A Cost-Benefit Scheme for High Performance Predictive Prefetching *

Vivekanand Vellanki
College of Computing, Georgia Tech
vivek@cc.gatech.edu

Ann L. Chervenak
Information Sciences Institute
annc@isi.edu

Abstract

High-performance computing systems will increasingly rely on prefetching data from disk to overcome long disk access times and maintain high utilization of parallel I/O systems. This paper evaluates a prefetching technique that chooses which blocks to prefetch based on their probability of access and decides whether to prefetch a particular block at a given time using a cost-benefit analysis. The algorithm uses a probability tree to record past accesses and to predict future access patterns. We simulate this prefetching algorithm with a variety of I/O traces. We show that our predictive prefetching scheme combined with simple one-block-lookahead prefetching produces good performance for a variety of workloads. The scheme reduces file cache miss rates by up to 36% for workloads that receive no benefit from sequential prefetching.

We show that the memory requirements for building the probability tree are reasonable, requiring about a megabyte for good performance. The probability tree constructed by the prefetching scheme predicts around 60-70% of the accesses. Next, we discuss ways of improving the performance of the prefetching scheme. Finally, we show that the cost-benefit analysis enables the tree-based prefetching scheme to perform an optimal amount of prefetching.

1 Introduction

High-performance computing systems will increasingly rely on prefetching data from disk to overcome long disk access times and to maintain high utilization of parallel I/O systems. Disk access times of up to 10 milliseconds may cause processors to idle for millions of cycles. This idle time can be reduced or eliminated if data blocks are prefetched before a process requests access to the data. Effective prefetching schemes based on known or predicted data access patterns can also improve the bandwidth and utilization of disk arrays and parallel I/O systems.

Operating system prefetching schemes make two decisions: *which blocks* to prefetch and *when/whether* to prefetch. A prefetching system can use probabilistic or deterministic information to determine which blocks to prefetch. A probabilistic scheme uses past access patterns to infer which blocks have a high probability of being accessed in the future [19, 5, 6, 8, 10, 9, 12]. Since future accesses are predicted rather than known, such schemes may prefetch blocks that are never accessed. A deterministic scheme chooses blocks to prefetch using application-provided hints [14, 15, 7, 18, 2]. The application uses its knowledge of I/O access patterns to give the prefetching system an ordered list of blocks that will be accessed in the future.

After choosing good candidates, the prefetching scheme must decide whether to prefetch a candidate block and when prefetching is most advantageous. Prefetched blocks displace blocks that were fetched on demand or blocks that were prefetched earlier. If these displaced blocks are later needed, they must be re-fetched from disk, which may hurt performance. Consequently, prefetching schemes avoid retrieving blocks earlier than necessary.

This paper evaluates a prefetching technique that chooses which blocks are candidates for prefetching based on their probability of access and decides whether to prefetch a particular block at a given time using a cost-benefit analysis. The benefit of prefetching each candidate block is compared to the cost of replacing another block from the cache; a block is prefetched only if the benefit exceeds the cost.

Our cost-benefit analysis is based on Patterson's informed prefetching scheme [14, 15, 18]. The main difference from informed prefetching is that we infer probable future access patterns based on past accesses by constructing a prefetch tree. This scheme requires no application modifications. However, since some predictions are incorrect, some blocks will be prefetched and never accessed. By contrast, in the informed prefetching scheme, applications are modified to provide

*This work was supported in part by NSF CAREER Award CCR-9702609.

deterministic hints about future access patterns. All hinted blocks are eventually accessed. Based on this difference in how prefetch candidates are identified, our cost-benefit analysis accounts for probabilities of access as well as additional overheads for blocks that are prefetched but never accessed.

We focus on two prefetching schemes, the basic *tree* algorithm, which performs predictive prefetching based on past accesses using our cost-benefit analysis, and the *tree-next-limit* algorithm, which combines the *tree* scheme with one-block-lookahead prefetching. We evaluate these prefetching schemes using trace-driven simulations. We show that *tree-next-limit* performs well for a variety of workloads. For workloads with substantial sequential accesses, *tree-next-limit* performs as well as a simple one-block-lookahead prefetching scheme, reducing cache miss rates by up to 73%. In addition, for workloads without much sequential access, our tree-based prediction scheme is able to effectively predict future accesses based on past access patterns, reducing cache miss rates by up to 36%.

Next, we examine the *tree* algorithm in greater detail. We examine the effect of increasing overall cache size on the performance of the algorithm. We also present the hit rate in the prefetch cache. We show that the memory requirements of the *tree* algorithm are reasonable, requiring about a megabyte for good performance. We also show that the prefetch tree is very effective in identifying candidates for prefetch, predicting 60-70% of the future accesses. Next, we show that there is potential for significantly improving the performance of the *tree* algorithm and present one particular variation that we considered. Finally, we determine the effectiveness of the cost-benefit scheme used by the *tree* algorithm by comparing it to other tree-based prefetching schemes that do not use our cost-benefit analysis.

The remainder of this paper is organized as follows: In the next section, we explain our prefetching scheme. Next, we explain the system model for our experiments. Section 4 gives an overview of the cost-benefit algorithm, followed by a detailed analysis in Sections 5, 6 and 7. Section 8 discusses the implementation of our trace-driven simulator. We evaluate the performance of our prefetching scheme in Section 9. Section 10 discusses related work.

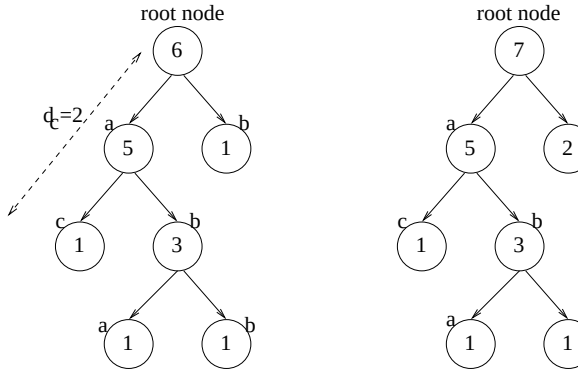
2 The Prefetch Tree

In this section, we explain how we construct the prefetch tree that is used to make predictions about future accesses. We use the Lempel-Ziv (LZ) scheme from Duke University [19, 5]. The LZ scheme constructs a directed tree based on disk accesses. The tree contains a root node where the algorithm begins. The remaining nodes in the tree correspond to disk blocks. An edge exists from node A to node B in the tree if disk block B was accessed immediately after disk block A. Each node has a *weight* that is equal to the number of times the node has been accessed. The probability that block B will be accessed after block A is the weight of node B divided by the weight of node A. Figure 1 shows an example of a prefetch tree. In Figure 1(a), the probability of accessing nodes a and b from the root node are $p_a = 0.83$ and $p_b = 0.17$, respectively. For nodes at deeper levels in the tree, we multiply the probabilities along the edges that make up the path from the current node to the block that is a candidate for prefetching. In Figure 1(a), the probability of accessing block a followed by b from the root node is $p_b = (0.83)(0.6) = 0.5$.

The prefetch tree is constructed as follows. An application makes a series of disk accesses. We divide the list of these accesses into "substrings", where each substring consists of a previous substring plus one additional disk access. Figure 1 shows a tree constructed after the following series of disk block accesses: (a)(ac)(ab)(aba)(abb)(b), where the parentheses indicate substrings. When processing a new substring of disk accesses, the LZ scheme starts from the root of the prefetch tree. When the first block of a new substring is accessed, the LZ scheme checks the tree to see if the block was accessed from the root before. If so, it traverses the edge to the node corresponding to that disk block and increments the weight of the node. The algorithm continues to traverse edges in the tree corresponding to disk accesses until it encounters a disk block for which an edge in the tree does not exist. A nonexistent edge in the tree suggests we are encountering this order of accessing disk blocks for the first time. When this occurs, the LZ scheme adds a new edge and a new node corresponding to the disk access to the prefetch tree. At this point, we have defined a new substring, and we return to the root to process the next substring. Figure 1 shows the prefetch tree before and after visiting block b from the root node.

Since edges in the prefetch tree correspond to blocks that are accessed in order, we can initiate prefetches "early" if we initiate prefetches for nodes that appear several edges away in the prefetch tree. More precisely, when we prefetch a block b, we can define the depth or *distance* d_b of a prefetch as the number of disk accesses after which block b is expected to be accessed. d_b reflects the number of edges in a path in the prefetch tree between the current block and block b. Figure 1(a) illustrates a distance of two from the root node.

