

# RxW: A Scheduling Approach for Large-Scale On-Demand Data Broadcast

Demet Aksoy and Michael Franklin

**Abstract**—Broadcast is becoming an increasingly attractive data dissemination method for large client populations. In order to effectively utilize a broadcast medium for such a service, it is necessary to have efficient, on-line scheduling algorithms that can balance individual and overall performance, and can scale in terms of data set sizes, client populations, and broadcast bandwidth. We propose an algorithm, called RxW, that provides good performance across all of these criteria and that can be tuned to trade off average and worst case waiting time. Unlike previous work on low overhead scheduling, the algorithm does not use estimates of the access probabilities of items, but rather, it makes scheduling decisions based on the current queue state, allowing it to easily adapt to changes in the intensity and distribution of the workload. We demonstrate the performance advantages of the algorithm under a range of scenarios using a simulation model and present analytical results that describe the intrinsic behavior of the algorithm.

## 1 Introduction

Advances in telecommunications, internetworking, and mobile computing have enabled the development of large-scale data dissemination applications that provide information to vast numbers of users distributed around the world. Examples of such applications include election result servers, sporting event kiosks, news providers, and software distribution, to name a few. The World Wide Web has provided a universal platform for developing dissemination-based applications, but all too frequently web-based systems fail to meet the high user demands placed on them, particularly under peak loads. The result is unacceptably slow response and poor availability. As the scale of such systems in terms of the number of users and the amount of information continues to grow, these problems become even more critical.

Technology improvements in delivery mechanisms such as direct broadcast satellite, cable, cellular and even traditional telephone networks are providing high bandwidth delivery channels to homes, offices, and mobile users. The increasing availability of high bandwidth links, however, is only a partial solution to scalability problems, as improved communications bandwidth does not ease the burden on overloaded servers.

For dissemination-based applications unicast data delivery as used by current Web servers is wasteful of resources.

With unicast, a data item must be transmitted for each client that requests it, so the load on the server and the network increases with every client. In contrast, data broadcast can satisfy the needs of potentially many clients using a single transmission. Furthermore, the emerging infrastructure for large-scale data delivery is in many cases organized in a way that supports broadcast. For example, satellite and cable technologies provide high, shared bandwidth from servers to clients. Such an arrangement lends itself well to broadcast. As a result, there is increasing commercial and research interest in data broadcast.

In this paper, we examine *on-demand* broadcast, where a large and dynamic client population requests data items from an information server which broadcasts data items to the clients based on these requests. Such an arrangement is *pull-based* (as opposed to *push-based*) because the transmission of data is initiated due to client requests rather than based on profiles or subscriptions [15].

A key consideration in the design of an on-demand broadcast system is the algorithm used to schedule the broadcast. The challenge is to devise a scheduling algorithm that provides good average and worst case performance, scales well (in terms of increasing request arrival rates, database sizes, *and* bandwidth), and is robust in the presence of typical environmental changes. Previous studies of broadcast scheduling algorithms (e.g., [13, 26, 25, 24]) have failed to address one or more of these issues.

This paper makes three main contributions. First, we explicitly identify the performance and scalability criteria that must be met by scheduling algorithms for large-scale on-demand broadcast environments, and describe how previous algorithms fail to meet one or more of them. Second, we develop a parameterized scheduling algorithm, called *RxW*, that performs well for all of these criteria and can be tuned to focus on the needs of a particular application or system. *RxW* is robust to changes in the client population and workload because it makes scheduling decisions based on the current request queue state rather than depending on estimates of data access probabilities. Third, we present an analytical study along with detailed simulation results and sensitivity analyses to back up these claims.

Our work is focused on systems that disseminate relatively small (i.e., tens of KBytes), distinct objects such as web pages or database pages. A motivating example of such a system is a broadcast-based web proxy server where clients are Internet browsers (or other proxy servers) that request web pages. Such a system could conceivably

An earlier version of this paper appears in the proceedings of the IEEE INFOCOM Conference, San Francisco, CA, March 1998. This work has been performed while the authors were with the Department of Computer Science and the Institute of Advanced Computer Studies at the University of Maryland and has been partially supported by the NSF under grant IRI-9501353, by Rome Labs agreement number F30602-97-2-0241 under DARPA order number F078, and by research grants from Intel and NEC.

be used by millions of clients and could provide access to millions of data items, and therefore, the issues of performance, scalability, and robustness are paramount.

The remainder of this paper is structured as follows. In Section 2 we give a brief description of the problem and define the important criteria for evaluating scheduling algorithms for large-scale broadcast. We then describe how previously proposed algorithms measure up to these criteria in Section 3. In Section 4 we describe the  $RxW$  scheduling algorithm and its approximation. In Section 5 we present an analytical treatment of the algorithm. Section 6 presents an evaluation of the various approximation settings in terms of their performance, scalability, and robustness to workload changes. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

## 2 Background

### 2.1 Environment and Assumptions

We begin by presenting a simple satellite-based broadcast scenario to motivate the scheduling problem being addressed. In this scenario (depicted in Figure 1) there is a single server and a large population of clients. Two independent networks are used: a terrestrial network over which clients send requests to the server, and a satellite downlink over which the server broadcasts data to the clients. Such an arrangement is similar to Hughes Network System’s DirecPC architecture [12] and other satellite data services.

When a client needs a data item (e.g., a web page or database object) that it cannot find locally, it sends a request for the item to the server. These requests are queued up at the server upon arrival. The server repeatedly chooses an item based on these requests, broadcasts it over the satellite link, and removes the associated request(s) from the queue. In our system, clients monitor the broadcast to receive the requested data.

Similar to previous work on broadcast scheduling [13, 26, 24, 25] we make several assumptions about the environment: We assume that there is a single broadcast channel that is monitored by all clients and that the channel is fully dedicated to the data broadcast (i.e., the data server can use the entire bandwidth). Clients continuously monitor the broadcast after they make a request. We do not consider the effects of transmission errors, so that when an item is broadcast it is received by all clients that are waiting for it. We focus on the case where the data items are fixed-length *pages*, such as database or memory pages. The use of fixed-length pages simplifies the algorithm descriptions and analysis; Recent work in broadcast scheduling has shown how to extend scheduling algorithms to incorporate variable-length items [25, 24].

Each page is broadcast in a single burst; The time it takes to broadcast a page is referred to as a *broadcast tick*, and we use such ticks as a unit of performance measurement. The use of ticks emphasizes that our results apply to systems of many different scales. For example, in a system with 16 KB pages and 1 Mb/sec downstream bandwidth a

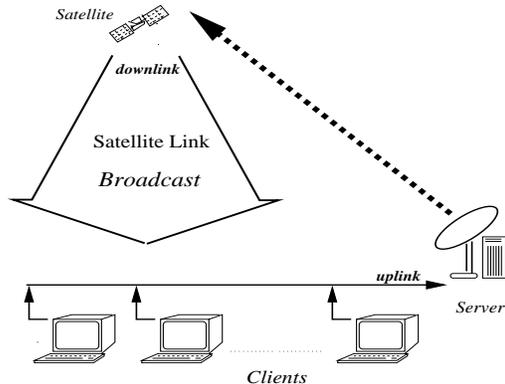


Figure 1: Example Data Broadcasting Scenario

broadcast tick would be approximately 0.13 seconds, while if the bandwidth was 100 Mb/sec a tick would be 1.3 milliseconds. Regardless of these absolute values, if algorithm A is shown to be 50% faster than algorithm B in terms of ticks, then as long as both algorithms are sufficiently fast to allow full utilization of the broadcast bandwidth, the same relationship will hold in terms of absolute time as well.

Finally, it is important to emphasize that our focus is on data servers rather than multimedia servers. In particular, we do not address continuous media delivery as provided by Video-On-Demand servers [11, 4]. Such systems, while sharing many of the same challenges as broadcast data servers, have different properties due to timing and ordering constraints and much larger broadcast granularities. The smaller granularity of broadcast in data servers results in having more items to track and more frequent requests. The lack of explicit ordering among data items results in less predictable delivery requirements. Thus, while there is some commonality between the two types of broadcast systems, there are sufficient differences to warrant the development of different solutions.

### 2.2 Performance Issues

Given the application environment described in the preceding section, we can now state our criteria for evaluating scheduling algorithms for large-scale on-demand data broadcast. We divide the criteria into three groups: responsiveness, scalability, and robustness.

#### 2.2.1 Responsiveness

The success of a scheduling algorithm is ultimately determined by its ability to get requested data to the clients quickly. There are two metrics of importance in this regard. First, we define *average wait time* as the average time from the instant that a client request arrives at the server, to the time that the item is broadcast. Previous work on broadcast scheduling has focused almost exclusively on such average performance. For interactive systems, however, it is also important to ensure that the scheduling algorithm does not induce *starvation* of requests for unpopular items. Thus, we also measure the *worst case wait time*, which is

the maximum amount of time that a client request waits in the service queue before being satisfied.

There are two factors that determine the responsiveness of a scheduling algorithm. The first is the *quality* of the schedule produced. It has previously been shown that the optimal allocation of broadcast bandwidth for fixed-length data items should be in proportion to the square roots of their probability of access [7]. A good scheduler should therefore approach such an allocation. The second factor, however, is the *decision overhead*, that is, the cost of making a scheduling decision. In order to make full use of the broadcast bandwidth, the time required to make a scheduling decision must be less than the length of a broadcast tick. A scheduler that makes decisions more slowly will stall the broadcast, resulting in unused bandwidth, thereby wasting a critical shared resource.

