Sequential prefetch cache sizing for maximal hit rate

Swapnil Bhatia*
Computing Science Laboratory
Palo Alto Research Center
Palo Alto, CA 94304
sbhatia@parc.com

Elizabeth Varki
*Department of Computer Science
University of New Hampshire
Durham, NH 03824
{sbhatia, varki}@cs.unh.edu

Arif Merchant
Storage Platforms Laboratory
Hewlett-Packard Laboratories
Palo Alto, CA 94304
arif_merchant@hp.com

***Abstract*—We propose a prefetch cache sizing module for use with any sequential prefetching scheme and evaluate its impact on the hit rate. Disk array caches perform sequential prefetching by loading data contiguous to I/O request data into the array cache. If the I/O workload has sequential locality, then data prefetched in response to sequential accesses in the workload will receive hits. Different schemes prefetch different data, so the prefetch cache size requirement varies. Moreover, the proportion of sequential and random requests in the workload and their interleaving pattern affects the size requirement. If the cache is too small, then prefetched data would get evicted from the cache before a request for the data arrives, thus lowering the hit rate. If the cache is too large, then valuable cache space is wasted. We present a simple sizing module that can be added to any prefetching scheme to ensure that the prefetch cache size is adequately matched to the requirement of the prefetching scheme on a dynamic workload comprising multiple streams. We analytically compute the maximal hit rate achievable by popular prefetching schemes and through simulations, show that our sizing module maintains the prefetch cache at a size that nearly achieves this maximal hit rate.**

I. Introduction

Disk arrays consist of several disks, an array controller, and one or more array caches. In addition to increasing the capacity and availability of storage systems, disk arrays improve the speed of storage access by distributing the load across disks. However, it still takes milli seconds to access data from the disks, so the speed differential between host computers and storage is substantial. The speed of disk array data access can be improved significantly if data are already loaded in the array caches when I/O requests for the data arrive. Unlike disk caches which are small, array caches are large and are often larger than file system caches. Unlike disk controllers that are only capable of elementary operations, disk array controllers are capable of performing complex tasks to speed data access. The size of the array cache along with the power of the array controller gives the disk array the hardware capability of executing sophisticated caching and prefetching schemes capable of speeding access to the data on the storage device.

Prefetching techniques speed up storage access by loading data from the disks into the cache before I/O requests for the data arrive. To determine the data to be prefetched into the cache, a prefetching scheme must predict future data needs of applications accessing the storage system. Several applications may be accessing the storage system concurrently. Each application may simultaneously have several files open for reading, or may open a file more than once. Each open command generates a stream of I/O requests to a single file over a period of time. The operating system submits the outstanding requests from various streams to the storage system. An I/O request submitted to the storage system consists of the request's address (block number), whether the request is of type read or write, and the number of blocks to be accessed. The storage system is not provided any information about the applications, streams or files; it does not know which stream relates to an I/O request. A storage controller does not know when a new stream starts and when an existing stream closes. Therefore, a storage prefetching technique has to infer the future data needs of applications using only I/O block numbers.

This "narrow" I/O interface limits the prediction capabilities of storage prefetching techniques [19]. Luckily, file data are often read sequentially, and operating systems try to store a file's data contiguously on storage disks [10], [22]. Therefore, there is a non-zero probability that some of the streams accessing storage are sequential or partly sequential. Consequently, sequential prefetching is the most common prefetching technique implemented in storage systems [11]–[13], [18], [27]. A sequential prefetching technique generates prefetch requests for data that are stored contiguously to I/O request data. A central goal of a prefetching scheme is to keep prefetched data in the cache until I/O requests for the data arrive.

In addition to prefetched data, an array cache is used for temporarily storing I/O write request data and I/O read request data. While an array cache is large compared to a disk cache, space is still scarce since the size of an array cache is just 0.1% to 1% of the storage capacity [10], [26]. One of the issues that need to be addressed is the space requirement of a prefetch cache. Prefetch data are brought in on the assumption that requests for them will arrive eventually. If the prefetch cache is too small, prefetched data may get ejected from the cache before I/O requests for the data arrive. Ejecting useful prefetched data results in a double whammy performance drop, since the data have to be reloaded from the disks when I/O requests corresponding to the ejected prefetched data arrive.

The performance of a storage system would improve if the cache size is sufficiently large so that useful prefetched data remain in the cache long enough to result in cache hits. On the other hand, the prediction capabilities of a storage prefetching technique are limited, so it is possible that some of the prefetched data are wrongly prefetched and never receive hits. This is especially true when the I/O workload contains several random streams and a "blind" prefetching technique like Prefetch-Always (that prefetches data contiguous to every I/O request) is used. If the prefetch cache is too large, then a lot of space is wasted storing wrongly prefetched data. Thus, the size of a prefetch cache plays a key role in the performance of a storage system.

Prefetching has been around since the dawn of computer systems, so there are numerous papers in this area [2]. A majority of the papers, however, focus on the specifics of prefetching, namely, what to prefetch, when to prefetch, and how much to prefetch. This paper does not develop a new prefetching technique—we develop a prefetch cache sizing module that can be incorporated into any storage prefetching technique. The sizing module tracks the degree of sequential locality in the dynamic workload and adapts the prefetch cache to a size appropriate for the workload. The contributions of this paper are the sizing module and the evaluation of the impact of size on the performance of a sequential prefetching technique. A key result proved here shows that under the simplest workload assumptions, as the prefetch cache size decreases below the minimum necessary for a prefetching scheme, the loss in hit rate increases exponentially. We also demonstrate the performance of the sizing module via simulations using different workload models. We believe that this is the first paper to systematically address storage prefetch cache sizing and analyze the impact of sizing on prefetch hit rate for various prefetching schemes.

The rest of the paper is organized as follows: Section II describes the I/O workload. Section III computes the maximum prefetch hit rates of various prefetching schemes. Section V presents our dynamic cache sizing technique, and Section VI validates the efficiency of our technique. Related work is outlined in Section VIII.

II. Streams

A prefetching technique prefetches data that it expects to be requested in the near future. A cache contains data loaded in response to on-demand requests and prefetch requests. We logically partition the read cache into the on-demand cache and the prefetch cache, each part storing data loaded as a result of the respective type of request. When an on-demand request arrives for prefetched data, we assume that the hit prefetched data are served and removed from the prefetch cache. The hit data could either be moved into the on-demand cache or removed altogether from the read cache.

Sequential prefetching techniques generate prefetch requests for data stored contiguously to on-demand request data. Sequential prefetching is common in file and storage systems since applications typically read files sequentially and many file systems try to store a file's data contiguously [10], [22]. A *stream* is a sequence of I/O requests for a particular file's data and is denoted as $\langle a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n \rangle$. A new stream starts when a file is opened for reading and the stream stops when the file is closed.

Definition 1: Consider a stream $\langle a_1, \ldots, a_n \rangle$. Request a_{i+1} is said to be *sequential* iff its data are stored contiguously after the data for a_i on the storage device. Otherwise, a_{i+1} is said to be *random*.

