Prefetching Without Hints: A Cost-Benefit Analysis for Predicted Accesses *

Vivekanand Vellanki and Ann L. Chervenak
College of Computing, Georgia Tech
vivek@cc.gatech.edu, annc@cc.gatech.edu

Abstract

Prefetching disk blocks to main memory will become increasingly important to overcome the widening gap between disk access times and processor speeds. We present a prefetching scheme 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. To calculate the probability of access of prefetch candidates, we construct a prefetch tree that records past access patterns. For the cost-benefit analysis, we derive equations for the benefit of prefetching an additional block and the cost of allocating a buffer to the prefetch.

We use a trace-driven simulator to evaluate the performance of our prefetching scheme. For an efficient implementation of the prediction algorithm, we limit the size of the prefetch tree. We show that our prefetching scheme lowers overall cache miss rates by up to 32% compared to a system that performs no prefetching. Coupled with the one block lookahead prefetching, this scheme improves overall cache miss rates by up to 52% over a scheme that performs no prefetching and by up to 27% over an aggressive one block lookahead prefetching scheme for small cache sizes. However, the frequency and effectiveness of prefetching decrease as cache size grows.

1 Introduction

Prefetching disk blocks to main memory will be increasingly necessary to overcome the widening gap between disk access times and processor speeds. Disk access times, which include mechanical seek and rotation operations, improve at a rate of less than 10% per year. Typical disk access times are approximately 10 milliseconds in 1998. By contrast, processor speeds are improving at a rate of over 50% per year. For CPU clock rates of 100 MHz to 500 MHz, a disk access takes 1 million to 5 million processor cycles, and this disk access penalty will worsen as clock rates increase.

Prefetching blocks before a processor requests them can reduce or eliminate the time a processor is idled waiting for data to arrive from disk. Without prefetching, a processor initiates a *demand fetch* when it discovers the data it requires is not contained in caches or in main memory. Disk I/O operations are so lengthy that they frequently cause a processor to idle, or stop executing instructions. Prefetched disk blocks can make data available to the processor sooner and minimize idle time.

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 [18, 4, 5, 7, 9, 8, 11]. 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 [13, 14, 6, 17, 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 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 work was supported in part by NSF CAREER Award CCR-9702609.

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 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 [13, 14, 17]. It differs from informed prefetching in the two ways.

- *Probabilistic hints:* We infer probable future access patterns based on past accesses by constructing a prefetch tree. Because some predictions are incorrect, some blocks will be prefetched and never accessed. By contrast, in the informed prefetching scheme, applications are modified to provide deterministic hints about future access patterns. All hinted blocks are eventually accessed.

- *Prefetching blocks along multiple paths:* When we consult our prefetch tree to decide which blocks to prefetch, we may find more than one block with a moderate or high probability of being accessed soon. Our cost-benefit algorithm may prefetch multiple blocks corresponding to different paths in the prefetch tree.

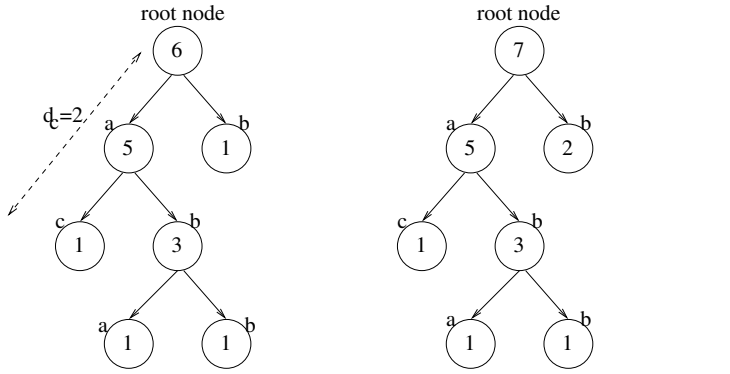
We describe an efficient implementation of our algorithm in a trace-driven simulator. Our prefetching scheme lowers overall miss rate up to 32% over a scheme that performs no prefetching for cache sizes that are small relative to the working set of blocks being accessed. Coupled with the one block lookahead prefetching, our predictive prefetching scheme improves overall cache miss rates by up to 52% over a scheme that performs no prefetching and by up to 27% over an aggressive one block lookahead prefetching scheme for small cache sizes. However, the frequency and effectiveness of prefetching decrease as cache size grows.

The remainder of this paper is organized as follows: In the next section, we explain how we predict future accesses based on past access patterns using a prefetch tree. Next, we explain the system model for our experiments. Section 4 gives an overview of the cost-benefit algorithm, followed by details of the analysis in Sections 5, 6 and 7. Section 8 briefly 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 [18, 4]. 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.



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.*

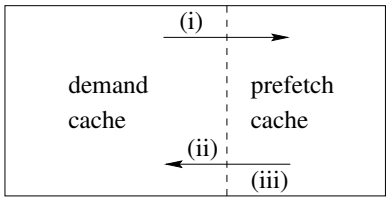


Figure 2: *Structure of the combined prefetch and demand cache. To accommodate a new prefetch block, a buffer must be reclaimed from either the demand cache (i) or the prefetch cache. Likewise, if a demand miss occurs, a buffer is reclaimed from either the prefetch cache (ii) or the demand cache. When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

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.

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 [13]. 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.

