PFC: Transparent Optimization of Existing Prefetching Strategies for Multi-level Storage Systems

Zhe Zhang [*] [†] Kyuhyung Lee [*] [‡] Xiaosong Ma [†] [§] Yuanyuan Zhou [‡]

Abstract

The multi-level storage architecture has been widely adopted in servers and data centers. However, while prefetching has been shown as a crucial technique to exploit the sequentiality in accesses common for such systems and hide the increasing relative cost of disk I/O, existing multi-level storage studies have focused mostly on cache replacement strategies. In this paper, we show that prefetching algorithms designed for single-level systems may have their limitations magnified when applied to multi-level systems. Overly conservative prefetching will not be able to effectively use the lower-level cache space, while overly aggressive prefetching will be compounded across levels and generate large amounts of wasted prefetch. We take an innovative approach to this problem: rather than designing a new, multi-level prefetching algorithm, we developed PreFetching-Coordinator (PFC), a hierarchy-aware optimization applicable to any existing prefetching algorithms. PFC does not require any application hints, a priori knowledge on the application access pattern or the native prefetching algorithm, or modification to the I/O interface. Instead, it monitors the upper-level access patterns as well as the lower-level cache status, and dynamically adjusts the aggressiveness of the lower-level prefetching activities.

We evaluated PFC with extensive simulation study using a verified multi-level storage simulator, an accurate disk simulator, and access traces with different access patterns. Our results indicate that PFC dynamically controls lower-level prefetching in reaction to multiple system and workload parameters, improving the overall system performance in 94 out of the 96 test cases. Working with four well-known existing prefetching algorithms adopted in real systems, PFC obtains an improvement of up to 52% to the average request response time, with an average improvement of 16% over all cases.

1 Introduction

Motivation Today's applications, commercial and scientific alike, rely more and more on the ability to store, share, and analyze large amounts of data. Yet with the growing performance gap between I/O systems and processor/memory units, data storage and accesses are inevitably becoming more bottleneck-prone. It has therefore become more critical to efficiently utilize main memories available in the system as buffer caches, through *demand paging* and *prefetching*, two widely adopted techniques.

Meanwhile, as service-based (especially web-based) applications prevail, cache management frequently has to be extended to multiple levels. For example, a web-based data center will have large storage caches equipped at both the front-end web servers and the back-end storage servers (see Figure 1(a) for a sample architecture). How to effectively manage the aggregate cache space and improve the overall system performance has been studied extensively in the recent years. However, existing studies have focused on the multi-level caching problem [9, 21, 41, 42], while to our best knowledge no research has targeted coordinating the prefetching operations in a multi-level storage system. In contrast to the relative lack of studies it receives, prefetching has a crucial role on multi-level server systems. Many service applications hosted by such systems are read-intensive and possess heavily sequential access patterns. Examples include commercial or scientific data queries, web document processing, and multimedia hosting/streaming. Such applications benefit tremendously from file system prefetching.

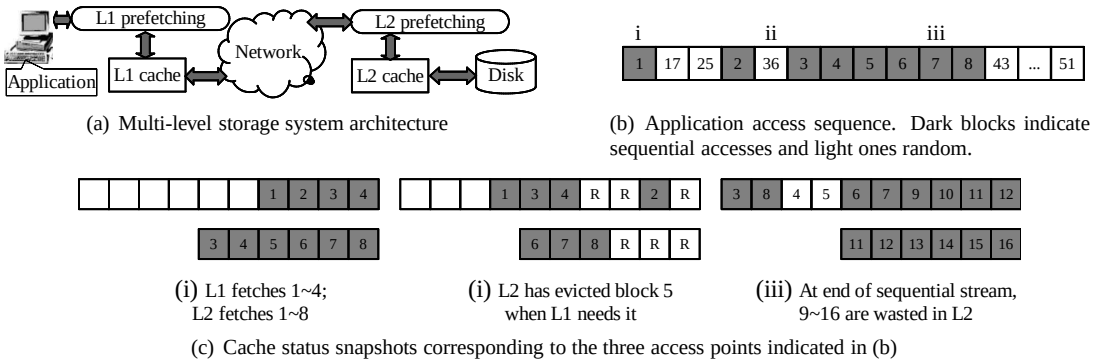
In a multi-level system, prefetching is needed at each level to hide the latency of fetching data blocks from the lower layer. However, if prefetching is carried out independently at each layer, the system will not be able to make a coordinated use of the combined cache space. In particular, when a single-level prefetching algorithm is too conservative or too aggressive for a certain application workload, this mismatch will be magnified when multiple levels

[*] Zhe Zhang and Kyuhyung Lee have contributed equally to this work.

[†] Department of Computer Science, North Carolina State University zzhang3@ncsu.edu, ma@cs.ncsu.edu

[‡] Department of Computer Science, University of Illinois at Urbana-Champaign {kyuhlee,yyzhou}@uiuc.edu

[§] Computer Science and Mathematics Division, Oak Ridge National Laboratory



(a) Multi-level storage system architecture

(b) Application access sequence. Dark blocks indicate sequential accesses and light ones random.

(c) Cache status snapshots corresponding to the three access points indicated in (b)

Figure 1. Sample architecture and behavior of uncoordinated multi-level prefetching

of prefetching are stacked up together.

On one hand, a conservative prefetching scheme, when applied to the lower level, may not be able to effectively use the cache space to hide I/O costs. Other researchers have recently observed that due to the low temporal locality at the lower level caused by higher-level caching, as well as the increasingly bottleneck-prone storage device performance, aggressive lower-level prefetching is especially appealing [28]. The lower level cache space may be better utilized when used as a staging area for higher levels to hide disk I/O costs, than as a backing store for demand paging. Also, aggressive lower-level prefetching helps handle the larger and more bursty requests caused by the batching effect of upper-level prefetching. Conservative prefetching at the upper level may use a small, fixed prefetch depth, or grow the prefetching depth very slowly. This in turn gives the lower level less encouragement for prefetching even with highly sequential accesses. Therefore, a slightly conservative single-level prefetching algorithm may need to be speeded up a lot when applied to multiple levels.

