Caching and Scheduling for Broadcast Disk Systems

Vincenzo Liberatore[1]

Institute for Advanced Computer Studies

[1]UMIACS, A. V. Williams Building, University of Maryland, College Park, MD 20742. E-mail: vliberatore@acm.org. URL: http://www.umiacs.umd.edu/users/liberato/.

Abstract

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 access probabilities. In this paper, we subject on-line paging algorithms to extensive empirical investigation. The Gray algorithm [25] always outperformed other on-line strategies on both synthetic and Web traces. Moreover, caching limited the skewness needed from a broadcast schedule, and led to favor efficient caching algorithms over refined scheduling strategies when the cache was not small. Prior to this paper, no work had empirically investigated on-line paging algorithms and their relation with server 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. 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* 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 site. Broadcast push is being incorporated in several commercial systems. For example, Hughes Network System [1] delivers Web pages via satellite links, and Hybrid Networks Inc. [2] will broadcast data via cable lines.

Broadcast Disks [20] attempt to improve broadcast push performance by the combination of two methods: they establish client caching and fix a cyclical broadcast schedule over long periods of time. Clients are helped by local caching 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 [4, 25]. Moreover, cyclical schedules lead to scalable and widely supported multicast techniques over the Internet [8] and are necessary in a mobile environment where clients need to know when to tune in to receive data [26]. Our main contributions are the first empirical study of on-line algorithms for broadcast disk caching, and the first analysis of the interaction between client caching and broadcast scheduling.

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.
- Prefetching can sometimes be executed at no cost for servers and clients [5].

Previous broadcast disk paging algorithms assumed that clients requested data items with given probabilities and that those probabilities were known to paging strategies [4, 5, 34]. In practice, probabilistic assumptions could be difficult to find and to validate, and so it is critical to have efficient paging strategies that operate without probabilistic parameters. A breakthrough came with the *Gray algorithm*, which works with no probabilistic assumption and that is provably optimal in terms of worst-case ratios [25]. In this paper, we subject on-line paging algorithms to extensive empirical investigation on both synthetic and Web traces. On-line paging algorithms had not previously been studied experimentally. The Gray algorithm always outperformed classical on-line strategies, and thus it is the first truly on-line paging strategy algorithm to offer performance improvements in broadcast disk systems.

Caching alters patterns of client accesses to non-local data because some data requests can be resolved locally. As a result, caching changes the frequency with which data items are accessed from the server broadcast. Caching cut long tails of the access distribution in our experiments. 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 [12, 31]. Scheduling is strongly interrelated with caching because client access patterns are modified by caching. Conversely, caching strategy are affected by scheduling because a schedule determines the cost for loading pages. In this paper, we investigate trade-offs between scheduling and caching. In no previous paper had scheduling been investigated in relation to the presence of caches at the client sites. Scheduling provides performance improvements when caches are small. However, schedules need not be very skewed when larger caches are used, and in this case efficient caching algorithms should be favored to refined broadcast schedules.

2 The Broadcasting Environment

The *broadcast disk* environment is well-known in the literature [4, 5, 20, 25] and we only outline it here. A database of *ServerDBSize* pages is cyclically broadcast by a server[1]. Pages have all the same size and it

[1]The *ServerDBSize* pages need not be the whole server database, but they can simply be the database portion that the server has assigned for broadcast (see e.g. [8, 30]). The important assumption is that the broadcast data set changes so slowly

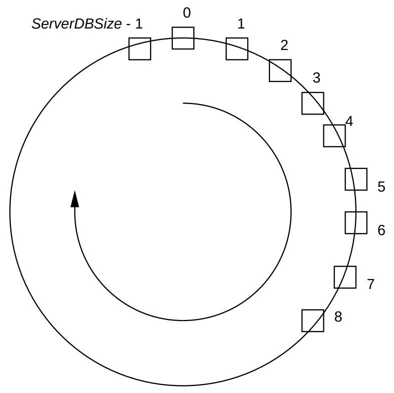


Figure 1: Example of a flat broadcast program. Pages are numbered from 0 to $ServerDBSize - 1$, and are cyclically transmitted by the server in that order.

takes the same amount of time to broadcast any page. At first, we will assume that pages are broadcast with the same frequency (*flat broadcast*), and we will discuss this assumption later on. We will say that a *broadcast tick* is the time needed to transmit a page. We will say that a *rotation* is the time needed to transmit the whole server database. Therefore, a rotation is *ServerDBSize* 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 $CacheSize < ServerDBSize$ pages. As in previous papers [4, 5, 25], we consider a special environment that 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 cannot communicate with each other, and so clients cannot exchange pages and cannot devise a common strategy.
- Clients receive pages only from the server broadcast. Such assumption is justified either when the broadcast channel is the only communication link between clients and server, or when additional channels exist, but the server refuses to replicate broadcast data on those channels.

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 the broadcast disk environment, each clients 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 other 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 *prefetches* 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 for free. 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 our definition, they would be prefetched. However, no time is wasted to load those pages as they are loaded while waiting for another page.

3 Page Replacement Strategies

In this paper, we will compare three different page replacement strategies. All strategies are *completely on-line*, that is, they do not assume neither knowledge of future page requests nor knowledge of a probability

that we can assume it remains constant for the duration of our simulations.

distribution over pages. We will also compare the three on-line strategies with PT [5], which is not on-line because it uses access probabilities. PT is included as a point of comparison for the other algorithms. We will define each algorithm and then describe our efficient implementation for the case of a flat broadcast schedule.

3.1 LRU and CF

3.1.1 Algorithm Definition

We now describe two completely on-line strategies: LRU and CF. The algorithm *LRU* responds to a miss by evicting the page that has been used least recently. 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. Both LRU and CF are completely on-line algorithms. 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 [33], 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 is a classic paging algorithm [33], while CF is a new, but natural, paging algorithm for broadcast disks.

3.1.2 Implementation

LRU can be implemented with a (binary) heap of size *CacheSize* that contains the cache elements ordered by the step of most recent usage. When LRU faults, it removes the top of the heap and inserts the newly requested page. Thus, LRU takes $O(\log \mathit{CacheSize})$ time per fault.

CF's implementation maintains the cache as a red-black tree [16] ordered by transmission times. When CF faults on page p, CF inserts p in the red-black trees. Then, CF tries 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 CF determines the minimum element in the tree and removes it. All these operations take $O(\log \mathit{CacheSize})$ time, and thus CF takes $O(\log \mathit{CacheSize})$ time per fault.

3.2 The Gray Algorithm

The *Gray* algorithm [25] (U.S. patent pending) combines LRU's history with CF's waiting times, executes prefetching, and is completely on-line. It has been shown that, in terms of worst-case performance ratio, Gray outperforms LRU by a factor proportional to $\mathit{CacheSize}/\log \mathit{CacheSize}$ and that in fact Gray achieves the best possible worst-case performance ratio [25]. In this paper, we show that Gray outperforms LRU also in simulations. For the sake of clarity, we will not present the complete algorithm immediately, but we will define it by steps. We will also explain the intuition behind Gray as we progressively shape the algorithm.

First, we consider a version of LRU that maintains only one bit for each page. At the very beginning, all pages are unmarked. When a page is requested it is marked. When a page fault occurs, an unmarked page is evicted from the cache. The 1-bit LRU algorithm proceeds in this way, and finally it replaces in the cache all unmarked pages with marked pages. At this point, we say that a phase ends: the algorithm unmarks all marked pages and starts another phase afresh. We remark that 1-bit LRU evicts unmarked pages in any arbitrary order (the 1-bit LRU algorithm is also known as the *marking algorithm* [23]). A theoretical result establishes that, in terms of number of page faults, the worst-case performance ratio of 1-bit LRU is exactly the same as LRU's [23]. In other words, one bit per page achieves the same worst-case performance ratio as the regular LRU algorithm. We remark that such result holds only in the worst case and when the only cost metric is the number of page faults. In broadcast disks, the cost structure is more complex, as the cost of a fault on page p is the time spent waiting for p. Since CF takes into account the different costs of reloading different pages, we will integrate 1-bit LRU with CF. In 1-bit LRU, unmarked pages can be evicted

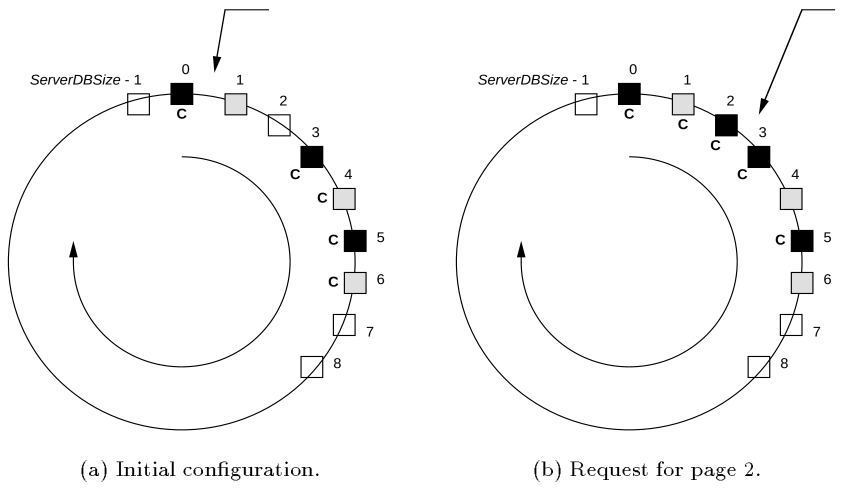


Figure 2: Gray's cache in the examples. The initial configuration is depicted in (a), and the configuration after a request for page 2 in (b). Pages are numbered from 0 to $\mathit{ServerDBSize} - 1$ and are transmitted in clockwise direction. Cached pages are marked by a **C**. The outer arrow gives the current time step reached along the transmission schedule.

