

Data Staging for On-Demand Broadcast

Demet Aksoy

University of California, Davis
aksoy@cs.ucdavis.edu

Michael J. Franklin

Univ. of California, Berkeley
franklin@cs.berkeley.edu

Stan Zdonik

Brown University
sbz@cs.brown.edu

Abstract

Due to the increasing deployment of broadband services on infrastructures that are inherently broadcast capable, wide-area data broadcast is becoming a highly attractive data delivery alternative for large client populations. Within the network community, there has been significant effort towards on-demand data broadcast server design. However, these studies have mostly been focused on developing on-line scheduling algorithms for optimizing bandwidth allocation based on the assumption that all data items are readily available in server's main memory. This assumption simply ignores data management issues that arise when data items need to be fetched from secondary storage or from remote sites before they can be broadcast. Such *data staging* concerns, if ignored, can result in significant performance degradation of the data broadcast server's performance. In this paper, we propose three data staging solutions: opportunistic scheduling, server caching, and prefetching, that are closely integrated with the broadcast scheduling algorithm. We then use a data broadcasting testbed based on IP-Multicast to examine the performance of these various solutions. Our results show that the hints provided by the scheduling algorithm can be used to dramatically enhance the performance of a large-scale on-demand broadcast system.

1 Introduction

The dramatic growth of the Internet has brought about the deployment of a new type of Internet infrastructure optimized for supporting high-speed storage and delivery of data for huge numbers of users. Examples include Content Delivery Networks, Cooperative Web Caches, and Wide-area Distributed Storage Networks. Of necessity, these systems have been built using private networks and application-level networking protocols. This is because the generic Internet infrastructure and protocols simply do not provide the required capacity and functionality (such as intelligent routing, multicast, broadcast, performance guarantees, etc.) in a universally accepted way.

These new services are typically implemented using an "overlay network" in which a privately maintained, small-diameter network consisting of high-speed communication channels, application-level routers, and caches reduces the need to use the public Internet backbone. The overlay network is used to move data closer to the clients that need it, thereby limiting the need to access data from distant servers. Prominent examples include the offerings of such companies as Akamai [Aka], Fast Forward Networks [Net] (now part of Inktomi), Edgix [Edg], and Cidera [Cid].

There is a great deal of flexibility in the types of communication these systems can use, since they use their own communication channels and application-level protocols. For example, services can be deployed

using broadcast or multicast, satellite communication, and other techniques that are not widely available on the public Internet. Broadcast is particularly appealing in a content delivery scenario because of its inherent scalability [AF99]. Unlike unicasting, where an item must be sent once for each outstanding request, with broadcast-based delivery a single transmission of an item can satisfy *all* outstanding requests for the item.

While it is tempting to look at such solutions merely as communications pipes, the heart of what they are trying to do is *data management*. That is, they are trying to intelligently manage and optimize the location and flow of large amounts of data. Thus, traditional data management techniques such as caching and prefetching play a key role in these systems. In this paper, we focus on data management issues for a particular type of broadcast system, namely an *on-demand broadcast* system.

1.1 On-Demand Broadcast

Data broadcasting systems can be distinguished according to whether they are based on a *push* model or a *pull* model [FZ98]. Using push, the data items are sent out to the clients without explicit requests for such items. In contrast, with a pull-based model, data items are broadcast by a server in response to requests received from clients. We refer to such an arrangement as *on-demand data broadcast*.

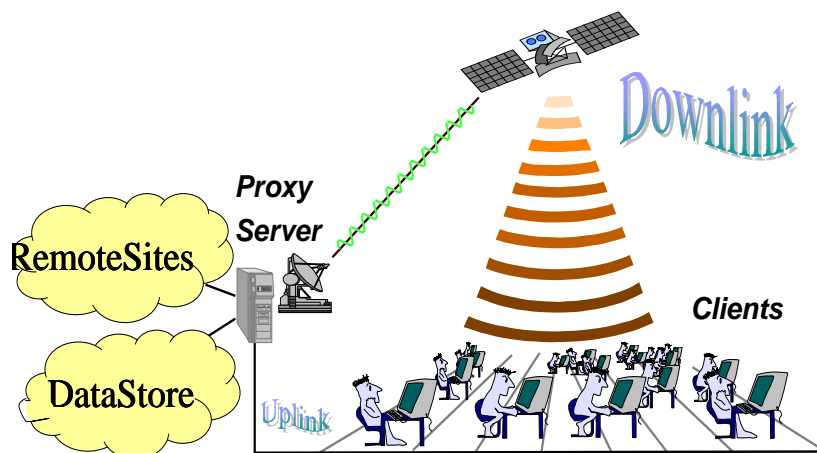


Figure 1: Example Data Broadcasting Scenario

An example of on-demand data broadcast environment is shown in Figure 1. In this scenario clients send requests for data items to a server via an independent uplink channel. The server receives these requests and, when necessary, places them in a service queue. Based on the received requests the server selects data items to broadcast, and sends them to the clients using the downlink channel. Clients monitor the broadcast to receive the items they are interested in. Broadcast can be used to send data directly to clients who are Internet browsers (e.g., in a system such as DirectPC [Dir96], Starband [Sta00], AlohaNet [Alo00] etc.) or to proxy caches from which the clients will retrieve the data using standard protocols. Such systems could conceivably be used by millions of clients and could provide access to millions of data items. Figure 1 depicts an on-demand broadcast environment similar to what could be provided using a Direct Broadcast Satellite infrastructure such as Hughes Network System’s DirecPC [Dir96]. In this case, the uplink is a primarily terrestrial, wired network such as the Internet, while the downlink is a high-bandwidth satellite link. Other technologies are of course, also possible.

1.2 Broadcast Scheduling and Data Staging

A key design consideration in the development of an on-demand data broadcast server is the *scheduling algorithm* used to select items to be broadcast. Such an algorithm aims to choose at each instance, the most beneficial data item to be broadcast based on the unfulfilled requests that have been received from clients. There has been significant work on the development of on-line scheduling algorithms for data broadcast (e.g., [DAW86, Won88, VH96, ST97b, AF99]). A key concern in the face of high downlink bandwidths¹ has been developing low overhead algorithms so that the available broadcast bandwidth can be fully utilized.

In practice, however, data may not be available immediately when required by the scheduler. As a result, processing efficiency may not be sufficient to guarantee the full utilization of the downlink channel. For cases where data items must be retrieved and brought into the server’s main memory before they can be broadcast, significant performance degradation can be experienced due to the additional latency prior to broadcast.

In an early study, Dykeman et al. [DW88] addressed this issue. They studied disk scheduling algorithms as well as server cache replacement policies in order to improve performance in the face of data retrieval latency. On-demand broadcast work done since that paper, however, did not address the issues that arise when the requested data items are not immediately available for broadcast. To address this problem, we designed a set of mechanisms that coordinate the broadcast scheduling with the location and retrieval of the data items to be broadcast. We refer to this integrated functionality as *data staging*.

Independent of our work, Triantafillou et al. [THP01] have also recently reported on the potential performance degradation that can arise when data needs to be brought into main memory before it can be broadcast, and have proposed disk scheduling and caching algorithms to improve the performance of an on-demand data server. While validating our concern with the data staging problem, the approach taken by Triantafillou et al., differs significantly from ours. In particular, our work has been focused on developing an integrated data staging mechanism that includes modified broadcast scheduling, cache management, and prefetching all within the context of the RxW broadcast scheduling algorithm. These emphases result in a different set of mechanisms, as discussed in Section 8.

1.3 Data Staging Solutions

In this paper, we attack the data staging problem using three complementary approaches. These approaches are designed to work integrated with each other as well as with the scheduling algorithm. The three data staging approaches we have developed are summarized in the following:

1. *Increasing bandwidth utilization:* It is not possible to fully exploit the high bandwidth of the downlink channel if the server stalls between successive broadcasts. In cases where the most beneficial data item to broadcast is not readily available, it is, therefore, unacceptable to wait idle until this item is fetched. In such cases another data item, can be broadcast while waiting for the most beneficial data

¹The downlink bandwidth is usually much higher than that of the uplink in order to match the asymmetry in data flow, i.e., a client request is typically only a few bytes while the requested data item can be quite large.

item to be retrieved. Intuitively, this is expected to be straight-forward approach of selecting the *next best page*. However, we will show that for performance purposes, this decision should be based on the overall bandwidth allocation rather than the scheduling heuristics. Previous work has shown that the optimal allocation of broadcast bandwidth to data items is proportional to the ratio of the *square roots* of their access probability [DAW86]. We exploit this property by sometimes broadcasting sub-optimal, but memory resident data items, while the scheduled items are being brought into the server's cache. We refer to this technique as *Opportunistic Scheduling*.