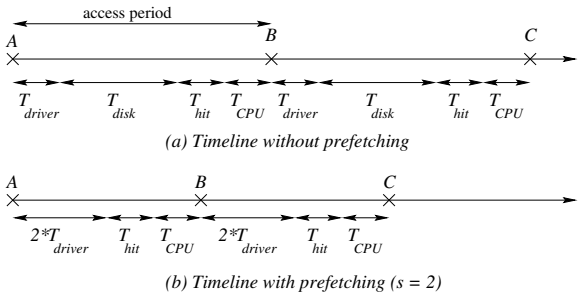


Figure 3: *Effect of prefetching on the execution timeline. (a)* **Timeline without prefetching.** *In each access period, there is a $T_{driver} + T_{disk}$ time before the block is demand fetched into the cache. Once the block is available, the CPU takes $T_{hit} + T_{CPU}$ time computing. (b)* **Timeline 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.*

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 [14].

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.

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 [14, 13]. 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 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

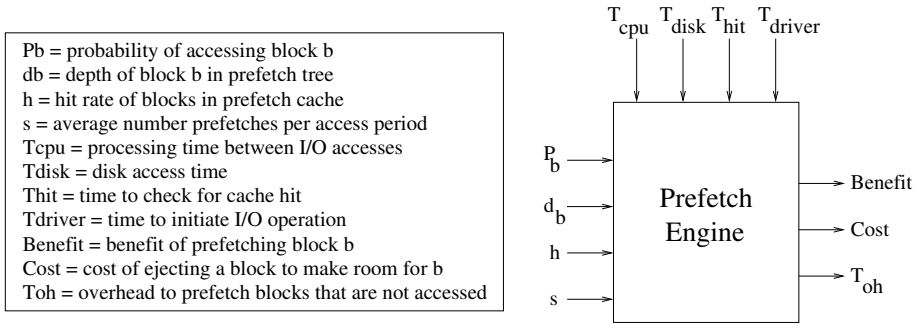


Figure 4: *Block Diagram for Prefetching Scheme. Left inputs are dynamically calculated, top inputs are constants.*

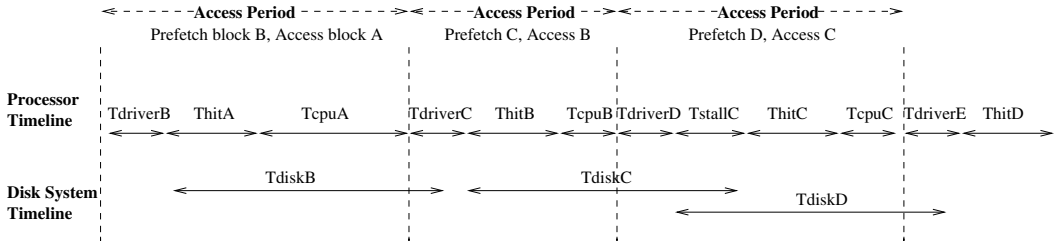


Figure 5: **Prefetching timelines:** *This figure shows timelines for a processor and disk system that are prefetching one block ahead of the block being accessed.* **(1) Access period 1:** *Initiate a prefetch of block B and access block A, which was previously prefetched. No processor stall.* **(2) 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.*

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:

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

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 [13].) Thus, on average, the stall time per block prefetched is given by:

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

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 refetched:

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

We will first derive an expression for bufferage using Patterson's analysis [14]. 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 refetch 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 $T_{stall}(x)$.

Refetch 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 [14]. 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. We use the same constant parameters as Patterson [13], namely $T_{hit} = 0.243$ milliseconds, $T_{driver} = 0.580$ milliseconds, and $T_{disk} = 15.0$ milliseconds. We vary T_{cpu} between 20 and 640 milliseconds.

8.1 Traces

As input to our simulator, we use the disk level traces generated by Ruemmler and Wilkes [16]. The traces contain disk accesses of 3 machines, `cello`, `snake` and `hplajw`. `cello` is a timesharing system with a file buffer cache of 30 MB for the traces used in this paper, while `snake` had a 5 MB file buffer cache and acted as a file server and `hplajw` is a personal workstation with a 3 MB file buffer cache.

A problem with using these traces for this study is that they are disk level traces. Thus, 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), but instead reflect only disk accesses. These disk accesses represent misses from the original system's file buffer cache. In Section 11 we discuss our experimental plans for file level traces.

8.2 Optimizing Simulator Performance

To reduce the time to construct the prefetch tree, to choose prefetch candidates from the tree, and to calculate the benefit and cost of allocating a buffer for prefetching, we make the following implementation decisions.

- We maintain the children of each node as lists ordered by decreasing probability of access and LRU order.
- On each iteration of the algorithm, we calculate the benefit of prefetching only one block at each level of the prefetch tree: the block with the highest probability of access of all the blocks not yet considered.
- We only consider blocks at lower levels of the tree if their parent nodes have previously been prefetched.
- The root node has many children; we implement this level of the tree as a hash table for fast access.
- To limit the size of the tree and reduce the number of nodes that need to be examined, we experiment with restricting the number of children allowed for a node in the tree.
- To limit the number of levels we check during prefetching, we use Patterson's notion of the *prefetch horizon* [13], which is the depth at which $T_{stall}(d_b) = 0$. We do not prefetch past the prefetch horizon.

- We experiment with two ways of assigning cost to "mispredicted" blocks. A mispredicted block is a prefetched block that is not accessed by the time we predicted it would be. (We maintain estimated access times for all prefetched blocks.) The first option for dealing with a mispredicted prefetch is to eject it from the prefetch cache as quickly as possible. In this case, the cost of replacing the mispredicted block is zero. Second, we consider that, although block b was not accessed at the predicted time, because of locality of reference in the prefetch tree, it may be referenced soon. As time advances, the probability of the block being accessed in the future decreases. The cost of ejecting the mispredicted block should reflect its *tardiness*, the number of access periods between the estimated access time and the current time. As a block's tardiness increases, the cost of ejecting it should decrease. To estimate the cost of ejecting a tardy block, we substitute the tardiness value for x in the cost equation (11).

