AD-A119 439 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
OPTIMAL FONT CACHING,(U)
MAR 82 D R FUCHS, D E KNUTH N00014-81-K-0269
UNCLASSIFIED STAN-CS-82-901 NL

END
DATE FILMED
11-82

larch 1982

Report. No. STAN-CS-82-901

Optimal Font Caching

by

David R. Fuchs
Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, CA 94305

Optimal Font Caching

by David R. Fuchs and Donald E. Knuth

Computer Science Department
Stanford University
Stanford, California 94305

Abstract. An efficient algorithm is presented for communicating letter-shape information from a high-speed computer with a large memory to a typesetting device that has a limited memory. The encoding is optimum, in the sense that the total time for typesetting is minimized, using a model that generalizes well-known "demand paging" strategies to the case where changes to the cache are allowed before the associated information is actually needed. Extensive empirical data shows that good results are obtained even when difficult technical material is being typeset on a machine that can store information concerning only 100 characters. The methods of this paper are also applicable to other hardware and software caching applications with restricted lookahead.

Keywords: Cache memory, data structures, lookahead, optimum allocation, prepaging, typesetting, data reduction.

This research was supported in part by National Science Foundation grant IST-7921977, by National Science Foundation grant MCS-77-23728, by Office of Naval Research contract N00014-81-K-0269, and by the IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States government.

1. Introduction.

The purpose of this paper is to study a data-reduction problem that arises when computers are applied to phototypesetting. A page that is printed with modern typesetting equipment may be regarded as a gigantic matrix of 0's and 1's, where 0 represents a blank space and 1 represents ink. For example, the particular machine used in our experiments has approximately 19,000,000 such bits per square inch; therefore a typical page of technical text from a book like [4], which was printed on that machine, is essentially a matrix of more than 727 million bits. This data must be reduced by more than three orders of magnitude in order to be transmitted from the host computer to the typesetter at a rate of 9600 bits per second, if the page is to be finished in less than two minutes. Typical methods of data compression are considered excellent if they achieve a reduction factor of only 50 per cent, so it is clear that special techniques are needed if high-resolution digital printing is to be efficient.

The main factor accounting for this thousand-fold reduction in the number of information bits is, of course, the fact that pages are composed from letters that have comparatively simple shapes. For example, a typical character that measures 5×10 printer's points, where there are 72.27 printer's points per inch, has a digital pattern occupying about 182,000 bits on the machine mentioned above, but this pattern can be specified satisfactorily with about 250 bytes = 2000 bits.

Even with this reduction, however, there remain about 2500 characters per page, so about 5,000,000 bits still need to be transmitted. The problem would be simple if the typesetter knew all of the digital patterns for all of the letters, since we would merely have to transmit letter codes. But typical technical texts involve a variety of different fonts and special symbols, and many typesetting machines have only a limited local memory for the storage of character patterns. Therefore the character shapes must be transmitted from the host computer to the typesetter, and the only way we can compress this data is by using the fact that most characters are used again and again.

For example, suppose that the typesetter has enough memory to record the shapes of 60 characters. This is just barely enough for the letters a to z and A to Z, but we also need to deal with numerals and punctuation marks, together with italic and bold variations, and with changes in size and style. The standard industrial practice has been to solve the size-change problem by doing simple scaling operations, so that "8-point type" is obtained as an 80% reduction of "10-point type"; but typographers are very unhappy about this compromise, because the results were much better on the old hot-lead machines when every point size was designed separately. Fortunately it turns out that individual lines of text hardly ever need a great variety of characters even without the compromise; therefore the typesetter can use its memory as a "cache" for 60 characters, including the 30 or so that it needs on the current line.

The typesetter might also be able to accept a few more character descriptions that will be needed on subsequent lines, at the same time as it is setting type on the current line; these new characters can replace "dead" ones in the cache, and with luck the cache will be up to date at all times. For example, if there is time to make five adjustments to the cache on each line, 30 new characters can be brought in when a new font is desired, if the changes begin six lines in advance. By looking ahead to see which characters need to be sent in the future, the host computer can

control the typesetter's cache contents in an efficient way. The purpose of this paper is to examine suitable algorithms by which the host computer can exhibit such clairvoyant behavior, and to study how much is gained by such techniques.

Section 2 presents a theoretical model of a general cache allocation problem, and Section 3 derives an optimal allocation strategy for that model. Data structures and algorithms by which the optimum strategy can be computed with reasonable efficiency are described in Sections 4 and 5. The concluding section presents empirical results that illustrate what can be achieved.

Although this paper is oriented towards a particular application to typesetting, the reader is encouraged to speculate about how the same methods could be applied to the design of ultra-high-speed computers. One can imagine a pipelined arithmetic unit, playing a role analogous to that of the typesetter, taking orders from another computer, whose function is to preload a cache memory with numeric data, based on the knowledge of a particular algorithm's control structure. Instead of relying on the conventional architecture of a general purpose computer, one could apply the methods of this paper to a large class of important computation-intensive algorithms whose control structure is predictable.

2. A cache-allocation model.

Consider an alphabet of m possible characters that might be kept in a cache that can hold at most s characters at once. We wish to implement a sequence of commands of the following three types:

$L(i)$	Lock character i in the cache, where $1 \le i \le m$.
$U(i)$	Unlock character i.
G	Get any character and place it into the cache.

Such a sequence is called a "job." Character i is said to be "wedged" at a certain point of a job if more $L(i)$ commands than $U(i)$ commands have occurred before that point. We assume that the $U(i)$ command appears only when character i is wedged, so at any point in reading through a job, we will never have seen more $L(i)$ commands than $U(i)$ commands for any i. Furthermore we assume that there are never more than s different characters wedged at any one time; a job that does not meet this requirement needs a larger cache.

