Computer Architecture and Systems

J. P. Hayes Editor

Long Term File Migration: Development and Evaluation of Algorithms

Alan Jay Smith
University of California, Berkeley

The steady increase in the power and complexity of modern computer systems has encouraged the implementation of automatic file migration systems which move files dynamically between mass storage devices and disk in response to user reference patterns. Using information describing 13 months of user disk data set file references, we develop and evaluate (replacement) algorithms for the selection of files to be moved from disk to mass storage. Our approach is general and demonstrates a general methodology for this type of problem. We find that algorithms based on both the file size and the time since the file was last used work well. The best realizable algorithms tested condition on the empirical distribution of the times between file references. Acceptable results are also obtained by selecting for replacement that file whose size times time to most recent reference is maximal. Comparisons are made with a number of standard algorithms developed for paging, such as Working Set, VMIN, and GOPT. Sufficient information (parameter values, fitted equations) is provided so that our algorithms may be easily implemented on other systems.

Partial support for this research has been provided by the National Science Foundation under grants MCS75-06768 and MCS77-28429, and by the Department of Energy under Contracts W-7405-ENG-48 (to the Lawrence Berkeley Laboratory) and DE-AC03-76SF00515 (to the Stanford Linear Accelerator Center).

Author's present address: A. J. Smith, Computer Science Division, EECS Department, University of California, Berkeley, CA 94720. The author is also on the staff of the Lawrence Berkeley Laboratory and is a visitor at the Stanford Linear Accelerator Center.

Key Words and Phrases: file migration, paging, replacement algorithm, memory hierarchies, mass storage.

CR Categories: 4.33, 4.35, 4.41

I. Introduction

Most large computer installations have memory hierarchies similar to that shown in Figure 1. The limited storage capacity of the disk system generally results in some form of file migration, whereby active files are kept on or moved to the disk and inactive files are moved to or kept on tape or mass storage. This migration may be managed by the user or done automatically by the system. The increasing need for such migration has led many large computer installations and manufacturers to supply migration as a feature of the file system, although such features vary widely in their transparency to the user and the extent to which they are automatic. Currently, SLAC [2], the Lawrence Berkeley Laboratory [13], and a number of other Department of Energy Laboratories [9, 17] have projects in progress to install such file migration systems. IBM now provides automatic file migration [5, 11, 14, 20]. Various independent software vendors also have available file migration packages. Users have also set up their own migration systems (e.g., [12]).

The spreading popularity of file migration has not been accompanied by the research results needed to select good or even satisfactory algorithms for the fetch, placement, and replacement of files. The problems are as follows: when to migrate a file from mass storage to disk (fetch algorithm), where on disk to place it (placement algorithm), and when to remove that file to mass storage when the disk space is again needed (replacement algorithm). In this paper, we develop and evaluate, based on real file system access data, a number of file replacement algorithms. Our approach to this problem is initially broad and we show a general methodology for the problem. The data do not permit us to consider the fetch problem in a useful way, and we choose at this time not to consider the placement algorithm problem.

There has been considerable study of the replacement algorithm problem in the context of main memory paging. We refer the reader to [23] for an extensive bibliography on the subject. With regard to files, however, the only useful study of file migration, because it uses real data, is one by Stritter [27]; we use the same data, but our analysis goes considerably further. To our knowl-

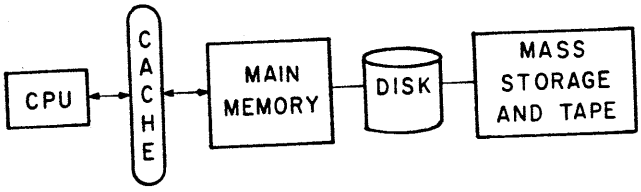


Fig. 1. Memory Hierarchy of a Large Computer Installation.

edge. the only other set of data in existence suitable for similar research is that described by Revelle [21], but he has not considered file migration algorithms. Zehab and Boies [28] describe the algorithm that they implement, but their system is specialized and no comparative evaluations or data are given.

A number of other researchers have considered the file migration problem, either in general, descriptive terms or by using unvalidated mathematical models. In the first category, we note [1]. Mathematical models and mathematical optimization results are presented in [15] and [18]. Some additional references appear in [26].

As noted earlier, this paper is concerned with the development and comparative evaluation of algorithms for the migration of files from disk back to mass storage. Our emphasis throughout is on the use of measured file system data to both construct algorithms and then to evaluate them. In a companion paper to this [24], we discuss the origin and features of these data in detail and analyze them extensively. A brief summary of this information is provided in the next two sections (II and III), but the reader is urged to refer to [24]. We then consider principles and criteria for replacement algorithms; that discussion is followed by a section in which file replacement algorithms are explained and/or developed. Experimental comparisons and parameter values appear in Sec. VI. We find that algorithms based on both the file size and the time since the file was last used work well. The best realizable algorithms tested conditions on the empirical distribution of the times between file references. Acceptable results are also obtained by selecting for replacement that file whose size times time to most recent reference is maximal. Comparisons are made with a number of standard algorithms developed for paging, such as Working Set, VMIN, GOPT, etc. Sufficient information (parameter values, fitted equations) is provided so that our algorithms may be easily implemented on other systems. Some consideration of various implementation issues appears in the last section.

II. Data Description

Our data consist of a record of each Wylbur [10] text editor data set that existed over a 384 day period (1974–1975) at the Stanford Linear Accelerator Center. (In fact, almost all user disk files are included. Our analysis, of course, reflects just this class of files.) These data were collected by E.P. Stritter, and a preliminary analysis of the data appears in his dissertation [27]. For each file and each day, one bit is available indicating whether or not that file was used that day. Also available is the file name, the account identifier of the file's owner, the date on which the file was created, the date (if any) on which the file was scratched, and the file size in disk tracks. No information is available as to whether the file was read or written, nor how many times per day it was used.

We were able to classify files by both size and "class." Files were placed into seven size groups, based on the base 2 logarithm of their size in tracks. The file size groups (numbered 0 . . . 6; called "logsize" or "Lsize" or "L") were: 1 track, 2–3 tracks, 4–7 tracks, 8–15 tracks, 16–31 tracks, 32–63 tracks, and ≧64 tracks. We specified three file classes: files with the name "Active" were created by the system to save Wylbur text editor data sets in use at the time of a system crash or an automatic logout. Files whose names began with "Lib" were classified as partitioned data sets and were placed in a class called "libraries." All other files were placed in a class called "other files." Some of our replacement algorithms use both the size and class groupings.

The 384 day period during which the system was observed consisted of working days, holidays, weekends, and 15 days during which either no data were collected or the system was down. Since file migration would most likely occur on working days only, we have restricted our work in this paper to using the data for working days only. Specifically, we have mapped all file events (create, scratch, reference) onto the next working day after the given holiday, weekend, or unmeasured day. Thus, for the purposes of migration, files created or accessed on a Saturday would be considered to have been acted upon on the following Monday. The intent is to represent a system in which the file replacement program would run every working day at midnight, using "time since last reference" based only on working days.

III. File Reference Patterns

As noted earlier, file reference patterns were extensively analyzed in an earlier paper [24]. Here we summarize the essential findings and present some results not shown earlier.

It was observed that most files were used on a small number of days: half of all files were accessed on two or fewer days during the measurement period. The mean number of days files were referenced, however, was 10.6; thus, the distribution of day-references/file is highly skewed. Of those files whose reference patterns permitted useful statistical analysis, about one-third were found to show a declining rate of reference (i.e., declining trend) with age. Considering only those files with no trend, about 5 percent showed statistically significant serial correlation of the interreference intervals (positive in almost all cases) and about 40 percent were found to have interreference time distributions that were more skewed than the geometric distribution. Significant differences in various parameters (interreference time distributions, number of references, etc.) were found between files of different sizes and classes.

