Caching and Scheduling for Broadcast Disk Systems

Vincenzo Liberatore
Case Western Reserve University

Unicast connections lead to performance and scalability problems when a large client population attempts to access the same data. Broadcast push and *broadcast disk* technology address the problem by broadcasting data items from a server to a large number of clients. Broadcast disk performance depends mainly on caching strategies at the client site and on how the broadcast is scheduled at the server site. An *on-line* broadcast disk paging strategy makes caching decisions without knowing future page requests or access probabilities. This paper gives new implementations of existing on-line algorithms and reports on extensive empirical investigations. The gray algorithm [Khanna and Liberatore 2000] always outperformed other on-line strategies on both synthetic and Web traces. Moreover, caching limited the skewness of broadcast schedules, and led to favor efficient caching algorithms over refined scheduling strategies when the cache was large.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*algorithm engineering*

Additional Key Words and Phrases: Broadcast Disk, Caching, Scheduling

1. INTRODUCTION

The demand of network data services has been growing exponentially during the past few years. More and more often, increased workloads cannot be satisfied by current technology. In particular, when clients requested several million connections to a hot Web site during peak periods (e.g. Deep Blue chess match, Olympic games), servers were overmatched by the heavy workload [Franklin and Zdonik 1997]. The problem was that point-to-point (unicast) connections satisfied each request individually, and server performance did not scale with the number of requests. In general, the point-to-point paradigm poses a scalability problem that is exacerbated by the exponential and sustained growth of data service demand. In this scenario, *broadcast push* [Stathatos et al. 1997] promises to address the issue. Broadcast push has servers broadcast the same data items to a large number of clients, and thus it overcomes the bottleneck that unicast creates at the server

Address: Department of Electrical Engineering and Computer Science, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, Ohio 44106-7071. Ph.: 216-368-2802. E-mail: vliberatore@acm.org.
Work done while at UMIACS, University of Maryland, College Park. The author was partly supported by grant CCR-9501355.

site. Broadcast push is being incorporated in several commercial systems. For example, Hughes Network System delivers Web pages via satellite links, and Hybrid Networks Inc.[1] will broadcast data via cable lines.

Broadcast Disks [Franklin and Zdonik 1997] attempt to improve broadcast push performance by the combination of two methods: they exploit client caching and fix a cyclical broadcast schedule over long periods of time. Clients are helped by local caches because they avoid waiting on the network if they can find data items in their own cache. Cyclical schedules help caching strategies to weigh different eviction choices [Acharya et al. 1995; Khanna and Liberatore 2000]. Moreover, cyclical schedules lead to scalable and widely supported multicast techniques over the Internet [Almeroth et al. 1998] and are necessary in a mobile environment where clients need to know when to tune in to receive data [Khanna and Zhou 1998]. Our main contributions are the first empirical study of on-line algorithms for broadcast disk caching and the analysis of the interaction between client caching and broadcast scheduling.

1.1 Caching

Cache management in a broadcast disk environment differs from other caching problems because caching aims at reducing the time spent waiting during a fixed broadcast schedule. By contrast, minimizing (say) the number of page faults in isolation could not bring in any performance improvement. Moreover, *prefetching* can sometimes be executed at no cost for servers and clients [Acharya et al. 1996]. Several broadcast disk paging algorithms assume that clients requested data items with given probabilities and that those probabilities are known to paging strategies [Acharya et al. 1995; Acharya et al. 1996; Tassiulas and Su 1997]. In practice, probabilistic assumptions could be difficult to find and to validate [Liberatore 1999], and so it is critical to have efficient paging strategies that operate without probabilistic parameters. The *gray algorithm* is on-line, works with no probabilistic assumption, and is provably optimal in terms of competitive ratios [Khanna and Liberatore 2000]. However, no efficient implementations were known thus far and no empirical analysis was performed on on-line algorithms. We report simple, but efficient implementations of existing algorithms, and subject paging algorithms to extensive empirical investigation on both synthetic and Web traces. The gray algorithm always outperformed previous on-line strategies.

1.2 Scheduling

Scheduling is the problem of establishing a broadcast schedule. Ideally, scheduling should depend on client access patters. For example, typical schedules broadcast hot pages more often [Bar-Noy et al. 1998; Su and Tassiulas 1997]. Scheduling with non-uniform transmission times [Kenyon and Schabanel 1999; Vaidya and Hameed 1997], multiple broadcast channels [Kenyon et al. 2000], and non-linear cost functions [Bar-Noy et al. 2000] has been investigated as well. A scheduling principle is expressed by the *square-root law*, which has been proposed independently by several authors [Ammar 1987; Ammar and Wong 1985; Gecsei 1983; Su and Tassiulas

[1]The company profiles can be found at `http://www.direcpc.com/` and `http://www.hybrid.com/`, respectively.

1997]: broadcast pages with frequency proportional to the square root of the probability those pages will be requested. The square-root law results in transmission frequencies that are in general non integral, and thus the law can be interpreted as a linear relaxation of an integer programming problem [Bar-Noy et al. 1998]. The problem of finding an optimal cyclic schedule is NP-hard [Bar-Noy et al. 1998], but not MAX-SNP-hard [Kenyon et al. 2000], and it can be solved in polynomial time if the server broadcasts only two pages [Bar-Noy and Shilo 2000]. A simple 2-approximation algorithm is expressed by the MAD rule [Su and Tassiulas 1997; Bar-Noy et al. 1998]. A broadcast disk will typically employ both scheduling and client caching. However, previous work has mostly examined scheduling and caching in isolation. This paper investigates the relationships between scheduling and caching. MAD scheduling provided larger performance improvements when caches were small. However, schedules need not be very skewed when larger caches are used, and, in this case, speed-up stemmed mostly from the choice of a page replacement strategy.

1.3 Organization

Section 2 describes the broadcasting environment. Section 3 overviews algorithms for broadcast disk paging and gives simple and efficient implementations. Section 4 describes our experimental set-up. Section 5 reports experimental results for the case of a flat broadcast. Finally, section 6 examines the interaction between caching and scheduling.

2. THE BROADCASTING ENVIRONMENT

The *broadcast disk* environment is well-known in the literature [Acharya et al. 1995; Acharya et al. 1996; Franklin and Zdonik 1997; Khanna and Liberatore 2000] and we will only outline it here. A database of n pages is cyclically broadcast by a server. Pages have all the same size and it takes the same amount of time to broadcast any page. At first, we assume that pages are broadcast with the same frequency (*flat broadcast*), and we will remove such assumption later on (*skewed broadcast*). A *broadcast tick* is defined as the time needed to transmit a page. In practice, a broadcast tick can take between 10ms to 30ms in Web broadcast push [Almeroth et al. 1998]. Although precise quantifications depend on actual platforms, such broadcast tick durations should be long enough to allow servers and clients to execute computation and some I/O during one tick. We define a *rotation* as the time needed to transmit the whole server database. Therefore, a rotation is n broadcast ticks. An example of a broadcast program is illustrated in figure 1. Pages are received by the clients in the same order as they are broadcast. Each client can cache a subset of $k < n$ pages. As in previous papers [Acharya et al. 1995; Acharya et al. 1996; Khanna and Liberatore 2000], the broadcast disk environment is restricted as follows:

—The broadcast schedule is fixed by the server, and is known by clients.

—Pages are read-only, and cannot be updated by either the server or the clients.

—Clients receive pages only from the server broadcast.

Since clients are isolated from each other and the broadcast schedule is fixed, the performance of each client is independent of the behavior of any other client. In

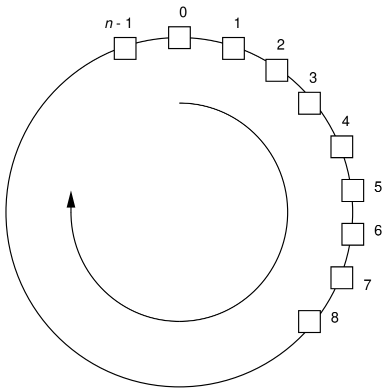


Fig. 1. Example of a flat broadcast program. Pages are numbered from 0 to $n-1$, and are cyclically transmitted by the server in that order.

the broadcast disk environment, each client requests a sequence of pages. At each step, the client finds the requested page either in its local cache or in the broadcast disk. If the client has cached the page, it can access it immediately. Otherwise, the client waits for the server to broadcast the desired page again. In broadcast disks, client computation is blocking, that is, no page request is issued while waiting for a faulting page. The main objective of broadcast disk paging algorithms is to reduce the total waiting time incurred by a client. An important characteristic of broadcast disk paging is the role of prefetching: a client will be said to *prefetch* a page p if p is loaded in the cache even though p is not requested by the client computation. In broadcast disks, some prefetching can be executed without waiting on the broadcast. Specifically, suppose that a client is waiting for a faulting page q to be retransmitted by the server. While the client is waiting, other pages are transmitted by the server and can be loaded on the fly. Those pages are not requested, and so, according to the definition above, they would be prefetched. However, no time is wasted to retrieve those pages as they are loaded while waiting for another page. A broadcast disk paging strategy is said to be *lazy* [Khanna and Liberatore 2000] if it prefetches pages only when they can be loaded on the fly, that is, without waiting on the broadcast.