Initially the cache has s empty slots. When an $L(i)$ command occurs and character i is not present in the cache, we say that there is a "fault." In the case of a fault, the typesetter comes to a halt while character i is brought into the cache, either going into an empty slot or replacing some unwedged character. Similarly, at the time of a G command, any character not present in the cache can be brought into a cache slot that is not occupied by a wedged character. In this case, we do not consider that a fault has occurred since the typesetter is still busy doing a previous line. Thus, G commands allow us to anticipate L commands so that future faults are avoided. It is also possible to "pass" a G command, leaving the cache unchanged, if this seems more desirable than bringing in a new character.

Note that a character must be in the cache whenever it is wedged, because an $L(i)$ command guarantees that character i is present, and because no character can be replaced until it has become unwedged.

This model is more general than the "page reference" model that is usually used to study cache behavior in a virtual memory system. The page reference model is the special case in which there are no G commands, and where each $L(i)$ command is immediately followed by $U(i)$. Our model also assumes that we know the entire sequence of commands in a job before the job is begun.

In our application, the typesetting of a full line must operate in real time with no waiting for faults. Thus, a line of type containing the characters $i_1 i_2 \ldots i_r$ might be represented by the command sequence

$$L(i_1)\,L(i_2)\ldots L(i_r)\,G^k\,U(i_1)\,U(i_2)\ldots U(i_r),$$

if there is time to bring into the cache as many as k characters for future lines while the line is being typeset. The actual typesetting of the line starts after the command $L(i_r)$ has been completed. This sequence of commands ensures that all characters needed on the line will be present in the cache before typesetting takes place. The L instructions for line $(i+1)$ are not begun until the typesetter has completed line i, so that no characters needed on the line will get over-written before the typesetter is done with them.

The model does not specify what character is replaced at the time of a fault or of a G command. A "caching strategy" is a set of rules that govern what happens to the cache at such times. A "strategy trace" is the output of a strategy when it is fed a job; in other words, it is a list that records, at each G and L command, which character, if any, is to be brought into which cache slot.

3. An optimum caching strategy.

In this section we shall see that an intuitively plausible strategy for cache allocation actually minimizes the total number of faults, among all possible strategies for a given command sequence. (This generalizes Belady's well known "MIN" method in the page reference model [1,2,5].)

The strategy is simply this:

1) Whenever a character is brought into the cache, place it in an empty slot, if possible; otherwise let it replace an unwedged character i that never appears in a subsequent $L(i)$ command, if possible; otherwise let it replace the unwedged character i in the cache whose next appearance in an $L(i)$ command is as late as possible. Since at most s characters can be wedged at once, one of these three cases must always hold.

2) Whenever a G command appears, bring in the character i not currently in the cache, whose next appearance in an $L(i)$ command is as soon as possible, unless all unwedged characters currently in the cache will be locked by L commands that occur between the current G command and this $L(i)$ command, or unless no such character i exists.

When a character is brought into the cache by rule (2), its cache slot is selected by to rule (1). The case in rule (2) where no such i exists occurs when all the chacters needed by the rest of the job are already in the cache.

To prove that this strategy S is optimum, we shall compare its trace on any job to any other possible trace for the same job, and show that S's trace leads to no more faults. More precisely, let S_J be S's trace for any job J. If X_J is different from S_J, we shall construct a trace X'_J that has no more faults than X_J at any time, and X'_J agrees with S_J longer than X_J does. In other words, if X_J agrees with S_J on the first $t-1$ commands but differs from it at command number t, then X'_J will agree with S_J for at least the first t commands. Repeated application of this argument will show that S leads to the smallest possible accumulated number of faults at all times.

The construction we shall define makes use of a "trace completion subroutine". The input to the trace completion subroutine consists of a job J, a trace W_J that implements J, and a partial trace Y_J that is only defined through the $(p-1)$th command of J. The subroutine will complete the definition of Y_J, such that it is at least as good as W_J. The following conditions must hold just after command $(p-1)$ for both W_J and Y_J (omitting the implied subscript 'J'):

i) There are characters w and y such that the cache for W has the form $\{w\} \cup C$ and the cache for Y has the form $\{y\} \cup C$, for some set of characters C, where $w \notin C$ and $y \notin C$. In other words, the caches are identical except for at most one element.

ii) Trace W has had at least as many faults as trace Y.

iii) If the sequence of future commands causes w to be locked before y, where w and y are the characters mentioned in condition (i), then W has already had more faults than Y.

The second condition says that Y is no worse than W. The third condition says in effect that character w cannot be a "better" thing to have in the cache than y, unless Y can afford one more fault without falling behind W.

If these three conditions are satisfied, we shall say that "relation (w,y) holds for (W,Y,J)" at the current position in the job. The trace subroutine is called only when relation (w,y) holds for (W,Y,J) at command $p-1$. The subroutine proceeds by figuring out what Y should do for the pth command in order to preserve these invariant conditions. In other words, if relation (w,y) holds before the pth command of J, the subroutine shall define the next step of Y so that relation (w',y') holds after the pth command, for some w' and y'. The subroutine can now do the $(p+1)$th command, and so on, until Y has been defined for all of J. Since the invariants still hold, we know by (ii) that Y is no worse than W, so the subroutine does what was claimed.

Now to define Y on the pth command. We know that relation (w,y) holds. If $w = y$, so that both traces currently have the same cache contents according to condition (i), we simply let Y be the same as W on command p. Relation (w,y) still holds. (In this case, the next iteration of the subroutine will have $w = y$, so the $(p+1)$th action of Y will again be defined to be the same as that of W by this rule, and so on, so from this time henceforth Y is the same as W.)