An attempt was made to fit the interreference distributions with the weighted sum of two geometric distributions. Let $g(i, L, C)$ be the empirical interreference time distribution for files of the given size group (L) and class (C), and let $gf(i, L, C)$ be the fitted distribution

(i.e., $g(i, L, C)$ is the probability that the time between consecutive references to a file of size group L and class C is equal to i). Then $gf(i, L, C)$ is of the form

$$gf(i, L, C) = ab(1 - b)^{i-1} + (1 - a)c(1 - c)^{i-1}. \quad (1)$$

The method of moments was used to select the values of a, b, and c ($0 \leqq a \leqq 1$, $0 < b < 1$, $0 < c < 1$) for each size/class combination, and the results are shown in Table I. Although the chi-square goodness of fit test showed that in most cases the fit was unsatisfactory, we do make use of the fitted distributions later in this paper and find them useful.

Figures 2 and 3 show the empirical and fitted interreference time distributions for references to files. These figures give the results for working days (as does Table I) as compared to data in [24] which show similar information for all (including nonworking) days.

IV. Principles, Criteria, and Computations for File Replacement Algorithms

In order to compare the relative performance of file replacement algorithms, it is necessary to have some criterion for evaluation. Our criterion is as follows: Define A to be the replacement algorithm and P to be the parameter associated with that algorithm (e.g., the working set window size). Let $M(A, P)$ (the "miss ratio") be the fraction of all file "references" that require a fetch from mass store. (The term reference is used here to refer to the first time a file is used on a given day; it is assumed that files stay resident for the entire remainder of the day and thus subsequent reads, writes, or opens are ignored in our computations.) $S(A, P)$, ("space") is the mean number of disk tracks occupied by on-line files. The pair $[S(A, P), M(A, P)]$ constitutes an operating point and may be plotted on an x–y plot as shown in Figures 6–13.

An operating point $(X0, Y0)$ is considered to be better than an operating point $(X1, Y1)$ if and only if (a) $X0 \leqq X1$ and $Y0 < Y1$, or (b) $X0 < X1$ and $Y0 \leqq Y1$. It should be clear that it is possible for two operating points to not be ordered as better or worse (e.g., $X0 < X1$, $Y0 > Y1$). An algorithm A is considered to be uniformly better than an algorithm B if for all operating

Table I. Working Days.

Size (tracks)	Class	Fitted Parameter Values a	b	c	$Niref$
1	Libraries	0.924	0.612	0.035	189
2–3		0.952	0.523	0.029	887
4–7		0.924	0.644	0.047	4302
8–15		0.974	0.612	0.039	14947
16–31		0.981	0.719	0.051	21099
32–63		0.965	0.851	0.126	7820
≧64		0.989	0.802	0.048	1792
All		0.977	0.677	0.043	51036
1	Other files	0.835	0.458	0.032	23745
2–3		0.880	0.418	0.029	25500
4–7		0.916	0.466	0.031	24893
8–15		0.941	0.491	0.032	22824
16–31		0.954	0.543	0.034	15223
32–63		0.980	0.597	0.027	10576
≧64		0.982	0.749	0.045	9814
All		0.914	0.490	0.031	132575
1	Active	0.818	0.280	0.026	3664
2–3		0.793	0.362	0.032	1017
4–7		0.928	0.265	0.020	516
8–15		0.958	0.379	0.017	216
16–31		0.875	0.873	0.045	139
32–63		...	...	...	56
≧64		...	...	...	3
All		0.843	0.288	0.026	5611
1	All files	0.836	0.417	0.030	27598
2–3		0.879	0.418	0.030	27404
4–7		0.922	0.468	0.031	29711
8–15		0.956	0.527	0.033	37987
16–31		0.972	0.627	0.037	36461
32–63		0.987	0.655	0.028	18452
≧64		0.983	0.756	0.045	11609
All		0.930	0.518	0.031	189222

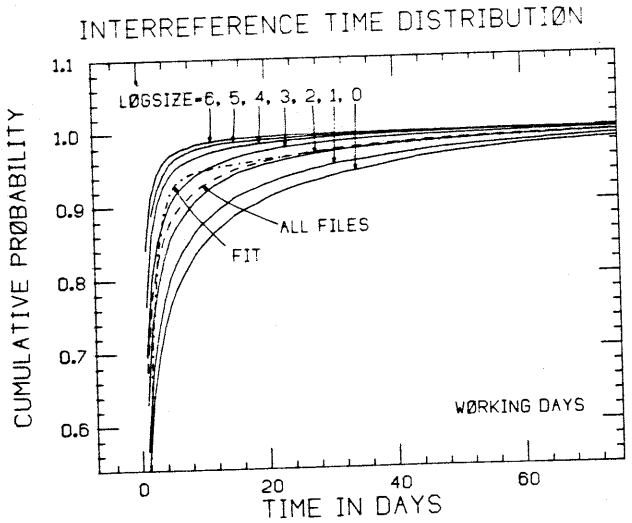


Fig. 2. Empirical and Fitted Interreference Time Distribution for References to Files.

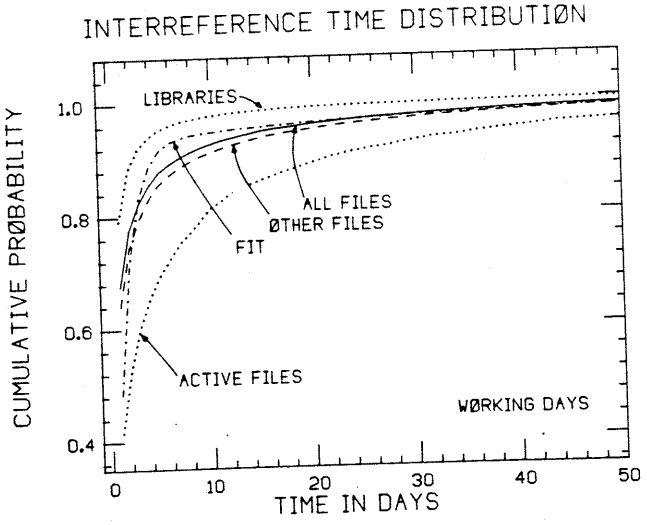


Fig. 3. Empirical and Fitted Interreference Time Distribution for References to Files.

points (XA, YA) for A and (XB, YB) for B, $XA = XB \rightarrow YA < YB$. [The better algorithm has an (S, M) curve which is to the left and below the worse algorithm.] Algorithms which are not uniformly better or worse may only be compared at specific operating points or ranges of operating points.

There are two important observations to be made about our criteria for comparison. First, we have treated all file misses as equally important. In particular, a missing large file is considered to be no more costly than a missing small file. Since the largest file observed (252 tracks) can be transmitted in less than 1.5 sec. and since the mean latency time for most forms of mass storage is upwards of 10 sec. (more likely several minutes), ignoring file size in the miss penalty seems to be appropriate. Second, we have not selected a specific operating point. In contrast, a very popular criterion for comparing paging algorithms is to compare minimal space–time products (see, for example, [3]). We do not believe that this criterion is appropriate even for paging algorithms, since the (uniprogramming) minimal space–time product may not indicate the correct operating point in a multiprogramming environment. This criterion is not at all applicable in the context of file migration because missing file faults do not convert in some standard way to the same units used to measure the space–time integral. An operating point may be determined by selecting a conversion factor [see Eqs. (4), (5), and (6) and discussion below]. We prefer to leave it to the reader or system implementor to pick the most suitable operating point; in some cases, a miss ratio of 10 percent may be tolerable; in other cases, 1 percent or less may be appropriate.

