Dynamic File Management Techniques

Milind B. Deshpande *Richard B. Bunt*

Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

ABSTRACT

Consideration of the dynamic characteristics of file referencing behaviour provides an appropriate basis for the design of effective algorithms for the management of a hierarchical file system. Several new algorithms are proposed, based on similar algorithms in the memory referencing context. Trace-driven simulation experiments were conducted to assess the performance implications of these approaches. The results of these experiment show that the dynamic algorithms do achieve their performance objectives. More generally, the results reveal some interesting aspects of the problem and indicate some important differences between the domains of file referencing and memory management.

Key Words and Phrases: operating systems, file systems, storage hierarchies, program behaviour, performance

1. Motivation

In a computer system, files containing programs and data are organized into a logical structure called the file system. To manage the vast amount of data in a file system effectively, storage hierarchies have been intoduced, combining fast but expensive devices with slower and cheaper ones to reduce storage costs and improve access speed. The management of multilevel storage hierarchies deals with the problem of assigning files within a file system to different levels of the hierarchy, based upon expected use.

Since it is economically infeasible to store all files on the fastest device, files that are used most often should be placed on faster devices, while files that are used infrequently should be placed on slower devices. In this way, it is possible for average access time to approach that of the fastest (and most expensive) device while storage cost approaches that of the least expensive (and slowest) device. *File assignment*, also known as *file allocation* or *file loading*, is the problem of assigning files to various levels in a storage hierarchy so as to achieve performance objectives. Changes in file assignment are essential when the usage pattern of the files changes with time and computational loads.

Solutions to the file assignment problem can be classified broadly as *static* solutions or *dynamic* solutions, depending on how frequently the file placement operation is carried out. Changing file referencing patterns and the fluctuating nature of workload suggest that an optimal placement achieved through a static algorithm (e.g., [9] will rarely remain optimal for long. Dynamic algorithms (e.g.,[6]), employing simple heuristic strategies for dynamic file allocation based on frequency of reference, file size, storage capacity of devices, etc., are more robust and can provide near optimal file placement for much of the time, albeit with increased overhead.

While certain "dynamic" algorithms can be applied as frequently as permitted by the overhead costs, they fail to exploit important characteristics that may exist in file referencing behaviour. A comparison with virtual memory management algorithms is appropriate here. It has been observed frequently that the memory reference patterns of executing programs are non-random. Programs tend to refer only a small subset of their pages over a significant execution interval, a phenomenon known as *locality of reference*. This characteristic of memory referencing behaviour has been exploited extensively in the design effective memory management policies such as LRU, PFF [2], and WS [3]. These use the notion of locality to predict the future behaviour of programs and bring into main memory only those pages that are likely to be referenced in the near future. In a similar way it may be possible to base dynamic file assignment algorithms on certain characteristics that may exist in file referencing behaviour. The study of such characteristics may pave the way for more effective file management techniques.

File referencing behaviour can be classified as either *short-term* or *long-term* depending upon the granularity of measurements taken. In recent years researchers have studied both short-term and long-term file referencing behaviour and have observed that they are somewhat similar to memory referencing behaviour. Lawrie [7] and Smith [11, 12] proposed successful file migration algorithms based on the locality principle in long-term file referencing behaviour. More recently, Majumdar and Bunt [8] analyzed locality characteristics of short-term file referencing behaviour and suggested ways to exploit them.

While the locality phenomenon seems to pervade the entire storage hierarchy, it has been applied only at the upper (*i.e.* file migration) and lower (*i.e.* memory management) ends. This paper considers the intermediate range, proposes several algorithms for truely dynamic file assignment, and examines the possibility of exploiting the locality phenomenon in short-term file referencing behaviour for the design of such algorithms.

2. The Experimental Environment

The performance of the dynamic file management algorithms proposed in this paper was assessed by means of a set of trace-driven simulation experiments. The workload traces used to drive these experiments were taken from a DEC VAX 11/750 system running 4.2 BSD UNIX™ in the Department of Computational Science Research Laboratory at the University of Saskatchewan. The workload on the system represents a typical university mix of computational activities such as text processing and program development, as presented by faculty, graduate students, and research staff. Each of the various strategies proposed in this paper was simulated, with input in the form of *file reference strings*, a collection of file system events recorded using a software monitor. In the file reference strings a *file access* refers to a logical file request issued by a program. This definition avoids such system dependent factors as buffering mechanism and block size. Data was captured for the 8 busy hours of a day, during which an average of 7 to 10 users accessed approximately 10 megabytes of data. Since the usage of this particular system changes little throughout the year, it is reasonable to assume that the particular strings used for the experiments represent typical file referencing activity on this system.

For the performance experiments a three-level storage hierarchy was simulated, consisting of a drum, a fast disk and a slow disk. An average

1. Milind Deshpande is presently a Ph.D. student at Rutgers University, New Brunswick, N.J.

2. UNIX™ is a trademark of AT&T Bell Laboratories

access time of 5 msec was assumed for the first level (the drum), with a ratio of 1:5:20 throughout the rest of the hierarchy. These values are typical of devices commonly available and provide an opportunity to quantify the performance results presented. The qualitative nature of the results, however, does not depend on the specific configuration of devices.