On the other hand if $w \neq y$, note that both w and y must currently be unwedged, since w does not occur in Y's cache and y does not appear in W's. The following subcases arise in defining Y on command t:

a) If the command is $L(y)$, so that a fault occurs with trace W, suppose W replaces z by y. Trace Y has no fault, so it can't bring a character into the cache; but after the command $L(y)$, it is easy to check that relation (w, z) holds, because condition (ii) implies that W has now had more faults than Y.

b) If the command is $L(w)$, so that a fault occurs with Y but not W, then Y replaces y in its cache by w. This replacement is legitimate, because y is currently unwedged. Afterwards relation (w, w) holds, since condition (iii) implies that this case can arise only if Y could afford at least one fault.

c) If the command is $L(i)$ where $i \neq w$, $i \neq y$, and $i \notin C$, a fault occurs in both traces. If W replaces w by i, then Y replaces y by i; relation (i, i) holds. Otherwise, if w replaces z by i for some $z \in C$, then Y also replaces z by i, and relation (w, y) still holds.

d) If the command is $L(i)$, where $i \in C$, or if it is $U(i)$, no fault occurs for either W or Y, and relation (w, y) remains true.

e) If the command is G and if W replaces w by v, then Y replaces y by v, and relation (v, v) holds.

f) Finally, if the command is G and if W replaces z by v for some $z \in C$, or if W does nothing, then Y likewise replaces z by v or does nothing. Relation (w, y) still holds.

This completes the definition of Y from W, except in one degenerate case: Suppose that the command is $L(y)$ and that W brings character y into an empty position in its cache. This is a variation on case (a), where Y cannot braing in a character because no fault has occurred. We can avoid this situation by assuming that the set C in condition (i) always contains $s - 1$ elements, i.e., that there are no empty positions. For we can fill each empty position with distinct dummy characters that do not appear in any commands; this convention makes the proof go through.

Now that the trace subroutine has been specified, we shall use it to prove the optimality of strategy S. Suppose X_J is any trace different from S_J for some job J. The first difference occurs at the tth command in the traces. We will create a trace X'_J to be the same as S_J up to and including the tth command, such that relation (x, y) holds for (X_J, X'_J, J), for some x and y. We can then call the trace subroutine to complete X'_J such that it is at least as good as X_J. Then we will be able to repeat the process with S_J and X'_J, getting X''_J, which is like S_J through the $(t+1)$th command and at least as good as X'_J (and therefore at least as good as X_J), and so on. The final result is that S_J is the same as $X_J^{(n)}$ for some $n \leq \text{length}(J)$, and S_J is no worse than X_J. Since J and X are arbitrary, this will prove that S is optimal. (Once again, we will drop the J when it is understood.)

So the only task left is to show that if X' is defined to be the same as S through command t, then relation (x, x') holds for (X, X', J) for some x and x'. Just before command t, both X and X' have had the same number of faults, and both their caches have the same contents. The tth command must either be an L command that causes a fault, or a G command on which S and X

didn't both pass. Suppose first that command t is $L(i)$, where i is not in either cache at time t, and X replaces character j by i while S replaces character k by i. Relation (k,j) holds for (X,X',J) because rule (1) guarantees that character k is not locked before character j.

Similarly, if the tth command is G, and if X passes while S replaces k by z, relation (k,z) holds for (X,X',J) since rules (1) and (2) imply that k is not locked before z. And if X replaces j by w when implementing a G command, while S passes on that G, relation (w,j) holds, since rule (2) ensures that w is not locked before j.

The only remaining case is that the tth command is G, and that trace X replaces j by z while S replaces k by w. If $j=k$, relation (z,w) holds for (X,X',J) because of rule (2). On the other hand, if $j\neq k$, we have to invoke the trace subroutine twice before obtaining a trace that dominates X and agrees with S on commands 1 through t: we first let Z be a trace that replaces k by z, so that Z is a mixture of X and S. At this point, relation (k,j) holds for (S,Z,J), because of rule (1). Completing Z with the trace subroutine, we now have a trace that is still different from S in the tth command. This command, however, is a G command where Z replaces k by z, while X' replaces k by w, and so relation (z,w) holds for (Z,X',J).

We have now shown that it is always possible to set up X'_J to obey the invariant conditions, and this finally completes the proof that S is optimum.

4. Implementing the optimum strategy.

Let m be the total number of possible characters, let s be the size of the cache, and let n be the number of commands in job J. Our goal is to have an algorithm that computes the optimal trace S_J. Job J's commands are in arrays $\hat{o}p$ and *char* before the algorithm begins. If the jth command is $L(i)$, $U(i)$, or G, then

$$\begin{aligned} & op[j]='L', \quad && char[j]=i, \\ \text{or}\quad & op[j]='U', && char[j]=i, \\ \text{or}\quad & op[j]='G', && char[j]=\text{undefined}, \end{aligned}$$

respectively, for $1\le j\le n$. For the present we shall pretend that we have enough memory to store all n of the commands at once.

The algorithm records the resulting trace in the *cache* and *char* arrays. If $cache[j]>0$, step j of the trace says to bring character $char[j]$ into cache slot $cache[j]$; and when $cache[j]=0$, then no character is to be brought into the cache during step j. Thus, if a fault occurs at the jth command, the algorithm should set $cache[j]$ to the cache position that S allocates to $char[j]$, where $1\le cache[j]\le s$. If $op[j]='G'$ and if strategy S replaces cache position k by character c, the algorithm should set $cache[j]\leftarrow k$ and $char[j]\leftarrow c$. In other cases the algorithm should set $cache[j]\leftarrow 0$. Note that the *char* array is altered by this algorithm, but only in G commands.

Our algorithm works with two pointers p and q, where $1\le p\le q\le n+1$. Pointer p represents the current position where we are defining the trace; we shall say that the trace has been defined "up to time p," thinking of a clock that advances when p increases. Pointer q looks ahead to the first L command that locks a character not in the cache at time p; if no such commands

exist, we have $q = n + 1$. For each character i there are two values

$$slot[i] = \begin{cases} 0, \text{ if } i \text{ is not present in the cache at time } p; \\ \text{the cache position of } i, \text{ otherwise.} \end{cases}$$

$usage[i]$ = the number of $L[i]$ instructions before command q minus the number of $U[i]$ instructions before command p.