It is straightforward to compute the values for $S(A, P)$ and $M(A, P)$ for a given algorithm A and parameter P. We make the following definitions to aid in our computations:

- $Nref(i)$: number of times (days) that file i is referenced;
- $Nref$: total number of references to all files (=213,692);
- $Niref(i)$: number of interreference intervals to file i = $Nref(i) - 1$ if $Nref(i) \geqq 1$; 0 otherwise;
- $Niref$: total number of interreference intervals to all files = (189,222);
- $Sz(i)$: size of file i in tracks (sometimes abbreviated to Sz);
- Dayno: the number of days in the measurement period (256 working days);
- $I(i, j)$: the length of the jth interreference interval for the ith file. $(I(i, j) \geqq 1)$.

For each interreference interval (i, j), the file will be retained in memory after the reference beginning the interval for some number of days $Kp(A, P, i, j)$, including the day of reference, where A and P are again the algorithm and the parameter value. We will generally abbreviate this as $Kp(i, j)$. We define this in such a way that $Kp(i, j) \geqq 1$; that is, the initial day of reference is included in the residence time. Please note that if $Kp(i, j) \leqq I(i, j)$, then there is a fault at the time of the next reference; otherwise there is no fault. Let $F(A, P, i, j)$ [abbreviated $F(i, j)$] be either 0 if there is no fault or 1 if there is a fault on the jth reference to the ith file. Then we compute $S(A, P)$ and $M(A, P)$ as follows:

$$S(A, P) = \sum_{i,j} (\min[Kp(i, j), I(i, j)]Sz(i)/\text{Dayno}, \quad (2)$$

$$M(A, P) = \sum_{i,j} F(i, j)/Niref. \quad (3)$$

We note that our method of computation implies a small inaccuracy. We have ignored the boundary condition of a finite measurement period—files in existence at the end of our measurement period have an unknown interreference interval, as do files in existence at the beginning. We have chosen to omit the contribution to $M(A, P)$ in both cases, and we have not counted the first reference to a file as causing a fault. Thus we assume that the file is resident on the disk at the time of the first reference to it. We are interested only in comparative results (among algorithms), and there is no reason to think that the relative performance of the algorithms would have shifted through such a simplification.

It should be clear that there is a trade-off of space for file faults (instances of a missing file). That is, if $Kp(i, j)$ is reduced, the value of $M(A, P)$ will either increase or remain the same, and the value of $S(A, P)$ will decrease. More generally, for all stack algorithms [16] it can be shown that the (S, M) curve is monotonically nonincreasing for increasing S. All of the algorithms that we consider are stack algorithms. We also note that our criterion for algorithm evaluation tends [as is evident from Eqs. (2) and (3)] to make large files much better candidates for replacement than small files.

From Eqs. (2) and (3), it can be seen that an optimal look-ahead algorithm would select any file which has just been referenced and remove it if and only if for its upcoming interreference interval, $[Sz(i)(I(i, j) - 1)] > P$; otherwise the file would be retained until the next reference. By varying P, an (S, M) curve is traced out. This algorithm is known as GOPT (see [8] for details). VMIN [19] is the algorithm which removes all files (or pages) whose time to next reference is greater than P. VMIN is optimal only in the case of fixed size files or pages. VMIN and GOPT will be used for purposes of comparative evaluation later in this paper.

V. Replacement Algorithms: Methodology, Definitions, and Derivations

In this section we describe and/or derive the remaining file replacement algorithms considered. Our approach employs a general methodology, as shown below, and is applicable to many similar problems. The class of algorithms considered will be of the "variable space"

variety, which means that the total volume of on-line files may vary even though the parameter value P for the replacement algorithm remains fixed. This is in contrast to the fixed space algorithms such as LRU, FIFO, MIN, etc., which always maintain a fixed volume of on-line files or pages.

At the end of the previous section (IV), we described the optimal look-ahead replacement algorithms VMIN and GOPT. We proceed in this section to consider realizable algorithms, beginning with probabilistically "optimal" ones and moving to algorithms which are successively more *ad hoc* and less likely to perform well.

Our approach to algorithm design is largely empirical. Based on our analysis of the observed file reference patterns (see [24]), we create a general model for the file reference process; we assume *a priori* that a file is accessed independently of all others. This general model suggests a file migration strategy. Parameters for the migration algorithm can be obtained from analyzing the file reference patterns in the context of the model. Simpler algorithms can be derived by eliminating "nonessential" parts of the model or by creating simpler but less accurate models. For example, the working set algorithm [6] implicitly assumes that pages are referenced independently as a renewal process with decreasing hazard rate; this suggests that pages should be retained in memory for some period T. The reasoning in this section is similar.

A. A Stochastically Optimal Algorithm

It was noted earlier that the optimal but unrealizable algorithms VMIN and GOPT used the known time to next reference to determine when to remove a file. A realizable algorithm must, of course, deal only with known information, i.e., past history, and therefore it can at best estimate which file will not be used. A realizable algorithm which provides accurate estimates of the time to next use can be expected to yield good performance. We use a general approach below to derive effective realizable algorithms.

Let $Q(A, H, t, Sz, C, P)$ be the policy for whether to remove a file; that is, $Q(A, H, t, Sz, C, P)$ specifies whether the file should be kept or removed, given the values of the parameters. (The word "policy" here is chosen to correspond to its use in dynamic programming.) A is the algorithm; throughout this section (V.A.) we shall consider the stochastically optimal policy (denoted Stochopt, abbreviated Sopt) only. Unless it causes confusion, we shall omit A as a parameter. H is the entire previous history of reference to the file, t is the time (date), Sz is the size of the file, C is the class of the file, and P is the "cost" of fetching the file when it is next referenced (should we decide to remove it). P in this case is the parameter and will be varied to produce the (S, M) curve; its value is only useful in specifying an operating point. We have already assumed that files are accessed independently, so the optimal policy Q includes all of, but only the information relevant to the file in question. Further, for files with identical values of H, t, Sz, C, and P, the policy will be the same.

It is both possible to reduce the number of parameters for Q (since some of them are not helpful), and desirable (since there are more parameters than can be conveniently used). We first assume that file reference patterns are independent of the absolute time, all other things being equal. To some extent this is a simplification, since summer usage patterns should differ from winter, etc., but it does not appear to be unreasonable. Thus we replace t as a parameter with a for the age of the file (in days since it was created). Our analysis [24] has shown little if any serial correlation between interreference intervals, so we choose to replace H, the entire previous reference history of the file, with h, the time since the last reference. We also choose to consider only demand fetch policies, i.e., only that fetch algorithm which fetches a file at the time it is referenced, and not before. In this case, the optimal policy Q is more general than necessary, since the file will either be kept for the entire interreference interval or will be removed from higher level storage and only fetched when again referenced.

Therefore, Q is replaced with the optimal (equivalent) policy $K(A, a, Sz, C, P)$, where K is the number of days the file is to be kept resident from and including its last day of reference $[K(A, a, Sz, C, P) \geqq 1]$, A (= Sopt), a, Sz, C, and P are as before. K and Q are equivalent, but K is simpler to compute and easier to use. Referring to earlier notation, $Kp(i, j)$ will in each case be computed using the policy K.

It is worth observing here that the policy approach $K(A, a, Sz, C, P)$ is equivalent to a working set policy in which the window size may be different for a file depending (only) on its size, class, and age. As the independent parameter P varies, the window size may change in a different way for files which have different values of Sz, C, and a.

K is computed by using previous patterns of reference to this and similar files to estimate the probability distribution for the time to next reference to this file. We would thus like to measure $R(a, Sz, C)$, which is the empirical distribution of time (in days) between references to a file, given that it is of size Sz, class C, and is age a at the date of its last reference. Again, this parameter space is too large, primarily in that we have insufficient data to estimate distributions reliably. Therefore, in almost all cases a is dropped as a parameter, and Sz is replaced by Logsize (L), i.e., the size class into which Sz falls. Some observations in [24] tended to indicate that this aggregation of files into size classes grouped files of similar reference patterns. In that paper, we also noted that about one-third of all files showed declining rates of reference with age. While we cannot at this time directly substantiate the insignificance of this observation, we believe that its effect is extremely small compared to file size, class, and time since last reference. One simple age based algorithm was developed and was found to perform poorly. It has been omitted for brevity.