On the other hand, as people realize the growing appeal of prefetching [33], more aggressive prefetching strategies are likely to be adopted at each level. When multiple such uncoordinated software layers are stacked together, there is a compounding effect causing overly aggressive prefetching, which may significantly increase the burden of the storage devices, waste both the cache space and the I/O bandwidth, and degrade the application performance. Known problems associated with aggressive prefetching will be intensified when a system is expanded into multiple layers with uncoordinated prefetching: namely *cache pollution* (too eager prefetching taking space from more useful data) and *prefetch wastage* (prefetched data being evicted before they are used) [19]. Also, aggressive multi-level prefetching may accumulate many prefetched data blocks in the lower level cache even after such blocks have been passed up to the upper level, which is especially undesirable when the access pattern is mostly random or a mixture of sequential and random accesses. Finally, these problems related to aggressive prefetching can further be aggravated when the system employs a n-to-1 or n-to-$m(n > m)$ mapping between the clients and servers, requiring each server's space and bandwidth resources to be split between multiple clients.

Figures 1(b) and 1(c) demonstrate the problems with uncoordinated multi-level prefetching, using a fixed prefetching depth of 4. In this example, the upper level cache is larger than the lower level one. After access point (i) the upper-level will prefetch blocks 2-4, triggering the lower level to prefetch blocks 5-8. With a limited lower-level cache size, prefetched block 5 will be flushed out of cache at point (ii) by the accesses of three random blocks. Therefore when block 5 is needed by the upper level, both levels will suffer a miss. Besides, after both access points (i) and (iii), redundant blocks are cached in both levels, while prefetched data have a lower chance of being requested again, at least at the lower level. In addition, the unnecessary prefetching of blocks 9-12 at the end of the sequential run will be extended to blocks 13-16 at the lower level.

Contributions In this paper, we present a novel approach to the above problems. Rather than designing new, coordinated multi-level prefetching algorithms, we look into how to extend *existing* single-level algorithms to work in a coordinated manner in multi-level systems. This is motivated by the fact that different prefetching algorithms are chosen by real-world systems based on the expected application access pattern, and the complexity of multi-level servers or data centers is growing. The coordination component built with this approach, called PreFetching Coordinator (PFC) will act like an "extension cord" that connects the existing prefetching algorithms at different levels, each working on top of an existing cache management strategy. Further, PFC enables coordinated prefetching across more than two levels, and potentially the stacking of different prefetching algorithms. Finally, PFC does not require any application hints or a-priori knowledge on applications' access pattern.

The main idea behind PFC's operation is to place an

immediate layer of intelligence between the upper- and lower-level strategies for prefetching and cache replacement. By observing the upper-level requests and the lower-level prefetching behavior, PFC detects whether the current prefetching is too aggressive or too conservative, and tries to refrain or boost the lower-level prefetching activity, while at the same time avoiding caching prefetched data redundantly at multiple levels.

We designed a novel PFC algorithm that acts as a middleman between two adjacent levels of caching/prefetching, and is independent of the specific prefetching or replacement algorithms adopted at each level. PFC is adaptive, transparent to applications, and maintains the I/O interface between the multiple levels.

We evaluated PFC with a verified multi-level simulator, along with an accurate disk simulator, using multiple storage system and application access traces. In our simulator we implemented four well known prefetching algorithms used in real systems: P-Block ReadAhead (RA), Linux kernel prefetching, SARC, and AMP. Our experiments show that for all the algorithms, when they are applied to a two-level storage system, the addition of PFC can considerably improve the overall system performance (in terms of average request response time), by up to 52% and on average 16%. In particular, PFC is able to regulate the prefetching aggressiveness and achieve a performance improvement for all types of trace workloads: highly sequential, highly random, and mixed patterns. Besides lowering the request processing time, PFC enhances the overall system resource utilization by reducing redundant caching and controlling wasted prefetch when cache space becomes tight.

2 Background

2.1 Related Work

Prefetching Prefetching techniques for single-level systems have been widely studied [38]. Many prefetching algorithms were proposed to answer the key questions of "what to prefetch" and "when to prefetch". Gill et al. recently gave a quite comprehensive classification of existing prefetching approaches [19]. It indicates that although many sophisticated algorithms have been proposed to perform stride-based [17, 24, 12, 1] or history-based prefetching [27] to "guess" the best blocks to prefetch next, most commercial storage systems adopt simple schemes such as sequential prefetching. The reason is that sequential prefetching is able to provide good long-term prefetching accuracy for diverse workloads, without imposing the cost of extra I/O involved in maintaining and using the access history. While PFC is algorithm-independent, our discussion and evaluation are focused on sequential prefetching.

Although there are a wealth of studies on multi-layer cache management, existing work on multi-layer prefetching is quite limited. Research efforts that we are aware of are on multi-layer hardware prefetching for the CPU caches [6, 16], which use fixed, uncoordinated strategies at different cache levels. The DiskSeen [15] technique exploits knowledge of on-disk data layout to direct efficient file-level prefetching. The most related work to ours is STEP [28], which is motivated by the need to perform aggressive lower-level prefetching. STEP accurately detects sequential access patterns as well as disk thrashing patterns, and makes prefetching decisions accordingly. Like PFC, STEP optimizes the lower-level prefetching behavior with the awareness of upper-level prefetching or caching. However, it promotes aggressive lower-level prefetching, while PFC may moderate the lower-level activity both ways. In addition, STEP is itself a stand-alone lower-level prefetching algorithm, while PFC is a portable, generic optimization that can be applied to any single-level prefetching/caching algorithms. Finally, STEP was shown to improve the multi-level system performance significantly with sequential workloads while having no impact on handling random workloads. In contrast, our results show PFC brings considerable performance gain to both types of workloads.

Some other studies on multi-level systems utilize application hints to direct prefetching [7, 34]. PFC, on the other hand, does not require such hints or modification to the inter-level I/O interfaces.

Space coordination between prefetching and demand paging Besides the problems of "what to prefetch" and "when to prefetch", another important issue in prefetching is how to allocate the shared memory cache space among prefetched and demand paged data. There are a number of previous studies about managing prefetched data in a cache shared with demand paged data [5, 23, 32, 26]. Several other solutions alleviate the problem of *cache pollution* [35, 31] by carefully limiting the space that prefetched data can use. As a recent example, the SARC algorithm [20] uses two separate LRU queues for sequential and random data respectively, and adjusts their sizes according to the access pattern. As we show in this paper, PFC can seamlessly work together with such techniques.