The first request of a stream is assumed to be random. A sequential request in a stream is denoted by s, and a random request, by r. Thus, a completely random stream will be denoted by a string in $\langle r(,r)^\star \rangle$, a completely sequential stream, by one in $\langle r(,s)^\star \rangle$, and a partly sequential stream by one in $\langle r(,r)^\star \cup (,s)^\star \rangle$. A stream is called *sequential* if it is completely or partly sequential. Every contiguous subsequence of sequential requests in a stream is called a *sequential run*. A *run* comprises all the requests between two consecutive sequential runs, including those in the first one. Random requests in a run form a *random run*. As per the above definitions, the minimum length of a sequential run and a random run is one.

III. Sequential Prefetching Schemes

Sequential prefetching techniques decide which contiguous blocks to prefetch based on past hits and misses. On this basis, sequential techniques are broadly classified into the following: Prefetch Always (PA), Prefetch On a Miss (PoM) and Prefetch On a Hit (PoH). We discuss each of these below.

In the PA technique, each request, regardless of whether it hits or misses in the read cache, results in the prefetch of data contiguous to its data. The advantage of PA is that since every sequential request from a stream is prefetched, the hit rate obtained is high and is the highest among the three techniques under ideal conditions. For random and partly sequential streams, however, PA wastefully prefetches large amounts of data that never receive prefetch hits.

In the PoM technique, each request that misses in the read cache results in the prefetch of data contiguous to its data. The advantage of PoM is that the prefetch request can be *piggybacked* onto the on-demand request. This can result in significant savings in response time when the system is heavily utilized. However, PoM does not prefetch on hits and hence loses about half the hits in a sequential run, while wastefully prefetching all random requests which will never receive hits.

In the PoH technique, each request that hits in the prefetch cache results in the prefetch of data contiguous to its data. Every request that misses in the prefetch cache activates the *sequential access detection module* which determines if the missed request is a sequential request. The module searches past request addresses for requests whose data are stored contiguously preceding the data of the missed request. If such an address is found, then PoH assumes that the missed request is the start of a sequential run. Therefore, PoH prefetches data contiguous to the missed request's data by generating a piggybacked request for the on-demand data and prefetch

data. PoH is the only technique that initiates a prefetch after identifying that a request is part of a sequential run. The prefetched data are stored in the prefetch cache while the on-demand data are written into the on-demand cache.

The *prefetch hit rate* of a stream is the proportion of the requests in the stream that hit in the prefetch cache. The *maximum prefetch hit rate* of a stream is the proportion of sequential requests in the stream. Since prefetching techniques differ on their prefetching decision, not all of them can achieve this maximum hit rate. The PA technique can achieve this maximum hit rate as it prefetches data contiguous to every on-demand request. The PoH technique detects a sequential run only after the arrival of the first on-demand request in the sequential run. Therefore, it cannot prefetch the first request in each sequential run, but can prefetch the remaining sequential requests in the run. The PoM technique can obtain hits for at most half the requests in a sequential run.

IV. Cache Size

We define the size of a cache as the number of lines in the cache. For expositional clarity, we assume that each cache line can hold data from a single prefetch request. To maximize prefetch hit rate, the cache size should be large enough to ensure that no prefetched sequential request is evicted from the cache before the arrival of an on-demand request for it. If the workload submitted to the storage system consists of a single stream, then the prefetch cache only needs a single cache line, regardless of the scheme. The following example illustrates this point.

Example 1: Consider the partly sequential stream $\langle 1, 2, 3, 45, 67, 83, 11, 12, 13, 14, 32, 76, 98\rangle$ submitted to the storage device, where each number represents the single block to be read from the storage device. The stream contains two sequential runs: $\langle 1, 2, 3\rangle$, and $\langle 11, 12, 13, 14\rangle$. The blocks prefetched by the PA technique are 2, 3, 4, 46, 68, 84, 12, 13, 14, 15, 33, 77, and 99. The blocks prefetched by PoM are 2, 4, 46, 68, 84, 12, 14, 33, 77, and 98. The blocks prefetched by PoH are 3, 4, 13, 14, and 15. The cache size needed for obtaining the maximal hit rate by each technique is one, since prefetched blocks 4, 15, 46, 68, 84, 33, 77, and 98, can be evicted from the cache without loss in hit rate, because on-demand requests for these blocks will never arrive. The maximum prefetch hit rate of this stream is 5/13. The hit rate achievable by PA is 5/13, and that by PoM and PoH is 3/13.

An I/O workload consists of interleaved requests from various streams. Let $a_{i,j}$ denote request a_i from stream j. Consider an I/O workload comprising requests from five streams. Since requests from the five streams may be interleaved arbitrarily—$\langle\ a_{5,1},\ a_{2,3},\ a_{8,4},\ a_{9,4}\ a_{6,1},\ a_{19,2},\ \ldots,\ a_{3,3}\ \rangle$—consecutive requests from a stream may be separated by several requests from other streams. This arbitrary mixing of requests from different streams impacts the sizing requirement of the prefetch cache in the following way. A sequential run from a stream may become embedded in a long sequence of requests from other streams. If a prefetching technique prefetches a request from this run, then it must ensure that the prefetched request is preserved in the cache until it is hit. However, the interleaved requests may themselves trigger prefetches and lead to insertions into the cache. Therefore, the cache must be sized to be large enough so that a prefetched request is preserved in the cache until it is hit, with a high probability.

The cache replacement scheme must evict requests without jeopardizing prefetched data that is yet to be hit. It must preserve a unhit prefetched request in the cache as long as possible allowing sufficient time for its on-demand request to arrive in workload comprising an arbitrary number of streams. It may however, evict hit prefetched requests immediately. Thus, from the viewpoint of the replacement scheme, there is no difference between two cache blocks other than the time at which they were loaded into the cache. The First In First Out (FIFO) replacement scheme is a good choice in this setting because it evicts blocks based on their loading order. Note that a scheme like Least Recently Used (LRU) is not valid in this context because all hit blocks are evicted immediately from the prefetch cache.

The workload characteristics that impact on the size of a sequential prefetch cache are the number of streams, the sequentiality of each stream, the arrival rates of the streams, and the interleaving pattern of the streams. The size requirement of a prefetch cache not only depends on the workload characteristics but also on the prefetching scheme itself. For example, blind prefetching schemes like PA and PoM may require larger cache sizes in order to achieve their maximum hit rate than informed prefetching schemes such as PoH. An off-line prefetching scheme that reserves a cache line for each sequential stream in the workload can get the maximum hit rate for the workload. The *optimal* cache size is the number of sequential streams in a workload. The *minimum* cache size for a scheme on a given workload is the cache size necessary for the scheme to achieve its maximum hit ratio on that workload. The next example illustrates some of the points stated here.