3. Characterizing The Experimental Workload

This section examines the file reference strings used in the simulation experiments (See [5] and [13] for more detailed analysis). The purpose of this analysis is to expose various characteristics of file reference patterns, both generally and specifically, which are subsequently exploited in the design and analysis of algorithms for storage hierarchy management. System files are not considered: first, because they represent a small fraction of the total files on the system; and second, because they are heavily referenced, any reasonable approach to dynamic hierarchy management would find it more efficient to assign such files to the fastest device permanently and concentrate instead on assignment of user files.

File Size: The analysis of file sizes (see Fig. 1) shows a heavy skewing towards smaller sizes with a significant percentage of referenced files smaller than 10 Kbytes. This observation is in agreement with the studies conducted by Satyanarayanan [10] in a different environment.

Inter-reference Duration: Inter-reference duration gives the average interval between two successive accesses to the file system and is a measure of how busy the file system is. To be able to service all requests, the average request rate should be lower than the average service rate. Table 1 shows the typical values of average inter-reference duration for user files and Fig. 2 shows how inter-reference duration varies during the eight busy hours of the day. Such analysis can be used to determine if a given storage hierarchy has the capacity to satisfy all the requests presented to it.

Table 1: Statistics On Inter-Reference Duration.

Ref. String	Inter reference Time (in sec.)
apr4.85	0.9275
apr10.85	0.8077
apr24.85	1.3457
apr26.85	0.4177
may3.85	0.3200

Locality: Bunt *et al.* [1] characterized the locality phenomenon in the domain of memory referencing behaviour in terms of *concentration* of reference and *persistence* of reference. Concentration refers to the tendency of a program to refer to a small subset of its pages; persistence is a phenomenon whereby a series of references is directed towards the same page. Both are important to the success of conventional memory management approaches. Analysis of file reference strings has shown that similar properties exist in the context of file references. A summary is provided here; a more complete analysis is found in [13].

a. *Concentration:* Table 2 shows the percentage of files referenced in the reference string that are responsible for 50% and 90% of the total file activity on the system. The percentages are small in every string, with from 4% to 23% of the referenced files accounting for 90% of the references. This tendency for referencing activity to be concentrated to a small fraction of a total file system provides clear motivation for dynamic file assignment.

b. *Persistence:* Persistence of reference can be observed in two ways:

   i. *By examining the distribution of references to various levels of the LRU stack:* Table 3 shows that this distribution is heavily skewed towards the top of the stack, an indication of strongly persistent referencing patterns. Although program reference strings show persistence, the behaviour appears to be even more prominent in file reference strings [13].

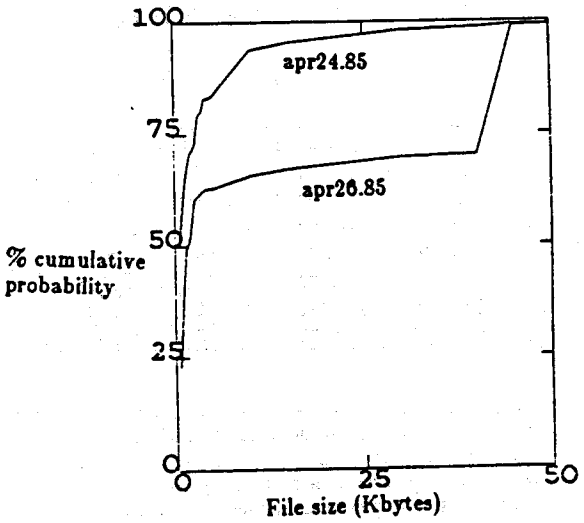

Figure 1: Distribution of References Across File Size Range.

Table 2: Percentage of Files Causing 50% and 90% References (Total).

Ref. String	50%	90%
apr4.85	0.39	16.92
apr10.85	0.97	20.79
apr12.85	0.44	18.46
apr15.85	1.37	23.2
apr19.85	0.57	19.52
apr24.85	0.07	3.94
apr26.85	0.16	7.78
may3.85	0.65	12.96

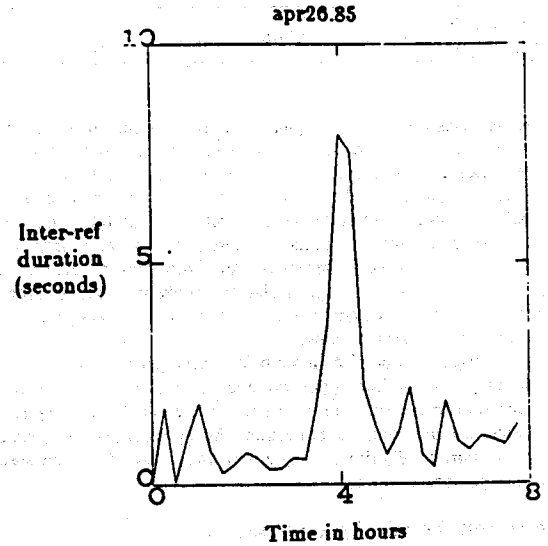

Figure 2: Variation of Inter-reference Distribution With Time.