in any arbitrary order. Therefore, we could combine 1-bit LRU with CF as follows: the unmarked page to be evicted should be the one that will be retransmitted earliest in the future. The current version of (1-bit LRU + CF) works as follows. At the very beginning all pages are unmarked. When a page is requested, it is marked. When a page fault occurs, the algorithm evicts the unmarked page that will be retransmitted the soonest in the future. When all pages in the cache are marked, the algorithm unmarks all pages and a new phase starts. In this way, the algorithm combines the best worst-case page fault ratio of (1-bit) LRU with the more complex cost structure of broadcast disks that is exploited by CF. However, at this point, our algorithm still does not take advantage of prefetching[2]. The Gray algorithm will use prefetching to keep a dynamically changing set of unmarked pages in cache. Specifically, Gray will maintain a color with each of the *ServerDBSize* pages in the server database (later, we will show how to reduce the number of marks to $2 \cdot \mathit{CacheSize}$). Pages that are marked by 1-bit LRU will be colored black. Pages that were black in the previous phase (i.e. that were requested during the previous phase) are marked gray, and all other pages are white. The Gray algorithm works as follows. Initially, all pages in the cache are gray and all other pages are white. When a page is requested, it is marked black. When all pages in the cache are black, the algorithm changes the color of gray pages to white and the color of black pages to gray, and starts a new phase. At each step, the algorithm keeps in the cache all the black pages, plus the set of gray pages that are furthest away along the transmission schedule. In other words, the set of uncached gray pages are the gray pages that can be reloaded with the smallest waiting time.

Example. In figure 2(a), $\mathit{CacheSize} = 5$, page 0 has just been received, page 0, 3, 5 are black and page 1, 4, 6 are gray. So, it will take 1, 4, and 6 broadcast ticks respectively before the gray pages are received again. Then, the Gray algorithm will have in its cache page 0, 3, 5 (all black pages), and have two more slots for gray pages. The Gray algorithm will then have also page 4 and 6 (the two gray pages that are furthest away along the transmission schedule from the current time step).

Again, we observe that Gray combines 1-bit LRU with CF. However, the Gray algorithm changes dynamically the set of Gray pages that reside in the cache, and so Gray needs to execute prefetching in order to keep the right set of pages in the cache at each step.

[2]We verified that the algorithm (1-bit LRU + CF) is almost always worse than LRU.

Example. Let us suppose that a request for page 2 arrives. Gray waits for two time units, loads page 2 and marks it black. Now the set of black pages is 0, 2, 3, 5, and there is only one slot for a gray page. Among gray pages, the Gray algorithm will choose to cache the one that is furthest away along the transmission schedule. It will take 2 broadcast ticks before page 4 is broadcast again, 4 broadcast ticks before page 6 is broadcast again, and $\mathit{ServerDBSize} - 1$ broadcast ticks before page 1 is broadcast again. So, the cached gray page is page 1, as shown in figure 2(b). On the whole, when page 2 was requested, Gray waited one time unit and loaded page 1 at the expenses of (say) page 4. Then, it received page 2 and evicted page 6. Page 1 was loaded without being requested, which is to say that page 1 was prefetched. Moreover, no time is spent waiting for page 1 because page 1 was loaded while waiting for page 2.

It can be shown that the right set of cached gray pages can be maintained by using prefetching, and so no additional time is to prefetch gray pages [25]. Intuitively, the reason is that, after a gray page has been broadcast, it immediately becomes the page that will be retransmitted the furthest in the future. Only the set of black and gray marks have to be maintained, and so the Gray algorithm needs only to maintain two sets of size at most *CacheSize* rather than *ServerDBSize* marks. The sets of black and gray pages are Gray's candidate set from which cached pages are selected. In general, prefetching algorithms, such as Gray or PT, maintain a candidate set of more than *CacheSize* pages, among which they select the *CacheSize* cached pages, and enforce their decision through prefetching.

3.2.1 Implementation

The Gray algorithm can be implemented to run in amortized $O(\log \mathit{CacheSize})$ time per request by using *order-statistic trees* [16]. Gray's implementation is worse than LRU's or CF's that use $O(\log \mathit{CacheSize})$ time per fault and $O(1)$ time on any other request. At any rate, Gray implementation would be especially useful in long simulations — in practice, the Gray algorithm executes only a constant number of operations during each broadcast tick. The implementation maintains gray pages in an order-statistic tree. When Gray faults, it searches the gray page that immediately follows the faulting page and determines its rank. Gray can now determine the maximum rank of a cached gray page. On a request for a non-black page, Gray looks it up in the order-statistic tree, determines its rank and decides if that page is cached or not.

3.3 PT

3.3.1 The PT Algorithm

The last replacement strategy we consider is PT [5]. 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. At each broadcast tick, PT maintains a *candidate set* consisting of the pages in the cache plus the currently broadcast page. The candidate set can have either *CacheSize* or *CacheSize* + 1 pages. The candidate set has *CacheSize* pages if the currently broadcast page is in the cache, and has size *CacheSize* + 1 otherwise. PT always keeps in the cache pages from the candidate set. If the candidate set has *CacheSize* pages, no further decision is required. However, if the candidate set has *CacheSize* + 1 pages, a subset of *CacheSize* pages has to be chosen. PT compares the values of $p_i t_i$ for all pages in the candidate set, and keeps in the cache the *CacheSize* pages with the largest value of $p_i t_i$. Intuitively, PT should minimize the expected cost per fault. Another characteristic of PT is that it executes prefetching, as we turn now to show. If the candidate set has size *CacheSize* + 1 and the page that is currently transmitted does not have the smallest value of $p_i t_i$, then the currently transmitted page is loaded in the cache, or, in other words, it is prefetched. There are some problems with the algorithm PT. In the first place, PT assumes knowledge of page accesses probabilities. The second problem is that PT uses $\Omega(\mathit{ServerDBSize})$ space to maintain those probability values. Finally, the running time of PT is much worse than LRU's, as we will discuss next.

3.3.2 Implementation

A previous implementation of PT takes $O(\mathit{CacheSize})$ time on each broadcast tick [5]. Therefore, when a page fault forces PT to wait for *Wait* broadcast ticks, PT's cost is $O(\mathit{CacheSize} \cdot \mathit{Wait})$ time. We will now

Parameter	**Description**	**Base Value**
ServerDBSize	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
CacheSize	client cache size	50,250,500,750,875
Length	trace length	15000

Table 1: Parameters used to generate synthetic workloads.

give a new implementation of PT. Our implementation uses a well-known algorithm in a trivial way and reduces the running time to $O(\mathit{CacheSize} + \mathit{Wait})$ per fault. Our implementation is based on the selection algorithm. The *selection algorithm* takes as input a set of n distinct elements and finds the kth largest element in time $O(n)$ [16]. In our simulator, we used the randomized version of the selection algorithm. We now describe how the selection algorithm can be used for an efficient implementation of PT. 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 $\mathit{CacheSize} - 1$ pages in C that have the largest value of $p_i t_i$. The new definition of PT suggests the following implementation. PT could invoke the selection algorithm with $n = |C|$ and $k = \mathit{CacheSize} - 1$ to determine the set of cached pages. However, the selection algorithm requires that all $p_i t_i$ values be distinct. We obviate the problem by numbering the elements in C from 1 to $|C|$. Then, we break ties among $p_i t_i$ values by order number and run the selection algorithm. In conclusion, PT takes $O(n) = O(|C|) = O(\mathit{CacheSize} + \mathit{Wait})$ time on each fault. Although this implementation improves on the previous one from quadratic to linear, PT's running time is at least exponentially worse than LRU's $O(\log \mathit{CacheSize})$ time per fault.

4 Simulation Set-up

In this section, we will describe how we set up the simulation of the page replacement strategies. Most of the environment was described in §2. Here, we will focus on the workloads at the client site. We defer some details to appendix A. The parameters that define the following workloads are stated in table 1. Most of these parameters are the same as those in previous papers [4, 5].

4.1 Basic Workloads

Our first workload is *Random*. In this workload, we extract a sequence of *Length* page requests uniformly at random from the server database. The *Random* workload is not likely to be representative of a realistic client workload. For example, *Random* lacks any form of locality. We include it because it is a natural workload and because we suspected that *Random* was going to tax our algorithms more than any other stochastic workload.

