Disseminating Updates on Broadcast Disks

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

Lately there has been increasing interest in the use of data dissemination as a means for delivering data from servers to clients in both wired and wireless environments. Using data dissemination, the transfer of data is initiated by servers, resulting in a reversal of the traditional relationship between clients and servers. In previous papers, we have proposed *Broadcast Disks* as a model for structuring the repetitive transmission of data in a broadcast medium. Broadcast Disks are intended for use in environments where, for either physical or application-dependent reasons, there is asymmetry in the communication capacity between clients and servers. Examples of such environments include wireless networks with mobile clients, cable and direct satellite broadcast, and information dispersal applications. Our initial studies of Broadcast Disks focused on the performance of the mechanism when the data being broadcast did not change. In this paper, we extend those results to incorporate the impact of updates. We first propose several alternative models for updates and examine the fundamental tradeoff that arises between the currency of data and performance. We then propose and analyze mechanisms for implementing these various models. The performance results show that, even in a model where updates must be transmitted immediately, the performance of the Broadcast Disks technique can be made quite robust through the use of simple techniques for propagating and prefetching data items.

1 Introduction

1.1 Broadcast Disks

Continuing advances in telecommunications, interconnectivity, and mobile computing have brought about a new class of information delivery applications based on *data dissemination*. In contrast to traditional, "pull-based" approaches, where data is brought from servers to client applications using remote procedure call (RPC) or request-response protocols, data flow in dissemination-based systems is initiated in a "push-based" fashion, i.e., from the server-side. Dissemination-based information systems have been proposed and/or implemented for both wired and wireless networks for a wide range of applications. Some examples include: stock quotation and trading systems [Oki93], advanced traveler information systems (ATIS) [Shek94], wireless classrooms [Katz94], fast database lookup [Herm87, Bowe92], and news delivery in both wireless [Giff90] and wired [Inte96] environments.

In previous work, we proposed and analyzed the *Broadcast Disks* model for disseminating data in a broadcast environment [Zdon94, Acha95b]. Specifically, Broadcast Disks are intended for environments where the communication channel among clients and servers is *asymmetric*. In an asymmetric communication environment, there is typically more communication capacity available from servers to clients than in the opposite direction. Such asymmetry can arise from physical network characteristics such as bandwidth differences, as in wireless mobile networks with cellular uplinks, cable TV with set-top boxes, and direct satellite broadcasting environments, or from properties of the workload as in information systems with a large ratio of clients to servers.

In an asymmetric communication environment, the data broadcast can be used to exploit the servers' advantage in transmission bandwidth in order to provide responsive data access to clients. The Broadcast Disks approach transmits data repetitively to a population of clients. A broadcast program is created containing all the data items to be disseminated and this program is transmitted cyclicly. The repetitive nature of the broadcast allows the medium to be used as a type of storage device ("disk"); clients that need to access a data item from the broadcast can obtain the item at the point in the disk's rotation when the item is transmitted. Repetitive broadcast has been investigated in several previous projects [Amma85, Wong88, Bowe92, Imie94a]. The Broadcast Disks model differs from this previous work in that it integrates two main components: 1) a novel "multi-disk" structuring mechanism that allows data items to be broadcast non-uniformly, so that bandwidth can be allocated to data items in proportion to the importance of those items; and 2) client-side storage management algorithms for data caching and prefetching that are tailored to the multi-disk broadcast.

Using the multi-disk model, data items can be placed on "disks" of varying size and rotational speed. An arbitrarily fine-grained memory hierarchy can be created by constructing a multi-disk broadcast such that the highest level is a disk that is relatively small and spins relatively quickly, while subsequent levels are pro-

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

gressively larger and spin progressively slower. Such a broadcast program is analogous to a traditional memory hierarchy ranging from small, fast memories (e.g., cache) to larger, slower memories (e.g., disks or near-line storage).

The client-side storage management policies take into account both the local access probability (using heuristics such as LRU or usage frequency) and the expected latency of a data item; items that reside on slower disks have higher average latencies. The caching policies [Acha95a] favor slow-disk data items, allowing clients to use their local storage resources to retain pages which are important locally, but are not sufficiently popular across all clients to warrant placement on a faster disk. Such policies allow clients to exploit their local resources in those cases where their access patterns differ from the average client population, while taking advantage of the improved delivery rate of globally popular items on the broadcast. The prefetching policies [Acha96] are more dynamic in the sense that they take into consideration not just the average latency for a data item, but rather, the time-to-arrival for items at a given instant. The prefetching policies strive to retain items in a client's cache during those portions of the broadcast cycle where their time-to-arrival is at its highest.

1.2 Coping with Updates

The focus of our previous studies was on the performance of the Broadcast Disks approach in the context of a read-only environment, where neither the broadcast program nor its contents were changed. These studies uncovered fundamental properties of the approach and demonstrated the impact of push-based data delivery on client cache management. Clearly, however, many of the applications that can best profit from a broadcast-based approach also require the dissemination of updates. Examples of such applications include: stock quotation systems, weather and traffic reports, and logistics applications.

In the Broadcast Disks environment, there are four types of changes that can be considered. These are:

1. *Item Placement* - Data items may be moved among the levels of a given Broadcast Disks hierarchy (i.e., a set of disks of varying sizes and speeds). Such movement may have an impact on the metrics used by the client-side caching and prefetching policies.
2. *Disk Structure* - The shape of the hierarchy itself may be changed; for example, the ratios of the speeds of the disks may be changed, slots may be added to or removed from a given disk, an entire disk may be added or removed, etc.
3. *Disk Contents* - The set of data items being broadcast changes, i.e., some pages being broadcast are no longer broadcast and/or new items are added to the program.
4. *Data Values* - Updates are made to data items that are already part of the broadcast program. Depending on the consistency model used, these updates may require clients to remove stale copies of the items from their caches.

For *Item Placement* and *Disk Structure* changes, there are no updates to the actual data values being broadcast; only the relative frequencies and/or the order of appearance of the data items already being broadcast are changed.[1] The impact of these first two types of changes, therefore, is primarily on performance. The metrics used by the caching and prefetching algorithms include information about the latency of data items. Thus, changes due to *Item Placement* and/or *Disk Structure* may cause the relative values of data items to be incorrect, resulting in sub-optimal usage of the client storage resources. Performance problems that might arise from such changes can be mitigated, to some extent, by broadcasting advance notice of the upcoming changes. This information would allow clients to appropriately adjust their caching behavior.

With the third change type, changes to *Disk Contents*, items that used to appear on the broadcast may be removed, while new items may appear. While this type of change does have an impact on the set of data items that appear on the broadcast, it can also be considered to be an extreme case of *Item Placement* change; items are moved to or from a disk having infinite latency. Viewed in this way then, given advance warning of the impending arrival or disappearance of a data item, clients may choose to cache an item before it is dropped from the broadcast or make room for an item that will appear.

The fourth type of change, *Data Value* updates, is qualitatively different from the others, as it raises the issue of *data consistency*. In Broadcast Disks or any other data-dissemination environment, clients access data from their local caches, while updates to data values are collected at a server site. In order to keep the clients' caches consistent with the updated data values, the client-cached copies of modified items must be invalidated or updated. Different applications, however, have differing consistency demands. Some applications need to access only the most recent values of data items, while others can tolerate some degree of staleness. Furthermore, in some consistency models, such as ACID transactions, consistency must be preserved across reads of multiple data items.