9 Performance

To evaluate the performance of the prefetching algorithm we compared it with:

- **no-prefetch** peforms no prefetching.
- **next-limit** always prefetches the next disk block after demand fetch. 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 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 Algorithm Performance

In this section we evaluate the behavior of our basic tree algorithm (*tree*) and describe an efficient implementation of the prefetch tree.

We assume the following experimental conditions for the results presented in this section. The simulations use a T_{cpu} of 150 milliseconds. (We discuss the T_{cpu} parameter in more detail in Section 9.1.1.) We limit the root node to 8192 children and the remaining nodes of the tree to 64 children. For the results in this section, we simulated a portion of the `cello` trace consisting of 3.5 million disk accesses, a portion of the `snake` trace containing 3.8 million disk accesses, and a 190,000 disk accesses long portion of the `hplajw` trace.

Figure 6 compares combined prefetch cache and demand cache miss rates for the three traces with and without prefetching. The horizontal axis of the graph varies cache size from 2 megabytes to 64 megabytes and uses a log scale. The figure shows that our prefetching scheme is quite effective for smaller cache sizes, lowering overall miss rates by up to 32%. However, the advantage of prefetching declines as cache sizes increase. In larger caches, a bigger fraction of the working set stays resident in the cache. 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; in Section 9.2, we show that combining the *tree* algorithm with one block lookahead further reduces the miss rate.

Figure 7 supports the assertion that the working sets of the `cello`, `snake` and `hplajw` traces fit in cache sizes of 32 Mbytes 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 90% of the blocks identified as prefetch candidates already reside in the cache.

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 three traces. For small cache sizes, the frequency of prefetching ranges from one to three blocks prefetched per I/O access. Thus, prefetching accounts for an increase of up to 300% in disk traffic for small cache sizes. At larger cache sizes, the *tree* algorithm prefetches one block approximately every three access periods.

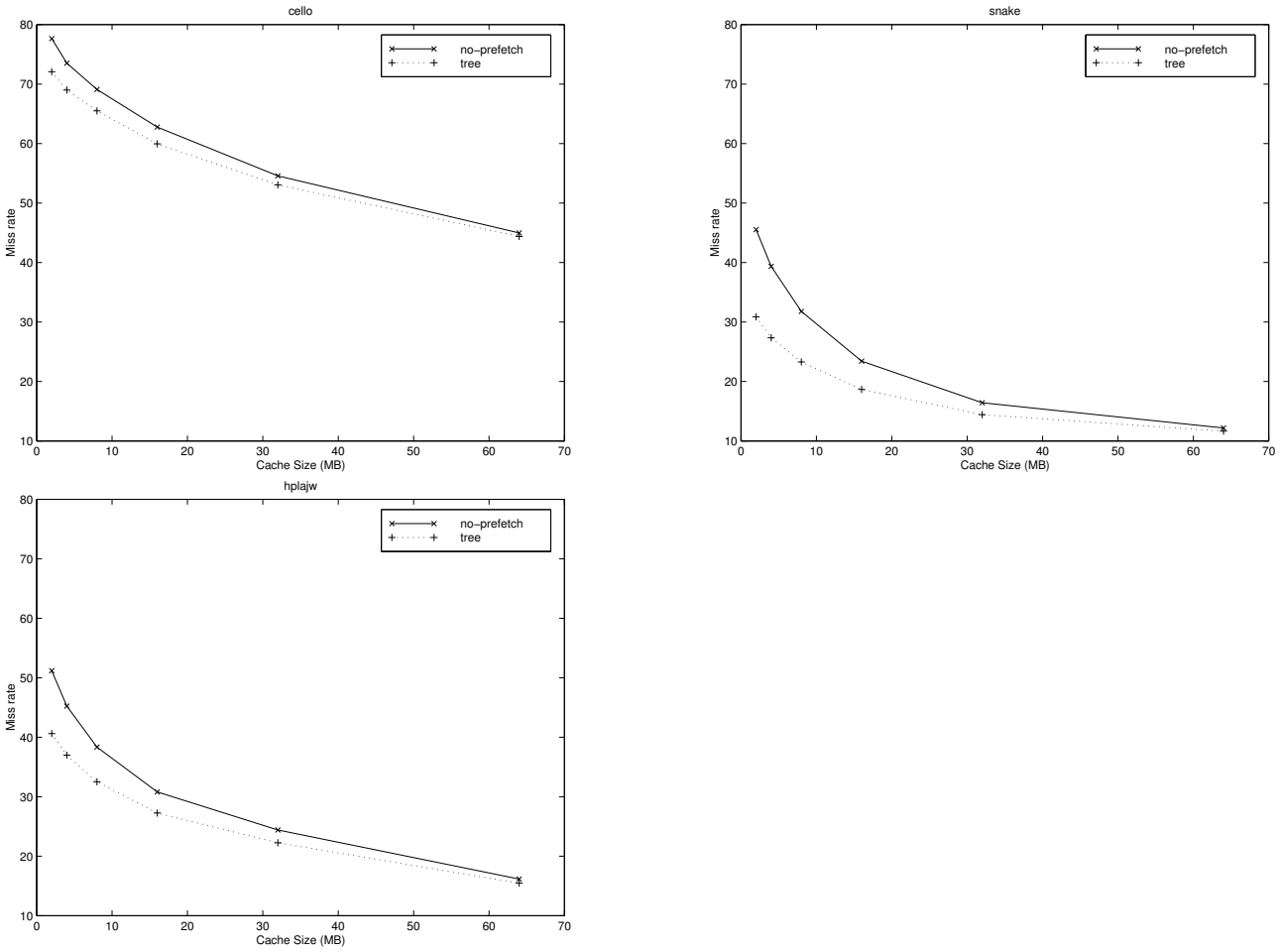


Figure 6: *For the three traces, shows the miss rate with and without the predictive prefetching scheme.*