Our second workload assumes a stationary Zipf distribution and is similar to the one defined in the existing literature on broadcast disks [4, 5]. The Zipf distribution is often used to model skewed access patterns because it gives some pages a higher probability of being requested [28]. The synthetic trace is generated as follows. At the very beginning, a set of $\mathit{AccessRange} < \mathit{ServerDBSize}$ pages is extracted 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. The access range is then partitioned into *NumRegions* regions of equal size $\mathit{RegionSize} = \mathit{AccessRange} \ / \ \mathit{NumRegions}$. At this point, the generation of a page sequence begins. First, we extract a region according to a Zipf distribution with parameter $\theta = 0.95$. In other words, the probability that region r is extracted is proportional to $1/r^\theta$ (r ranges between 1 and *NumRegions*). Then, we extract a page uniformly at random from the chosen region. The process is then repeated to generate a sequence of *Length* page requests. We will say that such traces are generated by the *default Zipf* workload. While pages in the same region have the same probability of being requested, pages in different region have different probabilities. We will say that a region is *hotter* than another if its pages have a higher probability of being accessed.

4.2 Robustness

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

First, we measured the robustness to changes in the Zipf parameter θ. When $\theta = 0$, we have a uniform distribution among regions, and no region is hotter or colder than any other region. Such workload is very similar to *Random* except that we use only *AccessRange* pages rather than the whole server database. As θ increases, the distribution becomes more and more skewed. We executed experiments for $\theta = 0, .25, .5, .75, .95$.

The second parameter change was region placement. In the default workload, we defined regions to be disjoint sets extracted uniformly at random from the server database. We intended to measure the algorithm performance when the placement of pages into region is not random, but follows a regular pattern related to access probabilities. In the *Uniform Region* workload, the regions are consecutive intervals of pages. The coldest region are the first in the broadcast schedule and the hottest are the latest.

Example. If we have three regions of size $RegionSize = 5$ (and so $AccessRange = 15$), then the uniform regions are given by the sets of pages $R_3 = \{0, 1, \ldots, 4\}$, $R_2 = \{5, 6, \ldots, 9\}$, and $R_1 = \{10, 11, \ldots, 14\}$. The probability that R_1 is selected is proportional to 1, that R_2 is selected is proportional to $1/2^\theta$, and that R_3 is selected is proportional to $1/3^\theta$.

The *Reverse Region* workload is identical to the uniform region workload, except that hot regions precede cold regions.

In the Zipf workloads above, access probabilities do not change with time. We intend to measure algorithm performance when client interests shift over time. We will model 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, we discard *Offset* pages, and we replace them with a new set of *Offset* pages. Regions will again be disjoint after the shift, but we will allow a discarded page to be extracted for the same or for another region. We experimented with $Offset = 45$, which corresponds to a radical shift of 90% of *RegionSize*, and with $Offset = 25$, which replaces only half a region. We also chose $SwitchTime = 1000$, which induces 15 shifts per trace, and with a milder $SwitchTime = 8000$, which changes the access pattern only once 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 Web Workload

We also executed experiments with two Web server traces. The simulation has significance in the context of information dissemination over the Internet when clients are intermediate information brokers [20]. The first trace was epa-http[3]. We discarded references to URLs containing a question mark and we identified URLs that can be syntactically determined to correspond to the same page (e.g. `~liberato` and `~liberato/index.html`). The second Web trace was the first half of the August 95 NASA trace[4].

5 Experimental Results

In this section, we present the results of our experimental comparison among LRU, CF, and Gray. We will also compare our on-line algorithms with PT. We will measure the waiting times of our heuristics in terms of broadcast ticks. Therefore, our measurements scale with channel bandwidth and express fundamental

[3] The trace is available at `http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html`.

[4] The trace is available at `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`.

	LRU			Gray		speed-up		Δ	
CacheSize	mean	σ	**CF%**	mean	σ	mean	σ	mean	σ
50	6644	32	0.13%	6523	35	1.9%	0.3%	121	18
250	3927	46	7.9%	3673	44	6.9%	0.6%	254	21
500	1909	31	19%	1704	29	12%	1%	204	19
750	692	20	30%	566	18	22%	2%	125	10
875	269	17	36%	198	10	36%	6%	71	13

Table 2: Performance of LRU, CF, and Gray in the default Zipf workload. The LRU and Gray column report the average cost and sample standard deviation of LRU and Gray over a sequence of thirty trials. Costs are given in number of rotation (1 rotation = *ServerDBSize* broadcast ticks). The CF column gives the percentage cost increase of CF over LRU; negative percentages correspond to cost reduction. The speed-up column gives the average and sample standard deviation of the speed-up of Gray over LRU. Positive percentage correspond to an improvement over LRU. The Δ column gives the average and standard deviation of the cost difference of Gray over LRU.

trade-offs among heuristics. In this section, we will give results for a flat broadcast, and we will discuss non-flat schedules in §6.

On the *Random* workload, Gray always outperformed LRU and CF. We defer further details to appendix B. We will now turn to the default Zipf workload.

5.1 Zipf Workloads

5.1.1 Waiting Times

We report the waiting time statistics in table 2. The first column gives the value of the *CacheSize* parameter. The LRU and Gray columns give the cost of LRU and Gray. Costs are expressed in number of rotations (1 rotation = *ServerDBSize* ticks). For each workload and for each value of *CacheSize*, we executed thirty experiments, as described in appendix A. The mean column reports the average cost of LRU and Gray in number of rotations, and the σ column reports the sample standard deviation. The CF% column gives the percentage waiting time increase of CF over LRU; negative percentages would represent improvements of CF over LRU. The speed-up and Δ columns are defined as follows. We executed thirty trials for each combination of workload and *CacheSize*. For each experiment $i = 1, 2, \ldots, 30$, we computed the performance difference Δ_i of Gray over LRU and the speed-up $speed - up_i$ of Gray over LRU. We then took the average $speed - up_i$ and its standard deviation and reported it under column speed-up. We took the average Δ_i and its standard deviation and reported it under column Δ. Although table entries are rounded to number of rotations, we computed speed-ups and Δ's with the exact number of broadcast ticks.

CF's waiting times were always worse than LRU's and increase with *CacheSize* from 0.1% to 36%. The mean waiting times of LRU and Gray were less than for the *Random* workload (compare table 2 with table 5). We were confirmed our belief that the *Random* workload is harder on page replacement strategies than a skewed access pattern. The Gray algorithm was always better than LRU. Gray outperformed LRU by 2% to 36% (column speed-up). The speed-up of Gray over LRU increased with *CacheSize*. Moreover, the value of the standard deviation is small compared to the mean. Consider for example *CacheSize* = 875. In this case, the mean speed-up is 36% and the standard deviation is 6%. Therefore, we can make the following claim: For 99.9% of all Zipf traces, the speed-up of Gray over LRU will exceed 36% -3σ = 18%. Analogous claims can be made for all other experiments. So, Gray is not only superior to LRU on average, but it is also consistently superior to LRU with high probability.

Similar considerations hold also for the cost difference Δ. The only difference between Δ and speed-up is that while speed-up increases with *CacheSize*, Δ increases to a peak and then decreases. The percentage difference keeps increasing because when Δ starts decreasing, the absolute cost also decreases.

Trace	*bcThresh*	*ServerDBSize*	*Length*
epa-http	1	2674	43845
	8	630	37016
NASA 95	32	1036	603907
	64	734	589701

Table 3: Characteristics of the Web traces. *bcThresh* is the broadcast threshold: only pages referenced more than *bcThresh* times are broadcast. *ServerDBSize* is the number of distinct pages in the broadcast, and *Length* is the length of the resulting trace.

5.1.2 Robustness

We defer further results to appendix C. In the appendix, we will report page fault statistics, a comparison of PT with LRU, CF, and Gray, and we will analyze the sensitivity of our results to skewness, changes in access patterns, and region placement. Gray always outperformed CF and LRU on average and with high probability.

5.2 Web Workloads

We turn now to examine the algorithm performance on the Web traces. CF was severly outmatched by the other strategies, and so we do not report its performance here. In Internet data delivery, pages that are referenced sporadically are not usually broadcast [8, 30] (as, for example, in the motivating examples in the introduction). We insert in the broadcast disk schedule only those pages that are broadcast more than *bcThresh* times. Different values of *bcThresh* entail a different *Length* of the resulting trace and a different number of distinct pages (*ServerDBSize*) that appear in the broadcast. We report the resulting values of those parameters in table 3. The NASA trace is longer than epa-http and we chose larger *bcThresh* values. For the two different values of *bcThresh*, the *ServerDBSize* is very different, whereas the trace *Length* is much closer. Therefore, there are many files that are accessed very few times. Since *ServerDBSize* is different in the two traces, we performed experiments for different sets of *CacheSize* values. Figure 3 and 4 report percentage speed-ups of Gray over LRU for various *CacheSize*s.

Again, Gray always outperformed LRU. In the epa-http trace, the speed-up is modest when $bcThresh = 1$, but becomes very significant when $bcThresh = 8$, especially for larger caches. Gray significantly outperformed LRU also on the NASA trace.