Most previous work in on-demand scheduling has focused on scheduling quality, making the implicit assumption that all of the algorithms had sufficiently low overhead. The point at which such overhead begins to hurt overall performance is highly situation-dependent, being impacted by among other things, the ratios among broadcast bandwidth, data item size, server processor speed, and database size. Thus, in this paper, we follow the approach of ignoring overhead when reporting response times, but also provide measurements of the overheads incurred by the various scheduling approaches under various scenarios.

## 2.2.2 Scalability

While responsiveness is a primary consideration, it is also crucial that good performance be provided over a wide range of environments. In particular, it is necessary for the scheduler to perform well as the problem grows in the following dimensions:

*Request Arrival Rates* - When a new request arrives at the server, the server must decide whether or not to place and/or update an entry in the service queue for the requested item. The speed of such *request processing* limits the rate at which requests can be processed by the server. Since a high-bandwidth broadcast channel is utilized most effectively when there are a large number of clients, the system must be able to handle heavy request traffic.

*Database Size* - Because a single broadcast of an item satisfies all outstanding requests for that item, the size of the service queue managed by the server is typically proportional to the number of items with outstanding requests rather than the number of individual requests. Therefore, as the amount of data that is available for dissemination increases, it is important that the scheduling algorithm decision overhead remains reasonable.

*Broadcast Rate* - Broadcast technologies are continually being improved to provide higher bandwidth. As the bandwidth is increased, the amount of time allowed to make a scheduling decision is decreased. Thus, the scheduler must have low overhead in order to avoid becoming a bottleneck.

## 2.2.3 Robustness

In order to achieve the goals of responsiveness and scalability, scheduling algorithms typically employ approximations and/or heuristics. Such techniques must not cause the algorithm to perform poorly if the workload or the environment changes either abruptly or gradually. In this study, we examine the robustness of scheduling algorithms to changes in access patterns and request arrival rates.

## 3 Previous Algorithms

In this section, we describe existing algorithms and discuss their limitations with respect to the criteria that were outlined in the preceding section. As stated in the Introduction, several algorithms for on-demand broadcast scheduling have been proposed previously. Dykeman et al. [13] studied on-line scheduling algorithms, and were the first to point out that traditional FCFS scheduling would provide poor average wait time for a broadcast environment when the access distribution for data items was non-uniform. They proposed several algorithms aimed at providing improved performance. The algorithms studied in [13] (and later in [26]) are the following:

- **First Come First Served (FCFS)**: broadcasts the pages in the order they are requested. To avoid redundant broadcasts, requests for pages that already have entries in the queue are ignored.
- **Most Requests First (MRF)**: broadcasts the page with the maximum number of pending requests.
- **Most Requests First Lowest (MRFL)**: Similar to MRF, but breaks ties in favor of the page with the lowest access probability.
- **Longest Wait First (LWF)**: selects the page for which the *total* waiting time of pending requests is the largest (i.e., sum of all request's waiting time).

In Figure 2 we show the average waiting time (in broadcast ticks) for a workload with a database of 10000 pages. Client requests for pages are generated using Zipf distribution (with  $\theta=1$ ) [20]. The request inter-arrival time is determined by an exponential distribution, the mean of which is varied along the x-axis (expressed in requests per broadcast tick). The results were generated using the simulation environment that is described in Section 6. As in [13], the overheads associated with running the scheduling algorithm at the server are not modeled here.

As can be seen in the figure, the best performance overall in this case is provided by LWF. As would be expected the average wait time increases for all algorithms as more requests are introduced, but perhaps less predictably, the average response time eventually levels off and becomes insensitive to additional load. At this point, the remaining algorithms are approximately 2.5 times slower than LWF. Unfortunately, a straightforward implementation of LWF is not practical for a large system, as at each broadcast tick, it would recalculate the total accumulated wait time for *every* page with pending requests in order to decide which

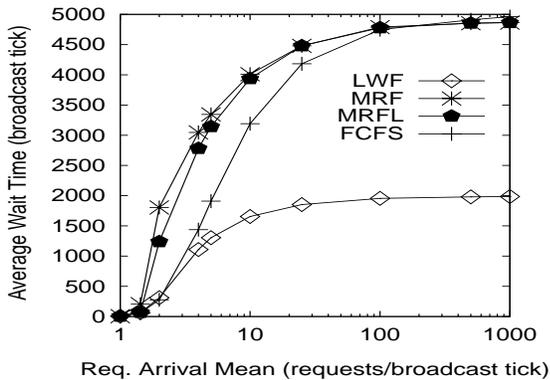


Figure 2: Algorithms of Dykeman et al.: Average Wait

page to broadcast next. For a high-bandwidth system with a large database, such a scheduling algorithm would likely to become a bottleneck.<sup>1</sup> In contrast, the other algorithms examined by Dykeman et al. lend themselves easily to lower overhead implementations.

The results of Figure 2 agree with those of [13, 26] except for two key points. First, the earlier work did not investigate the performance of the algorithms under very high loads, so it did not identify the flattening of the performance for all of the algorithms under high load. This behavior is a key property for broadcast based servers, and is explained in Section 5. Second, the performance of MRFL in the earlier study was seen to provide performance between that of FCFS and LWF, whereas in our experiments it was seen to perform worse than FCFS. The difference in these results stems from the fact that the earlier study was performed using a very small database (100 items, compared to 10000 in Figure 2). As the size of the database increases, the probability of having a tie for the largest number of requests diminishes. Without ties, MRFL degenerates to MRF.

The poor performance of MRFL for large systems has also been shown by Su and Tassiulas [24]. They proposed an alternative algorithm, called PIP-0.5 (Priority Index Policy), that performs as well as LWF in terms of average wait time. Unlike LWF, PIP-0.5 can be implemented to run with  $O(1)$  complexity, by carefully organizing the service queue and slightly increasing the work that must be done when a request arrives at the server. Even with this optimization, however, PIP-0.5 falls short of our performance criteria, because it is based on estimates of the probability of access for each item. As a result, its usefulness is limited to fairly stable environments where those probabilities do not often change significantly. Furthermore, the history mechanism that must be employed to obtain such probability estimates can result in additional overhead, particularly for very large data sets. Algorithms based on access probabilities and broadcast histories have also been proposed by Dykeman et al. [14] and Vaidya and

<sup>1</sup>In our implementation of LWF, we found that using one processor of a DEC Alpha 2100 4/275 server and assuming a broadcast bandwidth of 155 Mbps, the LWF scheduling decision time became a bottleneck with a database size of 5543 8KByte pages.

Hameed [25]. These algorithms have similar performance to the PIP-0.5 algorithm, and also share that algorithm’s limitations in terms of robustness to changing workloads.

## 4 Parameterized $RxW$ Algorithm

We have developed a new broadcast scheduling algorithm, called  $RxW$ , which is a practical, low-overhead, scalable approach that provides excellent performance across a range of scenarios. We begin the presentation of the algorithm by describing the intuition behind it. We then describe three forms of the algorithm: 1) an exhaustive search-based approach that finds the page with the maximal  $R \times W$  value; 2) a lower-overhead implementation, that prunes the search space, but also finds the maximal  $R \times W$ -valued page; and 3) an approximate version that can be adjusted to tradeoff scalability, average case, and worst case waiting time.

### 4.1 Intuition

The results described in Section 3 demonstrated that the low overhead algorithms investigated by Dykeman et al., have poor average case performance compared to the higher-overhead LWF algorithm. As described in [5], further analysis showed that MRF provides the lowest waiting time for hot pages, but its performance for cold pages is by far the worst. MRF chooses the page with the highest number of outstanding requests, so that requests for infrequently accessed pages must wait until sufficient requests have arrived. In fact, MRF is not a starvation-free algorithm; it is quite possible that a request for a very cold page is never satisfied. In contrast, FCFS allocates the same bandwidth to all requested pages regardless of their popularity. The success of LWF is due to the fact that it provides more bandwidth to hot pages while avoiding starvation of cold pages.

### 4.2 The Exhaustive $RxW$ Algorithm

Based on the above observations we developed the  $RxW$  algorithm, which combines the benefits of MRF and FCFS in order to provide good performance for both hot and cold pages, while ensuring scalability by having low overhead. Intuitively,  $RxW$  broadcasts a page either because it is popular or because it has at least one long-outstanding request. At each scheduling decision the  $RxW$  algorithm chooses to broadcast the page with the maximal  $R \times W$  value where  $R$  is the number of outstanding requests for that page and  $W$  is the time that the oldest outstanding request for that page has been waiting. In this section, we introduce an exhaustive version of the algorithm; In the subsequent sections we describe implementation techniques that lower its overhead.

The exhaustive  $RxW$  algorithm is an  $O(N)$  algorithm that has overhead similar to LWF. It maintains a structure containing a single entry for each page that has outstanding requests. This structure is hashed on page identifiers ( $PID$ ). In addition to a  $PID$  each entry contains  $R$ ,

the count of the number of outstanding requests,  $1stARV$ , the arrival time of the oldest outstanding request for the page. The  $W$  value of a page can then be computed as  $W = clock - 1stARV$  where  $clock$  is the current time. In the algorithm, all times are represented in broadcast ticks, rather than as wall-clock time. The algorithm works as follows:

When a request arrives at the server – a hash lookup is performed on the PID of the requested page. If an entry already exists, then the  $R$  value of that entry is simply incremented. If no entry is found (i.e., there are currently no outstanding requests for the page), then a new entry is created with  $R$  initialized to 1, and  $1stARV$  initialized to the current time in broadcast ticks (i.e., the number of items broadcast so far).

For each broadcast decision – all entries in the request structure are examined and the page with the largest  $R \times W$  value is chosen to be broadcast. The entry for this page is then removed from the service queue.