In this paper we focus on the problem of efficiently supporting *Data Value* updates. We propose and analyze extensions to the original Broadcast Disks model that are necessary to provide robust performance in workloads where the values of the data items being broadcast are allowed to change at the server while copies of those items may be cached at clients. As is discussed in Section 6, schemes for providing consistency in dissemination-based environments have been studied previously [Alon90, Bowe92, Barb94, Jing95, Wu96]. Our techniques differ from previous work due to the unique characteristics of the Broadcast Disks approach, which arise due to the combination of repetitive, multi-disk broadcast of data and associated client-side cache management protocols.

The key contributions of this work are twofold. First, the studies verify that the basic intuitions that were observed and the techniques that were developed previously for Broadcast Disks and the associated cache management policies in the read-only cases remain valid even in the presence of data item updates; and second, the results show that with the addition of a few simple mechanisms (described in Section 3), the performance of the broadcast can be made to approach that of the steady-state (no update) case for a fairly wide range of update rates.

1.3 Overview of the Paper

The remainder of this paper is structured as follows: Section 2 briefly describes the structure of Broadcast Disk programs, proposes several alternative models of consistency in the presence of data value updates, and lists our assumptions. Section 3 introduces the basic techniques used to disseminate updates in this environment. Section 4 presents a description of the simulation

[1] We assume that data items are self identifying. Alternatively, an indexing scheme can be used to locate and identify items, as proposed in [Imie94b].

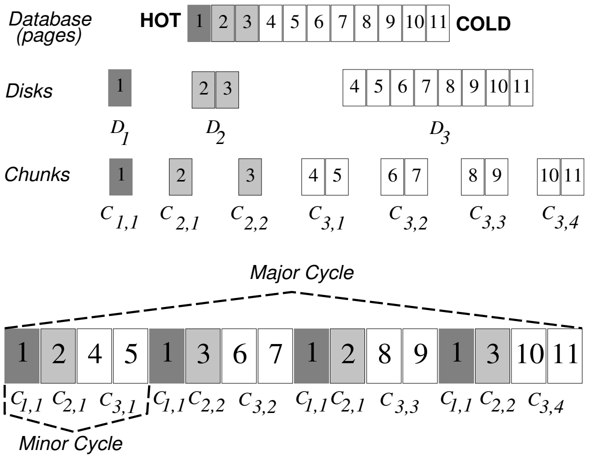


Figure 1: Deriving a Server Broadcast Program

environment used to study the various approaches to consistency. Section 5 contains the experiments and results. Section 6 provides a discussion of related work. Section 7 presents our conclusions and future work.

2 Updates and Broadcast Disks

2.1 Broadcast Disk Programs

In this section we briefly describe the structure of broadcast programs. A Broadcast Disk is a cyclic broadcast of data items (called *pages*) that are likely to be of interest to the client population. Multiple disks can be simulated on a single broadcast channel by broadcasting some pages more frequently than others. Each disk corresponds to those pages that have the same broadcast frequency. The desirable characteristics for a broadcast program have been outlined in [Acha95a]. Briefly, a good broadcast program is periodic, has fixed (or nearly fixed) inter-arrival times for repeated occurrences of a page, and allocates bandwidth to pages in accordance with their access probabilities.

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 process using a simple example. For a detailed description of the algorithm, the reader is referred to [Acha95a].

Figure 1 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* ($C_{i,j}$ 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.

2.2 Models of Consistency

As stated in the introduction, allowing disseminated (and possibly cached) data values to be updated creates the potential for data consistency problems. The notion of data consistency is, of course, application-dependent. In database systems data consistency is traditionally tied to the notion of transaction serializability. In practice, however, few applications demand or even want full serializability, and much effort has gone into defining weaker forms of correctness (e.g., [Bere95, Kort95]). Because dissemination-based information systems are only now beginning to emerge, the notion of data consistency for applications in such systems is even less well-understood.

In this paper we focus on a Broadcast Disks environment where all updates are collected at the server and clients access the data in a read-only fashion. Such an access pattern is likely in many data dissemination applications such as stock quotation systems, news and weather services, etc. In such an environment there are a number of reasonable choices for consistency models. These include:

- *Latest Value* - Clients must always access the most recent value of a data item. This level of consistency is what would arise naturally if clients performed no caching and servers broadcast only the most recent values of items. Note that this model, although often used in dissemination-based and/or mobile database work, is far weaker than serializability — there is no notion of mutual consistency among groups of items.
- *Quasi-caching* [Alon90] - Consistency can be defined on a per-client basis using constraints that specify a tolerance of slack compared to *Latest Value*. Some examples of such constraints include values that must be within $x\%$ of the latest value (e.g., for scalar attributes) or within y minutes of the latest value or within the last z changes of the value. Quasi-caching gives clients the ability to use cached data items in cases where they may otherwise be unsure of their correctness, and may allow the server to more lazily disseminate updates when the caching constraints of the clients are known.
- *Periodic* - Data values only change at specified intervals. In a Broadcast Disk environment, such intervals can be tied to the major or minor cycles of the broadcast. If clients are allowed to cache data, then they are guaranteed that such data will remain valid for the remainder of the interval during which the data was initially read. Such an approach was used in the Datacycle system [Bowe92].
- *Serializability* - If client reads and server updates are performed within the context of transactions, then serializability (or some reduced degrees of isolation [Bere95]) may be an appropriate notion of consistency. Datacycle implemented serializability using a combination of periodic update, optimistic concurrency control at clients, and the broadcasting of update logs.
- *Opportunistic* - For some applications (or under certain conditions) it may be acceptable to read any version of a data item that is available. This notion of consistency is implemented by allowing clients to freely access data from their cache without worrying about consistency. Such an approach has obvious advantages for applications where long-term disconnection from the server is likely. In some cases,

reading data of questionable quality may be preferable to having no data access at all.

In this paper we focus on two models of consistency: *Latest Value* and *Periodic*. These models allow us to study the consistency maintenance techniques under two different levels of overhead in a manner that is relatively independent of application semantics. *Latest Value* is likely to cause the greatest overhead for update dissemination— every update may require data cached at clients to be immediately invalidated or updated. Thus, *Latest Value* provides a good test of the efficiency of the mechanisms used to ensure consistency. The *Periodic* model, which places fewer demands on the efficiency of consistency maintenance, is useful for several reasons. First, it fits well with the cyclic behavior of the Broadcast Disk model. Also, it is similar to the basic approach used in Datacycle, and, as shown in the Datacycle work, it can be used as a foundation on which to build a serializable model. In this paper, we use a *Periodic* model in which consistency actions are performed once per major cycle.

Compared to the *Latest Value* and *Periodic* models, the *Quasi-caching* and *Opportunistic* models depend more heavily on the semantics of data access in specific applications. The *Serializable* model is not relevant for our current model of Broadcast Disks, as our clients are not transactional. Our focus in this paper is on the fundamental tradeoffs involved with the mechanisms for maintaining data consistency in the Broadcast Disks environment; thus, we perform our experiments using the *Latest Value* and *Periodic* models. Studies of the other three models, however, are an interesting avenue for future work, particularly in the context of specific applications.

2.3 Assumptions

The implementation of the various models of consistency requires the coordination of both the server broadcast policy and client cache management. For example, in order to implement the *Latest Value* model, the server must, at a minimum, broadcast an invalidation message when a data item is updated, and clients must either process that invalidation or refrain from accessing the data item from their caches until its validity can be ascertained. In this study, we make the following assumptions about the environment:

1. *All updates are performed at the server.* As stated previously, we assume a system in which clients access data from the broadcast in a read-only manner, while updates are applied at the server and disseminated from there.
2. *Clients have no back-channel capacity.* In this study, clients do not have a mechanism for sending messages to the server. Thus, for example, clients can not send special requests to the server for missing or invalidated data items.
3. *Active clients continuously monitor the broadcast.* We assume an environment such as CNN-at-Work, in which active clients can constantly monitor the broadcast. If a client stops monitoring the broadcast, then it can no longer trust the contents of its cache. Techniques for implementing cache consistency for intermittently connected clients have been studied in [Barb94, Jing95, Wu96].
4. *Clients remain active for significant periods of time.* Once a client begins monitoring the broadcast, it does so for a period of time sufficient for its cache to fill and reach a steady state. That is, we focus on the steady state behavior of connected clients.