5.3 Flat Disks: Discussion

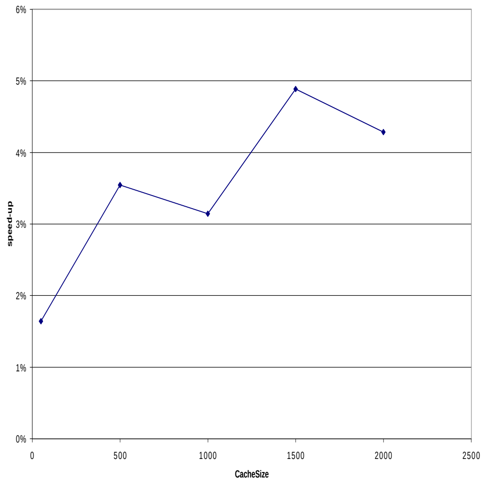
The Gray algorithm always outperformed the other on-line algorithms LRU and CF. The Gray algorithm was better than LRU and CF across a wide variety of synthetic workloads and two Web traces. Gray exhibited significant speed-up both in the average and with high probability.

While Gray was always better than LRU, there are a few factors that reduced the gap between the two algorithms:

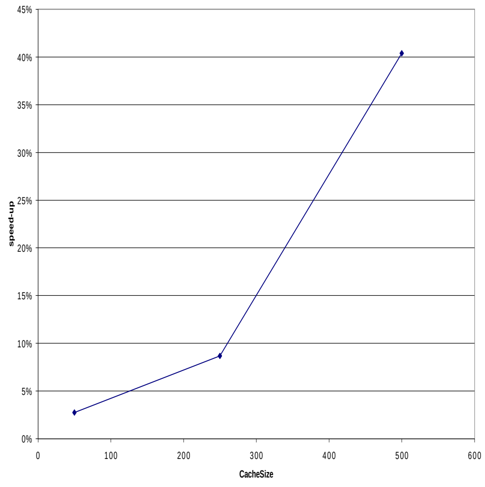
Small caches In general, Gray's speed-up increases with *CacheSize*. Small caches gave rise to the smallest speed-up percentages in almost all workloads. The only exception was when $SwitchTime = 1000$, where the speed-up showed an irregular behavior. In all other workloads, small caches corresponded to small speed-up values.

Randomness and small caches More random workloads were harder for all algorithms. Moreover, more random workloads resulted in reduced speed-up values when $CacheSize = 50$ or $CacheSize = 250$.

Skewness and large caches When we fixed the *CacheSize* value and increased the skewness θ, Gray's speed-up diminished.

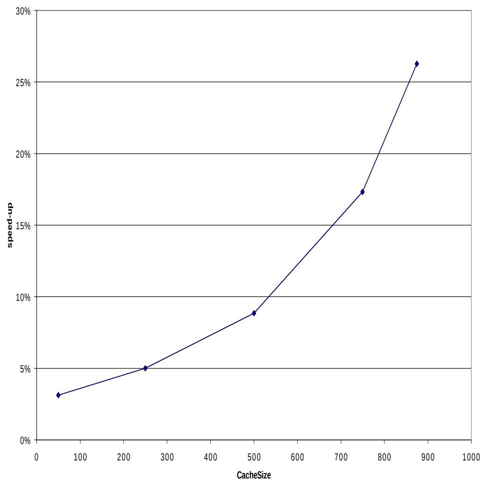


(a) epa-http ($bcThresh$ = 1)

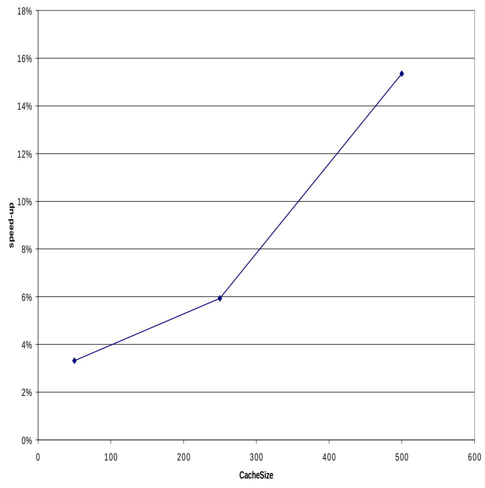


(b) epa-http ($bcThresh$ = 8)

Figure 3: Percentage speed-up of Gray over LRU on the epa-http Web trace. Graph scales differ.



(a) NASA 95 ($bcThresh$ = 32)



(b) NASA 95 ($bcThresh$ = 64)

Figure 4: Percentage speed-up of Gray over LRU on the NASA 95 traces. Graph scales differ.

Frequent and radical interest shifts As the access patterns shifted more frequently or more radically, the speed-up of Gray over LRU dropped.

We remark that these factors resulted in a reduction of Gray's speed-up, but they never resulted in LRU outperforming Gray. Finally, we found that Gray and LRU are not sensitive to the placement of pages in the broadcast schedule.

6 Caching and Scheduling

In this section, we will explore the relation between caching and scheduling. Caching and scheduling affect one another. A scheduling policy determines the cost of loading a page, and thus affects the caching policy. However, caching affects scheduling as well. For example, suppose that page p is hot. A straightforward schedule could decide to broadcast p often. However, a caching policy might end up caching p permanently. Therefore, a client would never access the broadcast disk for page p, and the best scheduling policy would be not to transmit p at all.

6.1 Background

Let p_i be the probability that page i is requested and

$$\tau_i = \frac{\sqrt{p_i}}{\sum_j \sqrt{p_j}} \ .$$

The *square-root law* suggests that page i should be broadcast with frequency τ_i (and not with probability p_i) [12, 21, 31]. The *Mean Aggregate Delay* (MAD) algorithm is a scheduling algorithm that approximates the square-root law [12, 31]. 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 minimum value of $(s_i + 1)^2 p_i$. In particular, when all p_i's are equal, MAD generates a flat broadcast. We remark that MAD is only an approximation of the square-root law, and does not guarantee optimal schedules in general, but that does guarantee a cyclical schedule. The MAD schedule can be generated very simply at the server site. On the other hand, MAD is very complex at a client site that runs either CF or Gray. Indeed, CF or Gray need to know the next broadcast tick when a page will be transmitted. Such information can be obtained through different implementations, but no known implementation is either space or time efficient. MAD needs an estimate of access probability, and thus it introduces into broadcast systems a component that is not completely on-line.

6.2 Generated and Filtered Traces

In this section, we will discuss how caches change client access patterns. Clients generate sequences of requests to data pages, and we will say that such sequences are *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. In fact, clients will access the broadcast disk only for faulting pages. We will say that the sequence of faulting pages is the *filtered trace*. The distinction between generated and filtered traces is depicted in in figure 5. Filtered traces depend on client access patterns, as well as on the paging strategy and on *CacheSize*. On the contrary, generated patterns depends 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.

We will now investigate the transformation from generated into filtered traces, and its effect on the resulting broadcast schedules. The histograms in figure 6 report the number of LRU faults on the horizontal axis and the number of pages that caused that many faults on the vertical axis. Bars are relative to different values of *CacheSize*.

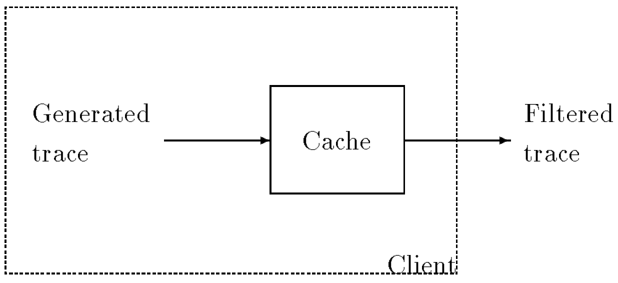
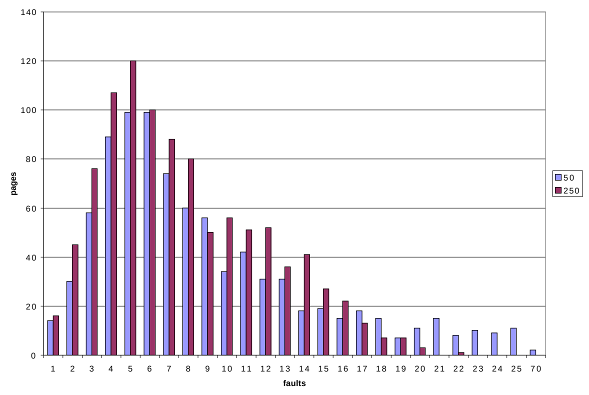
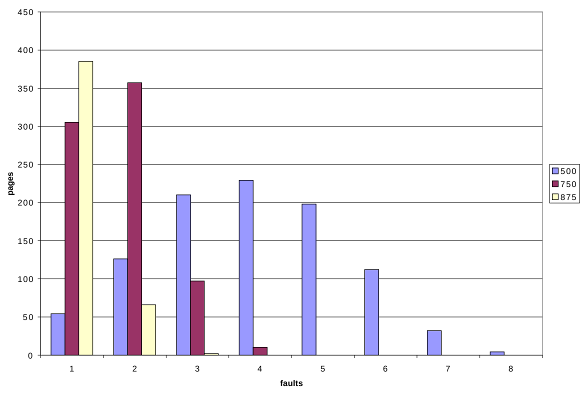


Figure 5: Caches filter traces.



