Multiple Prefetch Adaptive Disk Caching

Knut Stener Grimsrud, James K. Archibald, *Member, IEEE*, and Brent E. Nelson, *Member, IEEE*

***Abstract*—A new disk caching algorithm is presented that uses an adaptive prefetching scheme to reduce the average service time for disk references. Unlike schemes which simply prefetch the next sector or group of sectors, this method maintains information about the order of past disk accesses which is used to accurately predict future access sequences. The range of parameters of this scheme is explored, and its performance is evaluated through trace driven simulation, using traces obtained from three different UNIX minicomputers. Unlike disk trace data previously described in the literature, the traces used include time stamps for each reference. With this timing information — essential for evaluating any prefetching scheme — it is shown that a cache with the adaptive prefetching mechanism can reduce the average time to service a disk request by a factor of up to 3, relative to an identical disk cache without prefetching.**

***Index Terms*—Caches, disk caches, input/output, performance analysis, prefetching, trace driven simulation.**

I. Introduction

THE trend in computer development is for CPU speeds to double every few years. While CPU speeds seem to be growing exponentially, the improvements in the access times of mechanical disk drives have not improved significantly in the past years. In order to more closely match the bandwidths between the various layers in the memory hierarchy, caches have been incorporated in nearly all computers manufactured today. Continued progress in caching techniques is needed in order to keep pace with CPU improvements.

Disk caches serve as a buffer between the computer and the physical device as illustrated in Fig. 1. The disk cache may be implemented in the computer as part of the operating system, or it may be part of the disk controller. The speed of the cache memory used for the buffers is not critical as practically any memory technology will be much faster than a physical disk access. If the majority of the references to disk can be serviced by the disk cache, the I/O delay will be significantly reduced.

Caches, whether disk caches or memory caches, are typically organized as a number of equally sized slots which store recently used portions of the physical disk. The design parameters of such a cache include: cache size, slot size, placement policy (what data goes where in the cache), and replacement policy (which data are removed when the cache is full).

In addition to improving performance by optimizing the parameters above, the performance can be also be improved through the use of *prefetching*. If the cache can accurately predict future references and load the appropriate data from the physical disk ahead of time, the performance can be improved. If the accuracy of prefetching is poor, the performance can actually decrease due to cache pollution and channel congestion problems as described by Smith [13].

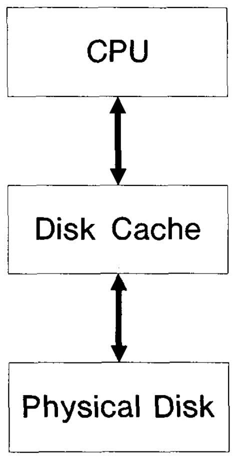


Fig. 1. Disk cache placement in a computer system.

In general, prefetching is done in the background. That is, the file system would initiate prefetch requests as a part of its normal operation. These prefetch requests would be satisfied asynchronously by the I/O subsystem as are all other requests it services. As with all caching schemes using prefetching, care is taken to keep track of outstanding prefetch requests until satisfied.

Most cache prefetching approaches use sequential prefetching either through explicit prefetches of successive blocks of data or through the use of large cache block sizes. In contrast, the method proposed here maintains an adaptive table of most probable successors (children) for each disk block. Each successor is tagged with a weight which indicates the likelihood that it will be referenced given that its parent is referenced. This table and the associated weights are used to control the prefetch mechanism. Unlike sequential prefetching, this algorithm functions well when logically successive disk blocks are not physically adjacent on the disk.

The remainder of this paper is organized as follows: Section II summarizes previous work in disk caching. The new adaptive prefetch algorithm is presented in Section III, and its performance is evaluated and compared to other techniques in Section IV. Conclusions and areas for future work are discussed in Section V.

Manuscript received December 27, 1989; revised August 10, 1990. This work was supported by the Department of Electrical and Computer Engineering, Brigham Young University.

The authors are with the Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah 84602.

IEEE Log Number 9205831.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

II. Previous Work in Disk Caching

Smith's comprehensive paper on cache memories [13] explores the many design parameters of caches and uses trace driven simulation to show how each can impact performance. Disk caches are generally much larger than memory caches and often have a lower hit rate (due to lower locality of reference) but many of the same design principles apply.

A more recent paper by Smith explores the design parameters for disk caches [14]. The paper uses trace driven simulation to show, among other results, that cache sizes on the order of 8 Mbytes can service 80–90 % of all disk requests. In addition to other issues, Smith explores a simple prefetching strategy of loading block $i+1$ into the cache when block i is referenced, but concludes that it is not uniformly effective for all types of files. In contrast with the trace data that we use in our study, Smith's traces did not include *timing information* recording the time when each request is generated. This information is critical in the evaluation of a prefetching scheme such as the one we present in this paper, since prefetching may congest the I/O channel if other disk requests arrive before the prefetches complete. Because of this congestion, the performance of some systems may actually decrease with the inclusion of a prefetching cache.

He has also evaluated variations of this sequential prefetching scheme. Smith has examined the sequentiality and suitability of prefetching in database applications [12] as well as a similar approach for prefetching to cache and main memory [11]. Like the prefetching technique described by Cray Research [4], these extend simple look-ahead to prefetch a variable number of sequential blocks past the current reference.

The study described by Buzen [3] evaluates the performance of a cached storage controller under a range of operating conditions. It shows, for example, that the response time for I/O operations can be reduced by as much as a factor of two if the cache hit ratio is high enough; and if the hit ratio is low, response time can be much worse than a noncaching controller. The paper does not evaluate or propose new cache organizations or management strategies.

Elaborate LRU-based replacement algorithms are described by Hugelshofer [7], but the increase in algorithm complexity is generally not justified by the small increase in performance that results.

A fetching algorithm for memory caches that does predictive loading based on historical information is described by Baer [2]. The reported performance improvement over the same cache without prefetching is limited to about 16 %.

The designers of the Sprite network file system have proposed an algorithm [9] which adapts its parameters to the current characteristics of the system. This adaptive algorithm does not necessarily maximize performance; instead, it seeks to maximize resource utilization.

As pointed out by Bach [1], a relatively common approach in current systems is to cache only those disk areas that exhibit high degrees of temporal locality and to use conventional I/O techniques for the rest of the disk. This technique will yield high hit rates for those areas, but the overall system performance improvement may be small unless accesses to the cached areas constitute a majority of the total disk accesses.

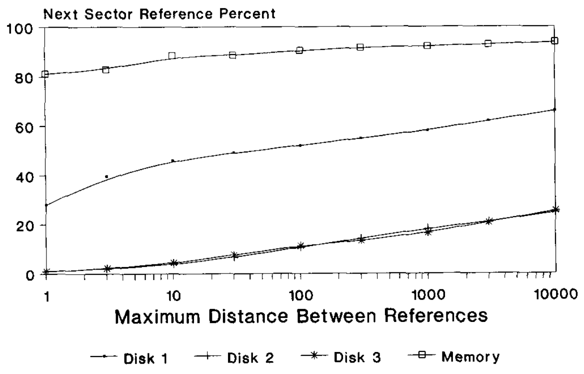


Fig. 2. The success rate of single sector look-ahead. Up to 50% of the time, the next sector will be referenced within the next 100 disk operations.

Previous work in disk caching has often assumed a locality of reference very similar to that of memory references generated by the CPU. Our work is based on the assumption that they are actually quite different. The spatial locality of the two is compared in Fig. 2 using trace data taken from the machines described later in this paper. The figure shows the percentage of time that the next sector (or next location, in the case of memory references) is referenced within the specified number of subsequent references. For example, the next sector will be referenced at most 50 % of the time within the next 100 disk operations, while 40 % of the time it will not be referenced within the next 2000 disk accesses. Because of the poor spatial locality exhibited by disk references, simple prefetching schemes which rely on spatial locality, such as sequential look-ahead, may actually degrade performance by congesting the I/O channel and polluting the cache by loading in blocks that are never referenced. The memory references in this figure were obtained using the tracer utility described by Henry [6].

Note that this lack of sequentiality of disk blocks does not contradict the conclusions of Ousterhout [10] that there is a high degree of sequentiality within *files* in the UNIX file system. According to this paper, "more than two-thirds of file accesses are whole file transfers," but it also states that "most sequential runs are short, rarely more than a few kilobytes in length." In other words, the high sequentiality within files does not imply a highly sequential access of disk blocks because the length of many files is less that the size of a disk block. Ousterhout's study takes the timing of references into account, but the traces used are at the *file* level and not at the *block* level. The traces record only file opens, closes, and seeks — not the actual disk blocks referenced.

To contrast our work described in this paper with the work discussed previously, the following major points are noted.

1) We propose a fundamentally new prefetching mechanism which may be used in conjunction with proposed cache organizations. The scheme does not assume high spatial or temporal locality, and it dynamically adjusts to changes in disk block reference patterns.
2) Block level disk traces with timing information were

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

taken from 3 minicomputers over extended periods of time during the execution of actual workloads. Trace driven simulation with timing was used to evaluate the impact of the proposed prefetching mechanism.

3) The performance gains of the prefetching approach are substantial; the average length of time to service disk references can be reduced by as much as a factor of 3 over the same cache without any prefetching.

III. A New Approach: The Adaptive Mechanism

This section describes a new prefetching algorithm that results in better performance than previous approaches. The fundamental difference with this cache design is that it uses an adaptive mechanism for accurately predicting future disk accesses.