Example 2: Consider a workload comprising two partly sequential— $\langle$ 909, 910, 588, 592, 593, 736, 737 $\rangle$ $\langle$ 1659, 1769, 1749, 1750, 1808, $\rangle$—and one random stream: $\langle$10, 12, 81, 36, 25, 46, 61, 89, 05, 01, 42, 19, 83, 16, 33, 13, 38, 74, 04$\rangle$ Suppose these streams are interleaved in the workload as follows: $\langle$ 10, 909, 82, 81, 1659, 36, 25, 46, 1769, 1749, 61, 89, 910, 1750, 05, 01, 1808, 588, 592, 593, 736, 42, 19, 83, 16, 737, 33, 13, 38, 74, 04$\rangle$ For clarity of exposition, let us assume that the prefetch cache is sufficiently large and can thus hold all loaded blocks without any evictions. The PA technique will prefetch a contiguous block for every request in the workload resulting in 31 total prefetches. The PoM technique will prefetch 27 blocks—all but those contiguous to its following four hits: 910, 1750, 593, 737. The PoH technique will prefetch the following four blocks: 911, 1751, 594, 738. The maximum number of prefetch hits possible for this workload is four. The number of hits achieved by PA, PoM, and PoH on this workload is four, four, and zero, respectively. The optimal cache size for this workload is two. The cache size necessary for PA, PoM, and PoH to achieve its maximal hit rates is 13, 11, and 0 respectively. None of

the schemes (except PoH, trivially) achieve their maximal hit rate on this workload with a cache size of two: PA and PoM both get a single hit (593). The only scheme that can achieve the maximum hit rate with a cache size of two is an off-line prefetching technique that assigns a cache line for each stream. As this example shows, the cache size is critical to a prefetching technique achieving its maximal hit rate. Each of the techniques prefetches a different number of requests so the prefetch cache size requirement varies.

Unlike PoH, PA and PoM initiate a prefetch without identifying whether the request is part of a sequential run. Therefore, PA and PoM prefetch upon arrival of every request, random or sequential, and subsequently need a larger cache to ensure that prefetched blocks from sequential runs are not evicted early. In the next section, we present a simple sizing module that can be added on to any prefetching technique to ensure that the cache is large enough.

V. Online sizing

Upon arrival of each I/O request, an online sizing module must decide whether to increment, decrement, or leave unchanged the size of the cache. The goal of the sizing module is to determine the smallest size that ensures prefetched blocks from sequential runs remain in the cache until the on-demand requests for the prefetched blocks arrive. The sizing module assumes no knowledge of file systems or streams. The cache replacement scheme is FIFO, a reasonable scheme, given that hit blocks are removed from the cache and that the only difference between two cached blocks is their insertion time. Since the focus of this paper is the sizing component, we assume that all other parameters of the schemes are set to basic values: each on-demand and prefetch request is for one block, and each cache line holds one block of data. The sizing module would be valid if the settings are changed, as long as the module scales appropriately. For example, if several blocks are prefetched at a time, then more cache lines would have to be set aside for each prefetch, but cache lines relating to a single prefetch would be treated as a set.

The details of such an online sizing scheme are listed in the pseudo-code of Scheme 1. The sizing scheme has to determine if the cache is large enough to hold prefetched blocks from sequential runs until their on-demand requests arrive. A block is loaded into the FIFO insertion end, and each time another block is inserted, this block will move toward the eviction end. In order to know if the prefetch cache is too small, we move each request evicted from the prefetch cache into the on-demand cache, and label the request as an evicted request. If an on-demand request for this evicted prefetch request arrives, then the prefetch cache is too small and the size of the prefetch cache is incremented. Whenever a request hits in the prefetch cache or misses in both the prefetch and the on-demand cache, the sizing scheme leaves the prefetch cache size unchanged.

The sizing scheme must also determine if the prefetch cache is too large. The number of cache lines needed is equal to the maximum number of prefetch cache insertions that can occur between the loading of a prefetched block and the arrival of its on-demand request. Since each insertion causes a prefetched block to move toward the FIFO eviction end, one would expect that the FIFO eviction end of a cache would receive hits unless the cache is too large. The eviction end of a cache is monitored, and the size of the cache is decremented if the eviction end cache line does not receive any hits during the monitoring period. For example, suppose the prefetch cache size is set to ten cache lines whereas the workload contains only two completely sequential streams. Requests from the two streams will be loaded at the insertion end of the prefetch cache. When these requests are hit, they will be evicted from the prefetch cache and succeeding requests will be loaded into cache lines at the insertion end of the prefetch cache. Eventually, all hits and insertions will be confined to the insertion end of the cache and the eviction end of the cache will see no hits. The sizing scheme exploits the presence of this "quiet zone" at the eviction end of an inflated cache. The scheme monitors the eviction end of the cache for a sufficiently long period (pseudo-code lines 12-16). During this period, if the requests residing near the eviction end do not receive any hits, then the scheme concludes that the cache is inflated and decrements the cache size. The request that is evicted as a result of the cache size reduction is loaded into the on-demand cache. This provides the decrement decision a level of self-correction: if an adequately sized cache is erroneously decremented, then the evicted request will lead back to an increment once the request for the evicted request arrives in the workload. Any decision to decrement the cache size is also ignored if the cache size has been incremented during the monitoring period: this ensures that the decision to increment trumps the decision to decrement because the former is likely

Scheme 1 Online prefetch cache sizing

```
 1: noEvictionEndHits ← true; noIncr ← true
 2: for every request req do
 3:     if req is a prefetch cache miss then
 4:         if req is a non-rereference hit in the on-demand
            cache then
 5:             Increment prefetch cache size by one line
 6:             noIncr ← false
 7:         end if
 8:     else if req is hit near the eviction end then
 9:         noEvictionEndHits ← false
10:     end if
11:     reqCount++
12:     if reqCount == monitoringPeriod then
13:         if noEvictionEndHits and noIncr then
14:             Decrement cache size by one line
15:             Move evicted request into the on-demand
                cache
16:         end if
17:         reqCount ← 0
18:         noEvictionEndHits ← true; noIncr ← true
19:     end if
20: end for
```

to be based on a more reliable indicator.

The monitoring period is set to the sum of current size of the prefetch cache and the size of the on-demand cache. The monitoring period is tracked by the variable reqCountwhich is initialized to 0; is incremented when an I/O request arrives (line 11); and reset to 0 when the value equals the monitoring period (line 17). The larger the sum of the on-demand and prefetch cache sizes, the longer is the monitoring period. A longer monitoring period is advantageous because it reduces the chance that an adequate prefetch cache size is erroneously decremented. On the other hand, a long monitoring period also delays the reduction of an inflated cache size to an economical value.

VI. Simulation results

The performance of the proposed sizing scheme is validated through simulations. We ran the simulations using the CMU Disksim [5] simulator. The simulator was used in slave mode by a caching and sizing module that implemented the PoM, PoH, and PA prefetching schemes and the online sizing scheme. The simulations were carried out using synthetically generated SPC-2-like read workloads [1] and partly sequential workloads. We tested the sizing scheme both under uniform and nonuniform interleaving as well as under static and dynamic workloads. We postpone comparison to other sizing schemes [12], [17] to future work.