### 4.3 Pruning the Search Space

The search overhead of  $R \times W$  can be reduced by performing more work at request arrival time to keep the request information better organized. Thus, we have developed a pruning version of  $R \times W$  that examines fewer entries while finding the maximal  $R \times W$ -valued page. This algorithm uses two sorted lists that are threaded through the request structure: 1) the  $W$ -List, ordered by increasing  $1stARV$  value, similar to the queue maintained by FCFS; and 2) the  $R$ -List, which is ordered by descending  $R$  value. As shown in Figure 3, these two lists are maintained as doubly-linked lists that are threaded through the request structure.

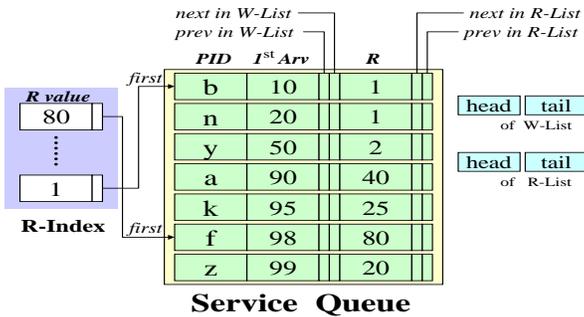


Figure 3: The  $R \times W$  Service Queue Data Structures

The  $W$ -List has very low maintenance overhead, as once an entry is added to the list, it is not moved until the corresponding page is broadcast, regardless of how many requests arrive for that page. The  $R$ -List, however, is much more dynamic. Every time a request arrives for a page, its entry must be moved in the  $R$ -List. Thus, an additional structure, called the  $R$ -Index is used to facilitate  $R$ -List maintenance. The  $R$ -Index has an entry for each distinct  $R$  value, with a pointer to the first entry in the  $R$ -List that has  $R$  outstanding requests. When a request arrives for an item that already has an entry, the  $R$ -Index is used to

quickly locate the place in the  $R$ -List to where the entry must be moved after its  $R$  value is incremented.<sup>2</sup> The  $R$ -Index makes request processing a constant time operation, thereby removing a potential bottleneck to handling large request arrival rates.

The pruning technique starts by examining the entry at the top of the  $R$ -List and setting  $MAX$ , the maximum  $R \times W$  value seen so far, to the  $R \times W$  value of that entry. The  $W$  values of interest can then be restricted using  $R'$ , the  $R$  value of the *next* entry in the  $R$ -List. Since the  $R$ -list is sorted in descending order, it is known that for any unexamined entry to have an  $R \times W$  value greater than  $MAX$ , it must have a  $W$  value satisfying the inequality:

$$W > \frac{MAX}{R'}$$

Thus, a limit can be placed on  $1stARV$  which is,

$$limit(1stARV) = clock - \frac{MAX}{R'}$$

Since the  $W$ -List is sorted in ascending order, this limit effectively truncates the  $W$ -List. That is, it defines a point in the  $W$ -List below which it is known that no entry can exceed the current maximal  $R \times W$  value.

Next, the entry at the top of the  $W$ -List is examined and a  $limit(R)$  is calculated using  $W'$ . The algorithm then keeps on alternating between the two lists, updating  $MAX$  when an entry with an  $R \times W$  value greater than  $MAX$  is encountered, and incrementing the limits if possible. The algorithm stops when the limit is passed on one of the lists or when the search reaches the halfway point on both lists. At this point,  $MAX$  is known to be the maximal  $R \times W$  value for all pages so the page with that value is chosen for broadcast. In the analysis and performance study that follow, we will use the term “ $R \times W$  algorithm” to refer to this pruning variant.



Figure 4: Pruning the Search Space

An example of the pruning technique is shown in Figure 4. In the example, the  $R$ -List and  $W$ -List are shown as two separate lists (for ease of explanation) and the current  $clock$  value is 100 ticks. First the entry for page  $f$  (the top

<sup>2</sup>When an item is broadcast, its entry is removed from the service queue and the  $W$ -List and  $R$ -List are patched using the doubly-linked lists. If the entry removed was the first with its particular  $R$  value, then the  $R$ -Index is updated to point to the next entry with that value, if any.

of the *R-List*) is examined resulting in *MAX* being set to 160 and *limit(1stARV)* being set to 96. Next, the entry for page *b* (the top of the *W-List*) is checked.  $R \times W$  of *b* is less than *MAX* (90 vs 160) so *MAX* is left unchanged, but *limit(R)* is set to 2 as shown in the figure. The algorithm then checks page *a*, which has an  $R \times W$  value of 400, and so *MAX* is updated to 400, and *limit(1stARV)* is set to 84. The algorithm continues searching until page *y* is examined, at which point the limit on the *W-List* is reached and the algorithm stops. In this example, page *a* has the highest  $R \times W$  value, so it is chosen to be broadcast.

#### 4.4 Approximating $R \times W$

The pruning technique described in the previous section is indeed effective in reducing the search space. For example, for our Zipf-based workload, it results in a 73% reduction in entries examined as will be discussed in Section 6.2.3. While such a substantial savings is helpful, it is not always sufficient to keep the scheduling overhead from ultimately becoming a limiting factor as the system is scaled to the large-scale applications that will be enabled by the national and global broadcasting systems being deployed.

Based on insight gained from early experiments with the *RxW* algorithm, we realized that the scheduling overhead can be reduced dramatically by backing off from the requirement of maximality for the  $R \times W$  value. Thus, we have developed an *approximate*, parameterized variant of *RxW* that allows the search space to be reduced further, at the possible expense of making suboptimal broadcast decisions. By varying a single parameter, the algorithm can be tuned from having the same behavior as the *RxW* algorithm described so far, to being a *constant time* approach.

The parameterized version of *RxW* is based on two insights. First, we found that with highly skewed access patterns (as would be expected in many dissemination-oriented applications), the page with the maximum  $R \times W$  value is typically found very near the top of at least one of the two lists. This behavior is due to the large differences between the top and bottom values in the sorted lists that arises in workloads of high and non-uniform request rates on large databases. As a result, even the pruning-based *RxW* algorithm can spend substantial resources examining entries after it has already encountered the maximum-valued entry. The second insight is that given a static workload (i.e., in terms of request arrival rate and access probability distribution) the average  $R \times W$  value of the page chosen for broadcast typically converges to some constant value. This latter insight is exploited to create a *self-adapting* approximation algorithm based on the  $R \times W$  value of the most recently broadcast page. We take care, however, to ensure that the approximation works well even in the presence of a changing workload.

The approximation algorithm requires a single parameter called  $\alpha$ , which can be set to any value 0 or greater.<sup>3</sup>

<sup>3</sup>Typically the  $\alpha$  parameter will be set to a value between 0 and 1. Larger values can also be used. In the limit, setting  $\alpha$  to  $\infty$  would result in behavior identical to that of the regular *RxW* algorithm.

To find the next page to broadcast the request structure is searched as in the regular (pruning) *RxW* algorithm, but rather than searching for the page with the *maximal*  $R \times W$  value, the algorithm chooses the *first* page it encounters whose  $R \times W$  value is greater than or equal to  $\alpha \times \text{thresh-}old$ . If no such page is found, then the algorithm acts like the regular *RxW* algorithm and returns the page with the maximum  $R \times W$  value. The self-adapting average, *threshold*, is initially set to 1 and then on each broadcast tick is recomputed as

$$\text{threshold}(t + 1) = \frac{\text{threshold}(t) + \text{lastRxW}(t)}{2}$$

where *lastRxW(t)* is the  $R \times W$  value of the page that was broadcast at tick *t*.

The setting of the  $\alpha$  parameter determines the performance tradeoffs between average waiting time, worst case waiting time, and scheduling overhead. The smaller the value of the parameter, the fewer entries are likely to be scanned. At an extreme value of 0, the algorithm simply compares the top entry from both the *R-List* and the *W-List* and chooses the one with the highest  $R \times W$  value. In this case, the complexity of making a scheduling decision is reduced to  $O(1)$ , ensuring that broadcast *scheduling* will not become a bottleneck regardless of the broadcast bandwidth, database size, or workload intensity.

In Section 6, we examine the performance tradeoffs of several settings of the  $\alpha$  parameter. First however, we present an analytical treatment of *RxW* and identify several intrinsic properties of the schedules it produces.

## 5 Upper Limit on Average Wait

In Section 3 we observed that as the request arrival rate is increased, the average waiting time of all algorithms eventually levels out at an algorithm dependent limit. This behavior is in contrast to unicast where the average waiting time in such a case would grow asymptotically. In this section we show that broadcasting is inherently scalable in terms of increased request arrival rates (the practical impact of client population growth) by analyzing the expected waiting time when the *RxW* algorithm is employed. These results also analytically show that *RxW* produces higher-quality schedules than MRF.

### 5.1 Time-Dependent $R \times W$ Priority Queue

The expected waiting time for a specific page when *RxW* is employed can be determined through the use of a priority queuing model [19]. In Section 4.2 we explained how the service queue is implemented using a single entry for each unique page of the database with outstanding requests. Here we refer to the queue entry that represents all individual requests currently outstanding for page *i* as a logical *page-i* request. Such logical requests will be serviced and removed from the queue when the requested page is broadcast. The priority of a logical *page-i* request at a given time is equal to the  $R \times W$  value of the entry at that time.

