Sequentiality and Prefetching in Database Systems

ALAN JAY SMITH
University of California-Berkeley

Sequentiality of access is an inherent characteristic of many database systems. We use this observation to develop an algorithm which selectively prefetches data blocks ahead of the point of reference. The number of blocks prefetched is chosen by using the empirical run length distribution and conditioning on the observed number of sequential block references immediately preceding reference to the current block. The optimal number of blocks to prefetch is estimated as a function of a number of "costs," including the cost of accessing a block not resident in the buffer (a miss), the cost of fetching additional data blocks at fault times, and the cost of fetching blocks that are never referenced. We estimate this latter cost, described as memory pollution, in two ways. We consider the treatment (in the replacement algorithm) of prefetched blocks, whether they are treated as referenced or not, and find that it makes very little difference. Trace data taken from an operational IMS database system is analyzed and the results are presented. We show how to determine optimal block sizes. We find that anticipatory fetching of data can lead to significant improvements in system operation.

Key Words and Phrases: prefetching, database systems, paging, buffer management, sequentiality, dynamic programming, IMS
CR Categories: 3.73, 3.74, 4.33, 4.34

1. INTRODUCTION

Database systems are designed to insert, delete, retrieve, update, and analyze information stored in the database. Since every query or modification to the database will access at least one target datum, and since the analysis associated with a query is usually trivial, the efficiency of most database systems is heavily dependent on the number of I/O operations actually required to query or modify and the overhead associated with each operation.

One method used in some systems to reduce the frequency of I/O operations is to maintain in a main memory buffer pool a number of the blocks of the database. Data accesses satisfied by blocks found in this buffer will take place

The author was supported during the bulk of this research as a visitor at the IBM San Jose Research Laboratory. Partial support was later provided by the National Science Foundation under Grant MCS-75-06768. Some computer time was furnished by the Energy Research and Development Administration under Contract E(04-3)515.

Author's address: Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California-Berkeley, Berkeley, CA 94720.

much more quickly and usually with much less computational overhead. IMS (Information Management System/360) [10–12] uses this strategy, as we discuss later. The INGRES database system [9] makes use of the I/O buffer pool maintained by UNIX [19] for this same purpose. There is considerable scope for optimizing the operation of the buffer pool, principally by controlling the selection of those blocks to enter into the buffer pool and those to be removed from the buffer pool. In this paper we will primarily be concerned with sequential prefetching as an algorithm for the selection of blocks for the buffer; we shall also look briefly at other aspects of the selection and replacement problems.

A. Sequentiality

A characteristic of the database system that we study in this paper, and we believe a characteristic of many other systems, is sequentiality of access. Many queries require scans of an entire database in order to compute an aggregate. An example might be: "Find the average salary of all residents of New York City," which would probably be calculated by scanning sequentially over the appropriate section of the database. In some systems a simple lookup evokes a sequential scan of the entire database, if the database is not indexed on the key, or a partial scan, if the index is not precise. For example, a query about "John Jones" might be answered by using an index to find the beginning of the records for names starting with "J," and then searching forward. "Range" queries, in which the database is searched for records with keys in a range, will also result in sequential accesses, either to the records themselves or at least to an index if the index exists for that field of the record. Implicit in this discussion of sequentiality is the assumption that the logical sequentiality of access is reflected in physically sequential accesses to the stored data. If the physical storage organization of the data differs significantly from the logical one, then little sequentiality can be expected.

A consistently sequential pattern of access will allow us to easily anticipate which data blocks, segments, or tuples are likely to be accessed next and to fetch them before their use is required or requested. In systems in which anticipatory data fetching is less costly than demand fetching, we can expect a decrease in the cost of I/O activity. It is obvious, and under certain conditions has been shown formally [1], that when multiple block or anticipatory fetching is no "cheaper" per block than demand fetching, demand fetching is an optimal policy. We contend, and we shall discuss this in greater detail in a later section of this paper, that fetching segments or blocks in advance of their use, and in particular fetching several segments or blocks at once, is significantly less costly per block fetched (with our cost functions) than individual demand fetching.

B. Previous Research

The prefetching of data blocks into a database system main memory buffer is very similar to the prefetching of program address space (instructions/data) in a virtual memory system. Simple sequential prefetching of pages has generally been found to be ineffective [2, 13, 24], but more sophisticated methods which either analyze the program in advance, accept user advice, or maintain relevant statistics during program execution can significantly improve system operation

[2, 8, 25–28]. Sequential prefetching of lines for cache memory has been shown to work very well [24] because the extent of sequentiality is large compared to the cache page (line) size; for most programs the sequentiality is not large compared to the main memory page size. Prefetching of I/O data streams has also worked well [22, 23] because of the frequent use of purely sequential files.

C. Information Management System/360

The previous research regarding prefetching which is discussed above has all been oriented toward the analysis and use of data describing real system operation. We shall do likewise, and in later sections of this paper we present data taken from an operational and long running Information Management System/360 (IMS), a data management system marketed by the IBM Corporation. This particular system has been in operation for a number of years in an industrial environment. The results observed are believed to be illustrative of an actual application, but they cannot be taken as typical or representative for any other IMS system. Despite this disclaimer, and the fact that our research is based on data taken from a specific installation running a specific database system, we believe that sequentiality is a characteristic common to many database systems, and that therefore the methodology developed in this paper for sequential prefetching will also be applicable to other IMS installations and to other database systems.

In order to provide some background for the analysis of our data, we describe below some of the significant features of IMS. A readable and more complete discussion can be found in a book by Date [5], which also discusses other database systems and organizations. The IMS manuals [10–12] provide much more thorough coverage.

IMS is a hierarchical database system; that is, segments (tuples) in a database are arranged logically in a tree structure such as that indicated in Figure 1. An IMS implementation may consist of several databases, each of which consists of

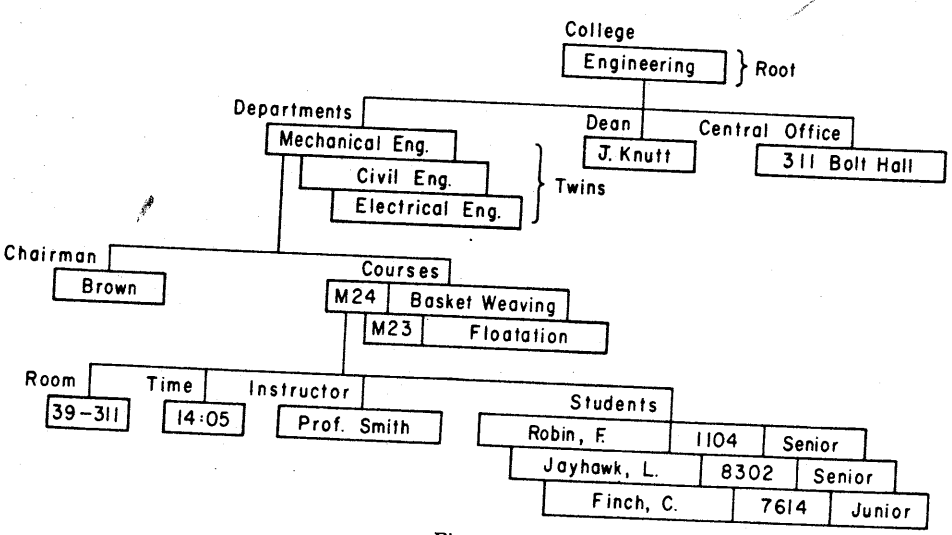

Fig. 1

a large number of such "trees." The roots may or may not be sorted and may or may not be indexed. In the system measured there were several databases; each had the roots indexed. Access to units within a tree is through the root. Within the tree, data is stored in a top-down, left-to-right order. That is, the following items from Figure 1 would be stored in this order: Engineering, Mechanical Engineering, Brown, M24 Basket Weaving, 39-311, 14:05, Prof. Smith, Finch C 7614 Junior, Jayhawk L 8302 Senior, Robin F 1104 Senior, M23 Floatation, ..., and so on.

Two physical storage organizations are possible, hierarchical sequential and hierarchical direct. In the former case segments are examined consecutively in the order in which they are stored in order to find some position in the tree. In the latter case, which describes the databases in this system, certain segment instances are related by two-way pointers and these pointers can be used to skip over intervening data. In either case the data is accessed using DL/1 (Data Language/1), which implements nine operations. These are: GET UNIQUE (find a specific item), GET NEXT (get the next item), GET NEXT WITHIN PARENT (get next item under the same root), GET HOLD UNIQUE, GET HOLD NEXT, GET HOLD WITHIN PARENT, INSERT, REPLACE, and DELETE. In each of the first six cases, the command can or must be qualified to partially or completely identify the tuple in question. GET HOLD (items 4, 5, and 6) is usually user prior to INSERT, DELETE, and REPLACE. Eighty-one percent of all accesses measured (see below) proved to be for lookup only (GET) rather than modification (INSERT, REPLACE, DELETE). Average frequency for INSERT, REPLACE, and DELETE was 11 percent, 0.7 percent, and 8 percent, respectively, with substantial variation between databases.

A search for a uniquely qualified item, such as GET UNIQUE (College(Engineering).Department(Mechanical).Chairman) will involve a search starting at the appropriate root and either scanning in sequential storage order if the storage organization is sequential, or following pointers if the organization is direct. Each of the segments referenced in the course of finding the target segment is called a *path segment.* A "GET NEXT" will also involve either sequential search or search following pointers. Unless the pointers lead immediately to the desired item, a significant amount of sequentiality will be evident in both cases; in following a path to the target the direction of search is always forward in the database.