The resulting distribution is referred to as $g(i, L, C)$, which is the empirical distribution of the times between references to a file of size class L and type class C. $G(i, L, C)$ is the cumulative distribution and $gf(i, L, C)$ and $Gf(i, L, C)$ are the fitted distributions [see Eq. (1)].

Finally, we make one additional simplification. Experiments which appear in the next section (VI.B.) tend to indicate that the class C of the file is not very useful in selecting a file for replacement. For this reason, and because the size of the computation is unpleasantly large [see Eq. (4) below], we have also dropped the class C from our computations in developing Stochopt.

Computing $K(\text{Sopt}, Sz, P)$ (Sz and P are the remaining parameters) is straightforward given the distribution $g(i, L, *)$ ($*$ indicates that the distribution has been aggregated over all values for that parameter). Thus,

$$K(\text{Sopt}, Sz, P) = \left\{ k \mid \min_{k} \left[\sum_{i=1}^{k} g(i-1, L, *)(i-2)Sz + (1 - G(k-1, L, *))((k-1)Sz + P) \right] \right\} \quad (4)$$

(where $g(0, L, C) = 0$, $G(0, L, C) = 0$). This equation simply selects that value of k which minimizes the cost of an interreference interval, where the cost of a fault is given by the parameter P. This is equivalent to specifying the cost of the algorithm as a whole as

$$\sum_{i,j} \min[Kp(i, j), I(i, j)] \times Sz(i) + \sum_{i,j} F(i, j)P \quad (5)$$

Table II. Stochopt File Retention Period.

File size	Miss ratio		
	1 percent	5 percent	10 percent
1	226	226	226
2	235	235	23
3	235	49	16
4	246	28	12
5	246	17	9
6	246	17	6
7	172	13	6
8	78	12	6
9	78	11	6
10	46	10	6
11	40	9	5
12	40	8	4
13	38	7	4
14	35	7	4
15	24	6	4
16	35	7	4
20	16	6	4
25	13	5	3
30	12	4	3
35	10	4	2
40	10	4	2
45	9	3	2
50	7	3	2
60	7	3	2
70	6	3	2
80	6	2	2
90	6	2	2

$$= S(A, P) \times \text{Dayno} + M(A, P) \times \text{Niref} \times P, \quad (6)$$

where $(P \times \text{Niref})/\text{Dayno}$ is just the constant of proportionality that relates the relative impact of $S(A, P)$ and $M(A, P)$ on the total cost of the replacement policy. We have selected a policy $K(\text{Sopt}, Sz, P)$ that minimizes Eq. (6); i.e., it implicitly specifies an operating point. By varying P, a curve in the (S, M) plane is traced out. (If this criterion is applied to page replacement algorithms and P is set equal to the product of the page fetch time and the mean number of pages in use at fault times, then this criterion is minimal space–time product.)

The reader will note that many of the algorithms defined elsewhere in this paper employ the residual lifetime function, i.e., the distribution of time to the next reference, given that the file has not been referenced for j days. Stochopt can also be expressed in such a manner. This formulation is not as straightforward, as it is based on dynamic programming; for that reason, and for brevity, we do not include it here. (See [25] for the use of a similar methodology.)

In Table II, we show the values for $K(\text{Sopt}, Sz, P)$ for a range of sizes Sz and for those values of P which yield values of $M(\text{Sopt}, P)$ of 1 percent, 5 percent, and 10 percent.

B. Expected Time to Next Reference

A simpler (to compute and to understand) approach than that used above to selecting the file whose [(time to next reference) × (size)] is maximal is to directly use the expected mean time to next reference. This method is not optimal, since the mean expected time to next reference contains far less information than the entire interreference time distribution. A counter example is provided in [4] for fixed size files (i.e., pages). The reason for this nonoptimality can be shown by the following example: Let the distribution of time to the next reference be bivalued, with the probability 0.5 that the reference is in 1 day and 0.5 that the reference is in 99 days; then the expected time to the next reference is 50 days. The optimal policy would most likely keep the file for one more day (which means that there is a 50 percent chance of using the file) and then if it is not used, discard it. A policy based on only the expected time to next reference would probably discard the file immediately, since the expected time is so large.

The expected time to next reference ($E(i, L, C)$, where i is the number of days since the file was last referenced; $i = 0$ implies that the file was referenced that day) can be computed from the interreference distribution g as follows:

$$E(i, L, C) = \sum_{j=i+1}^{\infty} g(j, L, C)(j - i)/(1 - G(i, L, C)). \quad (7)$$

The algorithm for file removal is then to remove any file for which

$$(E(i, L, C) - 1) \times Sz > P. \quad (8)$$

It is also possible to suppress either or both of L and C and just compute $E(i, *, C)$, $E(i, L, *)$, or $E(i, *, *)$. The file removal decision can then be based on only one or neither of L and C. We note that the size of the computation is no longer a problem, so it has not been necessary to reduce the parameter space as was done in Stochopt.

We let "Etnr" denote the file removal algorithm specified by Eq. (8). Then we can define a policy K(Etnr, Sz, C, P) as follows:

$$K(\text{Etnr}, Sz, C, P) = \{i \mid \min_i[((E(i, L, C) - 1) \times Sz) > P]\}. \quad (9)$$

The fitted distribution gf can be used in place of the empirical distribution g. The algorithm in that case is named "Etnrf" ("f" for "fitted") and we compute it in the same manner as in Eq. (7):

$$Ef(i, L, C) = \sum_{j=i+1}^{\infty} gf(j, L, C)(j - i)/(1 - Gf(i, L, C)). \quad (10)$$

We define the policy K(Etnrf, Sz, C, P) as

$$K(\text{Etnrf}, Sz, C, P) = \{i \mid \min_i[((Ef(i, L, C) - 1) \times Sz) > P]\}. \quad (11)$$

It occasionally happens that K(Etnr, Sz, C, P) = 1; that is, the file is always removed at the end of the day on which it is referenced. This seems counterintuitive to many people, although, of course, our formulas take this into account; thus we define an additional policy "Etnrb" (b for "bound") as equivalent to Etnr except that all files are kept for at least two additional days after the day on which they are referenced. Thus,

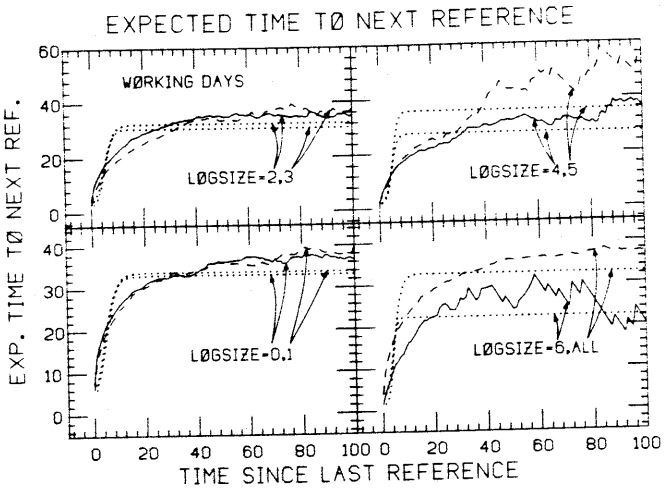
$$K(\text{Etnrb}, Sz, C, P) = \max(3, \{i \mid \min_i[((E(i, L, C) - 1) \times Sz) > P]\}). \quad (12)$$

In Figure 4 we show plots of $E(i, L, *)$ and $Ef(i, L, *)$. Figure 5 displays the values of $E(i, *, C)$ and $Ef(i, *, C)$. The dotted lines in all cases show Ef. There are two important observations to be made from these figures. First, the expected time to next reference is generally increasing with the time since the last reference; thus the desirability of retaining a file will decline with the time since the last reference. Second, the values of Ef are a mediocre fit at best to E. Despite this relatively poor fit, it will be observed when we show our experimental results that the K(Etnrf, Sz, C, P) policy is surprisingly effective.

