Black box to Gray box
Using sequential prefetching to move toward file-aware storage

Elizabeth Varki
University of New Hampshire

Arif Merchant
Hewlett-Packard Labs

Swapnil Bhatia
PARC

Abstract

Storage software has not kept pace with storage hardware. The software implemented in disk array systems is limited by the lack of knowledge of storage data resulting from the minimal I/O interface. This paper shows that even without any information on files, storage software can extract significant information about storage data. Adding meaning to storage data has a two-fold benefit. Firstly, storage data placement can be based on data access patterns and secondly, smarter storage software techniques can be implemented. This paper demonstrates the extraction of workload information by standard sequential prefetching techniques. A prefetch and cache sizing technique that adapts to the workload is designed.

1 Introduction

Computer hardware and software have evolved rapidly since the development of the first vacuum-tube electronic computer in 1937. The first commercial hard disk was introduced by IBM in 1957, but until the early 1990s there was little development on the storage front. Faster and smaller disks were developed, but disks were still mechanical devices tied to the file system, incapable of performing intelligent tasks. Disk controllers performed basic tasks like disk scheduling, error checking and remapping from bad sectors to good sectors. In 1988, the storage landscape changed with the proposal of RAID systems [42]. This technology became the driving force that transformed storage from a "dumb mechanical device" to a "smart storage system."

Current RAID systems are disk arrays with powerful controllers, large memory units and large caches. Technology such as SAN and storage virtualization allow storage units to be independent of file systems. The disk array controllers have the capability of running complex algorithms to speed up storage access. In fact, large storage systems often have more computational power and memory than the workstations they serve. Thus, it is reasonable to expect storage to step up to the plate and address the challenge of speeding data access. While it is true that current disk arrays implement scheduling, load balancing and caching techniques that are far beyond the capabilities of old disks, storage software is still very primitive when compared to file system software. Current storage software is not close to harnessing the power of storage hardware. The key reason for this shortfall is that storage devices have no information about storage data.

Storage devices store files. File systems control and manage file data placement on disks. The file system talks to storage via I/O read and write requests. A read request has the addresses (*i.e.,* block numbers) of the blocks to be read, while a write request also transmits the data to be written to disks. The function of storage is to transmit the requested blocks when a read request arrives, and to write the data to the requested blocks when a write request arrives. No information about files or applications are transmitted down to the storage layer, so storage does not know about the file whose data blocks are being retrieved or the application for whom the data blocks are being retrieved. From a storage device's viewpoint, the data blocks have no meaning, so storage data blocks and the I/O workload are a "black box." In order to develop software that speeds data access, information about data access patterns and access frequency is needed. A program can make intelligent decisions only if it has knowledge and can attach meaning to data.

There are several advantages to having a minimal I/O interface, so this paper is not proposing a change to the interface. However, throwing up one's hands and limiting storage software to basic functions is not the only alternative. It is possible to extract useful information about file accesses and traffic flow by analyzing the I/O workload. The programs analyzing the I/O workload need not be computationally intensive or invasive. This code could be inserted into standard storage algorithms. In fact, workload information could be extracted without changing standard techniques, and by merely viewing these techniques from a different perspective.

In this paper, we show how a standard storage algorithm, the sequential prefetching technique, could be used to extract useful information from the I/O workload. Sequential prefetching is a caching algorithm that loads data contiguous to I/O read request data into the cache. Most disk arrays implement sequential prefetching techniques. The technique assumes that on-demand requests for the prefetched data will eventually arrive and hit in the cache thereby speeding up access time. Since a sequential prefetching technique identifies sequential locality in the workload, this technique could be used to attach meaning to the I/O workload and storage data. Without explicit information about files and applications being passed down to storage, it is impossible to completely demystify the I/O workload. However, prefetching techniques can be used to bring some order and clarity to the workload.

The paper demonstrates how knowledge of data could be used to implement efficient, adaptive sequential prefetch techniques. First, a standard cache sizing module that assumes no knowledge of data is presented. The sizing module sets the prefetch cache size so that the technique achieves its maximum hit rate for the workload. The paper then analyzes the actions of the prefetching technique and attaches meaning to the data stored in the prefetch cache. Based on this understanding of the data stored in the prefetch cache, the paper proposes an improved prefetch sizing technique that adapts to the workload and the technique.

The **goal** of this paper is to show that a standard algorithm like sequential prefetching could be used not just to increase the hit rate of the cache but also learn about the I/O workload. This information could be used to tag and group storage data, thereby adding meaning to storage data. While this paper only evaluates the workload information that can be extracted by sequential prefetching techniques, a similar approach could be used with other storage techniques. Knowledge about the workload and storage data could lead to storage software that utilizes the power of storage hardware.

The paper is organized as follows: Section 2 explains the I/O workload from the viewpoint of a file system and a storage system. A goal of the paper is to bring the 2 viewpoints closer. We start by re-examining the storage workload and showing the sequentiality in the storage workload. In Section 3, we show how this sequentiality impacts on the hit rate that can be achieved by standard storage prefetch techniques. In order to achieve this hit rate, sequential prefetched blocks must stay in the cache until the on-demand requests for the blocks arrive. We present a "black box" technique for sizing the cache in Section 4. The experimental evaluation demonstrate that the sizing module implicitly gains knowledge of the number of sequential and random file accesses submitted by the file system. In Section 5, we analyze the actions of the sizing technique and gain an understanding of the prefetched data. Based on this "gray box" understanding of the storage workload, in Section 6 we propose a new sizing technique that adapts to the workload.

2 File vs Storage Workload

Sequential prefetching techniques are implemented in both the file system layer and the storage layer. The file system prefetch techniques are more sophisticated than the storage prefetch techniques since the file system can attach meaning to its workload. The workload seen by a disk array and a file system is essentially the same as regards the contents. The difference between the two workloads is the information attached to the workload. This information allows a file system to see file requests while a disk array sees block requests. Here, we examine the workload at both levels.

A disk array's workload consists of read and write requests submitted by file systems. A read request consists of the starting block number to be read, followed by the number of blocks to be read. For notational simplicity, we assume that each read request is for a single block. Suppose the workload of a disk array is observed. Let t_n represent the time instant at which the n^{th} read request arrives. The n^{th} read request submitted to the disk array is described by:

$$\mathsf{X_n} = \mathsf{i};\ \ \mathsf{i} \in \{1, 2, 3, \cdots, \mathsf{MaxBlocks}\}$$

where i is the block number to be accessed, and $\mathsf{MaxBlocks}$ is the maximum number of blocks in the disk array. The time instant t_n at which the read request $\mathsf{X_n}$ arrives at the disk array is implicitly defined in the notation. If a time instant is defined to be smallest, indivisible interval of time, then only 1 read request can arrive at a time instant, and $\mathsf{t}_n < \mathsf{t}_{n+1}\ \forall n$. The read workload is given by the sequence:

$$\mathcal{X} = < \mathsf{X}_1, \mathsf{X}_2, \mathsf{X}_3, \cdots, \mathsf{X_N} >$$

where N is the number of read requests that arrive during the observation period. The disk array workload appears as a single sequence of seemingly random blocks.

Example 1 *An example of a disk array's I/O workload:* $\mathcal{X}$ *= < 1, 51, 99, 151, 89, 2, 3, 152, 999, 52, 4, 5, 251, 799, 53, 6, 351, 299, 199, 899, 7, 54, 699, 599, 252, 499, 253, 3999, 352, 353, 451, 399, 8999, 7999, 6999, 254, 452, 8, 453, 501, 5999, 4999, 502 >*

Here, $\mathsf{X}_1 = 1, \mathsf{X}_2 = 51, \mathsf{X}_3 = 99, \cdots, \mathsf{X}_{42} = 4999, \mathsf{X}_{43} = 502$. X_1 *arrives at* t_1 *and* X_{43} *arrives at* t_{43}.

