STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers

SHUANG LIANG, SONG JIANG, AND XIAODONG ZHANG

Technical Report

OSU-CISRC-3/07-TR23

STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers

Shuang Liang[1], Song Jiang[2], and Xiaodong Zhang[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
{liangs,zhang}@cse.ohio-state.edu

[2]Dept. of Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

Abstract

State-of-the-art networked storage servers are equipped with increasingly powerful computing capability and large DRAM memory as storage caches. However, their contribution to the performance improvement of networked storage system has become increasingly limited. This is because the client-side memory sizes are also increasing, which reduces capacity misses in the client buffer caches as well as access locality in the storage servers, thus weakening the caching effectiveness of server storage caches. Proactive caching in storage servers is highly desirable to reduce cold misses in clients. We propose an effective way to improve the utilization of storage server resources through prefetching in storage servers for clients. In particular, our design well utilizes two unique strengths of networked storage servers which are not leveraged in existing storage server prefetching schemes. First, powerful storage servers have idle CPU cycles, under-utilized disk bandwidth, and abundant memory space, providing many opportunities for aggressive *disk data prefetching. Second, the servers have the knowledge about high-latency operations in storage devices, such as disk head positioning, which enables efficient disk data prefetching based on an accurate cost-benefit analysis of prefetch operations.*

We present STEP – a Sequentiality and Thrashing dEtection based Prefetching scheme, and its implementation with Linux Kernel 2.6.16. Our performance evaluation by replaying Storage Performance Council (SPC)'s OLTP traces shows that server performance improvements are up to 94% *with an average of* 25%*. Improvements with frequently used Unix applications are up to* 53% *with an average of* 12%*. Our experiments also show that STEP has little effect on workloads with random access patterns, such as SPC' WebSearch traces.*

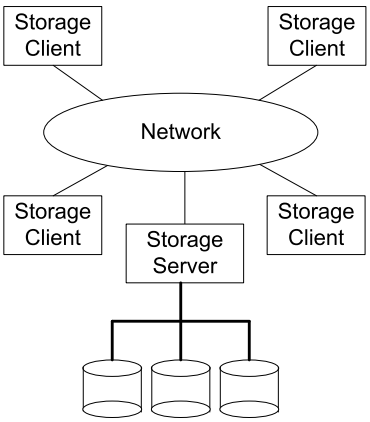


Figure 1. Networked storage architecture. Applications running in each client access files in the storage server through a shared file system, which locates in the storage client nodes and coordinates accesses from multiple clients to the shared storage server to maintain consistency. The caches are located in both L1 level in client and L2 level in the storage server.

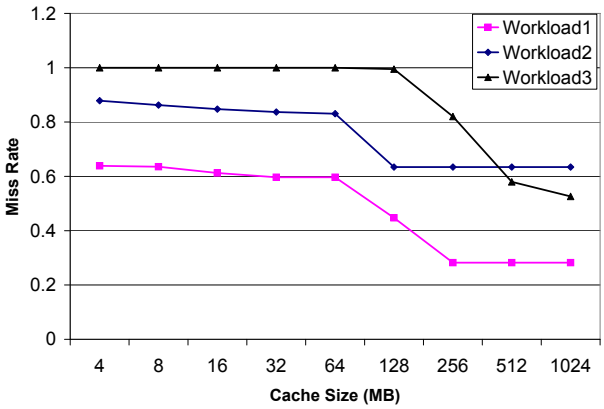


Figure 2. A simulation result illustrating the relationship of storage cache size and miss ratio using three different production-level storage system I/O traces from Storage Performance Council (SPC) with LRU replacement algorithm. Workload 1 and workload 2 contain mostly sequential accesses, and workload 3 contains mostly random accesses.

1 Introduction

Motivation. The fast growth of data resources in modern society has made networked storage servers increasingly popular for the ease of mass on-line data storage and management [21]. According to a survey of 260 businesses with more than 500 employees conducted for a Business Insight storage market research report [2], enterprises are adopting networked storage at the cost of directly attached models because of the growing storage requirements; and a majority of enterprises both in the United States and Europe have either installed a storage area network (SAN) or are considering doing so. This trend demands strong system support for networked storage server to facilitate efficient data accesses in such distributed environments.

In a typical storage area network (SAN) as shown in Figure 1, file system and storage devices are separated by networks. Data blocks are accessed through the linear block interface by multiple storage clients. Unlike the traditional directly attached storage devices, storage servers are resource-rich systems equipped with powerful CPUs and large memory as storage caches. In this networked storage architecture, there are at least two levels of data caching: buffer caching in each client (first level, or L1 caches), and storage caching in the storage server (second level, or L2 caches). In addition, plentiful CPU cycles are also available on the storage server for network and storage protocol processing.

The second level (L2) caches in storage servers play important roles in data access performance of clients. However, studies have shown that storage traffic exhibits much longer reuse distances[1] at L2 level due to the existence of L1 buffer caches at the clients, weakening its caching effectiveness (e.g. [29]). With a continuous increase of memory capacity in clients, this reuse distance is expected to grow accordingly at the L2 level, which can lead to a low utilization of storage cache.

[1] Reuse distance is defined as the number of distinct accesses between two consecutive accesses to the same block.

Two main factors underlie this effect. First, block access frequencies in most workloads is decreasing as their reuse distance increases, because most workloads follow the locality principle. Second, most cache replacement algorithms conforming to the LRU principle tend to keep recently accessed blocks and evict less recent ones, thus those potentially reusable blocks of long reuse distance are evicted before being re-utilized. Our simulation results using production-level storage system traces plotted in Figure 2 show that the storage cache miss rate would not be reduced by increasing caching capacity immediately after the cache reaches a relatively small size (around 256 MBytes in our example, with miss rates of the three workload flattening at 28%, 52%, and 63%, respectively).