2. *Decreasing the need to fetch an item*: One obvious way to reduce the need for fetching data items is to make the best use of the available memory space on the server. The key to successful caching for on-demand broadcast servers is to retain those items that are most likely to be scheduled. For this purpose, we exploit hints maintained by the scheduling algorithm in order to differentiate between hot (popular) and cold (not-so-popular) items to make caching decisions. We refer to this technique as *Hint-based Cache Management*.
3. *Decreasing the fetch latency*: As in any information system architecture, access latency can be reduced by obtaining items from slow or remote locations *before* they are needed. In the broadcast data staging context this translates to predicting which items will be broadcast in the near future and bringing them into the cache before they are actually scheduled for broadcast. We refer to this technique as *Prefetching*. To this end, we are able to exploit hints provided by the scheduling algorithm to identify non-cache-resident items that have a high probability of being broadcast in the near future.

These three techniques all have analogs in more traditional data and proxy servers. However, as we will see, the use of broadcast for delivering responses to user requests results in significant changes to their implementation and in the inherent tradeoffs that arise when employing them. In the following, we first describe the general architecture of an on-demand broadcast server and then describe the mechanisms we propose to perform data staging in detail. We then describe our data broadcast prototype testbed which has been implemented using IP-Multicast on a cluster of Pentium-based workstations running Windows NT. We have used this testbed to perform a detailed set of performance studies to evaluate the effectiveness of the data staging techniques. The results of these studies are reported in Sections 5, 6 and 7. We describe related work in Section 8 and present our conclusions in Section 9.

2 Broadcast Data Staging: Motivation, Context, and Techniques

In this Section we introduce the design of our on-demand broadcast-based server that we have implemented using a prototype testbed. In particular, we describe our approach to broadcast scheduling for data that may not be memory resident, and our approach to managing the server cache.

2.1 Motivation and Context

Like any pull-based server, an on-demand broadcast server accepts requests for data items from clients and replies to those requests. If the system is lightly loaded, i.e., with a few or a single client, the job of the

server is quite simple: accept a request, wait until the item is located (assuming that meanwhile there are no other outstanding requests), and then transmit it. In such a lightly-loaded system, there is little or no benefit to be obtained from using a broadcast delivery mechanism.

In contrast, our interest is in large-scale dissemination systems that are geared to support high numbers of users — that can be hundreds of thousands or more. Thus, we expect that for substantial periods of time requests will be arriving at a much faster rate than can be served individually. Assuming that there is significant overlap in the items of interest to the users, broadcasting has the advantage that a single transmission of an item will satisfy *all* of the clients that are waiting for the item at that time. In this situation, the broadcast bandwidth becomes a shared, and therefore limited, resource that must be used efficiently and effectively if client requests are to be handled in a timely fashion.

The shared nature of the broadcast channel raises an interesting *scheduling* problem. Given a list of outstanding requests for data items, the server must choose which item to broadcast at any given instant. One obvious heuristic would be to choose the item with the largest number of outstanding requests. Such an approach, however, results in poor overall performance because it results in unacceptably high response times for not-so-popular data items [AF98]. An on-demand broadcast-based data server must balance the treatment of both hot (popular) and cold (less popular) items. The development of such schedulers in the absence of the data staging problem has been the focus of much previous work, and solutions that make effective use of the broadcast bandwidth have been developed [DAW86, Won88, VH96, ST97b, AF99].

For a large regional or national-scale information service the *implicit* assumption of having all data items immediately available for broadcast is inappropriate for two reasons: First, the sheer size of the data available and the highly-skewed nature of accesses to it (see [BCF⁺99] for access distributions for WWW proxies) make a main-memory-only system both technically infeasible (for the near term) and economically wasteful (for the foreseeable future). Second, even if it were possible to cache the entire data set in memory at the server, the costs of keeping the cached copies up-to-date would be prohibitive. Thus, in a large-scale system we expect the majority of the data to reside in locations with significantly higher latency than the server's memory, such as local secondary and tertiary storage, as well as at remote sites around the Internet. If simply ignored, this added latency would result in huge amounts of unused broadcast bandwidth, rendering the on-demand broadcast server unusable. In this study, we maximize the effective bandwidth utilization in order to improve the responsiveness of a large-scale on-demand broadcast server.

2.2 Server Overview: Broadcast Scheduling with Non-Resident Data

We have developed an on-demand broadcast server that can provide very good performance even when much of the data to be broadcast is not memory-resident at the server. The server integrates data staging solutions with the scheduling algorithm. Obviously, a primary concern is to ensure that none of the precious broadcast bandwidth goes unused due to the latency incurred to obtain a scheduled item from slow storage or from a remote site. To accomplish this, the scheduler must be non-blocking. That is, if the scheduler chooses a non-resident item to be broadcast, it should initiate an *asynchronous* request for that item and continue.

Practical considerations, however, limit the number of outstanding asynchronous requests a system can

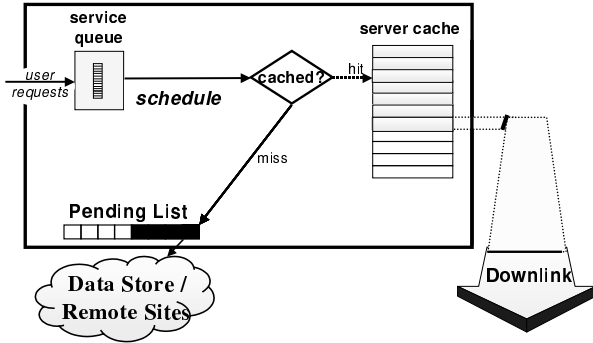


Figure 2: Normal Scheduling

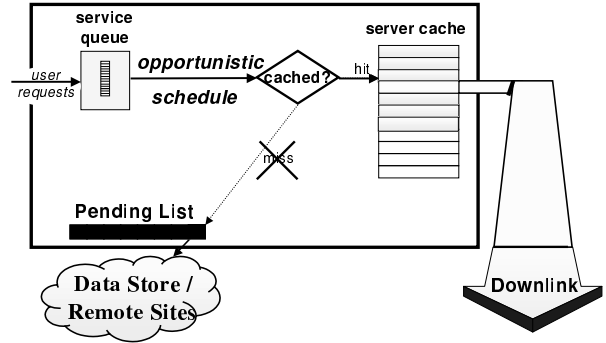


Figure 3: Opportunistic Scheduling

tolerate. For example, in our Windows NT-based implementation, we found that beyond a (configuration-specific) point, allowing too many outstanding I/O requests became detrimental to performance. Data retrieval rate can not be increased any further after a point and therefore threading overhead becomes dominant. For this reason, our server respects a preset limit on the number of fetch requests for data that can be outstanding at a given time. It then uses two modes of scheduling: 1) a “normal” mode, when the system is below this limit, and 2) an “opportunistic” mode, when it is not. The server alternates between these two modes. Opportunistic mode ensures that a data item be broadcast and, therefore, downlink bandwidth utilization can be significantly improved when it is applied. We will explain these two modes in greater detail after introducing the server architecture.

Figure 2 depicts the server architecture that we have used. As shown in the figure, the server maintains two structures for managing requests for data items: the *Service Queue* and the *Pending List*. The Service Queue contains information about outstanding client requests. Since the broadcast of an item satisfies all outstanding requests for that item, a single entry in the Service Queue is used for each unique item that has outstanding requests. This entry typically contains information about the number and arrival time(s) of the requests which is used in making scheduling decisions.² Thus, when a request is received at the server, the Service Queue is checked to see if an entry already exists for the requested item, and if so, the information in that entry is updated correspondingly, otherwise a new entry is created and added to the Service Queue. When an item is chosen to be broadcast, its entry is removed from the service queue. The Pending List is used to keep track of items for which an asynchronous fetch request is pending. The limit on the number of outstanding asynchronous requests is enforced by bounding the size of the Pending List.

The scheduler loops continuously. The loop begins by checking for the completion of any asynchronous fetches in the Pending List. Any such items that have arrived in the server’s memory are broadcast in the order they were received, and their entries are removed from the Pending List. Note that the order in which items are received is not necessarily the same as the order in which they were requested. After all such items have been processed (if any), the scheduler is run to select a data item to broadcast among those that

²For generality we delay the discussion of the specific queue structure and scheduling algorithm used until Section 3.

have an outstanding request in the service queue: If the Pending List is not full (i.e., the system is below the specified limit on outstanding asynchronous fetch requests), then the scheduler is run in *Normal* mode. Otherwise, it is run in *Opportunistic* mode. These two modes are described in the following sections.

2.2.1 Normal Scheduling Mode

In the *normal* scheduling mode, the server searches the Service Queue to select an item to broadcast. When the scheduling decision is made and an item is chosen for broadcast, a look-up is done in the local cache. If the scheduled item is present (i.e., a cache “hit”), then the item is handed over to the network controller to be transmitted over the downlink in the next available broadcast slot. If, however, the chosen item is not present (i.e., a cache miss) then an entry for the item is made in the *Pending List* (see Figure 2), and a request is sent (asynchronously) to the device or remote site containing the item. In this latter case, the server continues to the next iteration of the processing loop without broadcasting an item. In either case (cache hit or miss) the entry for a selected item is removed from the Service Queue once a scheduling decision is made. Note that as long as an item has an entry in the Pending List, the server makes sure not to create a new service queue entry for any incoming requests for the item. This is because at that point the item is already slated to be broadcast when the item eventually arrives. That broadcast will satisfy all requests for the item regardless of whether or not they have been entered into the Service Queue.

