Data Staging for On-Demand Broadcast *

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

Michael J. Franklin
University of California, Berkeley
franklin@cs.berkeley.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Abstract

The increasing deployment of broadband services that are inherently broadcast-capable has made wide-area data broadcast an attractive data delivery alternative for large client populations. There has been significant work towards developing on-line broadcast scheduling algorithms for systems where all data items are readily available in the server's main memory. These studies ignore the data management issues that arise when the data to be broadcast must be obtained from secondary storage or remote locations. In this paper, we propose three complementary solutions to such *data staging* concerns: opportunistic scheduling, server caching, and prefetching. These techniques exploit hints provided by the scheduling algorithm. A detailed performance evaluation using an IP Multicast-based testbed shows that these data staging techniques can 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 retrieval and delivery of data for huge numbers of users, such as Content Delivery Networks, Cooperative Web Caches, and Wide-area Distributed Storage Networks. 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. Prominent examples include the offerings of such companies as Akamai, Fast Forward Networks (now part of Inktomi), Edgix, and Cidera.

These systems have a great deal of flexibility in the types of communication they can employ, since they use their own communication channels and application-level routing 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.

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*.

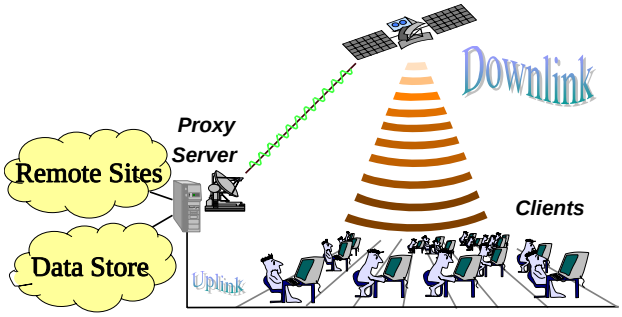

Figure 1: Example Data Broadcasting Scenario

*This work was supported by the University of California Davis – Junior Faculty Research Fellowship, and in part by the National Science Foundation under NSF grants IIS00-86057, IRI96-32629, and IRI95-01353, by DARPA under contract number N66001-99-2-8913, and by contributions from IBM, Microsoft, Sun Microsystems, and Siemens.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

An example of an 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 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. Our interest is in large-scale dissemination systems. 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. 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 also possible.

1.2 Broadcast Scheduling and Data Staging

A key design consideration in the development of a large-scale on-demand data broadcast server is the *scheduling algorithm* used to select items to be broadcast. Such an algorithm aims to choose the most beneficial data items 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, VH97, ST97a, AF99]). A key concern in the face of high downlink bandwidths[1] has been developing low overhead algorithms so that the available broadcast bandwidth can be fully utilized.

For a large regional or national-scale information service, however, 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.

Significant performance degradation can be experienced due to such additional latency. As a result, scheduling efficiency is not sufficient to guarantee the full utilization of the downlink channel. 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*.

1.3 Data Staging Solutions

In this paper, we attack the data staging problem using three complementary approaches based on:

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, an available data item should be broadcast. We refer to this technique as *Opportunistic Scheduling*.

2. *Decreasing the need to fetch an item:* Obviously one should try to make the best use of the available memory space on the server. The key to successful caching is to retain those items that are most likely to be scheduled. For this purpose, we exploit hints maintained by the scheduling algorithm to differentiate between hot (popular) and cold (not-so-popular) items. We refer to this technique as *Hint-based Cache Management*.

3. *Decreasing the fetch latency:* Access latency can be reduced by obtaining items from slow or remote locations *before* they are needed. This translates to bringing the items that are likely to be scheduled in the near future into the cache. We refer to this technique as *Prefetching*.

In the following, we first describe the general architecture of an on-demand broadcast server. We then describe the proposed data staging mechanisms in detail, and present performance analysis results obtained using our data broadcast prototype testbed.

2 Broadcast Data Staging Techniques

We have developed an on-demand broadcast server that can provide good performance even when much of the data is not memory-resident. 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. Therefore, the scheduler must be non-blocking; i.e., 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

[1] The 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.