Once weights are assigned to nodes, we can choose blocks as candidates for prefetching that have a high probability of being accessed.



a) After string: (a)(ac)(ab)(aba)(abb)(b) b) After accessing node b from root

Figure 1: *A prefetch tree for disk accesses (a)(ac)(ab)(aba)(abb)(b). Parts a) and b) show the prefetch tree before and after accessing node b from the root node. In part b), the weights of the nodes visited have been incremented. The figure also shows the distance $d_c = 2$ between the root node and node c, two levels deeper in the tree.*

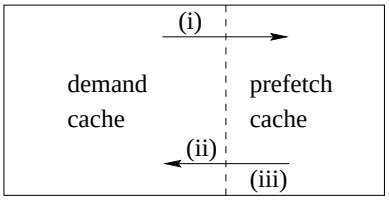


Figure 2: *Structure of the combined prefetch and demand cache. To accommodate a new prefetch block, a buffer must be reclaimed from either the demand cache (i) or the prefetch cache. Likewise, if a demand miss occurs, a buffer is reclaimed from either the prefetch cache (ii) or the demand cache. When a prefetched block is referenced, it moves to the demand cache (iii).*

3 System Model

Next, we describe the system model we use to calculate the cost and benefit of prefetching candidate blocks. We share many assumptions with Patterson's model [14]. We assume a uniprocessor running a modern operating system. The system includes a file buffer cache that is partitioned into a *demand cache* and a *prefetch cache*, as shown in Figure 2. The demand cache holds disk blocks that have been referenced previously and uses an LRU replacement policy. The prefetch cache stores disk blocks that have been prefetched but have not yet been referenced. A block "moves" from the prefetch to the demand cache when it is accessed. When a new prefetch or demand fetch is initiated, a buffer must be reclaimed from either the demand cache or the prefetch cache.

Other assumptions shared with the informed prefetching scheme include the following. An application issues I/O requests as single block requests that can be read in a single disk access. We assume disk access time is a constant, T_{disk}. We also assume that we have many disk drives and, therefore, no disk congestion. Between two I/O operations, the CPU performs computation for time T_{CPU}, on average. When the CPU finds the data it wants in the buffer cache, it takes T_{hit} time to read the block from the cache. Initiating a prefetch or a demand fetch requires device driver overhead T_{driver} to allocate a buffer, queue the request at the drive, and service the interrupt after the I/O completes. Figure 3(a) shows these parameters when no prefetching is performed.

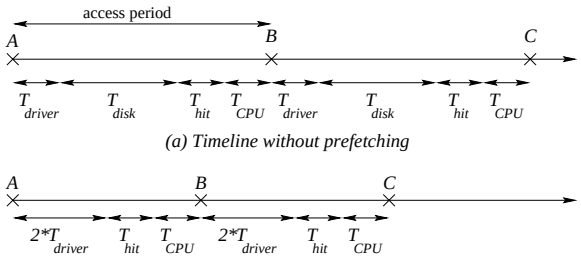
Additional parameters are required in our predictive prefetching scheme. We define an *access period* as the time between two successive disk accesses in the absence of prefetching. When consulting the prefetch tree, we can prefetch along multiple paths simultaneously. On average, at every step in the algorithm, we prefetch s blocks that may correspond to multiple paths in the tree. We calculate the value for s during execution. Figure 3(b) illustrates a prefetching timeline for $s = 2$.

We already defined the depth or *distance*, d_b, of a candidate block b as the number of edges along a path from the current position in the prefetch tree to the candidate block. Since not all prefetched blocks are accessed in our predictive scheme, we also define the hit ratio, h, as the fraction of prefetched blocks that are accessed. s and h are dependent; when we prefetch more blocks (increasing s), the prefetch hit ratio h decreases.

Finally, to make it possible to compare our cost and benefit estimates using the same units, we express all cost and benefit values as those achieved per unit of buffer usage. Using Patterson's definition of buffer usage or *bufferage*, we define one unit of bufferage as the occupation of one buffer for one access period [15].

4 Algorithm Overview

Recall that an access period is the time between two successive disk accesses. Our algorithm performs the following steps at the beginning of each access period.



(a) Timeline without prefetching

(b) Timeline with prefetching (s = 2)

Figure 3: *Effect of prefetching on the execution timeline. (a)* **Timeline without prefetching.** *In each access period, there is a $T_{driver} + T_{disk}$ time before the block is demand fetched into the cache. Once the block is available, the CPU takes $T_{hit} + T_{CPU}$ time computing. (b)* **Timeline with prefetching and no CPU stalls when ($s = 2$).** *Each block is prefetched sufficiently early so that the disk access time is overlapped with computation. In each access period, the CPU prefetches 2 blocks ($s = 2$) which takes $2 \times T_{driver}$ time. The amount of time spent computing does not change.*

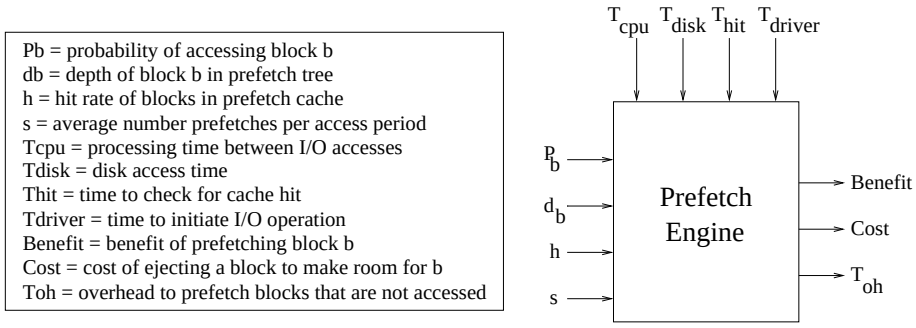


Figure 4: *Block Diagram for Prefetching Scheme. Left inputs are dynamically calculated, top inputs are constants.*

1. **Choose block to prefetch.** Consult the prefetch tree and identify blocks with a high probability of future access. For each block, calculate the benefit of prefetching. Choose the block with the greatest benefit from prefetching.
2. **Identify cache block to replace.** Identify the least valuable buffer in the demand cache or the prefetch cache. This block is a candidate for replacement.
3. **Decide whether to prefetch.** If the benefit of prefetching the new block exceeds the cost of replacing the old block, perform the prefetch.
4. Repeat these steps, prefetching multiple blocks, until the cost of replacing an existing block exceeds the benefit of prefetching a new block.

Figure 4 shows a block diagram of the inputs and outputs of our prefetching algorithm. It includes fixed values such as T_{driver} and T_{disk}, as well as dynamically calculated values such as s, h, and p_b, the probability of accessing block b. Outputs of the prefetching scheme include the cost and benefit of prefetching block b as well as the additional overhead, T_{oh}, incurred to initiate prefetches of blocks that are never accessed.

5 Benefit of prefetching a buffer

In this section, we derive an equation for the benefit of allocating an additional memory buffer to prefetch one access deeper. Allocating this buffer will increase the *bufferage*, or occupation of buffer space, by one unit of buffer usage per access period [15, 14]. Therefore, $bufferage = 1$.

Given this additional buffer for prefetching, we prefetch block b at a depth of d_b in the prefetch tree, with respect to the block currently being accessed. Block b has a probability of p_b of being accessed, according to the prefetch tree. As part of the benefit calculation, we define $\Delta T_{pf}(b, d_b)$ as the amount of time saved by prefetching block b at depth d_b, compared to

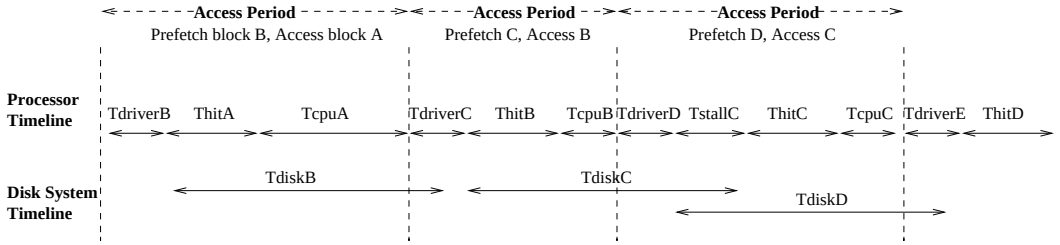


Figure 5: **Prefetching timelines:** *This figure shows timelines for a processor and disk system that are prefetching one block ahead of the block being accessed.* **(1) Access period 1:** *Initiate a prefetch of block B and access block A, which was previously prefetched. No processor stall.* **(2) Access period 2:** *Initiate the prefetch of block C and access block B. Disk access time for block B was completely overlapped by CPU activity. No processor stall.* **(3) Access period 3:** *Initiate the prefetch of block D and access block C. Processor stalls waiting for block C disk access to complete. Disk accesses for blocks C and D partially overlap; access of block D does not cause a stall in the following access period.*

the time required to fetch b on demand. Suppose that block b has a parent block x that is accessed with probability p_x. The time saved by prefetching one block deeper to fetch block b in addition to block x is $p_b \Delta T_{pf}(b, d_b) - p_x \Delta T_{pf}(x, d_b - 1)$. The benefit of allocating a buffer to prefetch one access deeper becomes:

$$B(b) = \frac{p_b \Delta T_{pf}(b, d_b) - p_x \Delta T_{pf}(x, d_b - 1)}{buffer age} = p_b \Delta T_{pf}(b, d_b) - p_x \Delta T_{pf}(x, d_b - 1) \quad (1)$$

We calculate $\Delta T_{pf}(b, d_b)$ by determining the amount of prefetch disk access time that is overlapped with other activity on the processor. Ideally, the disk access for a prefetched block completely overlaps computation on the processor or other I/O operations, effectively hiding the prefetch access time. If the entire prefetch is hidden, then the time saved by prefetching the block is the disk access time, or $\Delta T_{pf}(b, d_b) = T_{disk}$. In practice, the prefetch may not be initiated sufficiently early to hide the full disk access time. In this case, the CPU must stall while the disk access completes, and the time saved by prefetching will be

$$\Delta T_{pf}(b, d_b) = T_{disk} - T_{stall}(d_b) \quad (2)$$

$T_{stall}(d_b)$ is the average amount of time the CPU stalls waiting for a disk access to complete. Note that if $d_b = 0$, we are performing a demand fetch rather than a prefetch; we must stall for the entire disk access time, so $T_{stall}(0) = T_{disk}$, and $\Delta T_{pf}(b, 0) = 0$. Figure 5 illustrates the processor and disk system timelines for a series of prefetch operations. The figure shows one prefetch disk access that completely overlaps processor activity and does not stall the processor and another access that does cause a stall.