2.2.2 Opportunistic Scheduling Mode

The scheduler operates in Opportunistic Scheduling (OS) mode, when the Pending List is full. The goal of opportunistic scheduling is to ensure that the limit on outstanding asynchronous requests is not exceeded. This is accomplished by restricting the scheduler to choose only cache-resident pages to be broadcast. There are two potential dangers with such a restriction. First, it naturally disrupts the heuristics used by the scheduler in choosing the next broadcast item. If the scheduler makes too many poor choices, then the effectiveness of the broadcast schedule (and hence, the performance of the system) could be severely degraded. Second, if the process of finding good cache-resident items to broadcast adds too much overhead to the search process, then it is possible that broadcast bandwidth could go unused while waiting for the scheduler to complete its opportunistic scheduling decision. The challenge in designing the search algorithm for Opportunistic Scheduling is to strike a balance between efficiency and effectiveness.

We propose two approaches to broadcast scheduling in OS mode. Both approaches enforce two requirements on items chosen for broadcast: 1) they must have at least one outstanding request (i.e., they have a corresponding entry in the Service Queue), and 2) they must be cache-resident. The approaches are:

- *Best Available* - runs the normal search algorithm over the Service Queue but considers only the items that are known to be cache-resident.
- *Scan Cache* - foregoes the normal search algorithm, and instead, simply scans through the cache and chooses the first item with outstanding requests that it encounters.

While these approaches can be integrated with many different “normal” scheduling algorithms, their spe-

cific behavior is dependent on the details of the scheduler. Thus, we postpone further discussion of these approaches until after our scheduling algorithm of choice is described in Section 3.

2.3 Cache Management and Prefetching

As can be seen in the description above, the absence of scheduled items from the cache disrupts the functioning of the broadcast scheduling heuristics. Therefore, we have investigated two techniques for improving the hit rate at the server cache. The first technique is a specialized cache replacement policy that is used instead of the traditional Least Recently Used Policy. Our policy uses hints obtained from the broadcast scheduler to distinguish between hot and cold items, thereby directly improving the cache hit rate. This policy called “LH”, for Love-Hate hints, is described and studied in Section 6.

The second technique we use to improve the cache hit rate is *prefetching*. Our use of prefetching is complementary to the LH cache management approach. LH aims at keeping hot items (i.e., items that are likely to be broadcast *again* in the near future) in cache. In contrast, prefetching aims to bring into the cache, pages that have not been broadcast recently but that are likely to be broadcast soon. Like LH cache replacement, prefetching also uses hints obtained from the broadcast scheduler.

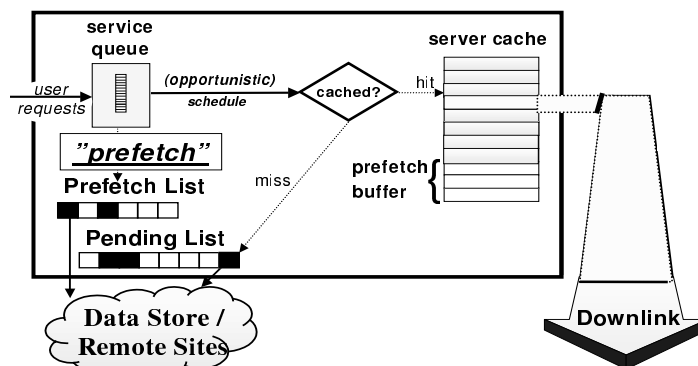


Figure 4: Broadcast Server With Prefetch

Figure 4 shows the server architecture augmented with the prefetching mechanism. In the server’s cache, a set of frames is reserved for holding prefetched items. The prefetch process snoops the service queue to identify candidates that are likely to be broadcast soon and tries to ensure that they are in memory when they are eventually chosen for broadcast. The maximum allowable number of such items is a configuration parameter referred to as the prefetch window. Pages within the prefetch window are guaranteed to be either in the cache or to be in the process of being prefetched at any time during the processing loop. Similar to the handling of fetches for cache misses, a prefetch request for a candidate item is sent asynchronously to the location of that item.

When a prefetched item arrives, it is pinned in the cache until it is broadcast, so that the normal cache replacement policy does not negate the prefetch by choosing it eviction victim. Remember that we chose to prefetch cold pages those are already in the service queue therefore these pages are destined to be scheduled

for broadcast at some point in time. Once such a page is broadcast, it is unpinned, and the choice of keeping it in the cache is left up to the regular cache replacement policy. If an item is scheduled before it has arrived in the cache (i.e., a prefetch request is pending for it), then when it arrives, the item is treated as if it had been fetched due to a cache miss. That is, it is no longer considered as a prefetch candidate, and it is broadcast the next time the server checks for arrivals of fetched items. In both cases, the prefetch process searches the Service Queue to identify a new candidate and initiate a prefetch request for it if necessary.

The three data staging techniques: *opportunistic scheduling*, *hint-based cache management*, and *prefetching*, work together in order to increase bandwidth utilization, decrease the need to fetch data items and decrease the latency of retrieving a data item when possible. These techniques must be closely integrated with the scheduling algorithm. We discuss one such algorithm in the next section.

3 The RxW Scheduling Algorithm

In this section, we present a brief sketch of the *RxW* broadcast scheduling algorithm [AF99], which serves as the basis for our integrated broadcast scheduling and data staging techniques. *RxW* has been shown to make fast scheduling decisions while providing high-quality schedules over a wide range of workloads. Intuitively, *RxW* schedules a data item either because it has many outstanding requests or because it has at least one outstanding request that has waited for a long time for that item. RxW was developed for scheduling the broadcast of fixed-length items such as disk or database pages. Scheduling approaches for variable-length data, particularly for multimedia applications, have also been proposed [VH97, ST97a, AM98]. In our server we have implemented the simpler fixed-length approach. This study uses that implementation to focus on the interaction of data staging techniques with the scheduler.

RxW maintains a service queue which contains a single entry for each requested page. Each entry in the service queue contains two values: 1) the number of outstanding request(s) for the page (R); and 2) the arrival time of the oldest of those requests, which is used to compute the waiting time of the oldest outstanding request for that page (W). *RxW* chooses the item to broadcast according to its $R \times W$ value. Ideally, the item with the highest $R \times W$ value is selected.

Literally searching for the item with the highest $R \times W$ value could be quite time consuming which risks wasting downstream bandwidth. Thus, reasonable approximate algorithms have been developed. Our server uses one called *RxW.alpha*. The α refers to a tunable parameter that can be set to control the desired level of approximation.

The server maintains two sorted lists threaded through the service queue: one based on the number of outstanding requests (referred to as the *R-list*) and the other based on the waiting time of the oldest request for that page (referred to as the *W-list*). The search for the page to broadcast starts from the entry at the top of the *R-list* (the page with the most outstanding requests). Its $R \times W$ value is computed and recorded as the current maximum. Next, the entry at the top of the *W-list* (the entry with the oldest outstanding request) is examined and its RxW value is compared to the current maximum. The algorithm then keeps alternating between the two lists; it selects the *first* page whose $R \times W$ value is greater than or equal to α times the current *threshold*. The *threshold* is computed as the running average of the RxW value of the last

page broadcast and the previous *threshold*. If there is no page with $R \times W$ value greater than or equal to the current value of $threshold \times \alpha$, the page with the highest $R \times W$ value is selected for broadcast. After each broadcast decision, the service queue entry for the selected page is removed from the service queue, and the *threshold* is updated accordingly.

The setting of the α parameter determines the performance tradeoffs among schedule quality and scheduling overhead. The smaller the value of the α parameter, the fewer entries are scanned and thus the lower is the overhead with a correspondingly lower accuracy. When α is set to 0 the algorithm examines only two entries (the one at the top of the *R-list* and the one on the top of *W-list* to make the scheduling decision. At the other extreme when *alpha* is set to ∞ , the algorithm stops when the selected page is guaranteed to have the maximal $R \times W$ value.

4 Experimental Environment

In order to study the data staging solutions we propose, we have implemented a prototype on-demand broadcast testbed. Using this testbed we can incorporate the actual overhead of our data staging solutions in our performance results. In this section we describe the testbed, as well as the workload and metrics used in the performance evaluation.

4.1 Prototype

Our on-demand broadcast testbed has been implemented on a cluster of pentium-based machines running Windows NT 4.0. Each machine in the cluster has two network cards, so the testbed environment consists of two separate Ethernet networks: a 10 Mb/sec network used as the uplink (i.e., for requests) and a 100 Mb/sec network used as a broadcast downlink. The uplink employs TCP for sending requests to the server. The downlink employs UDP (Unreliable Datagram Protocol) for *multicasting* the data to all of the workstations in the cluster using the IP-multicast support provided with Windows NT 4.0. During experiments, the testbed is isolated from all external networks to avoid external interference.