A segment can be referenced only if it is in memory. For several efficiency reasons, IMS groups segments into fixed size storage units called *physical blocks* (henceforth called *blocks*). In the particular implementation in question the block size was 1690 bytes for data blocks and 3465 bytes for blocks used to hold index entries for the root nodes. The blocks are the unit of physical storage and data transmission; a request for a segment will result in the block containing the segment being transmitted, if necessary, from secondary storage to a *buffer pool* of blocks kept in main memory.

The search for a target segment proceeds as follows:

i. Determine the first path segment to be examined.

ii. Search the buffer pool for the block containing the segment. If the block is missing, then fetch the block (and remove a block if necessary to make room).

iii. Find the segment within the block. If this is the target segment, then perform the desired operation. Otherwise, determine the next path segment and continue at step 2.

We note that segments are commonly much smaller than blocks; for our data the average segment size was 80 bytes. A large block size will result in many segments being fetched at once; because of the sequential nature of segment access, this often means that one block fetch will satisfy many segment requests.

D. The Data

The experimental data discussed in this paper was obtained as the result of a two-step process. From the source IMS installation a trace of each DL/1 call issued over a period of a week was made. The key portion of the database was unloaded to tape, and the entire database system was reloaded at the IBM San Jose Research Laboratory. The DL/1 calls were then run against the copy of the database and a record of the target and path segments and block references was made. No effort was made to tune the copied system; the original database design was used without change, except that the reloading ensured that the logical and physical database organizations coincided.

The database system used was IMS/360 version 2.4, running under OS/VS2, release 1.6. The total size of the entire database was about 200 megabytes.

Our data analysis, in later sections, uses three sections of the entire seven-day block reference trace. The part labeled "full tape" is the trace for the first day. This first-day tape actually was generated in seven sections of approximately equal size; the parts of the trace referred to as "part 1" and "part 2" are just the first two segments of this day-long trace.

This data was gathered by researchers at the IBM San Jose Research Laboratory, and further discussions of IMS, the data gathering methodology, and the data analysis may be found in a number of papers. The reader is referred to papers by Tuel and Rodriguez-Rosell [30], Rodriguez-Rosell and Hildebrand [21], Ragaz and Rodriguez-Rosell [18], Rodriguez-Rosell [20], Gaver, Lavenberg, and Price [7], Lavenberg and Shedler [14], Lewis and Shedler [16], Lavenberg and Shedler [15], and Tuel [29] for further information.

2. OPTIMIZATION OF PREFETCHING

A. The Model

As was discussed in the beginning of this paper, the efficiency of operation of a database system can be increased by increasing the fraction of data references captured by the main memory buffer. The only general method known by the author to be effective in increasing the buffer hit ratio (for a fixed size buffer) is to take advantage of sequentiality in the series of accesses—either by prefetching or by increasing the block size. This issue is discussed further in the data analysis section of this paper; for the formulation of our optimization results we rely on a *model* as follows:

i. The physical layout of the database is the linear projection (as defined earlier) of the logical structure of the database. Segments in the database will be assumed to be numbered consecutively from 1 to S. Blocks in the database

(physical groupings of adjacent segments) are numbered consecutively from 1 to B.

ii. The reduced block reference string is the concatenation of a sequence of *runs*, each run having a length that is independently and identically distributed with probability mass function $l(k)$, $k = 1, ..., \infty$.

The *reduced block reference string* (RBRS) is the sequence of block references from which all immediate rereferences have been deleted. Thus if the original block reference string was $b_i, b_i, b_j, b_j, b_j, b_i, b_t, b_g, b_g, ...$, then the RBRS would be $b_i, b_j, b_i, b_t, b_g, ...$ It isn't known whether knowledge of immediate rereferencing provides useful information for either prefetching or replacement, but we choose not to use that information here.

A *run* is a sequence of reduced block references, such that the blocks are numbered consecutively. More specifically, let $b_j, b_{j+1}, ..., b_{j+k}, b_{j+k+1}$ be an RBRS, and let $b_{j+1} \neq b_j + 1$, $b_{j+k+1} \neq b_{j+k} + 1$, and $b_i + 1 = b_{i+1}$ for $j < i < j + k$. Then the RBRS sequence $b_{j+1}, ..., b_{j+k}$ is defined to be a *run of length k*. We define the *predecessor* of block b_i to be block $b_i - 1$ and the *successor* to be block $b_i + 1$. From our definition, it may be seen that a block which is not preceded in the reference string by its predecessor nor succeeded by its successor constitutes a run of length 1.[1]

We denote the probability distribution for the run length as follows:

$l(k)$ = the probability mass function for the run length. $l(k)$ is the probability that a run is exactly k blocks long.

$L(k)$ = the cumulative mass function for the run length = $\sum_{i=1}^{k} l(i)$.

$L^c(k)$ = the survivor function for the run length = $1 - L(k)$.

$L^c(j|k)$ = the survivor function, given that the run is of length k or greater, equal to $L^c(j)/L^c(k-1)$, $j \geq k$; equal to 1 otherwise.

$H(k)$ = the hazard rate, equal to $l(k)/L^c(k-1)$, the probability that the run ends with the kth element given that it is at least k blocks long.

$\bar{L}$ = mean run length = $\sum_{i=1}^{\infty} il(i)$.

B. The Costs of Sequential Prefetching

Our probabilistic model for the reference string as a concatenation of runs of independent and identically distributed length suggests that we condition on the previously observed run length to decide how many blocks ahead to prefetch. To do so, we must determine the utility of successful prefetches and the problems associated with unsuccessful ones. We therefore identify several *costs* in executing I/O operations: CPU time is necessary to initiate I/O operations and to update all appropriate tables. The transfer of data requires memory cycles and thus interferes with CPU operation. Transfer time, and often seek and latency time, depending on the secondary storage device, prevent data from becoming imme-

[1] The function of our definition of a run is to allow us to estimate the utility of prefetching by assuming that blocks are only used in runs. In fact, blocks may be referenced out of order, and a prefetched block might be used, but not as part of the run for which it was fetched. Thus we could define a *generalized run* such that block b_j is the kth element of a generalized run if the predecessor of block b_j is in memory at the time block b_j is referenced and if block $b_j - 1$ is the $(k-1)$-st element of a generalized run. Such a generalization is beyond the scope of this study, both because of its complexity and because of its dependence on the size of the memory.

diately available. The CPU in this case must either remain idle while the information is fetched, or it must switch to another task, which again involves overhead. Every time a block is fetched it may displace another block that is about to be rereferenced. Thus, excessive fetching of unused blocks may actually increase the miss ratio.

We shall group all of the above costs into three *strategy costs* and one *effect cost.* Possible strategies, which involve strategy costs, are the following: (a) Fetch a block on demand (as needed), (b) fetch a block in advance but not simultaneously with a demand fetch, and (c) fetch additional sequentially adjacent blocks (*"tag-along"* blocks) when a block is fetched according to strategy (a) or (b). We identify these costs as follows:

DFC: Demand Fetch Cost—the cost of fetching one block when it is needed immediately.

PFC: Pre-Fetch Cost—the cost of fetching one block when it is not needed immediately and when it is not a "tag-along" block.

TAC: Tag-Along Cost—the cost for each additional block transferred from secondary storage when the initial block was fetched by strategy (a) or (b).

We also identify the following effect cost:

BFC: Bad Fetch Cost—the cost, in additional fetches, resulting from bringing in a block that is not actually used. This latter effect is also known as *memory pollution.*

In determining and evaluating our cost function, we will make the following assumptions in addition to those of our model:

iii. *DFC, PFC, TAC,* and *BFC* include all important costs, and further, the total cost can be expressed as a linear combination of these individual costs.

iv. Blocks that are prefetched and used while still in memory do not contribute to memory pollution.

v. The effect of memory pollution is independent of the block replacement algorithm and the memory capacity.

vi. There is nothing other than run length that contributes to effective prediction.

We will discuss the validity of some of these assumptions in later sections of this paper.

Let us assume some strategy for prefetching. Associated with this strategy is a set of numbers $\mathbf{E}(DF)$, $\mathbf{E}(PF)$, $\mathbf{E}(TA)$, and $\mathbf{E}(BF)$, where these are respectively the expected number of demand fetches, prefetches, tag-along blocks, and bad fetches per run. The *expected cost per run C* under this strategy is then

$$C = DFC^{*}\mathbf{E}(DF) + PFC^{*}\mathbf{E}(PF) + TAC^{*}\mathbf{E}(TA) + BFC^{*}\mathbf{E}(BF). \qquad (1)$$

From our assumptions above, specifically from ii and vi, to optimize system operation it is sufficient to minimize the expected cost per run by choosing the best prefetching strategy. We find that strategy below.

C. Prefetch Strategy Optimization

We shall consider two classes of prefetch policies; the second will be a generalization of the first:

Class 1 *prefetch policies* permit any number j additional sequential blocks to be prefetched at a time when a demand fetch is required. No fetches are permitted at other times. In many systems, *PFC* is comparable to *DFC*, so this restriction follows naturally.

Class 2 *prefetch policies* permit any number j additional sequential blocks to be prefetched beyond the current reference at any time, whether or not a miss occurred on this reference. From our set of costs, it is clear that there is no advantage to prefetching if at the time of a reference to block b_k, block $b_k + 1$ is already in memory. Therefore it is assumed that prefetches are initiated only when the current reference causes a fault or the block beyond the current reference is not in the buffer.

1. Class 1 Prefetch Optimization. We define the *remaining cost* for a run $C(k)$, and the *minimum expected remaining cost* for a run $C_{min}(k)$, as follows: The remaining cost $C(k)$ for a run is the expected cost of fetching all remaining blocks of the current run, given that (exactly) $k - 1$ blocks have already been fetched and that the run is k or more blocks in total length. The minimum expected remaining cost is taken over all possible class 1 fetch policies.