3. PAGE REPLACEMENT STRATEGIES

We consider three different on-line page replacement strategies. The three on-line strategies are also compared with PT [Acharya et al. 1996], which is not on-line because it uses access probabilities. PT is included as a point of comparison for the other algorithms. Algorithm performance can be expressed in two metrics:

(1) Client *waiting time* on the broadcast, and
(2) Time needed to run the algorithm itself on the client (*running time*).

Previous work focused on waiting time, and so-called "implementations" of previous algorithms reduce running times, but are only faster heuristic approximation of the original procedures that targeted waiting times. We introduce new implementations for several existing algorithms. Our implementations are simple, efficient in terms of running times, and, unlike previous heuristics, do not affect waiting times. It is assumed that a client can perform a constant number of operations during each broadcast tick to observe the transmitted page and possibly to load it into its cache. We will also assume that a page can be located in the cache in constant time, which can be achieved, for example, by using tables. The main objectives for our implementations are to minimize the running time per fault and the running time per request without affecting waiting times.

3.1 LRU and CF

The *LRU* algorithm responds to a miss by evicting the page that has been used least recently. LRU is $O(nk)$-competitive for broadcast disk paging [Khanna and Liberatore 2000]. The *Closest-First* (CF) strategy works as follows. When a page request causes a page fault, CF waits for the requested page to be broadcast, loads it into the cache, and evicts the page that is currently in the cache and that will be retransmitted first. The evicted page is the one that can be reloaded with the least waiting time. No competitiveness result is known for CF. LRU and CF are antithetic in the following sense. LRU evicts pages independently of the waiting time needed to reload them. The gist of LRU is that past accesses should predict future accesses [Tanenbaum 1992], and so LRU should incur few page faults. To the contrary, CF does not base evictions on previous history, but only on waiting times. However, if CF makes an eviction mistake and if CF immediately detects it, then it can recover from it at little cost. In conclusion, LRU and CF are antithetic because LRU uses past history independently of waiting times, while CF does not use any history, but only waiting times. LRU and CF are similar in the following respect: neither algorithm executes prefetching.

LRU can be clearly implemented with a (binary) heap ordered by least recent usage, so that LRU takes $O(\log k)$ time per request. We now turn to our implementation of CF on a flat broadcast. CF maintains the cache as a red-black tree [Cormen et al. 1990] ordered by transmission times. When CF faults on page p, we insert p in the red-black trees. The page q evicted by CF is the page that will be retransmitted the soonest. In terms of page identifiers, q is either

(1) the smallest cached page q greater than p, or,

(2) if, no such q exists, the cached page with the smallest identifier.

We try to find p's successor in the tree, that is, the smallest tree element q that is larger than p. If there is a successor q of p, then q is removed from the tree. If there is no successor, then we locate the minimum element in the tree and remove it. All these operations take $O(\log k)$ time, and thus CF takes $O(\log k)$ time per fault. Observe that, in both implementations above, there is no time associated with a single broadcast tick. This feature is particularly useful in simulations as we do not need to simulate the low-level system operations that take place during each tick.

3.2 The Gray Algorithm

The *gray* algorithm [Khanna and Liberatore 2000] combines LRU's history with CF's waiting times, executes prefetching, is on-line and $O(n \log k)$-competitive, which matches the lower bound except in the constant factors [Khanna and Liberatore 2000]. This paper will show that gray outperforms LRU also in stochastic and trace-based simulations. The gray algorithm is lazy [Khanna and Liberatore 2000], that is, it prefetches pages only when they can be loaded on the fly.

The gray algorithm owes its name to a special marking scheme that assigns a color (black or gray) to each cached page. Page colors guide algorithm evictions and prefetching decisions. The gray algorithm is described in the Appendix, and here we detail our new implementation. It is not hard to see that there is an implementation of the gray algorithm which takes $O(1)$ time per tick and $O(\log k)$ time per request. Although such implementation is satisfactory to run on a client host, for the purpose of simulation, we seek an algorithm whose running time is independent of broadcast ticks and is affected only by the number of page requests. This is achieved with the following scheme that takes $O(\log k)$ time per request. The implementation uses *order-statistic trees* [Cormen et al. 1990]. Order-statistic trees allow gray to obtain the relative position (*rank*) of cached gray pages in logarithmic time. Hence, we establish whether a gray page is cached by comparing page rank with the current broadcast tick and the number of black pages. Specifically, we keep the set of gray pages in an order-statistic tree (the *gray tree*) ordered by page index. Analogously, we keep the set of black pages in an order-statistic tree (the *black tree*). We also maintain the index p of the last broadcast page and the number b of cached black pages. On a page request, we attempt to locate the requested page in the black and in the gray trees in $O(\log k)$ time. If the requested page is black, no further action is required. If the requested page is white (a miss), then it is inserted in the black tree, which takes $O(\log k)$ time. If the requested page is gray, then we access the gray tree to locate the gray page q that immediately follows p (the last broadcast page) and compute its rank in $O(\log k)$ time. We now use the rank of q and the number b of cached black pages to compute the rank of the gray page that is cached and that will be the first to be transmitted after p. We access the gray tree to compute the rank of the requested page, and a comparison establishes if the requested gray page is cached or not. Finally, gray removes the requested gray page from the gray tree and inserts it into the black tree. All these operations take $O(\log k)$ time. At the end of a phase, the gray tree is removed and the black tree becomes gray. The removal of the gray tree takes $O(k)$ time per phase, which becomes $O(1)$ time for each phase request if we stagger tree removal through the next phase, with a technique similar to [Overmars 1983]. On the whole, gray's implementation takes $O(\log k)$ time per request.

3.3 PT

The last replacement strategy considered here is PT [Acharya et al. 1996]. PT maintains two values for each page i in the server database. The first value is p_i, the probability that page i will be requested. The second value is t_i, which is the waiting time needed to load i once the current request has been satisfied. There are some limitations to the algorithm PT: PT assumes knowledge of page accesses probabilities, uses $\Omega(n)$ space to maintain probability values, and finally, its implementation is worse than LRU's and gray's, as will be discussed next.

A previous implementation of PT takes $O(k)$ time on each broadcast tick [Acharya et al. 1996]. Therefore, when a page fault forces PT to wait for w broadcast ticks, PT takes $O(kw)$ time. We will next give a new implementation of PT. Our im-

Parameter	Description	Base Value
n	number of pages in the broadcast	5000
AccessRange	number of pages accessed by a client	1000
RegionSize	number of pages with the same access probability	50
ThinkTime	delay between requests	0
k	client cache size	50,250,500,750,900
Length	trace length	100000
θ	skew of Zipf distribution	.95

Table 1. Parameters used to generate synthetic workloads.

plementation uses the selection algorithm in a trivial way and reduces the running time to $O(k+w)$ per fault. An important observation is that PT's implementation achieves the same time bound also on non-flat schedules as long as the t_i values can be obtained in $O(1)$ time. When PT faults, it moves one broadcast tick at a time, forms a candidate sets, and rejects at most one element in the candidate set. However, PT can be equivalently described in the following, rather different way. When PT faults, it forms a set C that contains all the pages that were in the cache before the fault, plus all the pages that are transmitted while waiting for the faulting page. After the fault, PT caches the faulting page plus the $k-1$ pages in C that have the largest value of $p_i t_i$. PT invokes the selection algorithm to determine the set of cached pages. In conclusion, PT takes $O(|C|) = O(k+w)$ time on each fault, and, as observed above, such bound carries over also to non-flat schedules. Our simulator implements PT with the randomized version of the selection algorithm [Cormen et al. 1990].