Given these assumptions and the *Latest Value* and *Periodic* data consistency models as described in the previous section, we next describe techniques that will be useful for disseminating updates in a consistent manner in the Broadcast Disks environment.

3 Consistency Techniques

The implementation of data consistency requires mechanisms to ensure that clients see only valid states of the data, or at least, do not unknowingly access data that is *stale* according to the rules of the consistency model. Each model of data consistency places demands on both the server broadcast and on client cache management. The server is responsible for informing clients either prior to or at the instant that a page is broadcast that their cached copy may be stale. In the broadcast environment, where servers have no knowledge of the contents of client caches, the server must broadcast consistency information for any clients that may be affected. On the other hand, it is the responsibility of the clients to obtain the consistency information and take appropriate actions. In the case where a client may have missed consistency information due to, say, disconnection or to broadcast errors, the client must avoid accessing any mutable cached data until it can ascertain the validity of its cache contents.

Problems related to consistency arise in most other systems where client caching is used, such as multi-processor architectures [Arch86], distributed file systems [Levy90], distributed shared memory [Nitz91], and client-server database systems [Fran96]. In such systems, there are two basic techniques for communicating consistency information: invalidation (also called Write-Invalidate) and propagation (also called Write-Broadcast). For invalidation, the server sends out messages to inform the client of modified pages. The client removes these pages from its cache. For propagation, the server sends updated values, and the client replaces its old copy with the new one.

Invalidation and propagation are also useful techniques in the Broadcast Disk environment. The nature of this new medium, however, changes the relative tradeoffs between them and introduces some additional opportunities. In this section, we briefly describe how these techniques are introduced into the broadcast paradigm.

3.1 Invalidation

The quickest way to ensure that clients do not access stale data is through the use of invalidation. Invalidation messages can be quite small, requiring only that the page that has become invalid be identified. When a client receives an invalidation, it checks to see if it has a locally cached copy of the affected page, and if so, removes the page from its cache (or simply marks it "invalid"). In this paper invalidations are implemented through the use of an *invalidation list*. An invalidation list is simply a structure that contains the IDs of pages that should be invalidated at a particular instant. Under the Periodic consistency model, updates are saved up and an invalidation list containing possibly multiple page IDs is broadcast at a pre-specified interval, such as at the beginning of a major or minor cycle. In the systems studied in this paper, invalidation lists contain the IDs of all pages that have been updated since the last invalidation list was broadcast.

Invalidation, while efficient to implement, can have a dramatic effect on the performance of client caches. When a client first "tunes in" to a broadcast program, it faults in (or prefetches) pages from the broadcast until its cache is full. From that point on, the cache is managed using a *cache replacement policy* — if a new page is to be brought into a full cache, an existing cached page must be chosen as a victim and replaced from the cache. The phase in which a client is initially loading its cache is known as the *warm-up phase*. Once the cache has been full for some time, and data is being cycled in and out of the cache, the client is said to be in *steady state*. In the Broadcast Disk model, the best broadcast program for clients in the warm-up phase is different than the best program for clients in steady state. As described in [Acha95a], once a client has warmed up its cache, it is wasteful of bandwidth to frequently broadcast pages that are highly likely to be in the cache. To address this issue, the Broadcast Disk model includes the notion of *offset*. As is described in Section 4, the *offset* parameter moves the *hottest* pages from the fastest disk of a Broadcast Disk hierarchy to the slowest disk. The use of *offset* provides the most efficient use of bandwidth when client caches are in steady state.

Invalidations, however, have the effect of moving client caches away from steady state. If a page is updated, it is removed from the client's cache regardless of how recently or how often it has been accessed. Thus, invalidations undermine the steady-state assumption on which the use of *offset* is based, so a program that is generated using *offset* is likely to be sub-optimal in the presence of invalidations.

3.2 Auto-prefetch

The potential negative performance impact of invalidations is exacerbated if clients access the Broadcast Disk in a purely demand-driven manner. When important pages are invalidated, a demand-driven approach will fault them into a client's cache one-at-a-time. If *offset* is used, the performance penalty can be tremendous, as the most important pages have been placed on the slowest disks. This problem, however, can be easily addressed in the Broadcast Disk environment through the use of a technique we call *auto-prefetching*. As demonstrated in [Acha96], the broadcast medium is in general, highly conducive to data prefetching; data pages continually flow past clients so the clients can choose to prefetch passing pages without imposing any additional load on shared resources (only the local client resources are impacted).

Auto-prefetching is based on the intuition that with good cache management, the pages in a client's cache tend to be pages that are valuable to the client, whereas pages that are not cached tend to be less valuable. Auto-prefetching simply turns on prefetching at a client for any data pages that are invalidated from that client's cache. After a page is invalidated and marked for auto-prefetch, it is automatically brought into the cache the next time it appears on the broadcast. As will be shown in the performance experiments, auto-prefetching goes a long way towards mitigating the negative impact of invalidation.

3.3 Propagation

In contrast to invalidation, propagation delivers the new value of a changed data page, allowing a client to install that new value as a replacement for a cached value that would otherwise become stale. The advantage of propagation is that it can help clients stay closer to their steady state, especially when used in conjunction with auto-prefetch, as described above. The disadvantage of propagation is that it can be wasteful of bandwidth. When a page that is not cached at any clients (or cached at only very few clients) is propagated, the bandwidth used for that propagation effectively goes unused.

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 1: Client Parameter Description

In this paper, we implement propagation through the use of a *propagation list*, which is simply a concatenation of new page values. As with invalidations, propagations can be saved up and broadcast periodically, or they can be broadcast immediately. It is important to note however, that the dissemination of invalidations and propagations need not be done using the same policy. Correctness can be ensured through the timely broadcast of invalidations, in which case propagation is performed as a performance enhancement. Also, it is important to note that the maximum rate of propagations is significantly lower than that of invalidations, due to the differences in the bandwidth requirements — propagations cannot be transmitted any faster than the broadcast speed (in pages per unit time) of the medium. In the system studied here, a propagation list contains the values of all pages that have been updated since the last propagation list was broadcast.

4 Modeling the Broadcast Environment

Our model of the broadcast environment has been described previously [Acha95a, Acha96]. The results presented in this paper are based on the same underlying model, extended to include updates. We briefly describe the model in this section.

In the broadcast environment, the performance of a single client for a given broadcast program is independent of the presence of other clients. As a result, we can study the environment by simulating only a single client. However, the server can potentially generate a broadcast program that is sub-optimal for any particular client since it tries to balance 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 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.

4.1 The Client Model

The parameters that describe the operation of the client are shown in Table 1. 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 read access, the client waits *ThinkTime* broadcast units and then makes the next read request. The *ThinkTime* parameter allows the cost of client processing relative to page broadcast time to be

ServerDBSize	No. of distinct pages in broadcast
NumDisks	No. of disks
$DiskSize_i$	Size of disk i (in pages)
$RelFreq_i$	Relative broadcast frequency of disk i
Offset	Offset from default client access
Noise	% workload deviation

Table 2: Server Parameter Description

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. Within the range, the access probabilities follow a Zipf [Knut81] distribution. 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. Similar to earlier models of skewed access [Dan90], 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.