Before calculating $T_{stall}(d_b)$ for $d_b > 0$, we first determine the amount of prefetch disk access time that is overlapped with processor activity. During each of the d_b access periods between initiating the prefetch and accessing block b, the processor performs computation (which takes T_{CPU} on average, according to our system model), reads the currently-requested block from the buffer cache (T_{hit}), and initiates on average s additional prefetches ($s * T_{driver}$). Thus, the total computation over the d_b access periods is:

$$T_{compute}(d_b) = d_b[T_{CPU} + T_{hit} + sT_{driver}],\ for\ d_b > 0 \quad (3)$$

Stall time can now be bounded by:

$$0 \leq T_{stall}(d_b) \leq T_{disk} - T_{compute}(d_b),\ for\ d_b > 0 \quad (4)$$

If $T_{compute}(d_b) \geq T_{disk}$, then the entire prefetch disk access time is hidden, and $T_{stall}(d_b) = 0$.

Besides computation on the processor, stall time for a particular I/O can also be reduced by stalls generated by other I/O operations that are executing concurrently. During d_b access periods, a total of d_b blocks will be accessed. If the CPU stalls waiting for one disk access, additional I/O operations can proceed in the background and suffer less stall time. Figure 5 shows an example when disk accesses for blocks C and D overlap. On average, only one of d_b accesses will stall for $T_{disk} - T_{compute}(d_b)$ time. (Here we use logic similar to Patterson's [14].) Thus, on average, the stall time per block prefetched is given by:

$$T_{stall}(d_b) = \mathbf{max}[\frac{T_{disk} - T_{compute}(d_b)}{d_b},\ 0],\ for\ d_b > 0 \quad (5)$$

Substituting Equation 3 in Equation 5 yields

$$T_{stall}(d_b) = \mathbf{max}[\frac{T_{disk}}{d_b} - (T_{hit} + T_{CPU} + sT_{driver}),\ 0],\ for\ d_b > 0 \tag{6}$$

We substitute this value for T_{stall} into Equation 2 to calculate $\Delta T_{pf}(b, d_b)$. Finally, we substitute the value of $\Delta T_{pf}(b, d_b)$ into our original benefit equation, $B(b) = p_b \Delta T_{pf}(b, d_b) - p_x \Delta T_{pf}(x, d_b - 1)$.

By prefetching earlier (increasing d_b), a larger portion of the disk access time can be overlapped with computation. By prefetching more blocks per access period (increasing s), the CPU executes more driver operations to initiate these accesses, and masks a larger portion of any individual disk access. Both circumstances reduce average stall time and increases the benefit of prefetching. However, prefetching more blocks that are never accessed will ultimately harm performance. We account for the overheads incurred in increasing the number of prefetches in Section 6.3.

6 Cost

Next, we give expressions for the costs associated with prefetching. These include the cost of ejecting a buffer from either the prefetch cache or the demand cache to make room for the prefetched block, as well as the additional overhead in the predictive prefetching scheme for fetching blocks that are never accessed.

6.1 Ejecting a block from the prefetch cache

Blocks in the prefetch cache have been predicted but not yet accessed. If a block is removed from the prefetch cache and later accessed or predicted again, it must be re-fetched. Thus, the cost of ejecting a block from the prefetch cache relates to the penalty for re-fetching and the probability, p_b, that the block will be re-fetched:

$$C_{pr}(b) = \frac{p_b \Delta T_{rf}(b)}{bufferage} \tag{7}$$

We will first derive an expression for bufferage using Patterson's analysis [15]. The change in service time for ejecting a block from the prefetch cache is a one-time cost that is borne by the next access to the ejected block. If a block b is ejected, and if we find block b below the current position in the prefetch tree at depth d_b, then we expect to access the block again in d_b access periods. We can initiate a new prefetch of the block x access periods before its use. When we eject the block and later prefetch it again, a single buffer is freed for $d_b - x$ access periods. Thus, $bufferage = d_b - x$.

Next, we calculate $\Delta T_{rf}(b)$, the extra time required to re-fetch a block vs. finding it in the prefetch cache:

$$\Delta T_{rf}(b) = T_{re-fetch}(b) - T_{hit} \tag{8}$$

T_{hit} is the time required to find the block in the buffer cache. $T_{re-fetch}(b)$ is the time to re-fetch a discarded block. $T_{re-fetch}(b)$ includes the time to initiate a disk access, possible CPU stall time waiting for the disk access to complete, and the time to access the buffer in the cache. The CPU stall time is $T_{stall}(x)$, where x is the distance of the block in the tree at the time of the re-fetch. We use equation 6 to calculate $T_{stall}(x)$.

Re-fetch time becomes:

$$T_{re-fetch}(b) = T_{driver} + T_{stall}(x) + T_{hit} \tag{9}$$

Substituting the value of $T_{re-fetch}(b)$ in Equation 8 gives:

$$\Delta T_{rf}(b) = T_{driver} + T_{stall}(x) \tag{10}$$

Substituting the bufferage value ($d_b - x$) along with the value of $\Delta T_{rf}(b)$ into Equation 7, the cost of ejecting block b from the prefetch cache becomes:

$$C_{pr}(b) = \frac{p_b(T_{driver} + T_{stall}(x))}{d_b - x} \tag{11}$$

Finally, we substitute the value for T_{stall} from Equation 6.

6.2 Ejecting a block from the demand cache

Blocks in the demand cache have already been accessed and may be accessed again. The cost of ejecting a block is the probability of re-access multiplied by the penalty for re-fetching the block on demand. (For simplicity, we assume the discarded block will not later be prefetched.) The demand cache uses a least-recently-used (LRU) replacement strategy.

Let $H(n)$ denote the hit rate of a demand cache of size n [15]. The reduction in cache hit rate caused by removing one buffer from the demand cache is $H(n) - H(n-1)$. Logically, this difference in hit rate is due to accesses to the least-recently-used block in the size n cache. Shrinking the number of buffers in the cache by one reduces the number of occupied buffers, or *bufferage*, by one per access period. Thus, $bufferage = 1$, and the cost of ejecting a block from the demand cache becomes:

$$C_{dc}(n) = \frac{(H(n) - H(n-1))(T_{miss} - T_{hit})}{bufferage} = (H(n) - H(n-1))(T_{miss} - T_{hit}) \tag{12}$$

Since $T_{miss} = T_{driver} + T_{disk} + T_{hit}$, this cost becomes

$$C_{dc}(n) = (H(n) - H(n-1))(T_{driver} + T_{disk}) \tag{13}$$

Cost equations 11 and 13 also determine the best buffer to replace during a demand fetch operation.

6.3 Prefetching overhead

To this point, we have not considered the overhead of issuing prefetch requests. We define this overhead as the time required to initiate new prefetch requests for blocks that are not eventually accessed. If we ignore this overhead, we risk initiating too many prefetch accesses. To calculate this overhead, we must multiply the probability that a prefetched block is not accessed by the time to initiate a prefetch, T_{driver}.

Assume that block b, our prefetch candidate, is one access deeper in the prefetch tree than another block x. Recall that p_x and p_b are the probabilities of blocks x and b being accessed, respectively, and that these probabilities are calculated by multiplying the probabilities along the edges in a path in the tree. The probability that block x is accessed but block b is not accessed is $1 - \frac{p_b}{p_x}$. The incremental overhead of issuing a prefetch request for block b if it is never accessed is:

$$T_{oh} = (1 - \frac{p_b}{p_x})T_{driver} \tag{14}$$

Another overhead that we ignore in our model is that of disks spending time fetching blocks that are never accessed. For simplicity, we assume an infinite number of available disks and no wait time for disk accesses.

7 Cost benefit analysis

As discussed in the algorithm overview, our prefetching scheme has three parts:

1. **Choose block to prefetch.** We consult the prefetch tree to find candidates for prefetching based on their probability of future access. We apply Equation 1 to find the block, b, with the greatest benefit, $B(b)$, from prefetching.
2. **Identify cache block to replace.** We use equations Equations 11 and 13 to determine the best buffers to replace in the prefetch and demand caches, respectively. Between these two blocks, we choose the one with the lower cost, C, of replacement.
3. **Decide whether to prefetch.** Compare the benefit of prefetching a block b to the cost of ejecting a block from one of the caches. Account for prefetching overhead, T_{oh}. Prefetch block b if $B(b) - T_{oh} \geq C$.
4. Repeat these steps, prefetching multiple blocks, until the above condition is not satisfied.