Table 3: Stack Depth Distribution.

stack depth	apr26.85	apr4.85	may3.85	apr24.85
1	0.87854	0.78448	0.87093	0.87084
2	0.02988	0.05169	0.05171	0.03632
3	0.00819	0.01507	0.01139	0.01127
4	0.00467	0.00752	0.00527	0.00501
5	0.00270	0.00570	0.00297	0.00344
10	0.00127	0.00367	0.00082	0.00240
20	0.00128	0.00091	0.00092	0.00077
100	0.00003	0.00003	0.00004	0.00003

ii. *By studying the distribution of consecutive references to the same file:* Table 4 shows the distribution of frequencies with which certain length sequences of references to the same file occur in a representative reference string. While a file reference string tends to have more short reference sequences than long ones, an average of between 3.5 and 12.1 references (depending on the file system observed) are made to the same user file before another file is referenced.

Table 4: Distribution of Consecutive References for the Ref String Apr26.85.

Number of Consecutive Refs	Cumulative Distribution			
	All	MNT	USR	ROOT
1	0.45673	0.27301	0.32871	0.35261
2	0.70348	0.45724	0.58679	0.60805
3	0.80950	0.67769	0.71988	0.73787
4	0.87341	0.82290	0.83497	0.79643
5	0.90393	0.88306	0.88797	0.82370
10	0.96204	0.95187	0.97120	0.91559
20	0.98178	0.97593	0.98425	0.93896
50	0.99321	0.99369	0.99505	0.97678
100	0.99772	0.99755	0.99928	0.99229
200	0.99830	0.99883	--	0.99423
Ave. length of a string	5.4	4.5	3.5	12.1

Re-referencing Behaviour: Locality-based algorithms owe much of their success in the memory management domain to the fact that programs show a strong tendency to re-reference a small set of pages — the locality set. This provides important stability. Table 5 illustrates the re-referencing characteristics observed in file references. While the probability that a file will be re-referenced within 10 seconds is seen to be very high, it falls off rapidly outside the 10 second interval — much more rapidly than in the memory reference domain. This particular re-referencing characteristic in the file reference domain is significant and will be used to explain some of the results presented in later sections.

In conclusion, while file referencing behaviour seems to have substantial similarities with memory referencing behaviour, re-referencing and persistence characteristics are different. To the extent that these latter characteristics are important determinants, the performance of locality-based file assignment algorithms may differ from that of locality-based memory management strategies.

4. Algorithms for File Assignment

This paper examines the performance of several algorithms for dynamic file management. Three of the algorithms — LRU, WS, and FFF — are natural extensions of the corresponding algorithms from the memory management domain, and are designed to exploit the locality phenomenon

Table 5: Study of File Re-referencing Behaviour.

Time interval (sec.)	Probability that a file will be re-referenced in the given time interval.			
	apr4.85	apr15.85	apr24.85	apr26.85
10	0.562496	0.563980	0.803543	0.785350
20	0.034887	0.048863	0.021227	0.029416
30	0.022278	0.028360	0.013681	0.014912
40	0.018656	0.022220	0.010887	0.009626
50	0.011754	0.016808	0.007546	0.008009
100	0.008952	0.006036	0.002679	0.004340
300	0.001982	0.001873	0.000720	0.001587
500	0.000820	0.001249	0.000806	0.001078

directly. RAND and FIFO are non-locality algorithms. GOPT, an optimal algorithm that uses information about future referencing patterns, and RSA (Random Static Assignment), a static assignment algorithm, provide upper and lower bounds on the performance of file assignment algorithms.

As mentioned, a three-level file storage hierarchy comprising drum, fast disk, and slow disk is assumed for the purpose of discussion. The specific devices in the hierarchy are clearly not important to the discussion in more general terms.

LRU: LRU is extended for managing a multilevel storage hierarchy by considering drum as a repository of the most-recently-used files. A file recently cast out of drum, itself managed by the LRU strategy, is more likely to be referenced in the near future than a file forced out further in the past. This technique effectively partitions the LRU stack into different zones (see Fig. 3) so that the files in the topmost zone can be considered to constitute the current locality set and should therefore be assigned to the drum. Files purged from the drum are assigned to the fast disk. The net result of this strategy is to assign files to different devices based on expected usage. Movement of the files is shown in Fig. 3. When a required file is not found on the drum, it must be fetched to the drum from its device of current residence. Unused files are gradually aged out to the next slower device (a purge). Since each of the devices has a finite storage capacity, a file exceeding available space cannot be stored on that device. This constraint requires that such a file be stored on the next slower device that can accommodate it.

