leavevmode
      73.01 ms         10.31%        *        \leavevmode
      20.09 ms         27.52%      31011      \leavevmode
      52.92 ms         72.48%     495/1130    \output
```

Figure 3: Running `tprof -G bible-opt`

made it to the top of the list. The time recorded for a macro, on the other hand, does not depend on the layout of your source files. A macro gives a sequence of commands a common name, typically expressing its purpose, or the task that it will accomplish. To accomplish a task, a macro usually calls other macros, that we call child macros in the following. Such a child macro in turn might call again its own child macros. Along the chain of macro calls, a macro might eventually even call itself creating a recursive loop (as we will see below) where a macro becomes its own ancestor.

So when we look at the runtime of TEX from the macro perspective, we want to know how much time was spent in a certain macro, including all its descendants, because this is the time used to accomplish the task that the macro name promises to accomplish. We call this the cumulative time for the macro. Further we want to know how the cumulative time splits up into the time used by the macro itself and the time used by each of its child macros. This is the information that we gain by looking at the call graph. Fig. 3 shows the four macros that take the greatest percentage of the total run time.

We see the `\Verse` macro on top; 174.51ms are spent executing this macro which is almost a quarter of the total run time. But during the 31011 calls to this macro only 101.50ms or 58.16% of the 174.51ms are spent on the `\Verse` macro itself while the remaining 73.01ms are spent on calls to `\leavevmode`.

Looking at the last group of entries in Fig. 3, we see how the time used for `\leavevmode` is spent: roughly one quarter is spent on `\leavevmode` itself while the remaining three quarters are due to 495 calls to the `\output` routine. The total number of calls to the `\output` routine is 1130, which equals the number of pages of the document.

From the remaining entries, one for `\output` and one for `\plainoutput`, we see that `\output` does little more than call `\plainoutput`, which does most of the work itself, delegating only a small fraction of the work to properly named child macros.

3.4 Emulating pdfTEX

Our second example is `texprof` itself; to be precise: its documentation. `texprof` is an extension of TEX, and since TEX is implemented by a "literate" program, `texprof` is implemented by extending it. This literate program can be processed to obtain `texprof.c` from which a compiler can create an executable. Further it can be processed to obtain `texprof.tex` from which `tex` or `pdftex` can create a nicely typeset document.

Surprisingly creating a PDF with `pdftex` is significantly slower (2327ms) than creating a DVI file with `tex` (273ms). Of course, PDF is a much more complex file format than DVI and this accounts for some of the differences, but even if the creation of the PDF output is disabled (1602ms), there remains a considerable time difference. Just running

file	line	percent	absolute	count	average	file
9	156	20.29%	522.50 ms	225137	2.32 μs	cwebacromac.tex
9	157	12.86%	331.08 ms	140088	2.36 μs	cwebacromac.tex
9	158	9.13%	235.03 ms	140938	1.67 μs	cwebacromac.tex
9	159	5.88%	151.27 ms	115319	1.31 μs	cwebacromac.tex
9	172	3.68%	94.64 ms	15759	6.00 μs	cwebacromac.tex
9	173	3.18%	81.95 ms	36954	2.22 μs	cwebacromac.tex
system	shipout	3.16%	81.27 ms	775	104.87 μs	system
system	linebrk	3.14%	80.93 ms	27370	2.96 μs	system
9	152	2.16%	55.61 ms	67026	829.00 ns	cwebacromac.tex
9	166	1.86%	47.95 ms	13042	3.68 μs	cwebacromac.tex

Figure 4: Running `tprof -T texprof`