The PT algorithm requires that page requests be generated with stationary probabilities and that those probabilities are known to the caching site. In our simulation, we used the original PT algorithm, but we also performed simulations where the caching site has no prior information regarding access probabilities. In those experiment, PT estimates page access probabilities during a training period and uses those estimates during its operational lifetime. As a result, such strategy does not need prior information, but still requires that page requests be generated independently with stationary probabilities.

4. SIMULATION SET-UP

This section describes the simulation set-up. Most of the environment was described in Section 2. The parameters that define the following workloads are stated in table 1. Most of these parameters take the same base values as in previous papers [Acharya et al. 1995; Acharya et al. 1996].

4.1 Basic Workloads

Our first workload is *Random*. In this workload, a sequence of *Length* page requests is extracted uniformly at random from the server database. Our second workload assumes a stationary Zipf distribution and is similar to the one defined in the existing literature on broadcast disks [Acharya et al. 1995; Acharya et al. 1996]. The Zipf distribution is often used to model skewed access patterns because it gives some pages a higher probability of being requested [Knuth 1973]. We generate a synthetic trace as follows. At the very beginning, we extract a set of $AccessRange < n$ pages uniformly at random from the server database. The synthetic trace will contain only the pages in the access range, and will not use any other database page. We will then partition the access range into *NumRegions* regions of equal size *RegionSize* = *AccessRange* / *NumRegions*. At this point, we initiate the generation of a page request sequence. For each request, first we extract a region according to a Zipf distribution with parameter θ. In other words, the probability that region r is extracted is proportional to $1/r^\theta$ ($1 \leq r \leq NumRegions$). Then, we extract a page uniformly at random from the chosen region. We will repeat the process to generate a sequence of *Length* page requests. We will say that such traces have been generated by the *default Zipf* workload. While pages in the same region have the same probability of being requested, pages in different regions have different probabilities. We will say that a page (region) is *hotter* than another if it has (its pages have) a higher probability of being accessed and *colder* if it has (its pages have) lower probability of being accessed.

4.2 Robustness

We measure the robustness of paging algorithms by changing the parameters of the default Zipf workload and measuring sensitivity to those changes.

The first parameter change was region placement. In the default workload, regions were disjoint sets extracted uniformly at random from the server database. We measured algorithm performance when the placement of pages into regions is not random, but follows a regular pattern related to access probabilities. In the *Uniform Region* workload, regions are consecutive intervals of pages and regions are broadcasted in non-decreasing order of access probability. The *Reverse Region* workload is identical to the uniform region workload, except that hot regions precede cold regions.

In the basic workloads, when a page request is satisfied, the client immediately generates the next request. Suppose instead that, when a page request is satisfied, the client executes some local processing before issuing the next request. Such scenario is modeled by the parameter *ThinkTime* [Acharya et al. 1995]. After a page request is satisfied, the broadcast progresses for *ThinkTime* broadcast ticks before the client issues the next request. During the *ThinkTime* interval, the client processes the previous request and has no knowledge of which page it will request next. However, during the *ThinkTime* interval, the client has the ability to prefetch pages on the fly.

We measure robustness to changes in the Zipf parameter θ. When $\theta = 0$, the probability distribution among regions is uniform and no region is hotter or colder than any other region. Such workload is very similar to *Random* except that only *AccessRange* pages are used rather than the whole server database. As θ increases, the distribution increasingly becomes more skewed.

In the Zipf workloads above, access probabilities do not change with time. We measured algorithm performance when client interests shift over time. We modeled changing access patterns with two parameters: the *SwitchTime* and the *Offset*, which will be utilized as follows. The synthetic trace is generated as in the default Zipf workload, except that every *SwitchTime* page requests, the region contents are changed: in every region, *Offset* pages are discarded and replaced with a new set of *Offset* pages. Regions will again be disjoint after the shift, but a discarded page can be extracted for the same or for another region. Experiments were performed with *Offset* = 45, which corresponds to a radical shift of 90% of *RegionSize*, and with *Offset* = 25, which replaces only half a region. *SwitchTime* was set to 1000, which induces 100 shifts per trace, and to a milder *SwitchTime* = 8000, which changes the access pattern twelve times during a trace. When *SwitchTime* = 1000 and *Offset* = 45, the workload tends to be more random than for larger values of *SwitchTime* and smaller values of *Offset*.

4.3 Simulation Details

We generated thirty traces for each combination of parameters in order to establish confidence intervals around average values. In our simulations, when we change parameters that do not affect trace generation (algorithm and cache size), we use the same thirty traces for all algorithms and all cache sizes. We used the Mersenne twister random number generator [Matsumoto and Nishimura 1998] with a sequence of seeds that we entered manually. In preliminary versions of this paper, experiments where executed with the `random(3B)` and inversive congruential [Eichenauer-Herrman 1992; Eichenauer-Herrman 1995; Leeb and Wegenkittl 1997] random number generators, and those results are in agreement with the ones reported here. We generated Zipf random values with the approximate algorithm in [Gray et al. 1994].

We measured algorithm waiting times and number of page faults after the following *warm-up* process [Acharya et al. 1995]. Each algorithm's cache is filled with the first k distinct pages requested in each sequence. In addition, the LRU algorithm starts with those warm-up pages arranged in least-recently used order.

4.4 Web Workload

Experiments were also executed with four actual Web traces. The traces are extracted from the log of the servers for the Soccer World Cup 98. The World Cup trace includes more than one billion requests over a period of one month and a half and is the largest trace analyzed to date [Arlitt and Jin 1999]. The World Cup servers received up to 10 million requests per hour. As a result, the World Cup site is one of the most busy recorded so far, which makes it an ideal testbed for broadcast disk performance. In a preliminary version of this paper we executed experiments for older and less busy Web traces, which we do not report here. We extracted four client traces from the server trace. Specifically, we focused on the fifth subtrace of July 8, 1998, which covers the most busy period for the World Cup server. We collected the four client subtraces that contained the largest number of requests per session, where a session is defined as a sequence of requests issued no more than 5 minutes apart. Some document sizes changed during the course of the trace. Size change is due either to interrupted transfers or to document contents updates. Although updates can be incorporated in broadcast disk environments [Shanmugasundaram et al. 1999], we restrict this study to fixed documents and did not broadcast documents with changing sizes. Actually, there were only 8 documents whose size changed, and those accounted for less than 10% of all requests. Trace characteristics are given in table 2 and will be discussed next. We discarded requests for dynamic documents and those that caused a reply code other than 200(`OK`). We divided the documents into 512KB pages and broadcast them at a

client id	http 200(OK)	n	*Length*
9054	12230	553	40680
732	9477	536	28553
2896	8695	414	25065
647850	8553	659	28678

Table 2. Characteristics of client Web traces collected from the World Cup 98 server trace. The first column reports the client id number, the second column gives the number of http requests that were generated by that client and that resulted in a 200(OK) reply code, the third column gives the number of pages in the broadcast schedule, and the final column is the length of the broadcast disk request sequence.

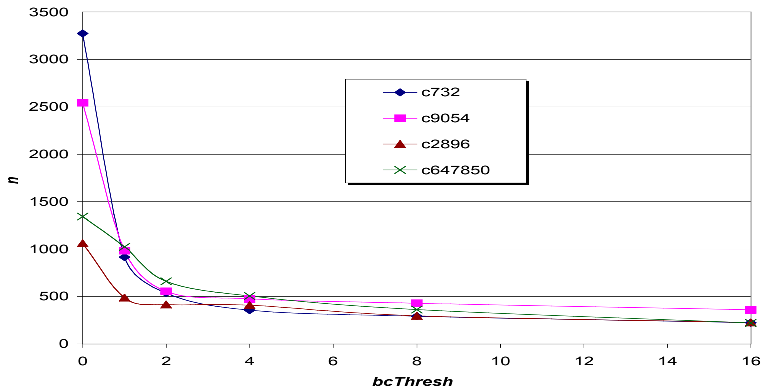
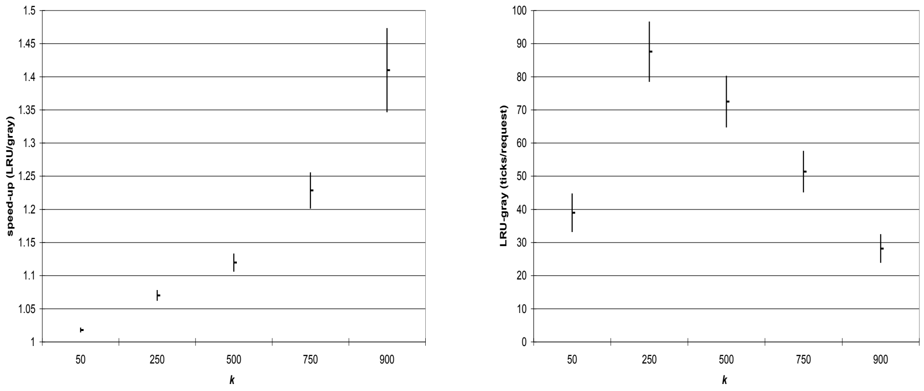