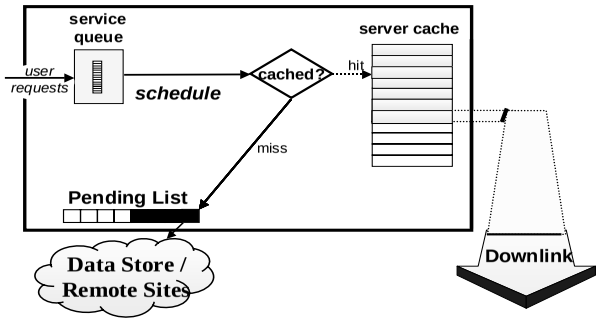

Figure 2: Normal Scheduling

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. For this reason, our server respects a preset limit on the number of outstanding data fetch requests.

Figure 2 depicts the server architecture. 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, there is a single entry for each unique item that has outstanding requests. 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 accordingly, otherwise a new entry is added to the Service Queue. We describe the scheduling algorithm in Section 3. Note that when an item is chosen to be broadcast, its corresponding 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 requests is enforced by bounding the size of the *Pending List*.

The scheduler loops continuously: First it checks 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, the scheduler is run to select a data item to broadcast from among those that have an outstanding request in the service queue: If the Pending List is not full, then the scheduler is run in *Normal* mode. Otherwise, it is run in *Opportunistic* mode.

In the **normal scheduling mode**, the server searches the Service Queue to select an item to broadcast. Once an item is selected, 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. Otherwise, (i.e., a cache miss) 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 case, the server continues to the next iteration of the processing loop without broadcasting an item. Note that as long as an item has an entry in the Pending List, the server does not create a new service queue entry for any incoming requests for this item. This is because at that point the item is already slated to be broadcast when it arrives.

When the Pending List is full, the scheduler operates in **opportunistic scheduling (OS) mode**. When in OS mode, the scheduler chooses *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. 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. The challenge in designing the algorithm for OS is to strike a balance between efficiency and effectiveness. We describe two OS approaches in Section 5.

2.1 Cache Management and Prefetching

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 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 in Section 6.

The second technique is *prefetching*. Our use of prefetching is complementary to the LH cache management approach. LH aims at keeping items that are likely to be broadcast *again* in the near future (i.e., hot items) in cache. In contrast, prefetching aims to bring into the cache, pages that are not cache resident (i.e., cold items) when they are likely to be broadcast in the near future.

With the prefetching mechanism, a set of frames in the cache is reserved for holding prefetched items. The prefetch process snoops the service queue to identify items that are likely to be broadcast soon and tries to ensure that they are in memory when they are eventually chosen for broadcast. Similar to the handling of fetches for cache misses, a prefetch request for an item is sent asynchronously to the item's location.

It is important to note that the three data staging techniques: *opportunistic scheduling, hint-based cache management, and prefetching* are integrated with the scheduling algorithm being used.

3 The RxW Scheduling Algorithm

In this section, we present a brief sketch of the *RxW* broadcast scheduling algorithm [AF98, AF99], which serves as the basis for our integrated broadcast scheduling and data staging techniques. *RxW* has been shown to provide good performance 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. In this work, we focus on scheduling the broadcast of fixed-length items such as disk or database pages.

RxW maintains two values with every service queue entry: 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 highest $R \times W$ value is preferred, however, this search could be quite time consuming which risks wasting downstream bandwidth. Thus, approximate algorithms have been developed. Our server uses one called *RxW.*α where α is a tunable parameter that controls the desired level of approximation.

The server maintains two sorted lists threaded through the service queue: 1) *R-list*: based on the number of outstanding requests, and 2) *W-list*: based on the waiting time of the oldest request for that page. 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 *threshold*. The *threshold* is the running average of the $R \times W$ values of the pages broadcast so far. After each broadcast decision, the *threshold* is updated and the service queue entry for the selected page is removed.

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 lower is the overhead (fewer entries are scanned) with a correspondingly lower accuracy. [2]

[2]When α is set to 0, only two entries (the one at the top of the *R-list* and the one on the top of *W-list*) are examined. At the other extreme when *alpha* is set to ∞, the algorithm stops the search when the maximal $R \times W$ value is guaranteed (see [AF99] for more details).