A. Prefetch Motivation and Theory

The performance gain possible through buffering accessed data is limited by the temporal reference locality of the accesses made to that data. Although a few sophisticated replacement policies have been proposed (e.g., see [7]) which focus on making the best choice when selecting a block to be removed from the cache, they are usually variations of an LRU cache. To increase the performance further, emphasis must be shifted from the replacement policy to the fetch policy, specifically through prefetching.

Prefetching can be done in various ways. Increasing the slot size of a cache accomplishes prefetching to a degree. Other methods of prefetching, such as single sector look-ahead and track buffering (where an entire disk track is loaded when a portion of it is referenced) have also been attempted, but the resulting performance improvement has not been significant. In some cases such prefetching actually degrades performance as Buzen [3] has shown.

With little spatial locality, next sector prefetching becomes ineffective since the "guess" of the next sector is not very accurate. If we knew with a degree of certainty what the next disk references for a given process would be, we could effectively prefetch those disk blocks for a substantial performance increase. The problem lies in accurately predicting the next disk references. As Fig. 2. shows, a method more sophisticated than single sector look-ahead or track buffering is needed.

Our method for accurately predicting the next disk references relies on information of the past disk reference patterns and adapts dynamically as the reference patterns change.

B. The Adaptive Table

In order to increase the accuracy of the prefetching, the sequence of fetches performed in the past must be analyzed. Ideally, for each cluster of disk storage (where one cluster corresponds to one slot in the cache and may comprise one or more disk blocks), we would like to keep a record of the clusters which were accessed immediately following it in the past. If we wish to record all previous disk accesses, this results in an unwieldy amount of information, part of which must be updated at each disk reference.

Index	Next Cluster	Weight
0	Next Block # from 0	Weight
1	Next Block # from 1	Weight
2	Next Block # from 2	Weight
3	Next Block # from 3	Weight
	: :	: :
n-2	Next Block # from n-2	Weight
n-1	Next Block # from n-1	Weight

Fig. 3. The adaptive table has a most probable next cluster associated with each cluster on the disk.

Index	Next Cluster	Weight
0	3	10
1	10	3
2	5	1
3	4	5
4	2	6
5	6	10
6	1	2
	: :	: :

Fig. 4. An example of an adaptive table fragment. Cluster number zero has cluster number 3 as its most probable next cluster. The weights range from 0 to 10.

Our approach uses a heuristic mechanism to predict the next most probable disk access using the most recent reference history. The scheme uses a large table structure with one entry for each of the clusters on the disk. The organization of the table is shown in Fig. 3.

The table is directly accessed using the current cluster number as an index. In the simplest form of the table (as shown), each of these entries contains two fields: *nextcluster*, and *weight*. Given a disk reference, the best candidate for prefetching is *nextcluster* in the corresponding table entry. The origin of the data in these entries is discussed in Section III-C.

An example will help clarify the operation of the adaptive table. Consider the table fragment in Fig. 4. Given that the *weight* ranges from 0 to 10, if the current disk reference is to cluster number 0, the probability is very high that the next reference will be to cluster number 3. If cluster 3 is not in the cache, preloading of that cluster should be initiated. Given that the next reference is indeed to cluster number 3, the probability is moderate that the next reference will be to cluster number 4: a prefetch of that cluster may or may not be worthwhile. In practice, the cache should initiate a prefetch only if the *weight* is above a certain *fetch threshold*. The choice of fetch threshold is an important design parameter and is discussed in Section III-D.3).

C. Table Adaptation

The data in the adaptive table is not static; as the name implies the data are constantly adapted to the disk accesses

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

that are made. This section describes the algorithm used to compute and update the table contents.

1) Basic Table Adaptation : If we assume that the table is already filled with entries (possibly incorrect as in the case of initial start-up), the following algorithm governs the adaptation of the table. Note that the algorithm is simple and places little additional computational demands on the processor.

> Given **current** and **previous** disk cluster accesses, go to the row in the table indexed by the cluster number of the **previous** access.

1) If the *nextcluster* field points to the **current** cluster, increase the *weight* field.
2) Else, if the *nextcluster* field does not point to the **current** cluster and the *weight* field is zero, place a pointer to the **current** cluster in the *nextcluster* field and increase the *weight* field.
3) Else, (the *nextcluster* field does not point to the **current** cluster and the *weight* field is nonzero) decrease the *weight* field.

Using this algorithm, the adaptive table will contain entries for the most probable next cluster references for each cluster on the disk. Since it continually records the reference patterns, this algorithm adapts to data migration, changes in the files, and changes in the access patterns to the disk.

In order to curb uncontrolled growth of the weight field, a *weight ceiling* is used. Note that if the ceiling is too high, adapting to disk changes will be slow, as the weight for a given entry must be decreased to zero before new entries may be inserted in the table. If the ceiling is too low, it will respond too readily to anomalies in the reference stream.

In practice the weight ceiling could be normalized to any convenient value. For our simulations, we chose 10 as the weight ceiling with 10 gradations in the weighting function.

2) Multitasking Adaptation: In a multitasking environment, the stream of disk accesses for a given process will be intermixed with requests from other processes. In some systems, the streams can be separated if the PID of the requesting processes are associated with the individual disk requests. However, the disk reference streams coming from the tasks cannot, in general, be separated as in the case of a cache implemented in the disk unit itself since the job process information is typically not available.

The requests from other processes intermixing with the requests of a given process may be thought of as interference or noise. In order to prevent the adaptive table from holding entries which refer to the noise instead of the actual reference stream, simple noise rejecting algorithms may be used to stabilize the pointers.

The noise has a very random characteristic as there is no correlation between the request stream of one task and the requests of the others. If there is a correlation between a reference stream and its noise, then it would be beneficial to prefetch these extraneous blocks and keep pointers to them. In this case the interference can no longer be considered noise. If the noise is random, the adaptive table will not significantly respond to it since the weight value for an entry pointing to an extraneous cluster will not be incremented — this requires

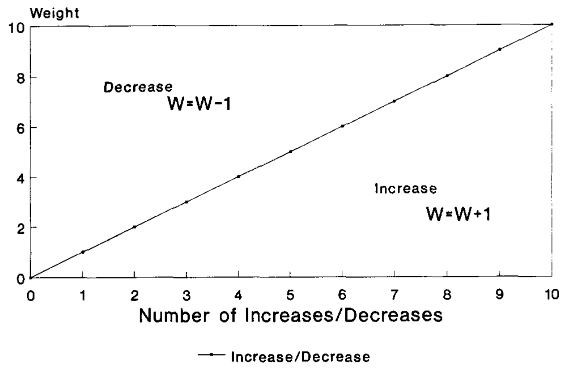


Fig. 5. A noise sensitive weighting function.

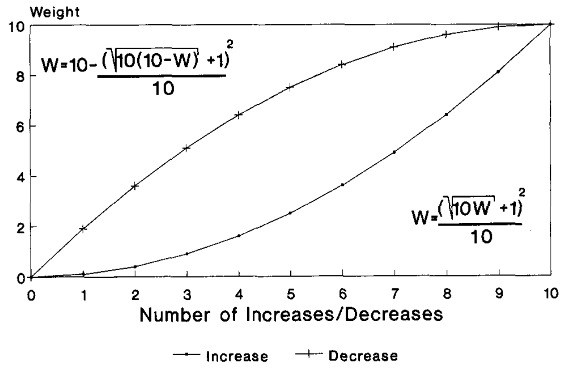


Fig. 6. A noise rejecting weighting function. The increase and decrease functions have different slopes.

a repetition of the access sequence. However, the noise may adversely affect the existing nextcluster entries by decreasing their weight fields.

If the algorithm used to update the weight fields employs simple noise rejection techniques, the stability increases. In the trivial case where the increase algorithm is merely an increment, and the decrease is merely a decrement, it can be shown that the average noise level may not exceed 50 %, meaning that an access sequence must not have intervening references from other processes at least 50 % of the time. The weight algorithm may in this case be presented graphically as in Fig. 5. The uncorrupted disk access stream, which tends to drive the weight up, and the interference which tends to drive the weight down, have equal impact.

If the probability function looks more like that of Fig. 6, involving hysteresis as indicated, the noise rejection is substantially improved. Using this function, once a pointer has settled to a high value (successor very likely to be accessed after its parent), the noise which drives this value down has much less impact than the reference stream which drives it back up. The function shown in Fig. 6 is essentially a parabola and its translated mirror image.

In Fig. 6 the lower curve represents the function used to

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

Index	First next	Weight	Second next	Weight
0	Next from 0	Weight	Next from 0	Weight
1	Next from 1	Weight	Next from 1	Weight
2	Next from 2	Weight	Next from 2	Weight
3	Next from 3	Weight	Next from 3	Weight
	⋮	⋮	⋮	⋮
n-2	Next from n-2	Weight	Next from n-2	Weight
n-1	Next from n-1	Weight	Next from n-1	Weight

Fig. 7. The adaptive table can be expanded to hold several next clusters for each entry in the table.

increase the weight and is described by:

$$W_{n+1} \leftarrow \frac{\left(\sqrt{10W_n + 1}\right)^2}{10}$$

while the upper curve represents the function used to decrease the weight and is described by:

$$W_{n+1} \leftarrow 10 - \frac{\left(\sqrt{10(10 - W_n)} + 1\right)^2}{10}.$$

The simulation results described later verified that this function performs well compared to the linear function of Fig. 5. However, further work is needed to compare it with other nonlinear functions.

Under ideal conditions, the process ID of the requesting process would be used to separate the intermixed reference streams eliminating the need for a noise rejecting function. This, too, is an area for future work.

D. Adaptive Table Usage Variations

This section examines slight variations of the basic adaptive algorithm that further improve its performance.