8 Implementation

We implemented our predictive prefetching scheme in a trace-driven simulator written in C and C++. Simulations run on a Sun UltraSparc workstation under the Solaris operating system.

8.1 System Model

We model a relatively simple uniprocessor running a modern operating system, for easier comparison with Patterson's similar model [14] and to reduce complexity of the simulator. The system includes a file buffer cache that is partitioned into a *demand cache* and a *prefetch cache*. The demand cache holds disk blocks that have been referenced previously; the prefetch cache stores disk blocks that have been prefetched but have not yet been referenced. A block "moves" from the prefetch to the demand cache when it is accessed. When a new prefetch or demand fetch is initiated, a buffer must be reclaimed from either the demand cache or the prefetch cache.

Our simulator shares several other assumptions with the informed prefetching scheme. In both schemes, an application issues I/O requests as single block requests that can be read in a single disk access. We assume disk access time is a constant, T_{disk}. We also assume that we have many disk drives and, therefore, no disk congestion. Between two I/O operations, the CPU performs computation for time T_{CPU}, on average. When the CPU finds the data it wants in the buffer cache, it takes T_{hit} time to read the block from the cache. Initiating a prefetch or a demand fetch requires device driver overhead T_{driver} to allocate a buffer, queue the request at the drive, and service the interrupt after the I/O completes. We use the same constant parameters as Patterson [14], namely $T_{hit} = 0.243$ milliseconds, $T_{driver} = 0.580$ milliseconds, and $T_{disk} = 15.0$ milliseconds. We use a value of $T_{cpu} = 50$ milliseconds. In Section 9.2.3, we vary T_{cpu} between 20 and 640 milliseconds.

8.2 Traces

We use several sets of traces as input to our simulator. Table 1 contains a brief description of the traces we used in the simulation study. One disadvantage of using the disk block level traces, `cello` and `snake`, is that they do not provide a complete record of I/O accesses. In particular, these traces do not contain I/O accesses that were hits in the original system's file buffer cache. For the object references in the `CAD` trace, we do not know the size of the objects referenced.

Trace	*Number of references*	*L1 Cache Size*	*Description*
`cello`	3530115	30 MB	Disk block traces from a timesharing system [17]
`snake`	3867475	5 MB	Disk block traces from a file server [17]
`CAD`	147345		Object references from a CAD tool [5]
`sitar`	664867		File block traces of normal daily usage of students [6]

Table 1: *Traces used in the study. The* `snake` *and* `cello` *traces do not contain I/O accesses that were hits in the original system's file buffer cache. We do not know the size of the objects referenced in the* `CAD` *trace.*

9 Performance

To evaluate the performance of our tree-based, predictive prefetching scheme using cost-benefit analysis, we compared the performance of the following algorithms:

- **no-prefetch** performs no prefetching.
- **next-limit** always prefetches the next disk block after a block is fetched on-demand. Since this aggressive scheme prefetches many blocks, we limit the fraction of the cache devoted to prefetch blocks to 10% to avoid harming performance.
- **tree** is our basic algorithm that chooses prefetch candidates using a prefetch tree and decides to prefetch using cost benefit analysis.
- **tree-next-limit** Uses our cost benefit analysis in combination with the *next-limit* algorithm. Thus, this scheme always prefetches the block after a demand fetch, while limiting 10% of the cache for these blocks. In addition, it maintains a prefetch tree and prefetches additional blocks according to our cost benefit analysis.

9.1 Comparison of Prefetching Schemes

In this section, we present simulation results comparing the four basic schemes: *no-prefetch*, *next-limit*, *tree* and *tree-next-limit*. Figure 6 shows the performance of these algorithms for the four traces used in this study. The vertical axis of each graph shows the miss rate in the combined demand and prefetch caches. In all cases, the prefetching strategies offer significant performance improvements over the system that performs no prefetching.

Figure 6 suggests that the reduction in miss rate of *tree-next-limit* compared to *no-prefetch* is the *sum* of the reduction in miss rates of *tree* and *next-limit* compared to *no-prefetch*. Recall that the *next-limit* prefetching scheme always prefetches the next sequential block. Therefore, we expect the *next-limit* scheme to perform well for workloads that contain a substantial number of sequential accesses by reducing the number of misses that occur the first time a block is accessed (often called *compulsory misses*). By contrast, we expect the *tree* prefetching scheme to reduce cache miss rate by making predictions based on access patterns that have previously occurred. These two types of misses are mutually exclusive. In the discussion below, we attribute improvements in the miss rate to either one-block-lookahead prefetching or tree-based predictive prefetching.

For the `cello` timesharing and the `snake` disk block traces, the *tree-next-limit* algorithm performs best, reducing miss rates by up to 54% compared to the *no-prefetch* case. The *next-limit* algorithm also performs well for these traces, reducing miss rates by up to 32%. For both traces, the *tree* prefetching scheme is less effective. As mentioned above, the benefits due to the *tree* and *next-limit* prefetching schemes are additive and for large cache sizes most of the benefit comes from the *next-limit* prefetching scheme.

For the `CAD` trace, which captures object references in a CAD system, the *tree* and *tree-next-limit* algorithms have similar performance. However, for this trace, the *next-limit* scheme performs no better than the *no-prefetch* scheme. This suggests that there is little sequential access in the object references, so that one-block-lookahead prefetching is ineffective. By contrast, our tree-based prefetching scheme proves very successful in predicting non-sequential accesses, reducing cache miss rates by up to 36%.

Finally, for the `sitar` file level traces, the *tree-next-limit* and *next-limit* algorithms perform similarly, reducing cache miss rates by up to 73%. By contrast, the basic tree algorithm performs poorly, producing about the same miss rate as the *no-prefetch* case. This suggests that the trace has a great deal of sequential access, so that prefetching algorithms that perform one-block-lookahead are very effective.

We have shown that the *tree-next-limit* prefetching scheme, which combines one-block-lookahead prefetching with predictive prefetching based on past accesses, produces good performance for a variety of workloads. For workloads with substantial sequential accesses, the scheme performs as well as a simple one-block-lookahead prefetching scheme. In addition, for workloads without much sequential access, our tree-based prediction scheme is able to effectively predict future accesses based on past access patterns.

9.2 Understanding Performance of the *tree* Prefetching Scheme

As mentioned above, Figure 6 suggests that the reduction in miss rate of *tree-next-limit* compared to *no-prefetch* is the sum of the reduction in miss rates of *tree* and *next-limit* compared to *no-prefetch*. Hence, we expect any improvements to the *tree* prefetching scheme to provide similar improvements to the *tree-next-limit* prefetching scheme. In the next few sections we analyze the performance of the *tree* prefetching scheme with the goal of improving its performance.

9.2.1 Increasing Cache Size

Figure 6 indicates that although the *tree* prefetching scheme is quite effective compared to *no-prefetch* in reducing cache miss rates for smaller cache sizes, the advantage of prefetching declines as cache sizes increase. In larger caches, a bigger fraction of the working set stays resident in the cache, reducing the effectiveness of the *tree* prefetching scheme. Most misses in large caches occur when a block is first accessed (often called *compulsory misses*). The basic *tree* algorithm cannot further reduce compulsory misses (unlike the *next-limit* and *tree-next-limit* algorithms which prefetch the next sequential block).

Figure 7 supports the assertion that the working sets of the `cello`, `snake`, `CAD` and `sitar` traces fit in cache sizes of 2048 blocks and above. This graph shows the fraction of blocks that the prefetch algorithm chooses to prefetch, only to discover that the blocks already exist in either the demand cache or the prefetch cache. For all three traces, over 85% of the blocks identified as prefetch candidates already reside in the cache for cache sizes above 2048 blocks.