4 Experimental Environment

4.1 Prototype

In order to study our solutions, we have implemented a prototype on-demand broadcast testbed on a cluster of pentium-based machines running Windows NT 4.0. Each machine 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 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.

4.2 Workload, Metrics and Measurements

For the experiments, 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. File system buffering was disabled during the experiments so as not to interfere with the measurements.

The cache size is varied between 5% and 100% of the database. We focus on the results for up to 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 the 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). Equilibrium occurs 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 we report additional metrics where necessary.

The scheduling algorithm used in these experiments is *RxW* as described in Section 3. 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 $\alpha = 0.9$. 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 changes somewhat for different approximation settings, the trends described in this paper hold for all cases tested.

Given this background we now examine the data staging approaches we have proposed in greater detail and present the experimental results we have obtained.

5 Opportunistic Scheduling

5.1 Algorithms

Opportunistic Scheduling (OS) is the scheduling mode that is used when the server has reached its limit on outstanding asynchronous requests, as described in Section 2. In OS mode the scheduling algorithm is restricted so that it selects only cache-resident pages for broadcast. 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 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 *RxW* criteria.

Figure 3 shows an example. 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 $R \times W$ 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 $R \times W$ value of page *c* is 90, which does not qualify. It then switches to the *W-list* where it first examines page *z*, which has an $R \times W$ value of only 60. 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 $R \times W$ value (1100) is larger than the stopping condition (450).

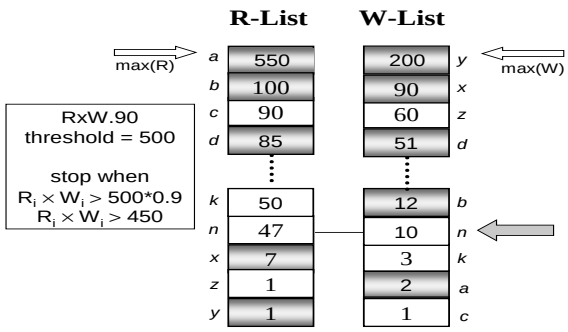

Figure 3: Scheduling for Best Available

Scan Cache (OS-SC) - The second opportunistic approach completely foregoes the use of the *RxW* algorithm during opportunistic scheduling. 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 omitted due to space considerations. In the example of Figure 3, OS-SC is likely to schedule page *z* for broadcast.

Before examining the individual approaches, it is important to emphasize that both OS approaches are *extremely* effective in solving the problems caused by non-resident data items. Thus, in the following 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.

5.2 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.[3]

Figure 4 shows the average waiting time for the two opportunistic scheduling approaches as the cache size

[3]We tested settings between 20 and 400 and found a steep 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 into NT thread limits.

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 (SYNCH) 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% SYNCH 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 for the OS cases, this result shows that as expected ignoring the data staging problem can result in very poor performance. The penalties would be far greater if the non-resident items were stored at higher-latency locations such as remote sites.

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 hardly ever 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%.

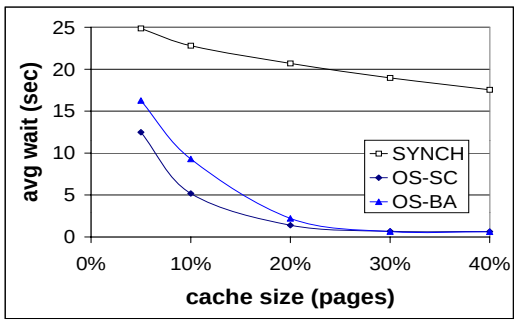

Figure 4: Average Waiting Time

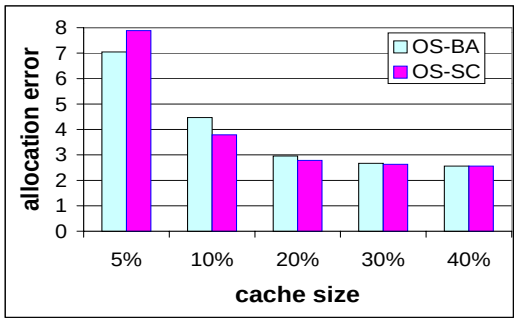

Figure 5: Scheduling Error

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.