Several studies have focused on improving the caching ability in storage servers by proposing various replacement algorithms to adapt to the weakened locality. A representative algorithm is the multi-queue cache replacement algorithm (MQ) [29]. Another approach is to make multi-level caches exclusive, such as demotion based placement [28], eviction-based placement [5], ULC [10], and X-RAY [1]. The effectiveness of these studies are limited in practice, because storage caches are mainly considered for holding previously demanded blocks in the hope of future reuse, whose probability is diminishing at the L2 level. While research efforts have been focused on caching, prefetching as another important mechanism to utilize caches is not effectively leveraged by existing networked storage systems. This is because existing systems do not well utilize two unique strengths of the networked storage servers. First, modern powerful storage servers have abundant CPU cycles, idle disk bandwidth, and large memory space, providing many opportunities for aggressive disk data prefetching to significantly improve server utilization and client data access performance. Second, since the servers have the knowledge about high-latency operations in storage devices, such as disk seeks and rotations, they are able to dynamically detect access patterns of high cost-benefit ratio to effectively guide disk data prefetching. The theme of our work presented in this paper is to leverage the unique strengths of storage servers to significantly improve the server and disk bandwidth utilization, and data access performance of network-connected clients.

Although prefetching has been extensively studied at the file system and disk device levels, they are not directly applicable for emerging networked storage servers. On one hand, storage server prefetching is based on raw blocks rather than file objects, hence file-level semantic information is not available to guide prefetching. On the other hand, simply prefetching whole track(s) for sequential accesses as traditional disk-level prefetching does is inadequate, since it does not well utilize the workload access patterns and is likely to preload undesirable data. Moreover, previous prefetching schemes are mostly *conservative*, while the availability of system resources in storage servers allows us to design and implement *aggressive* and cost-benefit-aware prefetching schemes, which would be more effective than normal *conservative* prefetching methodologies. Recently, a study in [7] presents the prefetching algorithm used in IBM's high-end storage server – Shark. There are two limitations of this approach. First, it detects access sequentiality based on the residency of immediate previous page in storage cache. Second, the prefetch request always preloads a static number of pages. To address these issues, we detect sequentiality by using historical information of each sequential stream instead of relying on the residency of previous pages. And we determine the prefetching request's length dynamically based on cost-benefit analysis. Finally, we consider not only

sequentiality but also disk thrashing for prefetching decisions.

Main Ideas. We propose a Sequentiality and Thrashing dEtection-based Prefetching scheme (STEP) to aggressively prefetch disk data based on cost-benefit analysis for two typical storage access patterns: sequential access patterns and disk thrashing patterns. In the sequential data access pattern, blocks are accessed contiguously; in the disk thrashing pattern, blocks are accessed alternatively in multiple *neighborhoods* concurrently, which causes frequent disk head movements among them. We detect these two patterns from the intermingled request sequences received by storage servers. By maintaining access statistics in prefetching contexts for these patterns, we are able to keep track of per-pattern history information and support effective prefetching decisions based on cost-benefit analysis. We design a new cost-benefit analysis model, which takes each prefetch context as input and generates aggressive yet appropriate prefetch requests.

STEP focuses on critical system bottlenecks. It optimizes performance based on storage server's high-latency disk access operations such as disk seeks, and frequent access patterns such as sequential accesses. By identifying access patterns of high cost-benefit ratio using our analysis model, aggressive prefetching will be applied appropriately to hide disk access latency and reduce the number of expensive disk operations. STEP combines client access semantics (request sequentiality) with low level device operation awareness (high overhead of disk thrashing) by taking advantage of storage server's unique strength of being able to detect both patterns and to afford the analysis costs, yet it remains simple to be implemented.

Contributions. We have made the following contributions in this work:

- Very limited studies have been done on prefetching in storage servers. Our study makes a strong case for prefetching on storage servers through an implementation-based study, which has showed that significant performance improvement can be achieved by the proposed scheme STEP.
- STEP is based on cost-benefit analysis using historical access information, disk access costs and data access locality, which makes prefetching decisions accurate and reliable. Therefore, STEP allows prefetching to be aggressive with low risks rather than being conservative.
- Disk thrashing is a typical phenomenon that severely degrades storage system performance, which is not explicitly detected and prevented in existing systems. STEP detects access patterns that cause disk thrashing and significantly improve storage server performance through effective prefetching.
- Prefetching schemes in existing systems such as Linux and FreeBSD only consider single sequential stream accesses. In practice, sequential accesses often come from multiple streams. Using historical information, STEP can detect interleaved sequential streams to effectively guide prefetching.

We have implemented STEP on Linux 2.6.16 and evaluated our implementation by replaying production-level traces from Storage Performance Council (SPC) [23] and running several widely used Unix applications. The results demonstrate that

significant performance improvement can be achieved with STEP. The comparisons with Linux's default prefetching scheme and several heuristics show that our design increases the performance of the OLTP workload by 25% on average and up to 94%; other Unix applications' performance is also increased by 11.8% on average and up to 52.7%. At the same time, random workload performance has remained similar compared with other schemes including Linux. In addition, the results also show that only around 3% of additional CPU cycle is used for our implementation compared with the default Linux design.

Paper Organization. The rest of the paper is organized as follows: Section 2 provides additional related work; Section 3 discusses in detail the background of prefetching in storage servers; Section 4 presents our design of STEP; Section 5 describes our implementation; Section 6 presents the performance evaluation results. We conclude the paper in Section 7.

2 Other Related Work

I/O prefetching has been an active research topic since the early days of computer systems [20, 26]. As the technology trend allows CPU speed and memory/storage capacity to dramatically scale up, leaving I/O speed far behind to keep up with, I/O prefetching will continue to be important [13, 6].

Prefetching based on sequential access patterns is a conventional wisdom in file system and database systems [20, 26, 7, 18], due to the high predictability of sequential streams and low amortized costs for sequential accesses to hard disks. However, previous prefetching approaches have mainly remained *conservative* for the high penalty of miss predictions such as waste of precious disk bandwidth, cache pollution, and premature eviction of prefetched blocks [13]. At storage level, sequential prefetching is mostly device oriented, which operates only according to the physical data layout. In Ruemmler and Wilkes's disk model [16], the drive usually prefetches a whole track on a demanded request.