A file system's workload consists of read and write requests submitted by processes. Each request is for a particular file's data, and the file system has knowledge of the application and file relating to a request. A read request consists of a file id, the starting block address, and the number of blocks to be read. Let each read request be for a single block. Unlike the storage system that sees a single sequence of block numbers, a file system sees various sequences of read requests, where each sequence relates to a file.

Consider the disk array workload of Example 1. The disk array sees a single sequence of random blocks. The file system may have seen the following workload:

File fid1: $< 1, 2, 3, 4, 5, 6, 7, 8, 501, 502 >$ at time instants t_1, t_6, t_7, t_{11}, t_{12}, t_{16}, t_{21}, t_{38}, t_{40}, t_{43};

File fid2: $< 51, 99, 89, 52, 53, 54, 3999, 8999, 5999 >$ at time instants t_2, t_3, t_5, t_{10}, t_{15}, t_{22}, t_{28}, t_{33}, t_{41};

File fid3: $< 151, 152, 999, 251, 252, 253, 254 >$ at time instants t_4, t_8, t_9, t_{13}, t_{25}, t_{27}, t_{36};

File fid4: $< 351, 352, 353 >$ at time instants t_{17}, t_{29}, t_{30};

File fid5: $< 799, 299, 199, 899, 699, 599, 499, 451, 399, 7999, 6999, 452, 453, 4999 >$ at time instants t_{14}, t_{19}, t_{20}, t_{23}, t_{24}, t_{26}, t_{31}, t_{32}, t_{34}, t_{35}, t_{37}, t_{42};

It is clear that file fid4 is accessed sequentially. However, it is possible that all the files are being accessed sequentially, but the files are fragmented on the disks. For example, consider file fid3. The physical blocks $151, 152, 999, 251, 252, 253, 254$ may correspond to logical file system blocks $10, 11, 12, 13, 14, 15, 16$. Similarly, files fid1, fid2 and fid5 could all be accessed sequentially, but their read sequences look random as a result of file fragmentation. The file system knows the logical to physical mapping, so a prefetching technique at the file level could use this knowledge to prefetch blocks accordingly.

The above example shows how the same workload looks very different at the file system and storage levels. The difference lies in the meaning attached to the data. It is knowledge and subsequent understanding of data, not superior hardware, that allows file prefetching techniques to be far more sophisticated than storage prefetching techniques.

2.1 Storage workload re-examined

At first glance, the storage workload shown in Example 1 looks like a random sequence of block numbers with no discernible pattern. A more careful examination reveals the following sequential block accesses:

Stream 1: $1, 2, 3, 4, 5, 6, 7, 8$ at time instants t_1, t_6, t_7, t_{11}, t_{12}, t_{16}, t_{21}, t_{38};

Stream 2: $51, 52, 53, 54$ at time instants t_2, t_{10}, t_{15}, t_{22};

Stream 3: $151, 152$ at time instants t_4, t_8;

Stream 4: $251, 252, 253, 254$ at time instants t_{13}, t_{25}, t_{27}, t_{36};

Stream 5: $351, 352, 353$ at time instants t_{17}, t_{29}, t_{30};

Stream 6: $451, 452, 453$ at time instants t_{31}, t_{37}, t_{39};

Stream 7: $501, 502$ at time instants t_{40}, t_{43}.

The sequential block accesses are referred to as **streams**. The streams are numbered in increasing order based on the time at which the first request from the stream is issued. A request/block belonging to one of these streams is referred to as a **sequential** or **stream** request/block. During the observation period (t_1 to t_{43}), there are several lone block accesses that are not part of any stream. We refer to them as **random** requests/blocks. In Example 1, there are a total of 17 random requests for blocks: 99, 89, 999, 899, 799, 299, 199, 699, 599, 499, 399, 299, 8999, 7999, 6999, 5999, 4999. These requests may be random file accesses submitted by file systems or sequential file access where the file is fragmented. A random block could also be part of a stream where the request for a contiguous block is issued outside the observation period. For example, consider request for block 99 at time t_3: a request for block 98 may have been issued prior to the observation period and a request for block 99 may be issued after the observation period. The only sequentially prefetched blocks that will receive a hit during the observation period are the stream blocks. Without knowledge of files and the logical to physical file data mapping, storage sequential prefetch techniques can only target streams in the workload.

The **interleaved multi-stream** workload of Example 1 is typical of workloads submitted to disk arrays. A disk array gets I/O requests from various applications and these requests get interleaved. As a result, two requests from one application may not arrive during contiguous arrival time instants at the disk array. So, even if files are stored contiguously and each application accesses its files sequentially, the disk array workload appears random. The **task** of a sequential prefetching technique is to prefetch requests from streams and to keep them in the cache until the on-demand request arrives. Therefore, a sequential prefetching technique would identify streams in the workload. We show that without any knowledge of files or streams, a sequential prefetching technique can identify the number of streams and random accesses in the workload. In fact, it is even possible to extract the number of file accesses in the workload. Thus, sequential prefetching techniques can shed some light on the I/O workload.

3 Sequential Prefetch Techniques

The goal is to use sequential prefetching techniques to add information to the storage workload. A sequential prefetching technique has to deal with 3 issues: when to prefetch data, how much data to prefetch, and what cache replacement policy to use when the cache is full. Sequential prefetching techniques can be divided into three categories based on the first issue, namely, when to prefetch data [18]. The *Prefetch Always (PA)* technique prefetches data contiguous to every on-demand request. The *Prefetch on Miss (PoM)* technique prefetches data contiguous to every missed on-demand request. The *Prefetch on Hit (PoH)* technique prefetches data contiguous to every hit on-demand request. Thus, all 3 categories of prefetching techniques initiate prefetch when on-demand requests arrive. Most sequential prefetching techniques are variations of the above techniques. For example, a variation of PA is a technique that prefetches only if the disk is idle when an on-demand request arrives.

We examine the hit rate that can be achieved by each of these prefetching techniques. The maximum prefetch hit rate that can be achieved for a given workload depends on the number of streams and stream blocks in the workload. Let s refer to a sequential request/block and r refer to a random request/block.

N: number of requests that arrive during the observation period.

$\mathsf{N_s}$: number of sequential requests.

$\mathsf{N_r}$: number of random requests.

M: total number of streams that are observed in a workload.

The first block of a stream will not be prefetched by any sequential prefetching technique. Therefore,

$\mathsf{N_h = N_s - M}$: the maximum number of prefetched blocks that could receive hits.

$\mathsf{H = \frac{N_h}{N}}$: the **maximum prefetch hit rate for a workload** with N requests.

Since prefetching techniques prefetch different blocks, not all prefetching techniques can achieve this maximum hit rate. Before listing the hit rate achievable by each technique, we state two assumptions about prefetched blocks from streams: 1) each prefetched block is loaded in the cache instantaneously (before its on-demand request arrives); and 2) each prefetched block remains in the cache until its on-demand request arrives (*i.e.,* , no prefetched block is pre-evicted). Without these assumptions holding true, no prefetching technique can attain its theoretical maximum hit rate. In reality, for asynchronous I/O requests, it is not possible to guarantee Assumption 1. Assumption 2 can be satisfied if the cache is large enough to hold the sequential prefetched blocks until on-demand requests for the data arrive.