WS (Working Set): The working set policy [3] can be extended for managing a multilevel storage hierarchy by employing a set of nested working set windows for predicting future file referencing patterns (see Fig. 3). Assuming that the window sizes can be adjusted for a given environment, the working set corresponding to the smallest window can be

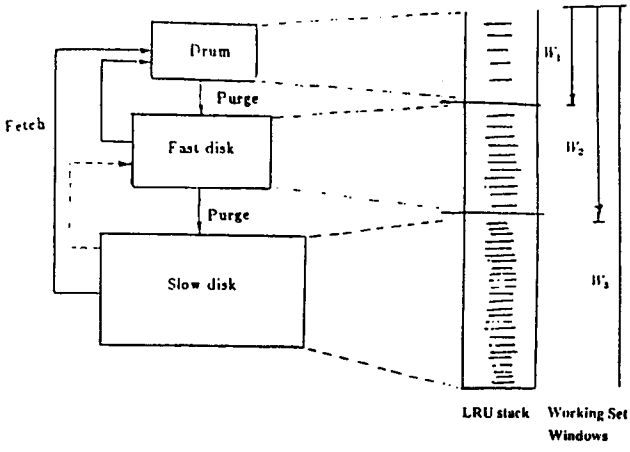

Figure 3: Managing the File Storage Hierarchy

considered as the current locality set and the corresponding files can be assigned to the drum. Files that fall in the next larger window, but outside the smaller one, have moved out of the current working set and therefore, should not be stored on the drum. This strategy results in a set of nested working sets that relate to successively slower levels in the hierarchy.

FFF (File Fault Frequency): The Page Fault Frequency (PFF) memory management algorithm [2] can be extended to the management of a file storage hierarchy by using a set of non-decreasing fff parameters, one for each level in the hierarchy. Files corresponding to the smallest fff parameter constitute the current locality set that can be assigned to the fastest device. The rest of the fff parameters can similarly be employed to assign files to other devices in the hierarchy.

FIFO: In this approach, every file moving to a device is time-stamped and, when space runs out, files are purged to the next slower device on a first-in-first-out basis. A file unavailable on the drum is searched for on successively slower devices, and when found, moved to the drum from its device of current residence. Once again, limited device capacities restrict the movement of files.

RAND: The RAND strategy chooses a file to be replaced simply at random; all the files on a device have equal likelihood of being replaced, regardless of measured (past) or expected (future) usage. A major weakness of this strategy is that it could select a file that is going to be referenced next. In certain situations, however, it will be shown that this strategy can be surprisingly effective.

Generalized Optimum (GOPT): This optimal strategy (from [4]) provides an upper bound on performance. Replacement decisions are based on the size of a file and the time of its *next* reference. Since such replacement is possible only if future referencing information is available, the algorithm is realizable only in simulation. It does indicate, however, how much potential for improvement exists in any realizable approach.

Random Static Assignment (RSA): This algorithm is employed to provide a lower bound on the performance of a storage hierarchy. In this algorithm, files are assigned at random to the various devices in the hierarchy, subject only to the constraint that the device storage capacities are not exceeded. While this algorithm is certainly not typical of actual static assignment algorithms, the data collected serves to indicate potential effects of lack of attention to the dynamic referencing behaviour. While dynamic algorithms tend to give uniform performance at all times, the performance of static (or even semi-static) algorithms degrades with the passage of time. After sufficient duration, the file assignment may become even as bad as random static assignment.

Several additional assumptions are necessary for these experiments. It is assumed that there is no internal buffering mechanism for the filesystem. Every reference to a file is assumed to be a reference to the storage hierarchy. It is also assumed that a file is read or written completely in a single access. The measures used for comparing the performance of algorithms include average time to access a file, hit ratios to devices, and overhead for file transfer within the hierarchy. "Overhead" relates to the cost associated with moving files between devices. The computational costs associated with any algorithm are assumed to be negligible. When files are fetched from slower devices, some files may have to be purged from faster devices to make room for incoming files. This increased file movement is undesirable, both because it requires time to do it, and because it represents increased activity in I/O channels. This is especially so because all file movement between levels is assumed to take place through main memory, using system resources such as CPU and main memory. Overhead cost is quantified in terms of average time spent in purging files for every file access.

5. Experimental Results

Results presented here were obtained from the trace-driven simulation of the various file management algorithms proposed using the various reference strings selected. A great volume of data was generated by these experiments, creating a problem of presentation. Since the analysis of file reference strings has shown that they possess similar characteristics in general, the choice of any particular string was felt to have no qualitative effect on the results. Reference strings collected on April 24th, 1985 and April 26th, 1985, therefore, form the basis of the discussion presented. By using two reference strings in the analysis rather than a single one, greater generality can be shown.

The performance of the various algorithms is compared for different values of storage space at the different levels. A typical experimental strategy is to hold the capacity of one device fixed while changing that of the other. WS and FFF are variable space algorithms, however, and care must be taken to choose proper values of the parameters. Low values of these parameters may mean low average resident set sizes and underutilized devices, whereas large values may cause the device capacity constraints to influence the file replacement strategy excessively. Since both algorithms assume unlimited device capacities, the parameters are chosen such that the average resident set size on each device approximates the space used in the fixed space strategies.

