Appears in *Proceedings of the International Conference on Data Engineering*, New Orleans, LA, Feb 1996

Prefetching from a Broadcast Disk

Swarup Acharya
Brown University
sa@cs.brown.edu

Michael Franklin
University of Maryland
franklin@cs.umd.edu

Stanley Zdonik
Brown University
sbz@cs.brown.edu

Abstract

Broadcast Disks have been proposed as a means to efficiently deliver data to clients in "asymmetric" environments where the available bandwidth from the server to the clients greatly exceeds the bandwidth in the opposite direction. A previous study investigated the use of cost-based caching to improve performance when clients access the broadcast in a demand-driven manner [AAF95]. Such demand-driven access however, does not fully exploit the dissemination-based nature of the broadcast, which is particularly conducive to client *prefetching*. With a Broadcast Disk, pages continually flow past the clients so that, in contrast to traditional environments, prefetching can be performed without placing additional load on shared resources. We argue for the use of a simple prefetch heuristic called $\mathcal{PT}$ and show that $\mathcal{PT}$ balances the cache residency time of a data item with its bandwidth allocation. Because of this tradeoff, $\mathcal{PT}$ is very tolerant of variations in the broadcast program. We describe an implementable approximation for $\mathcal{PT}$ and examine its sensitivity to access probability estimation errors. The results show that the technique is effective even when the probability estimation is substantially different from the actual values.

1 Introduction

1.1 Broadcast Disks

Many new network-based applications must address the problem of communications *asymmetry*, in which there is significantly more bandwidth from the server to the client than in the opposite direction. In the presence of such asymmetry, applications must conserve the use of the limited back-channel capability. Communications asymmetry can arise as a property of the hardware, as in cable television networks, or as a property of the application environment, as in Advanced Traffic Information Systems (ATIS)[ShL94], in which hundreds of thousands of motorists may simultaneously require data from the server. In some extreme cases, such as disconnection in mobile computing environments, there may be no upstream (from clients to servers) communication capacity at all.

Broadcast Disks ([AAF95], [AFZ95]) was introduced as a technique for delivering data to clients in asymmetric environments. It is a *push-based* technique— data transfer from the server to clients is initiated by the server, rather than by explicit client requests (as in a traditional *pull-based* system). Data items (i.e., pages) are broadcast to the client population optimistically, in anticipation of the need for those items at the clients. The broadcast is repetitive — simulating a rotating storage medium or a disk. In this scheme, groups of pages with different broadcast frequencies are multiplexed on the same channel to create the illusion of multiple disks each spinning at a different speed. The fastest disk is closest to the client in the sense that the expected wait time for a particular page is lowest, while the pages on the slowest disk are farthest since they will, in general, take a longer time to access. A designer has the flexibility to choose the number of disks, the size of each disk, the relative spinning speed of each disk, and the placement of pages across disks. Thus, a designer can create an arbitrarily fine-grained memory hierarchy that is tailored to meet the needs of any particular client population.

In [AAF95], we demonstrate how the multi-disk scheme can produce performance improvements over a flat (one-disk) broadcast when the client access pattern is skewed. We also show that performance can be improved further for the multi-disk case if the client uses a cost-based caching policy that is sensitive to the frequency of broadcast as well as the access probability. We propose such a policy (called $\mathcal{PIX}$) that chooses as a victim the cached page with lowest value for the quantity p/x where p is the page's access probability and x is its frequency of broadcast . We also present an algorithm, $\mathcal{LIX}$, that approximates $\mathcal{PIX}$ and is easily implementable.

In this paper, we explore the use of prefetching to further improve client performance in the broadcast environment. The Broadcast Disks technique is particularly conducive to prefetching because pages are continuously presented to clients via the broadcast, allowing the client to bring some of these pages into its cache in anticipation of future accesses. The dissemination-oriented nature of the environment changes the tradeoffs commonly associated with prefetching in more traditional environments, such as database and file systems. In particular, prefetching in a traditional system places additional load on shared resources (disks, network etc.), which can be potential bottlenecks. In contrast, prefetching from a Broadcast Disk only impacts the local client resources. Thus, the risks in prefetching are significantly lower than in a more traditional system.

1.2 Prefetching - A Motivating Example

For Broadcast Disks (as in most systems), the goal of prefetching is to improve the response time for the clients. Response time improvements are typically achieved in two ways: 1) by improving the client cache hit rate, and 2) by reducing the cost of a cache miss. The $\mathcal{PIX}$ algorithm and its approximation ($\mathcal{LIX}$), have been shown to achieve an effective balance between these two issues. They both accept a slightly lower client hit rate in order to achieve a reduction in the number of pages that must be obtained from the slower disks in a multi-disk broadcast.

$\mathcal{PIX}$ and $\mathcal{LIX}$ are strictly demand-driven strategies in which pages are brought from the broadcast into a client cache only as the result of a request for that page. The introduction of prefetching provides an additional way for pages to be brought into the cache, the potential benefits of which are illustrated by the following simple example (Figure 1). Consider a client that is interested in accessing two pages (x and y) with equal probability (i.e., $p_x = p_y = 0.5$), and has only a single cache slot available. In this example (as shown in the figure), the server places these two pages 180 degrees apart on the broadcast (for simplicity, a single, flat disk is used).

Under a demand-driven strategy, the client would cache one of

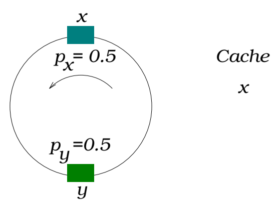

Figure 1: Tag-team Caching

the pages, say x, as the result of a request for that page. Subsequent accesses to x can be satisfied locally from the cache, and thus have no delay. If however, the client needs page y, it waits for y to come by on the broadcast, and then replaces x with y in the cache. Page y then remains cache-resident until x is brought in again. With this strategy, the expected delay on a cache miss is one half of a disk rotation (because the request can be made at a random time). The expected cost C_i of accessing a page i is given by the formula:

$$C_i = p_i * m_i * d_i$$

where p_i is the access probability, m_i is the expected probability of a cache miss and d_i is the expected broadcast delay for page i. The expected total cost of access over all pages for this *demand-driven* strategy is therefore:

$$\sum_{i \in \{x,y\}} C_i = C_x + C_y = 0.5 * 0.5 * 0.5 + 0.5 * 0.5 * 0.5 = 0.25$$

i.e., one quarter of a disk rotation.

In contrast, consider a simple prefetching strategy that fetches page x into the cache when it arrives on the broadcast and holds it until page y is broadcast. At that point, the client caches page y and drops x. When x is broadcast again, y is dropped and x is cached. We call this strategy *tag-team caching* because pages x and y continually replace each other in the cache. The expected cost of the tag-team strategy for this example is:

$$\sum_{i \in \{x,y\}} C_i = C_x + C_y = 0.5 * 0.5 * 0.25 + 0.5 * 0.5 * 0.25 = 0.125$$

Thus, the average tag-team cost is one eighth of a disk rotation — one *half* the expected cost of the demand-driven strategy.

It is important to note that the performance gain of tag-team comes not from an improved hit rate (both strategies have a hit rate of 50%), but rather because the cost of a miss using tag-team is half of the cost of a miss under the demand-driven strategy. With the demand-driven strategy, a miss can occur at any point in the broadcast. In contrast, using tag-team, misses can only occur during half of the broadcast. For example, a miss on a reference to page x can only occur during the part of the broadcast between the arrival (and prefetching) of page y and the subsequent arrival of page x. Tag-team reduces the cost of cache misses by ensuring that a page is cached for the portion of the broadcast when the wait for that page would be the longest.

The simple tag-team example can be generalized to multiple clients and multiple pages over different sized caches. The details of this formulation are beyond the scope of this paper. In this paper, we focus on a much simpler prefetching technique called $\mathcal{PT}$. $\mathcal{PT}$ is a dynamic algorithm that evaluates the worth of each page on the broadcast to determine if it is more valuable than some other page that is currently in the cache. If so, it swaps the cache-resident page with the broadcast page.