In terms of efficiency the data broadcast rate of OS-SC is very close to the maximum effective bandwidth throughout the whole range of cache sizes. On the other hand, OS-BA pays a price for searching the service queue, particularly for smaller cache sizes. With a small cache and a low hit rate, the OS mode is used more frequently. This 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 maximum obtainable bandwidth go idle. Beyond a cache size of 10% OS-BA is as efficient as OS-SC.

The other contributing factor to performance is scheduling quality. In order to examine scheduling quality we introduce a new metric called *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. 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 5. As can be seen, beyond a cache size of 5% OS-SC provides better scheduling quality. The reason behind this is that OS-SC distributes bandwidth to colder pages while OS-BA give too much bandwidth to hot pages. Since OS-SC is more efficient and produces better schedules, it has better performance than OS-BA for small cache sizes.

6 Server Cache Management

Next 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 OS.

The Least Recently Used (LRU) replacement policy, while extremely popular has a well-known weakness. "Cold" pages that are placed at the top of the

LRU chain remain in memory until they float down to the bottom and are finally replaced. These cold pages effectively reduce the cache size. 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 replaced.

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, we describe an alternative extension to LRU cache management that exploits this information. We refer to this policy as "LRU with Love/Hate Hints" (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 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 *RxW*. 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 3, 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 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 LRU 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 6. 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 4 of the previous experiment.

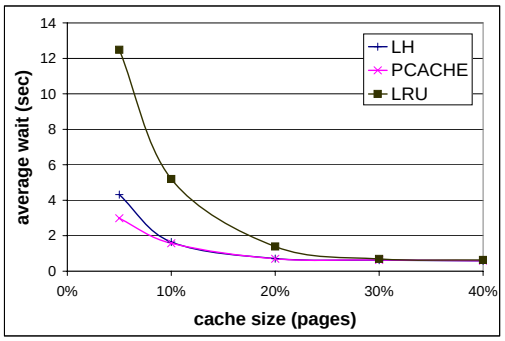

Figure 6: Average Wait Time

As shown in Figure 6, 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 improvement. Furthermore, although it may not be apparent due to the scale here, LH also provides a factor of 2 improvement over LRU

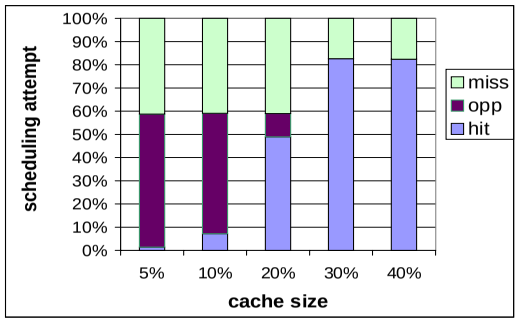

Figure 7: Success Rate for LRU

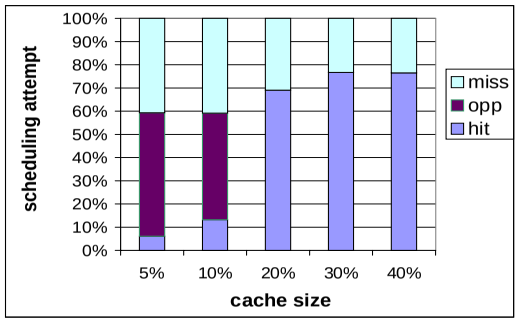

Figure 8: Success Rate for LH

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 have the same performance.

There are three possible outcomes for each scheduling iteration: 1) *cache-hit*: when we are in normal scheduling mode and the item is in the cache at the time it is selected by the scheduler; 2) *cache-miss*: when we are in normal scheduling mode and the selected page is not cache-resident; 3) *opp*: if we are in opportunistic scheduling mode, a specialized case of a hit, as we alter the heuristics in order to ensure a cache hit. Figures 7 and 8 plot the distribution of these outcomes for LRU and LH respectively. As can be seen, LH successfully improves the cache hit rate for small cache sizes 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.