A. Performance under uniform interleaving

In this experiment, the workload contains a total of 100 streams, all of identical sequentiality interleaved uniformly at random. The sequentiality is varied from zero (all streams completely random) to one (all streams completely sequential). The hit rate obtained by PA, PoH, and PoM is measured. The maximum prefetch cache size set by each scheme during each simulation run is recorded. Ten such runs are executed and the average of the maximum prefetch cache sizes and the hit rates obtained is computed. The results are plotted in Figure 1. The bottom graph in Figure 1 shows the hit rates obtained by the three schemes using the online sizing scheme (solid points) and the maximum theoretical hit rates achievable by those schemes (dashed lines). It is clear that the hit rate obtained by each scheme is within a few percent of the hit rate achievable by that prefetching scheme.

The top graph in Figure 1 shows the maximum prefetch cache size set by the online sizing scheme when it is coupled with each of the three prefetching schemes. The optimal prefetch cache size (dashed horizontal line) is 100 when the number of partly sequential schemes is 100 and zero when all the streams are random (zero sequentiality). PA achieves the highest hit rate of all the schemes but also requires the largest prefetch cache size. This is because PA prefetches succeeding requests blindly. When sequentiality is low, this results in a large number of requests being prefetched into the prefetch cache that are never hit. Nonetheless, these requests end up nudging hitable prefetched requests out of the cache. Such prehit evictions end up in the on-demand cache and hits to them are detected by the sizing scheme which then correctly increments the prefetch cache size. In effect, due to PA's blind prefetching policy, the sizing scheme is forced to maintain a large prefetch cache.

The prefetch cache size set by the sizing scheme for PoM is lower than PA although still close to that used by PA. Like PA, PoM too uses a blind prefetching policy. Thus, for low sequentiality workloads, it ends up paying a heavy price in terms of prefetch cache space as seen in the left portion of Figure 1. Moreover, because PoM only prefetches on misses, it fails to obtain hits on almost half of all the hitable requests. PoM makes up for this suboptimal performance in its reduced prefetching cost. Since PoM piggybacks its prefetch request onto the on-demand request that fetches the requested data, it saves the disk system an additional seek that would have resulted if a separate prefetch request were to be issued. This can result in big savings when the disk is heavily utilized.

Figure 1 also shows that among the three prefetching techniques, PoH, when coupled with the sizing scheme, is able to use the prefetch cache most frugally. This feature is attributable mainly to PoH's detect-and-prefetch approach to prefetching in contrast to PA and PoM's blind prefetching.

B. Nonuniform interleaving

In the first experiment, all streams are of identical sequentiality and contribute the same average number of requests to the workload. In effect, the average number of intervening requests from other streams between two requests from the same stream, is identical for all streams. In this experiment, the sizing scheme is subjected to a workload comprising streams of different rates and sequentialities. The workload contains a total of 100 streams. Of these, 50 streams are completely random (sequentiality zero) and the remaining 50 streams are of identical sequentiality. The sequentiality of these streams is varied from zero to one. The arrival rates of the random streams are identical and are set to be twice the arrival rates of the partly sequential streams. The arrival rates of all the partly sequential streams are also set to be identical.

Figure 2 shows the hit rates and cache sizes for each of the three prefetching techniques when coupled with the online sizing scheme. Comparing with Figure 1 reveals two striking differences. First, the effect of sequentiality on the prefetch cache size is weaker as seen by the relative flatness of the cache size curves. This is readily explained by observing that because half the streams in the workload are completely random and arriving twice as fast as the sequential streams in the workload, the total sequentiality of the workload in the second experiment is much lower than in the first. As a result, even when the sequential streams are completely sequential (right end of the top graph in Figure 2), the prefetch cache lines needed is larger than the number of sequential streams, owing to the presence of the high intensity random streams. Second, the hit rates obtained by each of the prefetching schemes are substantially lower than those observed in the first experiment due to the presence of the random streams. The theoretical hit rate obtained by a prefetching scheme on a

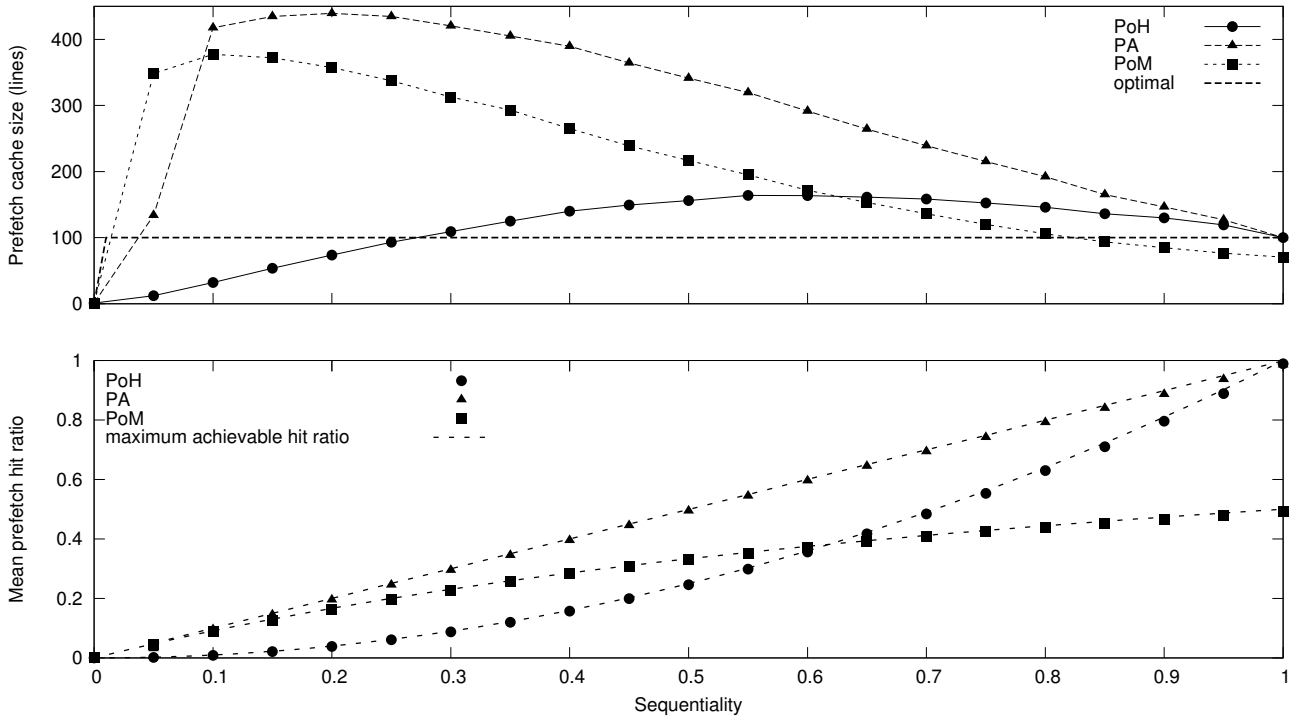

Fig. 1. The prefetch hit ratio obtained as a function of workload sequentiality with the online prefetch cache sizing scheme listed in Pseudo-code 1. The workload contains $M=M_s=100$ streams of identical sequentiality (X axis).

workload comprising uniformly interleaved streams is defined to be the average of the hit rates obtained by the scheme on each constituent scheme in isolation. When the workload comprises non-uniformly interleaved streams, however, the hit rate of the workload must be generalized appropriately. In this case, the hit rate of the workload is the hit rate obtained by the prefetching scheme on each constituent stream, weighted by its relative intensity.

