A High-Performance TEX for the Motorola 68000 Processor Family

Steve Hampson and Barry Smith
Blue Sky Research
534 SW Third Avenue
Portland, OR 97204
Phone: (800) 622 – 8398
Internet: barry@reed.edu

Abstract

TEX is a large, computationally-intensive program that, thanks to its author, is extremely stable and thoroughly debugged. We describe herein some aspects of our recent work on an assembly language implementation of TEX for the Motorola 68000 processor family. Particular attention is given to memory reference and procedure call templates, performance measurement tools and techniques, and stupid compiler tricks. We also compare the results of our work with those of more conventional methods.

Background

Each of the authors has a good deal of practical experience with Pascal compiler engineering. We were each previously employed in senior technical positions at Oregon Software, a small company specializing in high-performance compilers. (Curiously, our times at Oregon Software did not overlap, and we did not meet until later.)

TEX, of course, is a large computer program written (by D. E. Knuth) in WEB, a structured design language that can be processed to yield a (more or less) standard Pascal program. The TEX program is very stable and thoroughly debugged, thanks to its author and the many users involved in its development.

Motive

Blue Sky Research is the publisher of *Textures*, a TEX-based desktop publishing system for the Apple Macintosh family of computers. The interactive orientation of the Macintosh compels its programmers to be attentive to human-scale performance, and we found the speed of TEX itself to be a limiting factor in making Textures more interactive. (Also, frankly, as compiler writers, we became unhappy whenever we looked at the code produced by the commercially-available compilers.) We, of course, were aware of the traditional disadvantages of assembly code, but thought that the extreme stability of the TEX program made this a special case.

Procedure

We did not propose to recode TEX directly in assembly code, being aware of the programmer's maxim "80% of the time is spent in 20% of the code." (Actual measurements suggest that, for TEX, 95% of the time is taken by less than 3% of the instructions!) We instead proposed (and carried out) a course of action roughly as follows:

(1) Produce a special-purpose Pascal compiler to generate readable basic assembly code;
(2) tune the code generation of the compiler based on extensive measurement and examination of critical code sequences;
(3) when no further compiler improvements appear worthwhile, throw the compiler away and continue with the compiler-generated assembly code as a base;
(4) identify critical routines and segments and rewrite "by hand"; and
(5) repeat step 4 until it no longer appears worthwhile.

(Step 3 was referred to within our team as "making the jump to hyperspace." Steps 4 – 5 are continuing as of this writing.)

Examples

This portion of the paper assumes some familiarity on the part of the reader with the flavor of assembly language and the structure of TEX itself. For readers not familiar with the Motorola 68000 family architecture, it is by and large a 32-bit two-address multiple-word instruction set with 8 general-purpose

arithmetic registers (D0 – D7) and 8 32-bit address base registers (A0 – A7). A7 is conventionally used as the stack pointer (SP).

Example 1: Access to the "mem" array. The `mem` array is TEX's principal memory structure, and is referenced by many parts of the program. Since Textures incorporates a "large" TEX, each element of `mem` occupies 8 bytes; furthermore, since in Textures the `mem` array can grow as needed, it is referenced by a pointer that may change value when the array changes size.

The straightforward 68000 instruction sequence to access item "n" is as follows, assuming "n" is in D0:

```
asl.l  \#3,D0        ; shift N left 3 bits
move.l MEMPTR,A0     ; load address reg
move.l 0(A0,D0:L),D1 ; mem[N].rh -> D1
```

This is a relatively expensive sequence, considering its wide-spread use. Especially costly is the instruction to load the base address of `mem`, requiring 4 bytes for the instruction itself and also a 4-byte memory fetch.

On the 68000, address register A6 is conventionally used as a procedure frame pointer, maintaining the stack offset on procedure entry for reference to parameters and local variables. (This simplifies code generation within the procedure as the stack grows and shrinks.) As it happened, the Oregon Software Pascal – 2 compiler we used as a base for our efforts was originally designed for the Digital PDP – 11 computer, which has no such frame pointer. The compiler therefore already was capable of computing stack offsets without the benefit of the frame pointer, freeing register A6 for other uses. We dedicated it to serve as the pointer to `mem`, and modified the compiler to generate special code for references to that variable.