The server used for these experiments is a dual-processor machine with two 400MHz Pentium II CPUs and 256MB of memory. The server has two main responsibilities: request processing (queuing new requests), and broadcast management (making scheduling decisions and broadcasting pages). To ensure that the request arrival rate is fixed across all algorithms, the request processing thread is given top priority. The server has two 9.1GB fast wide SCSI disks. One of these was reserved exclusively for use as secondary store for data items. File system buffering was disabled during the experiments so as not to interfere with the data staging measurements.

4.2 Workload, Metrics, and Measurements

For the experiments reported here, we used a workload generator running on a dedicated machine. The generator produces item requests and sends them individually over the uplink. Requests are generated

according to a Zipf distribution [Knu81] with θ set to 1. Recall that the Zipf distribution produces skewed access patterns according to:

$$p_i = \frac{1}{i^\theta \sum_{j=1}^N \frac{1}{j^\theta}}$$

where p_i is the probability of accessing page i , N is the size of the database and θ is the skewness parameter. Unless noted otherwise, the results reported here were obtained using a request arrival rate of 1000 requests/sec. The database used in the experiments consists of 10000 16K pages. Note that this database size is chosen merely to ensure that the complete database can fit in the available amount of memory so that we can evaluate the relative performance for a limited cache. Since the file system buffering is disabled throughout the experiments and this is purely a storage hierarchy issue (no concurrency control is involved), we expect the results to scale as memory and data are added in the same proportion. The cache size is varied between 5% and 100% of the database. We focus on the results for upto 40% of the database where most of the differences are observed. Where noted in the sections that follow, we sometimes add a synthetic delay to the retrieval of items from disk in order to model higher-latency situations.

The main metric used in the performance study is *average waiting time*. In order to reduce overhead of individual measurements, we measure *average waiting time* after equilibrium is reached at the server by applying Little’s Law [Tri82] to the logical service queue length (i.e., the number of individual *requests* waiting in the queue).³ In the experiments, we first warm up the server cache and run the system until equilibrium is reached before taking measurements. Equilibrium occurs when the number of outstanding requests in the system stabilizes, i.e., when the request satisfaction rate converges to the request arrival rate. Thus, the data points in the experiments that follow were collected over runs of hundreds of thousands or more requests. In addition to average waiting time, where necessary, we also report other metrics such as the *data broadcast rate*, which is the rate at which data can be broadcast over the downlink channel, *bandwidth allocation error*, which is the deviation from the optimal allocation, and *weighted server hit rate*, which is the hit rate at the server cache scaled by the number of outstanding requests effected.

As described in Section 3, the scheduling algorithm used in these experiments is *RxW*. We have observed that in our experimental testbed configuration, *RxW.90* provides a good trade-off between scheduling overhead (i.e., the time it takes to make a scheduling decision) and scheduling quality (i.e., closeness to the optimal bandwidth allocation). Therefore, in our experiments, we use $\alpha = 0.9$ (referred to as *RxW.90* in [AF99]) in order to study our data staging solutions. It should be noted, however, that we have tested our solutions using the full *RxW* algorithm and its approximations with different α values. Although the specific behavior of the various data staging approaches varies somewhat for different approximation settings, the trends described in this paper hold for all cases tested.

There are two other important parameters used in the server: 1) the limit on the number of asynchronous fetch requests allowed to be outstanding at any time, and 2) the *prfWindow* which dictates how many pages the prefetcher will consider as candidates. In the experiments that follow, the asynchronous fetch limit is set at 100 items. We also report sensitivity results for this value. The *prfWindow* ranges from 75 items to 500 items as discussed in Section 7.1. Table 1 summarizes the system parameters used in the experiments. Given this background we now examine the data staging approaches we have proposed in greater detail and

³Recall that the length of the physical queue maintained at the server is proportional to the number of items rather than the number of requests so it is much smaller than the logical queue used here.

present the experimental results we have obtained using the prototype implementation.

| Symbol | Description | Value | Unit |
|---------------|---------------------------------------|--------------|---------|
| θ | Request Pattern Skewness (Zipf) | 1.0 | - |
| $dbSize$ | Database Size | 10000 | pages |
| $pageSize$ | Database Page Size | 16 | Kbytes |
| $requestRate$ | Request Arrival Rate | 1000 | req/sec |
| $cacheSize$ | Cache Size | 5-40 | percent |
| α | Approximation Parameter of RxW | 0.9 | - |
| $depth$ | Max number of outstanding asynch. I/O | 100 | - |
| $prfWindow$ | Maximum number of pages to prefetch | 75-500 | - |

Table 1: Workload and Prototype Parameters

5 Opportunistic Scheduling

As described in Section 2, Opportunistic Scheduling (OS) is an alternative scheduling mode that is used when the server has reached its limit on outstanding asynchronous requests. In OS mode the scheduling algorithm is restricted so that it selects only cache-resident pages for broadcast. In this section, we first describe the two OS approaches in the context of the RxW scheduling algorithm. We then describe a set of experiments to compare the performance of these approaches.

Before examining the individual approaches however, it is important to emphasize that both OS approaches are *extremely* effective in solving the problems caused by non-resident data items. Thus, in this section and the subsequent ones, it is important to keep in mind that, although we might be able to gain further improvements, we are tuning a basic technique that is already very effective.

5.1 Algorithms

Both OS approaches require that the scheduler be cognizant of which items are cache-resident. The first opportunistic scheduling algorithm we consider is a simple modification of RxW 's search of the Service Queue; the second approach does not use the RxW search algorithm. The two approaches are:

Best Available (OS-BA) - In this approach, the Service Queue entries are augmented with a bit that indicates whether or not the associated item is currently cache-resident. This bit is set when a page is brought into cache and reset when the page is replaced in the cache. The RxW scheduling algorithm is run over this queue just as described in Section 3, except that all non-resident pages are ignored. Thus, OS-BA attempts to find the *best available* (memory-resident) page according to the search criteria of the RxW scheduling algorithm.

Figure 5 shows an example of this algorithm. In the figure, the shaded Service Queue entries represent pages that are currently not in the cache. Assume that the *threshold* (i.e., the running average of RxW values) is 500 due to past history. Because we are using $RxW.90$, this results in a stopping condition of 450

(500×0.90). The search starts on the *R-list* and considers the first cache-resident page it encounters (page *c* in this case). The *RxW* value of page *s* is 90, which does not qualify. It then switches to the *W-list* where it first examines page *z*. The search continues until page *n* is examined. This page has an *RxW* value of 470, which is greater than the stopping condition, so the search halts and page *n* is chosen. In contrast, during normal scheduling page *a* would have been chosen since its *RxW* value (1100) is larger than the stopping condition (450).

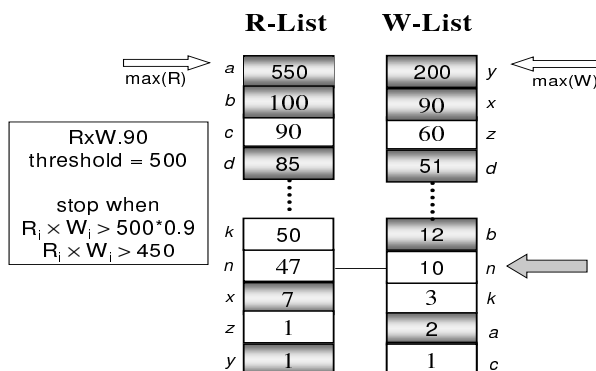


Figure 5: Modifying Scheduling

Scan Cache (OS-SC) - The second opportunistic approach completely foregoes the use of the *RxW* algorithm during opportunistic scheduling. Since during OS only cache resident pages can be scheduled, this algorithm directly searches the cache, choosing for broadcast the first page it encounters that has at least one outstanding request. The key question for OS-SC is where to start the scan of the cache. We tried several alternatives and found (much to our surprise) that the best performance was obtained when the scan proceeded up from the bottom of the LRU queue (as maintained by the buffer manager). The reason for this behavior is explained in the following section. For now, simply note that in the example of Figure 5, OS-SC is likely to schedule page *z* for broadcast. This is because, page *z* has been accessed only once sometime before 60 units ago. Of the three pages at the top of the *W-List*, *z* is the only memory-resident one, and the LRU queue would likely have page *z* close to the bottom.

5.2 Opportunistic Scheduling: Evaluation

In this section, we describe the results of an experimental comparison of the two opportunistic scheduling approaches. In this experiment, the disk is used as the backing-store for non-resident pages, and only the actual disk latencies are incorporated into the study. The size of the Pending List is set to 100.⁴

We do not report results for Pure Normal Mode Scheduling (PNMS) because its behavior is very close to that of the *Opportunistic-Best Available* algorithm described below. In the case in which the Pending Queue is full, PNMS would continue to add items to the Pending Queue without initiating asynchronous requests.