Figure 2 shows that the high intensity random streams have a significant impact on the cache size when the prefetching scheme used is PA or PoM. Since these two schemes prefetch blindly, the high intensity random streams result in a large number of useless prefetches which in turn result in the eviction of a large number of useful prefetches. This forces the sizing scheme to inflate the prefetch cache to a size that can accommodate the useless prefetching of random requests without evicting hitable prefetched requests. In summary, the online sizing scheme is able to deliver near-optimal hit rates even under this relatively challenging workload.

C. Sizing under a dynamic workload

In this next experiment, a completely dynamic workload is used. While arrival rates and sequentialities of the streams are held constant, unlike experiments in the previous sections, the number of active streams is allowed to vary arbitrarily. The starting time of each new stream is chosen at random within the experimental interval. A stream once opened remains active for a fixed duration. Figure 3 shows example runs under two such scenarios. In both scenarios, 150 random streams open and persist throughout the duration of the experiment. Another 150 streams are completely sequential in the first scenario (Figure 3 left) and partly sequential with sequentiality 0.8 in the second scenario (Figure 3 right).

It is clear from Figure 3 that the online sizing scheme is able to adjust the prefetch cache size in response to the changing workload. When coupled with an intelligent prefetching scheme like PoH, the cache size maintained by the scheme is within a small factor of the minimum necessary cache size. As indicated in previous results, the hit rate obtained in both scenarios is also within a few percentage of the maximum hit rate achievable by each prefetching scheme.

VII. Analysis of hit rate and cache size

The online sizing scheme increases the size of the cache whenever a sequential block is evicted before being hit. A sequential block is evicted when the number of cache insertions between two consecutive requests of a sequential run exceeds the number of cache lines. As the cache size is decreased, the likelihood of a sequential block being evicted due to cache insertions increases, and consequently lowers the

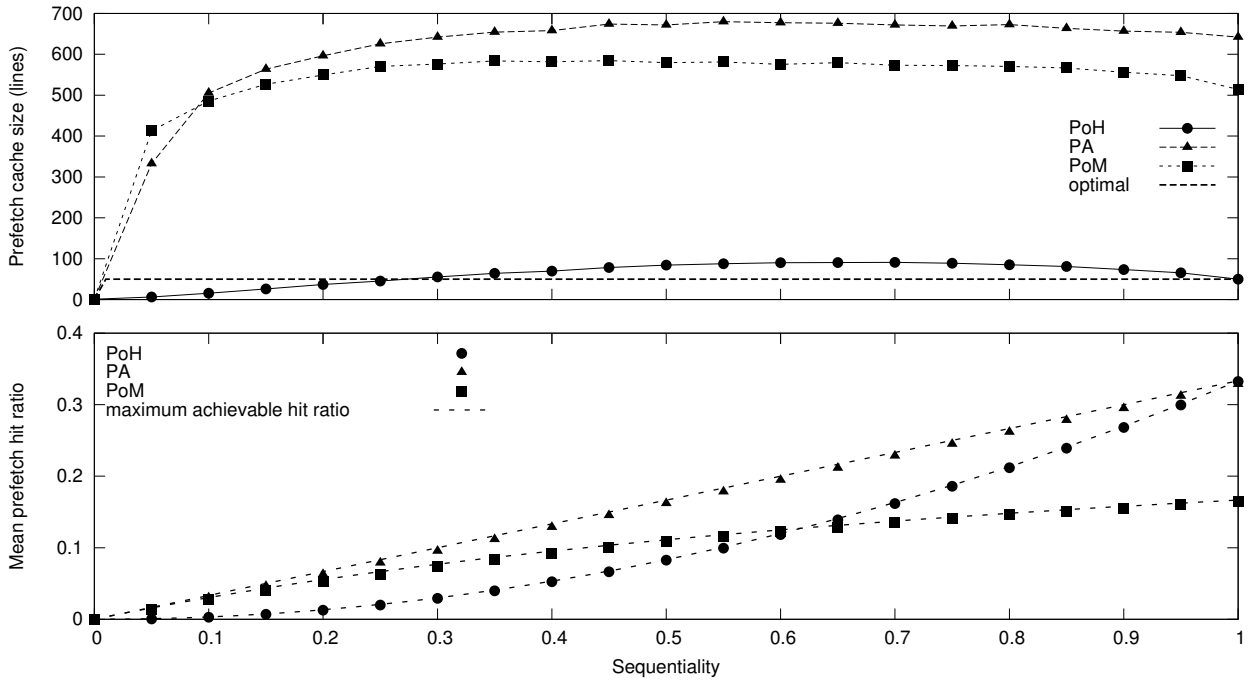

Fig. 2. The prefetch hit ratio obtained as a function of workload sequentiality with the online prefetch cache sizing scheme listed in Pseudocode 1 under a nonuniform workload. The workload contains 50 completely random and 50 streams of arbitrary (X axis) sequentiality. The arrival rate of the random streams is twice that of the partly sequential streams.

achievable hit rate.

In this section, we quantify the change in hit rate in relation to the cache size, analytically. We make the following assumptions about the workload. The number of sequential stream, $\mathsf{M_s}$, and the number of random streams, $\mathsf{M_r}$, in the workload is given. The streams comprising the workload are independent and interleaved uniformly at random. Each request in a stream $\langle r(,r)^\star \cup (,s)^\star \rangle$ is generated independent of other requests for the stream. The type of each request—sequential or random—is decided by flipping a coin biased with the sequentiality of the stream. (Synthetic I/O workloads in storage system simulators like Disksim [5] are generated using this assumption.) The sequentiality of the stream, S, is the probability that the next request generated is sequential. We let $\mathsf{R} = 1 - \mathsf{S}$. Thus, the subsequence $\langle r, s \rangle$ is obtained with probability S, the subsequence $\langle r, r, s \rangle$ is obtained with probability $\mathsf{R} \times \mathsf{S}$, and the subsequence $\langle \underbrace{r, r, ..., r}_{l}, s \rangle$ is obtained with probability $\mathsf{R}^{l-1} \times \mathsf{S}$. This is the probability mass function of the geometric random variable [21]. Thus, viewing a stream as an independent sequence of sequential and random requests, the length of a random (sequential) run has a geometric distribution, and it follows that, the expected (average) length of a sequential run is given by $1/\mathsf{R}$, the expected length of a random run is given by $1/\mathsf{S}$, the expected length of a run is given by $1/(\mathsf{S} \times \mathsf{R})$, and the expected number of runs in a stream with n requests is $n \times \mathsf{S} \times \mathsf{R}$. We now compute the maximum prefetch hit rate per stream, for each of the prefetching schemes. Since PA prefetches every request,