4.1.1 Client Cache Management

We use the $\mathcal{LIX}$[Acha95a, Acha96] algorithm to maintain the client cache. $\mathcal{LIX}$ is a constant time implementation of the pure cost-based algorithm $\mathcal{PIX}$. $\mathcal{PIX}$ chooses as a victim the page in the cache with the lowest p/x value, where p is the probability of access and x the broadcast frequency of the page. Cost-based caching strategies like $\mathcal{LIX}$ (and $\mathcal{PIX}$) have been shown to perform significantly better in a broadcast environment than strategies like LRU which just use probability information [Acha95a].

$\mathcal{LIX}$ is a modification of LRU that takes into account the broadcast frequency. LRU maintains the cache as a single linked-list of pages. When a cache-resident page is accessed, it is moved to the top of the list. On a cache miss, the page at the end of the chain is chosen for replacement. In contrast, $\mathcal{LIX}$ maintains a number of smaller chains: one corresponding to each disk of the broadcast ($\mathcal{LIX}$ reduces to LRU if the broadcast uses a single flat disk). A page always enters the chain corresponding to the disk in which it is broadcast. Like LRU, when a page is hit, it is moved to the top of *its own* chain. When a new page enters the cache, $\mathcal{LIX}$ evaluates a *lix* value (see next paragraph) only for the page at the bottom of each chain. The page with the smallest *lix* value is ejected, and the new page is inserted in the appropriate queue. Because this queue might be different than the queue from which the slot was recovered, the chains do not have fixed sizes. Rather, they dynamically shrink or grow depending on the access pattern at that time. $\mathcal{LIX}$ performs a constant number of operations per page replacement (proportional to the number of disks) which is the same order as that of LRU.

$\mathcal{LIX}$ evaluates the *lix* value of a page by dividing an estimated probability value of this page by its frequency of broadcast. Similar to [Acha96], we assume the client develops this probability model for each page by sampling its own requests over a 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. For experiments in this paper we chose a sample length of 10,000 accesses. Though very simplistic, this model provides acceptable results.

4.2 The Server Model

4.2.1 Server Execution Model

The parameters that describe the operation of the server are shown in Table 2. The server broadcasts pages in the range of 1 to *ServerDBSize*, where *ServerDBSize* $\geq$ *AccessRange*. These pages are interleaved into a broadcast program as described in Section 2.1. 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. $DiskSize_i$, $i \in [1..NumDisks]$, 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*. The frequency of broadcast of each disk i relative to the slowest disk is given by the parameter $RelFreq_i$.

The remaining two parameters, *Offset* 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, 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 heavily accessed pages in the client's buffer pool which makes broadcasting them frequently a waste of bandwidth. In such cases, the best broadcast can be obtained by shifting these hottest pages from the fastest disk to the slowest disk. *Offset* is the number of pages that are shifted in this manner. A typical value for *Offset* is *CacheSize*, wherein the cache size hottest pages are pushed to the slowest disk, bringing in the colder pages to faster disks.

In contrast to *Offset*, which is used to provide a better broadcast for the client and is a property of the broadcast, the parameter *Noise* is used to introduce disagreement between the needs of the client and the broadcast program generated by the server and is a property of the simulation. 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. As the noise increases, the client's performance can be expected to degrade. For further implementation details, the reader is referred to [Acha95a].

4.2.2 The Update Model

The parameters that describe the update model are shown in Table 3. Updates are generated in the system by simulating a writer process at the server. The writer process is a simple two-step write-wait loop. After each write access, the writer waits *UpdateThinkTime* broadcast units before making the next update request at the server. The *UpdateThinkTime* parameter determines the rate of updates in the system — smaller the value, more frequent the updates.

The writer updates pages at the server using a Zipf distribution (with parameter θ_u) similar to the read access distribution at the client. However, unlike the read access, which is restricted to the first *AccessRange* pages, the write distribution is spread over the complete range 1 to *DBSize*. This is because unlike reads which are local to a particular client, the updates happen at the server (and are not specific to a single client) and thus, encompass the entire database.

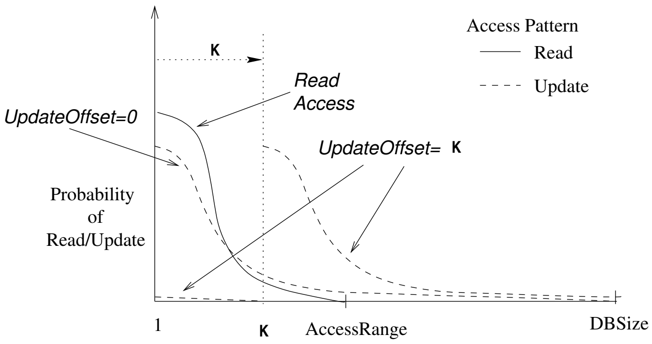


Figure 2: Read versus Update distribution

UpdateThinkTime	Time between updates at server (in broadcast units)
UpdateOffset	Update and read access deviation
θ_u	Update Zipf distribution parameter

Table 3: Update Parameter Description

We use a parameter *UpdateOffset* to introduce disagreement between the client access pattern and the update distribution at the server (Figure 2). When *UpdateOffset* is 0, the overlap between the two distributions is the greatest, i.e., the client's *hottest* pages (for read access) are also the most frequently updated pages. This scenario would be expected to produce the worst performance for the client. An *UpdateOffset* of K shifts the update distribution by K pages as shown in Figure 2 making the pages which are most often updated be those that are of lesser interest to the client (for read).

We assume that an invalidation list does not cost any bandwidth and can be piggybacked on a regular page broadcast. This is a reasonable assumption because the size of such a list would be negligible compared to the size of a page. Every propagation of a new value, however, consumes a page worth of bandwidth, i.e., one broadcast unit.

If the server uses a propagation-based scheme to mitigate the effect of invalidations, the regular order of the broadcast is disturbed due to the infusion of the propagation list. The propagation list is broadcast by first suspending the regular broadcast, then transmitting the propagation list and then continuing the regular broadcast. Note that the propagation list only changes the absolute broadcast time of the pages (by the length of the list); the relative order among pages in the broadcast structure remains the same.

5 Experimental Results

In this section, we use the simulation model to explore the influence of updates on the client. The primary performance metric 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. Two sizes for the client cache were used in the experiments — 100 and 500 pages. A three-disk broadcast was used for all the experiments. The size of the fastest disk was 300 pages, the medium disk was 1200 pages and the slowest disk was 1500 pages. The relative spin speeds of the three disks were 5, 3 and 1, respectively. As illustrated in Section 2.1, these parameters generate a broadcast program with a major cycle of 6600 pages and 15 minor cycles of 440 pages each. The results in these experiments were obtained once the client performance reached steady-state.

ThinkTime	2
UpdateThinkTime	2,5,10,20,25
ServerDBSize	3000
AccessRange	1000
CacheSize	100(Small), 500(Large)
θ, θ_u	0.95
Offset	0, *CacheSize*
UpdateOffset	0, 500
Noise	0%,10%,25%,50%
RegionSize	50
NumDisks	3
$DiskSize_{1,2,3}$	300,1200,1500
$RelFreq_{1,2,3}$	5,3,1

Table 4: Parameter Settings

As stated in Section 3, updates and subsequent invalidations disturb the steady-state behavior of the client cache that we observed in read-only workloads. This section shows that we do not have to abandon the intuitions that were developed there. In particular, the $\mathcal{LIX}$ caching algorithm is still sound and the use of a broadcast offset can be retained without significant performance degradation. The update case, however, introduces new trade-offs that must be considered in order to stay close to the steady-state performance shown before. This section explores these tradeoffs.