The maximum hit rate computation below is based on the following cache setup. The prefetch cache and the on-demand read cache are maintained as separate logical units. When an on-demand request hits in the prefetch cache, the hit block is moved from the prefetch cache into the on-demand cache for future re-reference hits. It is assumed that each on-demand request is for one block and that only one block is prefetched each time a prefetch is initiated.

<u>PA</u>: The PA technique initiates prefetch on arrival of every on-demand request, so all stream blocks, except for the first block of each stream, are prefetched. Therefore, the maximum hit rate for PA is equal to the maximum hit rate for the workload.

$\mathsf{H_{PA} = H}$

<u>PoH</u>: When an on-demand request for block i hits in the prefetch cache, the PoH technique submits a prefetch request for block (i+1). When an on-demand request for block i misses in the cache, PoH looks for block (i-1) in the on-demand cache. If block (i-1) is found in the on-demand cache, then PoH assumes that a sequential stream has started and submits a read request for block i and a prefetch request for block (i+1) to the disks. Block i is stored in the on-demand cache, while prefetched block (i+1) is stored in the prefetch cache. Thus, PoH prefetches all blocks of a sequential stream except for the first two blocks, so the maximum hit rate is given by:

$\mathsf{H_{PoH} = \frac{N_s - 2\times M}{N} = H - \frac{M}{N}}$

<u>PoM</u>: When an on-demand request for block i hits in the prefetch cache, PoM services the request from the prefetch cache and moves block i to the on-demand cache. When an on-demand request for block i misses in the cache, PoM submits a request for block i and a prefetch request for block (i+1) to the disks. Thus, for streams with even number of requests, PoM prefetches half the blocks, while for streams with odd number of requests, PoM prefetches the *floor* of half the blocks. Let $\mathsf{M_{ODD}}$ represent number of streams with odd number of requests.

$\mathsf{H_{PoM} = \frac{N_s - M_{ODD}}{2\times N} \geq \frac{H}{2}}$

Implication of hit rate: A technique such as PA can achieve the maximum possible hit rate for a workload. The fact that the prefetch techniques are able to get hits implies that these techniques implicitly view the storage workload as a multiple interleaved stream workload. Therefore, even though the workload is a single sequence, the sequential prefetch techniques split the workload into multiple streams. The input to a storage sequential prefetch technique is a single stream, but the output from the sequential prefetch technique is multiple streams and random blocks. Similar to a prism that splits white light into its component colors, a sequential prefetch technique divides the single sequence I/O workload into its component streams and random blocks. We analyze the degree by which each of the standard prefetch techniques divides the I/O workload.

4 Black Box Sizing

In theory, a scheme such as PA can achieve the maximum possible sequential prefetch hit rate for a workload. In practice, it may not be possible to achieve this hit rate due to 2 issues. First, some of the prefetched blocks may not get loaded in the cache before their on-demand requests arrive. Second, if the prefetched blocks are loaded in the cache, then the blocks must stay in the cache until their on-demand requests arrive. In order to use the prefetching techniques to extract knowledge of streams and random blocks, it is necessary to show that storage prefetching techniques are capable of achieving their maximum hit rate. Ensuring that prefetched blocks arrive before their on-demand requests is a timing issue. In our experimental evaluation we try to address this problem by ensuring that request arrival rate is less than disk service rate. Ensuring that prefetched blocks stay in the cache until their on-demand requests arrive depends on the cache replacement policy and the cache size.

The cache replacement policy determines which block is to be evicted when the cache is full. With on-demand blocks being moved out of the prefetch cache, the focus of a replacement technique is the storing of prefetched blocks until on-demand requests for the prefetched blocks arrive. Storage systems are not provided a priori information about the workload, so a storage prefetch technique cannot infer when a stream will start and when it will end. Consequently, a prefetch technique has no idea when, or even if, an on-demand request for a prefetched block will arrive. With this uncertainty about the workload, the goal of a prefetch replacement technique is to hold on to unaccessed prefetched blocks as long as possible. Every hit block is removed from the prefetch cache, so the only difference between two cached blocks is the order in which the blocks were prefetched into the cache. Therefore, the **First In First Out** (FIFO) replacement scheme is a good choice for a disk array's prefetch cache replacement scheme. When the cache is full, the block at the FIFO head is evicted and the free cache line is moved to the FIFO tail. When there is a hit in the prefetch cache, the hit block is moved to the on-demand cache and the free cache line is moved to FIFO tail. A newly prefetched block is inserted into the tail of the FIFO queue.

Once the cache replacement policy is fixed, the cache size is the factor that determines the hit rate for a given workload and prefetch technique. A sizing module that dynamically determines the prefetch cache size based on the workload is required. We experimentally determine the effectiveness of the sizing module. The prefetch cache setting used in our experimental evaluation is as follows: 1) each on-demand request is for a single block; 2) each prefetch request is for a single block; 3) a prefetched block that receives a hit is moved out of the prefetch cache immediately; and 4) the replacement technique is FIFO.

4.1 Online sizing module

The **size**, C, of a cache refers to the number of cache lines, where each line can store exactly 1 block. Upon arrival of each I/O request, the online sizing module decides 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 streams remain in the cache until the on-demand requests for the prefetched blocks arrive. The details of such an online sizing scheme are listed in the pseudo-code of Scheme 1. The scheme is based on intuitive reasoning and assumes no knowledge of streams and random blocks. Hence, this scheme is representative of storage algorithms and treats the workload like a black box.

Scheme 1 ONLINE PREFETCH CACHE SIZING

```
noEvictionEndHits ← true; noIncr ← true
for every request $req$ do
    if $req$ is a prefetch cache miss then
        if $req$ is a non-rereference hit in the on-demand cache then
            Increment prefetch cache size by one line
            noIncr ← false
        end if
    else if $req$ is hit near the eviction end then
        noEvictionEndHits ← false
    end if
    reqCount++
    if reqCount == monitoringPeriod then
        if noEvictionEndHits and noIncr then
            Decrement cache size by one line
            Move evicted request into the on-demand cache
        end if
        reqCount ← 0
        noEvictionEndHits ← true; noIncr ← true
    end if
end for
```

The sizing scheme has to determine if the cache is large enough to hold prefetched blocks from streams 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. The scheme monitors the eviction end of the cache for a sufficiently long period (pseudo-code lines 12-16). The monitoring period is set to the sum of current size of the prefetch cache and the size of the on-demand cache. 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.

4.2 Simulation results

We check whether this intuitive sizing scheme is able to set the cache size so that the prefetching schemes can achieve their maximum hit rate. The performance of the proposed sizing scheme is validated through simulations. We ran the simulations using the CMU Disksim [7] simulator. The simulator is used in slave mode by a caching and sizing module that implemented the PoM, PoH, POpt, and PA prefetching schemes and the online sizing scheme. The POpt technique is an off-line prefetching technique that has complete knowledge of the input workload. The simulations are carried out using synthetically generated SPC-2-like read workloads [1] and other workloads. We test the sizing scheme both under uniform and nonuniform interleaving of file access sequences as well as under static and dynamic workloads.

The purpose of these experiments is to check whether the cache size is appropriate for the workload and prefetch technique. We know that the cache size is large enough if the cache get the maximum hit rate for the workload and technique. Several file sequences are submitted to the storage device. The **sequentiality** of a file sequence is the probability that the next request generated is contiguous to the last request generated. Referring to the workload presented in Section 2, file sequences fid4 has sequentiality 1, fid5 has sequentiality 0, while the rest are partly sequential. In all the experiments, we present the size of the cache and the hit rate achieved. We also plot the maximum theoretical hit rate that can be achieved by the technique if the cache were large enough.