We then went through the code to TEX, identifying each variable and expression that could serve as a `mem` reference. The compiler pre-shifted constants in `mem` indices, and we changed each computed expression to pre-calculate the left shift (equivalent to multiplying by 8). (Most pointers are produced by arithmetic on pointers, so we changed statements like "`p:=p+1`" to "`p:=p+8`".) The resulting code eliminated the need for the shift instruction, so the code above simplifies to a single instruction:

```
move.l 0(A6,D0:L),D1 ;mem[N].rh -> D1
```

(This was *much* easier to write about than to actually perform; before we could be satisfied that we had identified every reference to `mem`, we found it necessary to create tools that would give us a full trace of each `mem` reference to compare with a standard trace.)

Example 2: instruction counting. We used several TEX jobs as performance benchmarks. The largest and most "life-like" of these was *The TEXbook*, which produces 494 typeset pages. The performance analyzer provided with the Macintosh development system is an interrupt-based profiler that samples the program counter every few milliseconds and produces an execution time profile by procedure blocks. While this was very indicative of major program flow, we found it somewhat unstable and too coarse for critical sections. We then built an instruction-by-instruction counter that produced an assembly listing with execution counts attached to each instruction. Here is a partial summary of counts for *The TEXbook* from April 20:

calls	instructions	%/total	procedure
5 703 564	190 236 375	16.435	get_next
1	152 982 983	13.217	main_control
82 008	60 914 195	5.263	hpack
53 313	57 517 172	4.969	hlist_out
229 313	55 271 629	4.775	try_break
8 036	46 325 505	4.002	line_break

This tool was somewhat expensive to use, slowing execution by a factor of 150 or so, but the information was happily consistent and precise. (Perhaps seductively so; we sometimes needed to remind ourselves that instruction counts were *not* directly related to processing time. In more than one case we chose instruction sequences with more instructions but faster execution, according to processor timing calculations.)

Our measurement tools gave us more than enough information to identify critical sequences for hand work. (a target-rich environment for assembler jockeys, so to speak.) Here is a similar summary from May 23 (the calls are identical):

instructions	%/total	procedure
128 071 928	13.322	get_next
109 146 891	11.353	main_control
53 136 456	5.527	try_break
51 951 382	5.404	hlist_out
46 203 165	4.806	line_break
43 949 702	4.572	hpack

Example 3: Procedure calling sequences. A slightly different view of instruction count data summarizes procedures in order of the number of calls:

calls	instructions	inst/call	procedure
5 703 564	190 236 375	33.4	get_next
2 840 411	32 782 196	11.5	get_xtoken
1 998 121	12 033 901	6.0	get_avail
1 150 392	39 422 420	34.3	get_node
1 150 334	14 954 342	13.0	free_node
1 057 213	11 336 157	10.7	get_token

Here, for example, we can see that we should consider replacing high-frequency calls to "get_avail" with the equivalent in-line code. (In fact, Knuth has already done this with a "fast_get_avail" WEB macro; we carried this idea a little further in our assembly code.)

More importantly, the sheer number of calls to these routines focused our attention on calling sequence overhead, both at the compiler template level, and later on special-case sequences for certain routines. Each single instruction eliminated from the calling sequence for "get_next" reduced the run-time for *The TEXbook* by almost 1 second on our Quadra test platform.

Perhaps the most interesting lesson from our experiments was the (in-)validation of some of the design concepts of the Pascal–2 compiler. The design team made some assumptions about program structure that are not true for TEX, e.g., that the procedure structure corresponded to significant work elements of the problem. We found instead that, in TEX, a typical procedure has a relatively small critical path that corresponds to the large majority of uses, and a much larger body of code for special cases and error recovery. Unfortunately, the compiler generated significant amounts of register traffic on each entry to "optimize" register usage over code that was never used!

Conclusions

The overall performance of TEX has improved by roughly a factor of three as of this writing, compared to previous Macintosh implementations via a standard (Apple) Pascal compiler. We believe there are significant gains still to be realized, although we are clearly seeing a diminishing return on efforts. So far, the reliability and maintainability are more than satisfactory; the assembly language TEX described herein is shipping in our current version of Textures, with no TEX problems reported to date.

Thanks

We would like to express our thanks to Professor Knuth for the TEX program itself, and especially for the TRIP test program, which has allowed us to easily locate bugs that would have been vanishingly obscure in more "normal" uses of TEX.