The rest of this section is as follows. We first study the client performance when updates are transmitted at the beginning of every major cycle. The next set of experiments are for the case when the updates are disseminated immediately as they happen at the server. Finally, we describe refinements to the strategies proposed in Section 3 and do a sensitivity analysis in the presence of *noise*. Recall that *noise* is used as a mechanism to simulate disagreement between the server broadcast and the client needs due to the presence of multiple clients in the system.

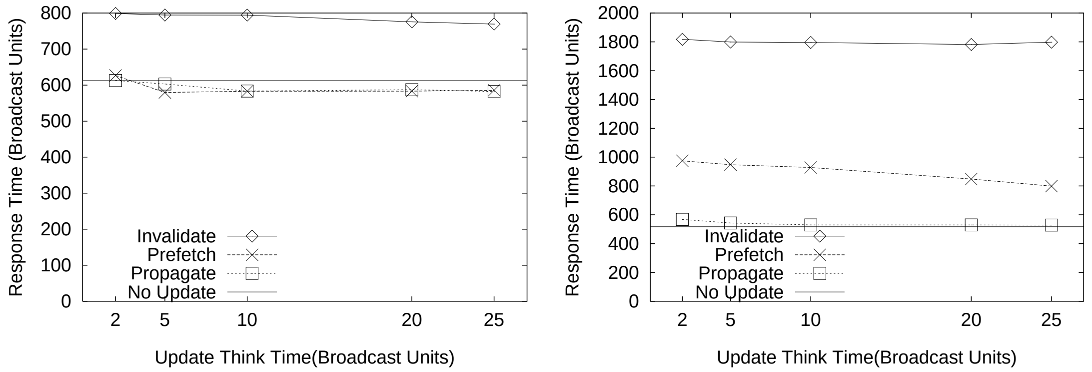
It should be noted that the results presented here are a small subset of those that have been obtained. They have been chosen as representative samples to make this discussion as compact as possible.

5.1 Major Cycle Update

First, we consider the case in which updates are communicated to the client at the start of a major cycle. As was mentioned earlier, this case is useful when the stability of values must be maintained. This case illustrates some very fundamental principles regarding the nature of the broadcast medium. The experiments in this section are for the case when there is no *noise* in the system.

We begin by comparing three basic cache update techniques—*invalidate*, *prefetch*, which uses auto-prefetch (as described in Section 3) to re-acquire invalidated pages, and *propagate*, which combines propagation with auto-prefetch. For invalidate, the server sends a list of updated page numbers at the end of the cycle. For propagate, the server follows the invalidation list with the actual updated pages, and for prefetch, the server sends an invalidation list at the end of a cycle, and the client prefetches new values of the pages from the next major cycle.

Figure 3 shows a comparison of these three techniques for a cache of 100 pages when the client's hot pages (for read) are also the most frequently updated (*UpdateOffset* = 0). The graph on the left (Fig 3a) is for the case of no offset while the graph on the right (Figure 3b) represents the broadcast with an offset of *CacheSize*,



(a)*Offset=0* (b) *Offset*=100

Figure 3: Sensitivity to updates once a cycle, Cache=100 pages, *UpdateOffset*=0

i.e., the first 100 pages are shifted to the slowest disk. The x-axis, *UpdateThinkTime*, reflects the intensity of updates at the server. This is also measured in broadcast units, and is the number of pages broadcast per update posted at the server. For an update think time of 2, a page is updated for every two pages broadcast. The solid line in all the graphs corresponds to the read-only case (an infinite update think time).

The first thing to notice in Figure 3a is that the response time of invalidate is very poor. Here, a large number of pages are being removed from the cache (recall that *UpdateOffset* = 0) at the end of the cycle and they do not return until they are requested again. The invalidation severely pushes us away from the steady-state case. The latency caused by having to fault in those missing pages one-at-a-time is so large that many of the benefits of caching are not realized and the performance suffers a great deal.

Prefetch and propagate have approximately the same performance as the read-only case here, because the pages that are updated tend to come back before a new request arrives. In the case of propagate, the pages return in the propagation list which is broadcast immediately after the changes are communicated. Prefetch does as well as propagate for two reasons. First, since there is no offset, the client's hottest pages are on the fastest disk and thus, they are auto-prefetched without significant delay. Secondly, the cache is small and it is quickly restored to the state before the invalidations. Thus, prefetching is required in this case. Nothing is gained from additional propagation when the updated pages are on the fastest disk since the broadcast disk itself is acting as an effective propagation medium.

Before leaving Figure 3a, we point out an interesting effect that is an artifact of the $\mathcal{LIX}$ implementation. Notice that both invalidate and propagate do a bit better than the pure read-only case for *UpdateThinkTime* greater than 3 or 4. $\mathcal{LIX}$ approximates probabilities (as does LRU) and thus, sometimes makes incorrect decisions for choosing victims during page replacement. Since, in this case, the updates coincide with the access pattern, updated pages (the ones $\mathcal{LIX}$ should cache) are being pulled out of the chains and reinserted at the top because they are later auto-prefetched. Thus, the pages that are accessed infrequently drift to the bottom of the LRU chains faster, flushing them more quickly from the cache than in the no-update case.

Figure 3b shows the same case as Figure 3a, but with an offset equal to the cache size of 100. Recall that offset is used to improve performance in the read-only case. Notice that here invalidate does even worse than before, but also the performance of prefetch deteriorates relative to propagate. At very high update rates, this difference is roughly an 80 to 90% degradation. By introducing an offset, the hot pages, which are also the most volatile pages, are being broadcast relatively infrequently since they are now on the slowest disk. The only way to improve this situation is to propagate the new values to counteract the slowdown introduced by the offset.

We now move on to the case in which the read access does not match the update pattern. This is introduced by setting the *UpdateOffset* to 500 as in Figure 4. We start by looking at Figure 4a in which *Offset* is zero. For lower update rates (the right-hand end of the graph), invalidate is similar to propagate for three reasons:

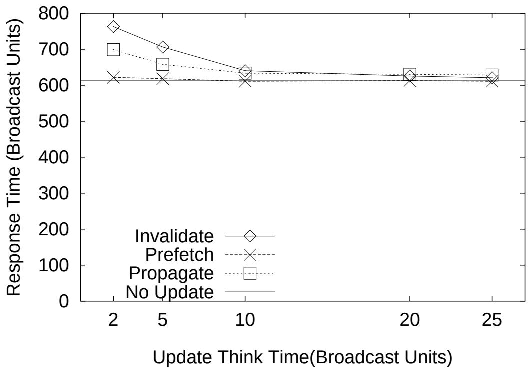
1. Not many cached pages are invalidated.
2. Most of the cached pages are from the fast disk and thus have a low latency.
3. The propagation list is small.

For the first two reasons, prefetch alone gets you back to the read-only case, much as it did in Figure 3a. The number of pages that must be re-acquired is small and the overhead of a large propagation list is not cost effective. As we move to the left, thereby increasing the update rate, the first and third reasons are no longer true. More cached pages are invalidated and the propagation list is longer.

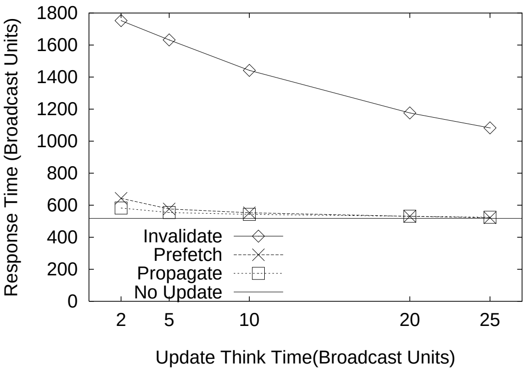
Notice that in Figure 4a, prefetch is better than propagation when the update rate is high. A fundamental principle at work here is that propagation only helps when you are propagating what the client needs. If the propagation includes pages that are not needed, performance degrades because of the wasted bandwidth. For items that are not accessed often or that come back quickly in the broadcast, it is good enough to allow them to drift back into the cache through prefetch from the broadcast.