```
156 \def\addtokens#1#2{\edef\addtoks{\noexpand#1={\the#1#2}}\addtoks}
157 \def\poptoks#1#2|ENDTOKS|{\let\first=#1\toksD={#1}%
158   \ifcat\noexpand\first0\countB=`#1\else\countB=0\fi\toksA={#2}}
159 \def\maketoks{\expandafter\poptoks\the\toksA|ENDTOKS|%
160   \ifnum\countB>`9 \countB=0 \fi
161   \ifnum\countB<`0
162     \ifnum0=\countC\else\makenote\fi
163     \ifx\first.\let\next=\maketoksdone\else
164        \let\next=\maketoks
165        \addtokens\toksB{\the\toksD}
166        \ifx\first,\addtokens\toksB{\space}\fi
167     \fi
168   \else \addtokens\toksC{\the\toksD}\global\countC=1\let\next=\maketoks
169   \fi
170   \next
171 }
```

Figure 5: Lines 156 to 171 of `cwebacromac.tex`

`texprof -prof texprof` will not suffice to find the source of the slowdown because `texprof` produces a DVI file and runs, when considering the profiling overhead, at approximately the same speed (443ms) as `tex`. The difference in speed is obviously caused by macros that are used only when producing PDF output. So we have to make `texprof` pretend to be `pdftex`. This can be achieved by processing a few macro definitions as shown in Fig. 7 before processing `texprof.tex`.

Using the file `fakepdf.tex` with these definitions, running `texprof -prof -jobname=texprof \input fakepdf.tex \input texprof.tex` will take 1771ms as expected. The top ten lines are shown in Fig. 4; they reveal that almost half of the runtime is caused by only four lines, 156–159, in file `cwebacromac.tex`. These lines, shown in Fig. 5, define macros `\addtoks`, `\poptoks`, and `\maketoks`. For information on the purpose of these lines, we can consult the call graph. Fig. 6 shows the three macros that take the largest percentage of the runtime. Let's consider them one by one.

3.5 Recursive macros

On top is the macro `\pdfnote` followed by the file number 4 and line number 152 in square brackets. This extra information is produced when using the `-i` option of `texprofile` and is needed to distinguish two macros that happen to have the same name, as we will see below. The macro `\pdfnote` creates links to the different sections of the documentation. `\pdfnote` is called 8473 times and each call makes a call to `\maketoks` which is responsible for almost all time needed for `\pdfnote`.

Each call to `\maketoks` in turn delegates most of its work to `\next`. If we look at the file and line information of `\maketoks` and `\next`, we discover that both macros are defined in the same file and on the same line 159. In Fig. 5, we see in line 159 the definition of `\maketoks` and in line 164 a `\let` command that makes `\next` an alias for `\maketoks`. The call to `\next` in line 170 ends the definition of `\maketoks`. So in fact `\maketoks` calls itself recursively. A recursive macro like this where the recursive call is at the very end of the macro is called "tail

```
      time     loop  percent   count/total    macro
\pdfnote [7,152]
    1.30 s            61.17%       *          \pdfnote [7,152]
   26.54 ms            2.04%     8473         \pdfnote [7,152]
    1.21 s            93.24%   8473/9271      \maketoks [7,159]
   46.59 ms            3.58%  24230/28824     \pdflink [7,24]
   14.67 ms            1.13%   4507/4507      \[ [5,334]
   99.89 µs            0.01%     80/80        \ETs [5,177]
   54.34 µs            0.00%     57/57        \ET [5,176]
  404.00 ns            0.00%      1/3         \glob [4,166]

\maketoks [7,159]
    1.24 s            58.35%       *          \maketoks [7,159]
    4.67 ms            0.38%     9271         \maketoks [7,159]
    1.21 s            97.76%  9271/130811     \next [7,159]
   14.59 ms            1.17%  9271/140082     \poptoks [7,157]
    8.51 ms            0.69%  9271/225136     \addtokens [7,156]