$$\mathsf{H}_{maxPA} = \mathsf{S}.$$

The PoH technique only prefetches identified sequential requests from all streams. All requests in a sequential run, except for the first sequential request, hit in the cache. The expected number of sequential runs in a stream with n requests is $n \times \mathsf{S} \times \mathsf{R}$. Therefore, the expected maximum hit rate is $(\mathsf{S} - \mathsf{S} \times \mathsf{R})$. Thus, for a given stream:

$$\mathsf{H}_{maxPoH} = \mathsf{S}^2.$$

The PoM technique prefetches on a cache miss. If a sequential run consists of an even number of requests, then half the requests hit in the cache. If a sequential run consists of an odd number of requests, then the ceiling of half the requests hit in the cache. Thus, the proportion of hits on even length sequential runs is $\mathsf{S}/2$, on odd length sequential runs is $\mathsf{S}(1 + \mathsf{R})/2$, and the probability of obtaining an odd length sequential run is $1/(1 + \mathsf{S})$. Hence, the maximum mean hit

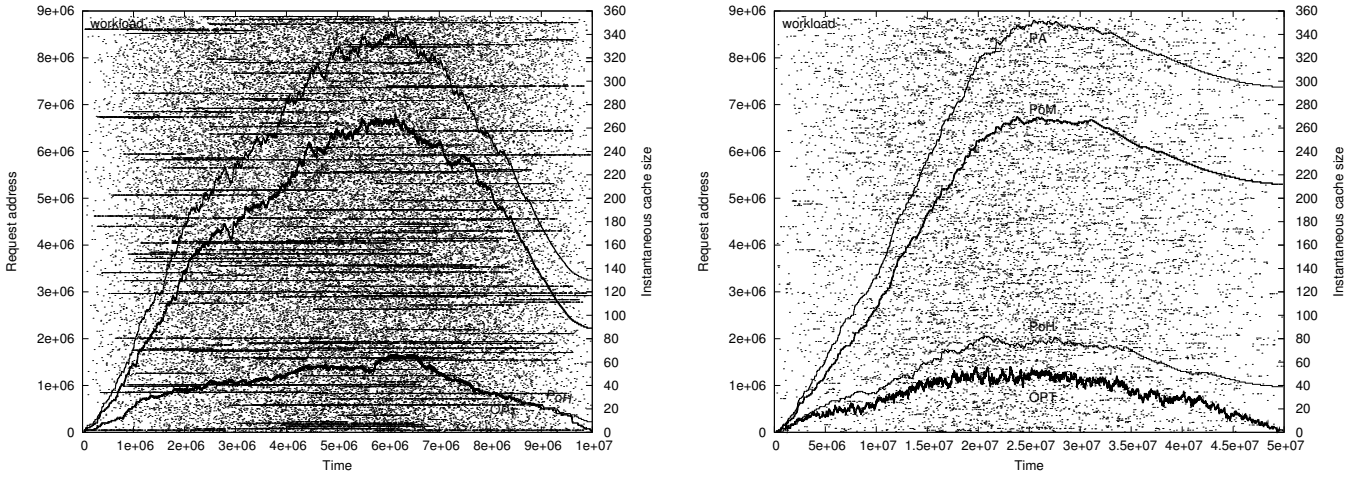

Fig. 3. An example run of the online sizing scheme on a dynamic workload. *Left:* A total of 150 completely sequential streams open and close dynamically while a background of 150 random streams persists. *Right:* A total of 150 streams of 80% sequentiality open and close dynamically while a background of 150 random streams persists.

rate for a stream under PoM is

$$\mathsf{H}_{maxPoM} = \frac{\mathsf{S}}{1+\mathsf{S}}$$

An I/O workload contains interleaved streams, and the maximum hit rate per stream can be achieved only if the cache is large enough so that every prefetched block from the sequential runs remain in the cache until the on-demand request arrives. The next theorem states the achievable hit rate for a prefetching technique if the number of cache lines is set to some number L.

Theorem 1: Suppose there are M uniformly interleaved streams in a workload, of which $\mathsf{M_s}$ streams are sequential or partly sequential. In each (partly) sequential stream, assume that with probability S, the next request is sequential to the current request. For this workload, let H_{max} represent the maximum hit rate that can be achieved by a prefetching technique for any stream i of the $\mathsf{M_s}$ streams.

Then, for a cache employing a FIFO replacement policy, the hit rate for stream i that can be achieved using a cache with L lines is at least $\mathsf{H}_{max}\left(1-\left(\frac{\mathsf{M}-1}{\mathsf{M}}\right)^{\mathsf{L}}\right)$.

Proof: Suppose a sequential prefetch request s_i from stream i is inserted into the cache at FIFO insertion end. We analyze the scenario when the next on-demand request from a stream j arrives.

With probability $1/\mathsf{M}$, $i=j$ and this on-demand request corresponds to prefetch s_i, so prefetch s_i hits in the prefetch cache.

With probability $\frac{\mathsf{M}-1}{\mathsf{M}}$, the on-demand request is from a stream other than i. If this on-demand request results in a prefetch cache insertion, then the cache line for s_i either retains its position in the FIFO queue or moves 1 line closer to the FIFO eviction end. If this on-demand request does not impact the prefetch cache, then the cache line for s_i retains its position in the FIFO queue.

Suppose the on-demand request corresponding to s_i does not arrive during the $(\mathsf{M_s}-1)$ on-demand request arrivals after loading s_i at the FIFO insertion end. Each arrival will result in s_i either retaining its position in the FIFO cache or moving 1 line closer to the FIFO eviction end. Since the cache size is L, prefetch request s_i is guaranteed to stay in the cache during the next $(\mathsf{L}-1)$ on-demand request arrivals. In the worst case, after $(\mathsf{L}-1)$ arrivals, the cache line holding s_i would be at the FIFO eviction end and would face eviction from the next insertion from a newly identified sequential run.

Thus, for a cache size of L, with probability $\left(1-\left(\frac{\mathsf{M}-1}{\mathsf{M}}\right)^{\mathsf{L}}\right)$, a sequential request will hit in prefetch cache within the next L insertions. If the maximum hit rate for the workload that can be achieved by the prefetching technique is H_{max}, then with a cache of size L, the prefetching technique would get the hit rate specified in the theorem. ■

The bound proved above is loose. Both PA and PoM blindly prefetch requests from the M streams, therefore the minimum cache size is M since requests from all M streams are prefetched. On the other hand, PoH only prefetches identified sequential requests from the $\mathsf{M_s}$ streams, so the minimum prefetch cache size is $\mathsf{M_s}$. This leads to a tighter bound.