As we move to a larger cache size, the cached pages will come from other than the fastest disk (ref. point 2 above). Recall that the size of the first disk is 300 pages. When this happens, the need for propagation becomes greater. The curves for large cache size have been omitted because of space limitations, but their shape is the same as those of Figure 4 with prefetch doing somewhat worse.

In Figure 4b, we introduce an offset equal to the cache size (i.e., 100). Here, the performance of invalidation is again extremely poor since the hottest pages are on the slowest disk. Thus, point 2 in the above list is not true for this case. Both prefetch and propagate perform well because they are both returning updated pages to the cache fast enough to keep up with demand. This differs from the the analogous *UpdateOffset*=0 case (Figure 3b)



(a) *Offset=0*



(b) *Offset*=100

Figure 4: Update does not match read access, Cache=100 pages, *UpdateOffset*=500

in that the number of hot pages that are being updated is small. Thus, prefetch is good enough (except at very high update rates).

In this set of studies, for the update once per major cycle case, we have shown that there exist algorithms that can compensate for updates in the system. The client, however, has to prefetch invalidated pages in order to do well. In general choosing propagation in this context produces good results, although for *UpdateOffset* = 500, we can do almost as well with prefetch alone. Given the extremely high response times of the invalidate strategy, we will no longer discuss its performance in the following sections.

As stated above, the best broadcast for steady-state situations is one in which the hottest *CacheSize* pages are offset onto the slowest disk. The benefit of an offset is visible in the lower response time (by about 16%) for the no update case in Figure 3b (*Offset*=100) over Figure 3a (*Offset*=0). We want our broadcast disk system to perform well for both high and low update periods and for clients that access volatile and non-volatile data. In order for the broadcast to provide clients with the best performance possible, we strive to retain the offset in the broadcast program and implement strategies which mitigate the effects of updates. Thus, we will only consider the case of a *CacheSize* offset in what follows.

5.2 Continuous Update

In this section, we study an alternative consistency model in which updates are communicated to the clients as quickly as possible. Since we are only considering broadcasts with an offset, as explained above, the server must propagate; however, we now must ask how often should the propagation occur since we are no longer constrained to once per major cycle. The options we consider are a) propagate continuously (immediately following invalidations) b) propagate at minor cycle boundaries (although this is an arbitrary intermediate point) and c) propagate at major cycle boundaries.

The four graphs in Figure 5 illustrate various propagation rates for small and large caches and for an *UpdateOffset* of 0 and 500. In all of these cases, continuous propagation never beats minor cycle propagation. The lesson here is that it is possible to propagate too often since this will waste bandwidth. In other words, it is best to propagate at a rate that just keeps up with demand— less often causes expensive faults, and too often wastes bandwidth. We will study this tradeoff in more depth in what follows.

The two graphs in the left-hand column of Figure 5 show the results for a small (Figure 5a) and a large (Figure 5c) cache with an *UpdateOffset* of zero. In both of these cases, because items of interest to the client are being updated, the server must propagate more than once per major cycle. Notice the large gap between major cycle propagate and minor cycle propagate. Surprisingly, there is little difference between minor cycle propagate and continuous propagate. In fact, with a small cache (Figure 5a) and a high update rate (e.g., *UpdateThinkTime* = 2), minor cycle propagate does slightly better than continuous propagate. This is because minor cycle propagation gets updates to the client fast enough and uses less bandwidth by propagating a page that is updated multiple times within a minor cycle only once. The additional bandwidth consumed in continuous propagate by repeatedly propagating the same page simply slows everything else down.

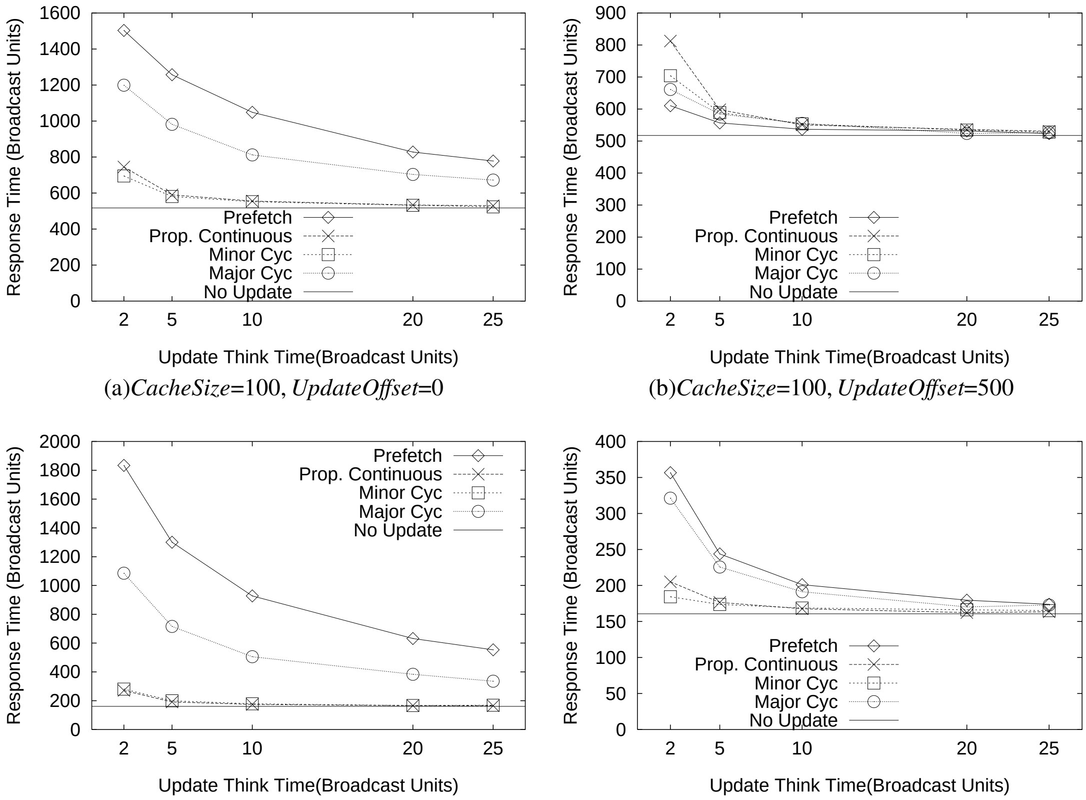
An *UpdateOffset* of 500 is introduced for both the graphs in the right-hand column of Figure 5. For a big cache (cache size = 500), minor cycle propagation is still preferred over continuous propagation (Figure 5d). Even though a smaller percentage of cached pages are changing, a larger cache means that there are still a significant number of pages that are being invalidated. The penalty of not getting these pages back quickly makes major cycle propagation unresponsive.

The small cache case (cache size = 100) with an *UpdateOffset* of 500 is the only case in which minor cycle propagation does not win(Figure 5b). Here, in the case of very high update rates, a pure prefetch scheme is preferable but only by about 10%. When the update does not match the read-access pattern and the cache is small, propagating the changes degrades performance since it uses bandwidth for uninteresting pages. The large number of propagated pages, even at low rates such as once per major cycle, is enough to slow down the normal broadcast.

In summary, minor cycle propagation does best in all cases except for a small cache and a high update rate and when there is a large discrepancy in the update and the access pattern. In this case, prefetch alone performs better but only by a small amount. Thus, for most cases minor cycle propagation is the technique of choice.