1) Multiple Prefetching and Branch Factors: In the earlier examples, each cluster on the disk had one nextcluster. In general, each disk cluster may have several nextclusters, particularly if the cluster size encompasses several disk sectors. Given the previous structure, the table can readily be modified to include more than one nextcluster as illustrated by Fig. 7.

In this case, the table stores the **two** most probable nextclusters for each cluster on the disk. In general, the table can be modified to store any number of nextclusters for each cluster on the disk with a separate weight field for each. The number of nextclusters for each cluster on the disk is referred to as the *branch factor* of the adaptive table. Note that the table can be viewed as a tree structure with each level branching into *branch factor* new branches.

In the case of an adaptive table with a branch factor of two or more, it may be desirable to prefetch **multiple** nextclusters, depending on their weights, as this may increase the probability that the correct one is fetched. Clearly, for larger branch factors, the amount of memory occupied by the table becomes prohibitive. Section IV-E.3) discusses a method for reducing the memory requirements of the adaptive table.

For an adaptive table with a branch factor greater than one, the following algorithm controls the adaptation of the table. Compare this algorithm to the one presented in Section III-C.1).

Given the **current** and **previous** disk cluster accesses, go to the row in the table indexed by the cluster number of the **previous** access.

1) If one of the nextcluster fields point to the **current** reference, increase its weight field.
2) Else, if none of the nextcluster fields point to the **current** reference and the weight field for one of the nextclusters is zero, place a pointer to the **current** cluster in that nextcluster field and increase its weight field.
3) Else, decrease all the weight fields.

2) Multiple Level Look-Ahead: Using the adaptive table, prefetching may occur not only for all the clusters in the adaptive table entry as dictated by the branching factor, but prefetching may also be done multiple levels ahead. As in the case of Fig. 4, when cluster zero is accessed, it may be desirable to prefetch clusters 3, 4, 2, and 5. Note that this, in effect, looks several generations ahead in the adaptive table.

3) Fetch Thresholds: When multilevel prefetching is combined with an adaptive table with a branch factor greater than one, the number of prefetches for a given cluster can become very large. In order to limit the number of prefetches for a given cluster, the weight field is used in conjunction with a **fetch threshold**. Only nextclusters with a weight greater than the fetch threshold are loaded in order to prevent channel congestion or saturation.

Take the case where the branch factor is 2 and look-ahead level is 4. As (3.1) indicates, the maximum number of prefetches possible for such a system is 30 clusters:

$$\begin{aligned} \text{First level } 2^1 &= 2 \text{ Clusters}\\ \text{Second level } 2^2 &= 4 \text{ Clusters}\\ \text{Third level } 2^3 &= 8 \text{ Clusters}\\ \text{Fourth level } 2^4 &= 16 \text{ Clusters}\\ \text{Total Clusters} &= 30 \text{ Clusters.} \end{aligned} \tag{3.1}$$

Equation (3.1) illustrates a full binary branching at each level, resulting in an unmanageable number of prefetch clusters. Clearly, 30 clusters of prefetch for a given cluster reference is not feasible as the next cluster reference will most likely arrive before the prefetch transfers are completed, and the number of prefetches would congest the I/O channel increasing the I/O delay associated with each disk transaction. In addition, since the branches from a node are for all practical purposes mutually exclusive, only one path or the other will be taken, and full fetching of both paths is neither practical nor beneficial.

If the fetch threshold is set such that only the more probable (higher weight) clusters are prefetched, the fetch tree will be significantly pruned. For example, assume that the fetch threshold is set such that each node branches into 1.5 new subtrees on average. In this case, the average number of prefetches would be reduced from a maximum of 30 to

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

approximately 12 as (3.2) indicates. Prefetching 12 clusters is more feasible, particularly if the clusters are physically in close proximity on the disk surface. Our simulation runs showed that the best performance was obtained with only a few levels of prefetching (about 2), and since many of the clusters on the disk have no next clusters above the threshold, the average number of prefetch blocks can be quite low. In addition, if a prefetch block is already present in the cache, no prefetch need be done, further reducing the number of prefetches initiated:

$$\begin{aligned} \text{First Level } 1.5^1 &= 1.5 \text{ Clusters} \\ \text{Second Level } 1.5^2 &= 2.25 \text{ Clusters} \\ \text{Third Level } 1.5^3 &= 3.375 \text{ Clusters} \\ \text{Fourth Level } 1.5^4 &= 5.0625 \text{ Clusters} \\ \text{Total Clusters } &\approx 12 \text{ Clusters.} \end{aligned} \tag{3.2}$$

E. Adaptive Cache Parameters

In contrast to an LRU cache which has relatively few parameters governing the operation of the cache, the adaptive algorithm has a host of parameters that govern its operational characteristics. These parameters can be optimized for each particular computer installation based on workload characteristics.

The parameters governing the operation have already been mentioned in the previous discussion. They are summarized here, and their impact on performance will be discussed in Section IV.

Branch Factor: The *branch factor* is the number of columns in the adaptive table and represents the maximum number of first level prefetches possible for any cluster. The adaptive table can be viewed as a tree structure where each node branches into *branch factor* new branches. The optimal branching factor of the adaptive table depends on the speed of the I/O channel, the amount of memory available to the adaptive table, the characteristics of the data on the disk, and the cluster size.

Look-ahead Level: If the adaptive table is viewed as a family tree, the *look-ahead level* is the number of generations into the future that are considered with each prefetch transaction. The number of levels to look ahead in the adaptive table depends on the I/O channel congestion, the weight of each cluster in the subtrees, the speed of the I/O channel, and the branch factor.

Weight Ceiling: The *weight ceiling* is used to control growth of the weight fields in the adaptive table. The optimal weight ceiling is a function of the amount of data migration on the disk and the amount of noise in the disk reference stream.

Weighting Function: The *weighting function* is the function used to increase and decrease the weight fields. The weighting function is selected so as to reject noise in the disk reference stream, yet keep the amount of processing low.

Fetch Threshold: The *fetch threshold* is the minimum weight a pointer may have in order for the corresponding cluster to be considered for prefetching. The fetch threshold is selected so as to complement the branch factor and look-ahead level in keeping the average number of prefetches relatively low. It is a function of the weight ceiling, branch factor, look-ahead level, pointer stability, I/O channel speed, and I/O channel congestion.

LRU Parameters: Since the adaptive algorithm incorporates an LRU cache, the same considerations of a conventional LRU cache in the selection of its parameters are also required here. Note, however, that the cache slot size particularly impacts the adaptive table as it defines the cluster size for the adaptive algorithm. If the slot size (cluster size) is particularly large, the size of the adaptive table decreases. However, the effectiveness of the table will decrease as each cluster may contain fragments of several different files. Hence, a significant increase in the cluster size should be accompanied with an increase in the branch factor.

Since the adaptive cache provides information concerning access patterns, the LRU replacement algorithm may be modified to use information from the adaptive table. For an adaptive table with a branch factor of two, for example, on each disk access, at least one of the two predicted access threads in the adaptive table is not used and all clusters along that thread are excellent candidates for removal from the cache. This replacement scheme can keep the cache from becoming severely polluted. Use of the adaptive table in selecting replacement blocks has not been examined in this paper and is an area of future study.

F. Strategic Data Layout

In prefetching a large number of clusters, as in the case of a multilevel, multibranching adaptive algorithm, the time required to service the additional I/O requests becomes critical. If the prefetched locations are in close proximity to each other, the delay in moving the head between these accesses will be greatly reduced, resulting in a significant reduction of the average disk access time. In the ideal case, a cluster and all its prefetches would be contiguous on the disk surface so they can be fetched in a single I/O operation. Alternately, the prefetches can be stored on the same cylinder (possibly on a different surface) or even on physically close cylinders and still result in a reduction in access time.

This section describes a method of restructuring the disk that uses the information stored in the adaptive table. It is important to note that disk layout optimization, unlike table adaptation, occurs infrequently. For example, a possible implementation might perform a rearrangement of disk sectors as part of the backup routine for a system, since the backup routine reads most or all of the disk in its normal operation.

1) Strategic Layout Requirements and Conflicts: In the case of an adaptive algorithm with a branching factor of two, the adaptive table prefetches may be viewed as a binary tree as illustrated in Fig. 8. If the look-ahead level is two, the root of the tree will prefetch clusters 1, 2, 3, 4, 5, and 6, assuming each of the weights is above the fetch threshold. Ideally, these clusters would be contiguous on the disk surface. At the same time, node 1 will prefetch clusters 3, 4, 7, 8, 9, and 10, and node 3 will similarly have its prefetches. It should be clear that in general there is no way of arranging the data on the disk to satisfy all these requirements simultaneously.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

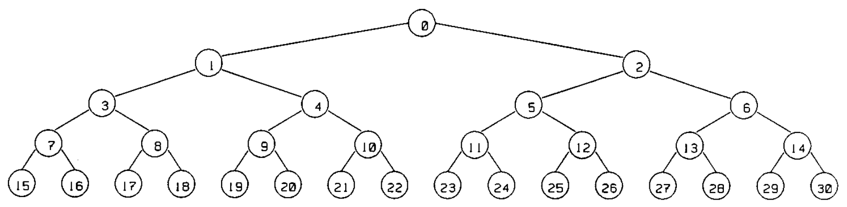


Fig. 8. The adaptive table with two threads viewed as a binary tree. In general, the tree will not be a **proper** tree.

