TEX on SMALL MACHINES

Kent S. Harris and Robert M. McClure

Unidot, Inc. Sunnyvale, CA

1. ABSTRACT

As small computers become more affordable, the demand for increasingly sophisticated software grows. Unfortunately, much of this software is large and does not easily fit on small computers. Emulation through interpretive techniques typically yields unacceptable performance. Reengineering of existing software is frequently desirable, therefore, for reasons of both cost and compatibility. The art of compressing large mainframe-developed programs into small address space machines has become very important. One such effort is the reengineering of the TEX typesetting system. Since this paper really deals with putting large software systems on small computers, and is not a paper on TEX itself, familiarity with TEX is assumed.

2. INTRODUCTION

TEX[1] is a recent creation by Donald E. Knuth at Stanford University. It is a system for typesetting beautiful books–especially books that contain a lot of mathematics–an area in which the cost effectiveness of computer typesetting over manual methods is obvious. The equation

$$\frac{1}{2\pi}\int_{-\infty}^{\sqrt{y}}\left(\sum_{k=1}^{n}\sin^2 x_k(t)\right)\bigl(f(t)+g(t)\bigr)\,dt$$

is a vivid example. That TEX is especially popular today compared to other computer typesetting systems is due primarily to TEX's level of sophistication and ease of use. TEX has recently been endorsed by the American Mathmatical Society for submission of machine readable input, a trend that is likely to grow quickly in the years ahead.

The goal of this project was to implement TEX on a small, preferably desktop size, computer. We also wanted to use a commercially available operating system that would provide additional typesetting tools such as editors and other text processing facilities. Since UNIX[2] had demonstrated superior text management functions over other operating systems and appeared as if it were becoming a de-facto standard for small machine operating systems, it was chosen as a basis for this implementation. UNIX was in the process of being ported to several 16-bit architectures by various manufacturers while already enjoying rather widespread usage in the DEC PDP-11[3].

Another issue was that of implementation language. The C language[4] seemed to meet the requirements best for programming this class of system on small processors. It provided the best combination of both high and low-level features of any language that had reasonable wide-spread distribution.

The final major goal was to avoid using compression techniques such as interpretive systems. Although interpretation is widespread today (most BASIC systems are interpretive), and do provide for very compact code, it usually incurs a large performance penalty, especially for programs requiring substantial computation. TEX was expected to be computation intensive.

The original version of TEX was written in SAIL, a language developed at the Stanford Artificial Intelligence Laboratory for DEC PDP-10's and 20's. Furthermore, TEX was a large program, even with 36-bit words. It seemed to have an insatiable appetite for memory while building pages of text. The idea of actually compressing TEX into a 16-bit address space without using interpretive techniques initially appeared quite absurd.

1) The "X" in TEX is actually a Greek chi, and therefore TEX is pronounced the same as the first syllable in technology. The name TEX is a registered trademark of the American Mathematical Society.

2) UNIX is a trademark of Western Electric.

3) DEC and PDP-11 are trademarks of Digital Equipment Corporation.

4) Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

The goal indeed seemed to be a formidable task. The methods employed to bring about the realization of "Table-top" TEX is what this paper's all about.

3. MULTI-PROCESS APPROACH

It was clear from the size of the SAIL version that TEX could not possibly exist as a single process within a 16-bit address space. Because UNIX does not support a runtime overlay system, we decided to split TEX into two separate but concurrent processes. We decided to use UNIX's simple but elegant system of *pipes*. The UNIX *pipe* is a mechanism by which the output of one process (*pass1*) is routed to the input of another process (*pass2*). One early question concerned whether there existed a division point in TEX where this simple tandem scheme could be implemented, or were feedback paths from *pass2* to *pass1* always necessary?

Figure 1 is a simplified block diagram of data flow. The vertical dashed line shows the most obvious place of division. The balancing of instruction and data space requirements between the two passes along with numerous other details also affected this division point. After examination of the boxes labeled "Main Control" and "State Stacks", it was determined feedback paths would, unfortunately, be required. *Pass1* and *pass2* had to exist as processes coupled by two *pipes*, one for communication in the forward direction and one for the reverse direction.