Enhanced I/O interface allows more accurate semantic information to be passed from applications to system for optimal prefetching. Cao *et al.* [4] used application-controlled prefetching and caching management for file systems. Patterson *et al.* [14] proposed an enhanced API to pass information (hints) to operating system for prefetching and caching cost-benefit analysis. Both implementations show excellent performance with real application workloads. However, due to the standard storage networking protocol, such semantic information cannot be passed to network storage servers [5, 19, 1].

Predictive prefetching approach does not require application-provided hints, which allows an easier deployment for a wider range of applications. With assistance from the compiler and a run-time layer within the operating system, Demke *et al.* [3] used locality analysis to generate prefetch requests. By tracing file access relationship, Lei *et al.* [11] used file access tree and dynamic pattern matching for file prefetching. Whittle *et al.* [27] provided a summary of recent file access predictive approaches and used a multi-predictor approach to improve file access performance. However, such file-level information is not available on storage server. Li *et al.* [12] used data mining techniques to find block correlation on storage server to direct prefetching for performance improvement. Although it is an interesting idea, this technique is not applicable

for storage server implementation in real time.

Some other interesting predictive approaches also exist in buffer cache management. Gniady *et al.* [8] proposed to use program context to predict I/O pattern for buffer cache management. *Grey Box* approach [19, 1] uses partial file level information to deconstruct storage for semantics to improve performance. However, the program context based approach is not applicable to storage server for the lack of program context information. The Grey Box approach assumes availability of upper level file system structures.

Our approach is a predictive and storage-oriented approach. We detect both sequential access patterns and disk thrashing patterns, and propose a new cost-benefit analysis model to make aggressive prefetching cost effective. The design is transparent to storage clients and we have shown its effectiveness through real system implementation and evaluation.

3 Background of Storage Server Prefetching

In this section, we compare storage server prefetching with standard directly attached storage systems prefetching and storage client block-level prefetching in detail. Then, we present the background for our detection-based prefetching scheme.

Comparison with Directly Attached Storage Prefetching. In traditional directly attached storage architectures, storage devices are assumed to be unintelligent. Disks with limited on-drive cache perform internal device-specific prefetching conservatively for speed-matching purposes [16]. Active semantic-driven prefetching is managed by the file system or the application itself, which determines the necessity for prefetching and issues appropriate prefetching requests. The directly attached storage architecture confines the application, file system and storage within a single system, so that an integrated approach [4] that considers application, file system and disk I/O scheduling can be applied to achieve excellent performance. However, the integrated approach is not applicable to networked storage architecture, as storage servers are independent systems across the network shared by multiple clients. Thus, the integrated approach used on the client side system cannot well utilize available server resources.

Comparison with Storage Client Block-Level Prefetching. Storage servers conduct prefetching at block level. However, the multi-level cache hierarchy in SAN enables block-level prefetching to happen in both storage clients and servers. Potentially, storage clients can, at the convenience of their location, leverage both semantic information, such as file usage pattern and application hints, and block level access patterns to issue aggressive prefetching requests to hide the I/O latency for applications. However, client prefetching performance is limited for several reasons.

First, client is essentially blind to storage server specific information such as storage data access costs. So client side prefetching can only be based on upper level semantics or logic block information. However, prefetching without accurate future access information as in most cases is inherently risky. Thus the cost-oblivious nature of clients makes it hard to generate aggressive prefetch request.

Second, client prefetching does not have global information of storage server's workload, such as aggregate workload information from multiple clients. Consequently, the clients can only issue prefetch requests based on their own local information. These requests are inherently "selfish" and may cause unnecessary throughput degradation of storage server. For example, client A issues request blocks $33, 34, 35, 36$, and client B issues request blocks $1024, 1025, 1026$, which the server receives in the order of $33, 34, 1024, 1025, 35, 36, 1026$, whose arriving interval is dependent on network characteristics. Without server prefetching, three disk seeks may be needed to service the requests, although only one seek is needed with storage server prefetching. The service time of two more seeks can defeat the well planned prefetching schedule of clients as well as degrade server throughput.

Finally, client-initiated prefetching requires that clients commit resources such as cache memory to store prefetched data, which keeps client prefetching from being aggressive, as valuable client memory is also used for virtual memory of various applications. In addition, directly prefetching data to clients also requires network bandwidth shared between clients and servers, which increases the penalty for miss prefetching.

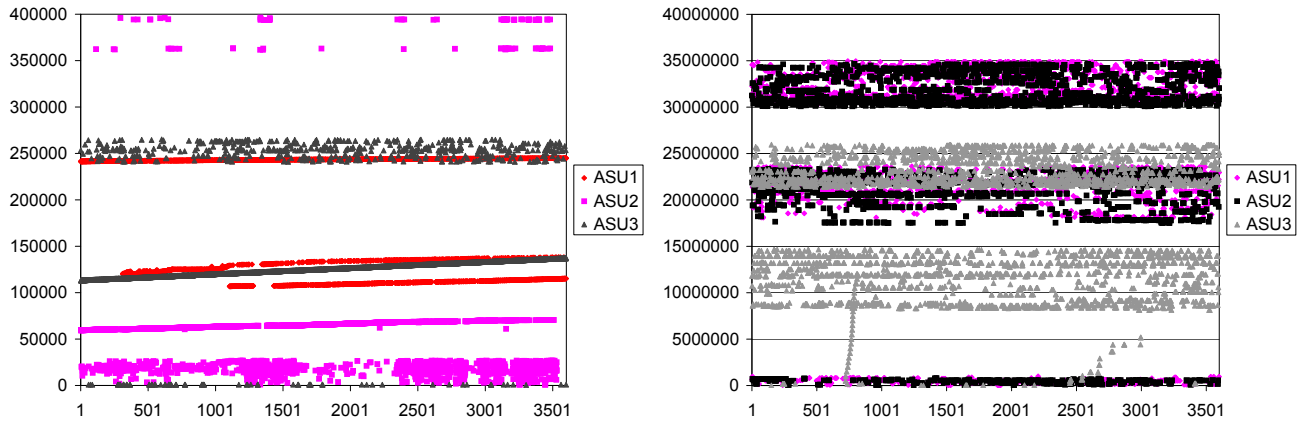
On the other hand, by detecting application access patterns, a storage server with its abundant computing power and storage cache space can leverage its internal knowledge of data access cost to conduct effective prefetching to improve application performance. In summary, although client prefetching can prefetch conservatively to mask storage and network latencies to a certain extent, it can never substitute server prefetching.