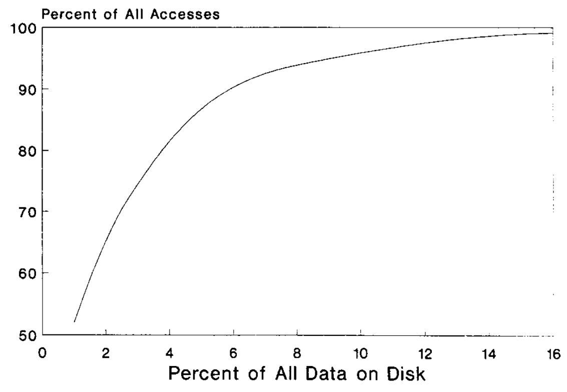


Fig. 9. A small portion of the disk sees a majority of the total disk activity.

2) Disk Usage Intensities: Although only a small portion of the disk surface can be optimally arranged for contiguous data prefetching, only a small portion of the disk needs to be rearranged. As Fig. 9 illustrates, measurements taken from the machines used in this paper show that 95 % of the references access 10 % of the data on the disk. If the correct 10 % of the disk are rearranged, most of the benefit of optimal data arrangement could be realized.

The profile of disk activity was consistent for the 3 computers used in this paper and agrees with measurements made on other systems.

3) Layout Implementation: Although only 10 % of the disk surface needs to be rearranged to obtain most of the benefit, determining the proper 10 % to rearrange is a problem. The adaptive table could be expanded to hold access intensity information, but this might not be practical for all systems.

While it is not possible to have all nodes strictly contiguous with their children, we have devised the following method of data arrangement to facilitate multicluster prefetching. We require that the root of a subtree be contiguous with its prefetches *only if it is the highest weighted child of its parent.* To illustrate, assume that the tree in Fig. 10 has been rearranged such that the highest weighted of the two children is to the right. Node 4 will be stored contiguous with its children since it is the highest weighted child of its parent, but node 1 will not be contiguous with its children because it is the lowest weighted child of its parent. Fig. 11 illustrates how the same tree can be rearranged on the disk using this technique. This approach effectively groups the data from the tree as shown in Fig. 12.

To take advantage of this data restructuring, we modify our prefetching algorithm so that the search is terminated with the lowest weighted child at every level. For example, Fig. 10 shows the group of nodes (1, 2, 5, and 6) that would be prefetched with node 0 in a two level prefetch. Note that the maximum number of prefetches has dropped from 6 to 4. Similarly, the three level prefetch group for node 0 consists of nodes 1, 2, 5, 6, 13, and 14, reducing the maximum number of prefetches from 14 to 6. This modified algorithm will be referred to as the *most likely path* algorithm throughout the rest of this paper.

There are two limitations of this disk restructuring approach that should be noted. First, the adaptive structure is not actually a binary tree — in reality the branches may combine and some backward paths may exist. Preliminary measurements taken from the systems used in our study indicate that backward paths occur infrequently and that a large portion of the disk can be restructured using the above approach. The second limitation is the requirement that certain filesystem *metadata*, such as inodes in UNIX, must reside in fixed locations on disk, and therefore, cannot be relocated at will.

Because the performance improvements resulting from this restructuring technique may vary widely between different implementations, our assumptions about the extent of the restructuring are conservative. Specifically, we assume for any group of prefetches that only the chain of highest weighted descendants of a node are contiguous. All other references are assumed to require a seek, and the average seek time is used in all cases. For example, in the two-level prefetch group of node 0 in Fig. 10, a seek is required to access the root, and additional seeks are required for nodes 1, 2, and 5, but node 6 is assumed to be contiguous with node 2. Thus the restructuring saves at most one of the maximum of five seeks that would be required to access a node and all of its two-level prefetch clusters, assuming that none were in the cache. The three-level prefetch group of node 0 would assume only nodes 2, 6, and 14 to be contiguous, and so on.

For the machines that we studied, inodes and backward paths allowed far in excess of 20% of the disk's contents to be restructured. Consider that most hard drives have several platters, so sectors do not need to be arranged in a linear list in order to be in close proximity to one another. In addition, a reasonable restructuring method could place lower weighted children in cylinders close to the parent, and therefore, require less than the average seek distance for those cases when a seek is required. Although our assumptions are, therefore, pessimistic about the benefit of restructuring, our intent in this paper is to focus on the improvement resulting from the prefetch scheme. The added improvement in performance

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

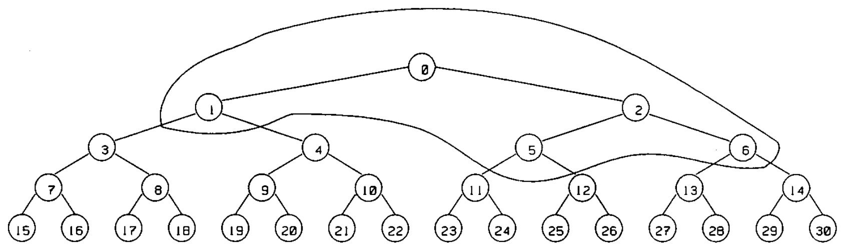


Fig. 10. The fetching is weighted towards the more probable path and the number of fetches is reduced.

resulting from rearranging the clusters on the disk is an area for future investigation. Note, however, that it is the information stored in the adaptive table that makes such a restructuring possible.

IV. Performance Evaluation

This section describes the measurement technique used for performance evaluation and presents measurement results. The performance of the adaptive prefetching strategy is then evaluated with respect to all its parameters.

A. Measurement Technique

The effectiveness of disk caching schemes has often been measured in terms of the hit rate, or conversely the miss rate, of the algorithm. However, the hit rate of an algorithm does not directly translate into performance figures. For a caching scheme that includes prefetching of data, hit rate measurements are insufficient. For example, in computing the hit rate for a prefetching caching scheme, how is the situation handled in which a prefetch for a disk sector is initiated, but the request for the prefetched data arrives before the prefetch operation has completed? Strictly speaking, it is neither a hit, nor a full miss since it requires less time than a "regular" miss. In the performance evaluation of a prefetching cache, a prefetched disk block may not be assumed available before an actual request for the prefetched data arrives.

In order to obtain accurate and meaningful results, the actual time required to service each disk request must be computed. The average time to service a request is a more meaningful measure of the performance of a cache than is hit rate analysis.

For the adaptive cache, all performance figures that will be presented are relative to a similarly configured cache with no prefetching. Since the adaptive algorithm only governs the prefetching of data from disk, the rest of the cache is unaffected. For example, both use an LRU replacement algorithm. By expressing the relative performance of our prefetching cache to a standard LRU cache, the improvement resulting from the adaptive prefetching mechanism can be evaluated. In addition, any anomalies caused by the replacement algorithm will effectively be canceled out.

B. Data Acquisition

In order to adequately assess the effectiveness of caching algorithms, the algorithms must either be implemented or accurately simulated using realistic disk reference patterns. If

```
Root                   0  1  2  5  6
Node 2                       2  5  6  13  14
Node 6                             6  13  14  29  30

Rearrangement list:
                       0  1  2  5  6  13  14  29  30
Root Contig            |___________|
Node 2 Contig                |_____________|
Node 6 Contig                      |__________________|
```

Fig. 11. Using the most probable weighting, the arrangement conflicts vanish, and the rearrangement may be cascaded indefinitely.

simulated, the simulation must be trace driven using traces collected from real systems running real user tasks in order to yield meaningful results.

In preference to implementing an untested algorithm, we chose trace driven simulation. The traces were obtained from several UNIX-based minicomputers by modifying their operating systems to log all the disk activity to a trace file. These modified kernels were traced at several different installations and under varying system loads. The machine characteristics are described in Section IV-C.

The traces consist of the information in Fig. 13 for **each** I/O request. The requests were trapped prior to service by any already installed disk cache.

In addition, the information shown in Fig. 14 was logged for every 500 I/O requests in order to track the system load.

C. Trace Machine Characteristics

All the traced computers are UNIX-based minicomputers of various architectures and user load characteristics. The characteristics of each traced machine are briefly discussed in the following. A more detailed presentation of the characteristics of these machines can be found in [5].

1) IconSys: IconSys is an ICON 4000 minicomputer utilizing the MC68020 processor that serves as an administrative computer. The machine supports an average of 8–12 users during the day, and processes a few batch transactions during the night.

The system is configured with 16 MB of primary memory, 22 MB of disk cache memory using an LRU caching algorithm, and 2 GB of disk storage. The machine has a computing throughput of approximately 2.5 VAX MIPS and runs a modified UCB UNIX 4.3 operating system utilizing the fast file system [8].

IconSys logged approximately 3.9 million disk accesses over a period of 2.6 d. Figs. 15–17 present the user load characteristics during the test period.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

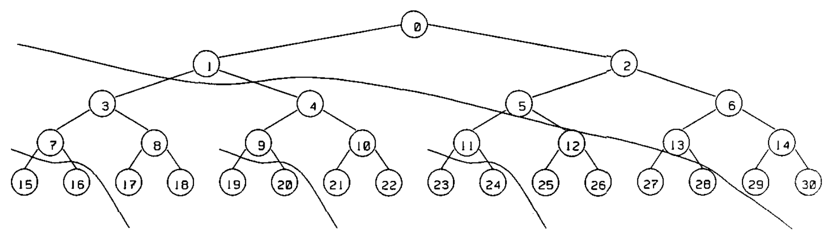


Fig. 12. Grouping the data under the new requirements is readily accomplished. Each subtree is sliced diagonally as indicated.

```
Time of event  :    Accurate to better than 20ms
Device         :    Device number
Block          :    Logical disk block number
Read/Write     :    Request was READ or WRITE
```

Fig. 13. The information logged for every I/O request. Each record has a time stamp associated with it.

```
Users          :    Number of users logged in
Load1          :    CPU load averaged over 1 min.
Load5          :    CPU load averaged over 5 min.
Load15         :    CPU load averaged over 15 min
```

Fig. 14. The information stored about the system load every 500 disk requests.