Figure 1 plots the experimental runs when the streams are interleaved uniformly. The top graph in Figure 1 shows the prefetch cache size set by the online sizing scheme when it is coupled with each of the three prefetching schemes. 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). that prefetching scheme.

Figure 2 plots hit rates and cache sizes when the streams are not uniformly interleaved Figure 3 depicts the performance when a dynamic workload is used. The number of active file sequences is allowed to vary arbitrarily. In each graph, we plot the simulation time on the X axis and the request addresses in the workload on the left Y axis. Thus, a request at time t for address a is plotted as a point at (t, a). In each graph, on the right Y axis, we plot the instantaneous prefetch cache size maintained by the online sizing scheme coupled with each of the three prefetching schemes and the instantaneous optimal prefetch cache size. The simulations show that the sizing module is able to set the cache to a large enough size so that close to the maximum hit rate is achieved.

5 Black to Gray: Analysis

The same sizing module is used for all the prefetching techniques. The experimental evaluation shows that it is possible for an on-line technique to size a prefetch cache using neither knowledge of the data nor actions of the prefetching technique. Here, we evaluate the actions of the sizing module and the degree by which the sizing module identifies the storage workload. First, we present the workload data from the viewpoint of the sizing module. The workload of Example 1 is used for explaining the terminology and notation.

The example workload has multiple streams that start up and end during various time instants (M=7). A stream, i, is said to be **active** at time t if the last on-demand request issued for this stream arrived at time $\mathtt{t}^- \leq \mathtt{t}$, and the next on-demand request for this stream will arrive at time $\mathtt{t}^+ > \mathtt{t}$ during the observation period. The times, $\mathtt{t}^-$ and $\mathtt{t}^+$ are referred to as the left and right **bookends**

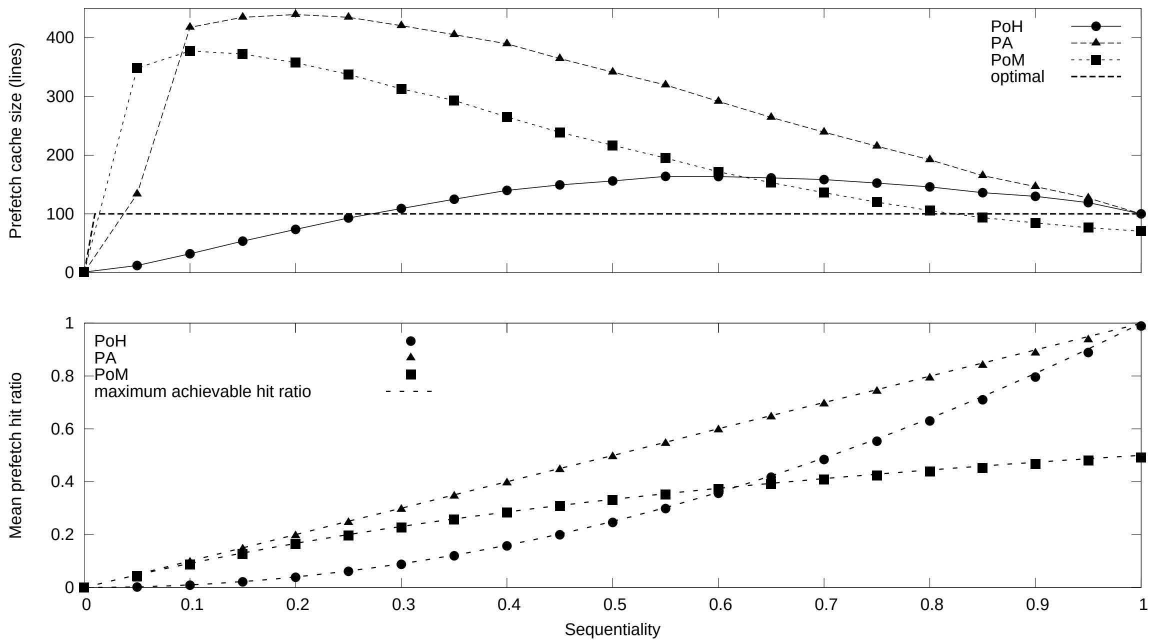


Figure 1: **Uniform interleaving:** The prefetch hit ratio obtained as a function of workload sequentiality with the online prefetch cache sizing scheme listed in Pseudo-code 1. The workload contains 100 file sequences of identical sequentiality (X axis).

for stream i. The requests that arrive at bookend times are referred to as the left and right bookend requests for stream i at time t.

$\mathsf{lb}_i(\mathsf{t})$: left bookend time for stream i active at time t.
$\mathsf{rb}_i(\mathsf{t})$: right bookend time for stream i active at time t.
$\mathcal{AS}(\mathsf{t})$: the set of streams active at time t.

At time t, a sizing module must ensure that the block prefetched by the arrival of the left bookend request of each active stream remains in the cache until the right bookend request arrives. For example, at time instant t_{27}, $\mathcal{AS}(\mathsf{t}_{27}) = \{1,4,5\}$. The left and right bookend times for the streams are $\mathsf{lb}_1 = \mathsf{t}_{21}, \mathsf{rb}_1 = \mathsf{t}_{38}$, $\mathsf{lb}_4 = \mathsf{t}_{27}, \mathsf{rb}_4 = \mathsf{t}_{36}$, and $\mathsf{lb}_5 = \mathsf{t}_{17}, \mathsf{rb}_5 = \mathsf{t}_{29}$. Consider stream 1. At arrival time t_{21}, request for block 7 arrives and may trigger a prefetch of block 8. The sizing module must ensure that the cache is large enough so that prefetched block 8 remains in the cache until right bookend time $\mathsf{rb}_1 = \mathsf{t}_{38}$.

$\mathcal{LB}(\mathsf{t}) = \{\mathsf{t}\} \bigcup \{\mathsf{lb}_i(\mathsf{t}) | i \in \mathcal{AS}(\mathsf{t})\}$: the set of left bookend times of streams active at time t. Note that $\mathsf{t} \in \mathcal{LB}(\mathsf{t})$, so this set is not empty even if there are no active streams at time t.

$\mathsf{lb}_{(i)}(\mathsf{t})$: i^{th} smallest element of the set $\mathcal{LB}(\mathsf{t})$. Note that t is the largest element of $\mathcal{LB}(\mathsf{t})$.

At time instant t_{27}, $\mathcal{LB} = \{\mathsf{t}_{21}, \mathsf{t}_{27}, \mathsf{t}_{17}\}$; $\mathsf{lb}_{(1)} = \mathsf{t}_{17}$, $\mathsf{lb}_{(2)} = \mathsf{t}_{21}$ and $\mathsf{lb}_{(3)} = \mathsf{t}_{27}$. At time t_{27}, the stream that submitted the oldest left bookend request is Stream 5 at t_{17}.

A stream, i, is said to be **old** at time t if the last request for this stream arrived at time $\mathsf{t}^- < \mathsf{t}$. A stream, i, is said to be **new** at time t if the first request for this stream will arrive at time $\mathsf{t}+ > \mathsf{t}$.

$\mathcal{OS}(\mathsf{t})$: the set of streams that are old at time t.
$\mathcal{NS}(\mathsf{t})$: the set of streams that are new at time t.

At time instant t_{27}, $\mathcal{OS}(\mathsf{t}_{27}) = \{2,3\}$ and $\mathcal{NS}(\mathsf{t}_{27}) = \{6,7\}$. From a sizing module's perspective, a new cache line may have to be added when a new stream starts and a cache line could be removed when a stream ends.