Detection-Based Storage Server Prefetching. A successful prefetching comes from an accurate prediction of future requests. However, due to standard network block protocol such as iSCSI [22], information useful for prefetching decisions, such as file-level access patterns and application prefetching hints used in previous research work cannot be passed to storage server effectively [1, 12]. Therefore, intelligent pattern detection techniques are needed for storage server prefetching management.

In general, an accurate pattern detection is a hard problem because of the dynamic nature of programs [3]. It is even more complicated by multi-tasking and multi-clients for the storage server, as different request patterns can be interleaved to cause pseudo randomness. In this paper, we focus on the detection of *sequential requests* and *disk thrashing*. Both patterns occur frequently in systems and can be accurately detected. Yet if not detected for optimization, they can incur considerable penalty.

Sequential pattern is a representative I/O workload characteristic in both application as well as block levels. This is due to the conventional wisdom of file system design, which maps file as sequential as possible to the block interface exposed by storage system. Figure 3(a) shows a snapshot of the histogram of SPC OLTP traces, from which it is clear that sequential patterns constitute a major part of these accesses. In fact, as mentioned in [7], SPC-2 benchmark specification observed by storage industry is specially designed to evaluate the performance of sequential workload.

Another critical issue in a modern disk-based storage system is disk thrashing, where disk heads moves frequently among several neighborhoods along the radius. While it is a major cause for throughput degradation [17, 16], thrashing can be easily



(a) OLTP application I/O (b) Search Engine I/O

Figure 3. Snapshot Histogram of SPC Storage I/O Traces of Different Workloads. ASU stands for Application Specific Unit of the storage server. The X axis is the request number and Y axis is the Logic Block Address (LBA)

found in common storage workloads. From Figure 3, it is clear that disk head can travel long distances constantly to serve sequential streams concurrently. This behavior is caused by both filesystem layout policies and multi-tasking. Layout policy decides the placement of meta-data and data. To utilize locality, a filesystem tries to pre-allocate space for logical objects (such as files) and place different semantic groups (such as directories) far away from each other so as to accommodate low overhead data expansion. With these polices, disks head tends to travel back and forth long distance for accesses to different semantic groups in a single application or more likely for multiple applications running concurrently. This significantly degrades server throughput, as disk seek is the most expensive disk operation.

Based on the above two observations, we propose algorithms to detect sequentiality and disk thrashing in Section 4. Using a new cost-benefit model that analyzes pattern attributes such as strength of sequentiality, intensity of disk thrashing and accuracy of predictions, we generate cost-effective prefetching requests for storage servers.

4 Detection-Based Prefetching Management

In this section, we describe in detail our STEP scheme. We first define the frequently used terms. Then we present our design of pattern detection and model based request generation. Finally, we discuss the interaction between prefetching and caching.

STEP is based on *sequentiality* and disk *thrashing* detections. In this paper, we define *thrashing* as accesses to an area of limited block address range, also called as a *neighborhood*, that are interleaved with accesses to other neighborhoods. This interleaving causes disk head movements (seeks) in and out of a neighborhood frequently within a short period of time. We restrict the definition of *sequentiality* to sequential accesses with stride of one for simplicity. In addition, a stream of

high confidence sequentiality refers to a sequential stream of long sequential access history and high *prefetching hit ratio* – the ratio which prefetched blocks are actually demanded by client and hit in the storage cache. A stream of *low confidence sequentiality* refers to a sequential stream of small length or large length but low prefetching hit ratio. Although "stride of one" seems restrictive here, the aggregate nature of block I/O at the granularity of block size (e.g. 512 or $1K$ bytes) as well as client-side sequential prefetching makes such assumption practically appropriate. In our experiments, strict sequentiality results in clear performance benefit for sequential workloads, as the strictness makes the choice of prefetching targets more selective, such that aggressive prefetching can be applied with low risks to maximize the benefit of prefetching for sequential patterns.

4.1 Pattern Detections

Storage devices, including single disks and storage arrays, expose their storage capacity as a large linear block array to clients. Logical Block Address (LBA) is used as address to reference this block array. When LBA is mapped to physical disk geometric address by a drive's internal algorithm, spatial locality is preserved as best as possible [17]. Therefore, file systems generally map logical sequential blocks to contiguous LBAs for performance. This allows us to detect client logical sequential accesses. On the other hand, this spatial locality preserving mapping also enables us to estimate disk access cost based on LBA for thrashing detection.

4.1.1 Detecting Sequentiality

A natural idea for detecting sequential patterns is to maintain recent access history. Upon a new request, the system refers to the history to decide if it is within a proximity of the last access to determine sequentiality. As an example, for most file systems, Linux creates two windows, current window and readahead window, to capture current accesses and prepare for future sequential accesses. On each new access, the windows are adjusted based on whether the new access is sequential. Usually, the readahead window is increased for sequential access and reduced for none sequential ones. At file system level, such windows are created per file with small additional overhead for each process, since file system needs to maintain these file objects at run time anyway. However, for a raw device, only one run-time object is created. Therefore, storage server can only benefit from a single object's information, since there is no notion of file at the storage server.

The single object scheme works conservatively for non-interleaved sequential streams. However, when multiple sequential streams occur at the same time, request mixture can cause the algorithm to recognize a sequential stream as a random one, or a high-confidence sequential stream as a low-confidence one. Clearly, the above conservative algorithm cannot support the storage server prefetching effectively.

With the computing power and large cache space available on storage servers, we are able to detect sequentiality from interleaved streams by creating multiple run-time objects for the storage device to record recent accesses. Each of these objects represents a potential or existing sequential stream. As new access comes, it is matched against these existing objects

to detect sequential access streams and update their sequentiality confidence.