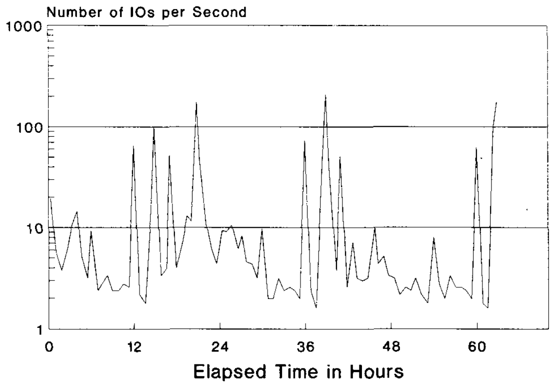


Fig. 15. The number of I/O's per second averaged over 45-min intervals for IconSys during the test period.

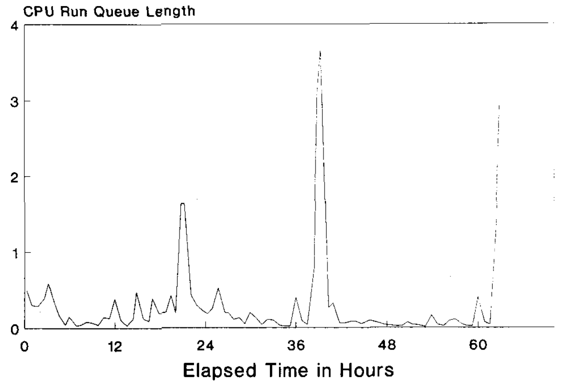


Fig. 16. The CPU run queue length of IconSys during the test period averaged over 45-min intervals.

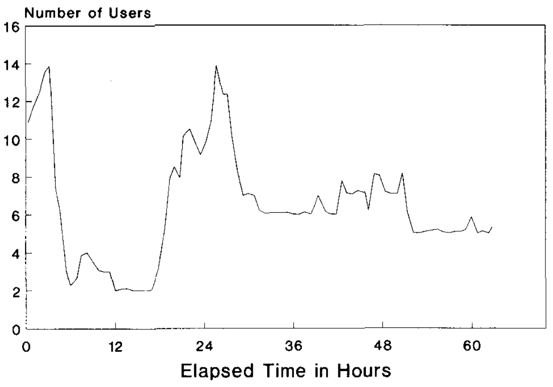


Fig. 17. The number of users on IconSys during the test period averaged over 45-min intervals.

2) EERes: EERes is a MicroVAX II used for research in image processing. The machine supports an average of 3–4 users during the day, and is mostly idle during the night.

The system is configured with 16 MB of primary memory of which 1 MB is disk cache memory in an LRU arrangement, and 0.9 GB of disk storage. The system runs an unmodified UCB UNIX 4.3 operating system utilizing the fast file system [8].

EERes logged approximately 6.1 million disk accesses over a period of 7 d. Figs. 18–20, present the user load characteristics during the test period.

As the figures indicate, the machine showed great fluctuation from day to day in terms of the number of I/O's, and seemed very load sensitive, probably due to a small number of users submitting intermittent heavy engineering tasks.

3) Adam: Adam is a VAX 11/750 used mainly for mail forwarding, editing, and message passing on the network. The machine supports an average of 1–2 users during the day, and processes accounting information at night.

The system is configured with 8 MB of primary memory of which 1/2 MB is disk cache memory in an LRU arrangement, and 1.2 GB of disk storage. The system runs an unmodified UCB UNIX 4.3 operating system utilizing the fast file system [8].

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

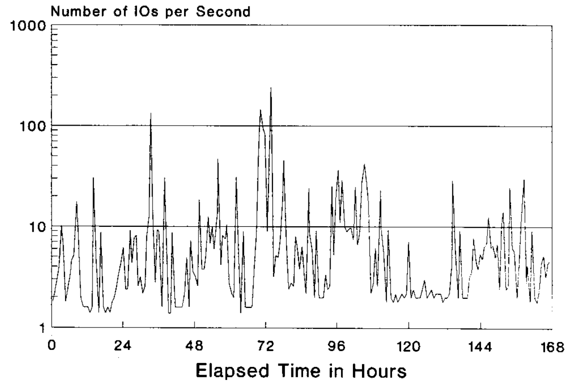


Fig. 18. The number of I/O's per second averaged over 45-min intervals for EERes during the test period.

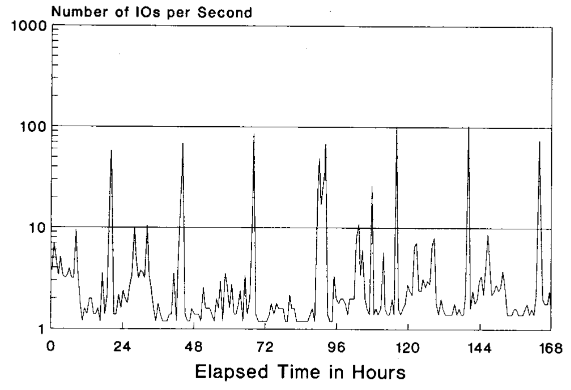


Fig. 21. The number of I/O's per second averaged over 45-min intervals for Adam during the test period.

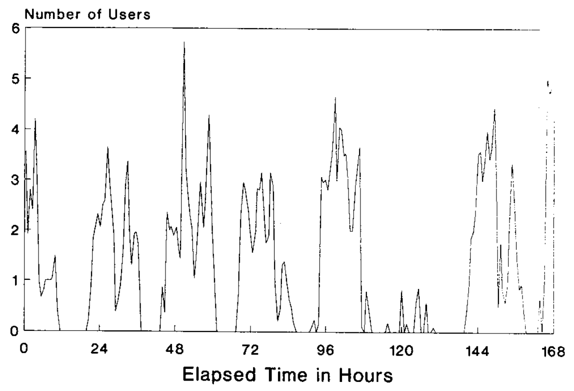


Fig. 19. The CPU run queue length of EERes during the test period averaged over 45-min intervals.

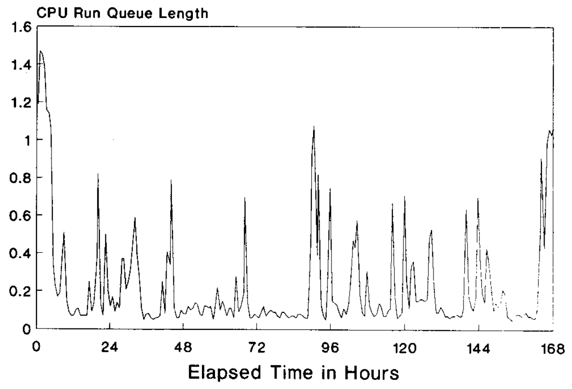


Fig. 22. The CPU run queue length of Adam during the test period averaged over 45-min intervals.

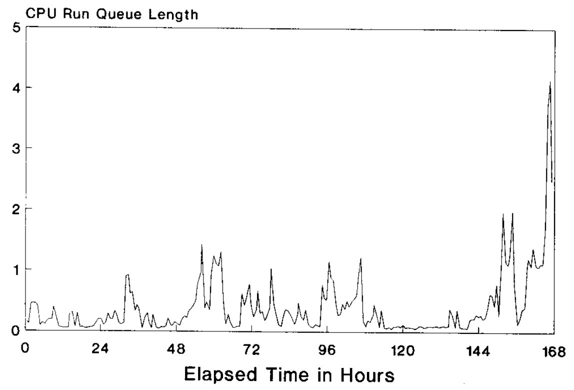


Fig. 20. The number of users on EERes during the test period averaged over 45-min intervals.

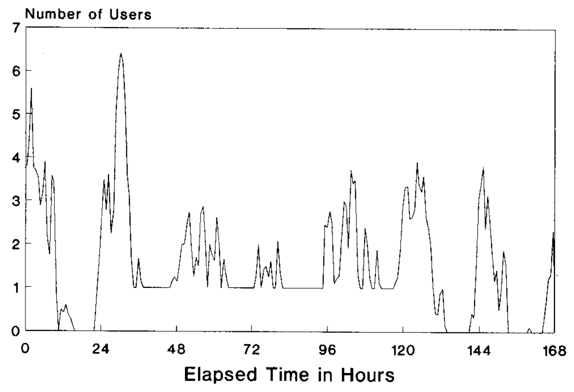


Fig. 23. The number of users on Adam during the test period averaged over 45-min intervals.

Adam logged approximately 3.4 million disk traces over a period of 7 d. Figs. 21–23 present the load characteristics during the test period. As the figures indicate, there was little fluctuation from day to day in terms of load.

Fig. 24 presents a summary of the traced machine characteristics.

D. Simulation Model and Parameters

Measuring the performance of a cache in terms of the average time required for a disk transfer may seem simple, but in order to obtain realistic results, the computer system's disk channel must be carefully simulated. The simulator must take the following variables into account:

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

Machine	Main Use	Users	Days	Traces	Mips
IconSys	Business	8-12	2.6	3.9M	2.5
EERes	Research	3- 4	7	6.1M	0.8
Adam	Messaging	2- 3	7	3.4M	0.7

Fig. 24. Summary of traced machine characteristics.