1.3 Overview of the Paper

The remainder of the paper describes and experimentally validates our approach. Section 2 presents a description of the simulation model. Section 3 describes the basic approach to prefetching and explains some of the intuitions that will be useful in understanding the experiments. In Section 4, we present a series of experimental results that examine the performance of the $\mathcal{PT}$ prefetching technique. Section 5 describes an implementable approximation of $\mathcal{PT}$, and compares its performance and execution cost with a non-prefetching cache management algorithm. Section 6 describes related work. Section 7 summarizes our results and describes future directions.

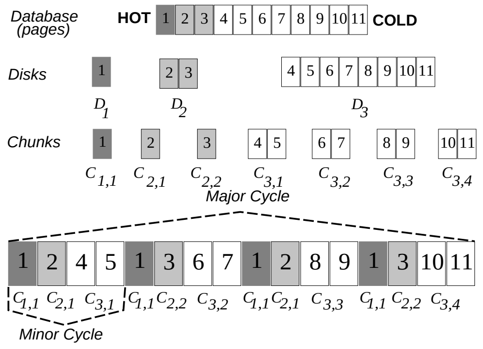
Database (pages)
HOT
1 2 3 4 5 6 7 8 9 10 11
COLD
Disks
1
2 3
4 5 6 7 8 9 10 11
D_1
D_2
D_3
Chunks
1
2
3
4 5
6 7
8 9
10 11
$C_{1,1}$ $C_{2,1}$ $C_{2,2}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$ $C_{3,4}$
Major Cycle
1 2 4 5 1 3 6 7 1 2 8 9 1 3 10 11
$C_{1,1}$ $C_{2,1}$ $C_{3,1}$ $C_{1,1}$ $C_{2,2}$ $C_{3,2}$ $C_{1,1}$ $C_{2,1}$ $C_{3,3}$ $C_{1,1}$ $C_{2,2}$ $C_{3,4}$
Minor Cycle

Figure 2: Deriving a Server Broadcast Program

2 Modelling The Broadcast Environment

Our model of the broadcast environment has been described previously ([AAF95]). The results presented in this paper are based on the same underlying model, extended to include prefetching. In this section we briefly describe the model, focusing on the modifications that were required to introduce prefetching.

As in [AAF95], we model a broadcast environment that is restricted in several ways:

- The client population and their access patterns do not change. This implies that the content and the organization of the broadcast program remains static.
- Data is read-only; there are no updates either by the clients or at the servers.
- Clients make no use of upstream communications, i.e., they provide no feedback to the server.

In the broadcast environment, the performance of a single client for a given broadcast is independent of the presence of other clients. As a result, we can study the environment by modeling only a single client. The presence of multiple clients, however, can potentially cause the server to generate a broadcast program that is sub-optimal for any particular client (since it is derived taking into account the needs of all clients). To account for this phenomenon, we model the client as accessing *logical* pages that are mapped to the *physical* pages that are broadcast by the server. By controlling the nature of the mapping, we vary how close the broadcast program of the server matches the client's requirements. For example, having the client access only a subset of the pages models the fact that the server is broadcasting pages for other clients as well.

2.1 The Server Model

A broadcast disk is a cyclic broadcast of database pages that are likely to be of interest to the client population. Multiple disks can be superimposed on a single broadcast channel by broadcasting some pages more frequently than others. Each disk corresponds to those pages which have the same broadcast frequency. The desirable characteristics of a broadcast program have been outlined in [AAF95]. Briefly, a good broadcast program is periodic, has fixed (or nearly fixed) inter-arrival times for repeated occurrences of a page, and

ServerDBSize	No. of distinct pages in broadcast
NumDisks	No. of disks
DiskSize$_i$	Size of disk i (in pages)
Δ	Broadcast shape parameter
Offset	Offset from default client access
Noise	% workload deviation
Scatter	Intra-disk spread

Table 1: Server Parameter Description

allocates bandwidth to pages in accordance with their access probabilities.

2.1.1 Broadcast Program Generation

The algorithm used by the server to generate the broadcast program requires the following inputs: the number of disks, the relative spin speeds of each disk and assignments of pages to disks on which they are broadcast. In this paper we explain the broadcast generation process using a simple example. For a detailed description of the algorithm, the reader is referred to [AAF95].

Figure 2 shows 11 pages that are divided into three ranges of similar access probabilities. Each of these ranges will be a separate "disk" in the broadcast. In the example, pages of the first disk are to be broadcast twice as often as those in the second and four times as often as those of the slowest disk. To achieve these relative frequencies, the disks are split into smaller equal-sized units called *chunks* (C_{ij} refers to the j^{th} chunk of disk i); the number of chunks per disk is inversely proportional to the relative frequencies of the disks. In other words, the slowest disk has the most chunks while the fastest disk has the fewest chunks. In the example, the number of chunks are 1, 2 and 4 for disks D_1, D_2 and D_3 respectively. The program is generated by broadcasting a chunk from each disk and cycling through all the chunks sequentially over all the disks. The figure shows a major cycle which is one complete cycle of the broadcast and a minor cycle which is a sub-cycle consisting of one chunk from each disk. As desired, page 1 appears four times per major cycle, pages 2 and 3 appear twice and so on.

2.1.2 Server Execution Model

The parameters that describe the operation of the server are shown in Table 1. The server broadcasts pages in the range of 1 to *ServerDBSize*, where *ServerDBSize* $\geq$ *AccessRange*. These pages are interleaved into a broadcast program as shown in the previous section. This program is broadcast repeatedly by the server. The structure of the broadcast program is described by several parameters. *NumDisks* is the number of levels (i.e., "disks") in the multi-disk program. By convention disks are numbered from 1 (fastest) to N=*NumDisks* (slowest). *DiskSize$_i$*, $i \in$ [1..N], is the number of pages assigned to each disk i. Each page is broadcast on exactly one disk, so the sum of *DiskSize$_i$* over all i is equal to the *ServerDBSize*.

To quantify the relative speeds of the disks we introduce a parameter called Δ, which determines the relative frequencies of the disks in a restricted manner. Using Δ, the frequency of broadcast $rel\_freq(i)$ of each disk i, can be computed relative to $rel\_freq(\mathrm{N})$, the broadcast frequency of the slowest disk (disk N) as follows:

$$\frac{rel\_freq(i)}{rel\_freq(\mathrm{N})} = (\mathrm{N} - \mathrm{i})\Delta + 1$$

When Δ is zero, the broadcast is flat: all disks spin at the same speed. As Δ is increased, the speed differentials among the disks increase. For example, for a 3-disk broadcast, when $\Delta = 1$, disk 1 spins three times as fast as disk 3, while disk 2 spins twice as fast as disk 3. When $\Delta = 3$, the relative speeds are 7, 4, and 1 for disks 1, 2, and 3 respectively. It is important to note that Δ is used in the study only to organize the space of disk configurations that we examine and is not a feature of the disk model.

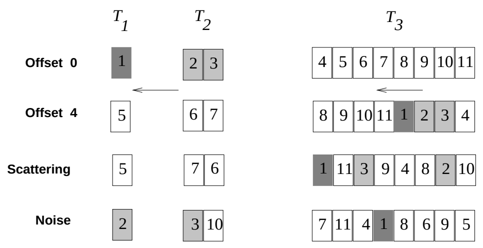

Figure 3: Organizing the Broadcast

The remaining three parameters, *Offset*, *Scatter* and *Noise*, are used to modify the mapping between the *logical* pages requested by the client and the *physical* pages broadcast by the server. When *Offset* and *Noise* are both set to zero, and the disk is *Unscattered*, then the logical to physical mapping is simply the identity function. In this case, the *DiskSize*$_1$ hottest pages from the client's perspective (i.e., 1 to *DiskSize*$_1$) are placed on Disk #1, the next *DiskSize*$_2$ hottest pages are placed on Disk #2, etc. However, this mapping may be sub-optimal due to client caching. Some client cache management policies tend to fix certain pages in the client's buffer which makes broadcasting them frequently a waste of bandwidth. In such cases, the best broadcast can be obtained by shifting the hottest pages from the fastest disk to the slowest. *Offset* is the number of pages that are shifted in this manner. An offset of K shifts the access pattern by K pages, pushing the K hottest pages to the end of the slowest disk and bringing colder pages to the faster disks.