The value of $C_{min}(k)$ may be simply computed: It is the cost of fetching the kth block in the run (*DFC*) plus the cost of j additional blocks (j**TAC*, $j \geq 0$), plus the cost of the remainder of the run ($L^c(k + j|k)$*($C_{min}(k + j + 1)$)), plus the pollution cost ($BFC^*\sum_{i=0}^{j-1} l(k + i)(j - i)/L^c(k - 1)$). This may be seen by inspection (when the terms are collected) to be:

$$C_{min}(k) = DFC + \min_{\substack{j \\ j \geq 0}} \left\{ j^*TAC + L^c(k + j|k)^*C_{min}(k + j + 1) + \mathrm{BFC}^* \sum_{i=0}^{j-1} l(k + i)(j - i)/L^c(k - 1) \right\}. \quad (2)$$

The mean minimum cost per run is then $C_{min}(1)$, and the mean cost per (reduced) reference is $C_{min}(1)/\bar{L}$. (The cost per actual reference can be obtained by dividing the cost per reduced reference by the ratio of real references to reduced references.)

If the run length is bounded, C_{min} may be computed straightforwardly: Let k_{max} be the largest possible run length. Then $C_{min}(k_{max}) = DFC$. $C_{min}(k)$, $k < k_{max}$, is a function of $C_{min}(j)$, $k_{max} \geq j > k$, and other known parameters only, and the computation is thus simple and proceeds from $k = k_{max}$ to $k = 0$. Since we shall estimate the run length distribution L(k) from the observed sample $\hat{L}(k)$, the run length is always bounded. We hypothesize that for "well-behaved" distributions $L(k)$, arbitrary truncation of $L(k)$ produces no ill effects. Since $L^c(k + j|k)$, $j \geq 1$, is always less than one, the effect of errors in $C_{min}(j + k + 1)$ can be seen to die out.

The computation of C_{min} also provides the optimal fetch policy. At the time of a fault to the kth block in the run, it is optimal to fetch $\alpha(k)$ additional blocks, where $\alpha(k)$ is the value of j that minimizes the expression above in eq. (2). We note that after this computation has been performed, many values of $C_{min}(k)$ and $\alpha(k)$ are superfluous since there will never be a demand fetch to the kth block in the run. For example, let the optimum values of α be respectively $\alpha = \{1, 1, 3, 3,$

2, 4, 4} for $k = 1$ to 7. Then a fetch to the first block in a run will result in its successor being fetched. Therefore, there can never be a fault to the second item in a run. Similarly, the fault to the third block in a run fetches also the fourth, fifth, and sixth. Thus it is sufficient to specify the fetch policy only at the possible fault points.

2. Class 2 Fetch Policies. In the case that fetching at other than fault times is permitted, we must define a slightly different cost function. Let $Cf_{min}(k)$ be the minimum *remaining future costs* of a run, where this is defined to be the cost of accessing all remaining blocks of the run, given that the run is k or more blocks in length and that exactly k blocks have already been fetched. (Note the difference from the preceding definition.) We are led to this different definition because at the time that we reference block k in the run, we must decide whether to bring in block $k + 1$ at this time, or to wait to see if it is referenced first. The cost breaks down into two parts, depending on this decision, and may be seen to be:

$$Cf_{min}(k) = \min \left\{ \left[PFC + \min_{j \geq 0} \left\{ j*TAC + L^c(j + k|k) Cf_{min}(k + j + 1) + BFC* \sum_{i=0}^{j} l(k + i)(j + 1 - i)/L^c(k - 1) \right\} \right], \left[L^c(k|k) \left(DFC + \min_{j \geq 0} \left\{ j*TAC + L^c(k + j|k + 1)*Cf_{min}(k + j + 1) + BFC* \sum_{i=0}^{j-1} l(k + i + 1)(j - i)/L^c(k) \right\} \right) \right] \right\}. \quad (3)$$

Again, we obtain the optimal fetch policy as a consequence of our computation of Cf_{min}. Let $\beta(k)$ be the value of j that minimizes the first term in eq. (3) above, and let $\gamma(k)$ be the value of j that minimizes the second term. If the first term is less, it is optimal to fetch a number of additional blocks ($\beta(k) + 1$ blocks) even though it is not a fault time; if the second term is less, it is optimal to wait until the $(k + 1)$-st block is referenced (if it is), at which time $\gamma(k)$ additional blocks should be fetched.

The mean minimum cost per run is $DFC + Cf_{min}(1)$, and the mean minimum cost per (reduced) reference is $(DFC + Cf_{min}(1))/\bar{L}$.

3. COST ESTIMATION

A. Estimation of *DFC*, *PFC*, and *TAC*

In Section 2C we described an algorithm for computing the cost to bring into the memory buffer the blocks in one run as a function of four parameters, *DFC* (demand fetch cost), *PFC* (prefetch cost), *TAC* (tag-along cost), and *BFC* (bad fetch cost), values for which have not yet been assigned. In this section we shall discuss the choice of values for these parameters.

We will arbitrarily assign *DFC* a value of 1.0. All other costs will be assigned relative to this fixed value. Since our cost structure uses a relative and not absolute scale, we have lost no generality.

The values assigned to *PFC* and *TAC* (relative to *DFC*) are a function of the architecture of the I/O devices, the load on the system, the quality of the system implementation, and the degree of multiprogramming, among other factors. The costs for a prefetch include the CPU time involved in constructing and initiating the channel program; the CPU idle time induced by the period during which the channel, controller, and device are unavailable; the memory interference produced by the transfer; and the possible delay in completing the fetch should the prefetched block be needed quickly. The tag-along cost is almost the same as the above, but includes the marginal additional cost for the channel program per block transferred rather than the bulk of the overhead. Each of the components of these costs will vary among machines and operating systems, with time dependent factors such as the load on the system, and with the system configuration (number and type of I/O devices). We will therefore discuss some different scenarios and some cost functions which might follow. Specific examples of cost functions will be chosen and calculations presented in Section 4, where we discuss the data we have analyzed. We note that the only way to determine a set of verifiable values for these costs is to vary the prefetching algorithm on a specific system driven repeatedly by the same benchmark and to estimate the parameters statistically from the observed behavior.

The simplest case is one in which the system is completely I/O bound; that is, the system is assumed to always be waiting for an I/O operation to complete. Consider quarter track blocks and random seeks; then the mean transfer, latency, and seek times for the IBM 2314 disk are respectively 25/4 msec, 25/2 msec, and 60 msec. For the 3330 disk, the values are 4.2 msec, 8.4 msec, and 30 msec. (In fact, random seeks are unlikely (see Smith [22, 23] for a relevant discussion), so the mean seek time should probably be less than given.) The marginal wait for a tag-along block is then $(25/4)/(60 + 25/2 + 25/4)$ relative to the demand fetch wait for the 2314, or 0.079 (0.099 for the 3330). For ½ track data sets, the value would be 0.147 and 0.1721 for the two disks. Adding in the costs of the other overheads mentioned above, a value of 0.2 for *TAC* seems reasonable. The value of *PFC* is higher, since it includes the overhead for performing the I/O and may include more nonoverlapped transfer time. *PFC* is almost certainly lower than *DFC* since some of the I/O wait time is assumed to be overlapped with the processing of the remaining block in the run. A value of 0.7 seems to be a plausible value for *PFC*.

An alternative assumption for system operation is that the CPU is fully utilized and that the demands on the CPU limit the throughput of the system. This would be reasonable in a system with many storage (I/O) devices and a great deal of memory, so that all I/O could easily be overlapped with computation. In this case the values assumed by *DFC*, *PFC*, and *TAC* depend on the exact time it takes to execute the appropriate code (I/O initiation, task switching, if any, etc.). It has been suggested to the author that values of 0.2 and 0.9 for *TAC* and *PFC* seem reasonable.

For other cases, intermediate between I/O bound and CPU bound, the values assigned to the cost variables will vary. It is to be hoped that because of the uncertainty in the cost variables, the cost function will be relatively flat for varying degrees of prefetching. Computations performed by the author indicate that this is the case.

B. Estimation of *BFC*

It remains to estimate a value for *BFC*, the additional cost for fetching a block that is never used. This additional cost arises in the case that a block is removed from the buffer pool in order to make room for the new fetch and that this old block is quickly rereferenced. We assume that every block that is expelled from the buffer and then reused is fetched at a cost of 1.0; we now estimate the probability that a block so expelled is reused.

Before we describe two methods of calculating the value of *BFC*, we define a specific paging algorithm. In MULTICS [4] and CP-67 [3], a paging algorithm commonly referred to as the "clock" paging algorithm is used. We define here the:

GENERALIZED CLOCK PAGE REPLACEMENT ALGORITHM. With each page frame in memory we associate a count field and we arrange these count fields in a circular list. Whenever a page is referenced, the associated count field is set to i. When a page fault occurs, a pointer that circles around this circular list of page frames is observed. If the count field pointed to is zero, then the page is removed and the new page is placed in that frame. Otherwise, the count is decremented by 1, the pointer is advanced to the next count field, and the process is repeated. When a new page is placed in the page frame, the count field is set to i if the page is to be referenced (demand fetch) and it is set to j if the page has been prepaged and is not immediately referenced. We abbreviate this algorithm by writing CLOCKP(j, i). The "P" indicates that this is a prepaging algorithm (the prepaging strategy has not been specified). When no prepaging is involved, the algorithm is abbreviated CLOCK(i). The algorithm used in MULTICS and CP-67 is CLOCK(1).

We have chosen to define this algorithm in such a manner because the existence of the parameters i and j will facilitate certain experiments to be described.

We will also have occasion to refer to the Working Set paging algorithm [6] and the Least Recently Used (LRU) paging algorithm [17], with which we assume the reader is familiar.