Figure 4 shows a plot of average access time versus drum capacity for the LRU algorithm. Rather than showing absolute values, drum capacity here and throughout is expressed as a percentage of the total file space referenced in the reference string (roughly 10 Mbytes); the 1% figure corresponds therefore to approximately 100 Kbytes. The behaviour exhibited in Fig. 4 is similar to that exhibited in the "parachor curve" familiar in the memory referencing domain. Access time drops quickly as drum capacity increases, until the current locality set is on the drum, at which point no further gain results. Similar results have been observed for other algorithms. It is noteworthy that the size of the locality set on the drum is quite small — about 0.1% of the total number of files referenced in the string (amounting to about 10 Kbytes). Some of the characteristics of the environment in which the reference strings were collected are responsible for this behaviour. In particular, 60% of the references in the string apr26.85 are directed towards files of size less than 10 Kbytes (see Fig. 1). This observation partly explains the curve in Fig.4 because when the drum size is 10 Kbytes, almost all the heavily referenced files (*i.e.*, files smaller than 10 Kbytes) can be moved to the drum. This is particularly effective because of the persistence tendency. After the first reference to a file, if the file can be transferred to the drum, all successive references to that file are directed to the drum, thereby reducing the average access time. In practice, the drum size of 10 Kbytes does not represent a locality-set size but rather a filter level which dictates which files should be allowed to come to the drum. Thus, the average access time curve is closely related to the distribution of references across the file range (see Fig. 1).

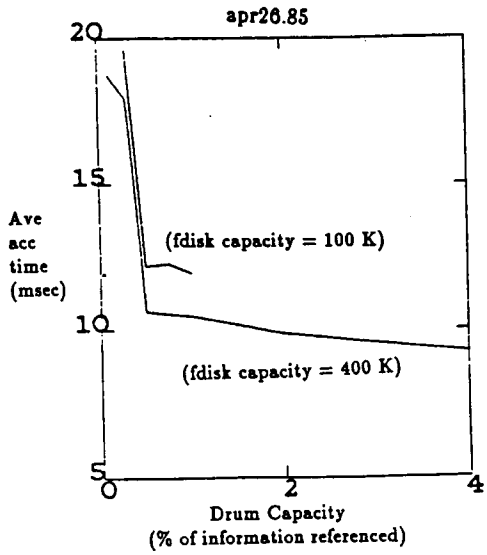

Figure 4: Variation of Average Access Time With Drum Capacity (LRU)

5.1 Performance of Locality-Based Algorithms

Apart from minor differences in file accessing overhead, all the locality-based algorithms (LRU, WS, and FFF) exhibit similar performance. As shown in Fig. 5(a), all three algorithms considered, perform well at small drum sizes (as small as 10 Kbytes). The comparison of overhead (Fig. 5(b)) shows that LRU has slightly lower file transfer activity. This is because file movement under the other algorithms is governed both by the replacement algorithm and by the device size constraint, whereas in the case of LRU it is controlled solely by the device size. The application of dual constraints generates more file activity in some algorithms, and this overhead is likely to deteriorate access time performance to some extent. In the experimental environment the adverse effect of overhead will be minimal because the inter-reference duration is sufficiently larger than average access time to allow the file transfer activity to take place in the background.

Figure 5(c) shows the variation of device hit ratios with varying drum capacity for the LRU algorithm. Similar behaviour was observed for other algorithms as well. While the hit ratios for the drum and fast disk show complementary variation, the hit ratio for the slow disk remains almost unchanged. For lower drum capacities, files that cannot be accommodated on the drum are stored on the fast disk thus directing many of the references to the fast disk. As the drum capacity increases, the active files move from the fast disk to the drum, causing changes in the hit ratios of these devices. Since most of the references to the slow disk consist of references to large files that cannot be moved to fast devices, the hit ratio to the slow disk remains unchanged with the variation in drum capacity.

This redundancy of fast disk for higher drum capacities was quite unexpected since the locality-based algorithms were designed to make use of successively slower devices for storing files that have progressively lower probability of being referenced in the near future. Since most of the references are directed to files smaller than 10 Kbytes and the referencing patterns show strong concentration and persistence aspects, it is possible that only 4% to 5% of the total information accessed in a day is actually needed at any point in time. Once this information is transferred to drum, the fast disk is mostly redundant. This is an interesting result for environments with little fast storage compared to the volume of data accessed in a day.

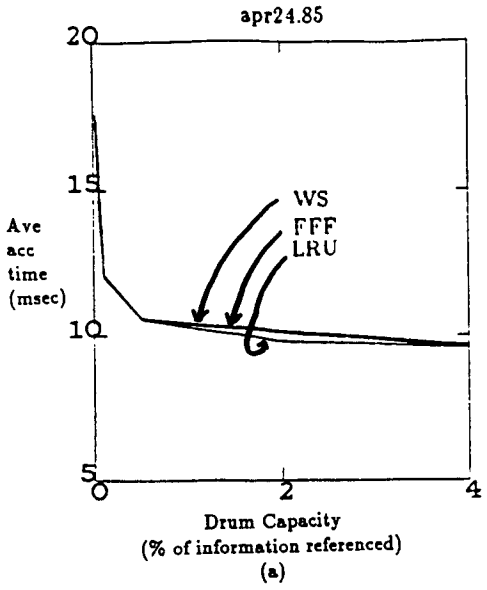

Figure 5: Comparison of Locality-Based Algorithms (Fdisk capacity of 400 Kbytes is assumed).