⁴We tested settings between 20 and 400 and found an increase in response times for limits less than 100 and a more gradual increase as the limit is increased beyond 100. Values less than 100 cripple the Normal Mode Scheduler; values greater than 100 run isnot NT thread limits.

When earlier pending items arrive, requests for unrequested pending items can be initiated. This will keep the number of threads within workable bounds and, at the same time, will allow PNMS to proceed. Of course, scheduled, memory-resident items are broadcast immediately. This is very similar to OS-BA since items that would have been put on the Pending Queue with PNMS, will get put there later by OS-BA.

We now turn to the evaluation of the Opportunistic Scheduling approaches. Figure 6 shows the average waiting time for the two opportunistic scheduling approaches as the cache size is increased from 5% to 40% of the database size. As a point of comparison we also ran the *RxW.90* algorithm in a blocking mode, i.e., where it simply stalls when a non-resident page is scheduled for broadcast. As would be expected, the performance with all of the approaches improves as the cache size is increased. Across all ranges the blocking algorithm performs worse than the opportunistic algorithms. At a cache size of 30% the performance of the blocking algorithm is approximately 27 times slower than either opportunistic solution. As would be expected, the blocking algorithm makes poor use of the broadcast bandwidth; The blocking approach uses only 22% of the available bandwidth even at a cache size of 40% of the database. Compared to the 98% utilization obtained when for the opportunistic scheduling cases, this result shows that completely ignoring the data staging problem can result in very poor performance due to wasted bandwidth. Of course, if the non-resident items were stored at higher-latency locations such as tertiary storage or remote sites, the penalties would be far greater.

For the opportunistic scheduling algorithms we see that beyond 30% both approaches converge because the hit rate becomes high enough that the Pending List is never full; thus, the OS mode is not used. OS-SC, which simply starts at the bottom of the LRU chain and picks the first item it finds with an outstanding request, performs significantly better than OS-BA – almost *twice* as good at a cache size of 10%.

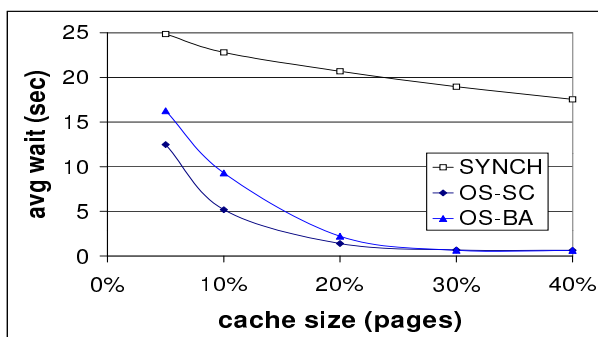


Figure 6: Average Waiting Time

In principle, the performance differences between the OS approaches can stem from two factors: the efficiency with which they make scheduling decisions, and the quality of the broadcast schedule they produce. Efficiency is important because if the scheduler takes too long, valuable broadcast bandwidth can go unused while the scheduler decides what to do. Schedule quality is important because even a fully utilized broadcast channel can provide poor performance if the pages being broadcast are not valuable.

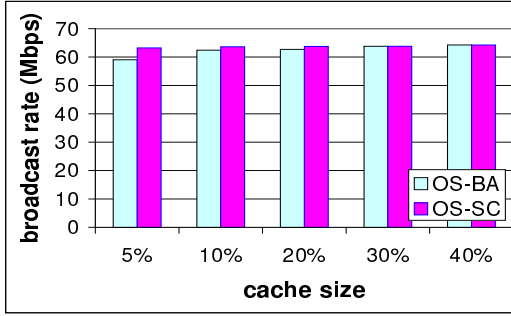


Figure 7: Data Broadcast Rate

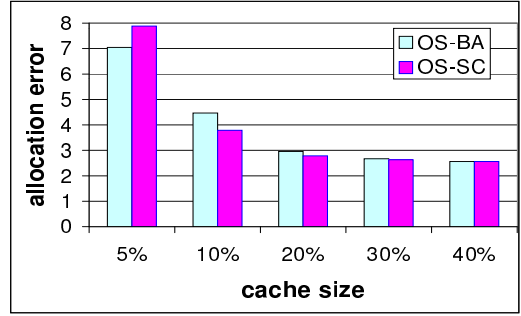


Figure 8: Scheduling Error

To demonstrate the efficiency of the two opportunistic scheduling algorithms we plot the data broadcast rate measured for this experiment in Figure 7. As can be seen in the figure, the data broadcast rate gradually increases as the cache size increases. The maximum effective bandwidth usage achieved in this experiment is 68Mbps on the downlink, even if all pages are in the cache. Note that bandwidth usage is measured by the amount of data being transmitted rather than by the bandwidth requirements of the complete UDP packets.

OS-BA applies the RxW heuristic to all cache-resident pages. This enforcement of the scheduling heuristics does come at some cost in efficiency, particularly for smaller cache sizes. With a small cache and a low hit rate, the OS mode is used more frequently, which makes its overhead more prominent. Less obviously, however, using the OS more also has the effect of making *each use* of OS-BA more expensive. This is because by choosing only cache-resident pages for broadcast, OS has the side-effect of pushing those pages lower in the Service Queue. This increases the work that OS-BA must do to find a qualifying cache-resident page. At a cache size of 5% OS-BA lets over 10% of the available bandwidth go idle. In this experimental setting, we found that the relative efficiency of the the two approaches was not a significant factor beyond a cache size of 10%.

In addition to efficiency, the other contributing factor to performance is scheduling quality. In order to examine scheduling quality we introduce a new metric that we refer to as *bandwidth allocation error*. To compute this metric, we measure the number of times each page is broadcast during the run of an experiment, and compare that with what the optimal bandwidth allocation would yield. Recall that in the optimal allocation, pages should be broadcast in proportion to the *square root* of their access probabilities [DAW86]. Based on this, we calculate the theoretically optimal allocation for each page according to the Zipf distribution and calculate the error as follows:

$$Error = \frac{\sum p_i * (optalloc_i - expalloc_i)}{all_cached_alloc}$$

where p_i is the probability of access for page i , $optalloc_i$ is the rate at which page i should be broadcast according to the theoretical optimum definition, $expalloc_i$ is the rate at which page i was broadcast during the experiment. We normalize this weighted difference by all_cached_alloc , the allocation error of the $RxW.90$ algorithm when all pages are in the cache, in order to focus on the error attributable to the OS approaches themselves. The quality differences among the approaches can be seen in the *bandwidth allocation errors*

shown in Figure 8. As can be seen, except for 5% cache size OS-SC provides a better scheduling quality. When we plot the bandwidth allocation error (over the 10000 pages in the database) and compare the algorithms for different cache sizes, the behavior is striking.

What is interesting about these results is that OS-SC, the approach that ignores the RxW metric and does a simple scan of the least recently broadcast pages with outstanding requests in the cache, produces schedules that are as good as or better than those produced by its more sophisticated counterpart. Since OS-SC starts at the bottom of the LRU chain, it tends to distribute bandwidth to the colder pages, thereby avoiding starvation of requests for those pages. Furthermore, OS-SC is more efficient in cases when the hit rate is low (except for the 5% cache size case). As a result, OS-BA has poorer performance than the much simpler OS-SC for smaller cache sizes.

These results demonstrate that the effectiveness of Opportunistic Scheduling can be improved by balancing the quality of schedules it produces with the overhead incurred to produce them. They also point out several subtle interactions between the cache management policy, the RxW scheduling algorithm, and the repeated use of OS itself. These interactions lead to the somewhat unexpected result that OS-SC, which simply chooses the first requested page that it encounters when scanning from the bottom of the LRU chain, provides the best performance in these experiments.

6 Server Cache Management

In the previous section we examined two different approaches to opportunistic scheduling. In this section we examine a modification to the cache replacement policy at the server, in order to improve the cache hit rate, thereby reducing the need for opportunistic scheduling.

The Least Recently Used (LRU) replacement policy, while extremely popular has several well-known weaknesses. One such problem has to do with items that are referenced far apart in time. Such “cold” pages are placed at the top of the LRU chain when they are accessed, and remain in memory until they float down to the bottom and are finally replaced. These cold pages effectively reduce the cache size, as they consume memory that could be used by more worthy pages. Like other systems, on-demand broadcast servers are susceptible to this problem. Recall that a good broadcast scheduler must balance the treatment of hot and cold pages. Following the square root law of the optimum bandwidth allocation [DAW86] and using a Zipf distribution, approximately 1/3 of the broadcast slots should be given to the top 10% hottest pages, with 2/3 going to the remaining 90%. Thus, many cold pages are accessed (i.e., chosen for broadcast) but for small to medium cache sizes, such individual cold pages are not likely to be chosen again before they are pushed out the bottom of the cache.