Fig. 2. Number n of unique broadcast disk pages in Web traces as a function of the threshold *bcThresh*.

rate of 256Kbps (these parameters are the same as in [Almeroth et al. 1998]). The broadcast was a random permutation of the set of pages and was fixed before initiating the broadcast, as suggested by [Acharya et al. 1996]. In Internet data delivery, documents that are referenced sporadically are not usually broadcast [Almeroth et al. 1998; Stathatos et al. 1997] (as, for example, in the motivating examples in the introduction). We insert in the broadcast schedule only those documents that are requested more than *bcThresh* = 2 times. The effect of *bcThresh* is shown in figure 2, and supports the observation that few documents are very popular. We report in table 2 the number n of pages in the final broadcast schedule and the total *Length* of the page request sequence. An important observation is that, in the World Cup workload, page access frequencies do not follow a Zipf distribution [Arlitt and Jin 1999].

5. EXPERIMENTAL RESULTS

This section reports on the results of our experimental comparison among LRU, gray, CF, and PT. We report waiting times of heuristics in broadcast ticks. This section gives results for a flat broadcast, and non-flat schedules will be discussed in §6.



(a) Speed-up (b) Difference

Fig. 3. Performance of LRU and gray in the default Zipf workload. Chart (a) reports the speed-up of gray over LRU. The horizontal mark is the average speed-up and the vertical interval is the average plus or minus three times the standard deviation of the average waiting time over thirty experiments. Chart (b) gives analogous quantities for the difference of the average waiting time per request.

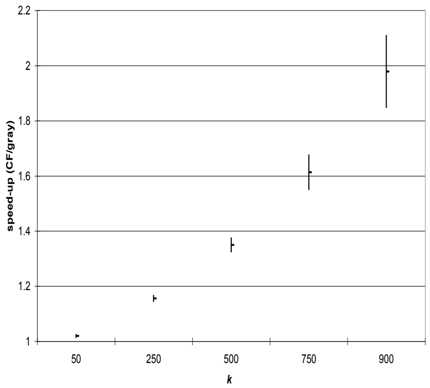
5.1 Basic Workloads

5.1.1 *Waiting Times.* For the default Zipf workload, waiting time statistics are reported in figure 3. We report the ratio of the total waiting times of LRU over gray, that is, the speed-up of gray over LRU [Hennessy and Patterson 1996], and the difference in the average waiting times of LRU minus gray. For each workload and for each value of k, thirty experiments were performed. Hence, we report both average values and confidence intervals around the average.

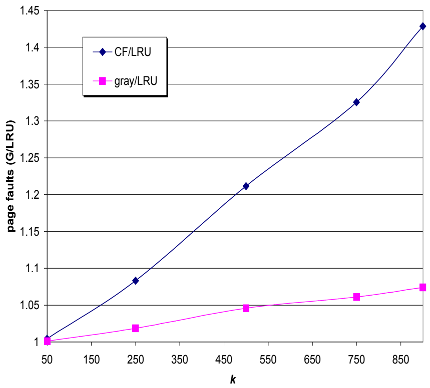
The gray algorithm was always better than LRU. Gray outperformed LRU by a factor of 1.02 to 1.41 on average. The speed-up of gray over LRU increased with k. The difference of average waiting times reaches a peak for $k = 250$ and then decreases. Standard deviations are small compared to means, and so our results can be interpreted with high confidence. For example, we can claim 99.9% confidence that the average speed-up of gray over LRU is at least 34% when $k = 900$.

The CF algorithm was worse than LRU and gray, as shown in figure 4(a). We report that CF was consistently worse than LRU and gray in all other simulations, and omit additional charts.

5.1.2 *Page Faults.* We give statistics on page fault rates in figure 4(b). The number of page faults is not the best performance measure for broadcast disk — waiting time is. However, we present page fault statistics to compare page fault rates with waiting times. We report the number of page faults for CF and gray normalized to the number of LRU page faults. CF's increase in number of page faults was bigger than CF's increase of waiting times, as compared to figure 4(a). Hence, CF spent less waiting time than LRU on each individual fault. However, the number of page faults is so much larger that its total waiting time is substantially worse than LRU's. The gray algorithm incurred more page faults than LRU as well, and the difference was as high as 7%. However, gray outperformed LRU in terms of waiting times, and so it spent less time than LRU on each page fault. In



(a) CF



(b) Page faults

Fig. 4. Chart (a) reports on the performance of LRU and CF in the default Zipf workload. The chart reports the speed-up of LRU over CF. The horizontal mark is the average speed-up and the vertical interval is the average plus or minus three times the standard deviation of the average waiting time over thirty experiments. Chart (b) gives the total number of page faults of CF and gray normalized to LRU's.

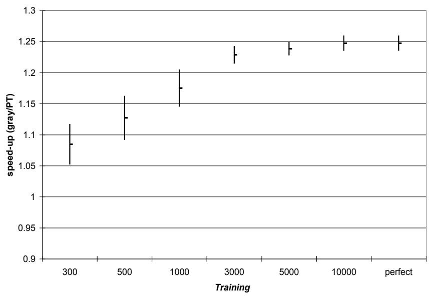
fact, the largest speed-up of gray over LRU in the waiting time metric occurs when $k = 900$, which is also when gray had the largest relative number of page faults. In conclusion, the number of page faults was not a predictor of waiting times.

5.1.3 *PT.* We now turn to compare gray with the PT algorithm [Acharya et al. 1995]. The PT algorithm depends on estimates of stationary access probabilities. Since those probabilities are not necessarily known in advance, we subject the PT algorithm to the following training period: we generate a sequence of *Training* requests according to their access probabilities and use the frequency of occurrences of page i as an estimate of the access probability p_i of page i. Then, we generate the sequence of requests on which we measure PT's waiting time. In general, PT improved with longer *Training* periods. The extent of the improvement depends on the cache size k: for the smallest value of $k = 50$, PT was always better than gray, but for larger cache sizes, PT is worse than gray for smaller *Training* intervals and becomes better than gray for larger *Training* values. Moreover, PT requires progressively longer *Training* as the cache size increases in order for it to outperform gray. We conclude that PT can be an effective caching algorithm as long as page requests are generated independently with stationary probabilities. We also observe that the stationary probability assumption becomes progressively more important for larger caches.

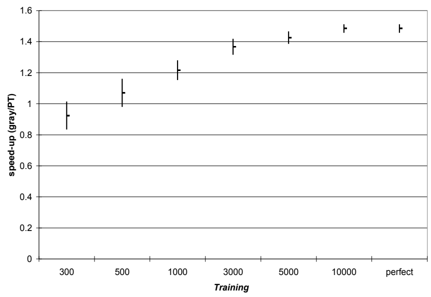
5.2 Robustness

In this section, we examine the performance of the on-line algorithms to parameter changes around the base values. We report on algorithm sensitivity to *ThinkTime*, to changes of access pattern skewness, to non-stationary access probabilities, and to region placement.