In our algorithm, we use a *Prefetching Context (PC)* to represent the run-time object mentioned above. It includes attributes that describe this stream, such as the current address that the stream has accessed most recently, the current prefetch address that a stream has recently requested prefetching, the total sequence length of the stream and the stream's recent prefetch hit ratio. A PC also keeps records for recent accesses to this stream for prediction of the next request, e.g. recent requests' lengths and timing intervals. Finally, a PC contains pointers for maintaining itself in different data structures such as indexing trees and PC queues. Overall, the PC occupies only several tens of bytes. As a new request comes, the request address is matched against the current prefetch address to see if it extends any existing access stream. If one is found, the total sequence length of the stream is increased by current request's length. Then statistics of this stream are updated with attributes of this new request. To track the effectiveness of our prefetching scheme, we also update the *prefetch hit ratio* based on whether current request can be fulfilled from storage cache or not. If none of the PCs is found to match the new request, then a new PC is created for it.

The key issue for the algorithm to work is to design an efficient data structure to locate existing PCs and purge out-of-date PCs. Without proper management, a large amount of PCs may be created and kept for non-sequential request streams, which can increase the overhead of locating relevant PCs and the memory consumption on data structures. To overcome the problem, we index the PC using a balanced tree and bound the number of active PCs with a purging process running in background to delete useless PCs including both obsolete sequential ones or non-sequential ones. Since the number of active sequential sequences during a certain period of time is limited, only a reasonably large access history window is needed to identify and maintain those active sequential streams.

As shown in Figure 4, the algorithm for purging operates on three PC queues: *high-confidence*, *low-confidence* and *new*. PCs are managed within these queues using promotion and demotion policies. Each time when a PC is created, it is added to the *new* queue. As PC's sequentiality and prefetch hit ratio increases, it is promoted into the *high-confidence* queue. If the prefetching hit ratio of PCs in the *high-confidence* queue drops below a threshold, it is demoted to the *low-confidence* queue. Each queue is maintained in the order of access recency to facilitate LRU based purging within a queue.

When the upper bound of the number of active PCs is reached, the algorithm purges PCs in batches using a weighted round robin algorithm which favors different queues in the order of *new*, *low-confidence* and *high-confidence*. For the limitation of space, we leave out the details in the presentation.

For locating the PC, we use a balanced index tree [25] to organize all the PCs. The balanced tree ensures that operations such as deletion, insertion, and search can complete in $O(\log n)$ time. Since there is a constant upper bound of n of the number of PCs, the indexing and purging process involves only a small overhead.

4.1.2 Detecting Thrashing

For hard disks, disk seek operations between tracks are expensive. Ruemmler and Wilkes [16] described a model of modern disks, which breaks seek time into *speedup* time, *coast* time, *slowdown* time, and *settle* time. In this model, very short seeks

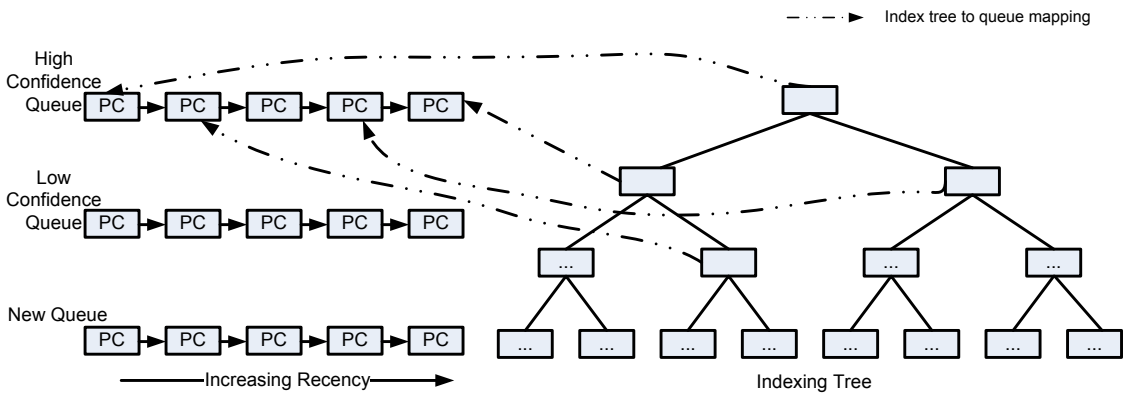


Figure 4. PC management data structures. Each PC is linked to one of the queues for purging based on recency and sequentiality. Meanwhile, it is also indexed by a balanced tree structure for efficient lookup.

are dominated by settle time; short seeks are proportional to the square root of the seek distance plus the settle time; long seeks are proportional to the distance plus a constant overhead. Although disk drive manufacturers intend to reduce the diameter of disks for average seek time reduction, the seek latency beyond around ten cylinders is still at the magnitude of milliseconds for current generation of technology [17]. Within this seek period, at least several tens of kilobytes sequential data can be read from a single drive. Therefore, if disk head is busy commuting between different locations, it may cause significant performance degradation. Although frequent seeks are inevitable for some workloads such as random accesses, thrashing as defined above can be effectively alleviated with better prefetching algorithms to reduce the number of seek operations.

For thrashing detection, we define *neighborhood accesses* as a series of spatially close accesses. For example, if we have a block access sequence of $20046, 46, 234546, 47, 48, 9848$. Then we can group subsequence $46, 47, 48$ as neighborhood accesses. To detect neighborhood accesses, the same algorithm of sequentiality detection is used in our work by keeping track of sequential streams [2]. Then for each neighborhood, we track the number of seeks and the sum of seek distances traveled to serve these accesses. In the above example, the distance traveled is $|46-20046|+|234546-46|+|47-234546|+|9848-48|$ blocks, and the number of seeks[3] is 4. These values indicate the intensity of thrashing.