Multi-level cache management There have been many research studies in the contexts of demand paging and general cache management for multi-level storage systems. Previous research in different environments has noticed the weakness of LRU-like algorithms for lower level buffer cache in a hierarchy [14, 30] and pointed out feasible solutions for different systems [4, 40, 43]. This group of work focuses on improving the lower-level caching performance in reaction to the upper-level caching effect.

There has also been work on collaborative caching across multiple layers of storage. Chen et al. categorized existing work on multi-level buffer cache collaboration into two paradigms [8]: *hierarchy-aware caching* [9, 2, 43]

and *aggressively-collaborative caching* [13, 41, 21]. The authors indicate that although aggressively-collaborative caching utilizes the aggregate buffer cache space more sufficiently, hierarchy-aware caching has the advantage of being transparent to the storage client software. Their empirical evaluation based on typical commercial storage system workloads reveals that if local optimizations are properly applied, the performance gain of aggressively-collaborative caching over hierarchy-aware caching is actually very limited. PFC can be viewed as a hierarchy-aware strategy for multi-level prefetching.

One category of collaborative cache management particularly related to our approach is *exclusive caching*, using mechanisms such as a DEMOTE operation [41], eviction-based data placement [9], or having the client side keeping track of the server cache status [2]. For single-level systems, the Free-behind technique [29], used in Solaris, tries to evict sequentially accessed blocks. In a sense, PFC implicitly performs exclusive caching by selectively bypassing the lower-level cache. Like the above approaches, this bypassing helps preserve the combined cache space. However, PFC is unique by performing prefetching-aware bypassing to actively throttle the prefetching aggressiveness.

Improving the cache hit ratio has traditionally been the goal of cache management research. In a more recent work, however, Yadgar et al. introduce a new algorithm called Karma [42], whose optimization goal is the overall I/O cost instead of hit ratios. Karma is shown to outperform existing multi-level caching solutions given certain I/O hints. Like Karma, PFC tries to minimize the overall I/O time. However, instead of using hints, PFC works through a feedback system based on the dynamic interplay of application access pattern, cache space distribution, and hardware speed.

Finally, most hierarchy-aware multi-level cache management schemes [9, 2, 43] also accommodate multi-client settings. The discussion and evaluation of this paper is limited to single-client scenarios (which indeed represent a significant portion of real-world multi-level storage environments [8]). However, PFC can be easily extended to work with both multi-client systems and multi-stream workloads, since it takes a light-weight approach by adjusting prefetching parameters rather than explicitly performing cross-layer data placement.

2.2 Sequential Prefetching Overview and Sample Algorithms

While there have been more sophisticated strategies proposed, such as history-based prefetching [22, 39, 25], our discussion focuses on *sequential prefetching*, where a set of contiguous blocks beyond the ones requested will be prefetched. Sequential prefetching is used by most commercial systems as it achieves long-term prefetching effectiveness without making assumptions on the application access pattern or incurring extra I/O in making predictions [19]. For sequential prefetching, there are two common decisions made by a prefetching algorithm: "how much to prefetch" and "when to prefetch". Many prefetching algorithms used in actual systems today are adaptive and adjust the *prefetch degree (p)* dynamically according to the access pattern observed, prefetching farther beyond the requested blocks if the sequential access pattern is confirmed by hits on prefetched blocks. Regarding the timing of prefetching operations, *synchronous* algorithms issue a prefetch request only when there is a cache miss, while *asynchronous* algorithms generally use a *trigger distance (g)* to start prefetching when there is a hit: the next batch of blocks will be prefetched when the block with a distance of g from the end of the set of prefetched blocks is accessed. Some asynchronous prefetching algorithms, such as RA and Linux (to be introduced below), do not have a trigger distance, but trigger prefetching on each hit and each miss.

Below we briefly describe several representative prefetching algorithms that are used in our study. We implemented these algorithms in our simulator and evaluated PFC's impact on their two-level performance.

RA The P-block Readahead prefetching algorithm (RA) is an extension of the OBL (One-Block Lookahead) [36] algorithm, by increasing the prefetch degree p from 1 to P. P can be either fixed or adaptive [11, 37]. In our experiments, we used a fixed degree of $P = 4$. Thus RA has a relatively conservative behavior compared with other algorithms for sequential workloads, but a rather aggressive behavior for random workloads.

Linux prefetching The Linux prefetching algorithm maintains for each file a *read-ahead group*, which contains all the blocks prefetched on the current file access and a *read-ahead window*, which contains both the current and the previous read-ahead groups. The algorithm determines that the file is accessed sequentially if the next access is within the read-ahead window and will prefetch another group with twice the size as the current read-ahead group. This way, sequential accesses will double the prefetch degree, until the read-ahead group size reaches a pre-defined maximum value, set to be 32 blocks in 2.6.x kernels. If the next access is outside the *read-ahead window*, the algorithm will resort to conservative prefetching and only prefetch a minimum number of blocks (by default 3) after the on-demand block.

Is has been shown that the Linux kernel prefetching has significant impact on buffer cache replacement algorithms [3]. Among the prefetching algorithms we have experimented with, the Linux kernel prefetching algorithm is the most aggressive one, with an exponentially growing prefetching degree, which is aggravated when performed at two or more levels. In addition, compared to other algorithms we found that Linux prefetching obtains consider-

able performance gain by maintaining per-file prefetching parameters.

SARC SARC [20] was developed at IBM and deployed in the IBM flagship storage controllers DS6000/8000. Unlike the other algorithms we examined, SARC is actually a combined algorithm doing both prefetching and cache management. It uses a fixed prefetch degree p and a fixed trigger distance g. To handle mixed workloads that contain both sequential and random accesses, SARC maintains two LRU queues, namely SEQ and RANDOM, for sequential and random data, respectively. It optimizes the use of the fixed cache space by equalizing the *marginal utility* of the two queues.

AMP The AMP algorithm [19] is proposed recently to coordinate prefetching in multiple sequential access streams and has been deployed by the new IBM DS8000 system released in Oct. 2007. It adjusts both p and g dynamically and coordinates the prefetching of multiple access streams. The design of AMP was based on the observation that the cache space is best utilized when the prefetch degree for stream i is equal to the product of the request rate of stream i and the average cache life. AMP increases p_i when the sequential access pattern is confirmed and reduces p_i when it detects overly aggressive prefetching (from the eviction of prefetched blocks that have not been accessed). The trigger distance g_i is reduced when p_i is reduced, and increased when the prefetched block is found to be waited on by an on-demand request, which indicates that the prefetching has not been triggered early enough.