Assuming Poisson arrivals, the priority ( $R \times W$  value) of a *page-i* entry is a function of time, denoted by  $R_i \times W_i(t)$ :

$$R_i \times W_i(t) = \lambda p_i (t - t_i)^2 \quad (1)$$

where  $\lambda$  is the client request arrival rate,  $p_i$  is the probability that a request is for page  $i$ , and  $t_i$  is the arrival time of the request that caused the *page-i* entry to be created. Equation 1 is a trivial formulation since the expected number of individual request arrivals for page  $i$  in the time interval  $[t - t_i]$  is  $\lambda p_i (t - t_i)$ . We simply multiply the expected number of requests by  $(t - t_i)$ , the waiting time of the request that caused the creation of the entry.

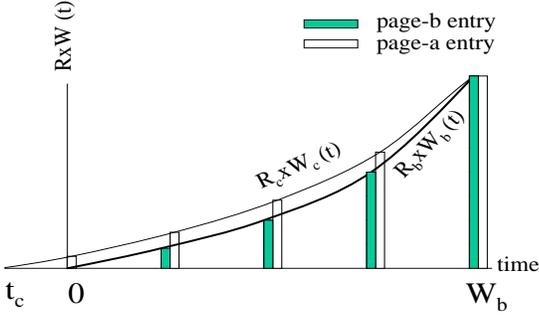


Figure 5: Request Entries Already Queued

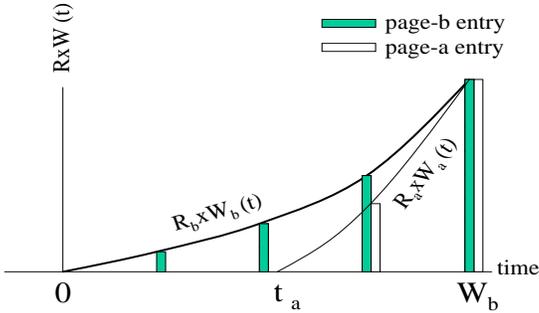


Figure 6: Request Entries Created after page-b Entry

In this time-dependent priority queuing model, we consider a specific queue entry, *page-b*, and examine its waiting time behavior relative to other entries. The waiting time of a *page-b* entry,  $W_b$ , depends on the number of entries that will be serviced before the *page-b* entry, in other words the number of pages that will be broadcast before page  $b$ , such that:

$$W_b = W_o + \mu L_b + \mu M_b \quad (2)$$

where  $W_o$  is the mandatory wait for the completion of an already initiated page broadcast,  $\mu$  is the service time, (i.e., one broadcast tick),  $L_b$  is the expected number of pages that will be broadcast before page  $b$  for which entries were already in the service queue when the *page-b* entry was created, and  $M_b$  is the expected number of pages that will be broadcast before page  $b$  even though their initial request

arrives after the *page-b* request was created. In the following paragraphs we derive the expected values of  $L_b$  and  $M_b$  and then combine the results in Section 5.2.

First, we calculate  $L_b$ , the number of pre-existing queue entries that will be serviced before the *page-b* entry. For Poisson arrivals, all pre-existing entries for more popular pages will typically be serviced before the *page-b* entry. An entry for a less popular page, on the other hand, can compete with the *page-b* entry only if it was created early enough before the creation of the *page-b* entry. For example, Figure 5 shows the evolution of the priorities of two queue entries over time. The origin of the graph is the time at which the *page-b* entry is created. Time prior to the creation of the *page-b* entry is shown on the negative x-axis. The priority values are plotted as a continuous function of time and the discrete  $R \times W$  values are shown with bars drawn at each broadcast tick. In Figure 5, page  $c$  is less popular than page  $b$  and thus, it can accumulate a higher priority before time  $W_b$  only if its initial queuing time is before  $t_c$ . We can find the value of  $t_c$  using the equation:

$$Q_c(W_b) = Q_b(W_b) \quad (3)$$

or

$$\lambda p_c (W_b + t_c)^2 = \lambda p_b (W_b)^2 \quad (4)$$

The expected number of entries included in  $L_b$  for less popular pages is calculated by solving Equation 4 (see Appendix A). The total for  $L_b$  is expressed as:

$$L_b = \sum_{i=0}^{N-1} L_{ib} \quad (5)$$

where  $N$  is the number of pages in the database, and:

$$L_{ib} = \begin{cases} \Lambda_i W_i \sqrt{\frac{p_i}{p_b}} & \text{if } i \text{ is less popular than } b \\ \Lambda_i W_i & \text{otherwise} \end{cases} \quad (6)$$

where  $\Lambda_i$  is the rate at which a *page-i* entry is created,  $W_i$  is the expected waiting time for a *page-i* entry and  $p_i$  is the access probability of page  $i$ .

We next calculate  $M_b$ , the expected number of *page-i* entries that will be serviced before the *page-b* entry even though they are created after the *page-b* entry is created. In this case, only entries for pages more popular than  $b$  can compete for service. For example, as shown in Figure 6, a *page-a* entry (such that page  $a$  is more popular than page  $b$ ), can reach the priority level of the *page-b* entry before time  $W_b$  only if it is created before  $t_a$ . We can calculate the value of  $t_a$  using:

$$Q_a(W_b) = Q_b(W_b) \quad (7)$$

or

$$\lambda p_a (W_b - t_a)^2 = \lambda p_b (W_b)^2 \quad (8)$$

We solve equation 8 for  $t_a$ , and

$$t_a = W_b \left(1 - \sqrt{\frac{p_b}{p_a}}\right)$$

Thus the expected number of entries that will be serviced before the *page-b* entry even though they are created afterwards is:

$$M_b = \sum_{i=0}^{N-1} M_{ib} \quad (9)$$

where

$$M_{ib} = \begin{cases} \Lambda_i W_b (1 - \sqrt{\frac{p_b}{p_i}}) & \text{if } i \text{ is more popular than } b \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

## 5.2 The Limiting Behavior of Waiting Time

In this section, we combine Equations 5 and 9 to derive the expected waiting time of the *page-b* entry. Assuming that all pages are ordered by decreasing popularity, such that page 0 is the hottest page and page  $N-1$  is the coldest, the expected waiting time for a *page-b* entry is:

$$W_b = W_o + \mu \left( \sum_{i=0}^{b-1} \Lambda_i W_i + \sum_{i=b+1}^{N-1} \Lambda_i W_i \sqrt{\frac{p_i}{p_b}} \right) + \mu \left( \sum_{i=0}^{b-1} \Lambda_i W_b (1 - \sqrt{\frac{p_b}{p_i}}) \right)$$

where  $\Lambda_i$ , the rate at which *page-i* entries are created, is

$$\Lambda_i = \frac{1}{W_i + \frac{1}{\lambda p_i}} \quad (11)$$

The creation of a new *page-i* entry is only possible after the broadcast of page  $i$ . Therefore, the expected time between two successive *page-i* creations is the sum of the expected time a *page-i* entry spends in the queue and the time until the next request for page  $i$  that results in the creation of a new *page-i* entry (i.e., after the previous *page-i* entry is serviced). Note that  $\Lambda_i$  is bound to  $\frac{1}{W_i}$  for large values of  $\lambda$ . The expected waiting time of a *page-b* entry, therefore, is equal to:

$$W_b = \frac{W_o + \sum_{i=b+1}^{N-1} \rho_i \sqrt{\frac{p_i}{p_b}} W_i + \sum_{i=0}^{b-1} \rho_i W_i}{1 - \sum_{i=0}^{b-1} \rho_i [1 - \sqrt{\frac{p_b}{p_i}}]} \quad (12)$$

where  $\rho_i$  is the utilization of the bandwidth by page  $i$  broadcasts, which is equal to  $\mu \Lambda_i$  and bounded by  $\frac{\mu}{W_i}$  for heavy request rates.

There are two observations that follow this analysis. First, for heavy request traffic, the expected value of waiting time is independent of request arrival rates. This explains the inherent scalability in terms of request arrival rates. The only factor that determines the average waiting time is the relative popularity of pages in the simple form of square root ratios. In general, the leveling off behavior of average waiting time is independent of the scheduling algorithm used as we have already observed in Figure 2. An intuitive explanation of this phenomenon is that for heavy request

rates, an increase in the number of requests implies a similar increase in the number of requests that can be satisfied by a single broadcast of an item. This increase in the service rate balances that of the request arrival rate.

The second observation is that the dependency on access probabilities in Equation 12 is in the form of a ratio of square roots. It has previously been shown that the optimal bandwidth allocation for push-based broadcast (i.e., broadcast without explicit requests) is by the ratio of square root of page access probabilities [7]. Thus, we expect the *RxW* algorithm to have good performance in terms of overall responsiveness. It is important to note, however, that *RxW* uses the current queue state for making scheduling decisions, rather than depending on estimates of access probability. Thus, the *RxW* algorithm can be robust to changes in the workload.

In Appendix B we apply a similar analysis to MRF. The expected waiting time formulation derived there is structurally similar to Equation 12. The only difference is that the equation for MRF scheduling has a plain ratio between the probabilities  $p_i$  and  $p_b$ , while *RxW* has a square root-based ratio as shown in Equation 12. This implies that MRF favors hot pages more than it needs to and therefore cannot exploit the benefits of broadcast bandwidth to its maximum benefit. Thus, we are able to analytically describe the reason behind MRF's poor performance: namely, over-emphasizing hot pages.

## 6 Experimental Results

### 6.1 Simulation Environment

We performed a detailed set of performance experiments using a simulation model written in CSIM [21]. The model represents an environment similar to that described in Section 2.1. The broadcast channel is modeled as a server with a fixed rate of broadcast. We do not specify an absolute value for this rate, but rather, use *broadcast ticks* as our (abstract) measure of time for all aspects of the simulation. As has been described previously, scheduling overhead is not included in the responsiveness numbers (average and worst case wait times) presented here. This simplification is equivalent to making the assumption that all of the algorithms are able to make scheduling decisions fast enough to keep the broadcast bandwidth fully utilized. As a result, the responsiveness numbers presented for slower algorithms such as LWF may be optimistic. In order to address the overhead issue, we provide detailed measurements of the average number of entries searched per broadcast decision by the various algorithms under different workloads.