These results indicate that LH is an effective replacement for LRU in an on-demand broadcast server, and that LH is able to distinguish between hot and cold pages almost as accurately as if it had perfect knowledge of the workload.

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 *RxW* algorithm to make such predictions.

7.1 Reducing I/O latency

The goal of prefetching is to predict which pages are likely to be needed in the near future and to take steps to ensure that they are brought into the cache by the time they are needed. Recall 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 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: new entries are appended to the tail of the *W-list*, and 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 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. The maximum allowable number of such items is a configuration parameter referred to as the *Prefetch Window (prfWindow)*. Pages within the prfWindow are guaranteed to be either in the cache or to be in the process of being prefetched at any time.

We have added a fairly direct implementation of this concept to our prototype broadcast server. When *prefetching* is enabled, 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 pinned in the cache until they are broadcast. When a prefetched page is eventually chosen for broadcast, the page is unpinned, and handled according to the LH replacement policy (most likely it will be marked with a hate hint and soon ejected). The process of broadcasting a page that was within the *prfWindow* brings a new page into the *prfWindow*. This page is prefetched asynchronously if it is not already cache resident. Otherwise it is pinned in the cache until it is scheduled. Note that if 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 (it will be broadcast) when it arrives at the cache.

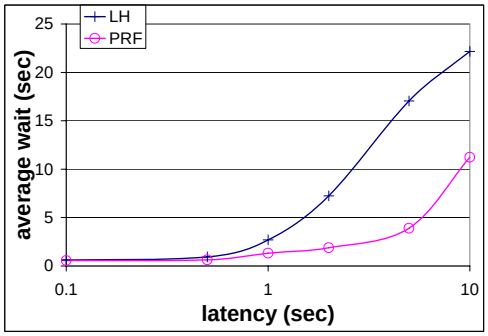

Figure 9: Average Wait varying Latency

7.2 Prefetching: Evaluation

We first repeated the previous experiment with prefetching. In this case, we have observed that prefetching provides little or no additional benefit across the entire range of cache sizes studied. The reasons behind this result are interesting. Our measurements indicated that as we expected, prefetching was improving the hit rate. However, in this case, the savings of using prefetching is the latency associated with retrieving cold items from the local disk. It turns out 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 of 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, we introduce a synthetic delay into 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 9 shows the performance of LH and LH with prefetching (labeled 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 already described, 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 those 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.

8 Related Work

There has been significant work on scheduling algorithms for on-demand data broadcast [DAW86, DW88, VH96, ST97b, AF99]. Scheduling approaches for variable-length data, particularly for multimedia applications have also been proposed [VH97, ST97a, AM98]. However, 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. However, since their environment had much lower bandwidths and much smaller databases to broadcast, efficiency was not an important concern. As a result, the proposed solutions used very expensive scheduling and cache replacement algorithms that are not appropriate for a large-scale system. Triantafillou et al. [THP01] have recently studied data staging for on-demand broadcast. They have used *RxW*, but did not exploit its inherent hints and did not consider prefetching of data items. Similar to Dykeman et al., their study addressed local disk latencies using disk scheduling algorithms. Finally, both Dykeman et al. and Triantafillou et al. studied their solutions using a simulation which did not capture the overhead and resource contention that arises in a real system. This is a key issue, as the use of Opportunistic Scheduling is motivated in large part due to the overheads observed with asynchronous processing in a working prototype.

In the multimedia community, Ozden et al. [ORS96] studied data staging for multimedia storage systems using buffer replacement algorithms and prefetching. The data staging problem in this case is different than ours due to the sequential access characteristics of video files. Data staging concerns also appear more traditional information systems. Tan et al. [TTS+98] have studied the distribution and the placement of data over numerous geographically dispersed locations based on the assumption that all requests and load as well as all network configurations are known apriori. Therefore the suggested solutions are not applicable for on-line algorithms of on-demand broadcast systems where the client requests are not known apriori.