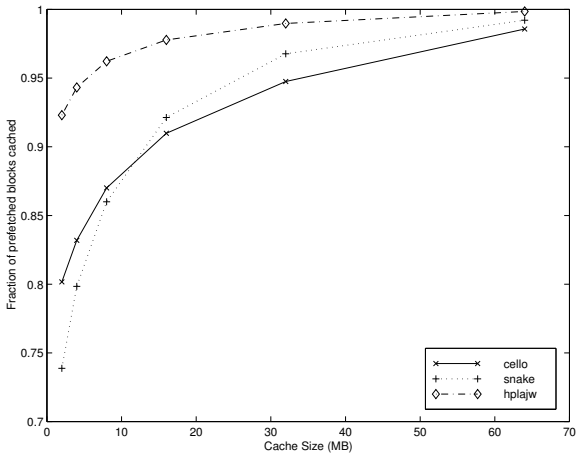


Figure 7: *The fraction of prefetch candidates chosen by the cost/benefit algorithm that already reside in the cache, as the size of the combined demand and prefetch cache increases.*

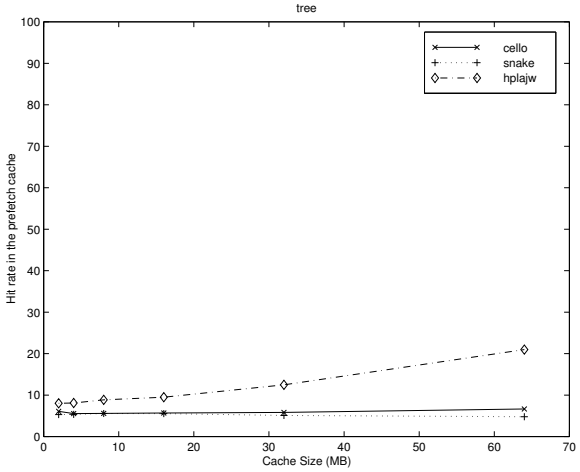


Figure 9: *The hit ratio of the prefetch cache for three traces, as the size of the combined demand and prefetch cache increases.*

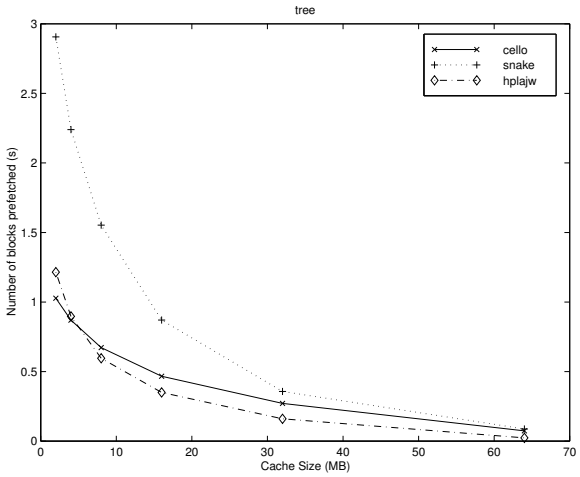


Figure 8: *The number of blocks prefetched by our prefetching scheme per access period for three traces, as the size of the combined demand and prefetch cache increases. These prefetched blocks contribute to an increase in the amount of disk traffic.*

metric	*cello*		*snake*		*hplajw*	
	cost		*cost*		*cost*	
	zero cost	locality cost	zero cost	locality cost	zero cost	locality cost
Miss rate	66.26%	65.50%	24.77%	23.28%	33.51%	32.52%
Prefetch cache hit rate	3.15%	5.57%	3.71%	5.56%	6.56%	8.84%
s	0.794	0.672	1.794	1.552	0.621	0.595

Table 1: *Effect of varying the cost of mispredicted blocks. Miss rate is combined demand and prefetch cache miss rate. Prefetch cache hit rate is the hit rate for blocks in the prefetch cache. s is average number of blocks prefetched per access period.*

Finally, Figure 9 shows that the prefetch cache hit rate is low for all the traces: around 7% for the `cello` and the `snake` traces, and up to 21% for the `hplajw` trace. 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.

In summary, the *tree* prefetching algorithm reduces the combined demand and prefetch cache miss rates by up to 32% compared to a scheme that performs no prefetching. The *tree* prefetching scheme performs best for file buffer cache sizes that are small relative to the size of the working set of blocks being accessed. For small caches, the *tree* scheme prefetches up to three blocks for every I/O access. At larger cache sizes, the amount of prefetching is reduced because most prefetch candidates are already resident in the cache.

9.1.1 Varying T_{cpu}

Figure 10 shows the impact of changing the value of T_{cpu}, the time that the processor spends computing between successive I/O accesses. The graph shows s, the average amount of prefetching performed per access period. This simulation used a cache size of 8 megabytes and varied T_{cpu} from 20 to 640 milliseconds. As 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 11 which shows that the hit rate for blocks in the prefetch cache initially decreases substantially with increasing T_{cpu} and remains under 10% for values of T_{cpu} over 100 milliseconds. For all the traces, the miss rate for the combined demand and prefetch caches remains constant with increasing T_{cpu} as shown in Figure 12.