In the model, the client population is represented by a single request stream. We use an open system model since our work is aimed at supporting extremely large, highly dynamic client populations, and such client populations cannot be realistically modeled with a closed simulation system. We do not model the costs of using the back-channel for sending requests to the server as these costs will be small for simple page requests.

The results reported here also do not include propagation delays between the clients and the server. While the propagation delay for (terrestrial) uplink requests is expected to be negligible, downlink technologies such as satellites can incur latencies of 100’s of milliseconds. This propagation delay creates a window in which if a request is generated for a page that is “in flight”, the request will be erroneously entered into the service queue at the server even though the request will be satisfied when that page reaches the client(s). We performed several experiments to determine the impact of such propagation delays on the performance of the algorithms. We examined latencies ranging from 1 to 1000 broadcast ticks<sup>4</sup>. The results showed that such latencies made a significant impact on responsiveness only for very small databases (e.g., 100 pages) or in an under-utilized system (e.g., arrival rates of 1 request per tick). For larger databases (e.g., 10,000 pages) and higher request rates (e.g., 10 requests/tick or higher) on which we focus, the impact of such latencies was fairly small. In general, the FCFS algorithm was impacted slightly less than the “smarter” algorithms, but the impact of latencies up to 1000 ticks was less than 5% for all of the algorithms in the cases we studied.

The main parameters and settings for the workloads used in the experiments are shown in Table 1. The client population model generates requests with exponential inter-arrival times with mean  $\lambda$ . The request pattern is shaped with a Zipf distribution [20] which produces access patterns that become increasingly skewed as its  $\theta$  parameter increases from 0 (uniform access probability) to 1 (highly skewed). The requests are distributed over a database containing  $dbsize$  fixed-sized pages. Two additional parameters, called *offset* and *interval*, are used to simulate interest shifts of the client population and the frequency of such shifts, respectively.

For the results presented in the following sections, each data point was obtained over a run of at least 300,000 broadcast ticks. To determine this value, we examined the execution of the various scheduling algorithms using the default parameters and observed that the queue length (in terms of outstanding requests, not the one-per-page service queue entries), and average waiting times stabilized typically after the first 50,000 broadcast ticks. For reporting worst case waiting times we run the simulation longer, ensuring that every page in the database is broadcast multiple times (i.e., that starvation is not occurring).

## 6.2 Responsiveness and Scalability

We begin by examining the responsiveness of several variants of the *RxW* algorithm, and comparing them to the LWF and FCFS algorithms of Dykeman et al. We report results for four variants of *RxW*: The maximal (pruning) *RxW* algorithm, and the approximate algorithm with  $\alpha$  values of 0.90, 0.80, and 0 (referred to as *RxW*.90, *RxW*.80, and *RxW*.0, respectively). We first examine the average

<sup>4</sup>For a system with 16KB pages and 100 Mb/s bandwidth, this ranges from 1.3 milliseconds to 1.3 seconds.

and worst case waiting times in the absence of scheduling overhead, and then explore the scheduling overheads of the various approaches in detail.

### 6.2.1 Average Waiting Time

Figure 7 shows the *average* waiting time for each of the scheduling algorithms as the mean request arrival rate is varied from 1 per tick to 1000 per tick along the x-axis (shown with a log scale). All of the algorithms exhibit similar characteristics here, with the average wait time increasing but ultimately leveling off as the request arrival rate is increased. As discussed in Section 5, this leveling off is an intrinsic behavior of broadcast data delivery to clients with shared interests.

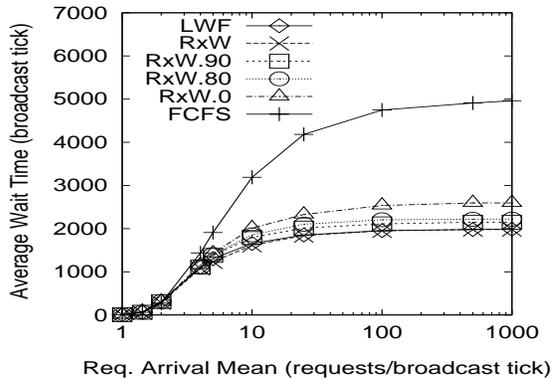


Figure 7: Average Waiting Time

In Figure 7, FCFS has the highest average wait in the entire range as expected. FCFS broadcasts the pages in the order requested, regardless of their popularity, resulting in poor overall bandwidth allocation. LWF and the maximal *RxW* algorithm provide the best performance (*RxW* actually does slightly better for loads between 5 and 50 requests/tick). The good performance of *RxW* in this case demonstrates that as predicted in the analysis of Section 5, the *RxW* heuristic is a reasonable substitute for the total waiting time used by LWF. The average waiting time of *RxW* is 1.98 times better than FCFS at 10 requests/tick and 2.51 times better than FCFS at 1000 requests/tick. As the  $\alpha$  parameter is decreased, the average wait time gradually increases.

As we have explained in Section 4.4 as the approximation parameter is decreased the scheduling overhead decreases at a possible expense of suboptimal decisions. In this experiment, however, the approximate algorithm with an  $\alpha$  value of 0.90 remains less than 10% worse than the maximal algorithm over the entire range. Even the constant-time *RxW*.0 pays a penalty of less than 33% compared to the maximal *RxW* algorithm. For all *RxW* settings, the average wait time is significantly better than that of FCFS in this case. These results are more pronounced when we consider the scheduling overhead studied in Section 6.2.3.

| Symbol          | Description                          | Default | Range       | Unit            |
|-----------------|--------------------------------------|---------|-------------|-----------------|
| $\lambda$       | Mean Req. Arrival Rate (Exponential) | 100     | [1-1000]    | requests/tick   |
| $\theta$        | Request Pattern Skewness (Zipf)      | 1.0     | [0.0-1.0]   | -               |
| dbSize          | Database Size                        | 10000   | [100-50000] | pages           |
| <i>offset</i>   | Shift in Interest                    | 0       | [0-5000]    | number of pages |
| <i>interval</i> | Period of Interest Change            | n/a     | [1-100000]  | broadcast ticks |

Table 1: Workload Parameters and Settings

### 6.2.2 Worst Case Waiting Time

We now turn to the worst case waiting time for the various approaches, which is shown in Figure 8. This figure shows the longest wait measured for *any* request during the simulation run for the same parameter settings used in the previous section.<sup>5</sup> As can be seen in the figure, the ordering of the algorithms for worst case behavior is reversed compared to the average case. FCFS has the lowest worst case waiting time. With FCFS, once a page has been requested, it is guaranteed to be scheduled for broadcast before any other page is broadcast twice. Thus, its worst case behavior is bounded by the number of pages (10,000 in this case). In contrast, the LWF and *RxW* algorithms make no such guarantees — popular pages may be broadcast multiple times while requests for less popular pages wait.

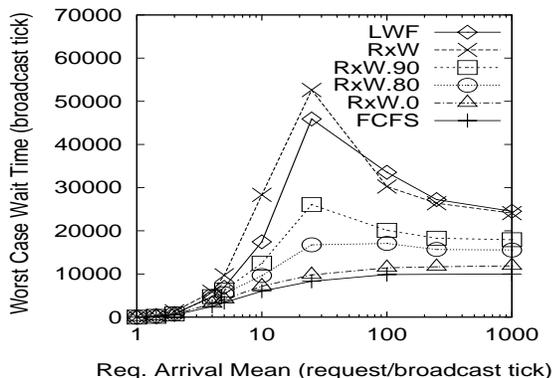


Figure 8: Worst Case Waiting Time

In Figure 8 we see that the highest worst case waiting times are recorded for LWF and *RxW*. As  $\alpha$  is decreased, the *W-list* begins to play a more important role in the scheduling process, and thus, the worst case waiting time decreases. For  $\alpha = 0$ , the worst-case waiting time is within 15% of that of FCFS. Comparing Figures 7 and 8, it is apparent that the  $\alpha$  parameter provides a flexible mechanism for trading-off worst case and average case waiting times for a particular application environment and that it can also be set to balance both concerns reasonably well (e.g.,  $\alpha = 0.80$  in this case). In the next section we show that  $\alpha$  can also be used to adjust the scheduling decision overhead in order to improve the scalability of a system.

<sup>5</sup>In this experiment the simulation was run for two million broadcast ticks, ensuring that each page was broadcast at least 24 times.