C. Time and Space-Time Algorithms

All of the realizable algorithms defined thus far (Stochopt, Etnr, etc.) have relied on interreference time distributions for the file reference process. It is also possible to define some algorithms which do not use measured data but which implicitly assume some model of file reference behavior. These algorithms are considered for several reasons. Comparing methods developed

Fig. 4. Plots of $E(i, L, *)$ and $Ef(i, L, *)$.



from the data with either *ad hoc* algorithms or those used elsewhere (e.g., paging) gives some indications of the performance improvement possible. These methods may also be of some interest in systems for which measurements are not yet available. Finally, some of these algorithms appear in the paging literature or are the analogous ones for variable size objects and therefore deserve to be evaluated. In this section, we define algorithms called Working Set (WS), Space–Time Working Set (STWS), and STP**y.

Working Set [6] is that algorithm which removes a file when it has been unreferenced for P or more days. This algorithm was originally designed for main memory paging, and in that circumstance it has been shown to work very well. It has the defect in this case that it takes no account of file size, and thus small files are as likely to be removed as large files. We let "WS" refer to the working set algorithm. Then the policy K(WS, P) may be defined as

$$K(\text{WS}, P) = P + 1. \quad (13)$$

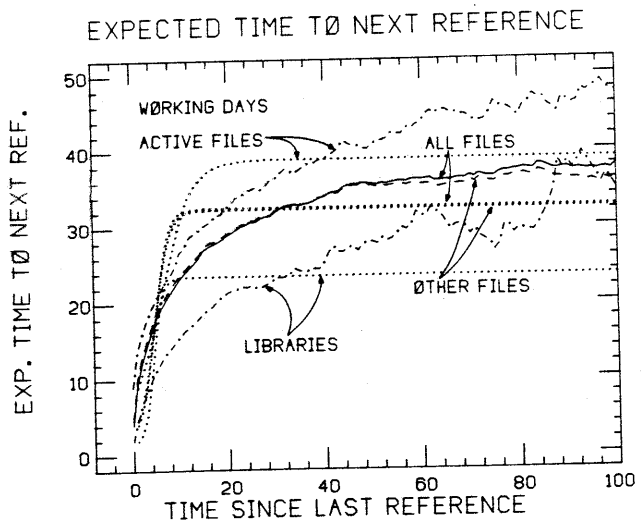


Fig. 5. Values of $E(i, *, C)$ and $Ef(i, *, C)$.

Space-Time Working Set (STWS) is the straightforward and obvious extension of working set to variable size objects; it removes any file for which the product (time since last reference) × (file size) is greater than the parameter P. The implicit assumption here is that the file that is likely to incur the largest cost of retention to the next reference is that which has already accumulated the largest retention cost since the last reference. The policy in this case may be written

$$K(\text{STWS}, Sz, P) = \lfloor P/Sz \rfloor + 1. \quad (14)$$

It can be observed from Figures 4 and 5 that the expected time to next reference climbs (initially) quite steeply with time since last reference. In [24] it was shown that larger files are used more frequently than smaller files. These two facts would suggest that a modification of STWS which weighted time since last reference more heavily than file size should perform better than STWS. We therefore define a class of algorithms which we denote as STP**y (STP stands for space–time product, y is a parameter, and ** is the exponentiation operator in many programming languages). For the algorithm STP**y, the following is computed: [Sz × (time since last reference)**y]; that is, the time since the last reference is raised to the (real valued) exponent y. If the value of the expression given is greater than the parameter P, the file is removed. This policy can be expressed as

$$K(STP^{**}y, Sz, P) = \lfloor (P/Sz)^{**}(1/y) \rfloor + 1. \quad (15)$$

Our selection of exponentiation as the way to increase the weighting of time relative to file size is based simply on convenience; many other functions could have been defined instead. We note that this choice is deliberately *ad hoc*; better results for this set of data could have been obtained by directly fitting (over some class of analytic functions) Stochopt.

D. Bernoulli Process Algorithms

In [26] it was stated that most files observed displayed reference patterns that could be characterized as Poisson. We assume that Stritter treated what was a discrete-time time series as a continuous-time time series. In any case, we found in [24] that of those files testable, the majority could not be characterized as Bernoulli (and presumably therefore not Poisson). Nevertheless, we decided to experiment with some file replacement algorithms based on the assumption that the actual file reference process was Bernoulli for each file (with a possibly different rate for each file). The Bernoulli process (geometric interarrival times) is such that the expected time until the next reference is constant, regardless of the time since the last reference. The replacement decision thus can be made immediately after reference to a file; the file will either be removed immediately (that night) or be kept until the file is used again. The only problem is to estimate the rate at which the file is being referenced, or equivalently, the expected time to the next reference (which is the reciprocal of the rate of reference). The time to next reference was predicted with an exponential estimator [7] using a variety of weights to compute the new value from the old estimate and the most recent interreference period. All algorithms of this class that were tested were found to perform very poorly. This confirms our earlier statistical tests. For brevity, we do not discuss Bernoulli process algorithms further.

VI. Experimental Results

A. Methodology

As noted earlier, our data consist only of one bit for each file for each day specifying whether that file was used that day. Therefore, we have had to do our experiments in a manner compatible with that restriction. Thus we assume the following: all files referenced on a given day are fetched at midnight + ϵ($\epsilon \to 0$) of that day and are retained on the disk until the end of the 24 − 2ϵ hour period at midnight − ϵ. At that time, file migration takes place, and any file which is to be removed is removed at that time. Thus, every time a file is referenced, it remains on a disk for at least one full day. Also, it is possible for a file to be referenced on two consecutive days and still experience a file fault on the second of those days.

Our algorithm for computing $M(A, P)$ and $S(A, P)$ is as given in Eqs. (2) and (3); we considered each interreference interval in turn for each algorithm and parameter value.

B. Miss Ratio Comparisons

Figures 6 through 11 give the performance of each of the algorithms described in the last two sections. Each is discussed below.

Figure 6 shows the behavior of the GOPT, VMIN, STWS, Working Set, and Stochopt algorithms., We see that GOPT, as expected, is the best of all of the algorithms by a substantial margin; it experiences a miss ratio about one-third as high as the best realizable algorithm (Stochopt) throughout much of the range of operation. Conversely, VMIN performs relatively poorly because it does not consider file sizes, even though it is a look-ahead algorithm. Working Set also does very poorly for the same reason. STWS, which does take into account the file size, acts fairly well above about 28,000 tracks but is not very good for smaller space allocations.

The Etnr class of algorithms [Eqs. (7)–(12)] are shown in Figure 7. We see that Etnr(Lsize, class) and Etnr(Lsize) perform well and are very close to Stochopt. Etnr(all files) and Etnr(class) perform relatively poorly. Interestingly, Etnr(class) is not uniformly better then Etnr(all files) and Etnr(Lsize, class) is not uniformly better than Etnr(Lsize). Although it was noted earlier that the Etnr algorithms were not in any sense optimal, it was expected that the more accurate $E(i, -, -)$, the better the performance of the algorithm. We find that

Fig. 6. Behavior of the GOPT, VMIN, STWS, Working Set, and Stochopt algorithms.