For each cache position $k \le s$ we will have

$$contents[k] = \begin{cases} i, \text{ if } slot[i] = k; \\ 0, \text{ if position } k \text{ is empty.} \end{cases}$$

Suppose that character i appears in r_i different "lock" commands, numbered $j_{i1} < j_{i2} < \cdots < j_{ir_i}$. A preliminary pass over the *char* array suffices to fill two auxiliary arrays $first[i]$ and $next[j]$, for $1 \le i \le m$ and $1 \le j \le n$, so that

$$first[i] = j_{i1}, next[j_{i1}] = j_{i2}, \ldots, next[j_{ir_i}] = n + 1.$$

If $r_i = 0$, we can set $first[i] = n + 1$, although this value won't be looked at so it really doesn't matter.

Initially $p = q = 1$, $usage[i] = slot[i] = 0$ for $1 \le i \le m$, and $contents[k] = 0$, for $1 \le k \le s$. The initial value of $first[i]$ will be j_{i1} as stated above; but as the algorithm progresses, $first[i]$ will be updated so that it is the smallest element $\ge q$ of the set $\{j_{i1}, j_{i2}, \ldots, j_{ir_i}\}$. For convenience, we also set $first[0] = n + 1$ and $usage[0] = 0$, so that 0 is essentially a character that never appears.

One more thing completes this family of data structures: There is a priority queue Q of all cache positions k such that $usage[contents[k]] = 0$; these positions are ordered by $first[contents[k]]$. Initially Q contains all positions $\{1, \ldots, s\}$ in arbitrary order. Any suitable scheme for implementing a priority queue can be used for Q; if s is small, a sorted linear list will be adequate, while if s is large a method that requires at most $O(\log s)$ steps per operation might be most appropriate. Note that Q contains all cache positions whose contents will be unwedged at all times between p and q inclusive, sorted in order of the first time they will be locked after time q.

The algorithm proceeds by advancing p one step at a time, first moving q as far as it can ahead of p:

```
while p ≤ n do
    begin integer i; comment bring this character into the cache next;
    (move q forward until reaching L(i) with i not present);
    (process command p, attempting to bring in i);
    p ← p + 1;
    end.
```

The subalgorithm that moves q forward will set i to the character that should be brought into the cache next; this is the character not present at time p that is going to be needed soonest. If no such characters exist, we will have $q = n + 1$ and $i = 0$:

```
⟨move q forward until reaching L(i) with i not present⟩ =
        begin i ← 0;
        while q ≤ n and i = 0 do
            if op[q] ≠ 'L' then q ← q + 1
            else  begin i ← char[q];
                  if slot[i] > 0 then
                      begin first[i] ← next[q];
                      if usage[i] = 0 then delete slot[i] from Q;
                      usage[i] = usage[i] + 1;
                      q ← q + 1; i ← 0;
                      end;
                  end;
        end;
```

When deleting *slot*[*i*] from Q, it may help to know that *slot*[*i*] is at the rear of Q; i is the character that would currently be chosen last for replacement in the cache on the basis of priority since it has the minimal value of *first*[*contents*[*i*]].

The processing of command p has two main components, depending on whether the command is for unlocking or bringing in a character:

```
⟨process command p, attempting to bring in i⟩ =
        begin cache[p] ← 0; comment this value may be changed later;
        if op[p] = 'U' then ⟨unlock char[p]⟩
        else if i > 0 and (op[p] = 'G' or p = q) then ⟨try to bring in i and advance q⟩
        end.
```

The first of these is a simple update to the data structures:

```
⟨unlock char[p]⟩ =
        begin integer j; comment unlock this character;
        j ← char[p];
        usage[j] ← usage[j] − 1;
        if usage[j] = 0 then insert slot[j] into Q with key first[j];
        end.
```

The other operation is the most interesting:

```
⟨try to bring in i and advance q⟩ =
        if Q is empty then
            begin if p = q then report overflow error;
            end
        else  begin integer k; comment change this cache position;
            delete k from Q with maximum first[contents[k]];
            cache[p] ← k; char[p] ← i;
            slot[contents[k]] ← 0; slot[i] ← k; contents[k] ← i;
            first[i] ← next[q]; usage[i] ← 1; q ← q + 1;
            end;
```

Note that if $p = q$, we have $op[p] =$ 'L' and a fault has occurred. An overflow error is detected if $p = q$ and Q is empty, since this means that the pth command is trying to lock some character not in the cache, while s other characters are already wedged.

It is straightforward to verify that the operations preserve the invariant relations we have stated for the data structures, and therefore that an optimum strategy S is being found.

Note that the running time of this implementation is of order $m + n \log s$. If a lot of G commands are present, the pointer q tends to be quite far ahead of p so that comparatively few characters in the cache will have zero usage; thus Q will not contain many entries, and t' running time will be essentially linear. Thus, additional G commands will make the algorithm er, even though they cause it to find the optimum over a larger space of possible strategies.

5. Refinements to the implementation.

The algorithm of Section 4 can be modified in various ways to improve its efficiency and to take account of practical constraints.

In the first place, the running time will be improved if we realize that p usually increases several times before q moves. If $i > 0$, so that $op[q] =$ 'L' and $char[q] = i$ needs to be brought in, pointer q will stand still until the code (try to bring in i and advance q) is actually executed. Therefore the main loop of the program can be reorganized with a loop on q followed by a loop on p followed by an operation that increases both p and q.