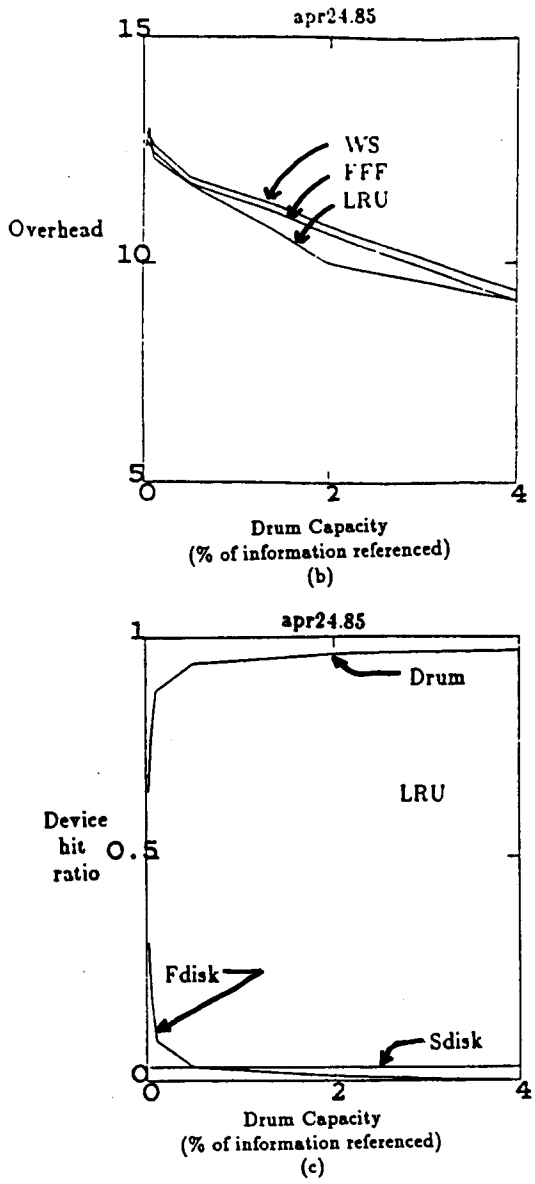

Figure 5: Comparison of Locality-Based Algorithms (Fdisk capacity of 400 Kbytes is assumed). (cont.)

5.2 Performance of the Non-Locality Algorithms

A comparison of the performance of the RAND and FIFO algorithms shows FIFO to be only marginally superior to RAND in all respects (see Fig. 6). To understand the surprisingly strong performance of RAND, it is necessary once again to look at the characteristics of the file referencing behaviour in the experimental environment. As has been shown, the referencing patterns are marked by long sequences of references to the same file and very little re-referencing of active files. This creates a situation where only a small number of the files resident on the drum are being heavily referenced at any given moment. When a file fault occurs, if RAND replaces a file that will not be referenced in the near future, RAND cannot perform worse than FIFO. On the other hand, even if RAND removes a file that will be referenced in the near future, it does little harm because it is highly probable that the file will experience a long sequence of references, of which only the first one will cause a file fault. Thus, in the long run, the percentage of references that cause file faults under RAND is

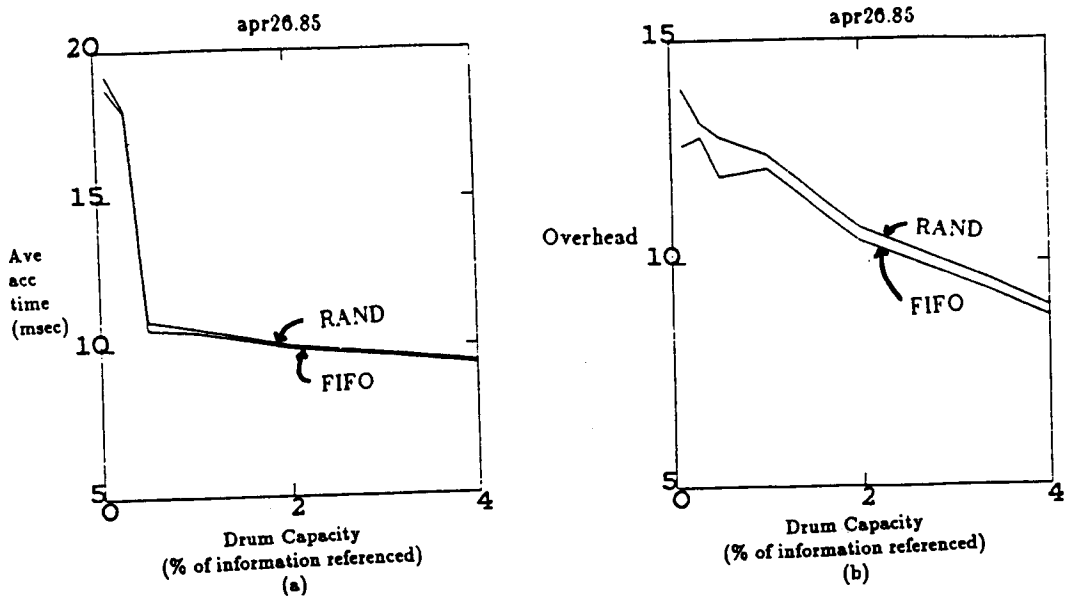

Figure 6: Comparison of Non-Locality Algorithms (Fdisk capacity of 400 Kbytes is assumed).

not far different from that under FIFO, or for that matter, under any of the locality-based algorithms. With stronger re-referencing characteristics the situation would be different.