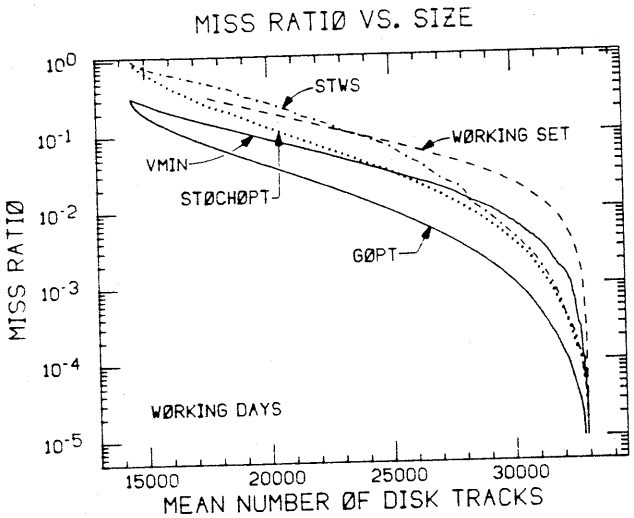
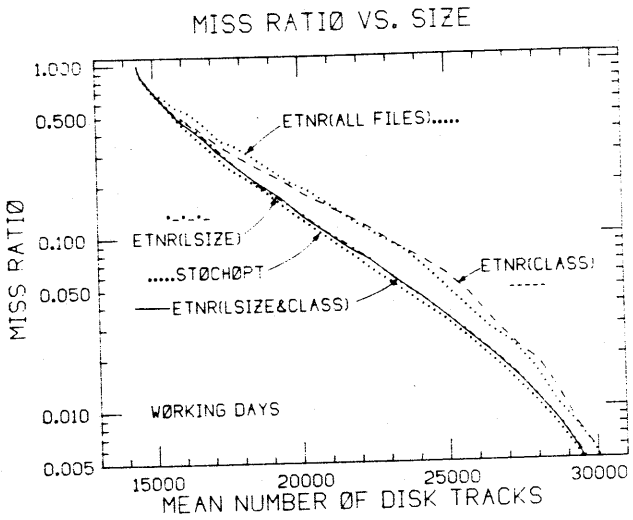


Fig. 7. The Etnr Class of Algorithms.



this is not necessarily so. We also note that the critical item in computing the Etnr policy is the parameter Lsize; the class has little if any effect. This tends to validate our decision to exclude the class in the computation of Stochopt. (Although the functions $E(i, *, C)$ (Figure 5) vary widely with the class. the number of interreference intervals for which $F(i, j)$ switches between 0 and 1 is so small that the miss ratio curves are barely affected, i.e., only about 30 percent of the interreference intervals belong to libraries or active files, and of those, a very large fraction (about 65 percent) are interreference intervals of 1 day to libraries.)

The fitted function $Ef(i, L, C)$ is used for the results presented in Figure 8. Comparing this figure with Figure 7, we see that the use of Etnrf is almost as satisfactory as the use of Etnr despite the poor quality of the fit between $E(i, L, C)$ and $Ef(i, L, C)$. This comparison is shown again directly in Figure 9, where the Etnr(Lsize, class) and Etnrf(Lsize, class) policy results are both given. Their closeness is again evident.

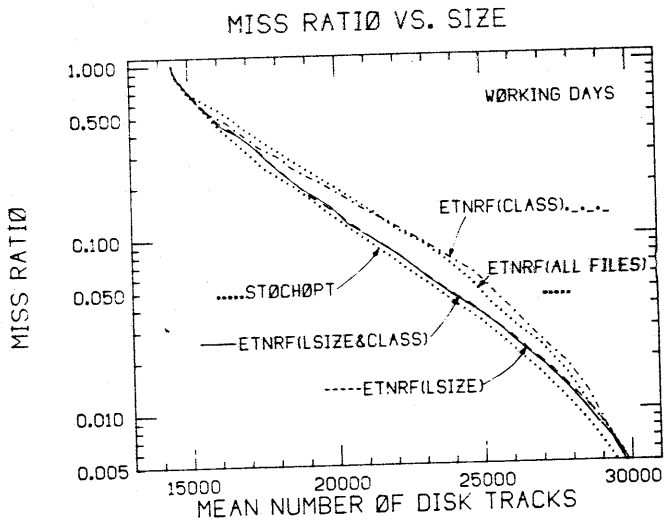


Fig. 8. The Fitted Function $Ef(i, L, C)$ Results.

Figure 9 also shows the performance of Etnrb (Eq. (12)). Keeping all files a couple of days as a minimum appears to be a poor policy; the original method of optimization is better.

A variety of space–time algorithms are compared in Figure 10. As we observed earlier, an algorithm which weighted the time since last reference more heavily than the file size would be expected to perform better than STWS. We tested STP**1.2, STP**1.4, and STP**.8 against STWS (which is STP**1.0), and the results appear in Figure 10. STP**1.4 appears to provide the best performance of any of these algorithms. Additional parameter values were also tested (STP**1.5, STP**1.6, STP**1.3). These three and STP**1.4 all perform about equally well, but none of them seems to be a good substitute for Stochopt.

Creating replacement algorithms on the basis of measured data and then testing them against the same data may cause one to doubt the robustness of our results. A test for the robustness of Stochopt and Etnr(L, C) was

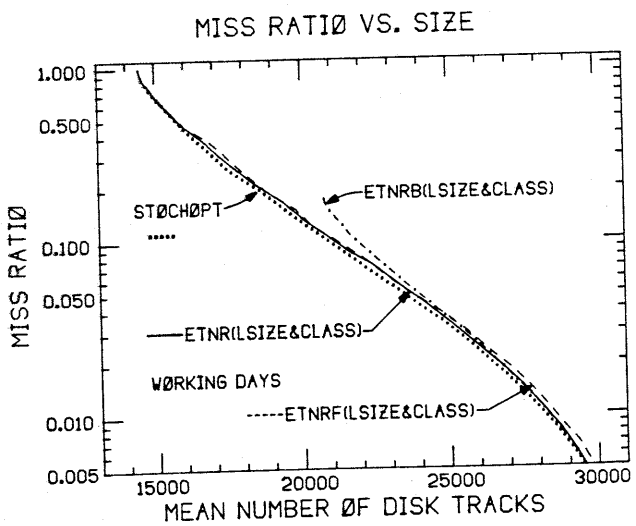


Fig. 9. Performance of Etnrb.

Fig. 10. Space-Time Algorithms.

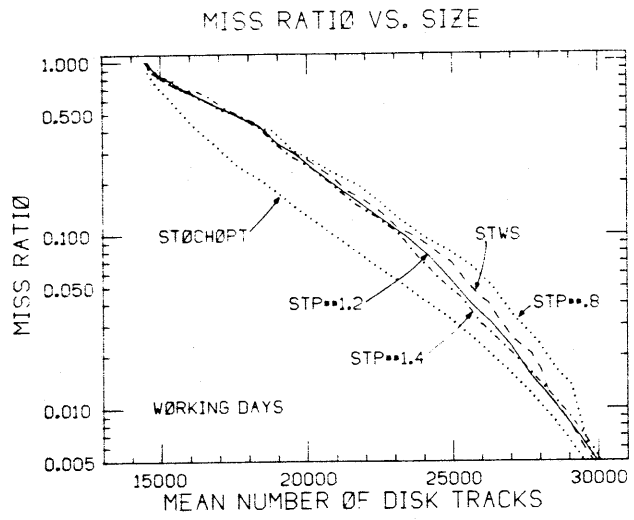
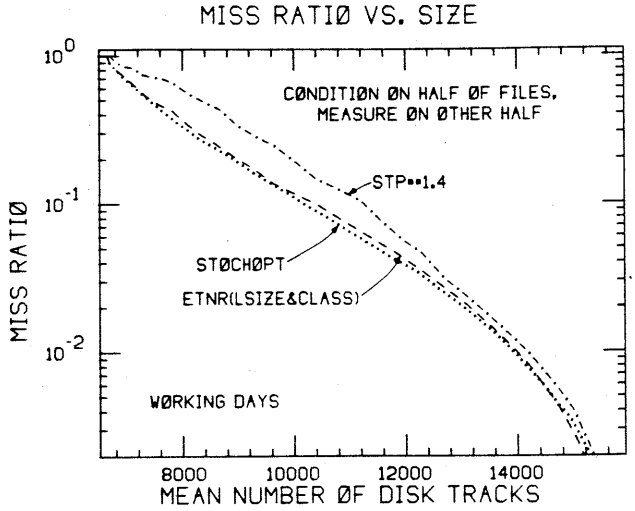


Fig. 11. Results of a Test for the Robustness of Stochopt and Etnr (L, C).



run by creating the Stochopt and Etnr(L, C) policies [Eqs. (4), (9)] using half of the files on the system and then measuring the miss ratio on the other half of the files. This was done, and the results appear in Figure 11, where measurements for Stochopt, Etnr(L, C), and STP**1.4 are given. It can be seen that Stochopt and Etnr continue to perform much better than STP**1.4. (We also observe that Stochopt is no longer uniformly better than Etnr).