5.3 Refining Our Results

In the previous section, we showed that under most circumstances, a propagation algorithm that strikes a balance between infrequent (once a major cycle) and continuous propagation is the most desirable. Propagating at minor cycle boundaries is one example of such a compromise. In this section, we investigate the possibility of propagating a "subset" of the changes as a way to ameliorate the waste of bandwidth introduced by frequent propagation. If we



(a)*CacheSize*=100, *UpdateOffset*=0 (b)*CacheSize*=100, *UpdateOffset*=500

(c)*CacheSize*=500, *UpdateOffset*=0 (d)*CacheSize*=500, *UpdateOffset*=500

Figure 5: Response Time vs. Continuous Update for various cache sizes and *UpdateOffset*

can propagate *good bets* frequently, we might be able to improve our results.

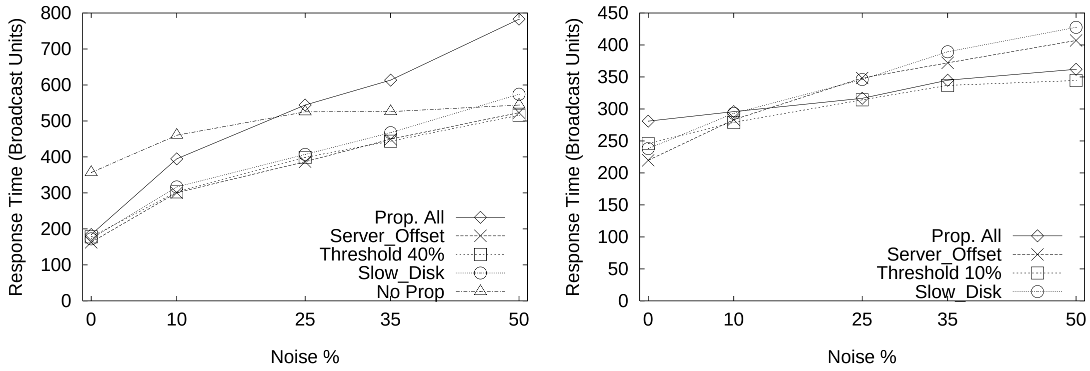
We study three variations on this theme. Each of these algorithms uses some static or dynamic information to filter the number of pages propagated. The first, called *Server_Offset*, propagates only the server's offset pages. These are the pages which are the hottest (for read) from the server's viewpoint. Recall that the server uses a probability distribution to generate the broadcast which is a cumulative function over the access patterns of all the clients in the system. When there is no *noise* in the system, the server's distribution is exactly the same as the client's distribution (and so are their offset pages). However, as *noise* increases the client's access distribution drifts away from the server broadcast and the overlap between the offset pages and the client's hot pages decreases, thereby, reducing the utility of the propagation list.

The next algorithm, called *Slow_Disk*, propagates updates only if they reside on the slowest disk. This is a simple strategy which requires no probability information. Its goal is to propagate the pages which, when invalidated, hurt the client most. Note that for the parameter settings under consideration, it propagates at least as many pages as the *Server_Offset* algorithm. The last algorithm, called *Threshold*, propagates a page only if its next arrival time on the broadcast is greater than a fixed threshold. The threshold is chosen as some percentage of the major cycle size. It uses a dynamic metric and thus, is potentially expensive to implement.

We performed a sensitivity analysis on the value of the propagation threshold. We found that for an *UpdateOffset* of zero, a propagation threshold of 10% of a major cycle provides the best response for most values of *noise*. For an *UpdateOffset* of 500, the best threshold value was around 40%. When the write and the read access coincide (*UpdateOffset*=0), the threshold is low because it is likely that an updated page will be accessed before it is broadcast again. We use these best cases in the graphs that follow. The results in this section consider the case when pages are propagated once a minor cycle (with invalidations occurring continuously).

Figure 6 shows the performance of all the algorithms for varying *noise* (on x-axis) for a cache size of 500 for the most update intensive case studied earlier (*UpdateThinkTime* = 2) and for two values of *UpdateOffset*. *Noise* represents variance in the access patterns of the clients. The key to good performance in steady-state, is to give priority for cache real-estate to those pages which are *hot* for the client locally but not globally popular and thus, not broadcast often. Other pages are accessed off the broadcast. In the read-only case, as long as the client has enough cache space for such locally important pages, it can maintain its steady-state performance even as the *noise* increases (up to a point). When there are updates, the cache never quite reaches steady-state since pages are being invalidated often. *Noise* only makes this worse, since with increasing variance among the client access patterns (as represented by *noise*), fewer of the clients' locally popular pages are also globally popular. The two graphs in Figure 6 show this decline in performance as *noise* increases.

We see that no algorithm performs uniformly well. The *Server_Offset* algorithm is by far the winner in most cases. It does quite well in Figure 6a in which *UpdateOffset*=500. *Server_Offset* is a conservative policy— it only propagates pages that are known to be hot from the server's point of view (the server's offset pages). Since the server distribution is generated as an average over all the client access patterns, it implies that the majority of clients want



(a) (b)

Figure 6: Sensitivity to Noise, Cache=500 pages, (a)*UpdateOffset*=500, (b)*UpdateOffset*=0

these pages. The best *Threshold* case does as well as *Server_Offset* with the *Slow_Disk* performing slightly poorer. The folly of aggressively propagating every updated page is apparent from the crossover of no propagate (i.e., prefetch alone) at around 25% *noise*.

Server_Offset is also a good bet for *UpdateOffset*=0 (Figure 6b) when the *noise* is low. As the *noise* increases beyond 10%, *Server_Offset* begins to do worse. With increasing noise, some of the pages that have high access for the client and high update no longer correspond to the server's offset pages. Thus, *Server_Offset* does not propagate them, and they do not come by fast enough. In fact, for *UpdateOffset*=0 as *noise* increases, propagating all pages performs very well. Since most of the updated pages are accessed by the client, the extra bandwidth is well spent. The best *Threshold* case (threshold of 10%) again performs the best when the *noise* is high. However, the inability to find a threshold value which does uniformly well over the complete simulation space makes *Threshold* a less than viable option in its current form. An algorithm which dynamically adjusts the threshold value can be expected to do well and is part of our future study. We do not show the performance of the no propagate case here because of its extremely poor performance (about 5 times worse than the others).

In general, *Server_Offset* is a good refinement to propagating everything. It does well except in cases in which the update and the read access coincide and in which the *noise* level is high. Note that even here the loss is minimal when compared to prefetch with no propagation. We would argue that high *noise* is not that interesting for us. In order to maintain a Zipf distribution at the server, there cannot be a wide variance in the client's behaviors. If the variance were high, the combined distribution at the server would tend more toward uniformity.

6 Related Work

As stated in the introduction, the basic idea of broadcasting information has been studied before ([Imie94a, Imie94b, Imie94c, Giff90, Wong88, Amma85]). Our work differs from these in that we consider multi-level disks and their relationship to cache management. Influence of updates on performance in dissemination-based environments has been studied in [Alon90, Bowe92, Barb94, Jing95, Wu96]. In [Alon90], the notion of *quasi-copies* was introduced where a replica was allowed to deviate from the original copy in a controlled fashion. In this environment, consistency could be defined on a per-client basis. Based on the constraints supplied, the client would be guaranteed to have the access to a value which was up-to date within the last t minutes of the most recent value or within $n\%$ of the current value (for scalar variables) and so on. The Datacycle Project [Bowe92] at Bellcore was intended to exploit high bandwidth, optical communication technology and employed custom VLSI data filters for performing associative searches in parallel on the broadcast data. The clients could execute queries or transactions locally and use the upstream network to send updates back to the server. The system provided consistency by ensuring that data items read within a single broadcast cycle were mutually consistent. For queries spanning over multiple cycles, the client used an optimistic protocol and the server broadcast a log of changes which was used to detect conflicts. This periodic consistency guarantee is similar to our once-a-major-cycle update technique.