Since performance is relatively insensitive to values of T_{cpu} above 100 milliseconds, we use a value of $T_{cpu} = 150.0$ milliseconds in our simulations. This value comes from the `cello` trace, which has an I/O rate of about 6 I/Os per second, suggesting CPU activity of approximately 150 milliseconds between successive I/O operations. In a file level trace, T_{cpu} might be smaller; our disk level trace does not include I/O operations that resulted in hits in the original system's buffer cache.

9.1.2 Ejecting Mispredicted Blocks

In the discussion of our implementation, we described two options for assigning cost for ejecting a prefetched block that is not accessed at the predicted time. Either we choose to eject the block immediately and assign it a cost of zero, or we assign a cost of replacement that decreases with the "tardiness" of the block. Table 1 shows the results of these two cost assignments. The tardiness scheme is labeled *locality cost*. While the average number of prefetches per access period, s, and the overall cache miss rate are approximately constant for the two schemes, the prefetch cache hit rate improves approximately 50% when we use the tardiness scheme. Using tardiness to assign costs keeps blocks in the prefetch cache longer, rather than replacing them as soon as a misprediction is detected. Because there is likely to be locality of reference in the prefetch tree, keeping blocks in the cache longer increases the prefetch cache hit rate substantially. (The simulation results presented in earlier sections used tardiness to assign cost to mispredicted blocks.)

depth	*cello*			*snake*			*hplajw*		
	children			*children*			*children*		
	128	8192	∞	128	8192	∞	128	8192	∞
1	128	8192	103143	128	8192	141642	128	8192	10890
2	27	11437	151617	268	2888	67315	59	7120	7928
3	6	4381	34003	94	1039	25048	17	1513	1706
4	4	1725	10916	35	424	9532	7	495	581
5	0	1060	5712	12	160	4521	5	242	288
6	0	873	3342	0	61	2442	0	128	146
7	0	476	1391	0	36	1462	0	65	70
8	0	291	876	0	18	919	0	42	43
9	0	175	519	0	8	523	0	11	11
10	0	147	469	0	6	366	0	4	4

Table 2: *Shape of the prefetch tree*

9.1.3 Limiting the Size of the Prefetch Tree

In this section, we show the shape of the prefetch tree and measure the performance of the prefetching algorithm. To reduce the overhead required to insert nodes into the tree, we experiment with limiting the number of children any node can have, including the root node. The simulations presented in this section use a cache size of 8 megabytes and use $T_{cpu} = 150$ milliseconds. For the results in this and the following section, we use only a single day's disk accesses from each trace, rather than the entire trace, to construct the prefetch tree. This corresponds to 640,000 accesses, 460,000 accesses, and 37,000 accesses for the `cello`, `snake` and `hplajw` traces, respectively.

Table 2 shows the number of nodes at each level of the tree below the root node. The first two columns represent trees in which each node is limited to 128 or 8192 children, respectively; beyond these limits, adding a new child ejects the least-recently-used child of a node to make room for the new child. The goal of eliminating the LRU child is to eliminate older access history in the tree, so that prefetch predictions are based on more recent accesses. The third column in the table shows a prefetch tree with no limit on the number of children a node can have. For each trace, the root node in an unrestricted tree has many children; nodes at lower levels have few children, on average. For example, for the `cello` trace, the root node has 103,143 children; each of these nodes has, on average, 1.5 children. The trees restricted to 8192 children per node have shapes proportional to those of the unrestricted trees; the trees restricted to 128 children per node have few children below the fourth level in each tree. In the experimental results in the previous section, we restricted the root node to 8192 children and lower levels of the tree to 64 children per node. This limits the overhead of the prefetching algorithm while still maintaining a prefetch tree shape proportional to the unrestricted tree.

Table 3 shows several performance metrics for execution of our prefetching scheme with the restrictions just discussed. These simulations were run using a cache size of 8 megabytes and a T_{cpu} value of 150.0 milliseconds. Simulation time refers to the execution time of the simulator. The miss rate is the combined miss rate of the prefetch and demand fetch caches. For the trees restricted to 8192 children per node, the miss rate with prefetching is 3% to 5% lower than the miss rate without prefetching. In the unrestricted tree, the miss rate with prefetching is 3% to 8% lower than the miss rate without prefetching. The parameter s is the average number of prefetches executed per access period. The restricted or *pruned* trees prefetch fewer blocks than the unrestricted trees. Thus, the pruned trees achieve combined miss rates close to those of the unrestricted trees while generating far fewer prefetch accesses.

9.1.4 Profiling Algorithm Execution

Table 4 profiles the execution time of the simulator for trees restricted to 128 and 8192 children per node. Cache lookup, to determine whether a desired block resides in the cache, takes a substantial amount of time for all the traces and for both tree sizes. In trees restricted to 8192 children per node, the simulator spends about 20% of the time sorting the children of the tree in order of decreasing probability, maintaining least-recently-used ordering among the children, and in determining the cost and benefit of blocks. Other operations such as

metric	*cello*				*snake*				*hplajw*			
	children			*without*	*children*			*without*	*children*			*without*
	128	8192	∞	*prefetch*	128	8192	∞	*prefetch*	128	8192	∞	*prefetch*
simulation time (sec)	777.71	422.02	843	89.36	588.66	333.96	475.23	65.49	85.58	15.39	31.05	9.51
miss rate	62.31	59.05	54.31	62.36	55.79	50.61	49.51	55.77	58.23	55.01	55.01	58.21
s	0	1.069	2.37	0	0	1.109	1.45	0	0	0.309	0.31	0