C. Parameter Selection

Each of the algorithms presented in this paper contains a parameter, the value of which specifies when to keep or remove a file from the disk. In general, our interest is not in the parameter value per se, but in the parameter value that will yield an acceptable miss ratio. But what is an acceptable miss ratio? The average user who is logged on uses a mean of 3.41 files that day. If fetching a file from mass storage to disk were to take one minute, then a 1 percent miss ratio would imply 0.0341 man-minutes/user/day. A mean of 183.5 users log on per day, so the loss would be 6.26 man-minutes/day. A 10 percent miss ratio would give a figure exactly 10 times as large or a little more than a man-hour per day. The one minute figure may be far too optimistic, however, since it assumes no queueing delays. (The average access time on most mass storage devices, such as the IBM 3850, the Ampex Terabit memory, the CDC 38500, etc., is on the order of 1 minute or less.) For example, at the Lawrence Livermore Laboratory, access times, including queueing and device malfunction delays, can stretch to hours to get a data set off of the photostore and tens of minutes are typical. We note in particular a batch arrival phenomenon, by which most users will log on early in the day and attempt to read several of their files; thus the mass storage device would be very congested early in the day. Our intuitive feeling is that miss ratios on the order of 1 percent to 10 percent would be the maximum tolerable by most users.

Table III. Parameter Values.

	Miss ratio		
Algorithm	1 percent	5 percent	10 percent
Working Set	57	15	7
STWS	540	225	140
Etnr (Lsize & Class)	570	200	110
Etnr (Lsize)	520	200	120
Etnrf (Lsize & Class)	620	210	105
Etnrf (Lsize)	700	210	100
STP** 1.4	1800	460	260

For several of our algorithms, the approximate parameter values that yield 10 percent, 5 percent, and 1 percent miss ratios are given in Table III. Thus the reader wishing to implement one of our algorithms can just use the figures given. The policy K(Sopt, Sz) appeared earlier in Table II.

A more complete mapping of parameter values into miss ratios is given in Figure 12 for the Working Set, STWS, and STP**1.4 algorithms.

D. Variable Space Buffering and Fixed Space Implementation

Figure 13 shows the volume of on-line files vs. time using the working set agorithm with a parameter $P = 30$, the STWS algorithm with a parameter $P = 200$, and the total volume of on-line files in the original, unmigrated system. As is evident, the volume of on-line files varies considerably from day to day and month to month. We note that the figures for Working Set and STWS do not become stationary (i.e., reach warm start) until 30 and 200 days, respectively, from the start of the measurement period. Initially, all files are assumed to be off-line.

The difficulty with this day-to-day variation in the volume of on-line files is that (a) the parameter value used to reduce the number of on-line files may have to change from day-to-day, and (b) enough space has to be left on the disks after a migration run so that user file fetches are unlikely to cause space to run out during the

Fig. 12. A More Complete Mapping of Parameter Values into Miss Ratios for the Working Set, STWS, and STP** 1.4 algorithms.

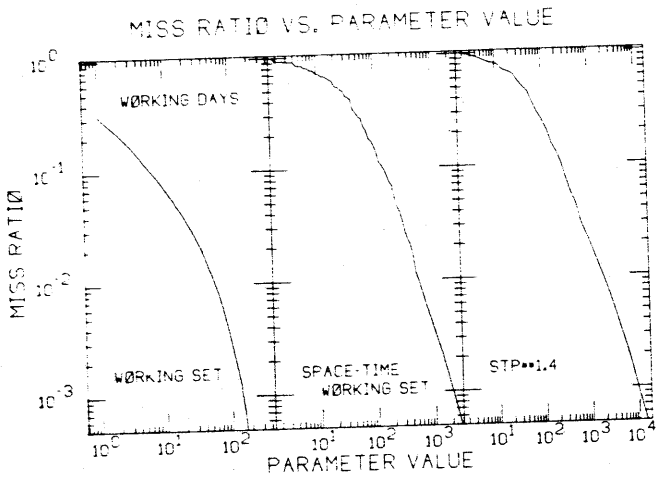
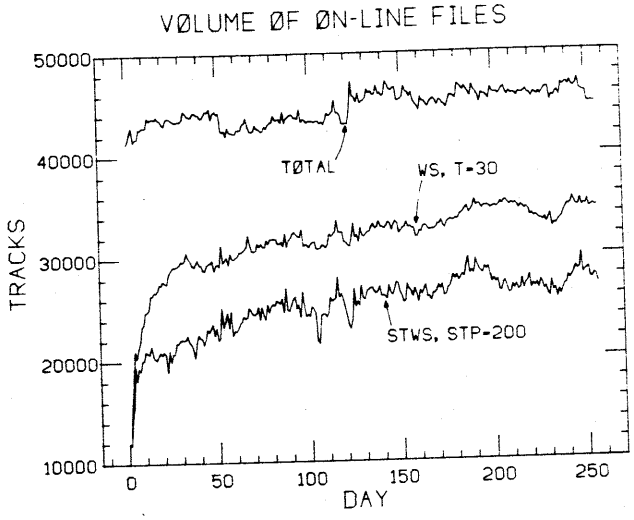


Fig. 13. Volume of On-Line Files vs. Time.



following day. We also note that many of our file migration algorithms favor the larger files for replacement; thus the files fetched are also likely to be large.

In actual fact, computer systems do not have a variable amount of space available for file buffering, but instead have a fixed number of disk tracks available. Therefore, in practice, a migration run would leave only a fixed volume of files, with some extra space available for files fetched or created during the subsequent day. Although the algorithms presented are all variable space and were developed on that basis, each has a straightforward fixed space analogue. Specifically, each algorithm described has a cost: The cost for the Etnr algorithms is the product of file size and expected time to next reference. The cost for Working Set is just the time since last reference, and so on. The cost for Stochopt is obtained explicitly from the dynamic programming formulation, which has been omitted for brevity; see Sec. V.A. A single pass through the list of on-line files can associate the cost for retaining each file (according to the algorithm in use.) The list of files can then be sorted, and the most costly removed.

Variable space replacement algorithms generally imply significant variations in the amount of space in use. Figure 13 shows 10 percent to 20 percent variations in the number of tracks required; measurements of paging algorithms (see [23]) show much greater variation still. Despite the very large variation observed in paging (e.g., + 100 percent, − 50 percent), the performance of the fixed space versions of algorithms (LRU for Working Set) is usually very close. The much smaller variation in space requirements for files should result in little or no performance differences between fixed and variable space algorithms. Direct comparisons, however, have not been made.

Table IV shows the approximate mean volume of the files fetched/day for several of the algorithms discussed and for three different miss ratio values. The volume of files fetched as a ratio to the mean volume of on-line files is also given.