Recently, cache replacement policies that are effective at avoiding LRU’s problems with cold pages have been developed (e.g., LRU-K [OOW93] and 2Q [JS94]). These policies maintain past reference history even for items that are no longer in the cache. While such algorithms could be used in our environment as well, we have a unique advantage: the RxW algorithm already provides valuable information that can reliably be used to distinguish hot pages from cold, without the need to store additional access history. In the following subsection we describe an alternative extension to LRU cache management that exploits this information.

We refer to this policy as “LRU with Love/Hate Hints”, or simply, LH.

6.1 LRU With Love/Hate Hints (LH)

The main idea behind the LH replacement policy is to distinguish between hot and cold pages in the Service Queue. When accessed (i.e., chosen for broadcast) hot pages are tagged with a “love” hint, which causes them to be placed at the top of the LRU chain; Cold pages are tagged with a “hate” hint, which causes them to be placed at the bottom of the LRU chain, where they are likely to be chosen as replacement victims. Distinguishing between hot and cold pages is straightforward using the hints provided by the RxW scheduling algorithm. Intuitively, a page can be considered hot if it is high on the R -list (i.e., it has many requests), and can be considered cold if it is high on the W -list (i.e., it has not been broadcast for a long time). For instance, returning to the example of Figure 5, if page a is scheduled for broadcast, we can assume that it is a hot page since it is being scheduled with a high number of outstanding requests and a low waiting time. On the other hand, if page y is being scheduled we can assume that it is a cold page since it is being scheduled only after accumulating a high waiting time. Using this intuition, a page chosen for broadcast is marked with a love-hint if it meets the following two tests:

1. During the scheduler’s search of the Service Queue, the page is encountered on the R -list *before* it is encountered on the W -list.
2. The page appears in the top $hotRange$ pages of the R -List, where $hotRange$ is a number of pages calculated using the cache size and other metrics described below.

The first test simply ensures that the page is higher on the R -list than on the W -list, which indicates that it might be a “hot” page. This test is not sufficient, however. In some cases, more pages than can fit in cache would satisfy this test, therefore, the second test is applied in order to further limit the number of pages marked with love hints. $hotRange$ is calculated as $cacheSize \times \frac{entryCnt}{dbSize}$ pages, where $entryCnt$ is the number of entries in the Service Queue at the time of scheduling decision, and $dbSize$ is (an estimate of) the number of unique pages that can be requested by the client population. This formulation allows the number of pages marked with love hints to be scaled to the cache size and to the intensity of the workload.

If a page does not receive a love-hint, it will receive a hate-hint. Pages with love hints are treated normally with respect to the cache management policy. Pages with hate-hints are demoted to the end of the LRU chain.

6.2 Caching: Evaluation

In order to evaluate the LH approach, we repeated the experiment of the previous section using LH instead of LRU as a replacement policy. As a comparison point, we also tested a third policy called PCACHE. PCACHE uses perfect *a priori* knowledge of the data access distribution and pins the pages with the highest access probabilities in cache without any replacement taking place. PCACHE is not a practical policy for our environment, rather it represents the ideal case that LH is attempting to approach. The average waiting

time for the three replacement policies is shown in Figure 9. In all cases, opportunistic scheduling with OS-SC (the best performing algorithm in the previous experiment) is being used in addition to the caching policy. Note that the curve labeled “LRU” is the “OS-SC” curve from Figure 6 of the previous experiment.

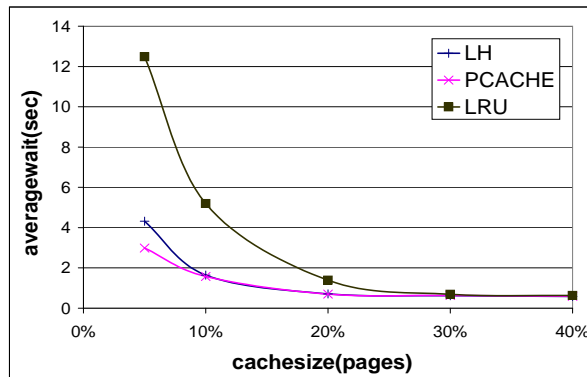


Figure 9: Average Wait Time

As can be seen in the figure, LH is able to provide substantial benefits over LRU for smaller cache sizes (where the replacement policy has the largest impact). At cache sizes of 10% or less, using LH instead of LRU results in a factor of 3 further performance improvement. Furthermore, although it may not be apparent due to the scale here, LH also provides a factor of 2 improvement over LRU at a cache size of 20%. Comparing LH to the idealized PCACHE, it can be seen that at a cache size of 10% and beyond, they provide nearly identical performance.

For the same experiment, we also measure the rate at which we obtained a cache hit or a cache miss and the rate at which we have applied an opportunistic scheduling decision in Figures 10 and Figure 11. For each processing iteration of the server we have one of the three outcomes: 1) if we are in normal scheduling mode and the item is in the cache at the time it is selected by the scheduler, the outcome is a *cache-hit*; 2) if we are in normal scheduling mode and the selected page is not cache-resident, the outcome is a *cache-miss*; 3) if we are in opportunistic scheduling mode, we alter the heuristics in order to ensure a cache hit and the outcome is a specialized case of a hit (referred to as *opp* in the graphs).

Figure 10 plots our measurement for LRU. We see that cache hit rate increases with the cache size and that for small cache sizes, opportunistic scheduling is applied beyond a miss rate of 40% by broadcasting cache resident pages as expected. Figure 11 demonstrates the case for LH. As can be seen, LH successfully increases the cache hit rate and thereby reduces the need for opportunistic scheduling. Increasing the cache hit rate is equivalent to increasing the variety of pages that can be broadcast.

The effectiveness of the increased hit rate is indicated in Figure 12, which shows the *weighted* hit rate at the server cache for this experiment. In a broadcast server, calculating the hit rate in terms of hits and misses each time a page is selected for broadcast does not provide an accurate measure of the impact of a cache management policy. This is because hits and misses affect different numbers of clients depending on how many requests were outstanding at the time the item was scheduled for broadcast. For this reason we introduce an additional metric, the weighted hit rate. The weighted hit rate takes client requests into

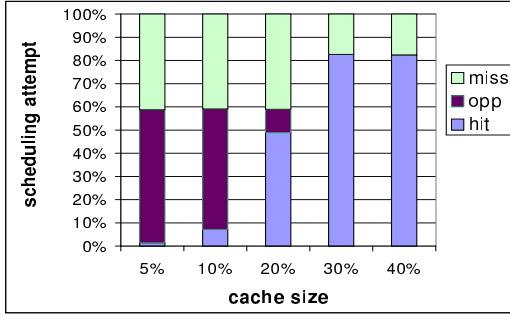


Figure 10: Success Rate for LRU

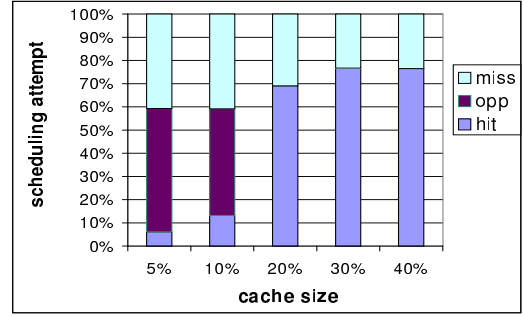


Figure 11: Success Rate for LH

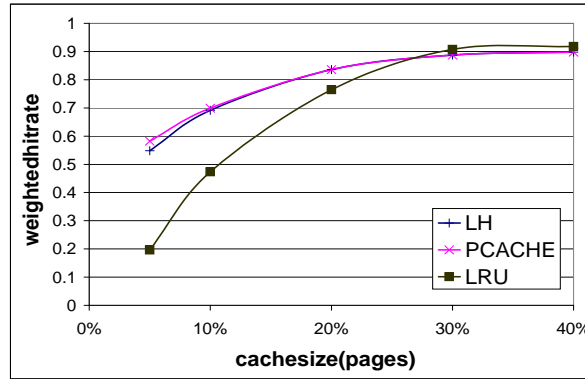


Figure 12: Weighted Hit Rate

consideration, and is calculated as:

$$w_hit = \frac{\sum h_i}{\sum h_i + \sum m_i}$$

where h_i is the total number requests that were outstanding for page i whenever there was a cache hit for page i (normal or opportunistic scheduling), and m_i is the total number of requests outstanding for page i whenever there was a cache miss for page i (normal scheduling). As can be seen in Figure 12, LH improves the weighted hit rate (over LRU) almost as much as the idealized PCACHE does, showing that LH is able to distinguish between hot and cold pages almost as accurately as if it had perfect knowledge of the workload. These results indicate that LH is an effective replacement for LRU in an on-demand broadcast server. Since LH uses only information that is already maintained by the broadcast scheduler, it has an advantage over other LRU replacements, which would require additional statistics and mechanisms.