We will employ two methods to estimate *BFC*; both methods are rather ad hoc. Although this parameter is of substantial importance, there does not seem to be any simple and accurate estimation method. From the experiments described later in this paper, it appears that the results are not unduly sensitive to inaccuracies in the estimate for *BFC*.

1. Estimation Using Rereference Probability. Our first estimate for BFC will be the following: The probability of removing a block that is about to be rereferenced (and thus initiating a block fault) is estimated to be the probability that a block whose count field is 0 is rereferenced before being removed. (The set of blocks eligible for replacement is the set with the count fields equal to zero.) This probability was estimated for CLOCK(1) and appears in Figure 2 for a range of memory sizes and for parts 1, 2, and the full tape. We note that this value varies widely over the range of memory sizes and over portions of the data traces. This variation suggests that the behavior of the system is not stationary. We will discuss this issue to some extent in Section 4.

2. Estimation Using Marginal Changes in Capacity. An alternative method for estimating the value of *BFC* is to use the observation that for every block

fault except those required to fill the buffer the first time, a block must be removed from the buffer. When a prefetch takes place, a block must be removed prematurely from the buffer; we choose to look at this premature replacement as a slight reduction in the effective size of the buffer.

Let $NF(A, C, T)$ be the total number of blocks fetched (using demand fetching) for a trace T using buffer capacity C (in blocks) under block replacement algorithm A. Let G be the (reduced) trace length and $M(A, C, T)$ be the miss ratio. By definition, $M(A, C, T) = NF(A, C, T)/G$. We abbreviate $M(A, C, T)$ as M and $NF(A, C, T)$ as NF. Let ΔC be the effective change in the capacity of the buffer from a single prefetch. We shall assume

$$\Delta C = C/NF. \tag{4}$$

The rationale behind eq. (4) is as follows: Consider a space-time plot of buffer capacity such as is shown in Figure 3(a), where time advances by one unit at fault times. If we do a prefetch, then the prefetched page will require approximately C fault times to be removed (e.g. for FIFO replacement), giving a net effect for the space-time plot of that shown in Figure 3(b). The unshaded part of the figure has an area equal to $(C - \Delta C)*NF$. ΔC is then found to equal the expression in eq. (4).

The change in the miss ratio (ΔM) is then equal to $\Delta M = (\partial M/\partial C) \cdot \Delta C$. The change in the number of faults ΔNF is simply $\Delta NF = \Delta M \cdot G$. So, continuing, we

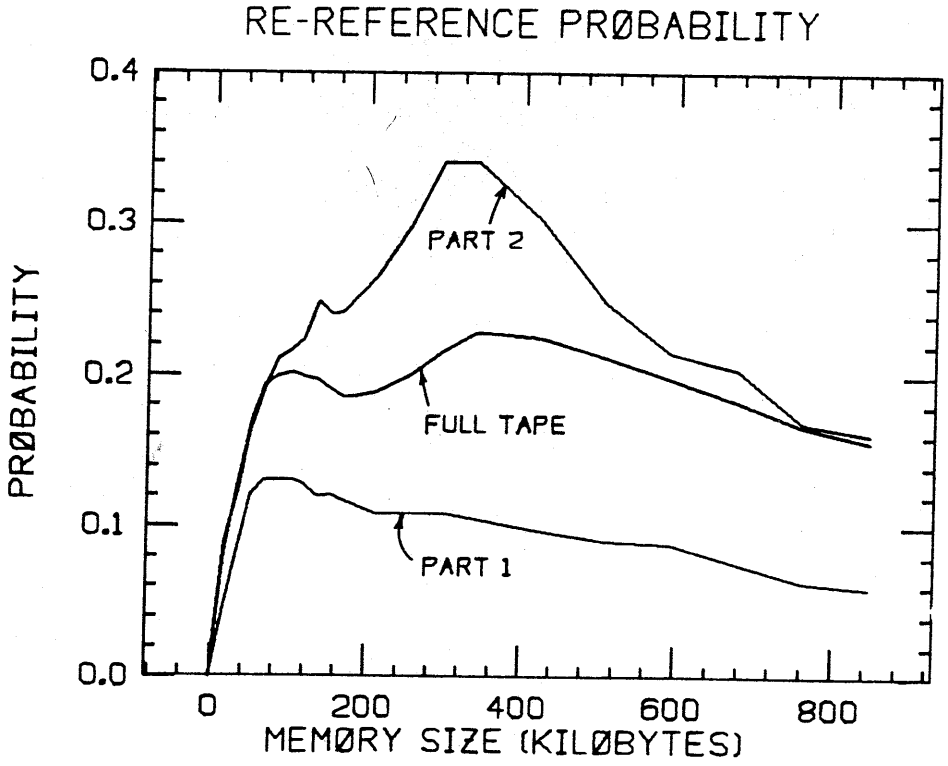

Fig. 2

have

$$\Delta NF = (\partial M/\partial C)\cdot G\cdot \Delta C = (\partial M/\partial C)\cdot G\cdot C/NF$$
$$= C\cdot(\partial M(A, C, T)/\partial C)/M(A, C, T). \quad (5)$$

In Figure 4 we present the miss ratio for the full (one-day) trace for three paging (block replacement) algorithms: CLOCK(1), LRU, and Working Set. This miss ratio is seen to be relatively insensitive to the algorithm, and it thus appears that any choice of replacement algorithm will yield approximately the same result. We note in particular that the results for LRU and CLOCK(1) are indistinguishable. Calculating the value of *BFC* that we obtain from eq. (5) yields

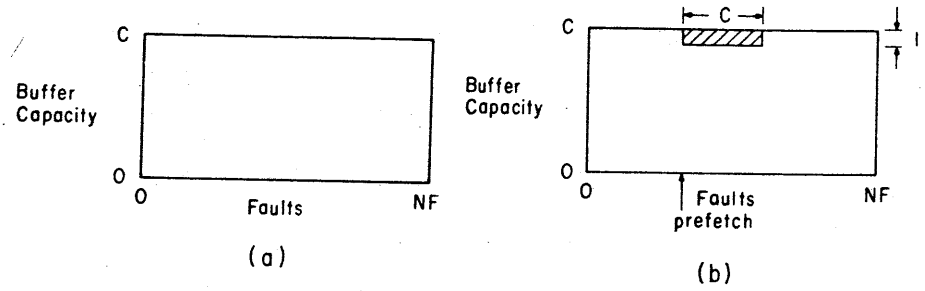

Fig. 3

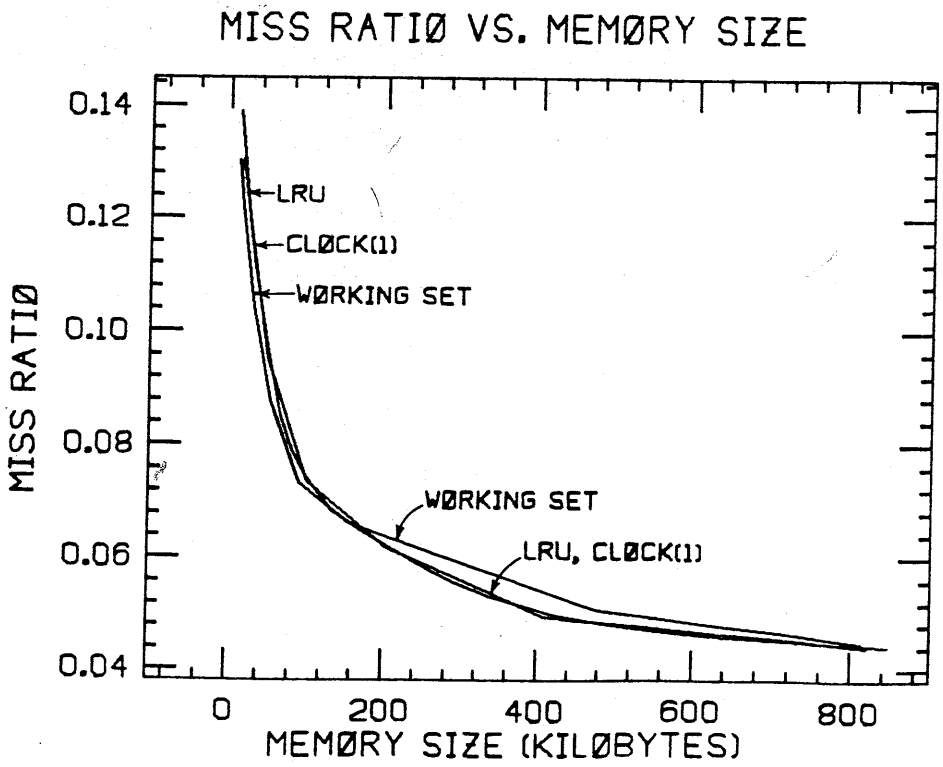

Fig. 4

values of 0.308, 0.3, and 0.21 at buffer sizes of 35K, 100K, and 500K, respectively. Since a prefetched but unused page would probably be flushed somewhat sooner than C fault times, these values might actually be a little lower. In any case these values are comparable to those obtained earlier from Figure 2. Henceforth, a value of 0.2 for *BFC* will be chosen as representative.