A few words about virtual memory techniques are in order. As much data (both predefined and generated) as possible was to be kept memory resident for obvious performance reasons. Since a 16-bit data space was simply not enough to hold the numerous tables required by TEX, most were moved to secondary storage and cached through memory resident buffers with buffer replacement done on a least-recently-used (LRU) basis. We will refer to these as virtual memory (VM) systems. How these VM systems are incorporated into various nooks and crannies of TEX will become clear shortly.

4. PASS ONE

The overall purpose of *pass1* is to break down input text into a stream of primitives and data for *pass2*, who does the real work. *Pass1* is responsible for macro definition and expansion and for managing other user defined token lists (such as the output routine, alignment texts, and mark texts). These token lists are kept as character strings instead of hash indexes or primitive codes as in the SAIL version. This provides for ease of *pass1* manipulation whereas the SAIL implementation reduces them to *pass2*-like entities. Subsequently, time honored string based algorithms can be used. Figure 2 shows our macro expansion stack frame layout.

The hash entry symbol table is managed on a linear collision basis–applying the hashing function to the symbol, using this value to index into the hash table, and then linearly scanning to find either a symbol match or a free cell. Contents of the hash table are address pointers to the symbol character string (see figures 3 and 4). The hash table is one-to-one with the first part of the equivalents table. The equivalents table contains key information for all primitive control sequences and user defined control sequences. This arrangement is essentially identical to the SAIL version with the exception that the hash table pointers and macro text pointers are actually VM pointers.

The VM system used to hold the output routine, alignment templates, and mark texts is illustrated in figure 5. The addresses shown are not mandantory and can be tailored for a particular user's needs. Additional VM systems can be added to expand allocated sizes of the various elements. However, the numbers shown are realistic. This mechanism of fixed size allocation fosters simplicity and saves memory with little loss of generality.

Two output routine definition areas are shown, allowing output routine redefinition without collision. The next four 4K (byte) blocks comprise the halign and valign template storage areas. The remaining memory up to the 32K point is currently unused. The second 32K is divided into 128 256-byte chunks. Each chunk can hold one *mark text*. Although there are only three possible marks per page of text (top, bottom, and first), *pass1* does not know where the page break will be. To solve this problem, we keep up to 128 *mark texts*. When *pass2* decides it's time to digest the output routine, it indicates to *pass1* how many marks were on the page just built. *Pass1* uses this information to manage the pointers and text buffers accordingly.

5. PASS TWO

The primary data structures of *pass2* are large and complex linked lists. For performance, these lists are memory resident. However, two tables are virtual in *pass2*-font information and the hyphenation exception dictionary.

Font information files typically range between 700 and 1000 bytes in length. Since TEX supports 32 font files, the need to keep these tables on secondary storage is clear. As before, font information files are cached on an LRU basis with a small number of files (usually four) in memory at one time. Since font changes are relatively rare, this causes little performance degradation.

In fact, the only VM system that has hindered performance significantly is the hyphenation exception dictionary. Currently, exceptions are kept in a file of sorted fixed length records and simply binary searched. There are superior methods that we expect to incorporate later.

Perhaps the most important implementation decision in *pass2* concerns execution speed rather than code volume. The SAIL version utilizes floating point exclusively for its *glue*[5] values. This approach is unacceptable considering the poor floating point characteristics of most 16-bit processors.

Moreover, output results will differ between various processors with slightly different floating point implementations due to differences in accumulated round-off. The solution, of course, is to use fixed point with appropriate scaling. There is a most conspicuous fly in this ointment. It is inherent in TEX's line-breaking and page-breaking algorithms that *glue* values have the same dynamic range between 0 and 1 as between 1 and ∞. This is essentially the definition of floating point!

The technique we chose is usually considered the worst possible solution–software emulation of floating point. The key, however, is the format used (figure 6). At first, a 16-bit exponent may seem a bit excessive, but since this is the natural width of arithmetic of most small machines, it provides for rapid manipulation. Astonishingly enough, TEX has performed very well utilizing this technique.