In the second place, the fact that n is large means that it is undesirable to have a separate array $next[j]$ for $1 \le j \le n$; this additional array limits the number of commands that can be accommodated. By looking at the way this algorithm uses $next$, we can see that the $next$ and $char$ arrays can be overlapped at the expense of a (shorter) array $second[j]$ for $1 \le j \le m$. The new conventions are as follows, if the "lock" commands following time q for character i are $j_{i1} < \cdots < j_{ir}$:

If $r_i = 0$: $first[i] = n + 1$, $second[i] =$ undefined.
If $r_i = 1$: $first[i] = j_{i1}$, $second[i] = n + 1$, $char[j_{i1}] = i$.
If $r_i = 2$: $first[i] = j_{i1}$, $second[i] = j_{i2}$, $char[j_{i1}] = i$, $char[j_{i2}] = n + 1$.
If $r_i \ge 3$: $first[i] = j_{i1}$, $second[i] = j_{i2}$, $char[j_{i1}] = i$, $char[j_{i2}] = j_{i3}$, $\ldots$, $char[j_{i(r_i-1)}] = j_{ir_i}$, $char[j_{ir_i}] = n + 1$.

The operation '$first[i] \leftarrow next[q]$', which appears twice in the algorithm of Section 4 at times when $r_i > 0$, is now changed to the following code:

```
begin integer j;
j ← second[i]; first[i] ← j;
if j ≤ n then
    begin second[i] ← char[j]; char[j] ← i;
    end;
end.
```

In the third place, we must face the fact that jobs generally have more commands than could possibly be held in our computer's memory. Rather than having the algorithm read in an entire

job, figure out the cache trace and put it into the *cache* array, we will instead regard the cache allocation algorithm as a coroutine that does the caching "on line" as it reads the commands. In other words, if we can store only n_0 commands in memory at once, we would like to have an algorithm that will have read n_0 commands ahead of the one it is actually implementing at any given time. Thus, when the coroutine is called on to provide the value of $cache[x]$, it has elements x through $(x+n_0-1)$ of the *char* and *op* arrays in cyclic buffers in memory. The coroutine figures out what to do for step x, and then it reads in command $(x+n_0)$, over-writing $op[x]$ and $char[x]$.

When lookahead is limited to n_0 future commands, we might not discover a truly optimum trace. But the only errors we make would be to remove certain items from the cache in a different order when those items are not used at all during the next n_0 steps. If n_0 is large enough compared to the cache size, it is highly likely that all such items will leave the cache anyway, even in an optimal trace; so a limited-lookahead method will usually be no worse than the optimum. Indeed, our proof of optimality in Section 3 shows that a variety of strategies will usually perform no worse than strategy S.

Implementation of the coroutine philosophy means that we need to update the *first*, *second*, and *char* arrays on-line instead of assuming that they have been initialized by a preliminary pass over all the commands. For this purpose we need another array $last[i]$ for $1 \le i \le m$, containing the value of j_{ir_i}, if $r_i > 0$; we leave $last[i]$ undefined if $r_i = 0$. Furthermore some other sentinel value must be used instead of $n+1$ in the *first* and *second* arrays, since we don't know what n is. We shall use 0; the test '$j \le n$' above should therefore be changed to '$j > 0$'.

The algorithm now starts by filling up the *op* and *char* arrays with the first n_0 commands in the job, the *first*, *second* and *last* arrays are set up to reflect these commands, and p and q are set to 1. The entire data structure must be kept up to date as p and q change. For instance, as p is incremented to 2, the algorithm should put command number n_0+1 into $op[1]$ and $char[1]$, and update the *first*, *second*, and *last* arrays to reflect this new command. Thus, the statement '$p \leftarrow p+1$' is replaced by:

```
⟨advance p⟩ =
    begin (op[p], char[p]) ← next command in the job;
    if op[p] = 'L' then
        begin integer i; i ← char[p];
        if first[i] = 0 then
            begin first[i] ← p; second[i] ← 0;
            ⟨if slot[i] ∈ Q then change its key⟩;
            end
        else  begin char[p] ← 0;
            if second[i] = 0 then second[i] ← p
            else char[last[i]] ← p;
            end;
        last[i] ← p;
        end;
    if p = n0 then p ← 1 else p ← p + 1;
    end.
```

The statements '$q \leftarrow q + 1$' are changed to:

if $q = n_0$ **then** $q \leftarrow 1$ **else** $q \leftarrow q + 1$;

A few other changes to the code are required to keep q from incrementing when it gets n_0 commands ahead of p.

A "dead" character is one that, as far a we can tell from our limited lookahead, will never again be used in the job. Thus, character i is dead if and only if $usage[i] = 0$ and $first[i] = 0$. It is convenient to split Q into two separate parts: Q_0, which is simply an unordered set of all cache positions which are empty or contain dead characters, and the remaining part Q_1, which is a priority queue ordered by the nonzero key values $first[contents[k]]$. These key values are to be "circularly ordered" in the sense that we regard $x > y$ if $x < p \le q \le y$, since x is one lap ahead of y in such a case. Note that the operation (**if** $slot[i] \in Q$ **then** change its key) simply removes $slot[i]$ from Q_0 and enters it into Q_1 with key p, which will be higher than any other key currently in Q_1. The elements of Q_0 are all regarded as having higher keys than those of Q_1.

It is a simple matter to fill in all the remaining details: to take care of shutting down the input operations when all commands have been read and to terminate the coroutine when all of the cache commands have been implemented.

6. Empirical tests.

The authors have used these procedures to drive an Alphatype CRS phototypesetter, producing such technical books as [4]. In this application the characters in the cache have variable size, so the actual cache storage is allocated dynamically. When a new character is brought into the cache, there might already be room, but on the other hand, it might be necessary to remove several other characters before a hole appears that is large enough to accommodate an especially large newcomer. The number of G commands at the end of a line is not fixed, because it depends on the sizes of characters that are actually brought in.

In other words, the theoretical model studied earlier in this paper was a rather drastic simplification of the actual problem that had arisen in practice. As usual. But (as usual) the theoretical considerations provided valuable guidelines for a practical implementation, and by using an algorithm that is optimal or near-optimal under the simplifying assumptions, the authors were able to achieve quite satisfactory results even though those assumptions were violated.