In contrast to *Offset*, which is used to provide a better broadcast for the client, the parameter *Noise* is used to introduce disagreement between the needs of the client and the broadcast program generated by the server. Disagreement can arise in many ways, including dynamic client access patterns and conflicting access requirements among a population of clients. *Noise* determines the percentage of pages for which there may be a mismatch between the client and the server. *Noise* is introduced as follows: for each page in the mapping, a coin weighted by *Noise* is tossed. If, based on the coin toss, a page i is selected, then a disk d is chosen randomly to be its new destination. To make way for i, an existing page j on d is chosen randomly, and i and j swap mappings.[1]

A new parameter that is necessitated by prefetching is called *Scatter*. As the results of Section 4 will show, the performance benefits of prefetching can improve if pages are spread over the broadcast rather than clustered together by access probability. When *Scatter* is turned on, every page is randomly swapped with another page *on its own* disk causing a random shuffle of pages within a disk.

The generation of the server broadcast program works as follows. First, the mapping from logical to physical pages is generated as the identity function. Second, this mapping is shifted by *Offset* pages as described above. Third, if *Scatter* is on, pages in each disk are shuffled among themselves. Finally, the outcome of a coin toss weighted by *Noise* determines if a page is potentially swapped to a new disk. The complete process is shown in Figure 3 for the toy example of Figure 2. The top two rows show the identity function (*Offset = 0*) and the shifting of the four hottest pages to the

[1]Note that a page may be swapped with a page on its own disk. Such a swap does not affect performance in the steady state, so *Noise* represents the upper limit on the number of changes.

CacheSize	Client cache size (in pages)
ThinkTime	Time between client page accesses (in broadcast units)
AccessRange	# of pages in range accessed by client
Zipf Distribution	
θ	Zipf distribution parameter
RegionSize	# of pages per region

Table 2: Client Parameter Description

slowest disk (*Offset = 4*). The third row demonstrates the *Scatter* effect. Finally, the bottom row shows how *Noise* moves pages across different disks.

2.2 The Client Model

The parameters that describe the operation of the client are shown in Table 2. The simulator measures performance in logical time units called *broadcast units*. A broadcast unit is the time required to broadcast a single page. The actual response time will depend on the amount of real time required to transmit a page on the broadcast channel. It is important to note that the relative performance benefits are independent of the bandwidth of the broadcast medium.

The client has a cache that can hold *CacheSize* pages. After every access, the client waits *ThinkTime* broadcast units and then makes the next request. The *ThinkTime* parameter allows the cost of client processing relative to page broadcast time to be adjusted, thus it can be used to model workload processing as well as the relative speeds of the CPU and the broadcast medium.

The client accesses pages from the range 1 to *AccessRange*, which can be a subset of the pages that are broadcast. All pages outside of this range have a zero probability of access at the client. In this study we assumed two models of access distributions:

- *Uniform Distribution*: The client accesses all pages in the *AccessRange* uniformly with the same probability. We use this distribution for pedagogical reasons to show the benefits of prefetching.
- *Zipf Distribution*([Knu81]): The Zipf distribution with a parameter θ is frequently used to model non-uniform access. It produces access patterns that become increasingly skewed as θ increases — the probability of accessing any page numbered i is proportional to $(1/i)^\theta$. Similar to earlier models of skewed access [DDY90], we partitioned the pages into regions of *RegionSize* pages each, such that the probability of accessing any page within a region is uniform; the Zipf distribution is applied to these regions. Regions do not overlap and thus, there are *AccessRange/RegionSize* regions.

When prefetching, the cache manager continually samples the broadcast for potential prefetch candidates. This is in contrast to demand-based caching, in which the broadcast is sampled only in response to a page fault. The same heuristic for determining a page replacement victim is used for both replacements due to demand-driven access (i.e., page faults) and those due to prefetch.

3 A Simple Prefetching Heuristic

When prefetching data from a Broadcast Disk, the importance of an arriving page must be estimated to determine if it should replace a less valuable cache-resident page. In this section we describe a simple heuristic algorithm to do this. The algorithm, called $\mathcal{PT}$, computes the value of a page by taking the product of the probability(P) of access of the page with the time(T) that will elapse before that page appears on the broadcast again. This is called the page's `pt` value. $\mathcal{PT}$ finds the page in the cache with the lowest `pt` value, and replaces it with the currently broadcast page if the latter has a higher `pt` value. It should be noted that `pt` values are dynamic — they change with every "tick" of the broadcast.

Page	Access Probability	Broadcast Freq. (per Period)	Repetition Period	`pix` Value
A	P	2	10 units	P/2
B	P/2	2	10 units	P/4
C	P/2	1	20 units	P/2

Table 3: Access and Broadcast Freq. for $\mathcal{PT}$ example

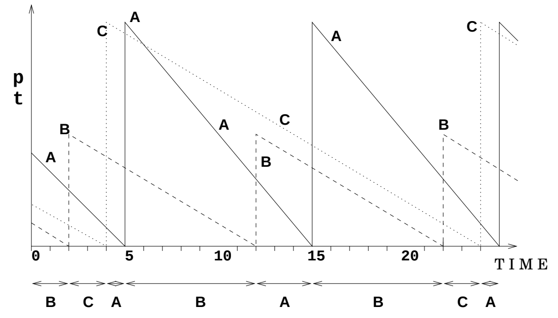

Figure 4: `pt` vs. Time

In our previous study, we describe a non-prefetching algorithm called $\mathcal{PIX}$, which uses the probability (P) of access divided by the broadcast frequency (X) (i.e., the `pix` value) as a replacement metric. This quantity provides a means of including the cost of a cache miss in the determination of which pages to keep cached. In contrast to `pt` values, however, `pix` values remain unchanged during a broadcast cycle. Thus, in the absence of accesses, the contents of the cache using $\mathcal{PIX}$ remain static; the cache will tend to acquire the pages with the highest `pix` values. In this paper we show that a good prefetch metric should be dynamic, i.e., sensitive to the current position of the broadcast.

The `pt` value of a page is dynamic because the time parameter of the metric is constantly changing. When a page is broadcast, `t` is highest since ignoring the page at that point will put the client at a maximal distance from the page. In other words, the moment that a page passes by is the moment at which there will be the longest wait to get it again. From that point on, the time parameter steadily decreases until subsequent re-broadcast of the page, thus creating a sawtooth function of time for $\mathcal{PT}$. Figure 4 traces this saw tooth effect for three pages A, B and C. The access probabilities and broadcast frequencies for the three pages are given in Table 3. Consider time unit 5. At this point, page A is broadcast and its `pt` value shoots up to its peak. A similar effect happens for B at time units 2, 12, 22$\cdots$;, however its maximum `pt` value is only half that of A because of its lower access probability. C, which has the same access probability as B, has the same maximum `pt` value (at times 4, 24, 44$\cdots$) as A because of its longer broadcast period. The line along the bottom of the figure shows the potential replacement victim at each time unit. Note that unlike $\mathcal{PIX}$, in which B is always the least valued page among the three, there are time intervals when $\mathcal{PT}$ would choose C or even A as a replacement victim.