We follow the following notational convention: In general, as, os, ns refer to active stream, old stream and new stream, respectively. A subscript adds additional meaning to the notation - for example N, $\mathsf{N_s}$, $\mathsf{N_r}$ mentioned in Section 3. Also, (t) indicates at time t, and $(\mathsf{t}_i, \mathsf{t}_j)$ indicates between and including times t_i and t_j. For example,

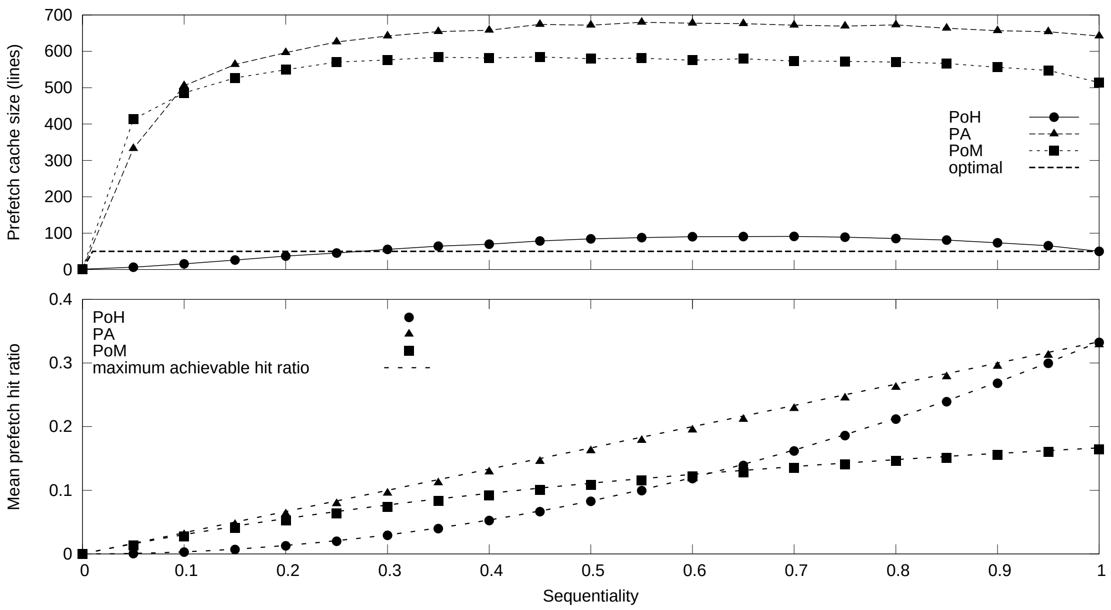


Figure 2: **Non-uniform interleaving:** The prefetch hit ratio obtained as a function of workload sequentiality with the online prefetch cache sizing scheme listed in Pseudocode 1 under a nonuniform workload. The workload contains 50 completely random and 50 sequences of arbitrary (X axis) sequentiality. The arrival rate of the random file sequences is twice that of the partly sequential sequences.

$\mathsf{M_{as}}(\mathsf{t})$: number of streams that are active at time t (so $\mathsf{M_{as}}(\mathsf{t}_{25}) = 3$).

$\mathsf{M_{as}}(\mathsf{t}_i, \mathsf{t}_j)$: number of streams that are active during the time period $[\mathsf{t}_i, \mathsf{t}_j]$ where the left bookend time is $\leq \mathsf{t}_i$ and the right bookend time is $> \mathsf{t}_j$.
The t is dropped when the meaning is clear from the context.

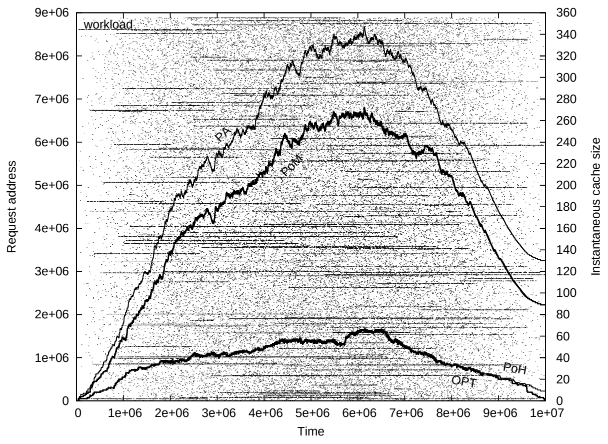
5.1 Minimum cache size

The goal of the sizing module is to keep the cache to a minimum size while achieving the maximum hit rate. Our experimental evaluation shows that the on-line sizing module is able to achieve close to the maximum hit rate. However, it cannot be determined if this hit rate is achieved with the minimum cache size for the technique. In order to determine if the size achieved by the black box technique is the minimum size, we analyze the actions of an off-line sizing technique. The off-line sizing technique maintains the smallest cache size for the workload and technique. Define the **minimum** cache size at time t as follows:

$\mathsf{minC_{Ptech}}(\mathsf{t})$: if $\mathsf{C}(\mathsf{t}) < \mathsf{minC}(\mathsf{t})$, then $\exists\ s$ in the prefetch cache at time t^-, such that s will be pre-evicted from the cache at time t.

At each arrival time instant t, the sizing technique ensures that the cache is at the minimum size for the technique. Initially, $\mathsf{minC_{Ptech}}(\mathsf{t}_0) = 0$. If a prefetch is initiated, then the new block has to be loaded into the cache regardless of whether the block is sequential or random. The minimum size ensures that the eviction end of the FIFO queue always has a prefetched block relating to an active stream. Therefore, every time a prefetch is initiated the cache size must be incremented. Every time there is a hit in the cache, the technique must determine if the cache size should remain unchanged or be decremented. If the hit cache line is at the FIFO head (eviction end), then the cache size may get decremented by more than 1 line (explained later in this section).

$\underline{\mathsf{POpt}}$: Instead of starting with one of the 3 standard prefetching techniques, we start by evaluating the cache size for an optimum off-line sequential prefetching technique. The optimum technique, POpt, knows when streams begin and end and therefore, POpt only prefetches stream blocks. When a stream starts, POpt increments the cache size by 1 line and prefetches the



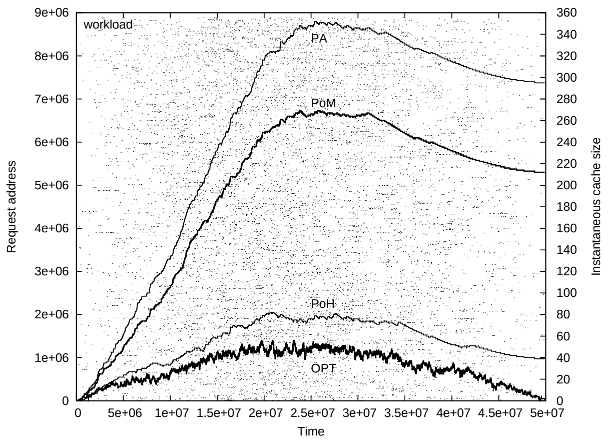


Figure 3: **Dynamic workload:** When the experiment begins, the file sequences start up gradually. As the experiment draws to a close, the file sequences taper off. *Left:* There are a total of 150 completely sequential sequences open and close dynamically while a background of 150 random sequences persists. *Right:* A total of 150 sequences of 80% sequentiality open and close dynamically while a background of 150 random sequences persists.

stream's blocks into this line. When a stream ends, and the last block of the sequential stream is moved into the on-demand cache after a prefetch cache hit, POpt decrements the cache size by 1 line. Let $\mathsf{minC}_{\mathsf{POpt}}(\mathsf{t}_{n-1})$ represent the minimum cache size at time t_{n-1}. Due to spacing constraints, we use the symbol "$= \mathsf{t}_{n-}$" to represent $\mathsf{minC}_{\mathsf{PA}}(\mathsf{t}_{n-1})$ in the equation below. When request $\mathsf{X_n}$ arrives at t:

$$\mathsf{minC}_{\mathsf{POpt}}(\mathsf{t}_n) = \mathsf{t}_{n-}\begin{cases} +0 & \text{if } \mathsf{X_n} \text{ is a random request;} \\ +1 & \text{if } \mathsf{X_n} \text{ is the start of a stream;} \\ -1 & \text{if } \mathsf{X_n} \text{ is the end of a stream;} \\ +0 & \text{otherwise.} \end{cases}$$

The last case is when $\mathsf{X_n}$ is a sequential request that is neither at the start or the end of a sequential stream.