7 Prefetching

In this section we examine a third complementary approach to the data staging problem, namely, prefetching items that are likely to be broadcast in the near future. As with cache management, we exploit properties of the scheduling algorithm to make such predictions.

7.1 Reducing I/O latency

The goal of prefetching is to improve the cache hit rate by predicting which pages are likely to be needed in the near future and taking steps to ensure that they are brought into the cache by the time they are needed. We have discussed that for a skewed request distribution an ideal broadcast schedule consists of a small number of frequently broadcast hot pages and a high number of cold pages. As shown in the previous section, the LH cache replacement policy is extremely effective at identifying *hot* pages and retaining them in the cache. Thus, when used in conjunction with LH, prefetching will likely be needed primarily for *cold* pages. Stated in RxW terms: the cache replacement policy is effective at keeping high R -valued pages in cache, so prefetching should concentrate on high W -valued pages.

Fortunately, the characteristics of RxW scheduling are such that the W -list is a much more stable structure on which to base prefetching than the R -list. The W -list is effectively a FCFS queue. Once items are placed in the W -list their relative ordering does not change. While hot pages are likely to be chosen for broadcast before they obtain a high position in the W -list, cold pages slowly but inexorably move towards the top. Note that due to the stability of the W -list, cold pages are always broadcast in the order that they are added to the list. Thus, an effective technique for improving the hit rate for cold pages is to keep the top W -list pages memory-resident, prefetching those items that approach the top, but are not yet cache-resident.

We have added a fairly direct implementation of this concept to our prototype broadcast server. When *prefetching* is enabled, the server aims to have the items appearing in the top *prfWindow* positions (where *prfWindow* is an integer smaller than the number of frames in the cache) of the W -list in memory by the time they are scheduled to be broadcast. A budget of *prfWindow* pages is reserved in the cache (i.e., taken out of the LH-managed space) for keeping prefetched pages. The mechanism works as follows: Initially, asynchronous fetch requests are made for the pages in the top *prfWindow* entries on the W -list. When they arrive, these pages are marked as prefetched pages so they will not be replaced by the cache manager. When a prefetched page is eventually chosen for broadcast, this special mark is removed, and the page is handled according to the LH replacement policy (most likely it will be marked with a hate hint and soon ejected). This process brings a new page into the *prfWindow*. This page is prefetched asynchronously (if it is not already cache resident) and marked to remain in the cache until it is scheduled. Note that in the case in which a page is scheduled for broadcast while it is in the process of being prefetched, that page will be treated as an asynchronously fetched page when it arrives (i.e., it will be broadcast when it arrives at the cache).

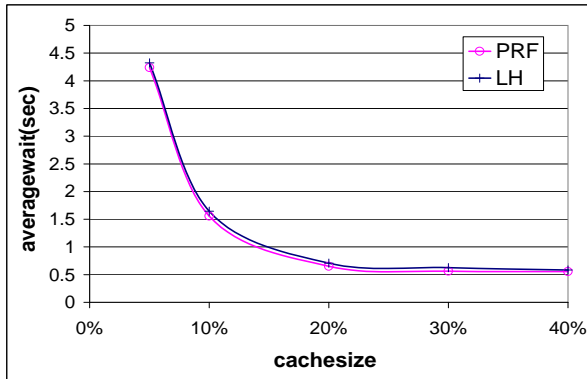


Figure 13: Average Wait (Disk Resident Data)

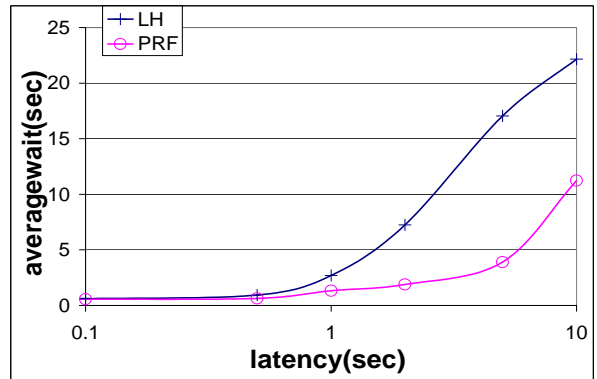


Figure 14: Average Wait varying Latency

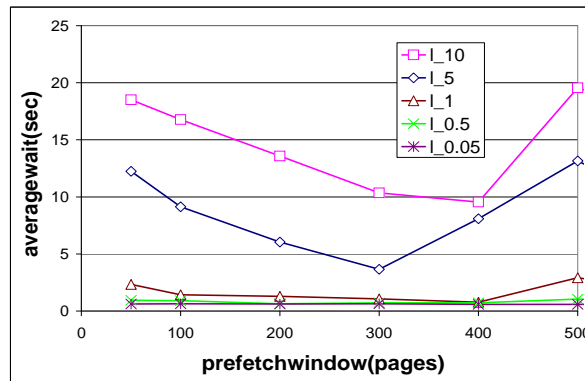


Figure 15: Sensitivity to Prefetch Window

7.2 Prefetching: Evaluation

Figure 13 shows average wait time for the LH algorithm (i.e., OS-SC scheduling with LH cache replacement) and for the LH algorithm augmented with prefetching (labeled “PRF”) for the same experimental settings as the previous sections. Essentially the curve labeled LH in this figure is the same as the LH curve that appears in Figure 9. In this experiment, the *prfWindow* was set to 75 pages, although as is discussed below, there is little performance difference for *prfWindow* sizes up to 500 pages. As can be seen in the figure, in this case, prefetching provides little or no additional benefit across the entire range of cache sizes studied.

The reasons behind this (admittedly, disappointing) result are interesting. Our measurements indicated that as we expected, prefetching was indeed improving the hit rate. However, since prefetching helps only with cold pages, the improved hit rate in terms of scheduling decisions translates to a much smaller improvement in weighted hit rate. Furthermore, in this case, the savings of using prefetching is to avoid the latency associated with retrieving cold items from the local disk. It turns out, however, that in this experimental setting, requests for cold pages typically accumulate waiting times on the order of seconds before the page is chosen for broadcast. Compared to this long wait, the latency associated with the disk read (on the order 10’s of milliseconds) is not significant. Thus, in retrospect, it is apparent that prefetching cold pages will

not help here.

This observation, however, leads us to investigate the benefits of prefetching for higher-latency situations, such as tertiary storage or obtaining the items from remote sites across the Internet. To examine these situations, as described in Section 4, we introduce a synthetic delay to the asynchronous fetching process. That is, we artificially delay the retrieval of non-resident pages by sending the asynchronous fetch requests to a wait list rather than to the local disk in order to simulate larger latencies. Figure 14 shows the performance of LH and PRF for a system with a 20% cache as the latency for obtaining non-resident pages is increased from 0.1 seconds (i.e., on the order of a random disk access) to 10 seconds (i.e., on the order of a request to a heavily-loaded site over a slow network). In this case the *prfWindow* is set to 300 pages. As seen in the previous results, prefetching does not help for relatively low latencies. For longer latencies, however, it can have very significant benefits. For example, at a latency of 5 seconds here (note the log-scale x-axis), prefetching provides a more than 3-fold improvement in performance. Note that this improvement is in addition to the improvements already obtained by the OS-SC and LH techniques.

These results lead us to conclude that prefetching has the potential to be a very important technique for use in environments where the items to be broadcast reside on tertiary storage or must be fetched from servers across the Internet. This latter class of systems is particularly important, as one emerging use of on-demand broadcast is for WWW proxy servers.

Finally, for completeness, we include Figure 15, which shows the sensitivity of prefetch performance to the Prefetch window size (x-axis). The curves are (from the bottom) for latencies of 0.05, 0.5, 1, 5, and 10 seconds. As can be seen in the figure, the sensitivity of the algorithm increases with the latency. Thus, this sensitivity must be studied further as part of the development of an on-demand broadcast server for high-latency environments.

8 Related Work

As stated previously, there has been significant work on developing scheduling algorithms for on-demand data broadcasting [DAW86, DW88, Won88, VH96, ST97b, AF99], but with the exception of Dykeman and Wong [DW88], all of this work ignored data staging issues. This latter study recognized the potential benefits to be gained by integrating data movement with the broadcast scheduling algorithm, and also exploited tradeoffs between schedule quality and bandwidth utilization. Triantafillou et al. [THP01] have recently demonstrated the importance of the data staging problem for secondary storage latencies, and have also proposed solutions based on disk scheduling solutions⁵. Similar to our work, this work also makes use of *RxW* scheduling for some of their proposed algorithms. However, there are substantial differences between our work and these two studies. First, the context of the earlier work was *Teletext* systems, which had much lower bandwidths and extremely small databases to broadcast. Thus, the proposed solutions used very expensive scheduling and cache replacement algorithms that are not appropriate for a large-scale system. Second, both studies addressed only local disk issues and the solutions assumed detailed control over disk devices, to a degree that is not applicable with today's commodity disk drives and controllers (e.g., SCSI

⁵We have been informed that the authors are currently looking at server cache management.

disks). In addition such solutions can not be generalized to the case of data retrieval at a remote site. Finally, it should be noted, that both studies were done using a simulation which ignored much of the overhead and resource contention that arises in a real system. In contrast, we developed and studied our algorithms in the context of a working prototype. This is a key issue, as the use of Opportunistic Scheduling is motivated in large part due to the overheads associated with asynchronous processing in modern operating systems.