Table 3: *Performance of the prefetching scheme with and without pruning. The simulation time refers to the running time of the simulation. The miss rate is the combined demand cache and prefetch cache miss rate. s is the average number of blocks prefetched per access period.*

primitive	*cello*		*snake*		*hplajw*	
	children		*children*		*children*	
	128	8192	128	8192	128	8192
Cache lookup	76%	60.9%	75.1%	63.1%	76.2%	64.7%
Sorting children in prefetch tree	0%	9.9%	0%	3.7%	0%	9.6%
Tree insertion	0.1%	0.4%	0.1%	0.2%	0.1%	0.4%
Hash table	0%	0.4%	0%	0.3%	0%	0.2%
Finding benefit	3.1%	1.1%	3.2%	1.3%	3.2%	0.8%
Finding cost	1.1%	10.1%	1.1%	12.5%	1.1%	3.4%

Table 4: *Profiling information for the prefetching scheme*

inserting a new node into a tree and using the hash table to look up a node in the first level of the tree take relatively little time.

9.2 Comparison with other Prefetching Schemes

In this section we present results comparing *tree* with other prefetching schemes. Recall that *tree* prefetches predicted blocks based on our cost benefit analysis, while *tree-next-limit* always prefetches the next block and also uses our cost benefit analysis to prefetch predicted blocks. Figure 13 compares the miss rates for *tree* and *tree-next-limit* for the three traces used in this study. For all the traces, *tree-next-limit* performs better than the basic *tree* algorithm. *tree-next-limit* reduces the overall miss rate by up to 30% compared to *tree*. By always prefetching the next disk block after an I/O access, the *tree-next-limit* scheme reduces compulsory misses in the cache. Since its performance is always superior to the *tree* algorithm, for the reminder of this paper we focus on the *tree-next-limit* scheme.

In Figure 14, we compare the *tree-next-limit* algorithm to two schemes: a system that performs no prefetching and the *next-limit* scheme. Recall that *next-limit* always prefetches the next disk block after an I/O access and places a limit on the number of prefetched blocks in the cache. Thus, *next-limit* and *tree-next-limit* differ only in the use of our cost benefit scheme to prefetch additional predicted blocks in the *tree-next-limit* scheme.

The two prefetching schemes, *next-limit* and *tree-next-limit*, perform significantly better than the scheme that does no prefetching for all cache sizes. This is largely due to the reduction in the number of compulsory cache misses that occurs when the next disk block is prefetched after an I/O access. *next-limit* and *tree-next-limit* reduce miss rates by up to 34% and 52% respectively for small cache sizes when compared to a scheme that does no prefetching. For large caches, *next-limit* and *tree-next-limit* reduce miss rates by up to 47% and 55% when compared to *no-prefetch*.

Figure 14 also shows that the *tree-next-limit* scheme performs better than the *next-limit* scheme, particularly for small cache sizes. *tree-next-limit* is effective at predicting future accesses. The algorithm reduces miss rates for small caches by up to 27% compared to the *next-limit* algorithm. At larger cache sizes, the performance of the two algorithms is similar.

9.3 Limitations of Experiments

Our current experimental results have several limitations. Most importantly, we used disk-level traces rather than file-level traces. The disk-level traces do not record accesses that were hits in the original system's file buffer cache. We need to use file-level traces to have a more accurate picture of the effectiveness of our prefetching scheme. Using disk-level traces, we have no way to estimate overall execution time for trace accesses. Such execution time estimates are necessary to determine the improvement in application workload performance with prefetching. We recently obtained file level traces from the University of Kentucky and will use these traces to evaluate our prefetching scheme and for comparison with informed prefetching. Finally, our current simulations make various simplifying assumptions, including disregarding the amount of memory consumed by the prefetch tree and ignoring issues of disk congestion. We plan to account for these parameters in future simulations.

10 Related Work

Our cost-benefit analysis is most closely related to Patterson's informed prefetching [13]. 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. Extensions to the original informed prefetching scheme include an implementation that uses disk striping [12], an implementation for a network file system [15], and a variation that prefetches more deeply during disk idle periods in anticipation of bursts of I/O activity [17].

Mowry *et al.* [10] describe a scheme in which the compiler automatically inserts hints into the program. The compiler analyzes the code and predicts page faults and when data are no longer needed. The operating system uses this information to manage I/O using prefetch and release operations. A run-time layer minimizes prefetching overhead by keeping track of in-core application pages. Their work focuses on scientific applications.

The prefetch tree we use to predict future accesses uses an algorithm developed at Duke University [18] that is adapted from a data compression technique. The scheme breaks a stream of disk accesses into substrings and creates a prefetch tree, where nodes correspond to accessed disk blocks. The probability of accessing a node in the future relates to the number of times the node was visited in the past. Curewitz *et al.* [4] describe a memory efficient implementation of the prefetch tree. 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 [7], using per-file hidden Markov models to predict future accesses [9], using associative memory and pattern recognition to identify access patterns [11], and prefetching whole files using a tree that records past file accesses [5, 8].

In additional prefetching work, Cao *et al.* [1] propose rules that every optimal prefetching and caching strategy must follow. The group proposes a modification to an LRU cache replacement policy to support application-controlled cache replacement [2, 3]. Tracy Kimbrel *et al.* [6] propose a prefetching scheme called *Forestall* that prefetches blocks based on future disk load; disk idle periods are used to prefetch blocks that will be needed during periods of disk congestion.