Corollary 1: For a cache size of L lines, a lower bound on the hit rate is given by:

$$\begin{aligned}\mathsf{H}_{PoH} &\geq \mathsf{H}_{maxPoH}\times\left(1-\left(\frac{\mathsf{M_s}-1}{\mathsf{M_s}}\right)^{\mathsf{L}}\right)\\ \mathsf{H}_{s} &\geq \mathsf{H}_{max_s}\times\left(1-\left(\frac{\mathsf{M}-1}{\mathsf{M}}\right)^{\mathsf{L}}\right), s\in\{\text{PoM, PA}\}\end{aligned}$$

If the cache size L is set to $k\times\mathsf{M}$ for PA and PoM and $k\times\mathsf{M_s}$ for PoH for some positive integer k, then, $((\mathsf{M}-1)/\mathsf{M})^{k\mathsf{M}}\rightarrow e^{-k}$ and $((\mathsf{M_s}-1)/\mathsf{M_s})^{k\mathsf{M_s}}\rightarrow e^{-k}$ as $\mathsf{M},\mathsf{M_s}\rightarrow\infty$. Thus, for large M and $\mathsf{M_s}$, the hit ratios are bounded as below:

$$\mathsf{H}_{s} \geq \mathsf{H}_{max_s}\left(1-e^{-k}\right), s\in\{\text{PoH, PoM, PA}\}$$

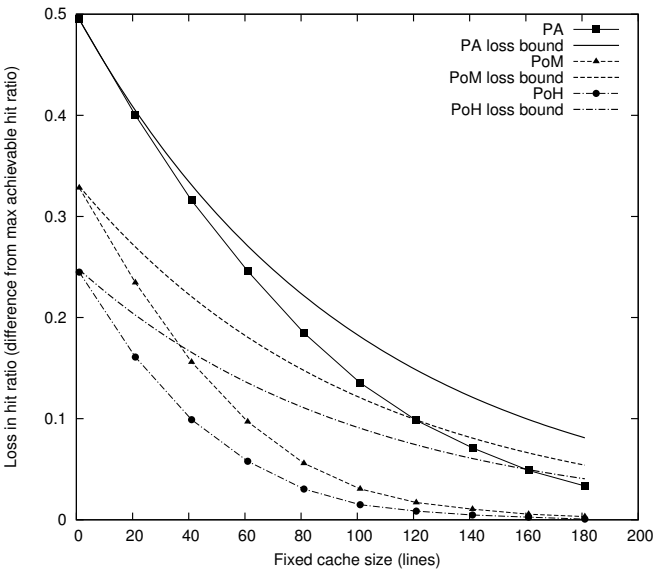

Fig. 4. Effect of increasing cache size on the loss in hit rate for a workload comprising 100 streams of sequentiality 0.5.

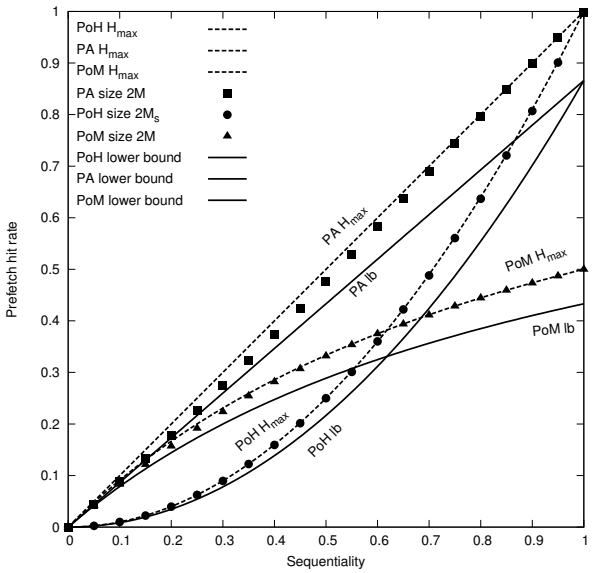

Fig. 5. Prefetch hit ratio obtained as a function of sequentiality with the prefetch cache size set to 2M, where M=M_s=100 streams of identical sequentiality (X axis).

This leads to the following observation. As the prefetch cache size is decreased from the minimum size necessary for a prefetching scheme, the loss in hit rate could increase significantly. Figure 4 confirms this hypothesis. In this simulation, the cache size was varied and the hit rate obtained by the three techniques on a workload comprising 100 streams of sequentiality 0.5 was measured. As the cache size is increased, the loss in hit rate drops dramatically. In other words, a small decrease in cache size can lead to a significant loss in hit rate.

Figure 5 shows the hit rate provided by the three techniques for a cache size set to 2M for PA and PoM, and 2M_s for PoH. Dashed lines indicate the maximum achievable hit rate for each technique and the solid lines mark the loose lower bound from the theorem above. The filled points show the hit rate measured from simulations. From these results, one can conclude that the three techniques can achieve nearly their maximal hit rate if the cache size is maintained greater than about twice the optimal size. However, the number of streams—which is necessary to compute the optimal size—is typically unknown. Therefore, our proposed online sizing scheme is useful in maintaining the cache at a size that is adaptively matched to the workload and the prefetching scheme.

VIII. Related Work

Cache size has a significant effect on a caching/prefetching technique's performance, since it determines the hit rate. Most studies have focused on on-demand caches where data are kept for re-reference hits. The $\sqrt{2}$ rule follows from an empirical observation that the cache miss rate decreases as a power law of cache size [**?**], [20]. Hartstein et al. [14] proved this rule both theoretically and through simulation. Jelenkovic et al. showed the relationship between the cache miss rate and the cache size for a LRU scheme with statistically dependent request sequences [16]. Singh et al. [23] developed a mathematical model that computes the dependence of the miss rate on the cache size.

There are some studies on web caching which focus on minimal total cost of caching given a cache size. A caching/paging algorithm called Greedy-Dual-Size is proposed and its competitiveness against an offline optimal algorithm is provided [6], [8], [28]. Curcio et al. [9] combined the Greedy-Dual-Size caching scheme with a prefetching scheme which needs information from the client user for determining which file to prefetch. Teng and Gumaer [24] managed the cache as multiple buffer pools and determined minimum and maximum buffer sizes. Bruening and Amarasinghe [4],time algorithm for dynamically bounding software code cache [4]. There are far fewer papers on prefetch cache sizing. Tse et al. [25] concluded that performance of a prefetching technique generally improves as the cache size increases. Baek and Park [3] studied the effect of cache size on ASP, a prefetching scheme they developed. They showed that ASP performs better than other techniques for small sized caches by maximizing the hit rate of both prefetched data and on-demand data. In these studies, however, the prefetched data and on-demand data are stored in the same cache and the cache size is fixed.