The SAIL version uses floating point exclusively for all *glue* and dimensional data. We limit the use of floating point to gain both space and speed performance. With the exception of 32-bit *glue* values and line widths used within the line-breaking code, all of TEX's internal dimensional values are kept as 16-bit integers. In this implementation of TEX, internal units are mils (.001 inch), providing for a range of ± 32.767 inches. For a standard 8.5 by 11 inch page, this is certainly suficient.

6. CONCLUSION

A few words about performance are in order. Actual measurements so far show that TEX on an Onyx C8002 system can process 8.5 by 11 inch pages of average complexity text in about 5 to 10 seconds per page (with hyphenation turned off). Currently, hyphenation degrades this by a factor of 2 or 3, but this is improvable. The system mentioned has as its processor a Zilog Z8002 16-bit microprocessor with 256KB of main memory, and a secondary store consisting of a 10MB Winchester disk with an average access time of about 55ms.

The techniques described here are only an example of those possible in the realm of software compression. The task of compressing software without gross performance degradation may not be a systematic one, but this example illustrates its feasibility.

7. REFERENCES

1. Knuth, D.E., *TEX and METAFONT, New Directions in Typesetting*, Digital Press, American Mathmatical Society (1979).

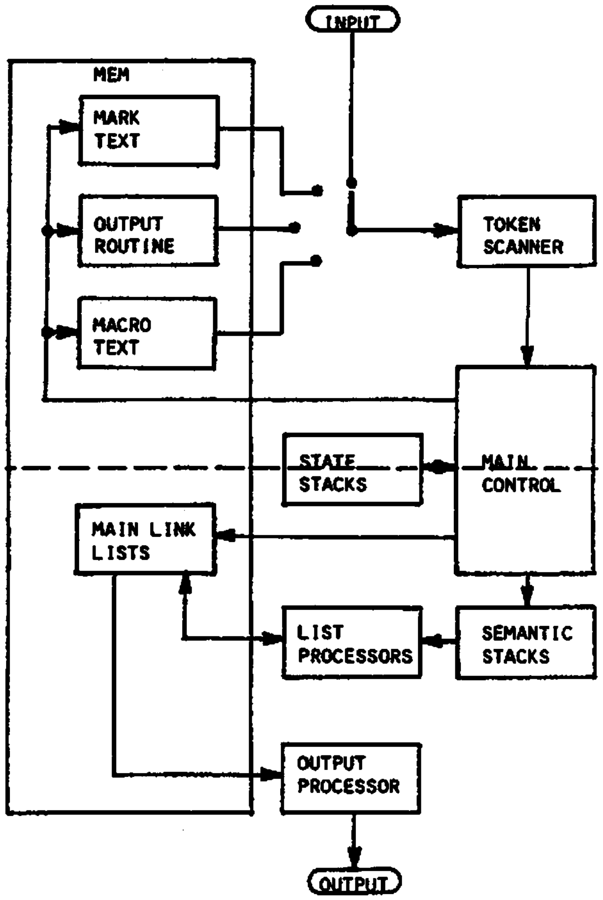

FIG. 1 - SIMPLIFIED DATA FLOW SAIL VERSION

5) One of the more interesting concepts in TEX is the idea that characters and combinations of characters (boxes) are held together with a flexible space called *glue*. After positional calculations are done, the *glue* is *set*.