11 Conclusions

We have 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 proposed by Duke University [18, 4]. We then decide whether to prefetch a candidate block using a cost-benefit analysis. This analysis modifies the informed prefetching scheme [13, 14] to account for probabilistic hints and to allow prefetching along multiple paths in the prefetch tree. We implemented our algorithm efficiently using a trace-driven simulator and evaluated its performance.

We demonstrated that this prefetching scheme improves overall cache miss rates by up to 32% over a scheme that performs no prefetching for the three traces we studied. Coupled with the one block lookahead prefetching, this scheme improves overall cache miss rates by up to 52% over a scheme that performs no prefetching and by up to 27% over an aggressive one block lookahead prefetching scheme for small cache sizes. Prefetching works best for cache sizes that are relatively small compared to the working set of data blocks being accessed. As the demand cache grows to accommodate a larger percentage of the file blocks being accessed, the frequency

of prefetching drops. We showed that prefetching increases overall disk traffic by up to 300% for smaller cache sizes and less than 40% for larger cache sizes.

We gained several insights by constructing the prefetch probability trees. The trees we constructed were very wide at the first level, with thousands of children for the root node; nodes below the root node have relatively few children. Because the prefetch tree contains many nodes, most of which have low probabilities of access, we experimented with limiting the size of the prefetch tree by limiting the number of children of each node. We found that if we allowed 8192 children for the root node and a smaller number (64) on lower levels of the tree, we could maintain the shape of the unrestricted probability tree, achieve similar overall miss rates, and significantly lower the number of prefetch accesses and the overhead of running the algorithm. When pruning children of the tree, we discarded least-recently-accessed blocks; this allowed us to make prefetch predictions based on more recent access patterns.

We will extend this work in several ways. In the short term, we will use object reference traces from Curewitz *et al.* [4] to further evaluate our *tree-next-limit* prefetching scheme. Our long term plans include using file level traces recently obtained from the University of Kentucky. In addition, we will compare our scheme with informed prefetching [13] and experiment with other schemes for choosing candidate blocks for prefetching. Finally, we plan to implement our prefetching algorithm in an operating system and evaluate its performance on a variety of workloads.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

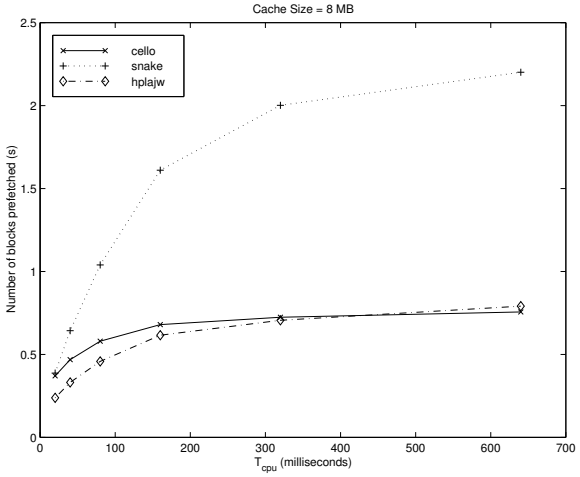


Figure 10: *Shows the amount of prefetching performed per access period as the amount of computation between I/O accesses, T_{cpu}, increases.*

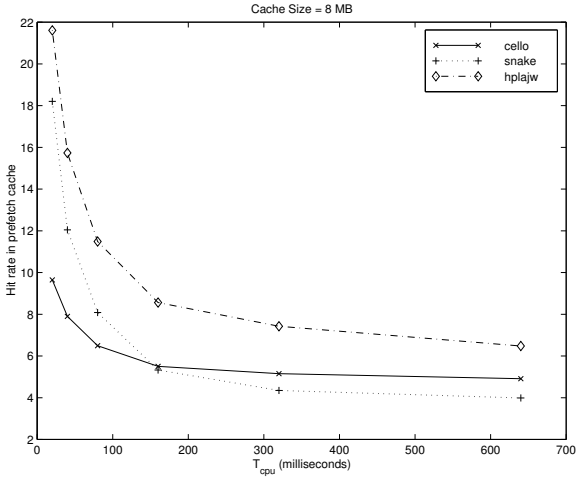


Figure 11: *Shows that the hit rate in the prefetch cache for the three traces decreases as the amount of computation between I/O accesses, T_{cpu}, increases.*

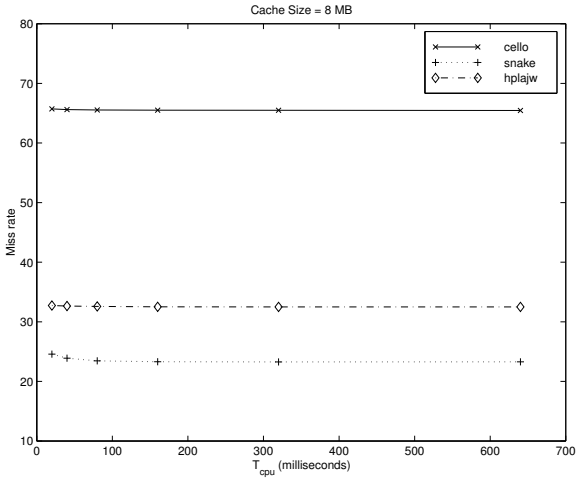


Figure 12: *Shows the combined demand and prefetch cache miss rate for the three traces as the amount of computation between I/O accesses, T_{cpu}, increases. Shows that T_{cpu} has little effect on the miss rate.*