3 Hierarchy-aware Prefetching with PFC

3.1 PFC Architecture

Although PFC is designed to be able to apply to systems with more than two levels, we focus on two-level systems (a common architecture in today's multi-level storage systems and data centers) in our discussion and experiments. For brevity, we refer to the upper (client) level as L1 and the lower (server) level as L2, for the rest of the paper.

As mentioned in Section 1, the multi-level prefetching problem presents somewhat conflicting demands. The goal of PFC is to moderate the prefetching process to achieve a desired level of aggressiveness in order to improve the overall system performance and resource utilization. The major design question here is "where should PFC reside", L1, L2, or both? We decide to place PFC in L2, as an intermediate layer between the client and the server's native prefetching and cache management, for a more portable and algorithm-independent design. A previous study [8] on multi-level cache management reveals that it is not worthwhile to sacrifice the transparency of the L1/L2 interface to add "aggressively-collaborative" mechanisms at the client side. In contrast, "hierarchy-aware" mechanisms sitting at the server side that leverage the knowledge gathered on the upper-level cache to perform intelligent cache replacement are more feasible and can achieve similar performance gains. We believe this also applies to the multi-level prefetching scenario, and our empirical results from implementing and evaluating a client-side prefetching coordination scheme indicate the same. Due to the space limit, in this paper we only discuss our proposed server-side PFC design.

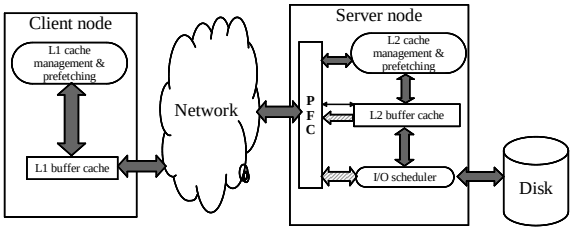


Figure 2. Software architecture for two-level systems using PFC.

Figure 2 illustrates the location and interfaces of the PFC module, and give an overview of how it functions (with more details and the algorithm discussed in 3.2). PFC resides at the server side as an intermediate gateway between the client node and the server-side I/O request processing. It intercepts the client requests (which may have included client-side prefetching), and relay data blocks between the client interface and the L2 I/O stack. It may query the L2 cache status and find out whether a certain block is currently cached. In general, PFC is aware of the existence of caching/prefetching both above and beneath the L1/L2 interface, but unaware of the actual strategies used.

Based on the observed L1 requests and the L2 cache inventory regarding requested blocks, PFC detects whether the L2 prefetching may be too conservative or too aggressive. For example, it can infer the aggressiveness of L1 prefetching by looking at the request size, or that of L2 prefetching by checking how many blocks beyond those accessed by L1 have been stocked in the L2 cache. Depending on the situation, PFC may take one or both of the following two actions:

- *bypass:* bypassing the L2 cache and directly feed some or all of the requested blocks to L1. PFC does this by interacting directly with the L2 I/O scheduler, or drawing cached blocks from the L2 cache without notifying the L2 native caching/prefetching unit of a hit. This action serves two purposes: 1) slowing down L2 prefetching by "faking" an L1 access stream with weakened sequential pattern, and 2) performing exclu-

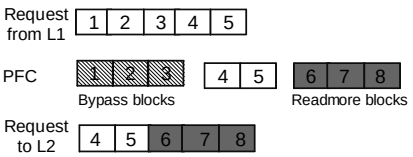


Figure 3. Sample PFC actions on L1 requests

sive caching on sequentially accessed blocks by avoiding caching them in L2.

- *readmore:* appending additional blocks to prefetch to the original L1 request. This action speeds up L2 prefetching when PFC decides the native L2 strategy is not aggressive enough.

3.2 PFC Algorithm

The goal of the PFC algorithm is to adaptively select a proper degree of aggressiveness in prefetching and make efficient use of L2 cache space. As mentioned in the previous section, PFC accomplishes this by activating one or both of two somewhat counter-acting operations: bypass and readmore. Figure 3 illustrates possible actions on a sample request issued by L1. In this example, the original request contains consecutive blocks 1-5. PFC may decide to bypass the first three blocks, and perform additional prefetching of its own for "readmore" blocks 6-8. This way, the request for blocks 1-3 are directly issued by PFC to the L2 I/O scheduler, while the L2 native caching/prefetching unit sees an altered request for blocks 4-8.

Algorithm 1: $PFC\_Process\_Req(req_u = [start_u, end_u])$

$req\_size = end_u - start_u + 1$;
avg_req_size = average request size so far;
$max\_rm\_size = MAX(req\_size, avg\_req\_size)$;
$PFC\_Set\_Param(req_u)$;

$start_{pfc} = start_u + bypass\_length$;
$end_{pfc} = end_u + readmore\_length$;
process request $[start_u, start_{pfc} - 1]$ directly;
forward request $[start_{pfc}, end_{pfc}]$ to native L2 processing;

/*Insert new items into queues*/
if *bypass_queue or* $readmore\_queue$ *is full* **then**
  evict oldest items until required space is available;
insert $[start_u, start_{pfc} - 1]$ into $bypass\_queue$;
$end_{rm} = end_{pfc} + max\_rm\_size$;
insert $[end_{pfc}, end_{rm}]$ into $readmore\_queue$;

Note that PFC's bypass action intercepts L1 requests from reaching L2, but may still access L2 cached data. In this example, blocks 1-3 will not be requested through the native L2 caching/prefetching modules. However, if some of these blocks are already cached in L2, PFC will serve L1 by reading them from the L2 cache rather than going to the disk. This way, the L2 cache may get a "silent hit" not registered with the native caching/prefetching algorithm.

The key decisions the PFC algorithm has to make, of course, are "when" and "how much" to perform bypass and/or readmore. Intuitively, the bypass blocks should be a prefix of the original request, as they are expected to be accessed first and ought to be cached closer to the application. At the same time, the readmore blocks should be the blocks immediately following the original request. The remaining challenge is to determine the trigger condition and the degree for each action.