Due to this saw-tooth behavior, $\mathcal{PT}$ will tend to drop high probability items just before they are re-broadcast. Items are most sticky right after they are brought into the cache. $\mathcal{PT}$ is in effect adjusting the allocation of cache slots to pages so that (subject to cache size) they are likely to be in the cache during those portions of the broad-

ThinkTime	2.0
ServerDBSize	3000
AccessRange	1000
CacheSize	1 (0.1%) to 1000(100%)
Δ	1,2,. . .4
θ	0.95
Offset	0, *CacheSize*
Noise	0%, 15%, 30%, 45%, 60%
Scatter	On, Off
RegionSize	50

Table 4: Parameter Settings

cast cycle when they would be most expensive to acquire from the broadcast. Note that this effect of cycling in and out of the cache is suggestive of tag-team caching. In fact, the results of the next section show that $\mathcal{PT}$ is able to achieve the performance of the tag-team example described previously.

It should be noted that any time-based prefetch scheme is potentially very expensive to implement as it could require computing the metric for every item in the cache at every clock tick. While we use pure $\mathcal{PT}$ as an ideal case in this study, we examine an implementable approximation to $\mathcal{PT}$ (called $\mathcal{APT}$), in Section 5.

4 Experiments and Results

In this section, we use the simulation model to explore the benefits of prefetching in a broadcast environment. The primary performance metric used is the average response time (i.e., the expected delay) at the client measured in *broadcast units*. Table 4 shows the parameter settings for these experiments. The server database size (*ServerDBSize*) was 3000 pages and the client access range (*AccessRange*) was 1000 pages. The client cache size was varied from 1 (i.e., no caching) to 1000 (i.e., the entire access range). The results in these experiments were obtained once the client performance reached steady state. The cache warm-up effects were eliminated by beginning our measurements 4000 accesses after the cache was full.

The first four experiments examine the relative performance of $\mathcal{PT}$ and $\mathcal{PIX}$ for combinations of uniform or skewed client access patterns with flat or multi-level disks. Analyzing the performance of $\mathcal{PT}$ under these varying conditions provides insight into how prefetching exploits client cache resources to reduce the expected delay for page requests.

4.1 Expt. 1: Uniform Access, Flat Disk

In the first experiment, we examine the performance of the $\mathcal{PT}$ prefetching heuristic when the client access pattern is uniform (i.e., all pages in the *AccessRange* are accessed with equal probability), and the broadcast is a single, flat disk (i.e., $\Delta = 0$) and the *AccessRange* is *scattered*. Figure 5 shows the expected delay for $\mathcal{PT}$ and $\mathcal{PIX}$ algorithms as the client cache size is varied from a single page to 1000 pages. In this case, $\mathcal{PT}$ and $\mathcal{PIX}$ have identical performance at either end of the spectrum, but $\mathcal{PT}$ has better performance throughout the entire intermediate range. When the cache size is 1 page, the client can perform no caching or prefetching and thus, the expected delay is 1500 pages (this is one half of the time for one disk rotation) for both algorithms. As the cache size increases, the performance improves under both algorithms, until at a cache size of 1000 pages, the response time for both is zero (because all accessed pages are cached at this point).

Between the two endpoints, the prefetching performed by $\mathcal{PT}$ leads to better performance than $\mathcal{PIX}$. Note the case when the cache can hold exactly half of the accessed pages (Cache size = 500). At this point, the response time of $\mathcal{PT}$ is half that of $\mathcal{PIX}$. This performance improvement is exactly what is predicted by the tag-team example described in Section 1.2, and occurs for similar reasons. In this case, $\mathcal{PT}$ and $\mathcal{PIX}$ have the same cache hit rates, but $\mathcal{PT}$ is able to reduce the cost of cache misses by better scheduling the cache residency of pages. To see why $\mathcal{PT}$ achieves the tag-team effect in this experiment, it is necessary to recall how the $\mathcal{PT}$ value of a given page changes over time (as described in Section 3). A page's $\mathcal{PT}$ value is at its maximum immediately after the page has been broadcast; it then linearly decreases until reaching zero at the instant the page is re-broadcast. As a result, using $\mathcal{PT}$, pages are more likely to be replaced from the cache as they get closer to being re-broadcast. In this experiment, which has uniform access and a flat disk, a cache size of half of the *AccessRange* allows $\mathcal{PT}$ to ensure that pages are cache resident for the first (most expensive) half of their inter-arrival gap, so that no cache misses occur during that half of the gap. Thus, as in tag-team caching, the heuristic used by $\mathcal{PT}$ improves performance by reducing the penalty of cache misses.

In relative terms (i.e., % difference) $\mathcal{PT}$'s advantage over $\mathcal{PIX}$ grows with the cache size up to the point at which the entire access range is cached. However, the advantage in absolute terms (i.e., time) reaches its maximum at a cache size of 500, and decreases beyond that point in this experiment. The reason for this behavior is that every cache slot that is added to the cache adds 0.1% (i.e. 1 out of 1000 pages) to the client hit rate for both algorithms. The cost of a miss remains constant for $\mathcal{PIX}$ but decreases with the cache size for $\mathcal{PT}$, because the latter algorithm uses the additional cache slot to also fractionally shorten the delay for all pages. Thus, the marginal benefit of an additional cache slot is constant for $\mathcal{PIX}$ (each slot reduces the expected delay by 1.5 units), as can be seen by the linear decrease in response time for $\mathcal{PIX}$. In contrast, beyond the 500 page point, the marginal benefit of an additional cache slot decreases for $\mathcal{PT}$, so that the absolute benefit of $\mathcal{PT}$ over $\mathcal{PIX}$ decreases.

4.2 Expt. 2: Skewed Access, Flat Disk

The previous experiment examined the case when the client's access pattern is uniform and the disk is flat. In this experiment instead study the case when the client's access pattern is highly skewed (according to the Zipf distribution described in Section 2.2). The results of this experiment can be seen in Figure 6, which shows the response time for $\mathcal{PT}$ and $\mathcal{PIX}$ when used on two different flat disk configurations: *Scattered* and *Unscattered*. The Unscattered disk lays out all the pages in the *AccessRange* sequentially. When the Zipf distribution is used, this results in a layout where the accesses are highly clustered to a small region of the disk. This clustering is avoided in the Scattered disk by randomizing the location of the pages on the flat disk (as described in Section 2.1).

Turning to Figure 6, it can be seen that all four curves have similar performance. As in the previous case, performance improves with additional cache, but is non-linear due to the skewed access pattern. As the cache becomes larger, the marginal performance gain from a new slot decreases here. When the Unscattered disk is used, $\mathcal{PT}$ provides no performance improvement over $\mathcal{PIX}$. With pages clustered together, $\mathcal{PT}$ cannot fully exploit the cache for tag-teaming pages with similar access probability, as the hot (i.e., high access probability) pages that appear earlier in the cluster tend to be pushed out of the cache by the hot pages that are ordered later in the cluster. The expected delay for the earlier pages, thus approaches half a rotation — as would be expected for a non-prefetching scheme. When the Scattered disk is used, $\mathcal{PT}$ has a noticeable performance improvement over $\mathcal{PIX}$ demonstrating the benefits of randomizing