E. Adaptive Algorithms

Different computer systems will have different user communities and different applications running. Therefore, there will be differences in file reference patterns. These patterns may change over time. This suggests that in practice file migration algorithms should be implemented adaptively. All of the good algorithms (Stochopt, Etnr) use cost functions which are parameterized and which are derived from parameterized interreference time distributions. By maintaining a time weighted running average for the interreference times (using exponential weighting [7]), the cost functions and the retention times for files can be recalculated periodically in order to track system workload changes. Evaluation of such adaptive algorithms is left for further research.

VII. Conclusions, Applicability of Results, Future Work

We have presented a methodology for the creation and evaluation of file replacement algorithms. Our approach has been as follows: (1) assume a model in which files are accessed independently, (2) examine file reference patterns and aggregate these data into groups based on file size and class, (3) use these data (in some way) to predict when the file will next be used, and (4) then select for replacement files whose expected cost of retention to next use is high. Not all of the algorithms consid-

Table IV. Tracks Transferred per Day/Fraction of Volume of On-Line Files.

	Miss ratio		
Algorithm	1 percent	5 percent	10 percent
STWS	540/0.020	3500/0.140	5900/0.252
Etnr (Lsize & Class)	600/0.033	1600/0.073	3500/0.200
Etnrf (Lsize & Class)	540/0.019	1400/0.060	3400/0.155
STP** 1.4	300/0.015	2200/0.090	5000/0.220
Stochopt	370/0.013	1440/0.061	2100/0.101

ered have been based on the data; *ad hoc* methods have been used in the less sophisticated algorithms.

Most of the algorithms considered have been of the class that use a reference as a renewal point; that is, history prior to the most recent reference is not used. We found that the Stochopt algorithm which uses the (entire) measured file interreference time distribution, performed well, and the Etnr algorithm, which uses some of this information, did almost as well. Algorithms which used less or no information about file reference patterns generally performed poorly. About the best of the other algorithms was the STP**1.4 algorithm, which might provide an acceptable performance.

The evaluation of the algorithms was done using file reference data taken at SLAC for Wylbur text editor data sets. (As noted earlier, though, this includes essentially all user disk files.) There is no reason to expect that text editor or time sharing data sets (e.g., TSO) would be referenced very differently in another system; we therefore believe that our results are applicable to such systems. Our data, however, are not concerned with large data files, system files, or scratch files, and no conclusions can be drawn about migrating such files based on the experiments described in this paper. Further, our results are for "long term" file migration, that is, migration that occurs over periods of days, weeks, or months. In many systems, it would be necessary for migration to occur over time periods of hours (as is done at Lawrence Livermore Laboratory), and our experiments here provide only methodological guidance for such systems.

An important point which must be noted is that if the user is significantly inconvenienced by file migration (with respect to the availability of his data sets), he will develop methods, such as synthetic file references, to maintain his files on line. This could impair the effectiveness of file migration.

We found that if file migration is properly implemented, it can substantially reduce the volume of on-line files without inflicting an unacceptably high miss ratio on most users.

A number of things have been left for future research. Adaptive algorithms have not been experimentally evaluated, nor have the fixed space analogues of the variable space algorithms discussed in this paper. Algorithms which use the file age intelligently should be studied. Prefetch algorithms, based on more sophisticated policies than the class of demand policies that we consider, may be worthwhile. Placement algorithms can be considered to some extent. Some marginal improvements can be obtained by investigating these additional items; we believe, however, that in this paper we have presented the bulk of the useful information. We very much hope that similar data will become available for other computer systems in order to make comparisons possible and in order to investigate the range of applicability of our results.

Acknowledgment. Many thanks to E. Stritter, who collected these data and made them available to the author for this research.

Received 10/78; revised 2/80; accepted 3/81

References

1. Boyd, D.L. Implementing mass storage facilities in operating systems. *Computr 11,* 2 (Feb. 1978), 40–45.
2. Chaffee, R.B., Challenger, M.A., and Russell, E.S. File migration task force study. Stanford Linear Accelerator Center, June 1977.
3. Chu, W., and Opderbeck, H. Program behavior and the page fault frequency replacement algorithm. *IEEE Computr* (Nov. 1976), 29–38.
4. Coffman, E.G., and Denning, P.J. *Operating Systems Theory.* Prentice–Hall, Englewood Cliffs, N.J., 1973.
5. Considine, J.P., and Myers, J.J., MARC: MVS archival storage and recovery program. *IBM Syst. J. 16,* 4 (1977), 378–397.
6. Denning, P.J. The working set model for program behavior. *Comm. ACM 11,* 5 (May 1968) 323–333.
7. Denning, P.J., and Eisenstein, B. Statistical methods in performance evaluation. Proc. ACM Workshop on Computer Performance Evaluation, Harvard University, Cambridge, Mass., April, 1971, 284–307.
8. Denning, P.J., and Slutz, D.R. Generalized working sets for segment reference strings. *Comm. ACM 21,* 9 (Sept. 1978), 750–759.
9. Proc. DOE/NCAR mass storage workshop, Dec. 1977, National Center for Atmospheric Research, Boulder, Colo., published May, 1978.
10. Fajman, R., and Borgelt, J. Wylbur: An interactive text editing and remote job entry system. *Comm. ACM 16,* 5 (May 1973), 314–322.
11. IBM. MVS hierarchical storage manager release 1 is available. DPD Program Product Announcement, IBM Corp., Armonk, N.Y., April, 1978.
12. Klorer, C.J. MSS/DASD space/dataset management system. Proc. Share 51 Conf., Boston, Mass., Aug. 1978, 1090–1096.
13. Knight, J. CASHEW—A proposed permanent data storage system. Computer Center Rept., Lawrence Berkeley Laboratory, May 1976.
14. LeHeiget, J.P., and Reich, D.L. MSSCOM, A conversational MSS command processor. IBM Res. Rept. RC 7167, Dec. 4, 1978.
15. Lum, V.Y., Senko, M.E., Wang, C.P., and Ling, H. A cost oriented algorithm for data set allocation in storage hierarchies, *Comm. ACM 18,* 6 (June 1975), 318–322.
16. Mattson, R.L., Gecsei, J., Slutz, D.R., and Traiger, I. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2 (1970), 78–117.
17. Michael, G.A. MASS archival storage: Some trends, needs and plans at DOE Laboratories. Lawrence Livermore Lab. Rept. UCRL 82354, May 21, 1979.
18. Morgan, H., and Dan Levin, K. Optimal program and data locations in computer networks. *Comm. ACM 20,* 5 (May 1977), 315–322.
19. Prieve, B.G., and Fabry, R.S. VMIN—An optimal variable space replacement algorithm. *Comm. ACM 19,* 5 (May 1976), 295–297.
20. Reich, D.L. Page fault model of staging for mass storage volumes. IBM Res. Rept. RC 7430, Nov. 30, 1978.
21. Revelle, R. An empirical study of file reference patterns. IBM Res. Rept. RJ 1557, April 1975.
22. Smith, A.J. Analysis of the optimal look-ahead, demand paging algorithms. *SIAM J. Computing 5,* 4 (Dec. 1976), 743–757.
23. Smith, A.J. Bibliography on paging and related topics. *Oper. Syst. Rev. 12,* 4 (Oct. 1978), 39–56.
24. Smith, A.J. Long term file reference patterns and their application to file migration algorithms. IEEE TSE (in press).
25. Smith, A.J. Sequentiality and prefetching in data base systems. IBM Res. Rept. RJ 1743, March 19, 1976, and *ACM Trans. Data Base Syst. 3,* 3 (Sept. 1978), 223–247.
26. Smith, A.J. Bibliography on file and I/O system optimization and related topics. *Oper. Syst. Rev.,* 1981.
27. Stritter, E.P. File migration. Stanford Computr Sci. Rept. STAN-CS-77-594, Ph.D. Disseration, Jan. 1977.
28. Zehab, D., and Boies, S. J. The SFS migration system. IBM Res. Rept. RC 6944, Jan. 1978.