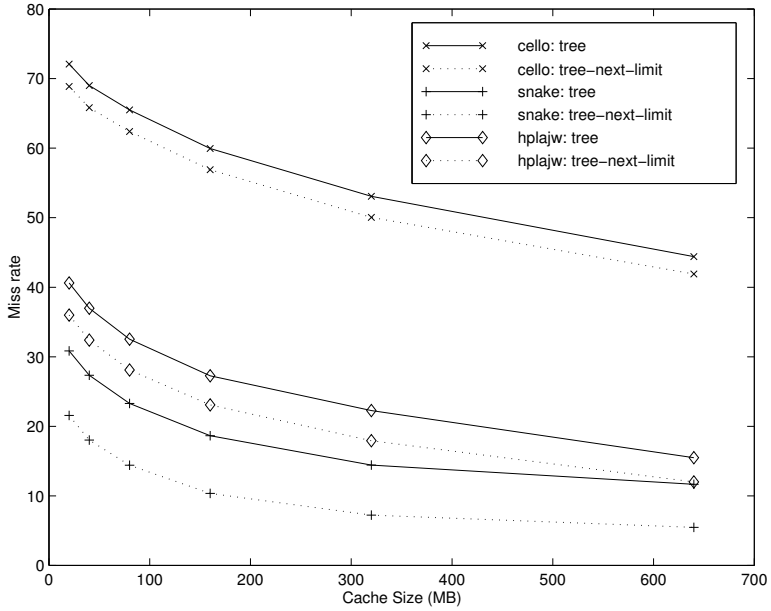


Figure 13: *For the three traces, compares the performance of tree and tree-next-limit. The miss rate for tree-next-limit is always lower than the miss rate for tree.*

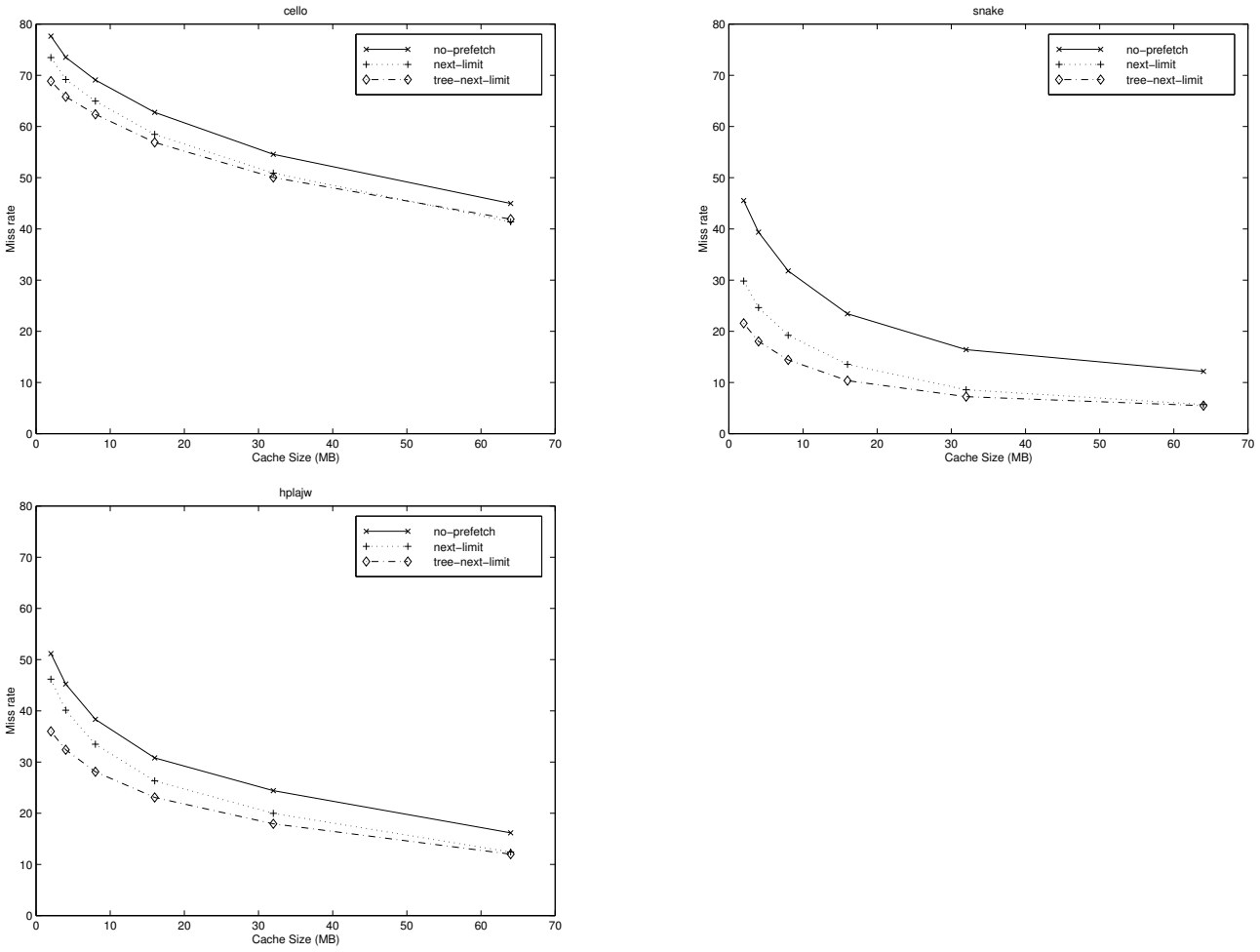


Figure 14: *For the three traces, compares the performance of no-prefetch, next-limit and tree-next-limit. With one exception, tree-next-limit has the lowest miss rate for all traces and cache sizes.*