To perform such dynamic decision-making, PFC manages two queues, the *bypass queue* and the *readmore queue*. These queues do not store real data blocks, but block numbers. Both are initially empty, and maintained with the LRU policy (the least recently inserted or re-accessed blocks are evicted when the queue is full). In our experiments, we set the maximum size of both queues to 10% of the L2 cache size. The queues help PFC detect the need to increase/decrease the bypass or readmore levels, by setting the key PFC parameters $bypass\_length$ and $eadmore\_length$, which are both initialized as 0.

The PFC request processing procedure is given in Algorithm 1, which takes the original L1 request, compute the PFC parameters by calling subroutine *PFC_Set_Param()* (Algorithm 2), and process the request with optional bypass and readmore actions accordingly. Finally, some of the requested blocks are added to the appropriate queue if they are not already there. Here PFC treats the two queues differently. The bypassed blocks are added to the bypass queue. However for the readmore queue, rather than adding the readmore blocks PFC appended to the L1 request, it adds blocks in a *readmore window* following those readmore blocks to the readmore queue. The size of this window is determined by a parameter max_rm_size, calculated in Algorithm 1 from the current and average request sizes.

Algorithm 2 describes how PFC set $bypass\_length$ and $readmore\_length$. Basically, PFC monitors the request pattern, as well as the hit status of requested blocks in the L2 cache and both PFC queues, to determine whether it needs to increase/decrease the bypass/readmore activities.

One upfront step is to check whether the L1/L2 prefetching is already quite aggressive. PFC considers the former true if the L1 request appears large (longer than half of the average L1 request size), and the latter true if as many blocks as requested immediately beyond the requested range are already stocked up in the L2 cache. In these cases, PFC will choose to bypass the entire L1 request, and set $readmore\_length$ to 0.

If neither condition above is satisfied, PFC will perform more detailed checking to see whether any blocks requested

Algorithm 2: $PFC\_Set\_Param(req_u = [start_u, end_u])$

```
hit_cache = hit_bypass = hit_readmore = false;

/* Check against aggressive L1/L2 prefetching */
if ((req_size > 1/2 avg_req_size) or
([end_u, end_u + req_size] ∈ cache)) then
    bypass_length = req_size;
    readmore_length = 0;
    return;
endif

/* Check hit status of L2 cache and PFC queues */
for start_u ≤ x ≤ end_u do
    if x ∈ cache then hit_cache = true;
    if x ∈ bypass_queue then hit_bypass = true;
    if x ∈ readmore_queue then hit_readmore = true;
endfor

/* Adjust PFC parameters */
if !hit_bypass then bypass_length++;
if !hit_cache then
    if hit_bypass then bypass_length--;
    if hit_readmore then
        readmore_length = max_rm_size;
    else readmore_length = 0;
endif
```

are found in the L2 cache, the bypass queue, or the readmore queue. If none of the blocks have been bypassed earlier, PFC assume the L1 cache can store more and increases $bypass\_length$. Otherwise, if there are blocks found in the bypass queue, but not in the L2 cache, PFC infers that the L1 cache space is tight, and previously bypassed blocks have been evicted prematurely. In reaction, it will reduce $bypass\_length$. The treatment of $readmore\_length$ is coarser: it will be increased to max_rm_size if the sequential access pattern anticipated is confirmed by having a hit in the readmore queue, and reset to 0 if otherwise.

From the algorithms given, it can be observed that random accesses are likely to be bypassed, except at the beginning of the run when $bypass\_length$ is zero or very small. This is desirable since the temporal locality of L2 accesses is expected to be low. A related issue is that in our current PFC implementation, the lower level maintains a single set of parameters. However, it is easy to extend PFC to maintain per-client or per-file contexts, in order to better handle multiple access streams.

4 Performance Evaluation

4.1 Simulator Overview

The simulator used in our trace-driven evaluation is extended from an existing two-level storage simulator that was used in several previous studies on multi-level cache management [43, 9, 8, 45, 46, 44], which has been validated against real systems and released to public.

We extended this base simulator in two areas. First, we added prefetching to both levels and implemented several prefetching algorithms, as well as our proposed optimization. Second, we made the simulator time-aware. The base simulator was designed to study cache replacement algorithms, for which it suffices to only consider the access sequence and ignore the actual timing of the requests. As existing research suggested [3], when prefetching is taken into the picture, one should examine the overall system performance rather than just on the cache hit ratios. To calculate the disk I/O time, we connected our simulator to the widely used disk simulator DiskSim [18]. We also implemented in the simulator an I/O scheduler that imitates I/O scheduling in Linux kernel 2.6. Finally, with the assumption that the network interconnection between L1 and L2 is unlikely the system bottleneck, we used a simple model [10] to compute the communication cost as $\alpha + \beta \times message\_size$, where α is a fixed startup latency and β determines the size-dependent cost. In our experiments, we set α as 6 ms and β as 0.03 ms/page, both measured through tests of TCP/IP data transfers between two computers in a LAN.

In this paper, we assume that the system is composed of one upper level cache, one lower level cache, and a disk, a valid setting for many real multi-level systems [8]. However, our simulator can be easily expanded both horizontally (to include multiple nodes at each level) and vertically (to add more levels), by replicating nodes and disks.

4.2 Test Workloads

Our simulation uses three large real-system traces, representing a variety of typical multi-level system workloads and carrying different degrees of randomness in accesses. Below we briefly describe these test traces.

SPC traces SPC[1] is a widely used benchmark collection provided by the Storage Performance Council, that has been adopted by many previous studies on prefetching and multi-level cache management [19, 20, 28]. We selected two workloads from SPC, "OLTP", traces from OLTP applications running at a large financial institution, and "Web", websearch traces from a popular search engine. The OLTP trace is the most sequential one in our test workloads, with only 11% of requests being random accesses. The Web trace, on the other hand, is the least sequential, with 74% of accesses random.

As our base simulator is not compatible with the newer version of DiskSim, we used a previous version (DiskSim 2), which has been used in several recent studies [27, 44].

[1] http://traces.cs.umass.edu/index.php/Storage/Storage

The problem is DiskSim2 supports only older disk models with limited capacity. For example, the largest disk capacity allowed by DiskSim 2 is 9.1GB (with the Seagate Cheetah 9LP hard disk model used in our experiments). Due to this limitation, also to control the total simulation time, we used only the first 10GB of data requests from the SPC traces. That accounts for 10.8% of requests from the OLTP and 31.3% from the Web trace, resulting in a total footprint of 529MB and 8392MB, respectively.