(a)



(b) .

Figure 6: Distribution of pages according to the number of LRU page faults they cause. The horizontal axis reports the number of page faults and the vertical axis the number of pages that caused that many faults. Fault values between 26 and 69 are small and omitted for lack of space. Bars are relative to different values of *CacheSize*. Graph scales differ.

CacheSize	**Gray**
50	6509
75	6058
80	5958
90	5811
100	5621

Table 4: Cost of Gray on one default Zipf workload trace.

The major effect of caching is that it broke very long tails in the distribution of faults. When *CacheSize* = 50, there were pages that cause between 1 and 70 faults. As the *CacheSize* increases, the number of pages that caused a large number of faults decreased and then disappeared. Correspondingly, the number of pages that caused few faults increased. In other words, traces filtered by large caches no longer showed a large variation in the number of faults a page causes. In conclusion, the filtered trace lost the long tails of the generated trace. However, more subtle effects appear when caching is used. In particular, while the generated trace is a sequence pages extracted independently one of the other, such assumption is no longer valid for filtered traces. For example, when LRU is used, if a page i causes a fault at request t, then it will not cause another fault before request $t + CacheSize$.

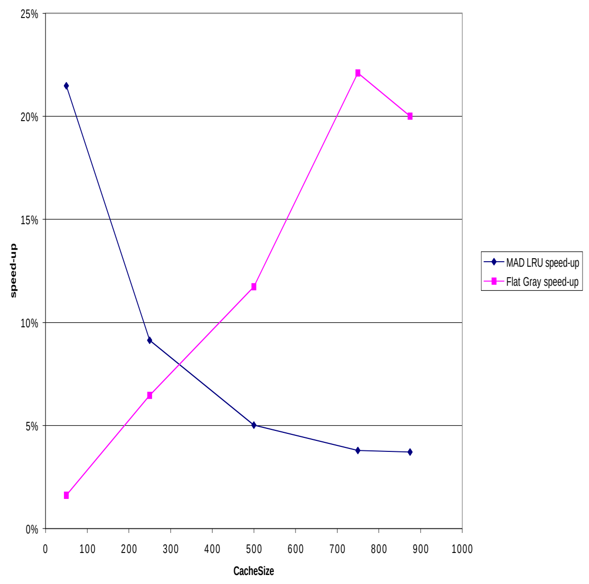
We now turn to discuss the effect of the tail cutting for a square-root broadcast scheduler. If *CacheSize* = 50, then the most frequently broadcast page will be transmitted $\sqrt{70} \simeq 8.4$ times more often than the least frequently broadcast page. If *CacheSize* = 250, the ratio already drops to $\sqrt{22} \simeq 4.7$, and if *CacheSize* = 875, the ratio is $\sqrt{3} \simeq 1.7$. In conclusion, caching cuts long tails in the fault distribution and the square-root law contracts the schedule skewness even more.

6.3 Should Gray go MAD?

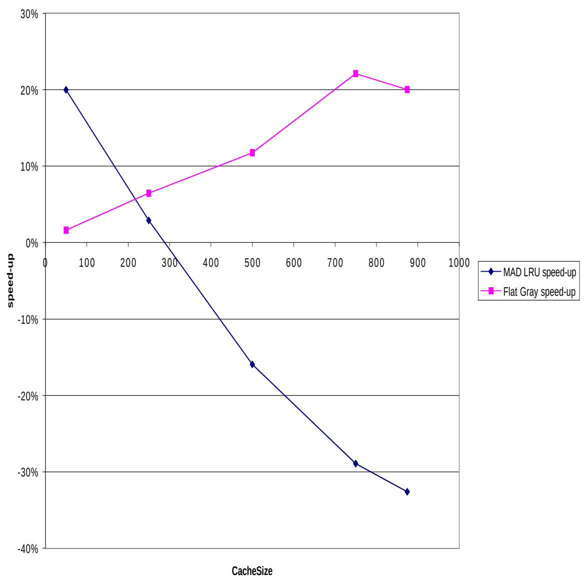
There are two serious difficulties in the way of an integration of Gray with MAD: circularity of the problem and efficiency of the implementation. The behavior of Gray depends on the broadcast schedule. Therefore, the filtered trace depends on the scheduling algorithm. However, scheduling parameters depend on page frequencies in the filtered trace. Therefore, the integration of Gray and MAD gives rise to a circular problem. Moreover, even if the probability estimates had been fixed and MAD was run, it is not clear how to efficiently calculate future transmission times and how to extend Gray's efficient implementation that we presented for a flat schedule.

6.4 Caching and Scheduling in Isolation

Both scheduling and caching improve performance in isolation. In this section, we measure how caching and scheduling compare with each other. On one side, we will have the scheduling algorithm MAD when there is no cache, on the other Gray as *CacheSize* increases and the schedule is flat. The set-up is the same as that described in §4 for the default Zipf workload. The only difference is that we use only one trace instead of thirty because we do not have an efficient implementation of MAD. MAD is subjected to a warm-up process during which it is run for a random number of times between 0 and *ServerDBSize*. The probability estimates p_i used by MAD are obtained as follows. First, we measured the frequency with which i occurs in the trace. Such frequencies are then divided by the trace *Length*. The resulting p_i's are the maximum likelihood estimates of actual probabilities. Several pages will have $p_i = 0$ and MAD will never broadcast them. In order to make a fair comparison between Gray and MAD, we will count a MAD rotation equal to the number of pages with positive p_i's. The algorithm MAD took 5909 rotations to satisfy the resulting trace. By comparison, Gray costs are reported in table 4. Scheduling brought about the same performance improvement of a cache of size between 80 and 90 managed by Gray.



(a) Probability estimates from filtered trace



(b) Probability estimates from generated trace.

Figure 7: Performance of Gray on a flat broadcast against LRU's performance on a MAD broadcast. The cost difference and speed-up are relative to the cost of LRU on a flat broadcast. Graph scales differ.

6.5 Flat Gray vs. MAD LRU

We now turn to examine the performance of Gray on a flat broadcast against that of LRU on a MAD broadcast. The probability estimates p_i used by MAD are obtained with a procedure similar to that in the previous section. Access frequencies are estimated from the trace filtered by the appropriate *CacheSize*, and then divided by the *Length* of the filtered trace. We compared flat Gray and MAD LRU against flat LRU and report speed-ups and Δ's in figure 7(a).

Both flat Gray and MAD LRU improved over flat LRU for all values of *CacheSize*. However, MAD LRU is more effective for smaller *CacheSize*s, whereas flat Gray is more effective for larger *CacheSize*s. An important remark is that, for all *CacheSize*s, the MAD schedule is obtained from filtered trace frequencies, and so it is perfectly tailored for that particular value of *CacheSize*. The two curves intersect for *CacheSize* between 250 and 500.

A summary of our findings is:

- The MAD schedule became closer and closer to a flat one as *CacheSize* increases (as discussed in §6.2).
- Figure 7 shows that also MAD LRU's performance became closer and closer to flat LRU.
- Table 2 and figure 7 show that Gray's speed-up over flat LRU increased with *CacheSize*.

In conclusion, Gray outperformed LRU when caches are big, and MAD had no power to help LRU in those cases because it generated a rather flat broadcast.

6.6 MAD and Noisy

MAD uses the parameters p_i, which are probability estimates that page i will be requested of the broadcast. In general, we cannot expect to have always a precise estimate of access probabilities, and in this section we will examine how imperfect estimates affect algorithm performance. Of course, Gray on a flat schedule does not depend on any probability estimate, but MAD LRU does. As a consequence, Gray performance will be

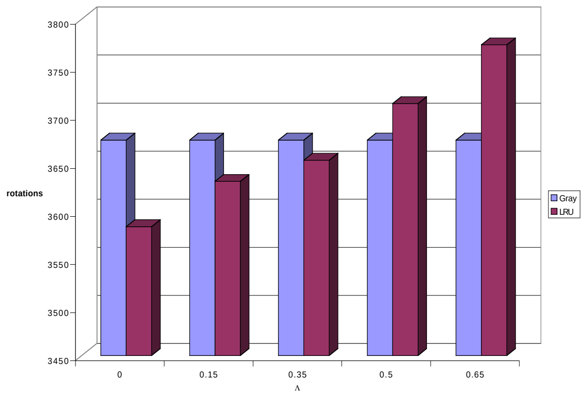


Figure 8: Performance of Gray and MAD LRU under noise for $CacheSize = 250$. The parameter Λ is a bound on the multiplicative noise introduced in the probability estimate.

plotted as a constant values for any type of noise, while MAD LRU's will change. First, we study the effect of noise by adding directly an error term to the probability estimates. We compute probability estimates from the filtered trace exactly like in the previous section, add a random noise, renormalize the resulting values, and then run MAD LRU with the noisy probability estimates. If p_i is the probability estimate obtained from the filtered trace and λ_i is a random value in the interval $(-\Lambda, +\Lambda)$, our noisy probability estimates are

$$p'_i = \frac{(1+\lambda_i)p_i}{\sum_j (1+\lambda_j)p_j} \; .$$

When Λ increase, the noise in the estimates increases as well. Our findings for different values of Λ and for $CacheSize = 250$ are reported in figure 8. MAD LRU outperformed flat Gray when there is little noise, but it was worse for $\Lambda > .35$.