The separation of prefetched data from the rest of the cache reduces the probability of eviction of prefetched data. Li and Shen et al. [17] implemented a prefetch cache sizing scheme based on a gradient descent-based greedy algorithm. The SARC technique [12] manages storage prefetched data and on-demand data by separating them into different LRU lists and dynamically adjusting the size of the two lists based on the sequentiality of the I/O workload. Sequential data are placed in the SEQ list and random data are placed in the $RANDOM$ list. The sizes of these two lists are adjusted by monitoring the miss rate of SEQ and hit rate of $RANDOM$.

The TaP technique [18] is a storage prefetching scheme that uses a memory table for sequential pattern detection. The prefetched data are saved in a prefetch cache, which can be adjusted dynamically based on the sequentiality of the I/O workload.

As the related work shows, there are very few papers that discuss prefetch cache sizing [12], [17], [18]. In fact, these papers focus on developing a prefetching technique, so sizing is just a secondary focus. This is the first paper that presents a systematic analysis of the impact of sizing on hit rate for various prefetching techniques. This is also the first paper to develop a sizing scheme that can be incorporated in the standard sequential prefetching techniques.

IX. Conclusions

This paper tackles the space requirements of disk array prefetch caches—an expensive and scarce resource. To our knowledge, this is the first paper to systematically address the issues and challenges in determining the adequate size of a storage prefetch cache for dynamic I/O workloads. We explore the tradeoff between sizing versus hit rate. We then compute the hit rate achievable with a given prefetch cache size for various sequential prefetching techniques. We propose an online prefetch cache sizing scheme that is able to dynamically maintain the cache size within a small factor of the minimum size necessary while achieving hit rates of over 95% of the maximum hit rate achievable by three popular prefetching schemes. We validate our sizing scheme through simulations with both uniformly and nonuniformly interleaved workloads and static and completely dynamic workloads. Our simulations show that the sizing scheme achieves near-maximal hit rates under all the scenarios tested. The cache size maintained is close to the minimum necessary with the PoH scheme and within a small factor of the minimum with blind prefetching schemes such as PoM and PA.

Acknowledgments

The first author was supported in part by a grant from the Office of Naval Research and by the UNH CEPS Teaching Award when this work was underway at UNH. The authors thank HP Labs for graciously providing funding for the publication of this paper at the MASCOTS 2010 conference. The first author thanks Mingju Li for discussions and comments.

References

[1] SPC Benchmark-2(SPC-2) Official Specification, version 1.2.1. Tech. rep., Storage Performance Council, Effective 27 Sept. 2006. http://www.storageperformance.org/specs.

[2] Anacker, W., and Wang, C. P. Performance evaluation of computing systems with memory hierarchies. *IEEE Transactions on Electronic Computers 16*, 6 (1967), 764–773.

[3] Baek, S. H., and Park, K. H. Prefetching with adaptive cache culling for striped disk arrays. In *Proceedings of the USENIX Annual Technical Conference* (June,2008).

[4] Bruening, D., and Amarasinghe, S. P. Maintaining consistency and bounding capacity of software code caches. In *Code Generation and Optimization archive Proceedings of the international symposium on Code generation and optimization* (2005), pp. 74–85.

[5] Bucy, J. S., and Ganger, G. R. The DiskSim simulation environment version 4.0 reference manual. Tech. Rep. CMU-PDL-08-101, Carnegie Mellon University, School of Computer Science, May 2008.

[6] Cao, P., Cao, P., Irani, S., and Irani, S. Cost-aware www proxy caching algorithms. In *In Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems* (1997), pp. 193–206.

[7] Chow, C. K. Determination of cache's capacity and its matching storage hierarchy. *IEEE Trans. Computers 25*, 2 (1976), 157–164.

[8] Cohen, E., and Kaplan, H. Caching documents with varying sizes and fetching costs: an LP-based approach. *Algorithmica 32*, 3 (2002), 459 – 466.

[9] Curcio, M., Leonardi, S., and Vitaletti, A. *Algorithm Engineering and Experiments*. Springer Berlin / Heidelberg, 2002.

[10] Farley, M. *Storage Networking Fundamentals: An Introduction to Storage Devices, Subsystems, Applications, Management, and Filing Systems*. Cisco Press, 2004.

[11] Gill, B. S., and Bathen, L. A. D. AMP: Adaptive multi-stream prefetching in a shared cache. In *Proc. of USENIX 2007 Annual Technical Conference* (Feb 2007), 5th USENIX Conference on File and Storage Technologies.

[12] Gill, B. S., and Modha, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. of USENIX 2005 Annual Technical Conference* (2005), pp. 293–308.

[13] Grimsrud, K. S., Archibald, J. K., and Nelson, B. E. Multiple prefetch adaptive disk caching. *IEEE Transactions on Knowledge and Data Engineering 5*, 1 (1993), 88–103.

[14] Hartstein, A., Srinivasan, V., Puzak, T. R., and Emma, P. G. Cache miss behavior, is it $\sqrt{2}$? *Proceedings of the 3rd conference on computing frontiers* (2006), 313 – 320.

[15] Jelenkovic, P. R., and Radovanovic, A. Least-recently-used caching with dependent requests. *Theor. Comput. Sci. 326*, 1-3 (2004), 293–327.

[16] Jelenkovic, P. R., Radovanovic, A., and Squillante, M. S. Critical sizing of LRU caches with dependent requests. *Journal of Applied Probability 43*, 4 (2006), 1013–1027.

[17] Li, C., and Shen, K. Managing prefetch memory for data-intensive online servers. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (2005), vol. 4, pp. 253 – 266.

[18] Li, M., Varki, E., Bhatia, S., and Merchant, A. TaP: Table-based prefetching for storage caches. In *6th USENIX Conference on File and Storage Technologies (FAST '08)* (2008), pp. 81–97.

[19] Li, Z., Chen, Z., Srinivasan, S. M., and Zhou, Y. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3th USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 173–186.

[20] Przybylski, S. A., Horowitz, M., and Hennessy, J. L. Characteristics of performance-optimal multi-level cache hierarchies. In *ISCA* (1989), pp. 114–121.

[21] Ross, S. M. *Probability models for computer science*. Academic Press, San Diego, CA, 2002.

[22] Ruemmler, C., and Wilkes, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994), 17–29.

[23] Singh, J., Stone, H., and Thiebaut, D. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Trans. on Computers 41*, 7 (1992), 811–825.

[24] Teng, J. Z., and Gumaer, R. A. Managing IBM database 2 buffers to maximize performance. *IBM System Journal 23*, 2 (1984), 211 – 218.

[25] Tse, J., and Smith, A. J. Performance evaluation of cache prefetch implementation. Tech. Rep. UCB-CSD-95-877, Computer Science Division (EECS), University of California, Berkeley, June 1995.

[26] Wong, T. M., and Wilkes, J. My cache or yours? Making storage more exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 161–175.

[27] Worthington, B. L., Ganger, G. R., and Patt, Y. N. Scheduling algorithms for modern disk drives. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1994), ACM Press, pp. 241–251.

[28] Young, N. E. On-line file caching. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998).