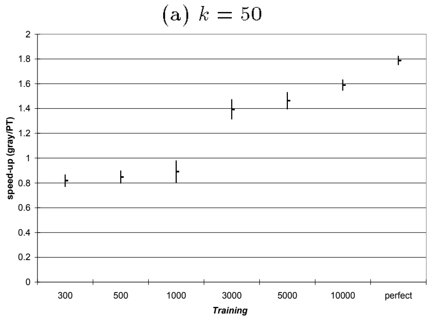
5.2.1 *ThinkTime.* Changes of *ThinkTime* are reported in figure 6. The parameter *ThinkTime* does not affect the speed-up of gray over LRU in any statistically



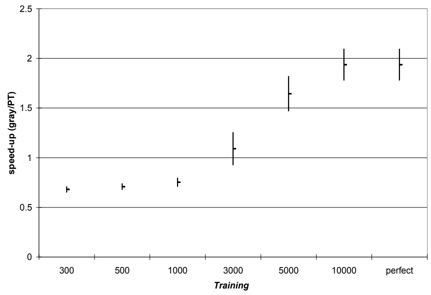
(a) $k = 50$



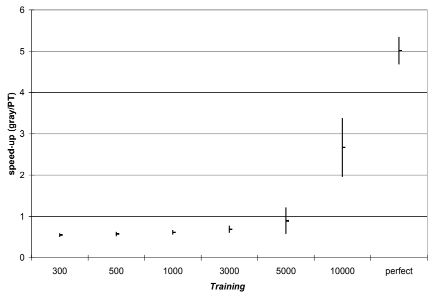
$k = 250$



(c) $k = 500$

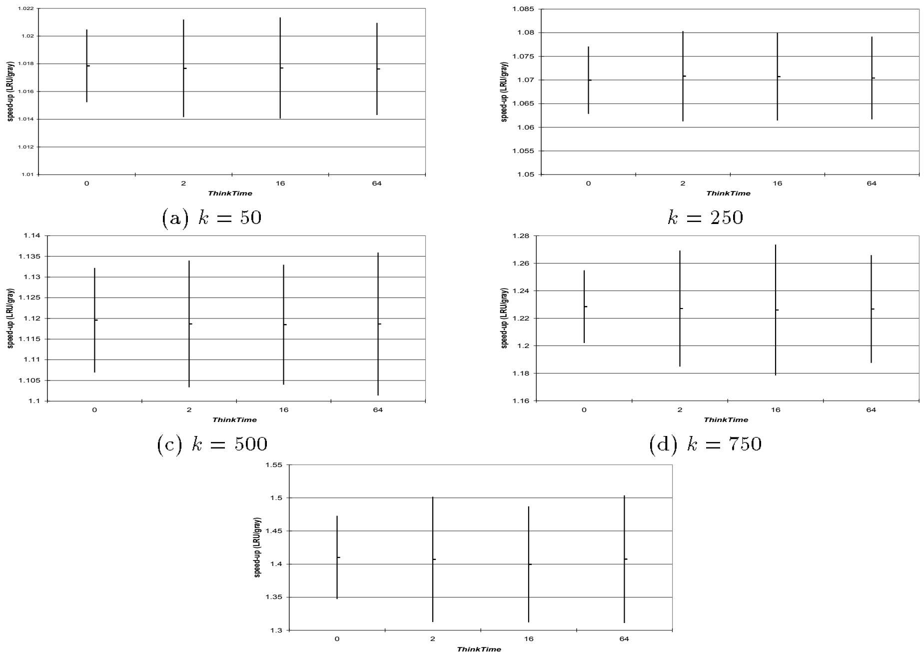


(d) $k = 750$



(e) $k = 900$

Fig. 5. Speed-up of PT over gray for various values k of cache size and various lengths of the *Training* period. The horizontal mark is the average speed-up and the vertical interval is the average plus or minus three times the standard deviation of the averages over thirty experiments. Chart scales differ.



(a) $k = 50$ $k = 250$

(c) $k = 500$ (d) $k = 750$

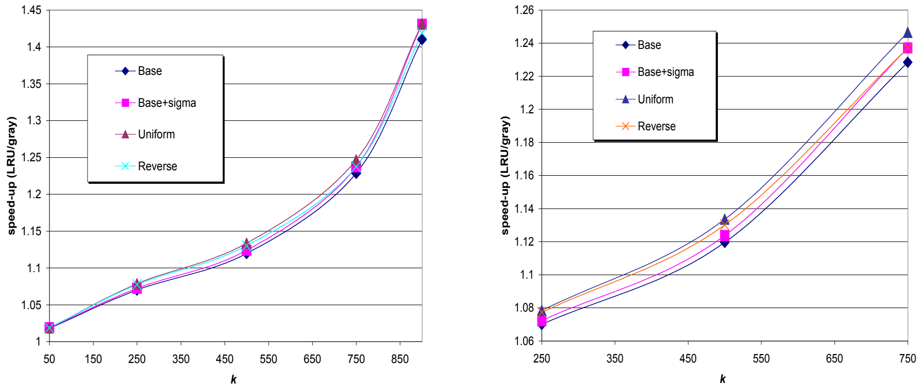
(e) $k = 900$

Fig. 6. Effect of changes of *ThinkTime* on the speed-up of gray over LRU for various values of cache size k. The horizontal mark is the average speed-up and the vertical interval is the average plus or minus three times the standard deviation of the average waiting times over thirty experiments. Chart scales differ.

significant way.

5.2.2 *Region Placement.* Figure 7 gives the average speed-up of gray over LRU for the default workload and the uniform and reverse region workloads. It also gives the average speed-up in the base configuration plus the standard deviation. The gray algorithm achieves better speed-up in the reverse and in the uniform workloads than in the base. The difference is small, but it is more than the standard deviation, and so we cannot attribute it to chance. We conclude that gray achieves some limited benefit when pages are grouped in the broadcast according to their access probabilities. By contrast, it was shown that some off-line policies benefit from scattering pages with same access probabilities throughout the broadcast [Acharya et al. 1996]. The improvement for the gray algorithm is due to the fact that when the broadcast progresses, hot pages will often be black or gray and, because of their placement, hot gray pages are either cached or quickly reloaded.

5.2.3 *Skewness.* Results for the *Random* workload and for various values of the Zipf skew parameter θ are given in figure 8 and 9. Figure 8 reports on gray performance as a function of θ, and waiting times have been normalized to their base values. As the parameter θ increases, the reference distribution becomes more and more skewed, that is, some pages become progressively hotter and others become colder. The gray algorithm improves as the the workload is more and more skewed,



(a) Overview (b) Details

Fig. 7. Effect of changes in region placement. The chart reports the average speed-up of gray over LRU in the base configuration, as well as in the uniform and reverse region placement workloads. The chart also gives the average speed-up plus the standard deviation in the base configuration. The left chart reports all data points and the right charts details the point with largest discrepancies.

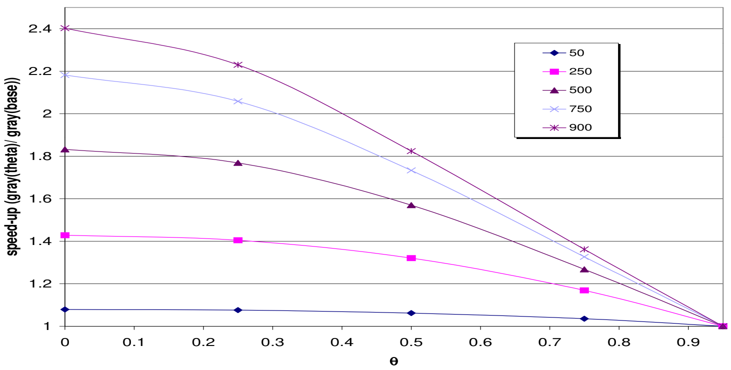
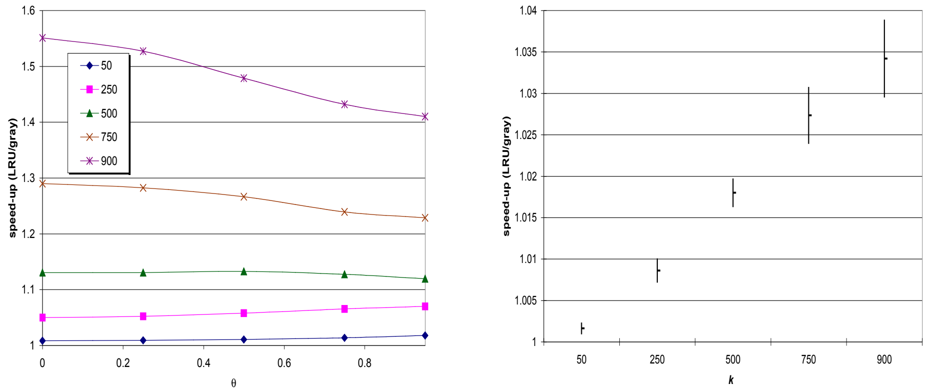


Fig. 8. Relative performance of gray for different values of θ. Waiting times of gray have been normalized to the value for $\theta = .95$. Chart lines are for cache sizes $k = 50, \ldots, 900$.