We turn to measure a type of error that arises as a direct consequence of caching and is due to a wrong estimate of the $CacheSize$. In §6.5, the p_i values were obtained from the frequencies in the filtered trace. However, the filtered trace depends on the value of $CacheSize$, and a wrong estimate of $CacheSize$ leads to a wrong estimate of the p_i's. The error is analyzed in figure 7. In figure 7(b), the MAD schedule is obtained under the assumption that there is no cache, while in fact the $CacheSize$ takes increasing values. In figure 7(a), the MAD schedule is obtained from the right estimate of $CacheSize$. While the difference between the two MAD LRUs is small for $CacheSize = 50$, MAD LRU (with wrong $CacheSize$) is much worse than MAD LRU (with right $CacheSize$) for larger values of $CacheSize$. When $CacheSize \geq 500$, MAD LRU (wrong $CacheSize$) is even worse than flat LRU. The intersection point between the flat Gray and the MAD LRU curves has now decreased to $CacheSize < 250$.

6.7 Web Workload

Figure 9 gives flat Gray's and MAD LRU's speed-up over flat LRU on the epa-http workload for $bcThresh = 8$. Again, MAD LRU is better for smaller caches and flat Gray is better for larger caches. The balance point was for $CacheSize$ slightly less than 250, which is about 1/3 of the $ServerDBSize$ for this trace (compare with table 3).

6.8 Discussion

In general, MAD helped LRU if the $CacheSize$ is small, while Gray on a flat schedule was better for larger caches. For large caches, Gray outperformed MAD LRU even when the broadcast schedule was perfectly tailored to LRU's access pattern. In synthetic workloads, the intersection point depended on the error in the probability estimates, but it was roughly for $CacheSize = 250$, a value that is not particularly big compared with the size of the $AccessRange = 1000$.

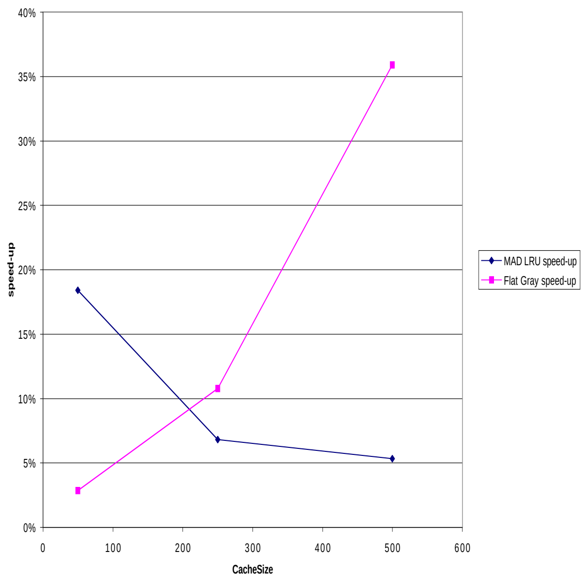


Figure 9: Speed-up of flat Gray over MAD LRU on the epa-http workload for $bcThresh = 8$.

On the positive side, MAD led to significant performance improvements for small caches. On the negative side, MAD is a qualitatively different algorithm from flat Gray because MAD requires knowledge of access probability. Thus, MAD is not completely on-line. Moreover, complicate scheduling is hard to justify when client access patterns do not match estimated ones or when actual cache sizes are bigger than anticipated. Non-flat scheduling makes cost-dependent paging strategies (e.g. Gray) hard to implement efficiently. In mobile computing, non-flat schedules require that a complicate indexing structure be maintained in order to keep track of the schedule itself [26]. As a consequence, complicate algorithms are needed at the server and client sites to build and look-up the index and part of the bandwidth is now employed to broadcast the index rather than data pages. In conclusion, the adoption of a non-flat schedule could be beneficial, but it depends on the accuracy of delicate assumptions on access patterns and cache sizes, and it complicates paging and indexing algorithms.

7 Related Work

Broadcast disks have received substantial attention in the literature. Survey papers summarize previous contributions to the broadcast disk literature, areas of research, and the position of broadcast disks among other push and push/pull data dissemination architectures [20]. Information dissemination on the Internet has been considered by various authors [14, 17, 37] and systems [1, 2]. Cyclic multicast over the Internet is discussed in [8].

Ammar gives a prefetching strategy that loads pages on the basis of links embedded in previously loaded pages [9]. Other approaches execute prefetching without using hints. Acharya *et al.* considered prefetching on broadcast disks, and propose the PT algorithm, which we described in §3 [5]. Subsequently, Tassiulas *et al.* considered optimal algorithms for prefetching when more than one page can be requested in one broadcast tick [34]. Both works assume that traces of page requests are generated by a stationary probability distribution and that paging strategies know it. The Gray algorithm was the first completely on-line strategy tailored for broadcast disk paging [25]. While the Gray algorithm had been analyzed in the worst-case, no simulation result was known before the present paper. No previous paper has given efficient implementation of broadcast disk paging algorithms.

Prefetching in broadcast disks is somewhat related to other techniques used in mobile environments [13, 27, 32]. The difference is that broadcast disks prefetching aims at improving performance, whereas other works focus on increasing availability or avoiding accesses to stale data.

Broadcast disk paging poses a trade-off between the number of page faults and the cost per fault. Similar trade-offs exist in a variety of context, as for example, Web caching [7, 35] and hierarchical paging [15].

The problem of finding an optimal cyclic schedule is NP-hard [12], but it can be solved in polynomial time if $ServerDBSize = 2$ [11]. The square-root law has been proposed by several authors [10, 9, 21, 31]. The golden ratio algorithm instantiate the rule and gives a 1.125-approximation for all *ServerDBSize*s [12]. A simpler approximation of the rule is the MAD algorithm [31, 12]. Scheduling with non-uniform transmission times has been investigated as well [24, 36]. In a mobile environment, the objective of scheduling is to minimize a combination of response time and tuning time. Khanna *et al.* present an algorithm that inserts index pages along the server broadcast and perform scheduling in order to reduce both response and tuning time [26]. Several authors have studied the problem of broadcast scheduling when pull is also supported [6, 3, 30]. The crux of scheduling is the estimation of page popularities. Stathatos *et al.* use a pull backchannel for data communication and, indirectly, to estimate page popularity and its dynamic over time [30]. Our work assumes that no backchannel is integrated with broadcast disks, and so scheduling is necessarily an off-line procedure. To the best of our knowledge, the relationship between server scheduling and client caching had not been studied prior to the present paper.

8 Summary and Conclusion

In this paper, we have studied client caching in broadcast disks and its relation with the broadcast schedule. We considered on-line algorithms, that is, algorithms that do not know future data requests and that do not have a probabilistic estimate of access patterns. On-line algorithms are critical in many circumstances because probability estimates are often unavailable, difficult to validate, or inaccurate. First, we considered a flat broadcast schedule, that is, one where all data items are broadcast with the same frequency, and then we turned to examine the relation between caching and broadcast schedules. For a flat broadcast:

- We gave new efficient implementations of various broadcast disk paging algorithms. Although we used well-known and practical algorithms, we reduced PT's running time per fault from quadratic to linear.
- We conducted the first experimental analysis of the Gray algorithm. Gray outperformed our other on-line algorithms (LRU and CF) on average and with high probability across a large number of synthetic workloads. Gray outperformed LRU also on Web trace simulations.
- In particular, Gray improved performance even if client access patterns shifted over time.

In conclusion, the Gray algorithm does not use any probabilistic assumption and substantially outperforms traditional on-line algorithms in a variety of settings.

We then turned to investigate several trade-offs between caching and scheduling algorithms. We consider MAD schedules, a simple, but provably good scheme to broadcast data on the basis of client access patterns [12, 31]. MAD LRU was effective when caches were small, but not for larger caches. Meanwhile, Gray on a flat schedules became progressively more effective. The balance point was for caches of size 250, which is between 1/4 and 1/3 of the working set in our experiments. However, MAD LRU is inherently different from flat Gray, because MAD LRU requires knowledge of access probabilities. Consequently, MAD LRU is affected by errors in probability estimates, while flat Gray is not. Moreover, MAD and all other non-flat schedules complicate the implementation of caching strategies and indexing structures. Flat Gray is more effective for larger caches, and it is a better choice in a truly on-line setting.

Acknowledgements

We would like to thank Swarup Acharya, Demet Aksoy, Allan Borodin, Mike Franklin, Bob Gruber, Sanjeev Khanna, and Samir Khuller for helpful conversations. The epa-http trace was collected by Laura Bottomley of Duke University. The author was partly supported by the CAREER Award CCR-9501355.

References

[1] `http://www.direcpc.com`.

[2] `http://www.hybrid.com`.

[3] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proc. of Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 1998.

[4] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the 1995 ACM SIGMOD Conference International Conference on Management of Data*, pages 199–210, 1995.