\next [7,159]
    1.21 s            57.05%       *          \next [7,159]
   53.94 ms            4.44%    130811        \next [7,159]
  501.23 ms           41.29% 142326/225136    \addtokens [7,156]
  462.00 ms           38.06% 130811/140082    \poptoks [7,157]
  182.12 ms           15.00%  28737/28737     \makenote [7,172]
   12.19 ms            1.00%   9271/9271      \next [7,174]
    2.53 ms            0.21%   1456/2254      \makenote [7,154]
    0.00 ns   1.19 s   0.00% 121540/130811    \next [7,159]
```

Figure 6: Running `tprof -G -i texprof`

recursive" and is optimized by TEX to run without growing the input stack as explained before.

The `\next` macro distributes the work among `\addtokens` and `\poptokens` and some calls to the `\makenote` macro. The 9271 calls of `\next` in the `\maketoks` macro eventually end in 9271 calls of `\next` as defined in line 174 where `\next` is redefined when the final "`.`" is found. Because `\next` calls itself, it is its own child macro and its own parent macro at the same time. This has consequences for the attribution of the cumulative times as shown in the call graph.

The first line in the table for the `\next` macro shows the total time spent in the next macro as 1.21 seconds; the following lines give a breakdown of these 1.21 seconds; the times given in their first column should add up to 1.21 seconds and the percentages given should add up to 100%. If `texprofile` only determined for each child macro the start and the end time and added up the time differences, the values for `\next` as a child macro of itself would come to 1.19 seconds, as shown in the column labeled "loop". But when the `\next` child macro returns, all of that time is already included in the time shown in the previous lines. Therefore `texprofile` maintains for each child two accumulators for the elapsed time: For the time shown in the column labeled "loop", `texprofile` adds up the time differences observed at the return of a child macro. For the time shown in the column labeled "time", it subtracts from the time differences observed at the return of a child macro all those time differences that were added to the macro itself or one of the other child macros since the start of the macro because these differences are already accounted for in the time breakdown. In a simple loop like the one we have here, all the time in `\next` as a child macro are already taken care of in `\next` as the parent macro. So the time column shows 0.00 nanoseconds. For more complex recursive loops this is not always the case.

4 Command line options and primitives

The command line options of `texprof` match those of other TEX engines. The only addition is the `-prof` option to switch profiling on right from the start. To profile only selected parts of a file, you can use the primitives `\profileon` and `\profileoff`.

The command line options of `texprofile` follow the general rule that options that select data tables use upper case letters and options that change the presentation of the data use lower case letters. We have seen already the `-T` option for the "Top Ten" table, the `-G` option for the call graph table, and the `-i` option to annotate (ambiguous) macro names with file and line numbers. `texprofile` can also display the cumulative times by input files with the `-F` option, by input lines with the `-L` option, or by TeX command with the `-C` option. Further the `-R` option displays the raw times for each and every command that was profiled. This table can get very large. It is useful if profiling was switched on for only a short time or if the data is sent to a file for further processing.

If the table data is intended for further processing, the `-m` option favors machine readability over human readability. Whereas by default times are displayed using an appropriate unit, either seconds `s`, milliseconds `ms`, microseconds `us`, or nanoseconds `ns`, the option `-m` will display all times in nanoseconds without specifying a unit.

The option `-p`n will suppress lines in the tables that fall below n percent. The option `-t`n with $1 \leq n \leq 100$ modifies the `-T` option to show the "Top n" lines. The option `-s` modifies the `-R` option to include in the table information about the changes in the macro stack.

5 Improvements, workarounds, and future work

Current versions of `texprof` and `texprofile` are available from `github.com/ruckertm/HINT`. Since the first presentation of the TeX profiler at the TUG 2024 conference in Prague, a few improvements have been made. Most notably, the format files now contain file and line information for all source files used in generating the format. Therefore the attribution of runtime to an unknown file should now be a rare exception.

```
\def\pdftexversion{140}
\def\pdfoutput{1}
\def\pdfdest#1fith{}
\def\pdfendlink{}
\def\pdfannotlink#1goto num#2%
  \Blue#3\Black\pdfendlink{#3}
\def\pdfoutline goto#1 #2 #3{}
\def\pdfcatalog#1{}
```

Figure 7: The file `fakepdf.tex` implementing stubs for pdfTeX primitives

The method shown above to make `texprof` expand macros as `pdftex` would do is a workaround. As can be seen in Fig. 7, the stubs for the pdfTeX primitives merely match the specific uses of these primitives in the given files. Some primitives, e.g. `\pdfannotlink`, have a complicated syntax that is easy to implement for engine primitives but quite complicated to achieve with TeX macros. Here some future work is necessary, either to make the implementation of macros with the desired syntax simple or to add a command line switch to `texprof` to make the necessary stubs available as engine primitives.

◇ Martin Ruckert
Hochschule München
Lothstrasse 64
80336 München
Germany
`martin.ruckert (at) hm dot edu`