3. Replacement of Prefetched Blocks. There is some question as to the treatment of prefetched blocks—should they be considered for replacement as if they had been brought into the buffer by a demand fault (e.g. placed at the top of the LRU stack) or should they be given somewhat less favorable replacement status (e.g. middle or bottom of LRU stack)? We looked at this question by experimenting with a variant of the CLOCKP(j, i) algorithm. The variant we used was to always fetch a block's successor if the successor was not in memory at the time the block was referenced. We define the *miss ratio* in this case as the fraction of data references that required an immediate data fetch (the block was missing). The *prefetch ratio* is the ratio of the number of blocks prefetched to the total number of references. The *transfer ratio* is the sum of the prefetch and miss ratios. Three values of (j, i) for this CLOCKP(j, i) algorithm were used: CLOCKP(0, 1) CLOCKP(1, 1), and CLOCKP(0, 3). The first places the prepaged blocks in the middle of the "stack," the second places them at the top of the stack, and the third near the bottom. It may be seen from Figure 5 that the lower the prefetched blocks are placed in the stack, the lower the miss ratio and the

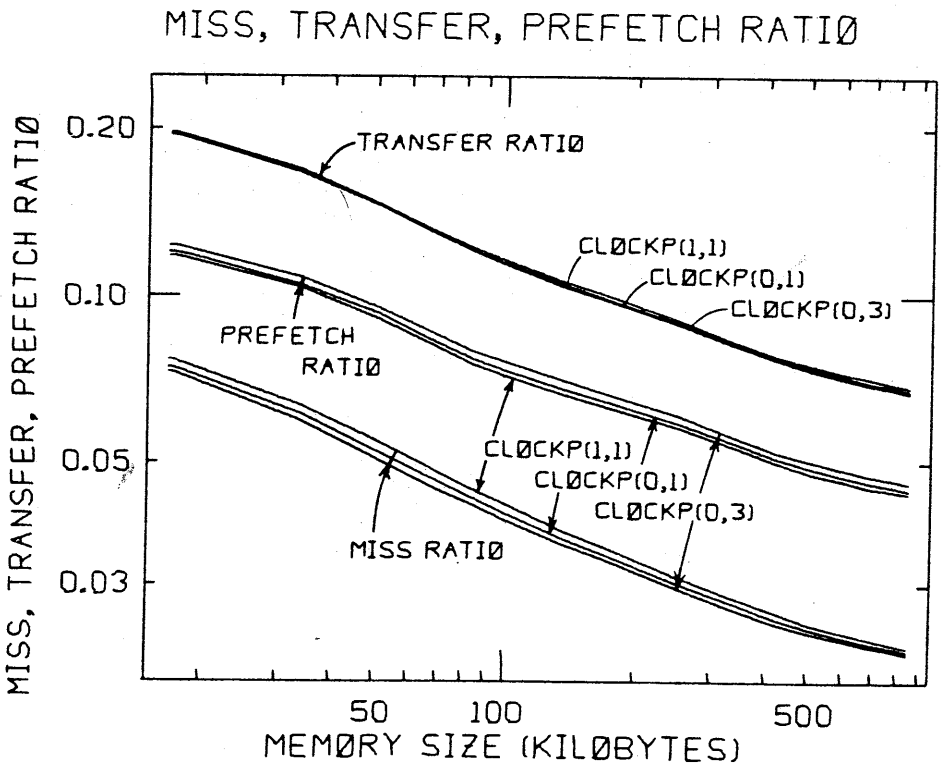

Fig. 5

higher the prefetch ratio. The transfer ratio, as the sum, is somewhat more stable. The effect of placing a prefetched block at the actual bottom of the stack has not been investigated here; the reader is referred to [18] for a further discussion of this question. In any case the effect of stack placement seems to be minor and will not be considered further.

4. DATA ANALYSIS AND OPTIMIZATION RESULTS

Our algorithms for cost minimization in Section 2 were derived by assuming that certain conditions held, in particular that the (reduced) reference string consisted of a sequence of runs, the length of each run being chosen independently from the stationary run length distribution. The utility of our optimization procedure, however, should not be judged by the accuracy of the model. The large sample size (there were 254,329 runs in the full one-day trace) virtually precludes statistical acceptance of such a simple model in any case, and the results presented in Figure 2 have already suggested that system behavior is not entirely stationary. To provide some insight into the effect of nonstationary behavior, separate analysis has been done on the aforementioned pieces of data: the first two trace sections and the full day's trace.[2] As will be seen, differences between the sections of the trace seem to be small (although not statistically acceptable) and the prefetching algorithms developed are similar.

A. Run Length Distribution

The empirical run length distribution, as defined in Section 2, was measured for the first two sections of the trace tape and for the full one-day tape. This distribution is plotted in Figure 6 on a log scale and is also listed in Table I. The observed maximum run length for the full tape was over 1000 blocks, but as may be seen in Figure 7, only 0.1 percent of the run lengths exceeded 20. When, however, one considers the number of blocks in long runs rather than the number of long runs themselves, the relative importance of optimizing the performance in this case becomes apparent. Further, we note that the possibilities for optimization lie largely with long runs. Thus, the low probability of finding long runs understates their importance.

Figure 7 presents the empirical survivor function ($L^c(k)$) of the run length on a logarithmic scale for the different parts of the tape. A number of observations are possible from this figure. Although many of the characteristics of the different parts of the tape are different (see Figure 2 for example), the log survivor function doesn't seem to vary greatly. It may be seen that the curve is largely along two straight lines, which suggests that should we wish to model the run length distribution, a two-part hyperexponential would be appropriate. This also suggests that we may wish to have at least two different prefetching policies—one for runs which have gone on for less than four blocks and one for runs of greater length.

[2] Further consideration of stationarity and a more sophisticated and extensive sectioning of the trace would probably have been desirable. Unfortunately, the author no longer has possession of the data. The treatment of this question is more than adequate, however, considering the difficulties inherent in such other problems as estimating the parameters.

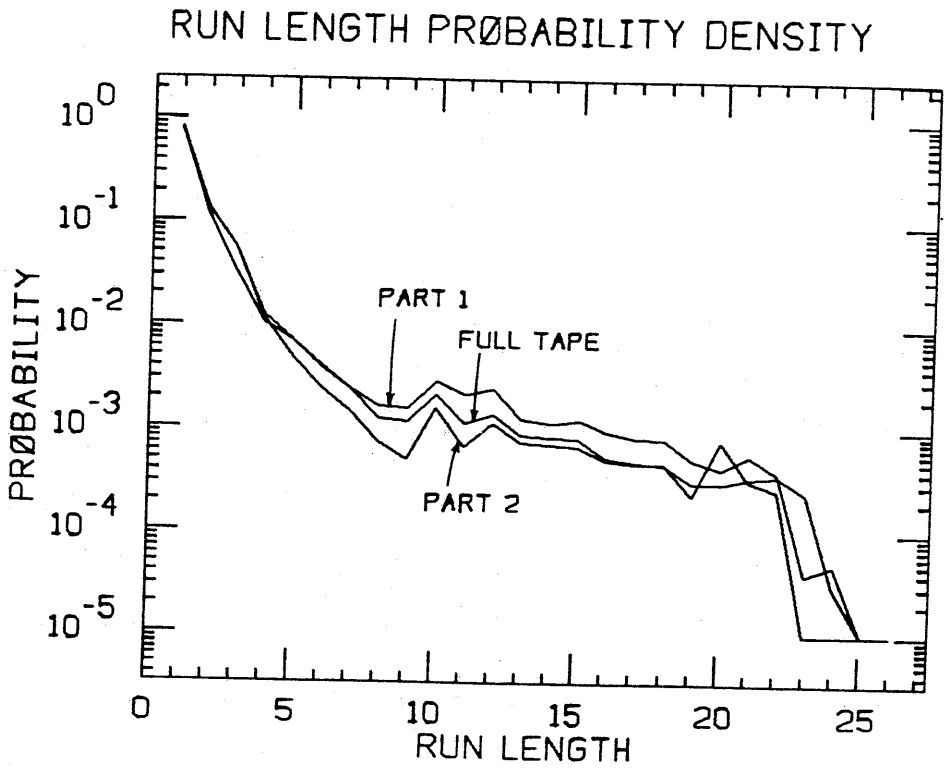

Fig. 6

Table I. Run Length Distribution

Length	Probability		
	Part 1	Part 2	Full tape
1	0.8144	0.7869	0.7774
2	0.1104	0.1281	0.1299
3	0.0319	0.0538	0.054
4	0.0103	0.0113	0.0122
5	0.0069	0.00463	0.00694
6	0.00394	0.00243	0.00379
7	0.00241	0.00147	0.00239
8	0.00163	0.00073	0.00124
9	0.00155	0.00050	0.00115

The empirical expected future run length (*efrl*(x)), that is, the expected additional run length given that the run length has already reached x, is plotted in Figure 8. We may define *efrl(x)* from the run length distribution as:

$$efrl(x) = \sum_{j=x+1}^{\infty} (j - x)l(j)/L^{c}(x).$$

We observe that *efrl*(x) is geneally increasing; that is, the longer the current run length, the longer the run is likely to continue. This suggests fetch policies that fetch progressively larger numbers of blocks as the length of the run increases.