[5] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Prefetching from a broadcast disk. In *Proceedings of the International Conference on Data Engineering*, 1996.

[6] Demet Aksoy and Michael Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. In *Proc. IEEE INFOCOM*, 1998.

[7] Suzanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *Proceedings of the Tenth ACM-SIAM Symposium on Discrete Algorithms*, 1999. (To appear).

[8] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of Web pages using cyclic best-effort (UDP) multicast. In *Proceedings of INFOCOM 98*, 1998.

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

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

[11] S. Anily, C. A. Glass, and R. Hassin. The scheduling of maintenance service. *Discrete Applied Mathematics*, 82(1–3):27–42, 1998.

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

[13] D. Barbara and T. Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. In *Proceedings of the 1994 ACM SIGMOD Conference International Conference on Management of Data*, pages 1–12, 1994.

[14] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3), September 1996.

[15] Marek Chrobak and John Noga. Competitive algorithms for multilevel caching and relaxed list update. In *Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms*, pages 87–96, 1998.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[17] S. Dao and B. Perry. Information dissemination in hybrid satellite/terrestrial networks. *Data Engineering*, 19(13), September 1996.

[18] Juergen Eichenauer-Herrman. Inversive congruential pseudorandom numbers: a tutorial. *International Statistical Review*, 60(2):167–176, August 1992.

[19] Juergen Eichenauer-Herrman. Pseudorandom number generation by nonlinear methods. *International Statistical Review*, 63(2):247–255, August 1995.

[20] Michael Franklin and Stanley Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of the International Conference Object Oriented Programming Languages Systems*, 1997.

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

[22] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD Conference International Conference on Management of Data*, pages 243–252, 1994.

[23] Sandy Irani and Anna R. Karlin. Online computation. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, chapter 13, pages 521–564. PWS Publishing Company, Boston, 1997.

[24] Claire Kenyon and Nicolas Schabanel. The data broadcast problem with non-uniform transmission times. In *Proceedings of the Tenth ACM-SIAM Symposium on Discrete Algorithms*, 1999. (To appear).

[25] Sanjeev Khanna and Vincenzo Liberatore. On broadcast disk paging. In *Proceedings of the 30th ACM Symposium on the Theory of Computing*, pages 634–643, 1998.

[26] Sanjeev Khanna and Shiyu Zhou. On indexed data broadcast. In *Proceedings of the Thirtieth ACM Symposium on the Theory of Computing*, 1998.

[27] J. Kistler and M. Satyanayaranan. Disconnected operation in Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[28] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[29] Hannes Leeb and Stefan Wegenkittl. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transaction on Modeling and Computer Simulation*, 7(2):272–286, April 1997.

[30] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. In *Proc. 23rd International Conference on Very Large DataBases*, 1997.

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

[32] C. Tait, H. Lei, S. Acharya, and H. Chang. Intelligent file hoarding for mobile computers. In *Proc. ACM Conf. on Mobile Computing and Networking*, 1995.

[33] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

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

[35] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, Department of Computer Sciences, University of Texas at Austin, February 1998.

[36] N. Vaidya and S. Hameed. 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.

[37] T. Yan and H. Garcia-Molina. Efficient dissemination of information on the Internet. *Data Engineering*, 19(13), September 1996.

A Simulation Details

We generated thirty sequences for each workload. The same thirty traces were used for all algorithms and all cache sizes. We collected average values across the thirty traces, as well as standard deviations. The sequences were generated with an inversive congruential random number generator. The properties of inversive congruential generators have been studied in theory [18, 19] and experimentally [29]. The implementation we used is contained in the pLab package, version 2.2 (`http://random.mat.sbg.ac.at/`).

	LRU			**Gray**		**speed-up**		**Δ**	
CacheSize	mean	σ	**CF%**	mean	σ	mean	σ	mean	σ
50	7395	40	-0.03%	7383	40	0.18%	0.06%	12	4.8
250	6996	45	-0.037%	6938	42	0.84%	0.2%	58	11
500	6508	43	-0.053%	6394	41	1.8%	0.2%	114	10
750	6028	44	-0.039%	5862	41	2.8%	0.3%	165	18
875	5786	36	-0.043%	5601	42	3.3%	0.4%	185	21

Table 5: Performance of LRU, CF, and Gray in the *Random* workload. The LRU and Gray column report the average cost and sample standard deviation of LRU and Gray over a sequence of thirty trials. Costs are given in number of rotation (1 rotation = *ServerDBSize* broadcast ticks). The CF column gives the percentage cost increase of CF over LRU; negative percentages correspond to cost reduction. The speed-up column gives the average and sample standard deviation of the speed-up of Gray over LRU. Positive percentage correspond to an improvement over LRU. The Δ column gives the average and standard deviation of the cost difference of Gray over LRU.

We used a period $p = 2147483647$, and parameters $a = 14288$, $b = 758634$. We validated some of our earlier experiments with `random(3B)`, the non-linear additive feedback random number generator in the SunOS/BSD compatibility library with a state of 256 bytes. We could not observe significant differences between `random` and the inversive congruential generator. We extracted Zipf random values with the algorithm by Gray *et al.* [22], which does not exactly extract Zipf values, but an approximation of the Zipf distribution. We wrote our simulator in C and performed our experiments on a SUN SPARCstation LX with the `gcc -O3` compiler. We measured the waiting time incurred by the algorithms after the following *warm-up process*. Each algorithm's cache is filled with the first *CacheSize* distinct pages requested in each sequence. In addition, LRU will have those warm-up pages in LRU order. In the Web workloads, we assumed that the broadcast pages are large enough to contain every file in the trace. We remark that this is only a simulation choice, and that we could have taken smaller page sizes. Another simulation choice was to scatter the pages in random order along the cyclical broadcast.

B *Random* Workload

We first report the results on the *Random* workload in table 5.

The algorithms had all roughly the same cost. However, Gray always outperformed LRU. The speed-up of Gray over LRU increases with *CacheSize* and it reaches 3% for *CacheSize* = 875. We now turn to the Zipf workload, which will be the main focus of the rest of the paper.

C Robustness

In this section, we report some results that have been omitted from the regular paper. The results are relative to the default Zipf workload. We examine the number of page faults (as opposed to waiting times) and algorithm sensitivity to skewness and changes in access patterns, we compare PT with the LRU, CF, and Gray, and we analyze the sensitivity of the on-line algorithms for changes in the region placement.

C.1 Page Faults

We report statistics on the page fault rate in table 6. The number of page faults is not the best performance measure for broadcast disks — waiting time is. However, we give page fault statistics because it is interesting to compare page fault rates with waiting times. We give the number of page faults incurred by LRU and the percentage increase in number of page faults incurred by CF and Gray. Negative percentages would show an improvement of CF or Gray over LRU.

CacheSize	**LRU**	**CF**	**Gray**
50	13284	0.42%	0.058%
250	7854	8.3%	1.8%
500	3821	20%	4.4%
750	1384	32%	6.1%
875	539	38%	7.8%

Table 6: Number of page faults incurred by LRU, CF, and Gray. The LRU column reports the number of page faults incurred by LRU. The CF and Gray columns report the percentage increase in the number of page faults incurred by CF and Gray.

CacheSize	**PT**	**LRU**	**CF**	**Gray**
50	5188	28.07%	28.24%	25.73%
250	2346	67.42%	80.66%	56.57%
500	951	100.7%	139.6%	79.13%
750	216	219.7%	316.3%	161.6%
875	47	470.4%	674.2%	319.4%

Table 7: Comparison between the on-line algorithm and PT. PT's cost is in number of rotations. The on-line algorithm columns report the gap between the algorithm and PT.

We notice the following facts. The number of page faults of LRU decreased with *CacheSize*. CF's was within .4% to 38% of LRU, and the gap increased with *CacheSize*. CF's percentage increase in number of page faults was bigger than CF's increase in waiting times (compare with table 2). Therefore, CF spent less waiting time than LRU per fault. Unfortunately, CF incurred such a larger number of faults that its total waiting time is much worse than LRU's. Gray was always worse than LRU in terms of number of page faults. The percentage difference was as high as 7.8%. The relative difference increased with *CacheSize*. However, Gray always outperformed LRU in terms of waiting times, which is to say, Gray waited less than LRU on a page fault. In fact, the largest speed-up of Gray over LRU (38%) occured when it CacheSize = 875, which is also when the number of page faults was favoring the most LRU over Gray (7.8%). In conclusion, the number of page faults was not a predictor of waiting times. For example, Gray became better and better than LRU while at the same time it occurred a larger and larger relative number of page faults. Gray was much better than CF in terms of page faults.

C.2 On-line Algorithms and PT

We compared the on-line algorithms with PT. Our findings are in table 7. PT significantly outperformed the three on-line algorithms in our experiments. When the cache size is 50, PT outperformed the on-line algorithms by 26% to 28%. As the cache grew larger, all percentage differences became larger. Eventually, when the cache size is 875, Gray was 3.2 times worse than PT and LRU was 4.7 times worse than PT. The gap between PT and the on-line algorithms was always much bigger than the gap between any two on-line algorithms. There was a very significant gap between our on-line algorithms and PT. However, it is not clear whether improved on-line algorithms could reduce such gap or if a substantial performance difference is intrinsically due to PT's use of off-line information.