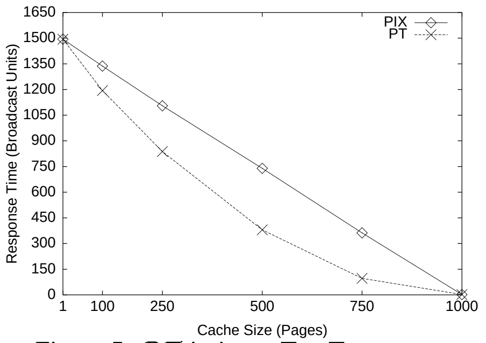

Figure 5: $\mathcal{PT}$ imitates Tag-Team case
Uniform Access, Flat Disk

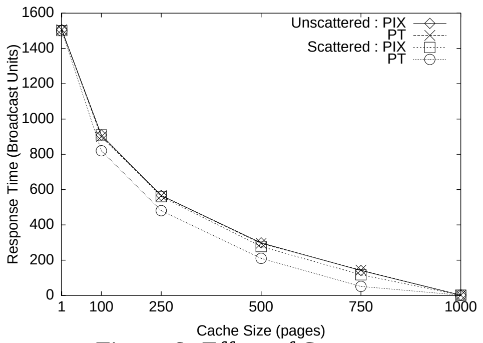

Figure 6: Effect of Scatter
Skewed Access, Flat Disk

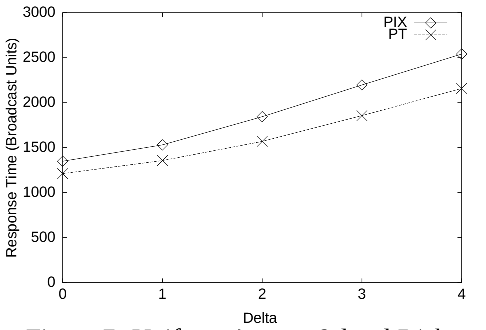

Figure 7: Uniform Access, 3-level Disk
Cache Size = 100, Varying Δ

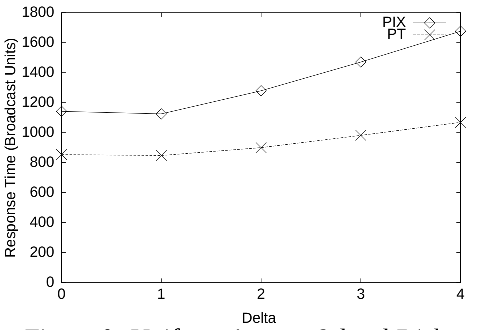

Figure 8: Uniform Access, 3-level Disk
Cache Size = 250, Varying Δ

the (flat) disk contents for $\mathcal{PT}$-based prefetching. However, the performance advantage of $\mathcal{PT}$ over $\mathcal{PIX}$ is somewhat less than the previous experiment, as the skewed access pattern allows $\mathcal{PIX}$ to better exploit the client cache. Yet, for a cache size of 500, $\mathcal{PT}$ on the Scattered disk obtains a 20% improvement over $\mathcal{PIX}$.

4.3 Expt. 3: Uniform Access, Multi-Level Disk

The third experiment examines the behavior of $\mathcal{PT}$ when a multi-disk broadcast is used. For simplicity, a uniform client access pattern is used in this experiment. In this case, the broadcast consists of three disks, for which the fastest disk holds 300 pages, the medium disk holds 1200 pages and the slowest disk holds the remaining 1500 pages. The relative spinning speeds of the disks (and hence the arrival rates of pages on those disks) is varied using the disk skew parameter Δ(Delta). Figures 7 and 8 show the performance of $\mathcal{PT}$ in this case for client cache size is 100 and 250 pages respectively.

In general, the two figures exhibit similar behavior. Due to the uniform client access pattern, increasing disk skew (Δ) harms performance here, as the best broadcast for a uniform access distribution is a flat broadcast[AAF95]. In both graphs, $\mathcal{PT}$ outperforms $\mathcal{PIX}$ and is less sensitive to Δ in the range of 0 (flat disk) to 4 (at which the pages on the fastest disk are broadcast nine times more frequently than pages on the slowest disk). The advantage that $\mathcal{PT}$ has over $\mathcal{PIX}$ is larger (in both absolute and relative terms) with a cache size of 250 pages than for a cache size of 100 pages. This is because $\mathcal{PT}$ can not fully achieve the tag-team effect with a small cache. Likewise, if the cache becomes too large, then the absolute advantage of $\mathcal{PT}$ decreases. In this experiment, at a cache size of 250 pages (Figure 8), $\mathcal{PT}$ provides a substantial improvement over $\mathcal{PIX}$, and this advantage increases as the disk skew (Δ) is increased.

$\mathcal{PT}$ is more tolerant of disk skew than $\mathcal{PIX}$ in this case because it is effectively able to adjust the cache residencies of various pages in order to cope with their differing arrival rates. The manner in which

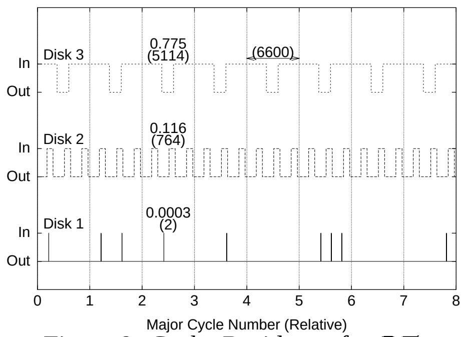

Figure 9: Cache Residency for $\mathcal{PT}$
Cache Size = 500, Δ = 2

$\mathcal{PT}$ accomplishes this can be seen in Figure 9, which shows the cache residency of a representative AccessRange page from each of the three disks of the broadcast. Since all pages in the AccessRange are accessed with equal probability, comparing the cache residency times of the three pages demonstrates the way in which $\mathcal{PT}$ exploits the client cache resources. The figure shown is for the case where the client cache size is 500 and Δ is set to 2. Time increases along the x-axis, which is divided into "major cycles" (i.e., periods of the entire broadcast). The three curves in the figure correspond to the pages from the three disks. When the curve is at the high position (denoted as "In"), the page is resident in the client's cache, and when the curve is in the low position (denoted as "Out"), the page is not in the cache. The figures on the graphs show the fraction of the cycle each page is cache-resident every time it enters the cache. The figure in parenthesis is the absolute time in broadcast units.

The top curve in the figure shows the cache residency of a page that is stored on the slowest disk (Disk #3). Pages on Disk #3 are broadcast once per major cycle, and therefore, $\mathcal{PT}$ removes the page