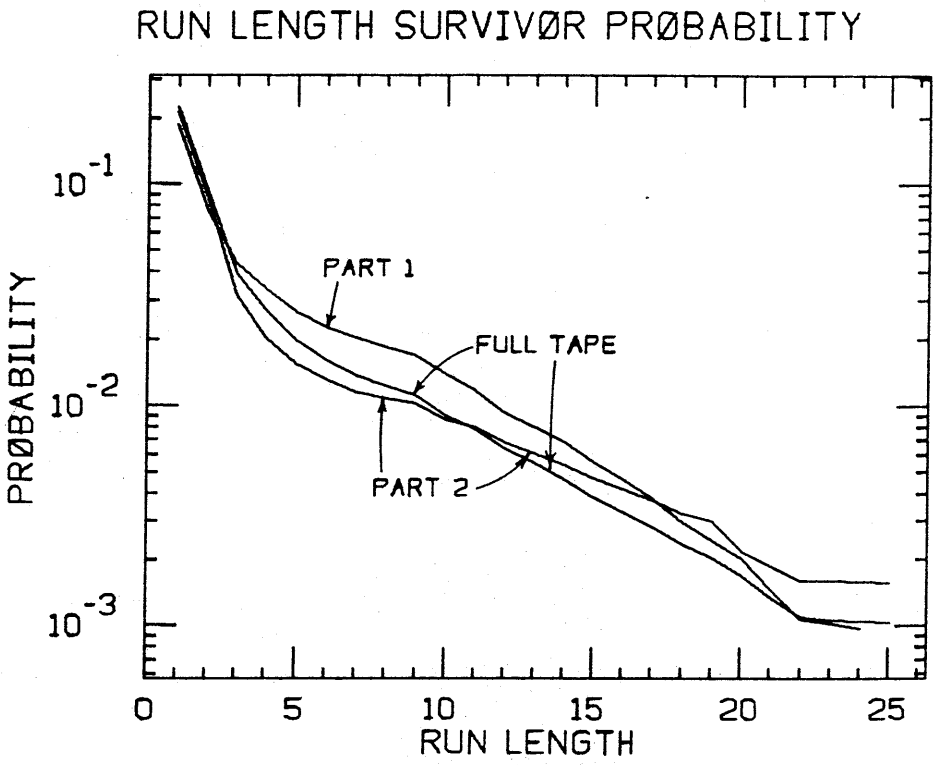

Fig. 7

The hazard rate for the distribution has been plotted in Figure 9. The hazard rate is defined as $H(k) = l(k)/L^c(k-1)$, which is simply the probability that the run ends with the kth block, given that it has reached the kth block. The hazard rate can be generally seen to be declining until about nine, after which it levels off. A high hazard rate is a good indication that prefetched blocks will not be used and conversely.

From the data presented in Figures 6–9, we can conclude that a prefetch policy that conditions on the run length would likely fetch increasingly larger numbers of blocks (up to some maximum) as the run progressed. Long runs, therefore, would not be as costly in terms of block faults as they would be without prefetching. The high hazard rate for short runs, however, indicates that always prefetching ahead a fixed number of blocks or increasing the block size would not yield the same improvement.

B. Optimized Prefetch Strategies

The optimized prefetch strategy has been calculated for a number of different cost functions using the methods of Section 2 of this paper and the empirical run length distribution. The results of these calculations appear in Tables II–V. The strategy indicated may be interpreted for a strategy on line j which is an integer k with no asterisk as follows: If the jth block in a run is referenced and the reference causes a fault, then at the same time bring in any of the k blocks that follow block j ($j+1, j+2, \ldots, j+k$) in the run if they are not already in memory.

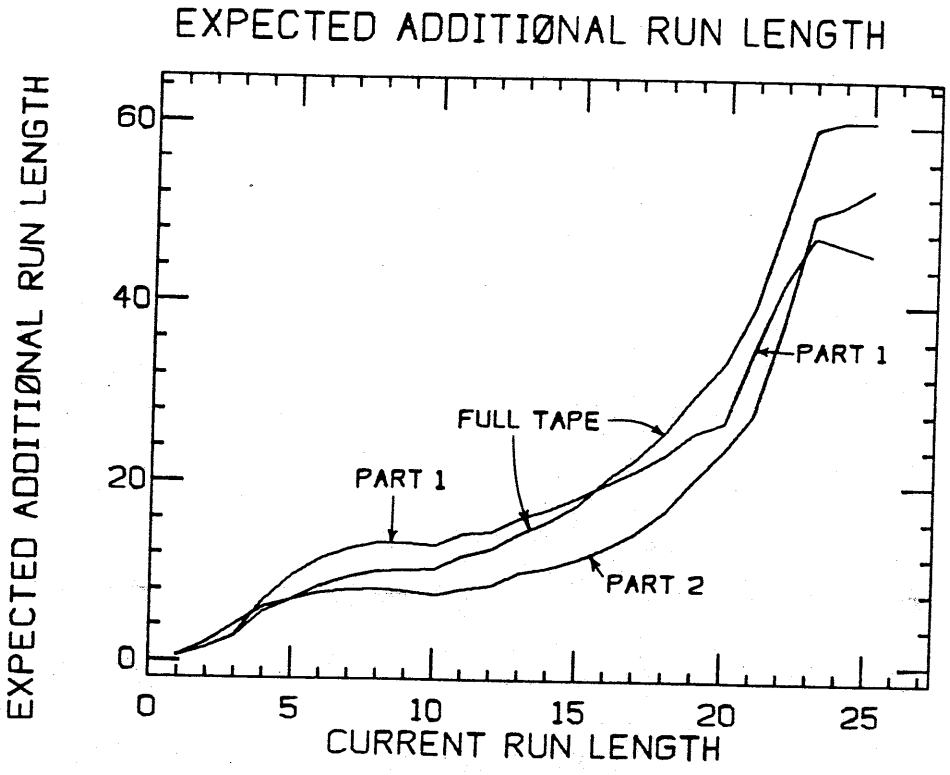

Fig. 8

If there is an asterisk (k^*) on the strategy, and if the successor of the jth block is not in memory, then bring in the next k blocks in the run (also providing they are not already in memory) whether or not a fault has occurred on this reference.

We note that although the strategies vary somewhat among the different parts of the trace, the variation is not very large, and that some very simple strategy, in line with the sample strategies indicated in Tables II–V, is likely to be adequate. We observe, conversely, that for the parameter values chosen, a fixed prefetch strategy in which every time a block is fetched, k successors are fetched as well, is not very close to optimal. In Subsection C we compare optimized with other strategies with regard to their costs and a significant (5–20 percent) cost difference will be observed.

C. Testing a Simple Strategy

Because there is some nonstationarity in the data, the optimal prefetch strategy can be seen to vary among sections of the trace. It also varies as a function of the estimates for the cost parameters, and as noted, those estimates are quite crude. Therefore, the usefulness of optimized prefetching in a real system depends not on whether an optimized strategy works well for the trace from which it is derived, but whether a simple strategy (not necessarily identical to those suggested) decided on in advance works well. We chose what we call Strategy(0, 1, 2, 3, 4): At fault times, if the run length is k, $k < 5$, prefetch $k - 1$ blocks in addition to the one needed; else prefetch 4 additional blocks. This strategy is

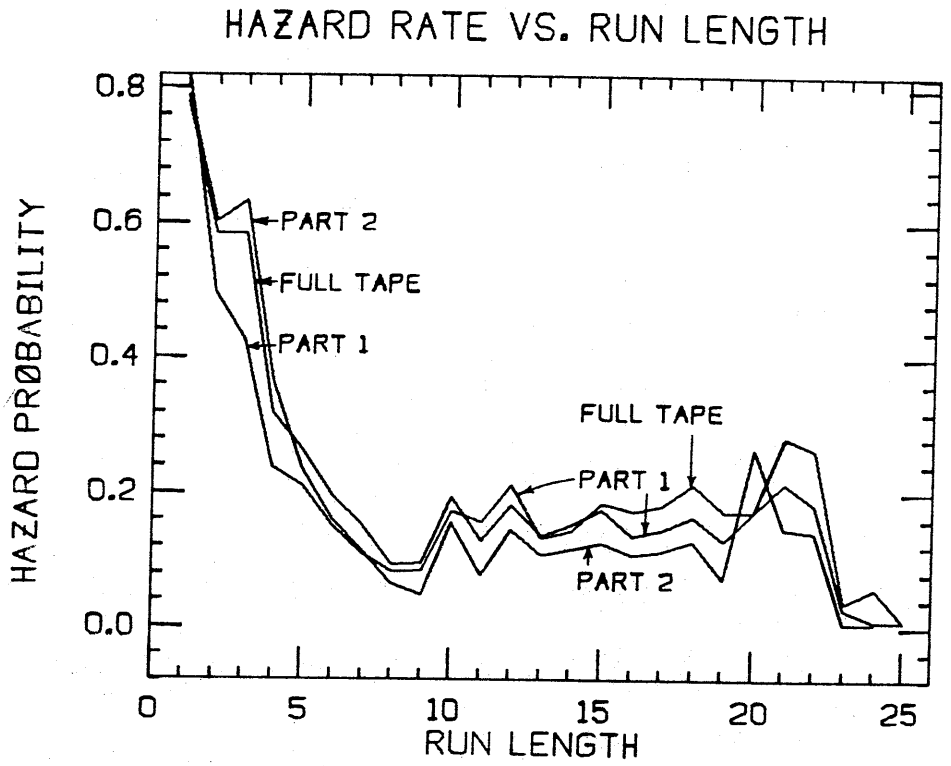

Fig. 9

Table II. Optimized Prefetch Strategy Cost: DFC = 1, PFC = 0.7, TAC = 0.4, BFC = 0.2

	Strategy		
Run length	Part 1	Part 2	Full tape
1	0	0	0
2	0	0	0
3	0	0	0
4	1	1	1
5	2	2	1
6	2	2	2
7	3	3	3
8	4*	3*	2*
9	3*	2*	1*
10	2	2	2
11	1*	3*	2*
12	0	2	1
13	2*	3*	2*
14	1*	2*	1
15	1	2*	2

Table III. Optimized Prefetch Strategy Cost: DFC = 1.0, PFC = 1.0, TAC = 0.0, BFC = 0.2

	Strategy		
Run length	Part 1	Part 2	Full tape
1	0	0	0
2	1	1	1
3	2	1	1
4	3	1	2
5	7	3	3
6	6	5	5
7	5	7	6
8	5	8	5
9	6	7	7
10	5	6	6
11	5	5	5
12	5	6	5
13	5	5	5
14	4	9	5
15	4	8	4

similar to the strategies in Tables III and IV, but is not the same as either one of them.