As a result, the *tree* algorithm performs less prefetching at larger cache sizes, as shown in Figure 8. The graph shows the average number of blocks prefetched per access period for the four traces. For small cache sizes, the frequency of

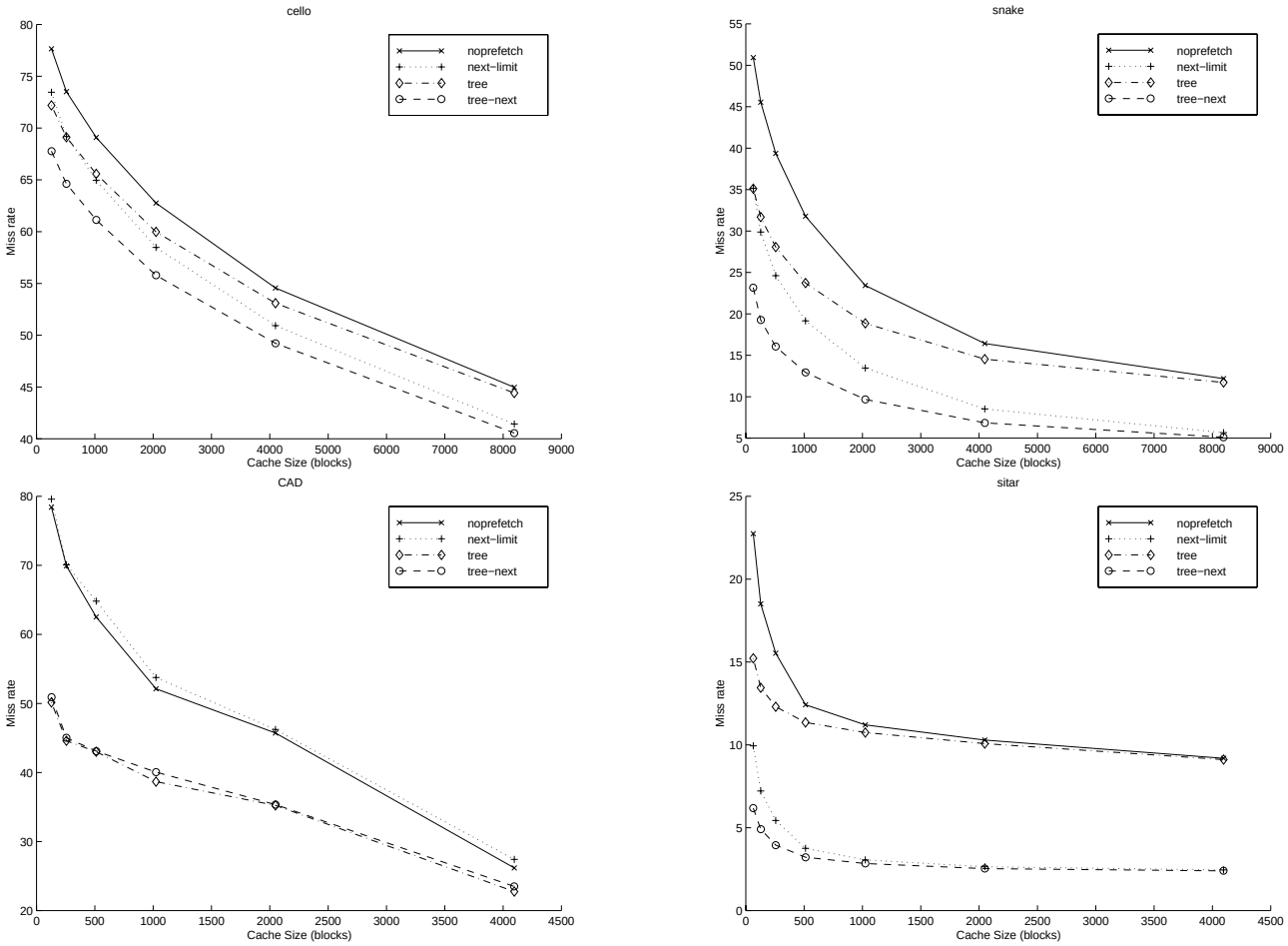


Figure 6: *For the four traces, compares the performance of no-prefetch, next-limit and tree-next-limit. With one exception, tree-next-limit has the lowest miss rate for all traces and cache sizes.*

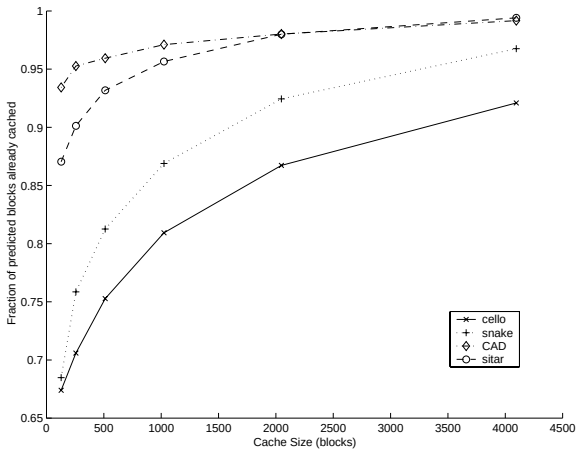


Figure 7: *The fraction of prefetch candidates chosen by the cost/benefit algorithm that already reside in the cache, as the size of the combined demand and prefetch cache increases.*

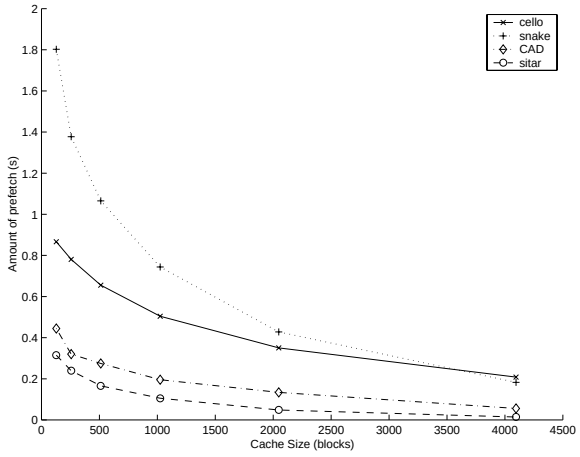


Figure 8: *The number of blocks prefetched by our prefetching scheme per access period for the traces, as the size of the combined demand and prefetch cache increases. These prefetched blocks contribute to an increase in the amount of disk traffic.*

prefetching for the snake trace is around two blocks prefetched per I/O access, while for the remaining traces it is much lesser. Thus, for the snake trace prefetching accounts for a 180% increase in disk traffic for small cache sizes. At larger cache sizes, the *tree* algorithm prefetches fewer than one block every three access periods for all traces considered.

9.2.2 Hit Rate in the Prefetch Cache

Next, we study the effectiveness of prefetching by considering the percentage of prefetched blocks that are hit in the prefetch cache, the *prefetch cache hit rate*.

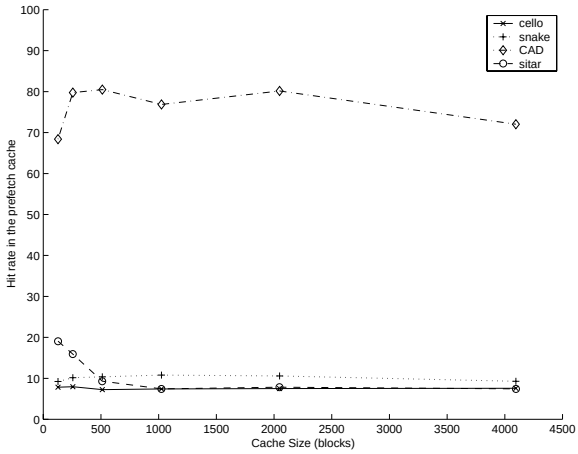


Figure 9: *The hit ratio of the prefetch cache for the traces, as the size of the combined demand and prefetch cache increases.*

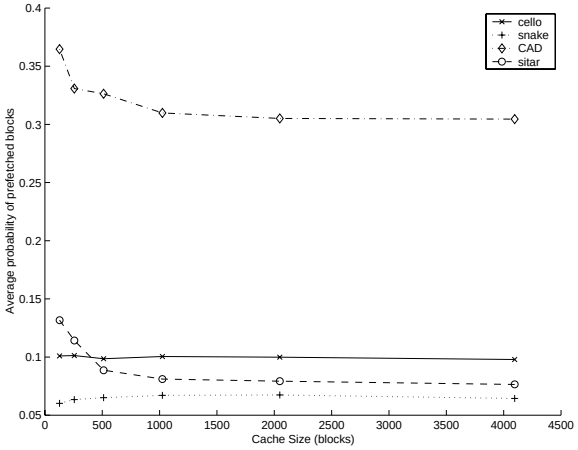


Figure 10: *The average probability of the prefetched block, as the size of the combined demand and prefetch cache increases.*

Figure 9 shows the prefetch cache hit rate for all traces. For the CAD trace, the hit rate in the prefetch cache is around 75%. As can be seen in Figure 6, this results in a significant reduction in the miss rate for the CAD trace. For the remaining traces, the prefetch cache hit rate is low, around 10%. This suggests that the basic *tree* algorithm prefetches many blocks that are either never accessed or are discarded from the prefetch cache before being accessed. Since the prefetch cache hit rate is relatively low, we are working on strategies to reduce the number of blocks prefetched by eliminated mispredicted blocks.

The high prefetch cache hit rate for the CAD trace is due to the higher probability of the prefetched blocks in the CAD trace. Figure 10 shows that the average probability of the blocks prefetched in the CAD is higher than the average probability of the blocks prefetched in the remaining traces.

9.2.3 Varying T_{cpu}

Finally, we study the impact of changing the amount of time the processor spends computing between successive I/O accesses (T_{cpu}).

Figure 11 shows the impact of changing the value of T_{cpu} for the CAD trace (the effect of changing T_{cpu} is similar for the remaining traces). The graph shows s, the average amount of prefetching performed per access period. This simulation used a cache size of 1024 blocks and varied T_{cpu} from 20 to 640 milliseconds. As the value of T_{cpu} increases, the number of blocks prefetched per access period increases initially and then stays fairly constant for larger values of T_{cpu}. The number of blocks prefetched initially increases with T_{cpu} because more I/Os can execute concurrently without stalling the processor. However, as we prefetch more blocks, the overhead of prefetching increases. Eventually, the cost of ejecting a block from the cache exceeds the benefit of prefetching a new block, and the rate of prefetching remains constant.