Indeed, it would almost surely be unfeasible to develop an optimum strategy that takes account of all the details of the actual application, since the problem of optimum dynamic storage allocation is already NP-complete before we add the extra complexities of cache management. (See [3], problem SR2.) Instead of worrying about special schemes for dynamic allocation, the authors found that it was sufficient to replace unwedged characters simply on the basis of their priority, without regard to their size or to the priorities or sizes of their neighbors.

Figure 1 shows a sample text that was subjected to a variety of experiments discussed below. This text had been used to debug the TEX typesetting system in 1978, and it also provided the style pages in the design of [4]; thus it represents a wide variety of different things that happen in a 700-page book, compressed into about four pages. It involves the typesetting of 5211 characters, of which 576 are distinct when size variations are taken into account.

The task of driving the authors' typesetting equipment can be described in terms of the abstract model of Section 2 as the problem of implementing a sequence of commands having the following general form:

> Lock all characters used on line k;
> Tell the typesetter to start setting line k;
> If time permits, issue G commands to bring in future characters;
> Unlock all characters used on line $(k-1)$.

We do this for $k = 1, 2, \ldots$, except that the pattern changes in special cases. The term "line" means a sequence of characters that are to be typeset at the same baseline; thus, a complex mathematical formula might actually occupy many lines. There is usually time to preload future characters into the cache, because the time to transmit the information about *what to set on line* k is usually less than the time for the actual typesetting of that line.

Note that there are generally two consecutive lines wedged in the cache at once, since line $(k-1)$ isn't unlocked until after line k has been locked; this is due to buffering inside the typesetting machine. In emergency situations, when the ordinary policy would overload the cache, line $(k-1)$ will be unwedged sooner and the controlling process will pause to make sure that the buffer is clear; the cache will also be repacked at such times in order to make all of the available memory appear in consecutive locations. Also, if the typesetter is still busy doing line k when the controlling process begins to tell it about typesetting line $(k+1)$, the typesetter will stop taking commands until it is through with line k. Note that this allows characters from line $(k+1)$'s G commands to be brought in while line k is still being typeset. Line $(k+1)$'s L commands that cause faults will not entirely overlap, since the G commands should account for most of the time that the typesetter spends on line k.

Special actions occur at the beginning of a page: If the film has to move comparatively far in order to be in the proper position to start the new page, there is extra time to preload font information, hence the controlling process issues additional G commands. In particular, the characters for the first lines of the first page will generally have been brought into the cache by the time the typesetter is positioned at the top baseline.

Several dozen experiments were performed on Figure 1 in order to get some idea as to how the algorithm performs under various conditions. The cache size was varied so that it would be able to hold approximately 50, 75, 100, 125, or 150 characters; we shall refer to these sizes as *C50, C75, ..., C150, respectively. The speed at which font information could be transmitted was* varied so there was free time to send either an average of six new characters per line (i.e., about six G instructions after each line), or about 4.5 new characters per line, or no such characters; in the latter case, no G commands are given, so the algorithm must minimize the total number of characters transmitted. We shall refer to these transmission speeds as G6, G4.5, and G0. The algorithm was also run in four modes: (i) with full lookahead; (ii) with internal memory cut back so that only about 12 lines of data could be accommodated at once; (iii) with internal memory cut back to only about 6 lines; and (iv) with full lookahead but with the priority queue decisions reversed so that the *worst* possible cache replacements were made whenever the algorithm had to

take something from the cache. These four lookahead modes will be called L∞, L12, L6, and L0, respectively. Five cache sizes, three speeds, and four lookahead modes make for sixty combinations, and so sixty experiments were performed and the resulting numbers of faults are shown in Table A.

Table A
FAULTS THAT OCCUR WHEN TYPESETTING FIGURE 1

	G0				G4.5				G6				
	L0	L6	L12	L∞	L0	L6	L12	L∞	L0	L6	L12	L∞	
C50	1881	1060	1040	1037	572	268	268	254	378	198*	204	197	C50
C75	1863	960	856	834	389	91	76	69	146	35	31*	32	C75
C100	1854	954	789	752	353*	79*	30	27	110	20*	0*	3	C100
C125	1821	941	786	699	381	83	26	12	22	22	0	0	C125
C150	1819	917	779	614	356	66	26	9	0*	22	0	0	C150

(Asterisks denote "anomalous" values that are surpringly low.)

These results are quite encouraging. Consider first the G0 case, when no "freeloading" is done: At least 576 faults must occur, since each distinct character must be brought in at least once, and the table shows that a caching strategy with lookahead is able to make sure that only a few characters need to be brought in twice. The number of faults under G4.5 is substantially less, even for the unusually complicated text of Figure 1; and with G6 and a moderately large cache the faults disappear entirely.

The starred entries in Table A show interesting anomalies where a lucky combination of circumstances led to fewer faults than would be expected. Consider, for example, the cases with G4.5 and L6 or L0, where the cache size C100 turned out to be slightly better than C125. The reason was that these inherently nonoptimal strategies made better guesses in the C100 case. Another interesting example is the case G6 and C150, where the supposedly pessimal strategy L0 actually did better than L6. The reason here is that L0 only pessimizes the choice of cache replacements. The other part of our algorithm, which looks ahead to find the next candidate for G bringing in, remains optimum; and when there are enough G's, this part of the algorithm is strong enough to make the replacement strategy immaterial. On the other hand the L6 restriction curtails the effectiveness of the G lookahead as well as the replacement lookahead, so L6 can come out worse. The 22 faults occurred at the beginning of Figure 1's page 3, where a conversion from nine-point to ten-point type takes place; L6 wasn't prepared for so many changes all at once.