The result of this simulation is presented in Figure 10, where we see the prefetch ratio, miss ratio, and transfer ratio. We have also plotted the miss ratio

Table IV. Optimized Prefetch Strategy
Cost: DFC = 1, PFC = 1.0, TAC = 0.2, BFC = 0.2

	Strategy		
Run length	Part 1	Part 2	Full tape
1	0	0	0
2	1	1	1
3	1	0	1
4	2	2	2
5	2	2	2
6	4	4	4
7	5	5	3
8	4	5	4
9	3	4	3
10	3	4	3
11	3	4	3
12	3	4	3
13	3	4	4
14	4	4	4
15	3	3	3

Table V. Optimized Prefetch Strategy
Cost: DFC = 1, PFC = 0.2, TAC = 0.2, BFC = 0.2

	Strategy		
Run length	Part 1	Part 2	Full tape
1	0	0	0
2	1*	1*	1*
3	1*	1*	1*
4	1*	1*	1*
5	1*	1*	1*
6	1*	1*	1*
7	1*	1*	1*
8	1*	1*	1*
9	1*	1*	1*
10	1*	1*	1*
11	1*	1*	1*
12	1*	1*	1*
13	1*	1*	1*
14	1*	1*	1*
15	1*	1*	1*

for the CLOCK(1) algorithm for comparison. It may be seen that a substantial reduction in the miss ratio has been achieved at a very small cost in wasted block transfers.

In Figures 11 and 12 the cost of our (0, 1, 2, 3, 4) strategy is compared with strategies which either are demand ($N = 0$), or prefetch fixed numbers of blocks ahead ($N = 1$, $N = 2$, $N = 3$). Prefetch costs (TAC) of 0.2 and 0.3 have been chosen as realistic and representative. The cost (C) is then C = miss ratio + TAC*prefetch ratio. Prefetching at other than fault times has not been considered in this example, and the effect of bad fetches (BFC) is automatically included in the miss ratio. We see that our "optimal" strategy has a cost of from 5 to 10 percent less than that for $N = 1$ and $N = 2$ strategies and from 10 to 15 percent less than that for $N = 3$, and is at least 20 percent better than simple demand fetching. (Because these values are for only part of the full day trace, they are not directly comparable with Figure 10.) These differences, in the opinion of the author, are significant.

5. OPTIMIZATION OF BLOCK SIZE

The methods of the preceding sections may be used to compute the optimal block size. In Section 2 the average cost per reference was given as $C_{min}(1)/\bar{L}$ or $(DFC + Cf_{min}(1))/\bar{L}$. The most straightforward way and also the most accurate way to optimize the block size is to measure the run length distribution function $l(k, B)$ where B is the block size; to determine $PFC(B)$, $TAC(B)$, $DFC(B)$, and $BFC(B)$ where the costs and the run length distribution are now functions of the block size; and to compute $C_{min}(1, B)/\bar{L}(B)$ or $(DFC(B) + Cf_{min}(1, B))/\bar{L}(B)$. The block size yielding the minimum cost per reference is then clearly the optimum block size.

One would expect that the run length as a function of B, the block size, could be determined from the run length in bytes, but this is not entirely correct. A

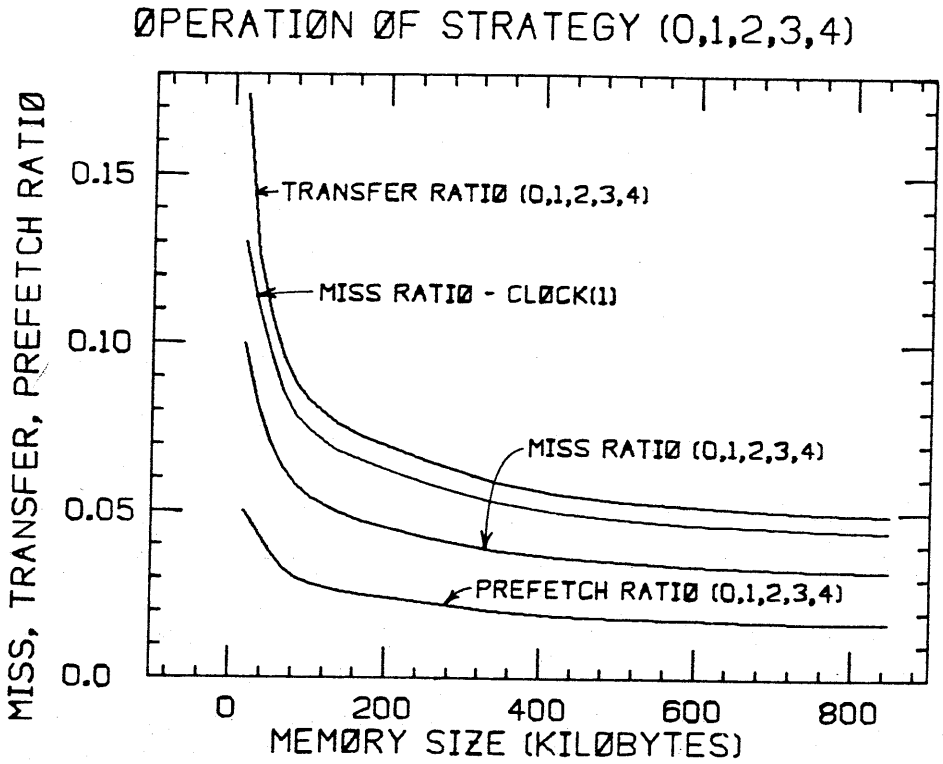

Fig. 10

certain fraction of the time [18] a run will not reference consecutive segments but will instead reference every second, third, or nth segment. In the case that the block size is large enough, this will appear as a run of consecutive blocks. These alternate segment references, however, do not appear as consecutive runs when taken at the segment or byte level. For all block sizes large enough to contain three or more segments [18], one run length distribution will suffice, since only rarely are there runs that skip more than two segments at a time; for smaller block sizes the calculation is not so straightforward. With most direct access storage devices, block sizes will be at least 800 or so bytes.

The optimization described in this section has not been performed, although it is straightforward but tedious. Ragaz and Rodriguez-Rosell [18] report that enlarging the block size produces miss ratios only slightly inferior to comparable fixed prefetching strategies such as were used for comparison in Figures 11 and 12. As noted in Section 4C, however, fixed prefetching is measurably inferior to optimized variable prefetching. There is, therefore, reason to believe that selective prefetching when combined with a carefully chosen block size can yield a significant improvement over the observed behavior.

6. ALTERNATIVE PREFETCHING STRATEGIES

A number of different characteristics of the trace data were examined in order to devise improved prefetching algorithms and improved block replacement algorithms. We briefly discuss two of these characteristics here.

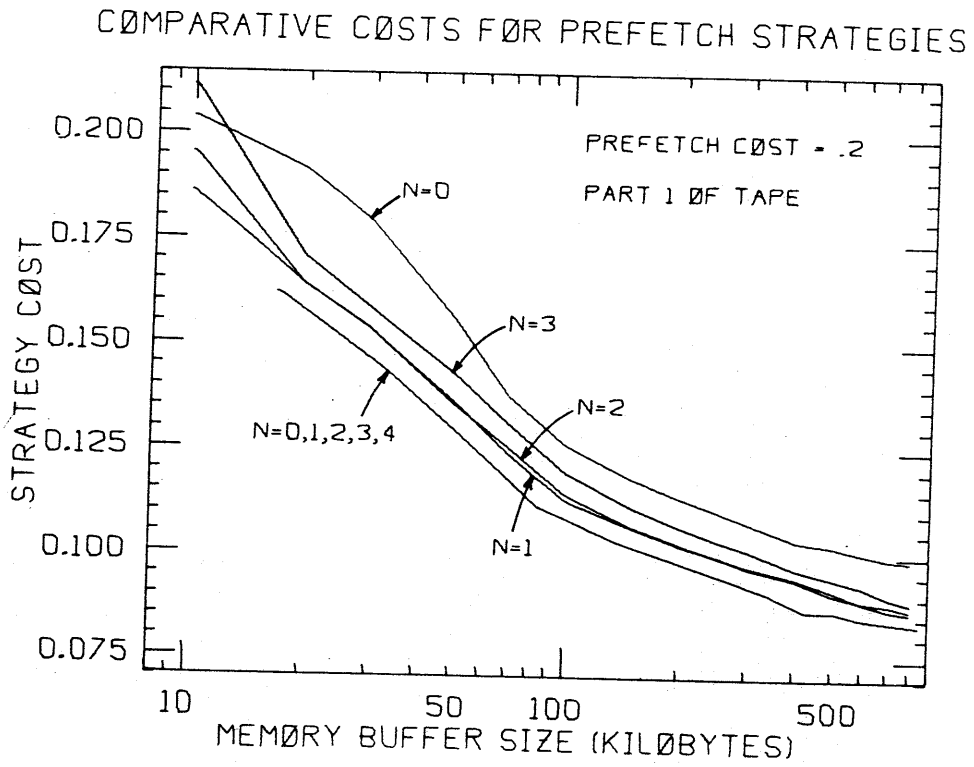

Fig. 11

In Table VI we present a number of statistics of the empirical run length distribution for the sections of the trace. The mean run length can be seen to be essentially the same for all sections of the trace, but the variance changes significantly. The changes in the variance may be due to nonstationarity or may be due instead to the random occurrence of samples from a highly skewed distribution. For example, one run of more than 1000 blocks was observed, and this instance would greatly perturb the estimate of the variance.

Autocorrelation coefficients for time series are defined only when the series is stationary. A stationary, uncorrelated series will have estimated autocorrelation coefficients of order k of mean 0 and variance $1/(n - k)$ for a series length of n. Despite the questions about the stationarity of the series, we have computed the first three autocorrelation coefficients for the sequence of run lengths and present that data in Table VI along with the 5 percent significance level. Even though some of the autocorrelations are statistically significant, they are far too small in magnitude to have any useful predictive power. We also note that trends in the data usually are reflected in large positive autocorrelation coefficients, so the lack of genuinely large correlations suggests that the run length distribution may be sufficiently stable for our purposes. We conclude from the measured autocorrelation coefficients that knowledge of previous run lengths is unlikely to be useful for estimating the length of the current one.