An interesting feature of the worst case result is an unexpected peak in worst case wait which is especially visible for *LWF* and maximal *RxW*. This peak occurs because the dominant factor in the scheduling decision changes from popularity to waiting time as the request arrival rate is increased. For example, the *RxW* algorithm initially favors hot pages when the request arrival rate is low, since there is little difference between waiting times at such light loads (everybody waits for a short time) and *R* is the dominant factor. This behavior results in cold pages accumulating higher waiting time than hot pages. As the system becomes more loaded, the server needs to broadcast more pages so the worst case waiting time of cold pages increases further. Since for both *RxW* and *LWF*, the priority of a page is based on the waiting time of its requests, heavier request rates can lead to higher priorities for cold pages. To explain this phenomenon, we examine the behavior of *RxW* for two pages, a hot (popular) page “*h*” and a cold (less popular) page “*c*”. The priority of the service queue entry for the cold page is:

$$Q_c = R_c * W_c$$

where  $R_c$  and  $W_c$  are the total request count and the waiting time for page *c*, respectively. Incrementing the request count by 1 increases this priority by a factor of the waiting time accumulated so far:

$$Q'_c = (R_c + 1) * W_c$$

or

$$Q'_c = Q_c + W_c$$

Therefore for heavy loads where  $W_c > W_h$ , an additional request has a greater impact on cold pages than on hot pages. As the load is increased beyond 25 requests/sec, this relative priority increase results in a decrease in the *worst case* waiting time of cold pages. As the waiting time for cold pages decreases, so does its impact on priorities. As a result, the worst case waiting time begins to stabilize, eventually becoming independent of the load. Using Equation 12 with  $\rho_i$  equal to  $\frac{1}{W_i}$ , it can be seen that for *RxW*, the waiting time of the coldest page (page  $N-1$ ) will eventually converge to:

$$W_{N-1} = \frac{W_o + N - 1}{1 - \sum_{i=0}^{N-2} \frac{1}{W_i} [1 - \sqrt{\frac{p_i}{p_{N-1}}}]}$$
 (13)

### 6.2.3 Scheduling Decision Overhead

As described previously, a critical aspect of scheduling algorithms for large-scale data broadcasting is scalability. The previous section focused on the performance of the algorithms in an ideal setting where there was no overhead for making scheduling decisions or processing requests. In practice, however, overhead can limit the ability for on-demand systems to support large-scale applications. All of the algorithms are fairly efficient in terms of request processing. They differ significantly, however, in terms of scheduling overhead. In this section we examine the issue of scheduling overhead in more detail.

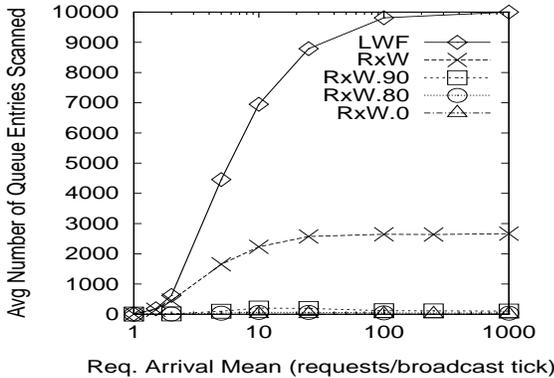


Figure 9: Varying Request Rate (dbsize=10,000 pages)

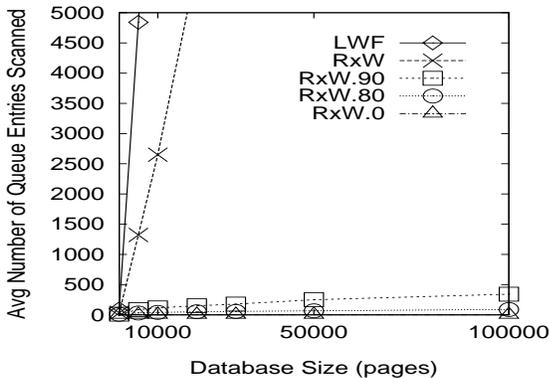


Figure 10: Varying Database Size (100 requests/tick)

Figure 9 shows the scheduling overhead (i.e., the number of request queue entries searched per broadcast decision) for the experiment in Section 6.2.1. As can be seen in the figure, LWF has the highest overhead, followed by *RxW*. LWF is an exhaustive algorithm, and thus the scheduling cost is proportional to the number of distinct pages requested by clients. Under heavy request loads, there is at least one pending request for each page. In terms of scalability in broadcast bandwidth and/or large database size, the scheduling overhead of LWF could easily become a bottleneck. The maximal *RxW* algorithm, on the other hand, examines significantly fewer queue entries at each scheduling decision. It provides 68% savings at a load of 10 requests/tick when compared to LWF. The savings increase

for higher request rates; *RxW* examines 2729 entries on average at a load of 1000 requests/tick; a savings of 73%. It is important to note that as was shown in Figures 7 and 8, these savings in search complexity come at no cost in responsiveness.

The savings provided by *RxW*'s pruning algorithm, however, are dwarfed by the tremendous savings provided by the approximate versions of the algorithm.<sup>6</sup> At a load of 1000 requests/tick *RxW*.90 and *RxW*.80 examine only 116 and 39 entries, respectively. With  $\alpha$  set to 0, the approximate algorithm becomes constant time (two entries are searched on each tick), thereby providing maximum scalability in terms of search overhead.

In the second experiment, shown in Figure 10, the request rate is fixed at 100 requests/tick and the database is scaled from 1 to 100,000 pages. As expected, the overhead of LWF grows at the highest rate while the overhead of maximal *RxW* grows at a somewhat slower rate. The approximation algorithms, however, have orders of magnitude lower overhead across the range of database sizes, with *RxW*.0 remaining constant. The practical impact of these results is that the approximate *RxW* algorithms provide tremendous scalability in terms of request arrival rate, database size, and even bandwidth, as higher broadcast bandwidth results in shorter broadcast ticks, and therefore less time for making scheduling decisions.

### 6.3 Robustness

The previous sections demonstrated the flexibility and scalability of the approximate *RxW* approach. Although approximate *RxW* does not depend on any access probability estimations, it is based on a heuristic that uses a self-adapting threshold for determining what item to broadcast. In this section we examine the sensitivity of the various scheduling algorithms to changes in the workload. The results indicate that the heuristics used do not make the approximate *RxW* algorithm overly fragile. We present results for varying the skewness in the access pattern, changes in access probability "hot spots", and changes in workload intensity.

#### 6.3.1 Changes in Access Pattern Skew

The previous experiments have used the default value for the Zipf parameter  $\theta$ , which creates a highly skewed access pattern. To study the effect of the workload skew, we ran the algorithms with a workload generated by using the default settings of Table 1, but with  $\theta$  varied from 1 to 0.

The average waiting time results are shown in Figure 11. At  $\theta = 0$  the access probabilities are uniform, and all algorithms converge to similar performance. As the skew is increased (moving left along the x-axis) all of the algorithms except FCFS provide improved responsiveness. Since the request rate is kept constant at 100 requests/tick in this experiment, the increased overlap in client interests

<sup>6</sup>FCFS is not shown on these graphs because it is a  $O(1)$  scheduling algorithm and so is insensitive to the parameters varied here.

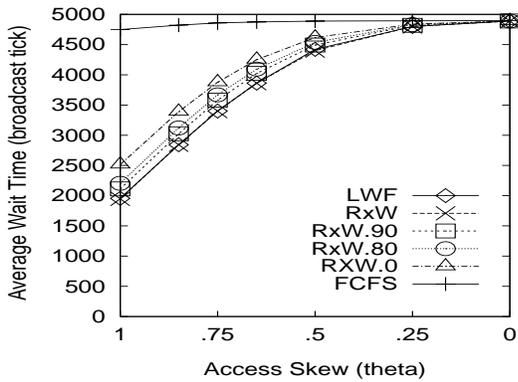


Figure 11: Varying Request Rate (dbsize=10,000 pages)

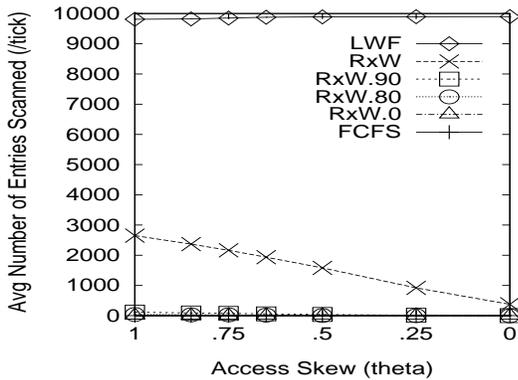


Figure 12: Varying Request Rate (dbsize=10,000 pages)

allows more efficient use of the broadcast bandwidth for the algorithms that take popularity into account. FCFS remains relatively stable because, under heavy loads the system approaches a state in which there is an entry in the queue for every page of the database. At this point FCFS simply cycles through the pages regardless of the overlap of requests. The arrival rate of 100 requests/tick used here is slightly below that level so there is a small increase in the number of service queue entries as the skew is decreased. This accounts for the slight changes seen in FCFS performance here. For *RxW*, the higher approximation settings improve more with increased skew, as they place additional weight on the overlap of requests.

In terms of worst case wait time (not shown here), the algorithms also converge as skew is decreased. For Uniform distribution, all of the algorithms' worst case wait time converges around 9800 ticks, which is the total number of entries in the service queue.

Figure 12 shows the number of queue items scanned for each algorithm in the same experiment as Figure 11. The number of entries in the service queue slightly increases as the skew is decreased, causing *LWF* to examine slightly more entries per decision. In contrast, all of the *RxW* variants (except for the constant time *RxW.0*) scan fewer entries as the skew is reduced. With lower skew, the values of the queue entries are much closer to each other, and this helps the pruning algorithm and the approximations

to stop the search much earlier.

### 6.3.2 Interest Changes

A key benefit of the approximate *RxW* algorithm is its ability to make scheduling decisions based on the current state of the request queue. We therefore expect the algorithm to be robust to changes in client interests. In this experiment we examine the impact of interest shifts of varying magnitude and frequency. To model the shift in interest, we use the *offset* parameter to shift the "hot spot" of the Zipf distribution within the database. Given a database size of  $N$ , the most popular (or hottest) page for a Zipf distribution is page 0 and the coldest is page  $N-1$ . With an offset of  $K$ , the probability distribution is shifted (with wraparound) by  $K$  pages; That is, page  $K$  becomes the hottest page and page  $K-1$  becomes the coldest. The simulation parameter *interval* determines how frequently (in broadcast ticks) the distribution is shifted.