PA: Unlike the POpt technique that only prefetches stream requests, the PA technique prefetches every time an on-demand request arrives. A prefetch on a miss requires a new cache line, so the minimum cache size would have to be incremented by one line. A prefetch on a hit does not require a new cache line, since the hit block is moved from the prefetch cache, and the newly prefetched block can be inserted into this free cache line at the FIFO tail. It "seems" to follow that the minimum cache size would be unchanged. If the hit block is in the middle of the FIFO queue, then this reasoning is correct and the minimum cache size remains unchanged. However, if the hit block is at the eviction end (*i.e.*, head) of the FIFO queue, then this reasoning is incorrect. In the FIFO queue, immediately following the FIFO head, there may be several non-sequential prefetched blocks pertaining to prefetches initiated for random requests and last requests from streams. These blocks can be safely ejected from the cache, so the minimum cache size can be reduced.

The minimum cache size always ensures that a block at the FIFO head is a sequential block. If the prefetched block pertaining to request $\mathsf{X_n}$ is at the FIFO head, then this block was prefetched at time $\mathsf{lb}_{(1)}(\mathsf{t})$ (*i.e.*, the oldest prefetched block pertaining to an active stream). All random and old stream blocks prefetched after $\mathsf{lb}_{(1)}(\mathsf{t})$ and before the arrival of another stream request at $\mathsf{lb}_{(2)}(\mathsf{t})$ can be ejected from the cache. (Note that if set $\mathcal{LB}$ only contains t, then let $\mathsf{lb}_{(1)} = \mathsf{lb}_{(2)} = \mathsf{t}$.) That is, all random and old stream blocks prefetched after $\mathsf{lb}_{(1)}$ and before $\mathsf{lb}_{(2)}$ can be ejected.

$$\mathsf{minC}_{\mathsf{PA}}(\mathsf{t}_n) = \mathsf{t}_{n-}\begin{cases} +1 \text{ if } \mathsf{X_n} \text{ is a random request;} \\ +1 \text{ if } \mathsf{X_n} \text{ is the start of a stream;} \\ +0 \text{ if hit for } \mathsf{X_n} \text{ is not at FIFO head;} \\ -\mathsf{N_r}(\mathsf{lb}_{(1)},\mathsf{lb}_{(2)}) - \mathsf{M_{os}}(\mathsf{lb}_{(1)},\mathsf{lb}_{(2)}) \text{ otherwise.} \end{cases}$$

The computation above assumes that the first request in the workload is a stream request. Without this assumption, an extra line would have to be added just to take care of the initial condition of having a random block at the FIFO head.

PoH: Unlike the PA technique, the PoH technique prefetches only after identifying streams in the workload. When an on-demand request for block i misses in the cache, and block (i-1) is found in the on-demand cache, a stream is identified. Therefore, PoH starts prefetching only upon arrival of the second on-demand request in a stream. Moreover, random blocks in the workload do not impact on the size of the prefetch cache. Unlike the POpt technique, PoH does not know when streams end, and prefetches a block past the end of each stream. This block will remain in the prefetch cache until it is evicted. Therefore, the prefetch cache stores **identified** stream blocks from active and old streams. To **specify an identified stream**, a subscript of ID is used. For example, $\mathcal{AS}_{\mathsf{ID}}(\mathsf{t})$ indicates the set of identified active streams - streams where the second request was submitted on or prior to time t.

$$\mathsf{minC}_{\mathsf{PoH}}(\mathsf{t}_n) = \mathsf{t}_{n-} \begin{cases} +0 \text{ if } \mathsf{X_n} \text{ random or start of stream;} \\ +1 \text{ if } \mathsf{X_n} \text{ request 2 from a stream;} \\ +0 \text{ if hit for } \mathsf{X_n} \text{ not at FIFO head;} \\ -\mathsf{M}_{\mathsf{IDos}}(\mathsf{lb}_{(1)}, \mathsf{lb}_{(2)}) \text{ otherwise.} \end{cases}$$

The variable $\mathsf{M}_{\mathsf{IDos}}(\mathsf{lb}_{(1)}, \mathsf{lb}_{(2)})$ refers to the number of identified streams that complete between the left bookend initiated prefetched blocks of the two oldest identified active streams at time t_n. These cache lines holding blocks pertaining to old streams are next to the FIFO head and can be removed from the cache when the hit occurs at the FIFO head.

<u>PoM</u>: The PoM technique prefetches each time an on-demand request misses in the cache. Therefore, PoM initiates a prefetch for every random request and for odd numbered requests in each stream (*i.e.,* request 1, 3, 5, ... from stream j). Every hit block is removed from the cache, thereby decrementing the cache size by 1 if the block is not at the FIFO head. If the hit block is at the FIFO head, then all random and old streams with odd number of requests issued that are next to the FIFO head can be removed from the cache. To **specify streams that have issued an odd number of requests**, we add the prefix ODD to the notation. Let $\mathcal{AS}_{\mathsf{ODD}}(\mathsf{t})$ refer to the set of active streams that have issued an odd number of requests by time t. Referring to Example 1, $\mathcal{AS}_{\mathsf{ODD}}(\mathsf{t}_{27}) = \{1, 4, 5\}$ while $\mathcal{AS}_{\mathsf{ODD}}(\mathsf{t}_{26}) = \{1, 5\}$.

$$\mathsf{minC}_{\mathsf{PoM}}(\mathsf{t}_n) = \mathsf{t}_{n-} \begin{cases} +1 \text{ if } \mathsf{X_n} \text{ random;} \\ +1 \text{ if } \mathsf{X_n} \text{ odd-numbered stream req;} \\ -1 \text{ if hit for } \mathsf{X_n} \text{ not at FIFO head;} \\ -1 - \mathsf{N_r}(\mathsf{lb}_{(1)}\mathsf{lb}_{(2)}) - \mathsf{M}_{\mathsf{ODDos}}(\mathsf{lb}_{(1)}\mathsf{lb}_{(2)}) \end{cases}$$

5.2 Infimum Cache Size

The off-line technique increments the cache whenever a prefetch is initiated. The black box on-line scheme of Section 4.1 does not increment the cache each time a block is prefetched. The on-line scheme only increments the cache when a block is pre-evicted before its on-demand request arrives. Therefore, the on-line scheme is not maintaining the minimum cache size. The on-line scheme tries to keep the cache large enough so that all prefetched blocks from active streams stay in the cache until their on-demand requests arrive. The **infimum** cache size, infC, is defined to be the smallest number of cache lines required to ensure that no prefetched stream block is pre-evicted. Let Ptech refer to the prefetching technique.