It was hypothesized that consecutive interreference intervals to the same block

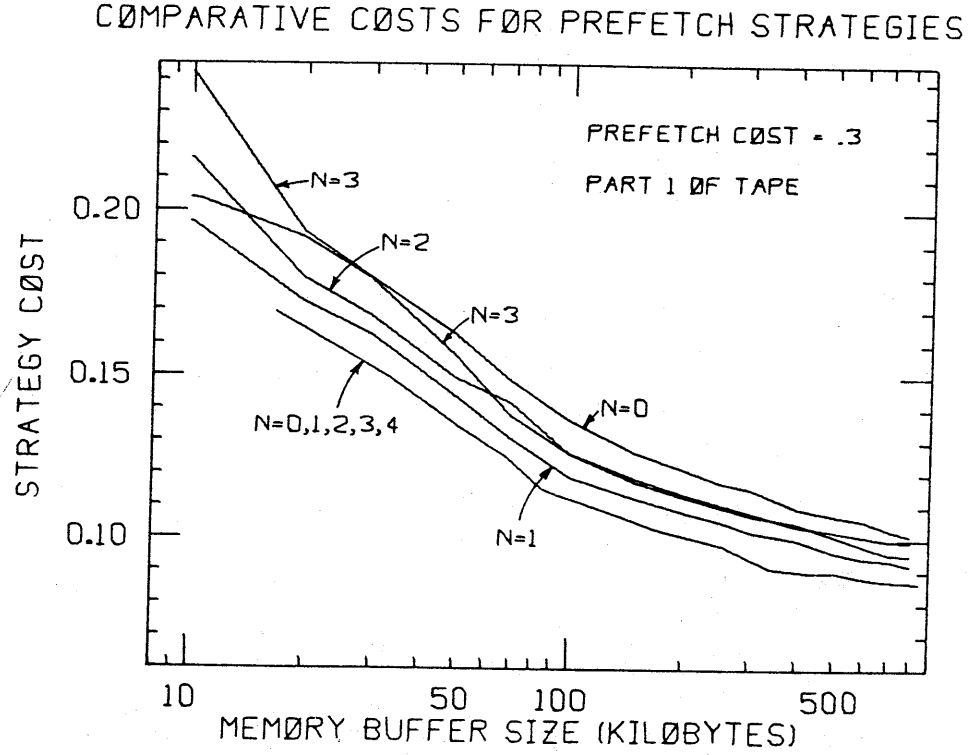

Fig. 12

Table VI. Run Length Characteristics

		Part 1	Part 2	Full tape
Mean run length	1.567	1.54	1.57	
Variance of run length	15.9	10.4	29.7	
Coefficient of variation	2.54	2.09	3.47	
Autocorrelation of run length				
First order	0.01	0.006	0.0031	
Second order	0.086	0.009	0.114	
Third order	0.014	−0.003	0.00265	
5 percent significance level	0.0085	0.0106	0.0039	
Number of runs	52,796	34,117	254,329	

would be related in some simple manner. A scatter plot of these interreference intervals was made and no useful pattern emerged.

The author ran other experiments (not described here) and discussed the data with other researchers who were also studying the same traces. No other useful predictor for prefetching was found. There is a large variety of experiments that were not tried (including some suggested by more recent research on prefetching), however, and thus whether something besides sequentiality is useful for prefetching remains an open question.

7. CONCLUSIONS

Sequentiality of access is a predictable consequence of many types of database organization. By anticipating this sequentiality of access, it is possible, using prefetching, to achieve a reduction in I/O delays and processing. By conditioning on the run length, we have been able to optimize the degree of prefetching. As indicated in Section 5, our method of calculating the best degree of prefetching is also applicable to the choice of block size.

We believe that the implementation of our results will yield measurable improvements in the operation of many database systems.

ACKNOWLEDGMENTS

The author would like to thank IBM as a whole, whose support and facilities made possible the research reported here. In addition, the following individuals (among others) were helpful both by discussing with the author the technical content of this work and in aiding in its realization: Edward Altman, David Choy, Thomas Price, Juan Rodriguez-Rosell, Donald Slutz, and William Tuel. Complete responsibility for the accuracy of the results reported here of course rests with the author.

REFERENCES

1. AHO, A.V., DENNING, P.J., AND ULLMAN, J.D. Principles of optimal page replacement. *J. ACM 18,* 1 (Jan. 1971), 80–93.
2. BAER, J.L., AND SAGER, G.R. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. Software Eng. SE-2,* 1 (March 1976), 54–62.
3. BARD, Y. Characterization of program paging in a time-sharing environment. *IBM J. Res. Develop. 17,* 3 (Sept. 1973), 387–393.
4. CORBATO, F.J. A paging experiment with the Multics system. In *In Honor of P.M. Morse,* M.I.T. Press, Cambridge, Mass., 1969, pp. 217–228.
5. DATE, C.J. *An Introduction to Data Base Systems.* Addison-Wesley, Reading, Mass., 1975.
6. DENNING, P.J. The working set model for program behavior. *Comm. ACM 11,* 5 (May 1968), 323–333.
7. GAVER, D.P., LAVENBERG, S.S., AND PRICE, T.G. JR. Exploration analysis of access path length data for a data base management system. *IBM J. Res. Develop. 20,* 5 (Sept. 1976), 449–464.
8. GOLD, D.E., AND KUCK, D.J. A model for masking rotational latency by dynamic disk allocation. *Comm. ACM 17,* 5 (May 1974), 278–288.
9. HELD, G.D., STONEBRAKER, M.R., AND WONG, E. INGRES—a relational data base system. Proc. AFIPS 1975 NCC, AFIPS Press, Montvale, N.J., pp. 409–416.
10. IBM CORP. Information Management System/360, Version 2, General Information Manual. Form GH20-0765-3, IBM Corp. Tech. Pub. Dept., Palo Alto, Calif., 1973.
11. IBM CORP. Information Management System/360, Version 2, Application Programming Reference Manual. Form SH20-0912, IBM Corp., Palo Alto, Calif., Nov. 1973.
12. IBM CORP. Information Management System/360, Version 2, System Programming Reference Manual. Form SH20-0911, IBM Corp., Palo Alto, Calif., Sept. 1974.
13. JOSEPH, M. An analysis of paging and program behavior. *Comptr. J. 13,* 1 (Feb. 1970), 48–54.
14. LAVENBERG, S.S., AND SCHEDLER, G.S. A queueing model of the DL/I component of IMS. Res. Rep. RJ 1561, IBM Res. Lab., San Jose, Calif., April 1975. Republished as [15].
15. LAVENBERG, S.S., AND SHEDLER, G.S. Stochastic modelling of processor scheduling with application to data base management systems. *IBM J. Res. Develop. 20,* 5 (Sept. 1976), 437–448.
16. LEWIS, P.A.W., AND SHEDLER, G.S. Statistical analysis of transaction processing in a data base system. Res. Rep. RJ 1629, IBM Res. Lab., San Jose, Calif., Sept. 1975. Republished as: Statistical analysis of non-stationary series of events in a data base system. *IBM J. Res. Develop. 20,* 5

(Sept. 1976), 465–482.
17. MATTSON, R., GECSEI, J., SLUTZ, D.R., AND TRAIGER, I.L. Evaluation techniques for storage hierarchies. *IBM Syst. J. 2* (1970), 78–117.
18. RAGAZ, N., AND RODRIGUEZ-ROSELL, J. Empirical studies of storage management in a data base system. Res. Rep. RJ 1834, IBM Res. Lab., San Jose, Calif., Oct. 1976.
19. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Comm. ACM 17,* 7 (July 1974), 365–375.
20. RODRIGUEZ-ROSELL, J. Empirical data reference behavior in data base systems. *Computer 9,* 11 (Nov. 1976), 9–13.
21. RODRIGUEZ-ROSELL, J., AND HILDEBRAND, D. A framework for evaluation of data base systems. Res. Rep. RJ 1587, IBM Res. Lab., San Jose, Calif., May 1975. Also in Proc. Int. Comput. Symp., Antibes, France, June 1975.
22. SMITH, A.J. A locality model for disk reference patterns. Proc. IEEE Comptr. Soc. Conf., San Francisco, Feb. 1975, 109–112.
23. SMITH, A.J. Analysis of a locality model for disk reference patterns. Proc. Second Conf. Inform. Sci. and Syst., Johns Hopkins U., Baltimore, Md., April 1976, 593–601.
24. SMITH, A.J. Sequential program prefetching in memory hierarchies. April 1977; submitted for publication. To appear, *Computer.*
25. TRIVEDI, K.S. Prepaging and applications to array algorithms. *IEEE Trans. Comptrs. C-25,* 9 (Sept. 1976), 915–921.
26. TRIVEDI, K.S. Prepaging and applications to the STAR-100 computer. Proc. Symp. High Performance Comptr. and Algorithm Organization, Champaign, Ill., April 1977.
27. TRIVEDI, K.S. An analysis of prepaging. Comptr. Sci. Rep. CS-1977-7, Duke U., Durham, N.C., Aug. 1977.
28. TRIVEDI, K.S. On the paging performance of array algorithms. *IEEE Trans. Comptrs. C-26,* 10 (Oct. 1977), 938–947.
29. TUEL, W.G. JR. An analysis of buffer paging in virtual storage systems. Res. Rep. RJ 1421, IBM Res. Lab., San Jose, Calif., 1974.
30. TUEL, W.G. JR., AND RODRIGUEZ-ROSELL, J. A methodology for the evaluation of data base systems. Res. Rep. RJ 1668, IBM Res. Lab., San Jose, Calif., Oct. 1975.

Received January 1977; revised October 1977