C.3 Sensitivity to Skewness

We now examine the algorithm behavior when the Zipf skew parameter θ was varied. Our results are in table 8. Notice that the subtable corresponding to $\theta = 0.95$ is identical to table 2. For any fixed *CacheSize*,

		LRU			**Gray**		**speed-up**		Δ	
$CacheSize$	θ	mean	σ	**CF%**	mean	σ	mean	σ	mean	σ
50	0	7097	39	-0.068%	7037	42	0.86%	0.2%	60	12
	.25	7080	33	-0.013%	7019	34	0.87%	0.1%	61	10
	.5	7010	36	-0.12%	6937	36	1%	0.1%	72	10
	.75	6855	34	-0.042%	6765	32	1.3%	0.2%	90	16
	.95	6644	32	0.13%	6523	35	1.9%	0.3%	121	18
250	0	5519	41	-0.14%	5252	41	5.1%	0.3%	266	16
	.25	5435	42	0.079%	5164	35	5.2%	0.5%	270	25
	.5	5143	46	1.1%	4861	40	5.8%	0.4%	281	21
	.75	4577	46	3.7%	4296	39	6.5%	0.5%	280	22
	.95	3927	46	7.9%	3673	44	6.9%	0.6%	254	21
500	0	3574	42	-0.0028%	3162	26	13%	1%	411	36
	.25	3458	44	0.8%	3054	30	13%	1%	404	34
	.5	3058	42	4.9%	2706	37	13%	0.7%	351	18
	.75	2458	41	12%	2182	37	13%	1%	275	25
	.95	1909	31	19%	1704	29	12%	1%	204	19
750	0	1704	23	-0.71%	1318	19	29%	2%	386	19
	.25	1594	28	2.3%	1236	21	29%	2%	358	20
	.5	1308	25	10%	1035	19	26%	2%	273	21
	.75	965	27	21%	775	18	25%	3%	190	19
	.95	692	20	30%	566	18	22%	2%	125	10
875	0	807	21	-1.2%	549	14	47%	3%	257	16
	.25	730	22	5.7%	500	15	46%	4%	230	20
	.5	574	17	14%	405	16	42%	5%	169	16
	.75	400	18	27%	287	13	40%	5%	113	13
	.95	269	17	36%	198	10	36%	6%	71	13

Table 8: Sensitivity to changes in access pattern skewness.

the absolute cost of LRU and Gray decreased as θ increased. Both algorithms were able to exploit the increased locality of a more skewed distribution. The result is consistent with our previous comparison of the *Random* workload with the Zipf workload. In fact, if we order the workloads according to their skewness from *Random* (no skew) to $\theta = 0, \ldots, .95$ (more skewed), we observe that the algorithm cost decreased as the distribution became more and more skewed. The Gray algorithm was always better than LRU. Once again, standard deviations were smaller compared to mean values, which implies that Gray was superior to LRU with high probability.

We now turn to summarize the trends of the speed-up and Δ as the parameters *CacheSize* and θ change. For a fixed value of *CacheSize*, the speed-up increased with θ when *CacheSize* $= 50, 250$ and decreased when *CacheSize* was larger. The largest value of the speed-up was found for the largest *CacheSize* $= 875$ and for the smallest $\theta = 0$. For any fixed value of θ, the speed-up increased with *CacheSize*. An increased θ worked against Gray's speed-up when the cache is large. In other words, LRU took better advantage of a skewed access pattern when there was a larger cache.

For a fixed value of *CacheSize*, the difference Δ increases with θ when *CacheSize* $= 50$, it reached a peak and decreased when *CacheSize* $= 250$, and decreased when *CacheSize* was larger. The largest difference occured when *CacheSize* $= 500$ and $\theta = 0$. For a fixed value of θ, Δ always reached a peak and then deceased as *CacheSize* increased.

We turn now to examine CF's performance. First of all, we notice that when CF outperforms LRU, the improvement is not statistically significant. For example, when *CacheSize* $= 875$ and $\theta = 0$, then CF has a 1.2% improvement over LRU. In other words, CF cost was $(100\% - 1.2\%)\times$ (the waiting time of LRU), which is $.988 \cdot 807 \simeq 797$ rotations. The cost difference between LRU and CF is 10 rotations, which is less than LRU's standard deviation. Analogous computations hold whenever CF outperformed LRU. In conclusion, there was no experiment where CF outperformed LRU in a statistically significant way. For a fixed *CacheSize*, CF became worse than LRU as θ increased. Consider now a fixed θ and vary the *CacheSize*. There was no statistically significant change with *CacheSize* when $\theta = 0$, but CF was worse and worse than LRU as the *CacheSize* increases when $\theta > 0$.

In conclusion, although speed-ups and differences varied with *CacheSize* and θ, Gray was always superior to LRU both on average and with high probability.

C.4 Sensitivity to Changes in Access Patterns

In table 9 we report our findings for the case when the client access pattern changes with time. For a given value of *CacheSize*, the rows are ordered from *SwitchTime* $= 1000$, *Offset* $= 45$ (frequent and radical interest shifts) to *SwitchTime* $= 8000$, *Offset* $= 25$ (infrequent and gradual shift). For a fixed *CacheSize*, the waiting time of LRU and Gray decreased as the shift become more gradual and infrequent. Again, Gray outperformed LRU and small standard deviations clearly separate the two algorithms. However, Gray's speed-up was not as large as when there was no access pattern shift. When *CacheSize* $= 50$, there is no significant difference between LRU and CF. However, significant differences were found for larger values of *CacheSize*, and CF was up to 72% worse than LRU.

C.5 Sensitivity to Region Placement

We now discuss how region placement affects algorithm performance. Our results are in table 10. We will also compare table 10 with the default Zipf workload of table 2. LRU and Gray were largely independent of the region placement. In fact, for any given algorithm and for fixed cache sizes, cost differences were small compared with standard deviations. CF improved on LRU only for *CacheSize* $= 875$ in the uniform region workload, but such improvement is not statistically significant. However, CF did better in the uniform and reverse workload than in the default Zipf workload (scattered regions) — compare table 2 and 10. In conclusion, LRU and Gray were not influenced by region placement, while CF actually improved if the regions are not scattered. By contrast, Acharya *et al.* showed two off-line algorithms that benefited from scattering [5]. We do not know if there is any "sensible" on-line algorithm that would benefit from scattering.

CacheSize	*SwitchTime*	*Offset*	**LRU**		**CF%**	**Gray**		**speed-up**		**Δ**	
			mean	σ		mean	σ	mean	σ	mean	σ
50	1000	45	6660	36	0.34%	6547	36	1.7%	0.2%	112	12
		25	6650	42	0.27%	6535	42	1.8%	0.2%	115	13
	8000	45	6642	39	0.22%	6522	41	1.8%	0.2%	120	14
		25	6643	38	0.24%	6525	36	1.8%	0.3%	118	17
250	1000	45	4307	43	12%	4147	38	3.9%	0.5%	160	22
		25	4163	44	11%	3980	40	4.6%	0.6%	183	23
	8000	45	3955	45	8.5%	3707	40	6.7%	0.6%	247	20
		25	3946	54	8.2%	3692	46	6.9%	0.7%	253	24
500	1000	45	3202	30	20%	3163	40	1.2%	0.6%	39	18
		25	2747	36	22%	2641	43	4%	0.9%	106	22
	8000	45	1998	30	24%	1807	32	11%	1%	191	21
		25	1960	32	23%	1763	32	11%	1%	197	19
750	1000	45	2846	33	15%	2782	34	2.3%	0.4%	63	12
		25	2135	36	23%	2055	37	3.9%	0.5%	79	10
	8000	45	890	22	51%	808	17	10%	1%	82	11
		25	829	23	46%	735	20	13%	2%	93	12
875	1000	45	2680	34	13%	2604	30	2.9%	0.4%	76	11
		25	1913	34	23%	1826	30	4.8%	0.7%	87	13
	8000	45	551	17	72%	495	24	11%	4%	55	19
		25	472	15	66%	400	20	18%	5%	71	18

Table 9: Sensitivity to changes in access pattern.

Workload	*CacheSize*	**LRU**		**CF%**	**Gray**		**speed-up**		**Δ**	
		mean	σ		mean	σ	mean	σ	mean	σ
Uniform	50	6646	35	0.082%	6525	34	1.8%	0.3%	120	18
	250	3937	40	4.4%	3653	41	7.8%	0.7%	284	23
	500	1909	34	2.9%	1687	29	13%	1%	221	17
	750	697	21	0.43%	563	20	24%	3%	133	14
	875	270	14	-0.96%	195	10	39%	5%	75	8.5
Reverse	50	6645	28	0.18%	6520	31	1.9%	0.2%	125	14
	250	3937	53	5.3%	3661	45	7.5%	0.6%	275	24
	500	1916	33	5.6%	1690	28	13%	1%	226	23
	750	694	19	7%	563	15	23%	2%	131	12
	875	272	12	4.4%	197	9	38%	6%	75	11

Table 10: Sensitivity to changes in region placement.