(a) θ (b) *Random*

Fig. 9. Chart (a) reports the effects of changes in the Zipf skew parameter θ. Speed-up of gray over LRU is plotted. Chart (b) gives the speed-up in the *Random* workload. The horizontal mark is the average speed-up and the vertical interval is the average plus or minus three times the standard deviation of the average waiting times over thirty experiments. Chart scales differ.

especially for larger caches. We now turn to the speed-up of gray over LRU as a function of θ. For any fixed value of θ, the speed-up increased with the cache size k. When k is fixed and θ increases, gray speed-up increased for small values k of cache size ($k \leq 250$), but decreased for larger cache sizes. The largest speed-up value (55%) was obtained for the largest cache size $k = 900$ and the smallest skew parameter $\theta = 0$. In other words, LRU took relatively better advantage than gray of a skewed access pattern when there was a larger cache. The speed-up in the *Random* workload has the same qualitative behavior as in the default Zipf workload, with gray consistently outperforming LRU and the speed-up increases with the cache size k.

5.2.4 *Access Pattern.* Figure 10 reports our findings for the case when client access pattern changes over time. The picture shows $SwitchTime = 1000$ and $Offset = 45$ (frequent and radical interest shifts) on the left and $SwitchTime = 1000$ and $Offset = 45$ (infrequent and gradual shift) on the right. The gray algorithm consistently outperformed LRU.

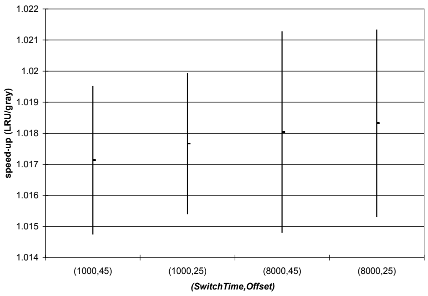
5.2.5 *Web Workloads.* Figure 11 reports speed-ups and waiting time differences of gray over LRU for various values of k. Again, gray always outperformed LRU.

5.3 Discussion

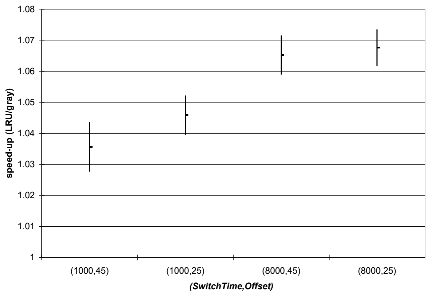
The gray algorithm always outperformed the other on-line algorithms LRU and CF. The gray algorithm was better than LRU and CF across a wide variety of synthetic workloads and Web traces. The PT algorithm proved to be effective for stationary probability distributions when the workload is preceded by a suitably long training period, with larger caches requiring longer training.

6. CACHING AND SCHEDULING

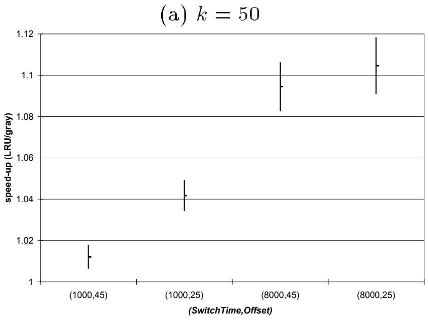
In this section, we will explore the relation between caching and scheduling.



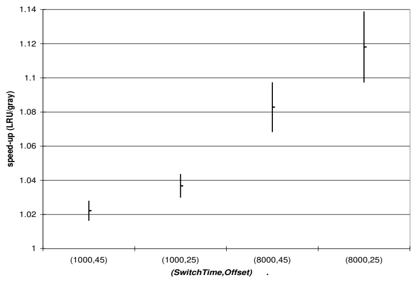
(a) $k = 50$



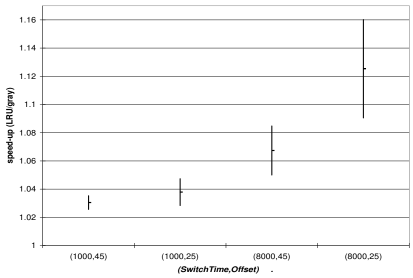
(b) $k = 250$



(c) $k = 500$

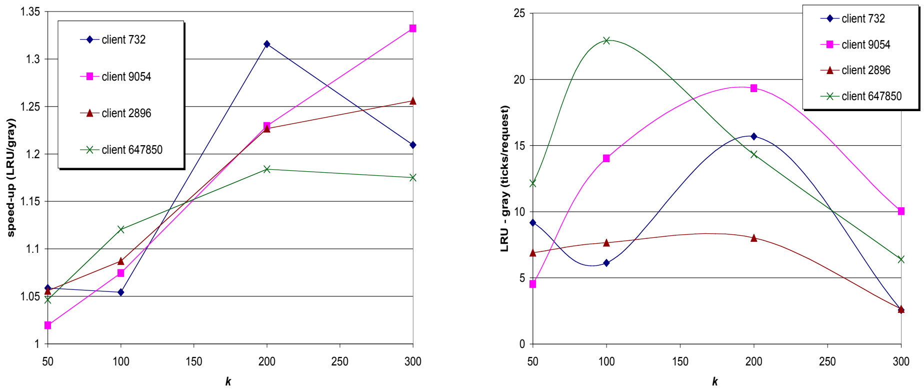


(d) $k = 750$



(e) $k = 900$

Fig. 10. Speed-up of gray over LRU for changing access patterns. Chart scales differ.



(a) Speed-up (b) Difference

Fig. 11. Performance of LRU and gray in the Web workloads. Chart (a) reports the speed-up of gray over LRU. Chart (b) gives analogous quantities for the difference of the average waiting time per request.

6.1 Background

Let p_i be the probability that page i is requested and define $\tau_i = \sqrt{p_i}/(\sum_j \sqrt{p_j})$. The *square-root law* suggests that page i should be broadcast with frequency τ_i (and not with probability p_i) [Bar-Noy et al. 1998; Gecsei 1983; Su and Tassiulas 1997]. The square-root law can be implemented as the randomized GR1 algorithm [Bar-Noy et al. 1998]: broadcast page i with probability τ_i. An important quantity in scheduling is the *stride* ξ_i, which is a random variable that represents the interval between two consecutive transmissions of page i. The GR1 algorithm does not necessarily result in a cyclical schedule. We found that GR1 was always outperformed by the MAD strategy described below. The problem of GR1 stems from the uncertainty with which pages are broadcast: it is easy to see that the expected coefficient of variation [Kleinrock 1975] of the strides is at least 0.5 for the average page. By contrast, the MAD schedule described below led to an average coefficient of variation that was less than 0.054. As a consequence, caching algorithms can obtain and exploit more precise estimates of future retransmission times. The *Mean Aggregate Delay* (MAD) algorithm is a deterministic scheduling algorithm that approximates the square-root law [Bar-Noy et al. 1998; Su and Tassiulas 1997]. The MAD algorithm maintains a value s_i associated with each page i. The quantity s_i is the number of broadcast ticks since the last time page i was broadcast. The MAD algorithm broadcasts a page i with the maximum value of $(s_i + 1)^2 p_i$. In particular, when all p_i's are equal, MAD generates a flat broadcast. MAD is a 2-approximation of the optimal schedule, and guarantees a cyclical schedule. MAD is a qualitatively different algorithm from gray, PT, or CF, in that it is not on-line (it requires probability estimates). We verified that errors in estimating the p_i's led to worse performance than that of a flat broadcast. Another issue is for caching algorithms to exploit different page strides. Clearly, a caching strategy cannot favor pages that are frequently broadcast because those pages can be loaded within a short time. However, a caching strategy cannot simply favor pages that are seldom broadcast because those pages are typically the least useful.

6.2 MAD Caches

There are three main issues in deploying MAD when clients use caches: estimation of future transmission times, laziness, and circularity of the problem.

6.2.1 *Future Transmission Times.* The MAD schedule can be generated very simply at the server site. At the client site, schedule-dependent caching strategies, such as gray, PT, or CF, can be implemented efficiently only if future transmission times (t_i values) can be estimated efficiently. Although exact t_i values are not easy to obtain from MAD quadratic recurrence, we approximate them by assuming that page i is transmitted every $\xi_i \approx 1/\tau_i$ ticks (incidentally, $1/\tau_i$ is not necessarily integer).