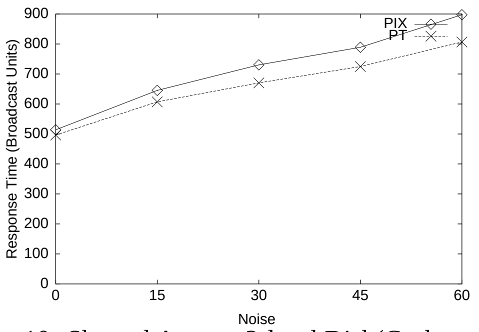

Figure 10: Skewed Access, 3-level Disk(Cache = 100)
Δ = 2, Varied Noise

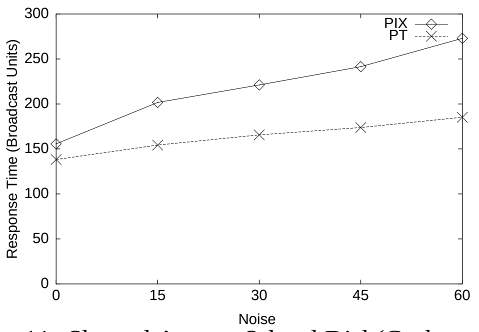

Figure 11: Skewed Access, 3-level Disk(Cache = 500)
Δ = 2, Varied Noise

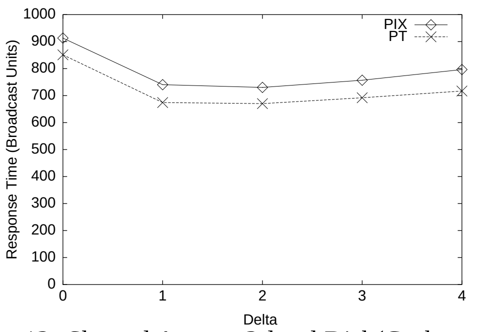

Figure 12: Skewed Access, 3-level Disk(Cache = 100)
Varied Δ, Noise = 0.30

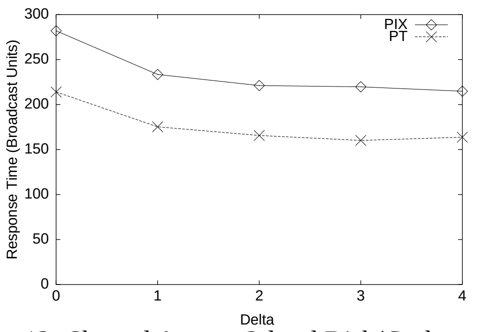

Figure 13: Skewed Access, 3-level Disk(Cache = 500)
Varied Δ, Noise = 0.30

from the cache once per major cycle. As the arrival time of the page nears, its $\mathcal{PT}$ value decreases, until the point it is replaced from the cache. The page is prefetched into the cache when it arrives again. The middle curve is for the page from the middle disk (Disk #2). Since Δ is set to 2, pages on this disk appear three times per major cycle, and therefore (as shown in the figure) $\mathcal{PT}$ expels the page from the cache three times during the cycle. Again, $\mathcal{PT}$ keeps the page in cache after it arrives, and expels it at the time closest to when it is due to arrive again. Comparing the curves for Disk #3 and Disk #2, it can be seen that $\mathcal{PT}$ gives more than twice the cache residency time to the page on the slower Disk #3 (77% per cycle) than to the page on Disk #2 ($0.116 \times 3 \approx 35\%$ per cycle).

$\mathcal{PT}$ is effectively determining the proper residency time for each page (based on its probability of access and its frequency of broadcast), and then distributing this time over the major cycle in a way that minimizes the expected delay for pages that have been removed from the cache. Finally, the bottom curve of Figure 9 shows the cache residency for a page that is stored on the fastest disk (Disk #1). Since this page is broadcast relatively frequently (5 times per major cycle), $\mathcal{PT}$ does not allocate any cache residency time to it. As a result, the page is brought into the cache only when there is an outstanding client request for it, and is replaced from the cache immediately. This is reflected by the fact that it stays only for 2 broadcast units (0.0003% of the cycle) in the cache – the think time before the next access. This behavior can be observed as the random spikes on the lower curve in the figure.

4.4 Expt. 4: Skewed Access, Multi-Level Disk

The fourth experiment investigates the combination of a skewed Zipf client access pattern (as was used in Expt. 2) with the multi-disk broadcast. Figures 10 and 11 show the performance of $\mathcal{PT}$ and $\mathcal{PIX}$ as *Noise* is varied using the 3-disk broadcast of the previous experiment with Δ set to 2, for the cache sizes of 100 pages and 500 pages respectively. Recall that increasing the noise parameter increases the discrepancy between the client access probabilities and the page broadcast frequencies chosen by the server. A higher degree of noise models the effect of a larger client population with differing access profiles.

When the cache is small (Figure 10), the performance of both algorithms is similar, and both are negatively impacted by increasing noise. With the larger cache (Figure 11), the performance of $\mathcal{PT}$ is much more robust. $\mathcal{PT}$'s tolerance of noise is due to its ability to fine tune the cache residencies of pages to account for their broadcast frequencies (as was described in the previous experiment). When the server places an important page on the slow disk, $\mathcal{PT}$ is able to compensate by giving that page more time in the cache, and ensuring that the time when the page is not cache resident is placed as close as possible to the subsequent arrival of the page on the broadcast.

Figures 12 and 13 show data points from the same experiment, with noise fixed at 30%, and Δ varied from 0 (i.e., a flat disk) to 4. In this case, it can be seen that with a large cache, $\mathcal{PT}$ is able to provide a significant performance improvement over $\mathcal{PIX}$ for a wide range of different disk shapes. Thus, it can be seen that $\mathcal{PT}$ is a very robust metric — it can better exploit client cache resources across a range of client access distributions, and broadcast programs, and can also help mitigate the negative performance impact of disagreement between client access probabilities and server broadcast frequencies (as represented by "noise").

5 Implementing $\mathcal{PT}$

The previous sections have shown the potential benefits of prefetching in a broadcast environment. However, the prefetching heuristic $\mathcal{PT}$ in its pure form is impractical— it requires $O(CacheSize)$ operations at every "tick" of the broadcast to determine the page with the lowest `pt` value. As a result, we have

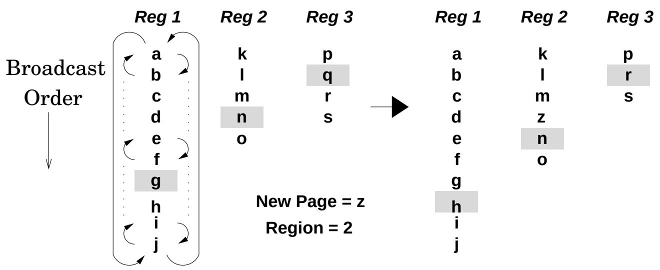

Figure 14: Page Replacement in $\mathcal{APT}$

designed an implementable approximation of $\mathcal{PT}$, called $\mathcal{APT}$, based on the insights gained from studying $\mathcal{PT}$.

$\mathcal{APT}$ begins with the space of cached pages ordered by probability of access and partitions this ordering into contiguous, non-overlapping regions. All pages in a region are assumed to have the same probability of access. Thus, each region has only one potential candidate for a replacement victim — the page in the region which arrives next on the broadcast, (i.e., the one with the smallest `t` value). $\mathcal{APT}$ evaluates `pt` values for each of these potential victims and evicts the one with the lowest `pt`. Consequently, the eviction decision for $\mathcal{APT}$ has a complexity of $O(Number\ of\ Regions)$ as opposed to $O(CacheSize)$ for $\mathcal{PT}$. Two remaining implementation problems must be addressed: 1) maintaining in constant time the next broadcast page in each region, and 2) determining the access probabilities of the client. We next address these problems in order.

Consider Figure 14. It shows an $\mathcal{APT}$ implementation with three probability regions for a flat (single-disk) broadcast. The pages in a region are organized in a doubly-linked circular chain sorted by their order of arrival in the broadcast. Region 1 shows examples of pointers connecting neighboring slots. Each region also has a pointer to that region's potential replacement victim (PRV), (i.e., the page with the smallest `t`). In the figure these are shown as shaded slots (pages g, n and q). Thus, one of the PRV must be the page with the smallest `pt` value since all pages in a region are assumed to have the same access probability. When a PRV is evicted (say g), the page following the old PRV becomes the new PRV (in this case, page h). Let the client access a new page z and q be chosen as the victim. Region 3 is then updated by freeing the slot for q and making r the new PRV. The new page is assigned a region based on its current estimated probability of access. Let z be allocated to Region 2. It is then inserted into the second chain before the PRV (n in the figure) because the PRV has not yet been broadcast and the new page will have the largest value for `t`. The size of the chains change dynamically as can be seen for regions 2 and 3 in the figure.

The implementation described above is for a flat broadcast; however, splitting each region into smaller sub-regions, each corresponding to a disk, allows us to extend this technique to handle a multi-disk. We do not give the details here due to space limitations. This extension requires no additional space and only a small constant number of extra pointer operations proportional to the number of disks.

As the number of regions (*NumRegions*) increases, the closer the caching decisions will approach those of $\mathcal{PT}$. Making *NumRegions* too large, however, will have an adverse effect on the performance of $\mathcal{APT}$ itself since its complexity is proportional to the number of regions. Our studies have shown that four regions give acceptable performance for the distribution under consideration. In the experiments that follow, we determined the probability range of each region by dividing the difference of the highest and the lowest access probability of any page in the database into *NumRegions* equal sized ranges. For example, let 0.001 and 0.041 be the lowest and the highest probability values and let the number of regions be four. Then the ranges of the regions are [0,0.011), [0.011,0.021), [0.021,0.031) and [0.031,1). Note that the first and the last regions have been extended to 0 and 1 to account for any pages which may, at some point, fall outside the range. A page's current probability determines the region it gets assigned to. This simple approach gave us acceptable performance.