5.3 Performance of the Optimal and Static Algorithms

As mentioned earlier, the GOPT and RSA algorithms give upper and lower bounds, respectively, on the performance of file management algorithms. A large difference between the optimal performance and the performance of the dynamic algorithms would indicate substantial scope for improvement through the use of dynamic algorithms. Figure 7 shows a comparison of average access time for the GOPT, LRU (chosen as a representative dynamic algorithm), and RSA algorithms. Although the optimal algorithm is clearly superior to LRU, the narrow margin is quite surprising. It can be explained by the same reasoning used to explain the strong performance of RAND. The RSA algorithm, on the other hand, performs poorly. Since RSA represents a static assignment it is unable to match the capabilities of the high performance devices to the dynamics of the referencing behaviour. This is clearly observed in Fig. 7 and presents a strong argument in favour of the dynamic approach.

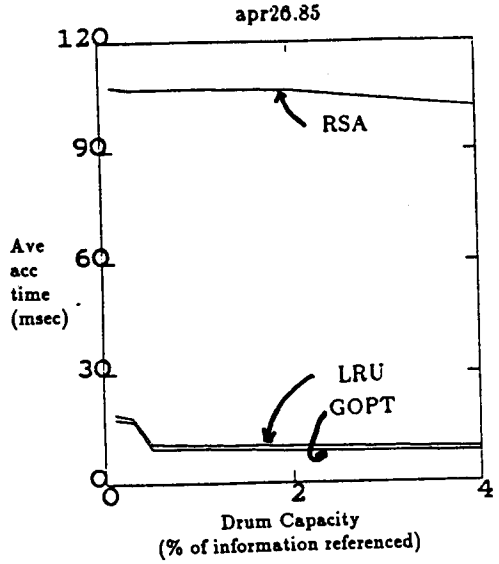

Figure 7: Comparison of RSA, LRU and GOPT (Fdisk capacity of 400 Kbytes is assumed).

5.4 Summary: Comparison of All Algorithms

In general, all the dynamic algorithms examined improve the average access time of file requests in a hierarchical file system while maintaining low storage costs. This performance is summarized in Fig. 8. Similar performance was observed on other file reference strings as well. The optimal algorithm, GOPT, gives the best performance, but the realizable dynamic algorithms are not far behind. The static algorithm, RSA, performs quite poorly. With a dynamic approach in the enviorment considered it is possible to keep the average access time to about 9 msec (nearer the 5 msec access time of the drum than to the 25 msec access time of the fast disk) by keeping as little as 4% of the information referenced in a day on each of the drum and fast disk. Thus, dynamic file management algorithms are able to approximate the performance of the fastest device in the storage hierarchy while maintaining a very low storage cost. Actual cost comparisons have not been shown.

The strong performance of the dynamic algorithms, RAND and FIFO included, can be attributed directly to the existence of certain aspects of the locality phenomenon in the file reference strings. Even RAND performs well. Although it replaces files at random, it does succeed in bringing the currently referenced files to faster devices, and thereby capitalizes on the tendency to reference the same files successively (persistence of reference). If the file references were not localized by the way of persistent references, neither the locality-based approaches nor RAND could be expected to do well. It should be noted that RAND and FIFO may not perform as well as the locality-based strategies if the environment has strong re-referencing characteristics.

There is one significant difference between locality-based memory management and locality-based file assignment. In a typical multiprogrammed environment, variable-space memory management algorithms such as PFF and WS are generally superior to fixed-space strategies such as LRU because they reduce the resident set size of one program to allocate the same space to another program and thereby reduce overall page faults. If there is only one program running in the system, however, LRU represents the best memory management approach because it uses a fixed amount of memory most effectively. Similarly, the prime