6.2.2 *Laziness.* Recall that on a flat broadcast, the gray algorithm is lazy [Khanna and Liberatore 2000], that is, it prefetches pages only when they can be loaded without waiting on the broadcast. If the broadcast is skewed, then the gray algorithm is not necessarily lazy. However, we will give a general condition on the set of cached gray pages that guarantees that gray is lazy, and later on we will show a natural instantiation of this general rule for MAD schedules. First, suppose that the two following *gray order conditions* hold:

(1) The gray algorithm maintains a total order on the set of gray pages (the *gray order*) and caches the first pages in such order. We will write $i \prec_t j$ if page i precedes page j in the gray order at step t.

(2) The gray order changes only when a gray page is broadcast. In this event, the relative order of non-broadcast pages remains the same, and the broadcast page advances in rank. In symbols, if page i is broadcast at step t, then $h \prec_t j$ entails $h \prec_{t+1} j$ for all $h, j \neq i$, and, if i is not requested, $i \prec_t j$ entails $i \prec_{t+1} j$.

The following proposition is both a simplification and a generalization of Lemma 5.12 in [Khanna and Liberatore 2000].

Proposition 1. *If the gray algorithm satisfies the two gray order conditions above, the gray algorithm is lazy, that is, it prefetches a page i only if i can be loaded without waiting on the broadcast.*

Proof. Suppose by contradiction that the gray algorithm waits on the broadcast to prefetch a page. Let t be the first step after which gray waits to prefetch a page, and let i be one of the pages gray is waiting to prefetch at time t. Since only gray pages are prefetched, page i is gray at step t. Since all gray pages are cached at the beginning of a phase and the gray page i is not cached at step t, step t does not mark the end of a phase. Hence, page i was gray at step $t-1$. Moreover, since step t does not mark the end of a phase, we have that $g_t \leq g_{t-1}$, where g_s is the number of gray pages at step $s = t-1, t$. Therefore, page i was not one of the first g_{t-1} pages at step $t-1$, but it is one of the first $g_t \leq g_{t-1}$ pages at step t. Hence, $r_{t-1}(i) > g_{t-1}$ and $r_t(i) \leq g_t$ where $r_s(p)$ is the *rank* of page p at step t, that is, $r_s(p) = |\{q : q \prec_s p\}| + 1$. We conclude that $r_t(i) < r_{t-1}(i)$. Since the rank of i decreases, by the second condition, at step t, the server broadcast either page i or a gray page $j \prec_{t-1} i$ that was requested and missing. Since at step t the algorithm is waiting to prefetch page i, we conclude that page i was not broadcast at step t. Therefore, at step t, the server broadcasts a gray page $j \prec_{t-1} i$ that was requested and missing. Hence, $r_{t-1}(j) > g_{t-1}$ and $g_t = g_{t-1} - 1$. Since $j \prec_{t-1} i$, we obtain $r_{t-1}(i) > r_{t-1}(j)$. Since page j becomes black and the relative order of all other gray pages does not change, we have $r_t(i) = r_{t-1}(i) - 1$. We then have $g_t = g_{t-1} - 1 < r_{t-1}(j) - 1 < r_{t-1}(i) - 1 = r_t(i)$, and so i should not be cached at step t. Hence, the gray algorithm does not attempt to load page i at step t. A contradiction is thus reached, and the proof is complete. □

Suppose that gray maintains for each page i, the time step l_i when page i was last broadcast, and orders gray pages according to the value $l_i + 1/\tau_i$. It is immediate to see that such ordering is a gray order, and so gray is lazy under this rule.

6.2.3 *Filtered Trace.* Another issue is the circularity that schedule-dependent caching strategies introduce in broadcast disk systems. Clients generate sequences of requests to data pages. Such sequences will be said to be *generated traces*. In other words, a generated trace is the actual sequence of pages needed by a client. Some requests in the generated sequence can be satisfied by the local cache, while others cannot and cause a page fault. Then, clients will access the broadcast disk for the faulting pages. The sequence of faulting pages will be said to be the *filtered trace*. Filtered traces depend on client access patterns, as well as on the paging strategy and on the value of k. On the contrary, generated patterns depend only on client access patterns. A simple, but important observation is that scheduling should take into account filtered traces rather than generated ones. Indeed, the broadcast is accessed only for pages in the filtered trace, while other page references are resolved locally. From the viewpoint of a broadcast scheduler, the filtered trace is the sequence of client requests. The filtered trace depends on the caching strategy, which in turn depends on the schedule, and a circular problem arises. We resolve the circularity by estimating p_i as the LRU fault rate on page i. Several pages will have $p_i = 0$ and MAD would never broadcast them. To obviate such problem, a new trace is formed as the filtered trace plus an initial sequence of requests for all pages. Then, p_i is estimated as being proportional to the number of occurrences of page i in the new trace. Such LRU filtering is a generalization of offset-based scheduling [Acharya et al. 1995] because hot pages are seldom broadcast if LRU fixes them in its cache. As a result, if a page i is not broadcast frequently, then i is either very hot or very cold. The gray algorithm implicitly distinguishes the two types because hot pages will be frequently marked gray whereas cold pages will be mostly unmarked.

The major effect of caching is that it broke very long tails in the distribution of faults, and so filtered traces were always less skewed than the originating generated trace. Then, the square-root law contracts the schedule skewness even more. Figure 12 reports the ratio of the average τ_i over the minimum for increasing values of k, and shows that this ratio decreases linearly with k. When $k = 900$, the average τ_i is 1.21 times the minimum, and so the schedule is nearly flat.

6.3 MAD Experiments

6.3.1 *Experimental Results.* Flat gray, MAD gray, and MAD LRU are compared against flat LRU and speed-ups are reported in figure 13. The combination of caching and scheduling brought about speed-ups of a factor of 1.8 to 2.8. Although MAD schedules are obtained from LRU filtering so that they are perfectly tailored

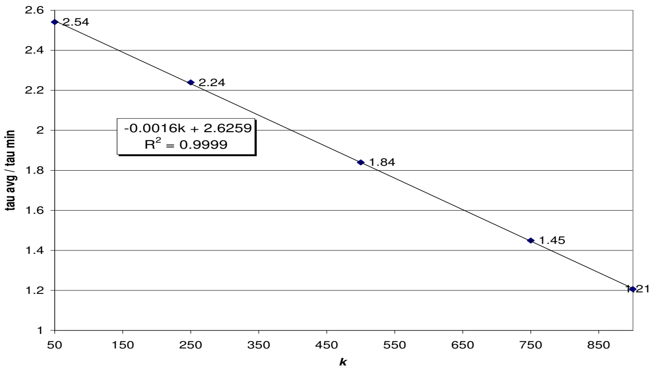


Fig. 12. Average value of τ_i over the minimum τ_i as k varies (replacement strategy is LRU). Connecting line is the least square linear interpolant $-.0016k + 2.6259$.

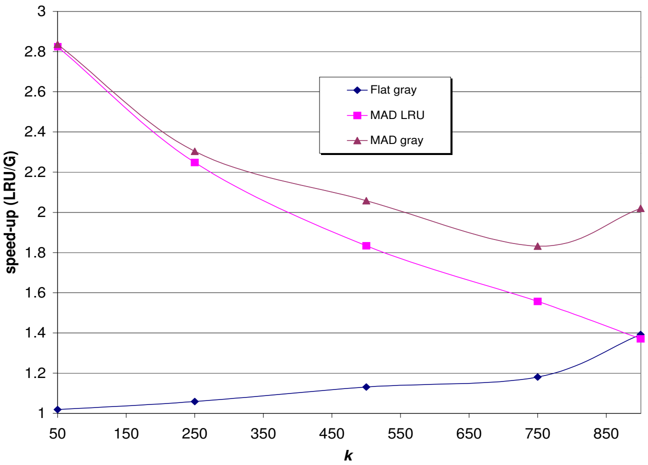
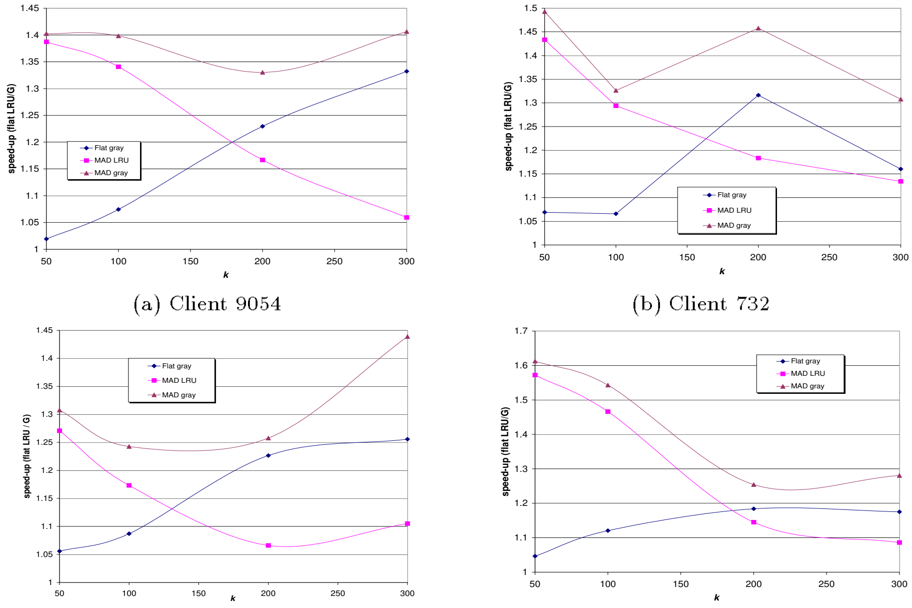