Unlike demand-driven caching, prefetching also requires the access probabilities of pages not in the cache to determine if they are worth prefetching. We use a technique similar to the 2Q approach proposed in [JoS94] of maintaining a second queue (twoQ). When a page is evicted from the cache, its page number and its probability of access is entered into the twoQ in a FIFO fashion. A page is considered to be a prefetch candidate only if it is in the twoQ.

5.1 Expt. 5: $\mathcal{LIX}$ vs. $\mathcal{APT}$

In this section, we compare the performance of $\mathcal{APT}$ with $\mathcal{LIX}$. $\mathcal{LIX}$ was introduced in [AAF95] as an efficient constant time approximation of $\mathcal{PIX}$. $\mathcal{LIX}$ maintains an LRU-style chain ordered by probability of access for each disk of a multi-disk broadcast. $\mathcal{LIX}$ also keeps a running probability estimate for each broadcast page based on its past history of access.

The probability estimate for a page divided by its frequency of broadcast (a constant for each chain) is said to be the *lix* value of that page. $\mathcal{LIX}$ evaluates the *lix* values of the pages at the bottom of each chain and then evicts the page with the lowest *lix* value. The new page then enters the queue corresponding to the disk it is on.

$\mathcal{LIX}$ uses a very simple formula to estimate the probability of a page. To be fair, we also ran $\mathcal{LIX}$ using the same probability model as was used by $\mathcal{APT}$. We call this algorithm $\mathcal{LIX}$(2).

Consider Figure 15. It shows the same simulation space as Figure 12, i.e., *CacheSize=OffSet=100* and a *Noise* of 30%. The size of twoQ in this experiment was twice the cache size, i.e, 200 pages. In this experiment, $\mathcal{LIX}$ performs the worst and quickly degrades as Δ increases beyond 1. $\mathcal{LIX}$(2) does significantly better than $\mathcal{LIX}$ reflecting the difference between the two probability estimation techniques. The last two lines show the performance of $\mathcal{APT}$ and $\mathcal{PT}$. The ideal algorithm $\mathcal{PT}$ outperforms $\mathcal{APT}$ by a small margin. $\mathcal{APT}$ improves on the response time of $\mathcal{LIX}$(2) by 10-30%. In these experiments *NumRegions* was set at 4.

We performed similar experiments for larger cache sizes. As the cache size increases, $\mathcal{LIX}$(2)'s performance improvement over $\mathcal{LIX}$ decreases and $\mathcal{APT}$ does not follow $\mathcal{PT}$ as closely as in the case of the small cache. This is a manifestation of the probability model used by $\mathcal{APT}$ (and $\mathcal{LIX}$(2)). The probability model (which is described in the next section) produces poorer approximations for pages with low actual probability of access. Since the pages which are not cache-resident are those with a low probability of access, $\mathcal{APT}$ (and $\mathcal{LIX}$(2)) tend to make more errors. In spite of this handicap, $\mathcal{APT}$ provides a significant performance improvement over $\mathcal{LIX}$ and $\mathcal{LIX}$(2) and is more robust over a wider spectrum of disk configurations (as determined by Δ).

So far, we have been assuming that the client has a fairly good knowledge of its access probabilities. This might be too strong an assumption. In the next section, we develop a probability model and investigate how accurate these probabilities must be in order to retain the acceptable performance. (The results presented in this section for $\mathcal{APT}$ and $\mathcal{LIX}$(2) were for a mean error in the probabilities of

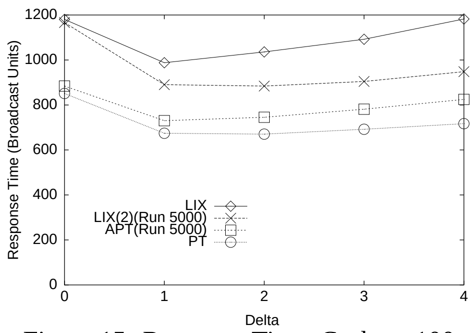

Figure 15: Response Time, Cache = 100
Skewed Access, Varied Δ, Noise = 0.30

Learning Run Length	Mean Error	Variance
500	1.380042	1.764039
1000	1.062268	0.739920
2000	0.705625	0.319217
5000	0.437711	0.146137
20000	0.227719	0.035724
35000	0.176085	0.022073
50000	0.146691	0.016047

Table 5: Error vs. Length of Run

about 0.4. The precise definition of error will be given in the next section.)

5.2 Expt. 6: Sensitivity of $\mathcal{APT}$

As we observed earlier, the $\mathcal{APT}$ algorithm relies on the client's knowing its own access probabilities. We assume that the client builds a model of these probabilities by sampling its own requests over some period of time. We count the number of accesses for a page and divide this number by the total number of accesses to compute the page's probability. As long as the access pattern is not changing, a longer sampling period should produce more accurate results. Such counting-based schemes have been proposed before for estimating probabilities for page replacement. The CLOCK [SPG90] and the frequency-based cache management algorithm introduced in [RoD90] are two examples.

In general, such a probability model will likely not be completely accurate. This section describes experiments which analyze $\mathcal{APT}$'s sensitivity to inaccuracies in the probability estimation. Different levels of accuracy are generated by varying the sample length in the learning run.

We use probability distributions that are derived from an initial run whose length (*InitRun*) ranges from 500 to 50,000 accesses. Let p_i and P_i be the approximate learned probability and the real probability of page i respectively. The error e_i for page i is given by the formula:

$$e_i = (|p_i - P_i|)/P_i$$

Thus, when $e_i = 1$, $0 \le p_i \le 2P_i$. The cumulative error in the distribution is expressed by the mean and the variance for these individual error values over all pages. The cumulative errors for the learned distributions that we used in the experiments are summarized in Table 5.

Figure 16 shows the response time of $\mathcal{APT}$ for different lengths of the learning run (*InitRun*). The simulation space is the same as Figure 15, i.e., *CacheSize=OffSet=100, Noise* = 30%. In general, higher sampling rates produce better performance. For the shorter initial runs of 500 and 1000 accesses, the response time is somewhat higher than the rest. However, beyond 5000 accesses, the performance does not improve substantially. The results in Section 5.1

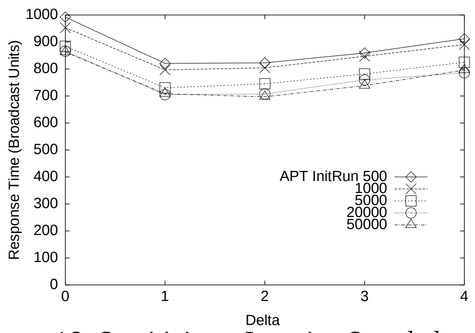

Figure 16: Sensitivity to Learning Sample length
Cache = 100, Varied Δ, Noise = 0.30

were for sample length of 5000. From Table 5, the error rate for this length of run is 0.437 with a variance of 0.146. This means that the approximate probability can be anywhere from half to 1.5 times the real probability. In spite of this error, $\mathcal{APT}$ does significantly better than $\mathcal{LIX}$.

We conclude from this that $\mathcal{APT}$ is fairly insensitive to perfect knowledge of the access distribution. This is the case since $\mathcal{APT}$ categorizes pages based on probabilities and in our distribution, the hot pages are very hot and the cold pages are quite cold. Thus, it is unlikely that pages will be incorrectly categorized. This is acceptable since multi-disks are beneficial only in environments where the access distributions are skewed.

Increasing the number of regions up to a point improved $\mathcal{APT}$'s performance as is expected. However, beyond that threshold, the response time degrades. Creating too many regions causes pages to get wrongly classified since a small error in probability can shift a page to the wrong region. As the accuracy of the probability estimation increases this effect is no longer seen. As described earlier, increasing the number of regions also linearly increases the cost of the algorithm. For the simulation space studied, four regions performed best.