1) the average access time of the disk drive;
2) the transfer rate of the drive;
3) the I/O queue to the drive;
4) channel contention and saturation;
5) the relative arrival time of disk requests.

The simulator must, in effect, simulate the complete disk system of a computer.

Of the variables that pertain to the physical disk hardware, values which were deemed typical were chosen. In general, we have found that varying these parameters has little effect on the relative performance of the adaptive cache to the LRU cache.

1) Access Time: The access time of a disk drive is the amount of time required to position the read/write head over the desired portion of the disk prior to a data transfer to or from the disk itself. The access time has two components, seek time and rotational latency. The head seek time constitutes a substantial portion of the total disk access time. Although typical drives have an average seek time in the area of 25–28 ms, the average distance of a seek operation in a real computer system is very small, often within the same track or to an adjacent track. Measurements have shown that the average seek time is typically 11 ms, so this value was used in our simulations.

The rotational latency is the amount of time required for the proper sector to be positioned under the head once the proper track has been selected. This latency will average 1/2 of a disk revolution, and for a 3600-rpm disk this will be 8 ms. The simulator then assumes that the average disk access time is 19 ms.

As shown in Section III-F, layout restructuring can be used to reduce the number of seeks required by placing related blocks adjacent on disk. The simulation study assumed that, of the prefetch blocks generated by the most probable path algorithm, only the most probable child and *its* most probable child were contiguous. In the case of a two-level prefetch used for the majority of this paper, this reduces the maximum number of seeks for prefetches from 4 to 3. In practice, a most probable child of a most probable child can always be grouped with the exception of interfering inodes or cycles in the adaptive table. All the remaining seeks were pessimistically assumed to be of average length, as were all seeks required in simulating the LRU cache.

2) Transfer Time: The transfer time is the amount of time required to transfer a single sector to or from the disk once the seek and latency is accounted for. The amount of time required to transfer one sector of data to or from the disk corresponds to the amount of time required for one sector to pass under the read/write head. For a typical 5-1/4-in hard disk with 34 sectors per track, this would be about 0.5 ms. As (4.1) indicates, the total service time for a disk I/O was assumed to be approximately 20 ms. Note that in practice, that the ratio of the performance of the LRU to adaptive algorithms is not significantly affected by a change in these values:

$$\text{SEEK TIME} + \text{LATENCY}+\text{TRANSFER TIME} = 20\text{ ms}. \tag{4.1}$$

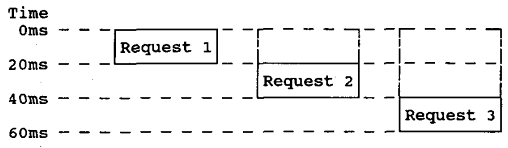


Fig. 25. The accumulation of delays by three simultaneous disk requests.

3) Drive I/O Queue: In typical systems the disk drive has a queue of pending I/O requests associated with it. The depth of this queue is a function of the speed with which requests can be serviced as well as the rate at which new requests are added to the queue. The depth of this queue affects the amount of time a given I/O request takes since new I/O requests are added to the tail of the queue when the request is made.

A system whose average queue length is greater than one is said to suffer from I/O channel congestion as the queued requests must wait for the completion of other previous requests before they may be serviced. Such congestion translates into poor performance.

Take, for example, the situation where 3 single-sector I/O requests are made at time zero. Each request will take 20 ms to complete once it reaches the head of the request queue as indicated by (4.1).

Fig. 25 illustrates the accumulation of delays due to channel congestion. For this scenario, 120 ms of delay is created as indicated in (4.2). These delays correspond to the amount of time each request takes to complete from the time they are initially requested:

$$\begin{aligned} &\text{Request 1: 20 ms}\\ &\text{Request 2: 40 ms}\\ &\text{Request 3: 60 ms}\\ &\text{- - - - - - - - - -}\\ &\qquad\text{Total: 120 ms} \end{aligned} \tag{4.2}$$

120 ms of I/O delay was accumulated in just 60 ms of real time. Had the three requests arrived in 20-ms intervals, only 60 ms of I/O delay would have accumulated. Instead, they arrived simultaneously and congested the I/O channel.

Most operating system implementations sort the I/O queue in order to minimize the disk head travel distance, and hence, increase performance. Although the simulator does not sort the I/O queue, the effect of sorting has been accounted for by using an average seek time which was measured on a system which does in fact sort the I/O queue.

In the performance evaluation of a prefetching cache, the channel congestion effects must be taken into account as a prefetching cache tends to aggravate channel congestion. In

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

Cache Size	=	2Mb
Slot Size	=	4 sectors of 1024 bytes
Branch Factor	=	2
Prefetch Levels	=	2 using most likely path
Average Access Time	=	19ms
Disk Transfer Time	=	1ms per sector
Disk Layout	=	Restructured as in Sec. 3.6

Fig. 26. The simulated system characteristics.

order to obtain reasonable results, the queuing of I/O requests to the disk must be taken into account.

4) Write Policy: The *write policy* of a cache determines its actions when its contents are modified. A *write through* policy updates both the cache and the disk, while a *delayed write* policy modifies only the cache initially, the disk is updated when the modified block is selected for replacement.

The write policy has little effect in the comparison of the LRU and adaptive caches as both caches use the same policy. Since one of the tested computers uses a delayed write policy, the simulator also uses this policy so that the simulation results could be compared to the performance of an actual computer. Also, the delayed write policy generally outperforms write through policies. For sensitive installations, a delayed write policy may not be acceptable. However, the write policy has little effect on the **relative** performance of the LRU and adaptive caches.

E. Measurement Results

This section covers the results of the simulations of the disk caches using the traces gathered from the test machines. All results should be interpreted by comparing the performance of the adaptive cache to the corresponding LRU cache without prefetching. Except where otherwise noted, the adaptive cache uses a two-level prefetch.

1) Simulated System Characteristics: The simulated results are for a system with the nominal characteristics of Fig. 26.

The effects of variations in these parameters is also presented in the next few sections. Note that the same simulator was used for the LRU and adaptive cache measurements since the LRU cache is a component of the adaptive cache. For the LRU measurements, prefetching was merely disabled.

2) Performance Relative to LRU Cache: Figs. 27–29 present the performance of the adaptive cache relative to the corresponding LRU cache for each of the tested computers. Fig. 27 represents 2.6 d of activity on IconSys, Fig. 28 represents 7 d of activity on EERes, while Fig. 29 represents 7 d of activity on Adam. The plotted values represent the accumulated average I/O time per disk access.

Fig. 27 presents the IconSys test results. For comparison, sequential prefetching is included in Fig. 27. It can be seen that the single-cluster look-ahead scheme actually degrades performance due to cache pollution and channel congestion effects as discussed in Section II. IconSys uses a smaller unit of transfer from disk, and is I/O bound, hence, the accumulated disk access times are generally higher for IconSys than for the other tested machines.

At the end of the simulation, the accumulated average access time for the nonprefetching cache is 2.8 times that of the adaptive prefetching cache.

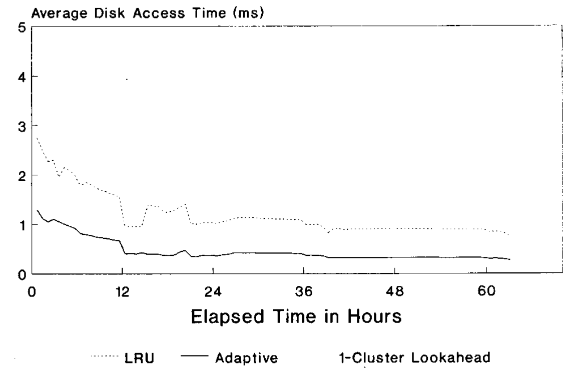


Fig. 27. The accumulated average disk access time for the LRU and adaptive disk caching schemes for the IconSystest case.

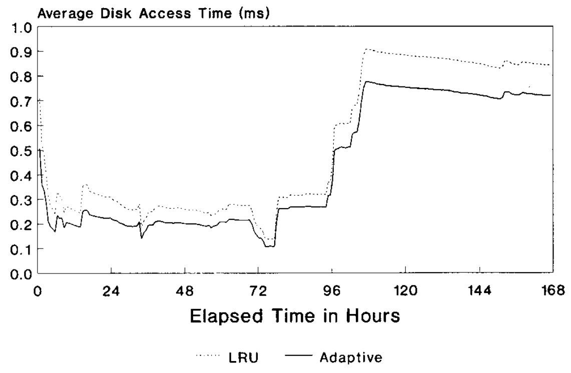


Fig. 28. The accumulated average disk access time for the LRU and adaptive disk caching schemes for the EERes test case.

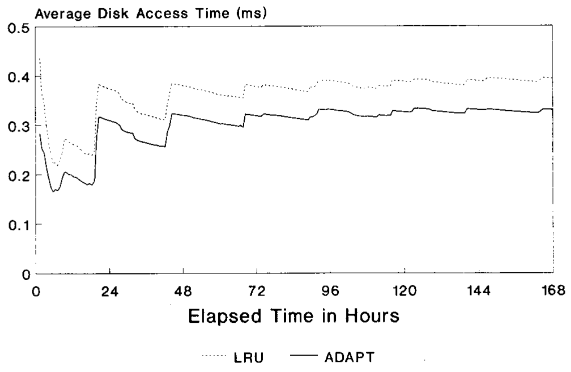


Fig. 29. The accumulated average disk access time for the LRU and adaptive disk caching schemes for the Adam test case.

From these curves, the warm-up time of the caches can clearly be seen as the curves tend to start high, and decrease as the caches warm up.