Substantial prior work also exists in prefetching for video-on-demand systems [FD95], client-server architectures [PZ91], for file systems [PGG+95] and for navigational access systems [BPS99]. Our work on prefetching differs from these studies because of specific attributes of the algorithm we use to implement our broadcast server. For example, we exploit 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.

9 Conclusion

We have investigated the integration of data staging concerns with an on-demand data broadcast server for large-scale applications using three complementary solutions. First, *Opportunistic Scheduling* is used to increase the bandwidth utilization. When running in OS mode, the scheduler is restricted to choosing cache-resident items for broadcast. We proposed two approaches to OS and showed that the decision on which available page to broadcast should be based on the overall bandwidth allocation rather than the scheduling heuristics used. We proposed the approach OS-SC which produces high quality of the schedules with low overhead. Second, we described the *LRU with Love/Hate hints* (LH) cache replacement policy, which exploits hints easily obtained from *RxW* to effectively keep hot pages in the cache. We showed that the performance of LH is very close to the idealized algorithm that makes use of perfect knowledge of page access probabilities. 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. Prefetching was found to be helpful in higher-latency situations, as would arise when obtaining data from remote sites across the Internet. These results demonstrate that while it is tempting to look at data broadcast merely as a communications problem, an essential component for large-scale data broadcast systems is data management, such as optimizing the location and flow of large amounts of data.

References

[AF98] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. In *Proc. of IEEE INFOCOM*, March 1998.

[AF99] D. Aksoy and M. Franklin. RxW: A scheduling approach to large scale on-demand broadcast. *IEEE/ACM Transactions on Networking*, (6):846–861, Dec. 1999.

[Alo00] Alohanet. SkyDSL. www.alohanet.com/, 2000.

[AM98] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proc. ACM/IEEE Mobicom*, 1998.

[BCF+99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. INFOCOM*, 1999.

[BPS99] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *Proc. VLDB*, 1999.

[DAW86] H.D. Dykeman, M. Ammar, and J.W. Wong. Scheduling algorithms for videotex systems under broadcast delivery. In *IEEE ICC*, 1986.

[Dir96] Hughes network systems, DirecPC homepage. www.direcpc.com, 1996.

[DW88] H.D. Dykeman and J.W. Wong. A performance study of broadcast information delivery systems. In *Proc. IEEE INFOCOM*, 1988.

[FD95] C. S. Freedman and D. J. DeWitt. The SPIFFI scalable video-on-demand system. In *ACM SIGMOD*, San Jose, CA, 1995.

[FZ98] M. Franklin and S. Zdonik. Data in your face: Push technology in perspective. In *Proc. ACM SIGMOD*, June 1998.

[JS94] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. VLDB*, 1994.

[Knu81] D. Knuth. *The Art of Computer Programming - Volume III.* Addison-Wesley, 1981.

[OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. ACM SIGMOD*, pages 297–306, 1993.

[ORS96] B. Ozden, R. Rastogi, and A. Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *IEEE Int. Conf. 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. SOSP*, 1995.

[PZ91] M. Palmer and S. Zdonik. Fido: A cache that learns to fetch. In *Proc. VLDB*, Sep. 1991.

[ST97a] C.J. Su and L. Tassiulas. Broadcast scheduling for distribution of information items with unequal length. In *Proc. Conf. on Information Sciences and Systems*, Kobe, 1997.

[ST97b] C.J. Su and L. Tassiulas. Broadcast scheduling for information distribution. In *Proc. IEEE INFOCOM*, 1997.

[Sta00] Starband. High speed sattellite Internet service. www.starband.com, 2000.

[THP01] P. Triantafillou, R. Harpantidou, and M. Paterakis. High performance data broadcasting: A comprehensive system's perspective. In *Proc. 2nd Int. Conf. on Mobile Data Management*, Hong Kong, Jan. 2001.

[Tri82] K.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 staging problem in heterogeneous networking environment. In *Proc. Heterogenous Computing Workshop*, March 1998.

[VH96] N.H. Vaidya and S. Hameed. Data broadcast in assymetric wireless environments. In *Proc. 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. WOSBIS*, 1997.