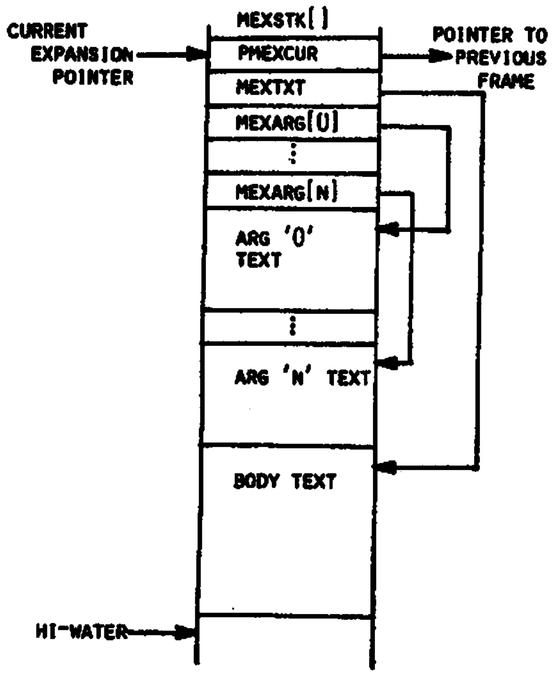

FIG. 2 - MACRO EXPANSION FRAME

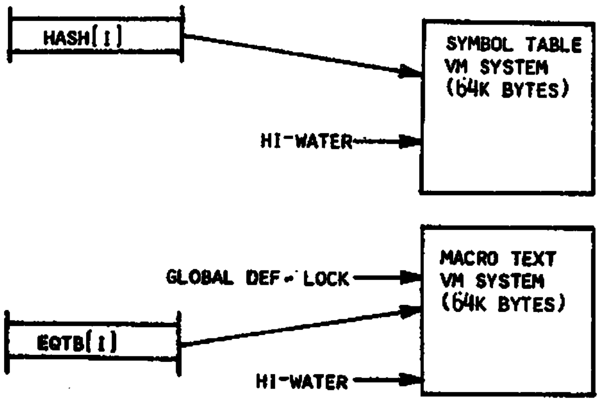
HASH[I]
SYMBOL TABLE VM SYSTEM (64K BYTES)
HI-WATER
GLOBAL DEF. LOCK
MACRO TEXT VM SYSTEM (64K BYTES)
EQTB[I]
HI-WATER

FIG. 4 - MACRO VIRTUALIZATION

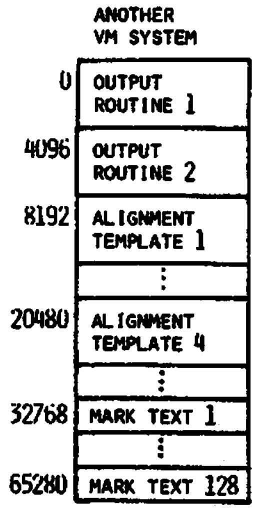

FIG. 5 - MISC. PASS1 VIRTUALIZATION

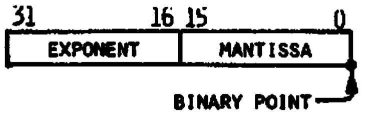

FIG. 6 - GLUE FORMAT

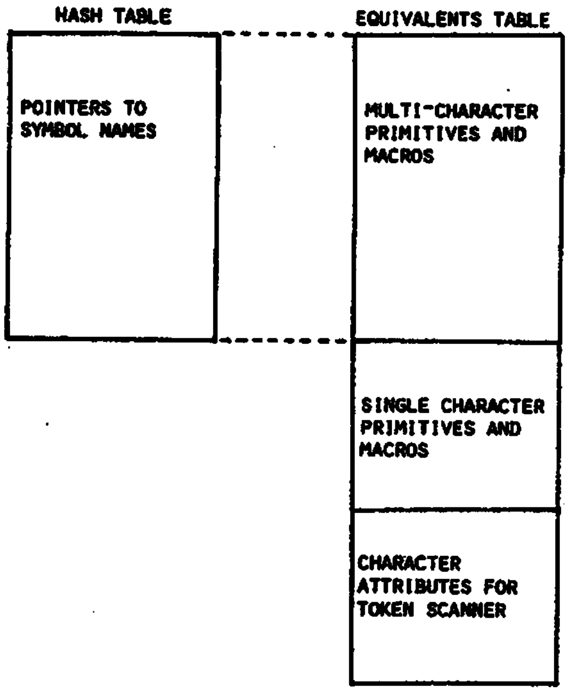
HASH TABLE
POINTERS TO SYMBOL NAMES
EQUIVALENTS TABLE
MULTI-CHARACTER PRIMITIVES AND MACROS
SINGLE CHARACTER PRIMITIVES AND MACROS
CHARACTER ATTRIBUTES FOR TOKEN SCANNER

FIG. 3 - EQUIVALENTS TABLE

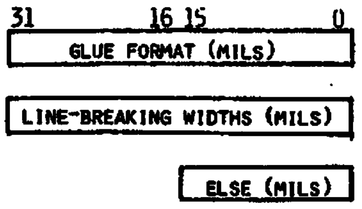

FIG. 7 - DATA FORMATS