Purdue Multi trace We also used one of the "Multi" traces collected by researchers at Purdue university in 2005 [3], from the concurrent execution of three applications: *csscope*, *gcc*, and *viewperf*. This trace accesses a total of 12,514 files, with a combined footprint of 792MB. This is a trace with mixed access patterns, with 25% of accesses being random. Unlike the SPC traces, where each trace record bears an application request timestamp, the Purdue traces were collected from running the workload benchmarks at the throughput allowed in a test system. We followed the way these traces were used in the Purdue researchers' work, by issuing the requests in a synchronous manner (only issuing the next request when the current one completes).

4.3 Evaluation Results

Overall performance of PFC We evaluated PFC on the three trace workloads described above and the four existing prefetching algorithms discussed in Section 2.2. Each algorithm is applied to both L1 and L2. For every trace-algorithm combination, we tested different cache settings. The L1 cache size is set according to the trace footprint, with a "high setting" (H) that amounts to 5% of the total trace footprint, and a "low setting" (L) to 1%.[2] When the L1 cache size is fixed to H or L, we varied the L2 cache size by adjusting the L2:L1 size ratio, using four configurations: 200%, 100%, 10%, and 5%. This simulates different L2 base configuration, as well as the scenario where a single server node is simultaneously serving multiple client request streams. At both levels, LRU is used as the cache replacement policy, except for SARC, which comes with its own cache management strategy. Finally, all the discussion on hit ratio in this section is regarding the L2 cache, as we found PFC, as a server-side optimization, has little impact on the L1 cache hit ratio.

To compare PFC with non-prefetching-aware exclusive caching, we implemented DU [8], which marks blocks that have just been sent to L1 with the highest priority for eviction, assuming those blocks are to be cached by L1. Like PFC, DU is an L2 local optimization aware of the existence of upper-level cache activities.

[2]The L1 cache size may seem quite small with moderate trace sizes, compared to today's server configuration. However, such cache sizes are reasonable for analyzing cache behaviors, considering the increasing application concurrency on each node and the relatively small disk size used in this version of DiskSim.

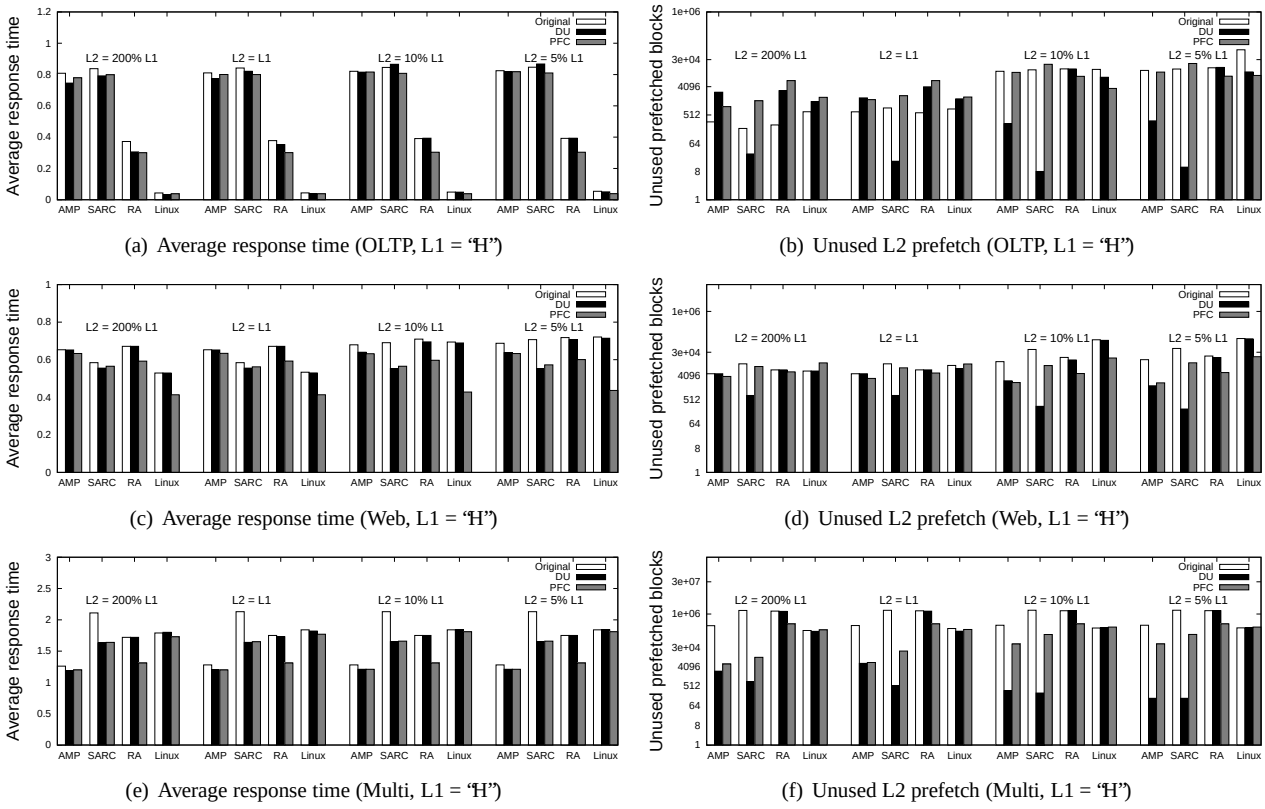
Figure 4 shows the results of the tests using the "high" L1 cache size setting. Since PFC is designed to moderate multi-level prefetching in storage systems, we consider the most important metrics to be 1) the overall system performance (in terms of the average request response time) and 2) the unused prefetch (in terms of the total number of blocks that are prefetched but not accessed when evicted or till the end of a test). The three figures in the left column plot the average response time, while the three in the right one plot the unused prefetch in log-scale. The "low" L1 cache setting tests yield similar results, and we omit the figures due to the space limit.