The most interesting anomaly arose in the case C100 and G6, when the suboptimal strategy L12 actually turned out to be better than the supposedly optimal L∞! A careful examination of what happened shows that this was a case of good luck for L12 and bad luck for L∞. It all started when the typesetting was going along routinely, about ten lines from the bottom of page 1; both L∞ and L12 were doing approximately the same thing, but with minor variations so that their dynamic storage allocation patterns in the font cache were quite different. Both strategies had succeeded in looking rather far ahead, and they were beginning to bring in the eight-point uppercase letters needed for the caption at the top of page 2. But when the "optimal" L∞ strategy had

successfully brought in the eight-point 'Q' and 'L', its cache had no free blocks big enough to bring in the 'S'. The restricted L12 strategy, on the other hand, had a fortuitous memory configuration that allowed it to bring in not only the 'S' but also the 'I' and 'N'. This put L12 three characters ahead of L∞, and it retained a three-character advantage all the way through page 2 and the beginning of page 3, where comparatively rapid font changes caused the lookahead to evaporate. Finally L12's lead manifested itself on the line before (1) on page 3; three faults occurred when L∞ had to bring in 'W' and the two pairs of quotation marks.

Note that L12 was almost never a great deal worse than L∞, in any of the cases, so it appears that a restricted lookahead still makes a satisfactory approximation to optimal behavior. In the authors' application it turns out that there is enough core memory to look about 2500 lines ahead; experiments show, however, that L50 is essentially equivalent to L∞, thus the storage requirements can be reduced greatly from what we originally thought would be necessary.

Figure 2 shows a detailed trace of what went on in the experiment for case (G4.5, L∞, C125). The horizontal axis separates the 834 characters that were brought in during the time Figure 1 was being typeset; all but 12 of these were brought in during *G* commands, while the remaining 12 were faults. The vertical axis represents the 314 lines in Figure 1. The graph shows two zig-zag paths, where the upper one represents each character's first use. Thus, the upper path is far above the lower path when characters are being preloaded many lines ahead, while the two paths touch each other when a fault has occurred. The lower path has a somewhat erratic behavior: occasionally we find a horizontal segment on that path, representing a line that introduces many new characters. (The worst cases are the line following 'EXERCISES—Special set' and the line beginning '3.3.3.3. This subsection doesn't exist', both of which required 31 new characters to be preloaded in order to avoid faults.) The upper path, on the other hand, is more regular, because there is roughly the same amount of time for preloading characters on each line. Variations in the upper path occur when the characters to be brought in are especially large or small, or when the line being typeset is short (as at the end of a paragraph), or when the baselines are far apart; but these changes are comparatively minor.

Sometimes the cache is full, so that the lookahead procedure stops and the current *G* commands are not used. This is indicated in Figure 2 by the symbol '⊢' on the upper path; the first such incidents occur near the bottom of page 2 in Figure 1, and a more significant stoppage occurs during the big displayed equations near the bottom of page 3.

Before developing the algorithms described above, the authors did a hand simulation on some sample text using the assumptions (C100, L∞, G6), since these parameters appeared to be appropriate for the typesetting equipment that Stanford planned to acquire. The success of caching with these parameters, in spite of the multiplicity of fonts needed to typeset difficult technical material, encouraged us to proceed further. Two years later, after the hardware and software were put into production, we found that G4.5 was more appropriate than G6, because time-sharing interfered with transmissions to the typesetter; however, this was compensated by saving space in the typesetter software so that C125 was more representative of the actual cache size. In fact, the new Alphatype model 400 arrived with additional cache memory, so that our current software corresponds to C155 and faults hardly ever occur.

Note that the strategy of Section 3 does not minimize the number of times a character is

brought into the cache; it only minimizes the number of faults. For example, consider a cache of size 2 and the job

$$L(1)\ L(2)\ U(1)\ G\ U(2)\ G\ L(3)\ L(1)\ U(1)\ U(3).$$

Strategy S will bring in 3 at the first G, then bring in 1 at the second; the alternative of passing on the first G and bringing in 3 on the second would be preferable if we were trying to minimize the number of brings.

Each time a character is brought into the cache and does not cause a fault, typesetting is not slowed down, but the amount of information that must be sent to the typesetter does increase, so it is desirable to try to minimize the number of times characters are brought into the cache. The algorithm of Sections 4 and 5 can be modified to "pass" a G if there are no dead characters in the cache (i.e., if Q_0 is empty), provided that the lookahead pointer q is sufficiently far from p that it is reasonably safe to assume we will be able to avoid faults by acting on future G's.

For example, suppose lookahead stops whenever it would require the replacement of a non-dead character, provided that the algorithm has looked ahead so far that the next character to be brought in is 16 or more lines away from the current line being typeset. Let us call this variant U16. Then the algorithm may well be able to avoid rashly replacing characters that are not dead, by holding back until a character becomes dead, without seriously risking future faults. Figure 2 shows 834 characters brought in when the parameters are (L∞, G4.5, C125); but if the distance between the upper path and lower path were constrained to be no more than about 16 or so, it is plausible to believe that we would end up bringing in characters fewer times, and we might even be able to approach the optimum of 699 achieved in the case G0. The following data show what happens for L∞, G4.5, and C125:

	U∞	U24	U16	U8	U0
faults	12	24	24	26	105
brings	834	831	795	761	725

With U16, there are 39 fewer characters brought into the cache, at a cost of 12 faults.

But Figure 1 is not a typical example. Therefore further tests were made on "real" data. The text of Section 3.5 of [4] is representative of the difficulties of a normal mathematical paper, so it serves as a good indication what we can usually expect. This second test case, which amounts to 28 typeset pages, involves the setting of 57912 characters, 660 of which are distinct. When the algorithm was applied with parameters (L∞, G4.5, C125, U∞) there were only 17 faults, and these all occurred near the very beginning. The total number of characters brought in to do the whole job was 2745; and with the U16 heuristic, this dropped to 2131, while the number of faults remained at 17.