Fig. 28 shows the EERes test results. Note that the first portion of the trace between 24 and 72 h is for the weekend, and although a number of users logged in during that period, the cache performed well, probably due to a small working

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

set. At the 96-h point in the trace, it appears that the working set changed drastically, and that the new set had to be loaded from the disk. A backup of the machine may have occurred at that time which accounts for the abrupt change in the average I/O time.

As Fig. 29 indicates, Adam was very lightly loaded, and the working set of the disk probably did not change during the entire test period. The daily cycles of use can clearly be seen. During the night, when the machine is unused, the access time decreases due to predictable disk access patterns, and the small working set used.

3) Adaptive Table Locality: The size of the adaptive table is quite significant, and may take up much of the available disk cache memory. Equation (4.3) presents the memory requirements of a moderate adaptive algorithm for a 300-MB disk:

$$\frac{300\text{ MB}}{1024\text{ Bytes/sector}} \times \frac{1\text{ cluster}}{4\text{ sectors}} = 76\,800\text{ clusters}$$

NextCluster requires 17 b

Weight requires 7 b (using fixed point arithmetic)

$$2\text{ Branches} \times (17 + 7\text{ b}) = 6\text{ Bytes}$$

$$76\,800\text{ clusters} \times \frac{6\text{ Bytes}}{\text{cluster}} = 460\,800\text{ Bytes}. \qquad (4.3)$$

This table occupies nearly 1/2 MByte of cache memory. If the amount of memory for cache storage is small, this memory would probably be better utilized for cache buffers. As Fig. 10 illustrated, only a small portion of the disk surface is in active use: for the tested machines 95 % of all accesses addressed 10 % of the disk surface. The concepts of locality imply this kind of usage profile, although the actual numbers will vary between installations.

Because only a small portion of the disk is in active use, the complete table need not be in memory at any given time; only those portions of the table corresponding to active disk areas need be memory resident. The table, then, could be placed in virtual memory with a small initial allocation size and be paged using the virtual memory mechanism.

Fig. 30 represents the virtual memory paging rate for the adaptive table versus the memory allocation size. As illustrated, even small allocations maintain low paging rates for the adaptive table. For example, with a 32-kB allocation and 1-kB pages, only about 0.25% of all references are a part of the table that is swapped out. In the figure, a constant memory allocation policy is assumed.

These paging rates are probably much higher than the reader is accustomed to seeing as acceptable memory paging rates. Note that the table is referenced only once every disk access, not once for every memory access. In addition, a page fault on the adaptive table does not have the performance impact of a page fault on program code, as a fault on the table merely implies that prefetching cannot be performed for the faulting I/O request. In effect, prefetching is only enabled for the portion of the table that is memory resident, and the paging mechanism will keep the most active (most recently used) portion of the table resident.

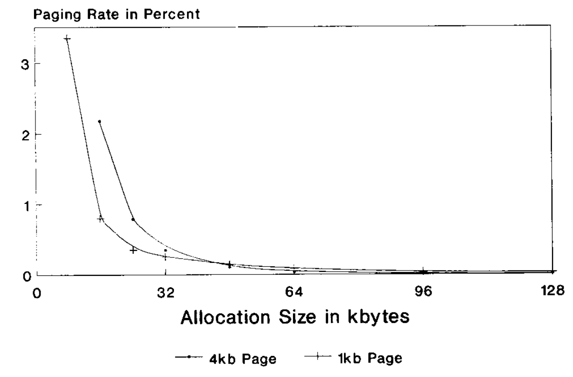


Fig. 30. Vitrual memory paging rate for the adaptive table versus the initial memory allocation size. The size of the complete table fro this measurement was 380 kBytes.

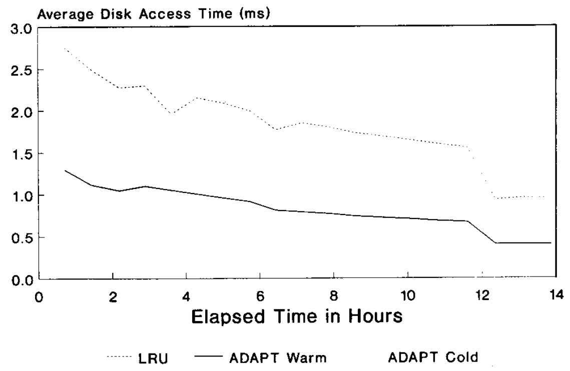


Fig. 31. The warmup characteristics of the adaptive cache for the IconSys test case. The table warms up after about 6 h.

4) Adaptive Cache Warm-Up Characteristics: The adaptive table itself is meant to be saved prior to a system shutdown, so the table experiences a warm-up period only once during the initial system start-up.

On start-up, the cache is obviously empty, and a warm-up period is associated with filling the cache buffers. Fig. 31 illustrates the superior warm-up characteristics of the adaptive cache over the LRU cache. The LRU cache relies solely on the cache buffers for its performance, and initially when the buffers are empty, the performance benefit is small. However, the adaptive cache gains much of the performance increase through prefetching, and since the adaptive table is restored on startup, the prefetching is immediately accurate.

In Fig. 31, note that the cache itself always starts out cold, but the adaptive table either starts out restored (warm) or unrestored (cold). As the figure indicates, the adaptive cache obtains substantial performance benefits even when the cache is cold as long as the adaptive table is warm.

The dotted curve in Fig. 31 depicts the case where the adaptive table itself is cold. As can be seen, the table warms up quickly, and provides a substantial performance increase after about 6 h of adaptation. When the adaptive table is cold, the algorithm functions like an LRU cache.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

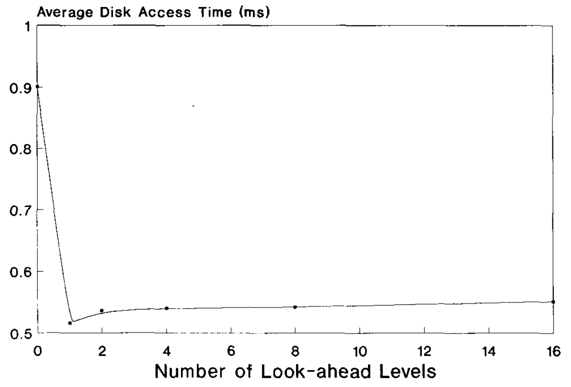


Fig. 32. The performance of the adaptive cache versus the number of look-ahead levels. The performance degrades only slightly for large look-ahead levels.

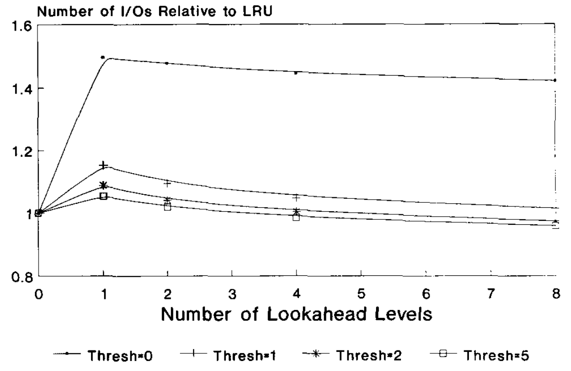


Fig. 33. The I/O overhead characteristics of the adaptive cache. Note that the graph plots the number of I/O operations, not the number of sectors transferred from disk.

5) Tree Traversal Characteristics: Although all the parameters of the adaptive cache have not been optimized and are in general a function of the particular installation, the key parameters have been experimentally measured. Fig. 32 presents the effect of varying the number of prefetch levels. Increasing the number of levels increases the number of prefetches and may congest the I/O channel. All of these experiments use the most likely path algorithm and the disk restructuring assumptions described in Section III–F.3).

In order to prevent channel congestion when increasing the number of prefetch levels, the load threshold could be increased. Schemes that vary the prefetch levels and load threshold dynamically according to channel utilization could also reduce the congestion problem.

6) I/O Overhead Characteristics: Unless the prefetching is extremely accurate, the number of I/O operations performed by a prefetching algorithm will significantly exceed that of a nonprefetching algorithm, potentially congesting the I/O channel. If the prefetching is extremely accurate, the channel congestion will not increase as the prefetches constitute data that would have to be transferred from the disk in any case.

Fig. 33 presents the I/O overhead of the prefetching cache for various load thresholds and prefetching levels. Note that for moderate load thresholds, the total number of I/O's is only marginally greater than the number of I/O's required by an LRU cache (note that a prefetching level of zero is equivalent to an LRU cache with no prefetching). Since the number of I/Os' is only marginally greater than the I/O's required by an LRU cache, the prefetching is extremely accurate and does not significantly add to the channel congestion.

As discussed earlier, the *load threshold* is the minimum *weight* a pointer may have in order for prefetching to take place. As expected, the lower the threshold, the greater the number of prefetches.

7) Adaptive Table Pointer Functionality: Now that the adaptive table has been shown to accurately determine the clusters to be loaded, the adaptive table itself is analyzed. If the pointers in the adaptive table merely point to the next cluster on the disk, a different approach may be used successfully for prefetching. However, if the pointers point to clusters other than the next logical cluster, the adaptive table pointers may not be replaced by a simpler sequential scheme.