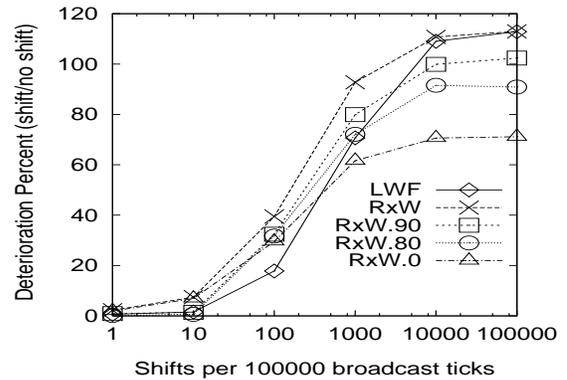


Figure 13: Offset 100: Gradual Shift

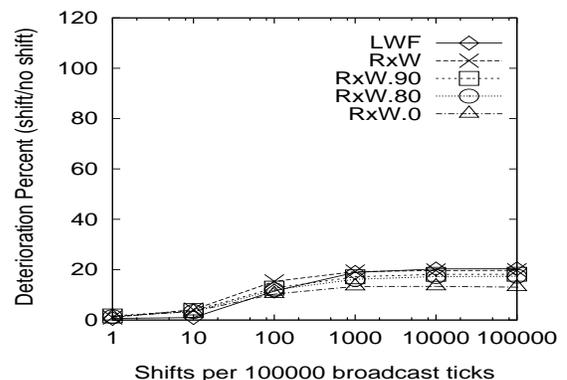


Figure 14: Offset 5000: Change and Restore

Figures 13 and 14 show the *deterioration* in average wait time, which is the percentage increase in average wait time compared to the waiting time experienced for the static case (i.e., no interest shifts). The interval is varied on the x-axis (shown in log scale) from a low value of once per 100,000 ticks to an extreme value of once for each tick. These experiments were run for 1.5 million ticks.

Figure 13 shows a fairly gradual shift of interest using an *offset* of 100 pages (recall that there are 10,000 pages in the default database). As would be expected, all of the algorithms deteriorate as the distribution changes more frequently. As the change rate increases the overall access pattern approaches a uniform distribution where, as shown in the previous experiment, all of the algorithms have similar absolute performance. The key point here, is that despite the adaptive heuristic used by the approximate *RxW* algorithm, it is not overly sensitive to shifts in client interest. Figure 14 shows that this result also holds for larger shifts of interest (i.e., 5000 pages, or half the database). Thus, the heuristic used by the approximate *RxW* algorithm allows it to adapt well even to abrupt changes in client interests.

### 6.3.3 Changes in Request Arrival Rates

Finally, we examine the sensitivity of the algorithms to changes in the request arrival rate. These changes are intended to model situations where there is a sharp increase or decrease in the number of clients using the system. The heuristic used by *RxW* stops searching when it encounters an entry that is within the acceptable range of the self-adjusting threshold. If no such page is found, then the algorithm searches (using pruning) until it finds the maximal  $R \times W$ -valued entry. We ran several experiments to see if the scheduling decision overhead of this heuristic was impacted by workload intensity changes.

Figures 15 and 16 show the search overhead during a time slice of an execution when the workload intensity is abruptly increased (from 5 to 100 req/tick) or decreased (from 100 to 5 req/tick) respectively at time  $t$  (as indicated on the x-axis). The average number of items searched per broadcast decision is shown for LWF, *RxW*, and *RxW.90*. As can be seen in the figures, even with the abrupt changes in intensity, the overhead of the approximate *RxW* algorithm remains far below that of the maximal algorithm (the results for *RxW.80* are not shown; it is even less sensitive than *RxW.90*). The self-adapting mechanisms used by approximate *RxW* are able to quickly detect the change in workload intensity and adjust accordingly.

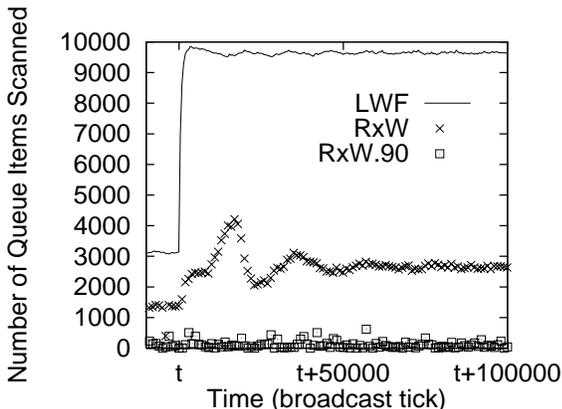


Figure 15: 20-fold Increase in Request Rate

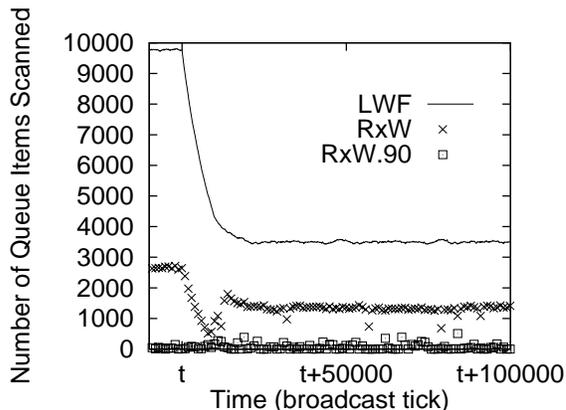


Figure 16: 20-fold Decrease in Request Rate

## 7 Related Work

In this paper we have presented a new algorithm for on-demand scheduling for large scale data broadcast. The directly relevant previous work on scheduling algorithms [13, 26, 25, 24, 14] has been addressed in detail in Section 2. In addition to this work, there have been a number of related efforts on other forms of data broadcast.

Researchers in the multimedia community have studied scheduling algorithms for Video-On-Demand (VOD) systems. As stated in Section 2 these systems involve broadcast of large, continuous objects (e.g., movies). The large granularity of these systems along with timing requirements that arise due to viewing quality concerns makes the problem different in many ways from the data broadcasting systems we have addressed in this paper. Still, there have been several efforts on scheduling for VOD systems that are closely related to our work. Dan et al. [11] have suggested an algorithm that tries to bound the maximum waiting time for popular videos. The algorithm reserves pre-allocated slots for the broadcast of popular videos while servicing other videos using FCFS. This technique requires determining the popularity of a video, deciding on how many popular videos will be serviced with pre-allocated slots, and the frequency of broadcast for the popular movies. In comparison, *RxW* is completely driven by the current state of the request queue, allowing it to quickly adapt to changes in the workload and avoiding problems that arise when estimating access patterns in a dynamic environment.

The VOD work most closely related to our work is that of Aggarwal et al. [4]. They proposed a scheduling algorithm that broadcasts the video with the highest  $R \times T$  value, where  $R$  is the number of outstanding requests, and  $T$  is the time since the last broadcast of that video. This metric is similar to *RxW* except that it requires maintaining a history of the last broadcast for all videos and requires special initialization for videos that have not yet been broadcast. Because of the large granularity of broadcast ticks for VOD systems Aggarwal et al. did not investigate the efficiency or scalability of their scheduling approach. Thus, they did not develop any techniques analogous to the *RxW* pruning and approximation approaches. Furthermore, they did not investigate the robustness of their approaches to changes

in the client workload.

There has also been much recent interest in other areas of data broadcasting for a range of system scenarios and applications. A taxonomy of data delivery mechanisms (including various forms of broadcast) along with a framework for describing dissemination-based systems is provided in [15]. Some recent applications of dissemination-based systems include information dissemination on the Internet [27, 9], Advanced Traveler Information Systems [22] and dissemination using satellite networks [12].

Much work has been done on “push-based” systems in which data is broadcast to clients without specific client requests. A number of systems have been proposed for broadcasting data using a *periodic* push. The Datacycle Project [10, 17] at Bellcore investigated the notion of using a repetitive broadcast medium for database storage and query processing. An early effort in information broadcasting, the Boston Community Information System (BCIS) is described in [16]. BCIS broadcast news articles and information over an FM channel to clients with personal computers specially equipped with radio receivers. The Broadcast Disks project [2, 1] has investigated a number of aspects of data broadcast using periodic push including scheduling and client behavior. Scheduling techniques from the real-time community have been applied to data broadcast by Baruah and Bestavros [8]. The issue of combining broadcast push and unicast pull is addressed in [3, 23]. The mobility group at Rutgers [18] has done significant work on data broadcasting in mobile environments. A main focus of their work has been to investigate novel ways of indexing in order to reduce power consumption at the mobile clients.

## 8 Conclusions

This paper has focused on the challenges of large-scale on-demand data broadcast introduced by high bandwidth broadcasting media such as satellite or cable networks. We began by providing a comprehensive set of performance criteria for scheduling algorithms. These criteria include average and worst case response time, three dimensions of scalability, and robustness to changes in the nature and or intensity of the workload. We then described how previous algorithms fail in one or more of these criteria.

We proposed a scheduling algorithm called *RxW*, that aims to provide balanced treatment of both hot and cold pages resulting in a good overall performance. The algorithm combines two previous approaches (called MRF and FCFS), and uses a novel pruning technique to reduce the search space for making broadcast decisions. While such pruning was shown to be effective, it was observed that such an algorithm could still eventually become a bottleneck when supporting very large applications.