This is confirmed by Figure 12 which shows the hit rate for blocks in the prefetch cache for the CAD trace as the value of T_{cpu} is changed. The figure shows that the hit rate for blocks in the prefetch cache initially decreases substantially with increasing T_{cpu} and remains around 74% for values of T_{cpu} over 50 milliseconds. However, we noticed that the miss rate for the combined demand and prefetch caches does not change significantly with increasing T_{cpu}.

Since cache miss rates are relatively insensitive to values of T_{cpu} above 50 milliseconds, the results presented earlier used a value of $T_{cpu} = 50.0$ milliseconds.

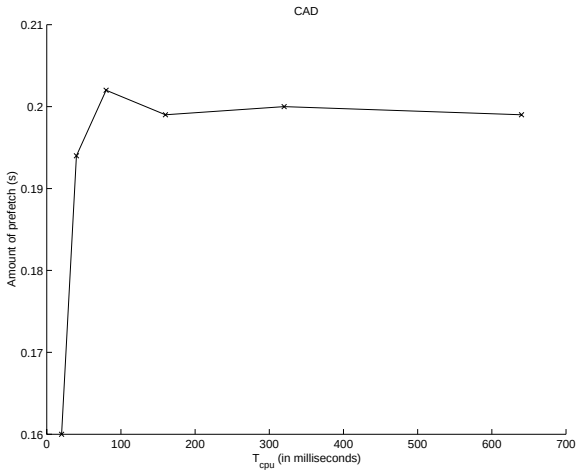


Figure 11: *Shows the amount of prefetching performed per access period as the amount of computation between I/O accesses, T_{cpu}, increases.*

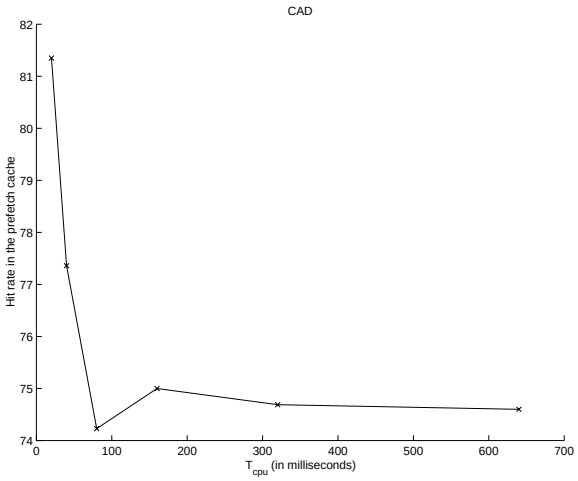


Figure 12: *Shows that the hit rate in the prefetch cache for the three traces decreases as the amount of computation between I/O accesses, T_{cpu}, increases.*

9.3 Characterizing Memory Usage of the Algorithms

One concern in using our prefetching algorithm is the amount of memory required to construct the prefetch tree. For the `CAD` trace, we found that the *tree* prefetching scheme was very effective in reducing the miss rate using a modest amount of memory of approximately a megabyte.

To determine the amount of memory required, we built a version of the simulator that limits the size of the prefetch tree. Recall that the prefetch tree is constructed by dividing the access pattern into substrings, as described in Section 2. We limit the size of the prefetch tree by maintaining these substrings in a least-recently-used list. We eliminate nodes from the tree by discarding the least recently used node from this list.

Figure 13 shows the performance of the *tree* prefetching algorithm when the amount of memory for the prefetch tree is limited. Over a range of cache sizes, the best performance of *tree* for the `CAD` trace is observed when the prefetch tree uses 32K nodes. In our simulation, each node corresponds to 40 bytes of storage (This can be reduced to 26 bytes by replacing 7 of the pointers by short integers). For 32K nodes, this corresponds to approximately 1.25 megabytes of memory.

9.4 Predictability of blocks in the prefetch tree

In this section, we evaluate the accuracy of the prefetch tree in predicting blocks that are going to be accessed. We find that in most cases, the prefetch tree is fairly accurate in predicting future blocks, predicting around 60-70% of the blocks accessed.

Trace	Prediction Accuracy
`cello`	35.78%
`snake`	61.50%
`CAD`	59.90%
`sitar`	71.39%

Table 2: *Prediction accuracies for the traces considered. Between 60-70% of all accesses can be predicted for the* `snake`*,* `CAD` *and the* `sitar` *traces.*

We define a block request to be *predictable* if it is present as a child node of the current node in the prefetch tree. Table 2 shows the *prediction accuracy*, the percentage of accesses that are predictable, for the traces considered. The table shows that for the `snake`, `CAD` and `sitar` traces, between 60-70% of all accesses are predictable. For the `cello` trace, the prediction accuracy is lower because the 30 MB first level cache captures most of the locality in the trace.

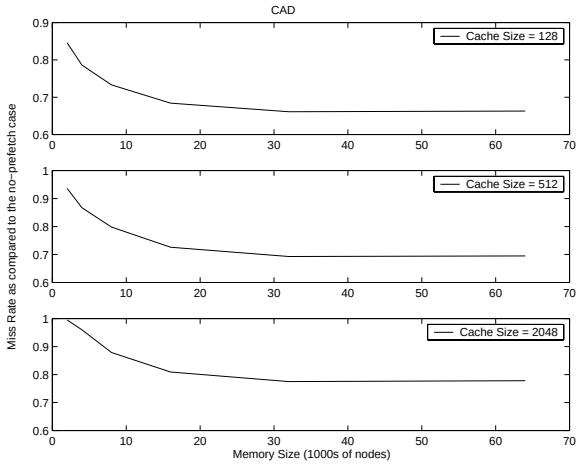


Figure 13: *For the* `CAD` *trace, plots the performance of the tree algorithm while limiting the amount of memory used for the prefetch tree. The y-axis represents the miss rate of tree as a fraction of the miss rate of no-prefetch. In the current implementation, each node corresponds to 40 bytes of storage.*

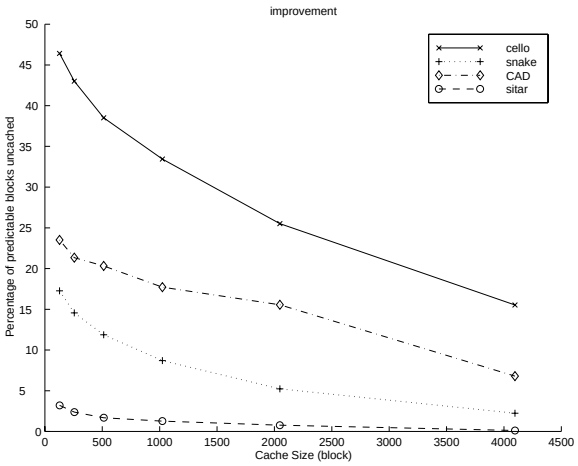


Figure 14: *For the traces considered, this figure plots the percentage of predicted blocks that are not cached by the tree prefetching scheme. For the* `snake`, `CAD` *and* `sitar` *traces most of the predictable blocks are already cached.*

Figure 6 shows that this high *prediction accuracy* for the `snake`, `CAD` and `sitar` traces results in the *tree* prefetching scheme reducing the miss rate by 33% compared to *no-prefetch* for small caches. The `cello` trace does not show similar improvements in miss rate due to its low prediction accuracy.

To determine the potential for improvement in miss rate using the *tree* prefetching scheme, we looked at the percentage of these predictable blocks that are already cached by the *tree* prefetching scheme. Figure 14 shows the percentage of predictable blocks that are *not* already cached by the *tree* prefetching scheme. For the `snake`, `CAD` and `sitar` traces, the fraction of predictable blocks that are not cached is low (around 15%). This suggests that although the prefetch tree does a good job of identifying prefetch candidates, most of these blocks are already cached by the *tree* algorithm. In the next section, we look at the possibility of improving the *tree* algorithm.

9.5 Potential for Improving the *tree* Algorithm

In this section, we determine the maximum possible improvement in miss rate that could be achieved for the *tree* algorithm. In particular, we focused on improving the scheme used to select among prefetch candidate blocks. For all the traces, we found that by changing the selection scheme it is possible to significantly reduce the miss rate of the *tree* prefetching scheme.

To determine the maximum improvement in miss rate achievable using the prefetch tree, we considered a *perfect selection scheme*. The perfect selection scheme assumes knowledge of the next disk access. The resulting prefetching scheme, *perfect-selector*, uses the knowledge of the next disk access to prefetch the next disk access only if it is predictable, *i.e.* the disk access has been identified by the prediction scheme as a candidate for prefetching.

Figure 15 compares the miss rates of *no-prefetch*, *tree* and *perfect-selector* for the four traces. For all traces, the *perfect-selector* prefetching scheme reduces miss rates by a considerable amount compared to *tree*. This indicates that there is considerable potential for improving the selection scheme of the *tree* prefetching algorithm.

9.6 Prefetching the last visited child

We next describe a scheme we considered for improving the performance of the selection scheme used in *tree*.