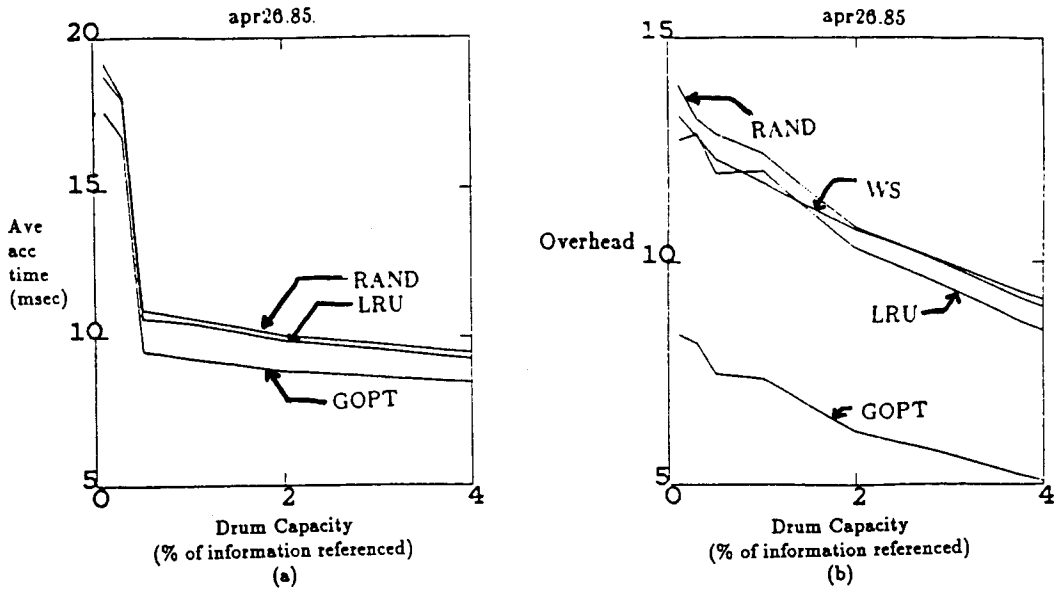

Figure 8: General Comparison of Dynamic Algorithms (Fdisk capacity of 400 Kbytes is assumed).

objective of a system-wide file management algorithm should be to maximize the utilization of fixed device storage capacities. In this case LRU again becomes the logical choice. While the FFF and WS file management strategies give good access time performance they tend to underutilize the device storage capacities. If these algorithms are operated with parameters chosen to maximize the device utilization, however, their performance approaches that of LRU.

6. Conclusions

This paper has examined several algorithms for the dynamic management of a file system residing on a hierarchy of storage devices. Some of these strategies attempt to exploit the locality properties exhibited in file referencing behaviour. It appears that a storage hierarchy, even one with small drum sizes, will perform well if there is some dynamic mechanism to bring the currently active files to the fastest devices and, at the same time, move out less active files in order to clear the faster devices. The unexpected result that the locality-based algorithms performed no better than the non-locality approaches can be attributed to the nature of the environment in which the experiments were conducted (and trace data was collected). While the locality-based strategies may perform well in other environments, characterized by more file re-referencing activity, the same cannot be said about the non-locality approaches.

While the experiments reported were based on a particular configuration of a file storage hierarchy, the results are clearly applicable more generally. File caches for example, are a natural extension. File allocation in a distributed system is likewise a similar problem. Although the specific algorithms may change, it is clear that whatever decisions are made should be based on the dynamic characteristics of file referencing behaviour.

7. References

[1] Bunt, R.B., J.M. Murphy and S. Majumdar, "A Measure of Program Locality and Its Application", *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Cambridge, Massachusetts, August 1984, 28-40.

[2] Chu, W.W. and H. Opderbeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm", *Computer*, Vol. 9, No. 11, November 1976, 29-38.

[3] Denning, P.J., "The Working Set Model of Program Behavior", *CACM*, Vol. 11, No. 5, May 1968, 323-333.

[4] Denning, P.J. and Slutz, D.R., "Generalized Working Sets for Segment Reference Strings", *CACM*, Vol. 21, No. 9, September 1978, 750-759.

[5] Deshpande, M.B., "Locality-Based Approaches to Dynamic File System Management", M.Sc. Thesis, Department of Computational Science, University of Saskatchewan, 1985 (available as TR. 85-14).

[6] Foster, D. and J.C. Browne, "File Assignment in Memory Hierarchies", *Modelling and Performance Evaluation of Computer Systems*, (E. Gelenbe, ed.), North Holland Publishing Company, 1976, 119-127.

[7] Lawrie, D.H., J.M. Randal and R.R. Barton, "Experiments With Automatic File Migration", *Computer*, Vol. 25, No. 7, July 1982, 45-55.

[8] Majumdar, S. and R.B. Bunt, "Measurement and Analysis of Locality Phases in File Referencing Behaviour", *Proc. Performance '86 and ACM SIGMETRICS 86 Joint Conference on Computer Performance Modelling, Measurement and Evaluation*, Raleigh, North Carolina, May 1986, 180-192.

[9] Ramamoorthy, C.V. and K.M. Chandy, "Optimization of Memory Hierarchies in Multiprogrammed Systems", *JACM*, Vol. 17, No. 3, July 1970, 426-445.

[10] Satyanarayanan, M., "A Study of File Sizes and Functional Lifetimes", *Proc. Eighth ACM Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981, 96-108.

[11] Smith, A.J., "Analysis of Long Term File Reference Patterns and Their Applications to File Migration Algorithms", *IEEE-TSE*, Vol. SE-7, No. 4, July 1981, 403-417.

[12] Smith, A.J., "Long Term File Migration: Development and Evaluation of Algorithms", *CACM*, Vol. 24, No. 8, August 1981, 521-532.

[13] Williamson, C.L. and R.B. Bunt, "Characterizing Short Term File Referencing Behaviour", *Proc. Fifth Phoenix Conference on Computers and Communications*, Phoenix, Arizona, March 1986, 651-660.

3. Financial support for this work was provided by the Natural Sciences and Engineering Research Council of Canada, through operating grant number A3707, and by the University of Saskatchewan.