Trace	Cache size	Prefetch Algorithm			
		AMP	SARC	RA	Linux
OLTP	200%-H	3.55%	4.64%	19.23%	11.7%
	200%-L	0.23%	32.9%	18.49%	13.07%
	5%-H	0.78%	4.35%	22.54%	28.38%
	5%-L	1.51%	21.96%	18.17%	51.91%
Web search	200%-H	3.03%	3.25%	11.77%	21.88%
	200%-L	-4.67%	15.74%	4.95%	39.39%
	5%-H	7.97%	18.96%	16.34%	39.46%
	5%-L	8.16%	19.47%	16.15%	37.00%
Multi	200%-H	5.34%	22.32%	23.89%	3.54%
	200%-L	5.48%	20.16%	24.74%	1.71%
	5%-H	5.85%	22.22%	25.23%	1.96%
	5%-L	5.74%	20.01%	24.95%	3.08%

Table 1. Summary of PFC's improvement on the system overall performance.

PFC is shown to improve the average response time for 94 out of the 96 test cases (with a degradation of 0.2% and 4.67% for the two left, respectively). The improvement is up to 52%, with an average of 15.7% over all cases. For the majority of the cases (around 75%), it also outperforms DU, which optimizes L2 space usage by evicting blocks passed to L1, but does not actively adjust the aggressiveness of L2 prefetching. PFC, on the other hand, may make the L2 prefetching more aggressive or more conservative based on the access pattern and cache status.

As can be seen from the three charts in the right column of Figure 4, when the L2 cache size is large and the access pattern is highly sequential (OLTP, 200% and 100% L2:L1 cache ratios), PFC will actually aggravate L2 prefetching, resulting in higher numbers of unused prefetch. However, the overall performance is improved due to better L2 hit ratio, and the unused prefetch has a lower impact on the cache space utilization as there are not enough random blocks to compete for the L2 cache space. When the L2 cache size is relatively small, or when the accesses become more random (Multi and Web, 10% and 5% size ratios), in most cases PFC will slow down the L2 prefetching and reduce unused prefetch. In many of those cases, the L2 hit ratio is worse than in the original case, but PFC helps improve



(a) Average response time (OLTP, L1 = "H")

(b) Unused L2 prefetch (OLTP, L1 = "H")

(c) Average response time (Web, L1 = "H")

(d) Unused L2 prefetch (Web, L1 = "H")

(e) Average response time (Multi, L1 = "H")

(f) Unused L2 prefetch (Multi, L1 = "H")

Figure 4. Impact of PFC on overall performance and unused prefetches. Figures in the right column use log-scale at the y-axis.

the average response time by reducing the number of disk requests and/or making shorter requests. Both of these help to lighten the disk workload and provide faster response to the application requests.

One final note on Figure 4 is that PFC appears to maintain the relative performance of algorithms under most circumstances. This is appealing as PFC is intended to extend existing single-level prefetching algorithms found suitable for certain workloads to multi-level systems.

Table 1 summarizes the improvement on the average request response time with both the "high" and "low" L1 cache size settings. We can see that the most significant improvement comes from the Linux tests (for all traces), where PFC regulates the L2 prefetching to avoid prefetching too much when two levels of aggressive prefetching are compounded. However, even with Linux, PFC may decide to prefetch more aggressively at L2, such as in the cases of OLTP, with "H" L1 cache setting and 200%/100% L2:L1 cache size ratios. In such cases, PFC generates more unused L2 prefetch, but improves the response time by over 10%. The other cases where PFC benefit considerably include the SARC and RA algorithms, again for most of the trace tests.

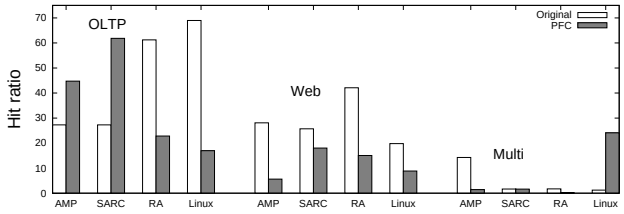
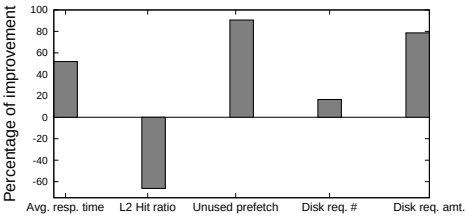


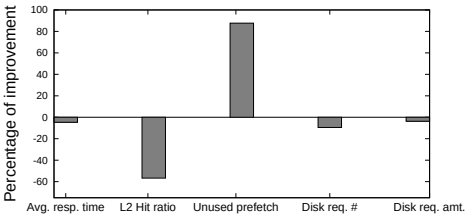
Figure 6. Average L2 cache hit ratio

For OLTP and large cache configurations, PFC makes the L2 prefetching more aggressive while for the other cases, it will suppress L2 prefetching. For random or mixed traces like Web or Multi, the reduced prefetching also translates to better L2 space utilization for random blocks, in addition to reduced I/O workload. In summary, PFC is able to make flexible, dynamic decisions, speeding up L2 prefetching in 32 test cases and slowing it down in 64 (collected from information not shown in the table).

Case studies To take a more detailed view into PFC's interaction with the native prefetching algorithms, in Figure 5 we plotted additional metrics for two test cases, where PFC



(a) Best case (51.9%): OLTP/Linux/L/5%



(b) Worst case (-4.7%): Web/AMP/L/200%

Figure 5. Case studies for tests where PFC obtained best and worst performance gain

obtained the most and the least performance gain (51.9% and -4.7% improvement on the average response time). These metrics include the L2 hit ratio, the total number of disk requests, and the total amount of disk I/O. For both cases, PFC decides to make the L2 prefetching more conservative, and as a result, the L2 unused prefetch is reduced, so is the L2 cache hit ratio. However, the overall performance is impacted in quite different ways.

For the best case (Fig. 5(a)), a sequential trace like OLTP has already triggered aggressive L1 prefetch, which is detected by PFC. With a smaller L1 cache size and a small L2:L1 cache size ratio, aggressive L2 prefetching should be avoided. Therefore PFC's bypassing bring a significant improvement to the total disk request size, which in turn generates a better average response time. For the worst case (Fig. 5(b)), however, the algorithm (AMP) is already quite conservative at L1 for this random trace. Therefore, the L1 to L2 requests tend to have small sizes, making it easier for sequential accesses to trigger the threshold for PFC bypassing. This way, L2 prefetching goes overly conservative and ends up *increasing* both the total disk request size and the total number of disk requests. Fortunately, such worst-case scenario only applies to a very small fraction of all the cases we tested.