6 Related Work

The basic idea of broadcasting information has been discussed before ([BGH92], [Gif90], [IVB94]). Our studies deviate from these in that we consider multi-level disks and their relationship to cache management.

Prefetching has been studied extensively as a technique to improve user performance in various areas including databases, operating systems and processor architectures. Prefetching techniques usually fall into two categories – prefetching based on user hints and prefetching based past access history.

[PaG94] proposes Transparent-Informed Prefetching which exploits I/O concurrency in disk arrays using hints from applications. Pre-paging based on hints was suggested in [Tri79]. The other class of prefetching algorithms is based on inferring access patterns from a stream user requests ([CKV93], [KoE91], [PaZ91], [GrA94] to name a few). Each of these techniques uses the access stream to develop a model of the user behavior. Our work leans towards the second category in the sense that we develop a model of client behavior based on past accesses. However, performance improvement in a broadcast environment comes not from achieving a higher hit rate but from reducing the cost of a cache miss. Consequently, the crux of our prefetching heuristic is not developing a better access model (which is sufficient to produce a better hit rate) but rather, using that knowledge to efficiently evaluate the cost metric like the `pt` value. Our simple counting-based probability estimator can, thus, be replaced by any of the other more sophisticated models to produce even better results.

There has also been some work on prefetching in Teletext systems([Amm87], [Won88]). In [Amm87], a caching strategy called the Linked Page scheme is described which uses embedded links in each page to decide what to prefetch next. Stashing and Hoarding used in the mobile computing environment ([KiS92], [TLA95]) are very similar to prefetching. However, their focus is to improve availability in the file system as opposed to performance.

7 Summary and Conclusions

In a previous work [AAF95], we described our notion of a multilevel broadcast disk and examined techniques for managing the client cache in this style of broadcast environment. In this work, we discuss the possibility of using the broadcast disk as a way to prefetch data into the client cache before the client requests it. We introduce a new prefetching heuristic called $\mathcal{PT}$ that takes into account the current position in the disk's rotation. We show that this style of prefetching can provide significant performance improvements. An important result of this work is that $\mathcal{PT}$ achieves better performance not by increasing the client hit rate, as in traditional systems like databases and file systems, but instead by reducing the cost of a cache miss.

We briefly introduced a simple technique that we have named tag-team caching. Tag-team caching illustrates that it is possible to increase the effective size of the cache by rotating pages in and out of the cache such that the cache residency time for a page is proportional to its probability of access and is inversely proportional to the amount of bandwidth allocated to that page in the broadcast. This, in effect, decreases the cost of a cache miss and is the primary intuition for analyzing our prefetch experiments.

The experiments showed that the $\mathcal{PT}$ heuristic does a very good job of balancing the bandwidth allocation with the cache residency time. It is also a robust metric— it can exploit the cache better across a wide range of broadcast programs and also, mitigate the negative impact of any disagreement between the client access pattern and the server program (as represented by "noise"). We then described $\mathcal{APT}$, a constant time approximation to $\mathcal{PT}$. Experiments with $\mathcal{APT}$ show that it is a good approximation to $\mathcal{PT}$ even when the estimated probabilities are significantly different from the real values. We showed that even a very simple learning scheme that infers the client access probabilities by sampling the access stream can provide reasonable results.

The techniques described in this paper along with their experimental validation suggest that the potential of a broadcast disk can be enhanced by adding a prefetcher to the client cache manager. The risks of prefetching in this setting are much less than that in conventional client/server systems since only local client resources are used to prefetch an item.

In terms of future work, we would like to look at other approximate implementations of $\mathcal{PT}$. For example, we have been considering an approach that is based on categorizing pages by time instead of probability. This could give us an algorithm that is even less dependent on knowing the access probabilities.

We also plan to revisit the tag-team approach as an alternative to the $\mathcal{PT}$-based approaches. In the single client case, we have preliminary results for setting up tag-team broadcasts for multiple pages with differing probabilities of access. We would like to extend this work to include the more general case of multiple clients. If tag-team caching could be made to perform as well or better than $\mathcal{PT}$, it would be an attractive alternative, since all decisions are made statically and impose no additional computational penalty on the client.

Acknowledgments

Franklin's research was supported in part by a grant from the University of Maryland General Research Board, NSF grants IRI-9409575 and IRI-9501353 and by a gift from Intel Corporation. Acharya and Zdonik were supported in part by ONR grant number N00014-91-J-4085 under ARPA order number 8220 and by a gift from Intel Corporation. Acharya was also supported in part by a fellowship from IBM Corporation.

References

[AAF95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," *Proc. of ACM SIGMOD*, Santa Cruz, CA (May 1995).

[AFZ95] S. Acharya, M. Franklin, and S. Zdonik, "Dissemination-based Data Delivery Using Broadcast Disks"," *IEEE Personal Communications* 2(6) , (Dec. 1995).

[Amm87] M. H. Ammar, "Response Time in a Teletext System: An Individual User's Perspective," *IEEE Trans. on Communications* 35(11):1159-1170, (1987).

[BGH92] T. G. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib, "The Datacycle Architecture," *CACM* 35(12) (December 1992).

[CKV93] K. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," *Proc. of ACM SIGMOD*, Washington, DC (May 1993).

[DDY90] Asit Dan, D. M. Dias, and P. S. Yu, "The Effect of Skewed Access on Buffer Hits and Data Contention in a Data Sharing Environment," *VLDB* (1990).

[Gif90] D. Gifford, "Polychannel Systems for Mass Digital Communications," *Communication of the ACM* 33(2) (February 1990).

[GrA94] J. Griffieon and R. Appleton, "Reducing File System Latency using a Predictive Approach," *Proc. of USENIX Summer Tech. Conf*, (June 1994).

[IVB94] T. Imielinski, S. Viswanathan, and B. R. Badrinath, "Energy Efficient Indexing on Air," *ACM SIGMOD* (1994).

[JoS94] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th VLDB Conf.*, Santiago, Chile (1994).

[KiS92] J. Kistler and M. Satyanarayanan, "Disconnected Operation in Coda File System," *ACM Trans. of Computer Systems* 10(1):3-25, (Feb 1992).

[Knu81] Don Knuth, *The Art of Computer Programming, Vol II*, Addison Wesley, 1981.

[KoE91] D. Kotz and C. S. Ellis, "Practical Prefeteching Techniques for Parallel File Systems," *Proc. of First PDIS*, Miami beach, FL (Dec 1991).

[PaZ91] M. Palmer and S. Zdonik, "Fido: A Cache That Learns to Fetch," *VLDB*, Barcelona (1991).

[PaG94] R. H. Patterson and G. A. Gibson, "Exposing I/O Concurrency with Informed Prefetching," *Proc. of Third PDIS*, Austin, TX (Sept. 1994).

[RoD90] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. of ACM SIGMETRICS*, Boulder, CO (1990).

[ShL94] S. Shekhar and D. Liu, "Genesis and Advanced Traveler Information Systems(ATIS): Killer Applications for Mobile Computing?," *Proc. of the Mobidata Workshop*, Rutgers Univ., New Brunswick, NJ (Nov 1994).

[SPG90] A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts, 3rd Edition*, Addison Wesley, 1990.

[TLA95] C. Tait, H. Lei, S. Acharya, and H. Chang, "Intelligent File Hoarding for Mobile Computers," *Proc. ACM Conf. on Mobile Computing and Networking*, Berkeley, CA (Nov. 1995).

[Tri79] K. Trivedi, "An Analysis of Prepaging," *Computing* 22(3), (1979).

[Won88] J. Wong, "Broadcast Delivery," *Proceedings of the IEEE* 76(12):1566-1577 (Dec. 1988).