To formulate the intensity of thrashing of each neighborhood, we use a weighted average seek cost that is biased to recent accesses. In another word, the intensity calculation gives more priority to recent access patterns to reflect up-to-date thrashing behaviors. In the following formula, $\mathbf{In}(i)$ is the estimated thrashing intensity at the ith seek; $\mathbf{Sc}(i)$ is the seek cost of the ith seek, which is based on the seek distance traveled and disk-specific parameters in Ruemmler and Wilkes's model; α is the weight given to the most recent thrashing request (e.g. $\alpha = 0.2$); and i is the count of seeks.

$$\mathbf{In}(i) = \mathbf{In}(i-1) \cdot \frac{(1-\alpha)(i-1)}{i} + \mathbf{Sc}(i) \cdot \frac{\alpha \cdot (i-1)+1}{i}.$$

[2]Actually, more coarse-grain detection method other than the strict sequentiality detection can be used in this scenario. But we exclude those cases for the simplicity of presentation here

[3]Due to low-level I/O scheduling, the actual number of seeks might not strictly correspond to disk head movement, but it is a good indicator statistically in storage systems.

Essentially, with the above formula, if a neighborhood is accessed continuously without thrashing, the intensity is converged to zero as i increases; if thrashing happens, intensity is a recency biased average seek cost, whose favor for recent seek cost decreases as the history length increases, asymptotically converging at α. When the intensity goes beyond a pre-defined threshold, a *thrashing condition* is reported, which triggers our aggressive prefetching driven by a cost-benefit model to be presented in the next section.

Unlike the sequentiality detection, the thrashing detection is based on storage server's internal knowledge about disk characteristics and its high cost operations. Therefore, the detection of thrashing should be based on addresses reflecting real seek costs rather than logical addresses which may not accurately reflect the distance-cost correlation. However, in real systems, logic block addresses(LBA) exposed to storage clients usually has a regular mapping to physical addresses to enable estimation of seek cost with high accuracy. Therefore, LBA is adequate in most cases. In fact, in low to medium end storage system configuration, as in our experiment settings, LBA is shown to be effective for thrashing detection. For highly virtualized high-end storage system with multi-level address translations, we can instead apply our algorithm after performing the address translation function which is available to storage servers.

Since LBA is used for thrashing detection in our work, we can combine the representation of neighborhood with PC for both sequentiality and disk thrashing detection algorithms. In addition to the attributes of PC described above, we add additional information such as recent seek cost, total number of seek operations and seek intensity to record thrashing related statistics.

4.2 Prefetching Model

With the patterns identified to initiate prefetching, we need to generate prefetch requests of appropriate lengths and schedule the requests to gain performance benefits. To design effective algorithms for such purposes, we first establish a cost-benefit model to determine the right length of a prefetch request. Since the prefetching cost is closely related to the I/O scheduling of the requests, we first discuss the I/O scheduling background, then derive the principles for generating prefetching requests based on a cost-benefit analysis. Our analysis uses time as metric to measure the cost and benefit.

4.2.1 Prefetching Cost and I/O Scheduling

The cost of prefetching is the time the system takes to bring the prefetched blocks into memory. Quantitatively, the cost of a single prefetching request includes the block transfer time, as well as, if any, the seek and rotation time for disk head to commute to the desired prefetching location and back where the disk head should have been otherwise. However, the same cost of prefetching can be perceived differently depending on the workload. For example, when the system is serving a workload with abundant "think time", the prefetching cost can be perceived by a client to be zero as long as it is scheduled to be overlapped with the think time to get full prefetching benefit. Given the luxury of time in this case, cost-benefit analysis is less interesting, as we just need to estimate the next fetch request and prefetch it whenever disk bandwidth is available during

the long interval. Therefore, we focus on the scenario when the server is kept busy with little "think time", which also applies to the situation of bursty traffic. In this case, the prefetch cost is largely dependent on I/O scheduling.

Generally, prefetch requests can be scheduled immediately or delayed with respect to the fetch request that leads to the prefetch decision. Issuing prefetch requests immediately following the fetch request has two benefits compared with delayed issuance where other disk accesses might interleave in between. First, immediate scheduling avoids additional seek cost, as prefetch blocks follow fetch blocks. Second, disk drives perform internal prefetching into their small on-drive cache; immediate scheduling can pick up those blocks without going to disk media. Both of the above benefits are due to the locality benefits of immediacy. The down side of immediate prefetching is longer latency serving the fetch request because of possible request merges at disk level. However, this can be solved by issuing the prefetching request asynchronously after the fetch request is fulfilled. So we choose immediate prefetch scheduling in our model.

4.2.2 A Cost-Benefit Model for Prefetching

To generate effective prefetch requests, we analyze the cost and benefit of prefetching based on a model to determine the appropriate prefetch length. In our model, we consider block transfer time, block access probability, and disk seek time. We do not consider rotational time in our model for two reasons: a) Average rotational time is smaller than average seek time, and its improvement rates is faster than seek time with current technology [9]; b) Using asynchronous prefetching, the prefetch request can cause rotational delay as well if the data is not already in the drive's cache, which can be common because with limited resources on disk drive, it is hard to detect sequentiality from the intermingled server traffic. The following notations are used for our presentation. ST is the average time for one seek; PT is the time to transfer the prefetched blocks; P is the probability that the prefetched data will be actually requested; and RT is the time needed to transfer the next request's blocks once the disk head is appropriately positioned.

From the server's perspective, the cost of prefetching for a (future) request is PT. The benefit has two different cases: a) the prefetched length is less than the (actual) request length, and b) the prefetched length is greater than or equal to the request length.

- For a), the benefit is $P \cdot RT \cdot Plen/Rlen$, where $Plen$ is the prefetched length, $Rlen$ is the requested length. This is because if the prefetcher does not retrieve enough for the next request, a seek is still required, if any, before reading the missing part due to requests interleaving on storage server, thus we only save the time to transfer the already prefetched part, which is $RT \cdot Plen/Rlen$. In this case,

$$\begin{aligned} \textbf{Earning} &= \textbf{benefit} - \textbf{cost} \\ &= P \cdot RT \cdot Plen/Rlen - PT \\ &\approx (P-1) \cdot PT. \end{aligned}$$

- For b), the benefit is $P \cdot (ST + RT)$. In this case,

$$\begin{aligned} \textbf{Earning} &= \textbf{benefit} - \textbf{cost} \\ &= P \cdot (ST + RT) - PT \\ &\leq P \cdot ST + (P-1) \cdot PT. \end{aligned}$$

From the formulas, we draw the following conclusions:

- Prefetching less than the next request length is almost always a waste for a busy server. In fact, in case a), the earning is less than or equal to zero. Therefore, estimating the future request length conservatively is usually a bad idea, which is a little surprising.