[Barb94] addressed the problem of updates and consequently, stale data in a client's cache in a mobile setting where clients could be disconnected for long periods of time. Among other ideas, they proposed periodic broadcast of invalidations using signature based schemes. [Jing95] and [Wu96] are recent papers which improve upon the algorithms proposed in [Barb94]. These papers differ significantly from ours, however, because they assume a *pull-based* system where the client fetches data by making explicit requests to the server via the back channel; only the invalidations are disseminated. The techniques developed in this area, however, may be useful for supporting intermittent connectivity in the Broadcast Disk environment as well.

Finally, a discussion and comparison of the caching algorithms (and techniques to cope with updates) in client-server systems is available in [Fran96]. For similar surveys in other areas the reader is referred to [Levy90] (distributed file systems), [Nitz91] (distributed shared memory) and [Arch86] (multi-processor architectures).

7 Conclusions

In this paper, we have examined the way in which updates affect the performance of Broadcast Disks. As in our previous work, the design considerations divide into client-side and server-side issues. The server must decide how to modify its broadcast program in order to most effectively communicate updates to the client. The client must perform appropriate actions (e.g., prefetch) to bring its cache contents back to steady-state. For the server, we have experimented with invalidation and propagation lists. At the client, our experiments examined cache invalidation and prefetch.

Two different consistency models were studied — immediate and major-cycle boundary. We showed that for low to moderate update rates, it is possible to approach the performance of the read-only case. We also showed that in some cases prefetching is often good enough. We demonstrated that propagating a page is useful if the client makes use of the page before it would naturally get the page back by prefetch or direct access. Otherwise, the propagation wastes bandwidth. This is one of the fundamental tradeoffs in the design of dissemination-based systems with update.

A key observation from our experiments is that, in general, the Broadcast Disks model is robust in the presence of updates. We have shown that with some very simple enhancements it is possible to produce results that closely track the read-only case. Thus, all of our intuitions and results that were observed for the read-only case ([Acha95a], [Acha96]) are still valid in the update setting.

In the future, we intend to extend this work by considering some of the alternative models of consistency that were listed in Section 2. For example, introducing a mechanism to support ACID transactions into our design and exploring techniques for query processing on the broadcast medium would be interesting extensions.

We also plan to relax our assumption of no backchannel use to see if small amounts of feedback from the client to the server can improve performance. Intuitively, this seems to make sense since the server can occasionally design a bad broadcast program because it does not have enough information regarding the current access patterns of the client or the current state of the client cache. Also, the use of the back channel would allow clients to make updates to their local copies and propagate the changes to the server.

Another important study involves comparing various algorithms on the basis of the listening requirements of the clients. Requiring that the clients monitor the broadcast continually or for extended periods of time may have deleterious effects on performance especially when the broadcast is through a very high speed channel (e.g., satellite) or if external constraints such as battery life [Imie94a] are an issue.

References

[Acha95a] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments", *Proc. ACM SIGMOD Conf.*, San Jose, CA, May, 1995.

[Acha95b] S. Acharya, M. Franklin, S. Zdonik, "Dissemination-based Data Delivery Using Broadcast Disks", *IEEE Personal Communications*, 2(6), December, 1995.

[Acha96] S. Acharya, M. Franklin, S. Zdonik, "Prefetching from a Broadcast Disk", *12th International Conference on Data Engineering*, New Orleans, LA, February, 1996.

[Alon90] R. Alonso, D. Barbara, H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System", *ACM TODS*, 15(3), September, 1990.

[Amma85] M. Ammar, J. Wong, "The Design of Teletext Broadcast Cycles", *Perf. Evaluation*, 5 (1985).

[Arch86] J. Archibald, J. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM TOCS*, 4(4), November, 1986.

[Barb94] D. Barbara, T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environments", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May, 1994.

[Bere95] H. Berenson, P. Bernstein, J. Gray, J. Melton, B. O'Neil, P. O'Neil, "A Critique of ANSI SQL Isolation Levels", *Proc. ACM SIGMOD Conf.*, San Jose, CA, June, 1995.

[Bowe92] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, A. Weinrib, "The Datacycle Architecture", *CACM*, 35(12), December, 1992.

[Dan90] A. Dan, D. M. Dias, P. Yu, "The Effect of Skewed Access on Buffer Hits and Data Contention in a Data Sharing Environment", *Proc. 16th VLDB Conf.*, Aug., 1990.

[Fran92] M. Franklin, M. Carey, "Client-Server Caching Revisited", in *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August, 1992, (Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds., Morgan Kaufmann, San Mateo, CA, 1994).

[Fran96] M. Franklin, *Client Data Caching: A Foundation for High Performance Object Database Systems*, Kluwer Academic Publishers, Boston, MA, February, 1996.

[Giff90] D. Gifford, "Polychannel Systems for Mass Digital Communication", CACM, 33(2), February, 1990.

[Herm87] G. Herman, G. Gopal, K. Lee, A. Weinrib, "The Datacycle Architecture for Very High Throughput Database Systems", *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May, 1987.

[Imie94a] T. Imielinski, B. Badrinath, "Mobile Wireless Computing: Challenges in Data Management", *CACM*, 37(10), October, 1994.

[Imie94b] T. Imielinski, S. Viswanathan, B. Badrinath, "Energy Efficient Indexing on Air", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May, 1994.

[Imie94c] T. Imielinski, S. Viswanathan, B. Badrinath, "Power Efficient Filtering of Data on Air", *Proc. Conf on EDBT*, 1994.

[Inte96] Intel Corp., "CNN-at-work", (description of product that supports continuous broadcast and filtering of CNN news stories over a departmental ethernet), `http://www.intel.com/comm-net/cnn_work/index.html`

[Jing95] J. Jing, O. Bukhres, A. Elmargarmid, R. Alonso" "Bit-Sequences: A New Cache Invalidation Method in Mobile Environments", *Technical Report CSD-TR-94-074*, Computer Sciences Department, Purdue University, revised May 1995.

[Katz94] R. Katz, "Adaption and Mobility in Wireless Information Systems", *IEEE Personal Communications*, 1st Quarter, 1994.

[Knut81] D. Knuth, *The Art of Computer Programming, Vol II*, Addison Wesley, 1981.

[Kort95] H. Korth, "The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept", *Proc. 21th VLDB Conf.*, Zurich, September, 1995.

[Levy90] E. Levy, A. Silbershatz, "Distributed File Systems: Concepts and Examples", *ACM Computing Surveys*, 22(4), December, 1990.

[Nitz91] B. Nitzberg, V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, 24(8), August, 1991.

[Oki93] B. Oki, M. Pfluegl, A. Siegel, D. Skeen, "The Information Bus - An Architecture for Extensible Distributed Systems", *Proc. 14th SOSP*, Ashville, NC, December, 1993.

[Shek94] S. Shekhar, D. Liu, "Genesis and Advanced Traveler Information Systems (ATIS): Killer Applications for Mobile Computing", *MOBIDATA Wkshp*, Rutgers Univ., NJ, 1994.

[Wong88] J. Wong, "Broadcast Delivery", *Proceedings of the IEEE*, 76(12), December, 1988.

[Wu96] , K. Wu, P. S. Yu, M. Chen, "Energy-Efficient Caching for Wirless Mobile Computing", *Proc. of ICDE*, New Orleans, Feb. 1996.

[Zdon94] S. Zdonik, M. Franklin, R. Alonso, S. Acharya, "Are 'Disks in the Air' Just Pie in the Sky?", *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December, 1994.