$\mathsf{infC}_{\mathsf{Ptech}}(\mathsf{t})$: if $\mathsf{C} < \mathsf{infC}$ at time t, then $\exists\ s$ in the prefetch cache at time t, such that s will be pre-evicted from the cache during the observation period.

The infimum cache size is larger than the minimum cache size since the minimum size at time t only guarantees that no stream prefetched block is pre-evicted at time t, while the infimum size at time t guarantees that a prefetched stream block will remain in the cache until its on-demand request arrives at time t^+. The infimum size ensures that the cache is large enough to handle all prefetch insertions into the cache after a sequential block is prefetched and before its on-demand request arrives. Random prefetched blocks may be evicted from the cache, but the infimum size ensures that all stream prefetched blocks stay in the cache until the on-demand requests arrive. For calculating the infimum size for a technique, it is necessary to compute the number of stream and random blocks that are inserted into the cache.

<u>POpt</u>:
$\mathsf{infC}_{\mathsf{POpt}}(\mathsf{t}) = \mathsf{M_{as}}(\mathsf{t})$
where $\mathsf{M_{as}}$ is the number of active streams at time t.

<u>PA</u>: At time t, a cache line must be reserved for each of the active sequential streams. Therefore, the prefetch cache must be at least $\mathsf{M_{as}}(\mathsf{t})$ lines long. We want to ensure that an active stream's prefetched block stays in the cache until its on-demand request arrives. That is, we want to ensure that the prefetched block triggered by the arrival of the left bookend request of a stream active at time t, stays in the cache until the on-demand right bookend request for this prefetched block arrives. The size must be large enough to handle all random block insertions between the bookend requests, the streams that end after the left book time and t, and the new streams that start after t and the right bookend time. The infimum size is the maximum of the infimum sizes for each of the streams active at time t.
$\mathsf{infC}_{\mathsf{PA}}(\mathsf{t}) = \mathsf{Maximum}\{\mathsf{M_{as}}(\mathsf{t}) + \mathsf{N_r}(\mathsf{lb}_j, \mathsf{rb}_j) + \mathsf{M_{os}}(\mathsf{lb}_j, \mathsf{t}) + \mathsf{M_{ns}}(\mathsf{t}, \mathsf{rb}_j) \mid \forall j \in \mathcal{AS}(\mathsf{t})\}$

PoH: The PoH technique prefetches identified active streams. The cache must be large enough to store blocks from active, old and new identified streams that arrive between the bookend requests of active streams.
$\mathsf{infC}_{\mathsf{PoH}}(\mathsf{t}) = \text{Maximum}\{\mathsf{M}_{\mathsf{IDas}}(\mathsf{t}) + \mathsf{M}_{\mathsf{IDos}}(\mathsf{lb}_j, \mathsf{t}) + \mathsf{M}_{\mathsf{IDns}}(\mathsf{t}, \mathsf{rb}_j) \mid \forall j \in \mathcal{AS}_{\mathsf{ID}}(\mathsf{t})\}$

PoM: With the PA and PoH techniques, a prefetched block's position either remains unchanged or moves toward the FIFO eviction end with each arrival instant. This observation does not hold when the PoM technique is used. A complication with the PoM technique is that a hit does not trigger a prefetch, and therefore, the hit cache line becomes free. If this free cache line remains in the prefetch cache, then the next prefetched block can be inserted into this free cache line. After each hit, there is one fewer prefetched block, so a hit effectively moves all the cached blocks one line away from the FIFO head (eviction end) toward the FIFO tail. Therefore, with PoM, a prefetched block's position could either move toward the FIFO eviction end or move away from the FIFO eviction end with each arrival instant. This oscillation of prefetched blocks with each arrival instant changes the computation of the infimum size for PoM when compared to that of PA and PoH. The infimum cache size at t is the maximum of the minimum cache sizes between the left and right bookend requests of all active streams in $\mathcal{AS}_{\mathsf{ODD}}(\mathsf{t})$. This number is hard to quantify in terms of insertions that occur between bookend requests.

Now, PoM initiates a prefetch for all random requests just like the PA scheme, but only prefetches some of the requests from streams. Therefore, the prefetch cache for PoM at time t is a subset of the prefetch cache for PA at time t and
$\mathsf{infC}_{\mathsf{PoM}}(\mathsf{t}) \leq \mathsf{infC}_{\mathsf{PA}}(\mathsf{t})$

6 Gray Box Scheme

The black box on-line sizing scheme attempts to set the cache to the infimum size for the workload and prefetch scheme. The experimental analysis shows that the sizing scheme is successful enough since a hit rate close to the maximum hit rate is achieved. It is hard to determine whether the size is smaller or larger than the infimum, but given the black box approach used by the on-line sizing scheme, the performance is pretty good. Here, we propose a prefetching/sizing scheme that uses a gray box approach. We are not presenting the details of the scheme, just suggestions on how better schemes could be developed.

The experimental and theoretical analysis show that the prefetch cache may hold several random blocks when the PA and PoM technique is used. Therefore, when the PA and PoM prefetch techniques are being used, it is not a good idea to use the black box sizing scheme. It is not cost efficient to keep increasing/decreasing the cache size. The analysis shows that if the size of the prefetch cache is C lines, then all active stream blocks that are separated by at most C requests will definitely hit in the prefetch cache. It would be better to set the cache size to some system determined initial size and keep track of where the hits occur in the FIFO queue, the number of hits versus the number of prefetches, etc. Based on the intensity of random blocks in the workload and the overall traffic, there may be periods when it is wiser to turn off prefetching, or switch to a scheme such as PoH.

All the prefetch schemes identify streams in the workload. The PA and PoM schemes identify the streams implicitly when a hit occurs in the prefetch cache. Since PA and PoM prefetch random and stream blocks, it would be prudent for PA and PoM to use stream identification to become more efficient. For example, the degree of prefetching could be increased for the identified streams. Furthermore, the FIFO cache replacement scheme could be supplanted by a replacement scheme such as Second chance [47] that allows identified streams to stay a little longer in the cache when they reach the eviction end.

The analysis suggests a better sizing scheme for the PoH technique. The cache contains data from identified active streams and from old streams. When a new stream is identified, it would be better to increase the cache size by a line and load the prefetched block from the newly identified stream into this line. In this case, the FIFO eviction end would eventually have blocks from old streams and would not receive hits. The cache size can be decreased using the black box decreasing technique. The analysis also implies that if the workload consists of a large number of short duration streams and few random blocks, then a scheme such as PA or PoM would be more suitable than PoH. Overall, a prefetch and sizing scheme that uses a gray box approach would make more efficient use of expensive cache lines while reducing the traffic at the disks.

7 Related Research on Prefetching

The memory hierarchy is intrinsic to computer systems, so prefetching techniques have been in use for a long time [2] and can be found at all levels - memory caches hardware based: [4, 10, 11, 12], memory caches compiler based: [5, 21, 35, 37, 38]; memory application programs based: [16, 24, 34, 40, 41, 45, 50]; file system caches [9, 28, 36, 43]; and storage system caches [18, 19, 20, 30, 31, 33, 50, 51].

Storage controller prefetching is initiated by storage software and the prefetched data are stored in the storage caches. The "narrow I/O interface" between the file system and storage system restricts the effectiveness of storage controller prefetching techniques [32]. Single disk prefetching techniques focus on sequential read-ahead prefetching. The cache is divided several segments so that requests from different streams can be prefetched and cached in different segments [23, 52]. However, prefetching is one of the least specified areas of disk system behavior [46].