- As shown in b), when a sequential stream is accessed in an interleaved manner, there is a large prefetching potential for performance improvement due to seeks. Since P is reversely proportional to $(ST + RT)$ for a given earning, even when the probability of a future sequential access is small, aggressive prefetching can still be beneficial when the seek time is large. The benefit of aggressive prefetching is further magnified when a single prefetching request can cover multiple fetch requests later, as the cost of multiple seeks can be avoided.

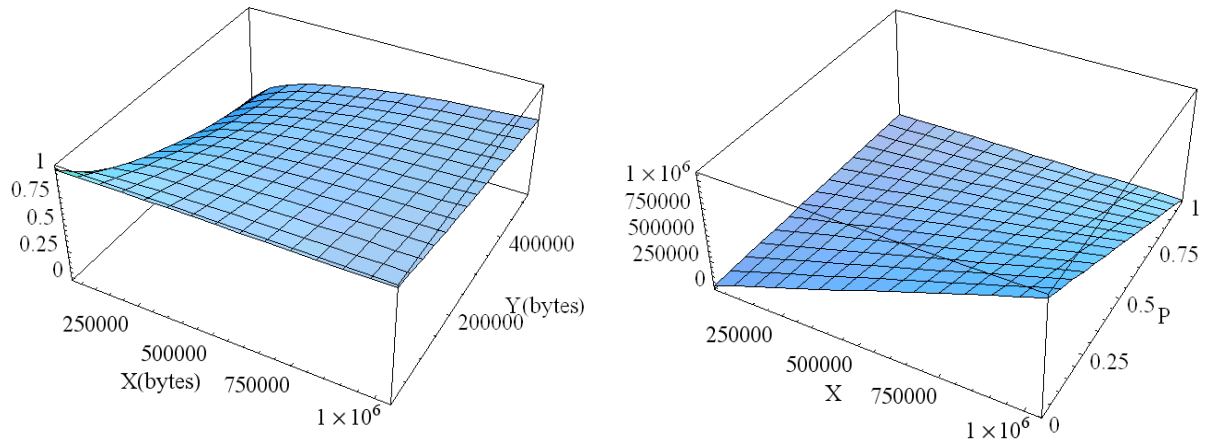
- For aggressive prefetching, the prefetch length should be bounded according to the probability distribution of the future sequential request. Roughly, to gain a certain benefit, as prefetching length PT increases, the prefetching hit ratio P needs to increase as well.

In summary, when the disk is busy serving request from different locations, prefetching has a great potential for performance improvement. At the same time, successful prefetching requires a delicate balance between the probability of a future access and the prefetching length.

4.3 Prefetching Request Generation

With the formula derived above, we estimate the next request length RT based on recent request history, and use a request *probability distribution function (pdf)* to generate the optimal prefetching request length.

Estimation of Prefetch Request Length. The estimation of the next request is based on recent history kept in the relevant Prefetching Context (PC), which stores the last N demanded fetch request lengths (N is very small, e.g. 4). Due to client-side prefetching/caching and the aggregate nature of block I/O, requests of a sequential stream perceived by storage server are comparatively predictable. For example, requests are multiples of page size; the request length of high confidence sequential streams is steadily increasing and then stays at constant due to client side prefetching. Therefore, prediction with limited history can be effective in many cases. Therefore, we use the average gradients of the past N request lengths and the latest request length to calculate the next one ahead as estimation.



(a)PDF for Sequential Workload

(b)Reverse PDF for Random Workload

Figure 5. Probability Distribution Function and Reverse Probability Distribution Functions for Different Workloads. For (a) Z-axis is the probability. For (b) Z-axis is the prefetching length

Prefetching Length Decision. Under two circumstances, we generate prefetching requests: a) detection of a high confidence sequential stream; b) detection of a thrashing condition. For high confidence sequential stream, we use its PC's prefetching hit ratio times the system-dependent prefetching upper bound to generate prefetching requests. For thrashing condition, we generate requests aggressively using case b) described in Section 4.2.2. We define a *probability distribution function (pdf)* $P(x, y)$, which takes the total length x of the past accessed blocks of this sequential stream and an expected prefetching length y as input, and outputs the probability that the whole length y is to be actually requested. Generally, this function is monotonically decreasing on y for a given x. However, depending on the workload, the decreasing rate varies. Intuitively, the longer a sequential stream is accessed in the past, the more likely the following sequential data is to be requested. Furthermore, the more sequential a workload is, the longer a sequential stream tends to be. These two observations lead to the following *pdf* generation guidelines. We use deflected exponential functions to approximate *pdf* for mostly sequential workload, and linear functions to approximate for mostly random workload. The coefficients of these functions are tunables, which can be determined empirically and modified as run-time parameters as system workload changes. The *pdf*s used in our experiments are shown in Figure 5.

In order to generate the optimal prefetch length, we extend the single request formula in Section 4.2.2 to include multiple subsequent requests. We use PL to represent total expected prefetching length and use RL to represent the average estimated length of the subsequent requests. The earning function $E(x, PL)$ is formulated as follows:

$$E(x, PL) = \int_{z=0}^{PL} P(x, z) \cdot \left(\frac{ST \cdot dz}{RL} + \frac{dz}{BW}\right) - \frac{dz}{BW}.$$

In the above function, BW is the disk streaming bandwidth. $P(x, z) \cdot (\frac{ST}{RL} \cdot dz + \frac{1}{BW} \cdot dz)$ represents the seek and transfer

time saved, i.e. the benefit of prefetching; $\frac{1}{BW} \cdot dz$ represents the prefetch data transfer time, i.e. the cost of prefetching. Thus we are ready to compute, using the reverse pdf function, the value of PL to maximize $E(x, PL)$. We omit the mathematical details.