We define the *last visited child* of a node in the prefetch tree as the child that was visited during the previous visit to the node. We determined the percentage of time the *last visited child* of a node is accessed on the next visit to the node, *i.e.* the same path is followed in the prefetch tree. Table 3 shows the percentage of time the *last visited child* of a node is accessed on the next visit to the node. For the `CAD` and the `sitar` traces, around 70% of the time the *last visited child*

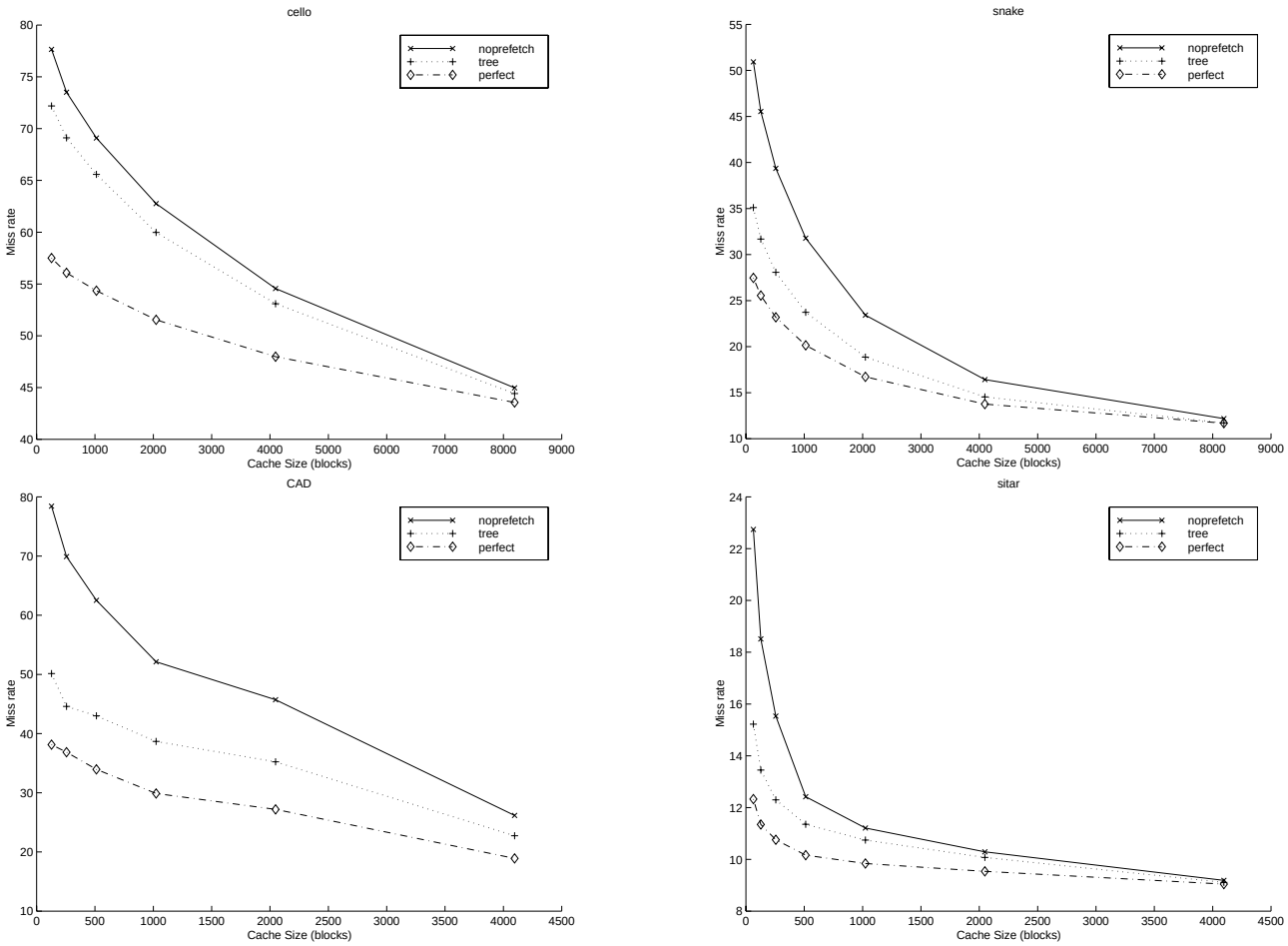


Figure 15: *For the four traces, compares the performance of no-prefetch, tree and perfect-selector. For all the traces considered, perfect-selector reduces miss rate by a considerable amount compared to tree.*

of a node is re-visited. This suggests that prefetching the *last visited child* of a node may reduce miss rate significantly. However, since the node was fetched the last time the parent node was accessed, it may already exist in the prefetch cache.

Trace	Percentage of successive visits to the last visited child
cello	24.37%
snake	38.49%
CAD	68.61%
sitar	73.61%

Table 3: *The percentage of time the* last visited child *of a node is accessed on the next visit to the node.*

To determine the usefulness of prefetching the *last visited child* of a node, we simulated an algorithm called *tree-lvc* which prefetches the *last visited child* of a node in addition to prefetching blocks determined by cost-benefit analysis.

A comparison of the *tree* and *tree-lvc* prefetching schemes revealed that there is no noticeable difference in the miss rates of *tree-lvc* and *tree* prefetching schemes. Figure 16 plots the percentage of the *last visited children* that are already cached by the *tree* prefetching scheme. For most cache sizes, more than 85% of the *last visited children* are already cached by the *tree* prefetching scheme.

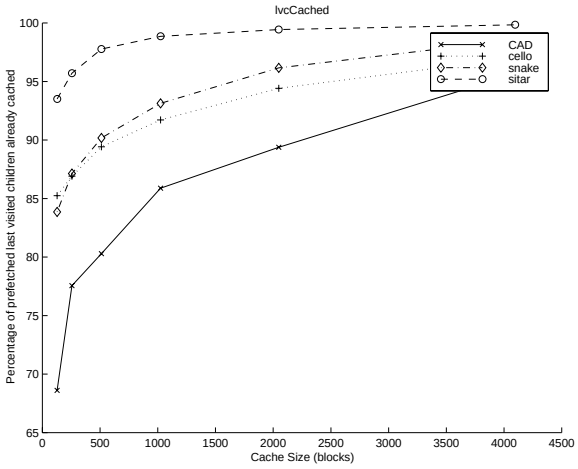


Figure 16: *For all the traces, shows the percentage of last visited children that are already cached by the tree prefetching scheme.*

Although prefetching the *last visited child* of a node proved ineffective, we are currently working on other ways of bridging the gap between the *tree* and the *perfect-selector* prefetching schemes.

9.7 Effectiveness of Cost-Benefit Analysis

Finally, we evaluate the effectiveness of the cost-benefit analysis by comparing the *tree* prefetching scheme with two other tree-based parametric prefetching schemes that do not perform cost-benefit analysis. We find that the performance of these prefetching schemes depends on the value of the parameters chosen. Further, we observe that with the help of the cost-benefit analysis, the *tree* prefetching scheme achieves miss rates similar to the best performance achieved by the parametric prefetching schemes.

The tree-based parametric prefetching schemes we considered are:

- **tree-threshold** After accessing a block in the prefetch tree, all child nodes with a probability of future access higher than a specified *probability threshold* are prefetched. This prefetching scheme is our implementation of the scheme proposed by Curewitz *et al.* in [5].

- **tree-children** After accessing a block in the prefetch tree, a *fixed number of child nodes* with the highest probability of future access are prefetched. This prefetching scheme is our implementation of the scheme proposed by Kroeger and Long in [8].

Trace	*tree-threshold*				
	Best		*Worst*		*Difference*
	Miss rate	*Threshold probability*	*Miss rate*	*Threshold probability*	
cello	76.587	0.025	77.817	0.2	1.60%
snake	31.508	0.002	36.272	0.1	15.12%
CAD	52.147	0.025	60.030	0.008	15.11%
sitar	15.444	0.05	17.135	0.002	10.95%

Table 4: *Comparison of the best and worst performance of the* tree-threshold *prefetching scheme. The performance of* tree-threshold *can be up to 15% worse compared to its best performance.*

Table 4 summarizes the effect of varying the threshold probability for the *tree-threshold* prefetching scheme. We varied the threshold probability from 0.4 to 0.001 and noticed that there is no single value of the threshold probability for which the performance of *tree-threshold* is best for all the traces considered. Further, the difference between the best and worst performances of *tree-threshold* could differ by up to 15%. We observed a similar dependence of *tree-children* on the number of children prefetched (The optimal value of the number of children prefetched ranged from 3 to 10). This suggests that the parameters for the prefetching schemes *tree-threshold* and *tree-children*, the threshold probability and the number of children prefetched respectively, should be chosen carefully.

Figure 17 compares the performance of the three algorithms, *tree*, *tree-threshold* and *tree-children*. Note that this comparison is with the best performance of *tree-threshold* and *tree-children*. The performance of *tree* is similar to the best performance of *tree-threshold* and *tree-children*, suggesting that the cost-benefit analysis performed by *tree* is dynamically able to perform the optimal amount of prefetching. One further detail worth mentioning is that the *tree* algorithm performs prefetching based on the computation to I/O ratio, while this is not the case with the prefetching schemes *tree-threshold* and *tree-children*.