Fig. 13. Performance in the default Zipf workload of LRU on a MAD broadcast and of gray on a flat and on a MAD broadcast. For these experiments, $Length = 15000$. The speed-up is relative to the waiting time of LRU on a flat broadcast.



(a) Client 9054

(b) Client 732

(c) Client 2896

(d)Client 647850

Fig. 14. Performance in the Web workload of LRU on a MAD broadcast and of gray on a flat and on a MAD broadcast. The speed-up is relative to the waiting time of LRU on a flat broadcast. Graph scales differ.

to LRU, gray was able to take advantage of such skewed schedules to outperform MAD LRU.

Figure 14 gives flat gray's and MAD LRU's speed-up over flat LRU on the Web workloads. The major difference between the Web and the synthetic traces is that the *bcTresh* parameter excludes from the schedules pages that are not requested (i.e. $AccessRange = n$ in the Web workloads). As a result, figure 13 is qualitatively similar to the left portion of figure 14.

6.3.2 *Discussion.* The combination of MAD and gray led to substantial speed-ups (factors of 1.25 to 2.8) across all values k of cache size. MAD helped more when k was small, while gray helped more when k was large. Moreover, flat gray outperforms MAD LRU when k is large. An explanation is that MAD schedules became closer and closer to flat as k increases (figure 12). In the meanwhile, gray improved more over LRU when k increases (figure 3). The combination of a scheduling algorithm, which was effective for small caches, and a paging algorithm, which was effective for large caches, led to significant speed-ups across the entire range of k's (figures 13 and 14).

APPENDIX

A. GRAY ALGORITHM

In this section, we report the gray algorithm as defined in [Khanna and Liberatore 2000]. The gray algorithm uses a set of three marks (black, gray, white) and maintains a mark for all n pages. We define b_j to be the number of black pages immediately before the jth request. The gray algorithm works in phases:

—A new phase is started when $b_j = k$. At the beginning of a phase, mark all gray pages white and all black pages gray.

—If gray faults on page i, then it loads i and marks it black.

—Before the jth request, gray keeps in its fast memory
 —The b_j black pages, plus
 —The set of $k - b_j$ gray pages that can be reloaded with the longest waiting times.

The gray algorithm needs to keep track only of black and gray pages, as all others are white. Thus, the gray algorithm uses at most $2k$ marks.

ACKNOWLEDGMENTS

We would like to thank Swarup Acharya, Demet Aksoy, Allan Borodin, Mike Franklin, Bob Gruber, Sanjeev Khanna, and Samir Khuller for helpful conversations.

REFERENCES

Acharya, S., Alonso, R., Franklin, M., and Zdonik, S. 1995. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the 1995 ACM SIGMOD Conference International Conference on Management of Data* (1995), pp. 199–210.

Acharya, S., Franklin, M., and Zdonik, S. 1996. Prefetching from a broadcast disk. In *Proceedings of the International Conference on Data Engineering* (1996).

Almeroth, K. C., Ammar, M. H., and Fei, Z. 1998. Scalable delivery of Web pages using cyclic best-effort (UDP) multicast. In *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1998)* (1998).

Ammar, M. H. 1987. Response time in a teletext system: An individual user's perspective. *IEEE Transactions on Communication COM-35*, 11 (Nov.), 1159–1170.

Ammar, M. H. and Wong, J. W. 1985. The design of teletext broadcast cycles. *Performance Evaluation 5*, 4, 235–242.

Arlitt, M. and Jin, T. 1999. Workload characterization of the 1998 World Cup Web site. Technical Report HPL-1999-35R1 (Sept.), HP Laboratories.

Bar-Noy, A., Bhatia, R., Naor, J., and Schieber, B. 1998. Minimizing service and operation costs of periodic scheduling. In *Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms* (1998), pp. 11–20.

Bar-Noy, A., Patt-Shamir, B., and Ziper, I. 2000. Broadcast disks with polynomial cost functions. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)* (2000).

Bar-Noy, A. and Shilo, Y. 2000. Optimal broadcasting of two files over an asymmetric channel. *Journal of Parallel and Distributed Computing 60*, 4 (April), 474–493.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Eichenauer-Herrman, J. 1992. Inversive congruential pseudorandom numbers: a tutorial. *International Statistical Review 60*, 2 (Aug.), 167–176.

Eichenauer-Herrman, J. 1995. Pseudorandom number generation by nonlinear methods. *International Statistical Review 63*, 2 (Aug.), 247–255.

Franklin, M. and Zdonik, S. 1997. A framework for scalable dissemination-based systems. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (1997), pp. 94–105.

Gecsei, J. 1983. *The architecture of videotext systems.* Prentice-Hall, Englewood Cliffs, NJ.

Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD Conference International Conference on Management of Data* (1994), pp. 243–252.

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture: A Quantitative Approach* (Second ed.). Morgan Kaufmann, San Mateo, CA.

Kenyon, C. and Schabanel, N. 1999. The data broadcast problem with non-uniform transmission times. In *Proceedings of the Tenth ACM-SIAM Symposium on Discrete Algorithms* (1999), pp. 547–556.

Kenyon, C., Schabanel, N., and Young, N. 2000. Polynomial-time approximation scheme for data broadcast. In *Proceedings of the Thirtisecond ACM Symposium on the Theory of Computing* (2000).

Khanna, S. and Liberatore, V. 2000. On broadcast disk paging. *SIAM Journal on Computing 29*, 5, 1683–1702.

Khanna, S. and Zhou, S. 1998. On indexed data broadcast. In *Proceedings of the Thirtieth ACM Symposium on the Theory of Computing* (1998), pp. 463–472.

Kleinrock, L. 1975. *Queueing Systems*, Volume 1. Wiley.

Knuth, D. E. 1973. *The Art of Computer Programming*, Volume 3. Addison-Wesley, Reading, MA.

Leeb, H. and Wegenkittl, S. 1997. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transaction on Modeling and Computer Simulation 7*, 2 (April), 272–286.

Liberatore, V. 1999. Empirical investigation of the Markov reference model. In *Proceedings of the Tenth ACM-SIAM Symposium on Discrete Algorithms* (1999), pp. 653–662.

Matsumoto, M. and Nishimura, T. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation 8*, 1 (Jan.), 3–30.

Overmars, M. H. 1983. *The design of dynamic data structures.* Springer-Verlag, Berlin.

Shanmugasundaram, J., Nithrakashyap, A., Sivasankaran, R., and Ramamritham, K. 1999. Efficient concurrency control for broadcast environments. In *ACM SIGMOD International Conference on Management of Data* (1999).

Stathatos, K., Roussopoulos, N., and Baras, J. S. 1997. Adaptive data broadcast in hybrid networks. In *Proc. 23rd International Conference on Very Large DataBases* (1997), pp. 326–335.

Su, C. J. and Tassiulas, L. 1997. Broadcast scheduling for information distribution. In *Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1997)* (1997).

Tanenbaum, A. S. 1992. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ.

Tassiulas, L. and Su, C. J. 1997. Optimal memory management strategies for a mobile user in a broadcast data delivery system. *IEEE Journal on Selected Areas in Communications.*

Vaidya, N. and Hameed, S. 1997. Log time algorithms for scheduling single and multiple channel data broadcast. In *Proc. of the 3rd ACM/IEEE Conf. on Mobile Computing and Networking* (1997).