Even simple access patterns are not be available in disk array workloads because of low re-reference locality [39, 55] and interleaving of the access patterns at the storage caches [31]. Based on the transparency and generality, there are three approaches to getting block correlations, namely, white box, gray box and black box [32]. The white box approach passes the storage front-end semantic information directly and explicitly to the storage system [17]. The gray box approach transmits file system information to the storage system [49]. The black box approach finds block correlations by analyzing I/O request addresses [22, 32].

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 [13, 44, 25] Jelenkovic et al. showed the relationship between the cache miss rate and the cache size for a LRU scheme with statistically dependent request sequences [26, 27]. Singh et al. [48] 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 [8, 14, 56, 15, 53, 6].

There are far fewer papers on prefetch cache sizing. Tse et al. [54] 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. [29] implemented a prefetch cache sizing scheme based on a gradient descent-based greedy algorithm. The SARC technique [19] 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 [31] 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.

8 Conclusions

Disk array systems not only have the capacity to store terabytes of data but also have the computational power to speed data access. The only caveat is that disk arrays do not know what is stored on the disks. That is, disk arrays store files for applications but have no knowledge of the files or the applications. This knowledge is controlled by file systems, and therefore, storage is controlled by file systems. On the flip side, file systems have knowledge of storage data but they see the disk array as a black box and have little knowledge of the physical placement of data. There is an obvious disconnect - file systems know the meaning of data but lack information on where the data are stored on the various disks, while storage systems attach no meaning to data but know specifics of where the data are stored. This disconnect between data placement and data meaning limits the ability of software to use the power of hardware, thereby hurting performance.

This paper tries to address this issue by showing that even without knowledge of files and applications, storage programs implicitly gain knowledge of the data. This knowledge could be used used to tag and identify how often data blocks are used and the data access patterns. This understanding of data usage and an intimate knowledge of array hardware could be used by disk array software to remap data placement to boost performance. Attaching meaning to data can also help replace standard storage software by more efficient software that adapts to the workload.

The paper shows how standard sequential prefetching techniques implicitly gain significant information on data access patterns. Based on this information, we propose an adaptive prefetching and sizing technique. As future work, we plan to develop the adaptive technique. This paper is just a small step toward file aware storage. More work is needed to fully understand how storage techniques can be used to extract meaning of storage data. Attaining knowledge of data is not sufficient since it is important to understand how this knowledge can be used to improve data placement and improve storage software.

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] BAER, J.-L., AND CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91* (1991), pp. 176–186.

[5] BROWN, A. D., MOWRY, T. C., AND KRIEGER, O. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems 19*, 2 (2001), 111–170.

[6] 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.

[7] 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.

[8] 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.

[9] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems 14*, 4 (1996), 311–343.

[10] CHEN, T.-F., AND BAER, J.-L. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V: Proceedings of the 5th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1992), ACM, pp. 51–61.

[11] CHEN, T.-F., AND BAER, J.-L. A performance study of software and hardware data prefetching schemes. In *Proceeding of the 21st Annual International Symposium on Computer Architecture* (1994), pp. 223–232.

[12] CHEN, T.-F., AND BAER, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44*, 5 (1995), 609–623.

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

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

[15] CURCIO, M., LEONARDI, S., AND VITALETTI, A. *Algorithm Engineering and Experiments*. Springer Berlin / Heidelberg, 2002.

[16] CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD* (1993), International Conference on Management of Data archive, pp. 257 – 266.

[17] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A cost-effective, high-bandwidth storage architecture. In *the 8th International Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS)* (1998).

[18] GILL, B. S., AND BATHEN, L. A. D. Optimal multistream sequential prefetching in a shared cache. *ACM Transactions on Storage (TOS) 3* (2007).

[19] 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.

[20] GINDELE, J. D. Buffer block prefetching method. *IBM Tech Disclosure Bull. 20*, 2 (July 1977), 696 – 697.

[21] GORNISH, E. H., GRANSTON, E. D., AND VEIDENBAUM, A. V. Compiler-directed data prefetching in multiprocessors with memory hierarchies. *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News 18*, 3 (1990), 354–368.

[22] Griffioen, J., and Appleton, R. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference* (1994), vol. 1, USENIX Association Berkeley, CA, USA, pp. 197–208.

[23] 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.

[24] Harizopoulos, S., Harizakis, C., and Triantafillou, P. Hierarchical caching and prefetching for high performance continuous media servers with smart disks. *IEEE Concurrency 8*, 3 (2000), 16–22.

[25] 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.

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

[27] 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.

[28] Lei, H., and Duchamp, D. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference* (Anaheim, California, USA, 1997).

[29] 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.

[30] Li, C., Shen, K., and Papathanasiou, A. E. Competitive prefetching for concurrent sequential I/O. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), 189 – 202.

[31] 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.

[32] 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.

[33] Liang, S., Jiang, S., and Zhang, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on* (2007), pp. 64–.

[34] Lipsa, D. R., Rhodes, P. J., Bergeron, R. D., and Sparr, T. M. Spatial prefetching for out-of-core visualization of multidimensional data. In *Conference on Visualization and Data Analysis* (2007).

[35] Luk, C.-K., and Mowry, T. C. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems* (1996), pp. 222–233.

[36] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for UNIX. *Computer Systems 2*, 3 (1984), 181–197.

[37] Mowry, T., and Gupta, A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing 12*, 2 (1991), 87–106.

[38] Mowry, T. C., Lam, M. S., and Gupta, A. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not. 27*, 9 (1992), 62–73.

[39] Muntz, D., and Honeyman, P. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter 1992 Technical Conference* (San Fransisco, CA, USA, 1992), pp. 305–313.

[40] Ng, C.-M., Nguyen, C.-T., Tran, D.-N., Yeow, S.-W., and Tan, T.-S. Prefetching in visual simulation. In *Proceedings on 14th IEEE Visualization Conference (VIS'03)* (2003).

[41] Palmer, M., and Zdonik, S. B. Fido: A cache that learns to fetch. In *In G. M. Lohman, A. Sernadas, and R. Camps, editors, 17th International Conference on Very Large Data Bases* (1991).

[42] Patterson, D. A., Gibson, G., and Katz, R. A case for redundant arrays of inexpensive disks (RAID). *ACM SEGMOD Conference* (1988), 145–185.

[43] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. SOSP Conf.* (December, 1995).

[44] PRZYBYLSKI, S. A., HOROWITZ, M., AND HENNESSY, J. L. Characteristics of performance-optimal multi-level cache hierarchies. In *ISCA* (1989), pp. 114–121.

[45] RHODES, P. J., TANG, X., BERGERON, R. D., AND SPARR, T. M. Iteration aware prefetching for large multidimensional datasets. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management* (2005), pp. 45 – 54.

[46] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994), 17–29.

[47] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating Systems Concepts*. Wiley, July 2008.

[48] 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.

[49] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., AIPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003).

[50] SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Transanctions on Database Systems 3*, 3 (1978), 223–247.

[51] SMITH, A. J. Cache memories. *ACM Computing Surveys 14*, 3 (1982), 473–530.

[52] SOLOVIEV, V. Prefetching in segmented disk cache for multi-disk systems. In *IOPADS ’96: Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems* (1996), ACM Press, pp. 69–82.

[53] TENG, J. Z., AND GUMAER, R. A. Managing IBM database 2 buffers to maximize performance. *IBM System Journal 23*, 2 (1984), 211 – 218.

[54] 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.

[55] 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.

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