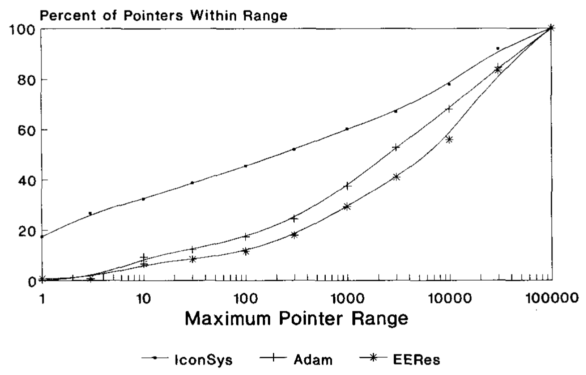


Fig. 34. The distance the pointers in the adaptive table point is well distributed over the entire possible range. For IconSys, only 60 % of all pointers point a distance of 1000 or less.

As Fig. 34 shows, the distance the pointers in the adaptive table point is distributed over the entire possible range. Note that less than 20 % of all pointers in the table point a distance of 1 cluster, confirming again that prefetching algorithms which rely on spatial locality will not be very effective. The graph indicates that for the IconSys test case, 60 % of all nonzero pointers in the table point a distance of 1000 clusters or less. The graphs reflect the spatial locality measured in Fig. 2; IconSys with the greatest locality is the higher of the three curves in the figure.

8) Pointer Stability Characteristics: The weighting function used in the simulator is the hysteresis function presented in Fig. 6. On increase the function looks like a parabola, while the decrease is a rotation of the increase function. The weight ceiling has little effect on performance for reasonable values and was set to 10. Although the weighting function is fairly simple, it is interesting to note the stability of the pointers in the adaptive table. Fig. 35 shows the adaptive pointer weights at the end of the simulation runs. As the figure indicates, the weight of the pointers tend to be either **high** or **low**, with few pointers with medium weights. This indicates that the pointers

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

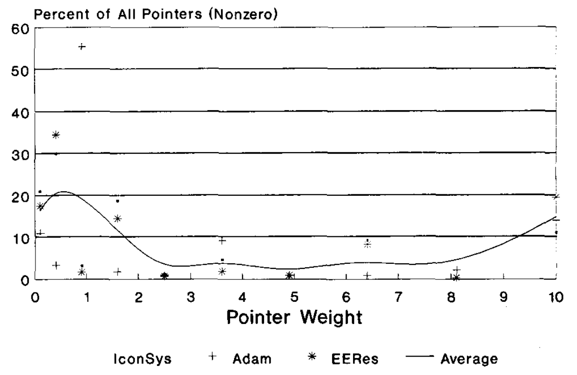


Fig. 35. The pointers in the adaptive table stabilize at either end of the weight scale. The X-axis points correspond to the points along the hysteresis curve of Fig. 6.

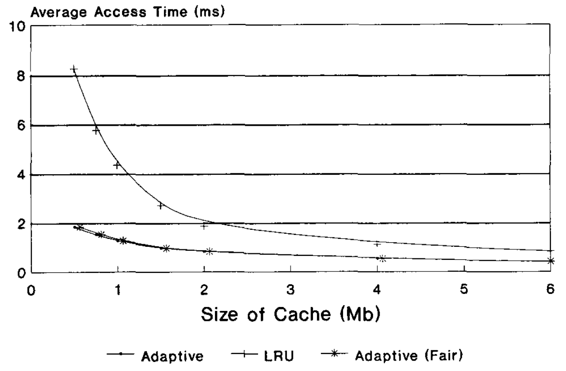


Fig. 36. The cache performance versus cache size. The improvement obtained by using the adaptive algorithms over an LRU cache is greater for small cache sizes.

either stabilize very strongly to a particular nextcluster, or they have such a low weight that they fall below the load threshold (assuming it is well chosen), and therefore, do not cause prefetching.

The stability characteristics are consistent with the fact that not all clusters have more than one next cluster pointer, and that the pointers indeed seem to stabilize well to the next cluster. The intermediate weight values would indicate instability as those values neither point strongly to the next cluster nor have a weight low enough to fall below the fetch threshold.

9) Cache Size Characteristics: Although the cache size is usually a parameter governed by other factors such as the availability of memory, it is interesting to note the ratio of performance for the LRU and adaptive caches for various cache sizes.

As Fig. 36 indicates, the adaptive cache outperforms the LRU cache for all cache sizes. The ratio of performance between the two caches decreases for large cache sizes as the size of the cache approaches the size of the working set for the disk. In the figure, the fair adaptive line represents the performance of an adaptive cache whose total size (including table) is the same as for the corresponding LRU case and where the paging effects of the table are taken into account.

As long as the size of the adaptive table can be controlled through paging schemes, the incurred complexity of the adaptive prefetching approach is offset by a substantial performance improvement. The performance gains possible are particularly significant for small cache sizes. For very large caches the performance improvement of the adaptive over the LRU cache is approximately a factor of 2.

V. Conclusions

This section concludes with a summary of test results and some recommendations for further study.

A. Performance Results

Due to the poor locality of disk block references, simple prefetching schemes such as single-sector look-ahead may not be effective, and in some cases may actually degrade the overall system performance by congesting the I/O channel.

The adaptive prefetching mechanism presented in this paper offers performance improvement by maintaining a record of the past history of disk accesses for all clusters on a disk. The adaptive table stores this information in a compact and efficient manner, and is effective in predicting future disk references.

The adaptive table contains sufficient information to strategically rearrange the data on the disk surface for fast data retrieval. The rearrangement algorithm effectively accomplishes strategic data layout on the disk and is meant to operate in conjunction with the adaptive cache.

Using traces gathered from several UNIX-based installations, simulation results indicate that the adaptive cache outperforms a similar LRU cache by a factor of up to 2.8, in terms of average disk access time.

B. Further Study

We have not yet implemented the proposed caching algorithm on any machine. The final analysis of the new caching scheme lies in implementing the algorithm and comparing its performance for a wide range of workloads with an identical computer not employing the algorithm.

The optimal parameters governing the operation of the adaptive prefetching will vary from installation to installation depending on machine characteristics and user load characteristics. The cache has not been parameterized in terms of any specific target computer system. For this paper, the parameters were chosen with the IconSys computer specifically in mind.

Additional studies of this approach for other types of machines such as database servers, transaction processing systems, or numeric processors is needed before its effectiveness is known in these areas.

The adaptive table may be used in other ways not mentioned in this paper. By adding a counter field to each entry in the table, it may be used to determine which portions of the disk are used the most, hence, identifying which disk areas should be rearranged for contiguous access. Such access information may also be used for disk balancing, and other disk optimization techniques. These extensions deserve attention.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENT

The authors express their thanks to Icon International, Dennis Fairclough, and Kelly Flanagan for their help in obtaining the traces used for this paper. They also thank the anonymous referees for their helpful suggestions.

REFERENCES

[1] M. J. Bach, *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

[2] J.-L. Baer, "Dynamic improvement of locality in virtual memory systems," *IEEE Trans. Software Eng.*, vol. SE-2, Mar. 1976.

[3] J. P. Buzen, "BEST/1 analysis of the IBM 3880–13 cached storage controller," in *Proc. CMG 13th Int. Conf.*, San Diego, CA, Dec. 1982.

[4] *Cray X-MP and Cray-1 Computer Systems: IOS Software Internal Reference Manual*, Cray Research, Inc., Mendota Heights, MI, 1984.

[5] K. S. Grimsrud, "Multiple prefetch adaptive disk caching with strategic data layout," M.S. thesis, Brigham Young University, Dec. 1989.

[6] R. R. Henry, "VAX address and instruction traces: A description of the tracer utility" Univ. California, Berkeley, 1983.

[7] W. Hugelshofer, "Cache buffer for disk accelerates minicomputer performance," *Electron.*, Feb. 10, 1982.

[8] M. K. McKusick, "A fast file system for UNIX," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, Aug. 1984.

[9] M. N. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite network file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, Feb. 1988.

[10] J. K. Ousterhout *et al.*, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proc. Tenth ACM Symp. on Operating System Principles*, Dec. 1985.

[11] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Comput.*, vol. 11, Dec. 1978.

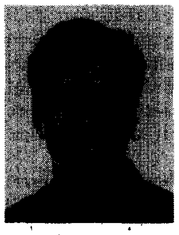
[12] ——, "Sequentiality and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, Sept. 1978.

[13] ——, "Cache memories," *Computing Surveys*, vol. 14, no. 3, Sept. 1982.

[14] ——, "Disk cache-miss ratio analysis and design considerations," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, Aug. 1985.

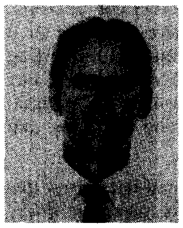
Knut S. Grimsrud received the B.S. and M.S. degrees in electrical and computer engineering from Brigham Young University in 1988 and 1989, respectively. He is currently working toward the Ph.D. degree at Brigham Young University.

His research interests include hardware tracing techniques and characterization of memory reference traces.



James K. Archibald (S'85–M'86) received the B.S. degree in mathematics from Brigham University in 1981, and the M.S. and Ph.D. degrees in computer science from the University of Washington in 1983 and 1987, respectively.

He is currently an Assistant Professor of Electrical and Computer Engineering at Brigham Young University. His research interests include parallel processing, high performance memory architectures, and simulation methodologies.



Brent E. Nelson (S'83–M'85) received the B.S., M.S., and Ph.D. degrees in computer science from the University of Utah in 1981, 1983, and 1984, respectively.

Since 1984, he has been on the faculty of Brigham Young University, where he is currently an Associate Professor of Electrical and Computer Engineering. His research interests include computer systems performance analysis, computer architecture, VLSI CAD, and design.

Authorized licensed use limited to: University of Illinois. Downloaded on March 28, 2009 at 11:20 from IEEE Xplore. Restrictions apply.