Data staging has also been studied for multimedia systems. Ozden et al. [ORS96] studied buffer replacement algorithms for multimedia storage systems that exploit the large file sizes and sequential access found in many multimedia applications. In the multimedia community, Aggarwal et al. [AWY96] have studied scheduling algorithms for Video-On-Demand systems, and proposed a heuristic that uses the number of outstanding requests and the broadcast history for each item. Like previous work on broadcast scheduling, however, this study assumes that there is no cost associated with obtaining the items to be broadcast.

Data staging concerns also arise in more traditional, geographically distributed information systems. For example, [TTS⁺98] study the distribution and the placement of data over numerous geographically dispersed locations using an analytical model. The model is based on the assumption that all network configurations, load and requests are fixed and known a priori. This problem is significantly different than the data staging problem that arises for on-demand broadcast.

Substantial prior work exists in the area of prefetching for various types of information systems. In [FD95], prefetching was shown to be an effective performance enhancer for video-on-demand systems. The success of prefetching in this study, however, is mainly based on the sequential access of video files. Palmer and Zdonik [PZ91] applied prefetching based on learning patterns from reference traces in a client-server database context. A more recent paper by Bernstein et al. [BPS99] examines the use of prefetching based on navigational access patterns for a similar environment. In the file system context, Patterson et al.'s work on Informed Prefetching [PGG⁺95] uses application-provided hints to exploit IO concurrency.

Our work on prefetching differs from this previous work because of certain characteristics of the on-demand broadcast environment in general, and due to specific attributes of the algorithms we use to implement our broadcast server. For example, we exploit the ability of the LH cache replacement algorithm to retain hot pages in the cache, and the predictable behavior of the RxW scheduling algorithm with respect to cold pages to be able to do prefetching without any trace analysis or hints from the application. This, combined with the unique tradeoffs inherent in a broadcast environment results in a radically different set of concerns compared to prefetching in more traditional information system architectures.

9 Conclusion

Data broadcast is increasing in popularity as high-bandwidth broadcast-capable infrastructures such as satellite or cable data networks are emerging for high-speed data service for large client populations. In this paper we have investigated the integration of data staging concerns with an on-demand data broadcast server for large-scale applications. Ignoring data staging issues can have disastrous implications for the utilization of the key shared resource in such systems, namely broadcast bandwidth. We described an on-demand broadcast server architecture that incorporates three complementary solutions to the data staging problem.

First, *Opportunistic Scheduling* is used when the system-dependent limit on outstanding asynchronous fetch requests is reached. When running in OS mode, the scheduler is restricted to choosing cache-resident items for broadcast. We proposed three different approaches to OS, two of which were extensions to the *RxW* broadcast scheduling algorithm and one which simply scans items in the cache. We also described how to integrate OS into the basic control flow of the broadcast server. Second, we described the *LRU with Love/Hate hints* (LH) cache replacement policy, which exploits hints easily obtained from the *RxW* scheduling algorithm to effectively keep hot pages in the cache. Third, we developed an integrated prefetching scheme, whose job is to ensure that cold pages are cache-resident by the time they are scheduled to be broadcast.

We then presented the results of a set of experiments performed on a prototype broadcast server that we extended with these techniques. The experiments identified some subtle interactions between the cache contents, the quality of the schedule produced, and the speed with which the scheduler makes decisions. In general, the results showed that the simple approach to Opportunistic Scheduling (called OS-ScanCache) and the LH cache replacement policy provided significant benefits even for a relatively low-latency environment such as secondary storage. In contrast, prefetching was found to be helpful only in higher-latency situations, as would arise when obtaining data from tertiary storage or from remote sites across the Internet. In such cases, however, the potential benefits are substantial.

This latter observation raises some important directions for future work. In particular, we plan to focus on wide-area systems in which the on-demand broadcast server acts as a proxy server for WWW or other Internet access. In such an environment, the scheduling and data staging approaches must be extended to cope with variable-length objects and the particular types of latencies that can arise. For example, in a WWW environment there is an important tradeoff between caching at the broadcast server and obtaining the most recent version of a page from its source. Furthermore, data staging in a web environment can benefit from parallelism by sending requests to multiple remote sources, either to obtain different items for broadcast or in order to obtain the fastest response from sites that store replicas of a single object. Our goal is to develop a scalable approach to on-demand broadcast that enables it to be used as a key technology in the global-scale information systems that are now beginning to emerge.

References

- [AF98] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. In *Proceedings of IEEE INFOCOM*, March 1998.
- [AF99] D. Aksoy and M. Franklin. RxW: A scheduling approach to large scale on-demand broadcast. In *IEEE/ACM Transactions on Networking*, volume 7, pages 846–861, Dec. 1999.
- [Aka] Akamai. Delivering a better internet. <http://www.akamai.com/>.
- [Alo00] Alohanet. Skydsl internet access. <http://www.alohonet.com/>, 2000.
- [AM98] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proc. of Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom)*, Dallas, TX, 1998.
- [AWY96] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On optimal batching policies for video-on-demand storage servers. In *The Third IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996.
- [BCF⁺99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of Infocom'99*, 1999.

- [BPS99] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *Proc. VLDB*, pages 327–338, Edinburg, Scotland, September 1999.
- [Cid] Cidera. The internet broadcast backbone. <http://www.cidera.com/>.
- [DAW86] H.D. Dykeman, M. Ammar, and J.W. Wong. Scheduling algorithms for videotex systems under broadcast delivery. In *IEEE International Conference on Communications*, Toronto, Canada, 1986.
- [Dir96] Hughes network systems, direcpc homepage. <http://www.direcpc.com>, 1996.
- [DW88] H.D. Dykeman and J.W. Wong. A performance study of broadcast information delivery systems. In *Proc. IEEE Infocom*, New Orleans, LA, 1988.
- [Edg] Edgix. Leading edge intelligence. <http://www.edgix.com/>.
- [FD95] C. S. Freedman and D. J. DeWitt. The spiffi scalable video-on-demand system. In *SIGMOD*, San Jose, CA, 1995.
- [FZ98] M. Franklin and S. Zdonik. Data in your face: Push technology in perspective. In *Proc. ACM SIGMOD Conference*, June 1998.
- [JS94] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, Santiago de Chile, Chile, September 1994.
- [Knu81] D. Knuth. *The art of Computer Programming - Volume III*. Addison-Wesley, 1981.
- [Net] Fast Forward Networks. Inktomi's media products. <http://www.fastforwardnetworks.com/>.
- [OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, Washington, DC, may 1993.
- [ORS96] B. Ozden, R. Rastogi, and A. Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *IEEE International Conference on Multimedia Computing and Systems*, June 1996.
- [PGG⁺95] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th Symp. On Operating System Principles*, Dec. 1995.
- [PZ91] M. Palmer and S. Zdonik. Fido: A cache that learns to fetch. In *Proc. of VLDB*, pages 255–264, September 1991.
- [ST97a] C.J. Su and L. Tassiulas. Broadcast scheduling for distribution of information items with unequal length. In *Proc. 31th Conf. on Information Sciences and Systems (CISS'97)*, Kobe, Japan, March 1997.
- [ST97b] C.J. Su and L. Tassiulas. Broadcast scheduling for information distribution. In *Proc. IEEE INFOCOM*, 1997.
- [Sta00] Starband. America's first consumer two-way always on, high speed satellite Internet service. <http://www.starband.com>, 2000.
- [THP01] P. Triantafyllou, R. Harpantidou, and M. Paterakis. High performance data broadcasting: A comprehensive system's perspective. In *Proc. 2nd Intl Conference on Mobile Data Management*, Hong Kong, January 2001.
- [Tri82] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Application*. Prentice-Hall Inc, 1982.
- [TTS⁺98] M. Tan, M. D. Theys, H. J. Siegel, N. B. Beck, and M. Jurczyk. A mathematical model, heuristic, and simulation study for a basic data satging problem in heterogenous networking environment. In *Proc. International Computing Workshop (HCW '98)*. IEEE, 1998.
- [VH96] N.H. Vaidya and S. Hameed. Data broadcast in assymmetric wireless environments. In *Proc. of Workshop on Satellite-based Information Services (WOSBIS)*, New York, November 1996.
- [VH97] N.H. Vaidya and S. Hameed. Log-time algorithms for scheduling single and multiple channel data broadcast. In *Proc. of Workshop on Satellite-based Information Services (WOSBIS)*, Budapest, Hungary, September 1997.
- [Won88] J.W. Wong. Broadcast delivery. *Proc. of IEEE*, 76(12):1566–1577, December 1988.