Such case studies reveal the fact that the impact of PFC on the L2 cache hit ratio can be far away from that on the overall system performance. To illustrate this, in Figure 6 we summarize the differences in L2 hit ratio with or without PFC, by showing the average L2 hit ratio for each trace-algorithm combination. Actually, for the majority of the cases, PFC reduces (sometimes quite significantly) the L2 hit ratio, while achieving an overall performance gain. Such results agree with a previous finding by other researchers that when combined with prefetching, the cache hit ratio is no longer a reliable indication of the system performance [3]. Our observation is the deviation between the two is much more evident in a multi-level system.

Impact of individual PFC actions Finally, we examine the necessity of having both the "bypass" and "readmore" actions in PFC, using the OLTP and the Web traces. Figure 7 demonstrates the effect of enabling the bypass or the readmore action only. In most of the cases, combining the two counter-acting operations, PFC obtains a better performance gain than applying a single action only. One notable exception is for the AMP algorithm, where "readmore only" consistently outperforms the full PFC. This indicates that PFC is not prefetching aggressively enough for AMP, which agrees with our previous analysis.

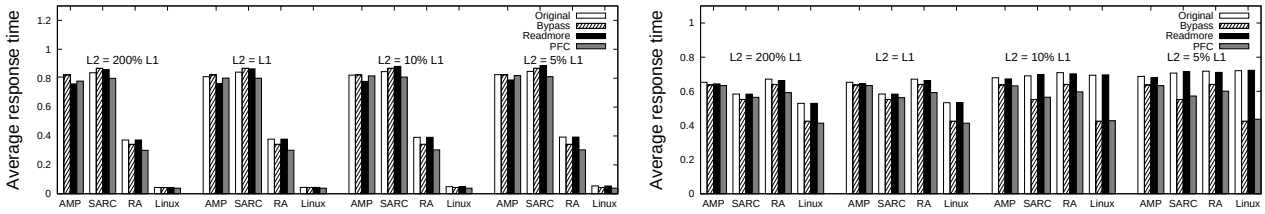
5 Conclusion and Future Work

In this paper, we analyzed the multi-level prefetching problem and presented PFC, a hierarchy-aware optimization that improves the performance of *existing* single-level prefetching algorithms when they are applied to multi-level systems. It automatically and dynamically adapts to the higher-level access pattern and the lower-level cache status, and controls the aggressiveness of the lower-level prefetching. PFC makes no assumption on the native prefetching algorithm or the application workload, and does not modify the I/O interface between neighboring storage system levels. Our extensive trace-driven simulation with diverse trace workloads and cache configurations demonstrates that PFC can deliver a consistent improvement on the average request response time. In addition, PFC enhances the I/O resource utilization and potentially the system scalability, by regulating the lower-level prefetching and reducing wasted prefetch.

There are several directions to pursue future work. Two particular interesting topics that we plan to study are to investigate 1) how PFC can enhance its decision making to better derive the upper-level prefetching behavior while maintaining its transparency, and 2) how to extend PFC to work with heterogeneous combinations of prefetching algorithms at multiple levels.

References

[1] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.



(a) OLTP (b) Web

Figure 7. Effect of combining the bypass and the readmore actions

[2] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. *SIGARCH Comput. Archit. News*, 32(2):176, 2004.

[3] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, 2005.

[4] P. Cao and S. Irani. Cost-aware web proxy caching. Usenix Symposium on Internet Technologies and Systems (USITS), 1997.

[5] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, 1995.

[6] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[7] Fay W. Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[8] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005.

[9] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, Jun 2003.

[10] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM.

[11] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, Washington, DC, USA, 1993. IEEE Computer Society.

[12] Fredrik Dahlgren and Per Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(4):385–398, 1996.

[13] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve fi le system performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[14] Asit Dan, Daniel M. Dias, and Philip S. Yu. Analytical modelling of a hierarchical buffer for a data sharing environment. *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

[15] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX '07)*, pages 261 – 274, 2007.

[16] J. Fritts. Multi-level memory prefetching for media and stream processors. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*, volume 2, pages 101–104, 2002.

[17] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 54–63, New York, NY, USA, 1991. ACM Press.

[18] G. Ganger, B. Worthington, and Y. Patt. The disksim simulation environment version 2.0, Dec. 1999.

[19] B. Gill and L. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies(FAST '07)*, pages 185–198, 2007.

[20] B. Gill and D. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the 2005 USENIX Annual Technical Conference(USENIX '05)*, pages 293–308, 2005.

[21] Song Jiang and Xiaodong Zhang. Ulc: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 168–177, Mar 2004.

[22] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM.

[23] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the third international symposium on Memory management*, 2002.

[24] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the International conference on parallel processing*, pages 28–31, 1987.

[25] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[26] Chuanpeng Li and Kai Shen. Managing prefetch memory for data-intensive online servers. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[27] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 173–186, 2004.

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

[29] Jim Mauro and Richard McDougall. *Solaris Internals*. Sun Microsystems Press , 2001.

[30] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 305–314, Berkeley, CA, USA, January 1991. Usenix Association.

[31] S. Devadas P. Jain and L. Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. In *Tech. Rep. CSG-462,M.I.T*, 2001.

[32] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference(USENIX '04)*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[33] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: an idea whose time has come. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[34] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[35] P. Reungsang, S. K. Park, S.-W. Jeong, H.-L. Roh, and G. Lee. Reducing cache pollution of prefetching in a small data cache. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, page 530, Washington, DC, USA, 2001. IEEE Computer Society.

[36] A. Smith. Cache memories. In *ACM Computing Surveys (CSUR)*, volume 14, pages 473–530. ACM Press, 1982.

[37] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, pages 306–313, Washington, DC, USA, 1997. IEEE Computer Society.

[38] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), 2000.

[39] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.

[40] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[41] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[42] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 169 – 184, 2007.

[43] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.

[44] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 177–190, New York, NY, USA, 2005. ACM.

[45] Qingbo Zhu, Francis M. David, Christo F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 118, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Qingbo Zhu, Asim Shankar, and Yuanyuan Zhou. Pb-lru: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 79–88, New York, NY, USA, 2004. ACM.