10 Related Work

In this section, we briefly discuss how our prefetching algorithms relate to other work in prefetching.

Our cost-benefit analysis is most closely related to Patterson's informed prefetching [14]. In this scheme, I/O-intensive applications disclose hints to the operating system regarding which data blocks will be accessed in the future. Based on these deterministic hints, the informed prefetching scheme uses a cost-benefit analysis to determine whether it is beneficial to allocate a memory buffer to prefetch an additional block. There have been a number of extensions to the original informed prefetching scheme that focus on disk striping [13], network file systems [16], bursts of I/O activity [18], and automatic generation of hints [4]. Mowry *et al.* [11] also describes a scheme in which the compiler automatically inserts hints into the program.

The prefetch tree we use to predict future accesses uses an algorithm developed at Duke University [19] [5] that is adapted from a data compression technique. There are several other schemes that predict future accesses based on past accesses. They include a scheme based on a multi-order context model compression technique [8], using per-file hidden Markov models to predict future accesses [10], using associative memory and pattern recognition to identify access patterns [12], and prefetching whole files using a tree that records past file accesses [6, 9].

In additional prefetching work, Cao *et al.* discuss optimal prefetching and caching strategies [1] as well as application-controlled cache replacement [2, 3]. Kimbrel *et al.* [7] use disk idle periods to prefetch blocks that will be needed during periods of disk congestion. 4.4 BSD UNIX prefetches based on the layout of file data on disk; when a block is demand-fetched, other file blocks that are laid out contiguously are prefetched to avoid later seek operations.

In addition to these block-based prefetching schemes, there are a number of file-based prefetching schemes; these prefetch entire files based on predictions of correlated file access.

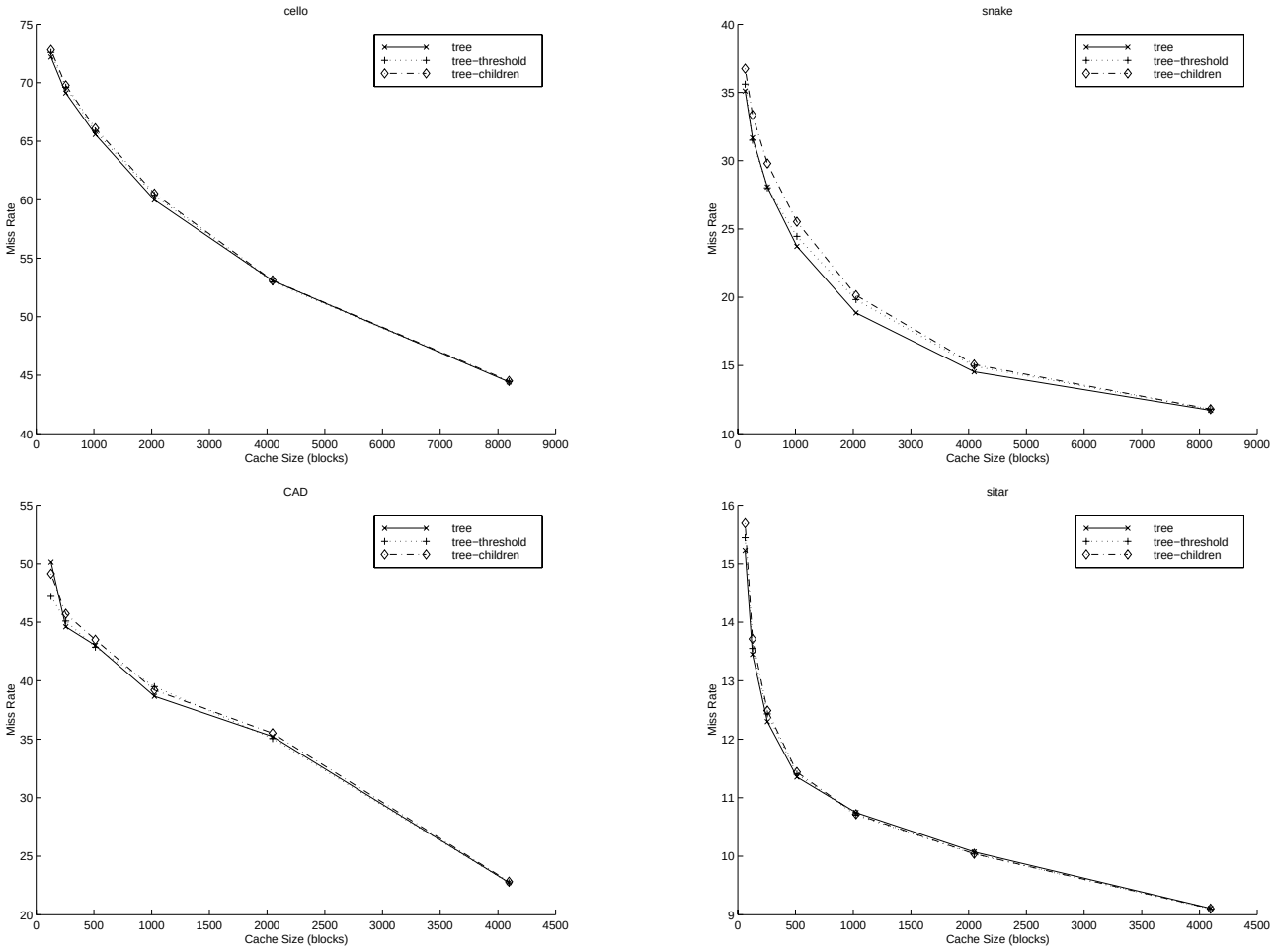


Figure 17: *For the* `cello` *and* `snake` *traces, compares the performance of tree, with the best performance of tree-children and tree-threshold. The tree prefetching schemes obtains miss rates similar to the best miss rates obtained by tree-children and tree-threshold.*

11 Conclusions

Prefetching schemes will become increasingly important in high-performance computing systems. Prefetching blocks in advance of their being requested by an application eliminates processor idle time and improves the performance and utilization of parallel I/O systems.

We described our scheme for prefetching using a cost-benefit analysis applied to predicted accesses. We decide which blocks to prefetch based on their probability of access; we calculate these probabilities from past access patterns using a prefetch tree. We then decide whether to prefetch a candidate block using a cost-benefit analysis. This analysis modifies the informed prefetching scheme [14, 15] to account for probabilistic hints. We implemented our algorithm efficiently using a trace-driven simulator and evaluated its performance.

Our *tree-next-limit* prefetching scheme combines one-block-lookahead prefetching with predictive prefetching based on past accesses. We evaluated this scheme using trace-driven simulations and showed that it performs well for a variety of workloads. For workloads with substantial sequential accesses, the *tree-next-limit* performs as well as a simple one-block-lookahead prefetching scheme, reducing cache miss rates by 73%. In addition, for workloads without much sequential access, our tree-based prediction scheme is able to effectively predict future accesses based on past access patterns, reducing cache miss rates by up to 36%.

We showed that the memory requirements of our algorithm were reasonable, requiring about a megabyte of memory for efficient performance. Next, we demonstrated that the prefetch tree was very effective in identifying potential candidates for prefetching, with the next block accessed identified as a candidate approximately 60 to 70% of the time. Finally,

we evaluated the effectiveness of the cost-benefit analysis by comparing the *tree* algorithm with two other tree-based prefetching schemes that do not perform cost-benefit analysis. We showed that with the help of the cost-benefit analysis, the *tree* algorithm achieves a performance similar to the best performance achieved by other tree-based prefetching schemes.

References

[1] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of 1995 ACM Sigmetrics*, pages 188–197, May 1995.

[2] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. *ACM Transactions on Computer Systems*, November 1996.

[3] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[4] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation*, February 1999.

[5] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM SIGMOD*, pages 257–266, May 1993.

[6] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proceedings of the 1994 USENIX Summer Technical Conference*, pages 197–207, June 1994.

[7] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward Felten, Garth Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34. USENIX Association, October 1996.

[8] T. M. Kroeger and D. D. E. Long. Predicting File System Actions from Prior Events. In *Proceedings of the 1996 Usenix Winter Technical Conference*, pages 319–328, January 1996.

[9] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 275–288, January 1997.

[10] Tara M. Madhyastha and Daniel A. Reed. Input/Output Access Pattern Classification Using Hidden Markov Models. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*, pages 57–67, November 1997.

[11] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, October 1996.

[12] M. Palmer and S. Zdonik. Fido: A Cache that Learns to Fetch. In *Proceedings of the 1991 International Conference on Very Large Databases*, September 1991.

[13] R. Hugo Patterson and Garth A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

[14] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[15] Russel Hugo Patterson. *Informed Prefetching and Caching*. PhD thesis, Carnegie Mellon University, December 1997. Technical Report No. CMU-CS-97-204.

[16] David Rochberg and Garth Gibson. Prefetching Over a Network: Early Experience with CTIP. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):29–36, December 1997.

[17] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 405–420, January 1993.

[18] A. Tomkins, R. Hugo Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of 1997 ACM Sigmetrics*, June 1997.

[19] Jeffrey Scott Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Foundations of Computer Science*, pages 121–130, 1991.