Building on specific behaviors observed for *RxW*, we developed an *approximate*, parameterized variant of *RxW* that allows the search space to be reduced further, at the expense of making somewhat less efficient use of broadcast

resources. By varying a single parameter, the algorithm can be tuned from the regular *RxW* algorithm, to a *constant time* approach that provides maximal scalability.

We demonstrated the performance, scalability, and robustness of the *RxW* variants through an extensive set of performance experiments and sensitivity analyses. We also provided an analytical treatment of the maximal *RxW* algorithm that provided insight into its efficient use of broadcast resources. This analysis demonstrated a key advantage of broadcast data delivery compared to unicast delivery. Namely, that for a user population with overlapping requests the expected average waiting time is bounded even if the request rate increases infinitely. This inherent scalability of broadcast data delivery makes it an ideal technology for large-scale data dissemination applications.

Our ongoing work is focused on the *Data Staging* problem for on-demand broadcast systems. The work reported in this paper, along with the previous work in the area has made the implicit assumption that all of the items to be disseminated are readily available at the server when they are scheduled to be broadcast. In many situations, however, the items to be broadcast may reside in secondary, tertiary, or even remote storage. The need to fetch data from such locations produces large variance in service times, which can destroy the performance of the broadcast scheduling heuristics. We have developed several promising approaches for coordinating broadcast scheduling. We have implemented these techniques in a data broadcasting testbed and are performing experiments on them. We also plan to integrate the on-demand scheduling techniques described here, with push-based and other forms of data delivery as part of a larger Dissemination-Based Information Systems (DBIS) framework [6].

## References

- [1] S. Acharya, R. Alonso, and M. Franklin. Broadcast disks: Data management for asymmetric communication environments. In *Proc. of ACM SIGMOD*, Santa Cruz, CA, May 1995.
- [2] S. Acharya, M. Franklin, and S. Zdonik. Dissemination-based data delivery using broadcast disks. *IEEE Personal Communications*, 2(6), December 1995.
- [3] S. Acharya, M. Franklin, and Stan Zdonik. Balancing push and pull for data broadcast. In *Proc. of ACM SIGMOD*, Tucson, AZ, May 1997.
- [4] 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.
- [5] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. In *Proc. of IEEE INFOCOM*, San Francisco, CA, March 1998.
- [6] M. Altinel, D. Aksoy, T. Baby, M. Franklin, W. Shapiro, and S. Zdonik. Dbis-toolkit: Adaptable

middleware for large scale data delivery. In *Demo description for the ACM SIGMOD Conference*, Philadelphia, PA, June 1999.

- [7] M. H. Ammar and J. W. Wong. On the optimality of cyclic transmissions in teletext systems. *IEEE Transactions on Communications*, 35(1):68–73, December 1987.
- [8] S. Baruah and A. Bestavros. Pinwheel scheduling for fault-tolerant broadcast disks in real-time database systems. In *13th International Conference on Data Engineering*, pages 543–551, April 1997.
- [9] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3), September 1996.
- [10] T. Bowen, G. Gopal, K. Lee, and A. Weinrib. The datacycle architecture. *Communications of ACM*, 32(12), December 1992.
- [11] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proc. of ACM Multimedia 94*, San Francisco, CA, October 1994.
- [12] S. Dao and B. Perry. Information dissemination in hybrid satellite/terrestrial networks. *Data Engineering Bulletin*, 19(3), September 1996.
- [13] H.D. Dykeman, M. Ammar, and J.W. Wong. Scheduling algorithms for videotex systems under broadcast delivery. In *IEEE International Conference on Communications*, pages 1847–1851, Toronto, Canada, 1986.
- [14] H.D. Dykeman, J.W. Wong, and M.H. Ammar. Scheduling algorithms for broadcast delivery information systems. Technical Report GIT-ICS-88/35, Georgia Institute of Technology, 1988.
- [15] M. Franklin and S. Zdonik. A framework for scalable dissemination-based systems. In *Proc. of ACM OOP-SLA Conference*, October 1997.
- [16] D. Gifford. Polychannel systems for mass digital communication. *Communications of ACM*, 37(10), October 1994.
- [17] G. Herman, G. Gopal, K. Lee, and A. Weinrib. The Datacycle architecture for very high throughput database systems. In *Proc. of ACM SIGMOD*, San Francisco, CA, May 1987.
- [18] T. Imielenski and B.R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10), October 1994.
- [19] L. Kleinrock. *Queueing Systems - Volume II*. John Wiley and Sons, 1976.
- [20] D. Knuth. *The art of Computer Programming - Volume III*. Addison-Wesley, 1981.
- [21] H.D. Schwetman. CSIM: A C-based process oriented simulation language. In *Proc. of the Winter Simulation Conf.*, pages 387–396, 1986.
- [22] S. Shekhar, A. Fetterer, and D.R. Liu. Genesis: An approach to data dissemination in advanced traveler information systems. *Data Engineering*, 19(13), September 1996.
- [23] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adap-

tive data broadcast in hybrid networks. *the 23rd International Conference on Very Large Data Bases*, 30(2), September 1997.

- [24] C.J. Su and L. Tassioulas. Broadcast scheduling for information distribution. In *Proc. IEEE INFOCOM*, 1997.
- [25] N.H. Vaidya and S. Hameed. Data broadcast in asymmetric wireless environments. In *Proc. of Workshop on Satellite-based Information Services (WOSBIS)*, New York, November 1996.
- [26] J.W. Wong. Broadcast delivery. *Proc. of IEEE*, 76(12):1566–1577, December 1988.
- [27] T. Yan and H. Garcia-Molina. Efficient dissemination of information on the internet. *Data Engineering*, 19(13), September 1996.

## A Derivation of $L_b$

In this Appendix, we present the derivation of Equation 6. In Figure 5, we have seen that an entry for page  $c$  (a less popular page) can compete for service if it is created before  $t_c$ . We try to find the value of  $t_c$  using Equation 4 which can be rewritten as,

$$(\sqrt{\lambda p_c}(W_b + t_c))^2 = (\sqrt{\lambda p_b}(W_b))^2$$

taking square root of each side and factoring out  $\sqrt{\lambda}$ ,

$$W_b = \frac{\sqrt{p_c}}{\sqrt{p_b} - \sqrt{p_c}} t_c$$

or

$$W_b + t_c = \left[ \frac{\sqrt{p_b}}{\sqrt{p_b} - \sqrt{p_c}} \right] t_c \quad (14)$$

The chances that a *page-c* entry has been waiting in the queue and that will accumulate enough priority in the interval  $W_b + t_c$  can be expressed as

$$L_{cb} = \int_0^\infty \Lambda_c P\{t < w_c \leq \frac{\sqrt{p_b}}{\sqrt{p_b} - \sqrt{p_c}} t\} dt \quad (15)$$

where  $\Lambda_c dt$  is the expected number of *page-c* entries created in the interval  $(t, t+dt)$  and  $P\{t < w_c \leq \frac{\lambda p_b}{\lambda p_b - \lambda p_c} t\}$  is the probability that a request arrived in that interval spends at least  $t$  and at most  $\frac{\sqrt{p_b}}{\sqrt{p_b} - \sqrt{p_c}} t$  time units in the service queue. Therefore,

$$L_{cb} = \Lambda_c \int_0^\infty [1 - P(w_c \leq t)] dt \quad (16)$$

$$- \Lambda_c (1 - \sqrt{\frac{p_c}{p_b}}) \int_0^\infty [1 - P(w_c \leq y)] dy$$

with the change of variable  $y = \frac{\sqrt{p_b}}{\sqrt{p_b} - \sqrt{p_c}} t$ . Since the waiting time of *page-c* requests by is

$$W_c = \int_0^\infty [1 - P(w_c \leq x)] dx \quad (17)$$

thus, applying Equation 17 on Equation 16:

$$L_{cb} = \Lambda_c W_c \sqrt{\frac{p_c}{p_b}} \quad \text{if } c \text{ is less popular than } b \quad (18)$$

This completes the case for lower priority entries. Any entry with higher priority will be serviced before *page-b* and therefore

$$L_{cb} = \Lambda_c W_c \quad \text{if } c \text{ is not less popular than } b \quad (19)$$

## B MRF: Expected Waiting Time

We use a similar time-based priority queuing model for MRF scheduling. However, the priority function in this case is simply defined as

$$Q_i(t)[MRF] = \lambda p_i(t - t_i)$$

where  $\lambda p_i$  is the request rate for page  $i$ , and  $t_i$  is the arrival time of oldest request represented in the queue entry. The expected waiting time of a *page-b* entry, is defined by the pages that are going to be broadcast before page  $b$ , as stated in Equation 2. When we calculate the expected number of entries that were already queued and will be serviced before *type-b* entry, the components to be summed up for  $L_b$  and  $M_b$  is:

$$L_{ib} = \begin{cases} \Lambda_i W_i & \text{if } i \text{ is more popular} \\ \Lambda_i W_i \frac{p_i}{p_b} & \text{otherwise} \end{cases} \quad (20)$$

$$M_{ib} = \begin{cases} \Lambda_i W_b (1 - \frac{p_b}{p_i}) & \text{if } i \text{ is more popular} \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

This leads to the following expected waiting time formula, which is structurally similar to Equation 12:

$$W_b[MRF] = \frac{W_o + \sum_{i=b+1}^N \rho_i \frac{p_i}{p_b} W_i + \sum_{i=0}^{b-1} \rho_i W_i}{1 - \sum_{i=0}^{b-1} \rho_i [1 - \frac{p_b}{p_i}]} \quad (22)$$

which also explains the scalability property in terms of request arrival rates, with a similar discussion as described in Section 5. Note the appearance of the access probabilities in straight ratio.