Several other experiments were made in the 3.5 file, holding all but one of the settings (L∞, G4.5, C125) fixed. When L∞ was changed to L12, there still were only 17 faults; restricting further to L6 increased them slightly to 44. And when L∞ was changed to the "pessimizing" L0, the result was 17 again! Thus, the lookahead process appears to be powerful enough to achieve optimality without the refinement of the priority queue, when we consider typical data, provided that the G speed and the cache size are suitably large.

As expected, the 17 faults vanished at speed G6. Reducing the speed to G3 increased the number of faults to 65; these occurred only at the beginning and at the switch to nine-point type for the exercises. With speed G1.5 there were 248 faults, and with speed G0.1 there were 1144; speed G0 gave 1569. (This compares with

G6	G4.5	G3	G1.5	G0.1	G0
0	12	112	333	613	699

in the case of Figure 1.)

Increasing the cache size to C150 did not reduce the number of faults below 17. With a setting of size C100 there were 26 faults, while C75 gave 201. Size C50 was not quite large enough to hold all of the characters wedged in one of the nine-point lines; C52 gave 1406 faults.

We can summarize these requirements by saying that typical technical text can be typeset with negligibly few faults provided that the algorithm of this paper is used in connection with the following resources:

i) A cache in the typesetter capable of holding about 125 character shape descriptions;
ii) Time to preload about 4 characters per line without slowing down the typesetting process;
iii) Enough memory in the host computer to look ahead about 12 lines (i.e., about 750 characters) in the text to be typeset.

Bibliography

[1] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal* 5 (1966), 78–101.

[2] L. A. Belady and F. P. Palermo, "On-line measurement of paging behavior by the multivalued MIN algorithm," *IBM Journal of Research and Development* 18 (1974), 2–19.

[3] Michael R. Garey and David S. Johnson, *Computers and Intractability*, San Francisco: W. H. Freeman, 1979.

[4] Donald E. Knuth, *The Art of Computer Programming* Vol. 2: *Seminumerical Algorithms*, Reading, Mass.: Addison-Wesley, second edition, 1981.

[5] R. L. Mattson, J. Gecsei, D. R. Sluts, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal* 9 (1970), 78–117

Figure 1

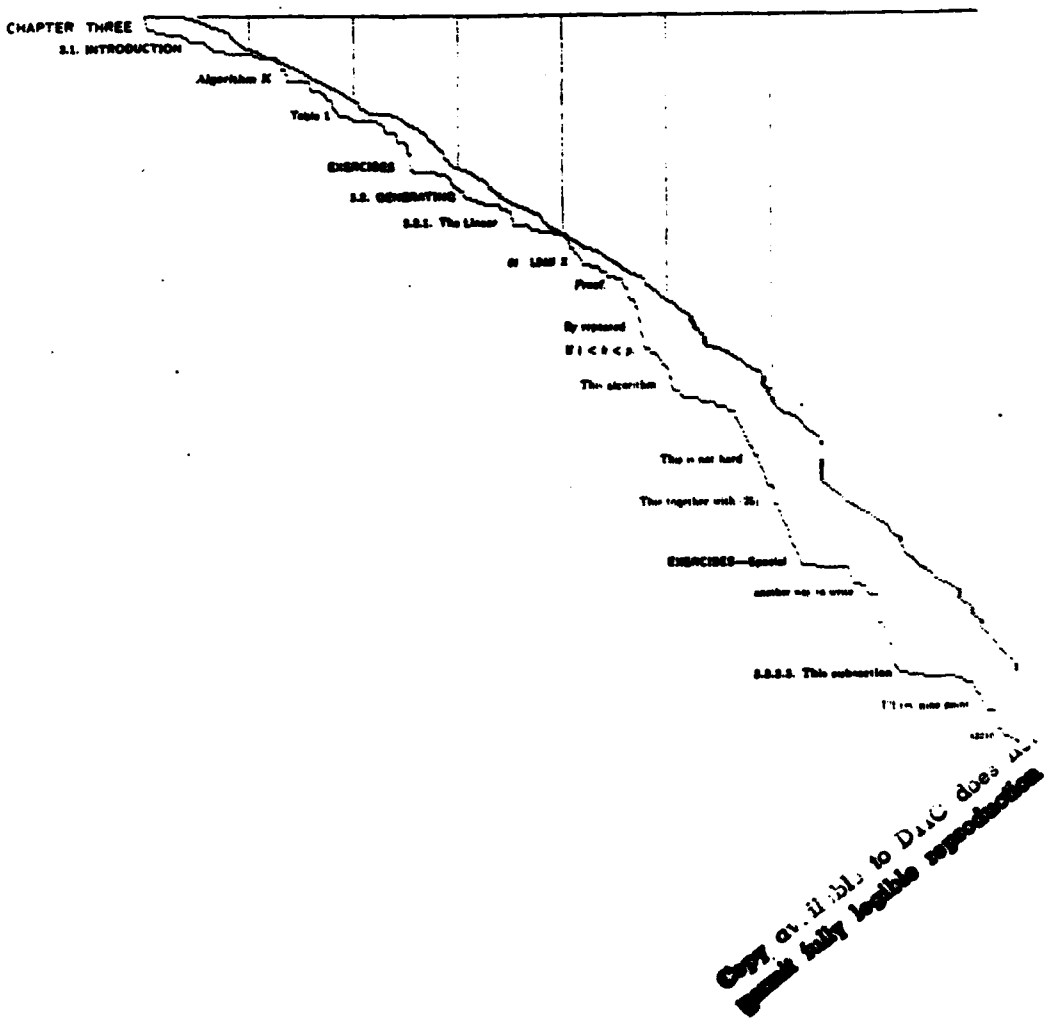
CHAPTER THREE
3.1. INTRODUCTION
Algorithm K
Table 1
EXERCISES
3.2. GENERATING
3.2.1. The Linear
Proof.
By repeated
The algorithm
The n not hard
EXERCISES—Special
3.2.2.2. This subsection
Copy available to DTIC does not permit fully legible reproduction

Figure 2