4.4 Interactions with Caching

Prefetched blocks need to be managed properly to balance the needs from both caching and prefetching, as well as to avoid *premature eviction*, where prefetched data are evicted before they are actually requested [4].

As shown earlier, storage server blocks are less likely to be reused and, if any, the reuse distance is long. On the other hand, with our pattern detection and cost-benefit analysis, the prefetched data are of high confidence to be actually requested in the future. Considering a unified caching and prefetching block cache, higher priority should be given to the prefetched blocks in the cache replacement decision. Instead of designing a completely new algorithm to achieve the goal, we advise the cache management to adapt to the difference of cached and prefetched blocks. We instruct the cache management subsystem in kernel to raise the priority of prefetched blocks before they are requested. Then when they are actually being requested, the priority is recovered to what they should have been to keep a fair status for cache replacement. This approach is compatible with any replacement algorithm the server chooses to use due to its non-intrusiveness in nature.

To further strengthen our approach, we also incorporate delayed I/O scheduling in our design. In addition to the estimation of the request length, as described in Section 4.3, we estimate the fetch time of the next request. For request whose estimated fetched time is too far into the future, we put the prefetching request in a delayed queue with a timer for later scheduling to avoid *premature eviction*.

Finally, in our choice of PC purging, we consider the cache usage effectiveness in terms of hit ratio, meaning only high hit ratio PCs will be promoted to *high sequentiality queue* to mitigate the chance of being purged. Therefore streams that utilize cache poorly only prefetch conservatively and their PCs are purged earlier, which also helps to balance prefetching and caching.

5 Implementation in Linux Environment

Our prototype implementation is based on Linux 2.6.16 kernel. We have implemented our storage server using NBD – a network block protocol implementation distributed with Linux kernel and used in Redhat Global File System (GFS) [15].

The prefetching management system is implemented as a dynamic library on top of the raw disk device, which is used by NBD server. Figure 6 shows the overall implementation architecture. NBD protocol is a client/server protocol. NBD client is a pseudo storage device driver on the storage client to provide access to the storage server, while NBD server is a userspace storage provider which exposes the storage capacity of the server. Our choice of dynamic library as an implementation layer makes the prefetching functionality transparent to any userspace storage servers for easy deployment. This choice also gives our design the flexibility for system management and fault tolerance, which is very important for current computer

systems [24]. For example, in case any system bug is triggered that crashes the server, the server can be restarted or migrated easily without reloading the kernel. In addition, userspace implementation also allows complex computation to be performed efficiently for the availability of float point operations.

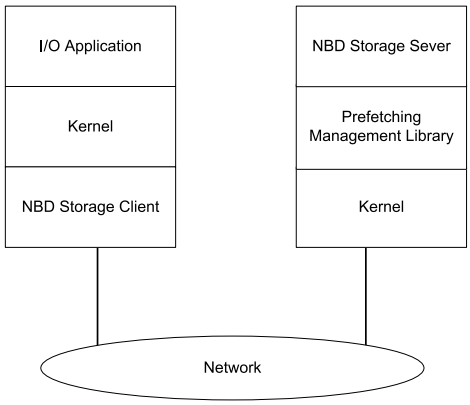


Figure 6. Storage Server Prefetching Management Software Architecture

On each block read request, our library creates or updates relevant PCs, then it generates corresponding prefetching requests when appropriate. To interact with the caching system, we add new system calls, which instrument the *read* and *readahead* system calls to give our library feedbacks about the cache status and tune the cache manager to promote or demote the pages of prefetched block for replacement algorithm.

We implements a Red-Black tree [25] as the balance tree in our design to index the PCs. Each PC is linked to one of the three link lists representing the PC three queues: new, low-confidence and high-confidence. Each list is maintained in LRU order to facilitate recency-based purging of victim PCs.

6 Performance Evaluation

In this section, we evaluate our prototype implementation of STEP. We first describe our methodology of evaluation, then describe the experiment platform setup, and finally present our results.

6.1 Methodology

The evaluation is conducted by replaying real production storage server traces and running frequently used I/O intensive applications. Our storage traces are from Storage Performance Council (SPC) [23], a vendor-neutral standards body. The traces include both OLTP application I/O and search engine I/O. As shown in Figure 3, the OLTP trace is mostly sequential, while the web search trace is random. We filter the traces to get read requests and classify resulting requests based on the Application Specific Unit(ASU) to create trace files used in our experiments. For application benchmarks, we use three frequently used Linux applications: *cscope*, *tar*, and *diff*. *Cscope* is a source code browser utility. It builds a cross-reference database from a set of files for the indexing purposes. We test the database build time as a benchmark for storage server

performance. *Tar* is a widely used Unix utility. We benchmark the tar ball creation time of a large source tree. Finally, *diff* is used for generating software patches. We benchmark the time of generating the patch to the kernel upon which STEP is built.

To demonstrate effectiveness of our design, we compare STEP with various Linux configurations. Due to the unavailability of the commercial storage server prefetch implementation used in Shark, we also compare STEP with several other heuristics below to illustrate the benefit of our design.

Conservative: prefetch the next estimated request only. This scheme does not prefetch based on cost-benefit analysis, neither does it prefetch on thrashing condition, thus clients only have the advantage of conservative prefetching for interleaved sequential streams compared with Linux.

Benefit Bounded Risk: bound the prefetching risk accordingly to the advantage a stream has benefited from prefetching. More specifically, we track the total prefetching hit length of each PC and generate prefetch length using a percentage of this total benefit length for prefetching request generation. This scheme does not consider the thrashing condition.

Sequential Aggressive: prefetch aggressively for high confidence sequential streams using prefetch upper bound times the past prefetching hit ratio on a cache hit. This scheme does not use cost-benefit analysis for request generation and does not consider thrashing. However, clients do benefit from aggressive prefetching for non-interleaved sequential streams.

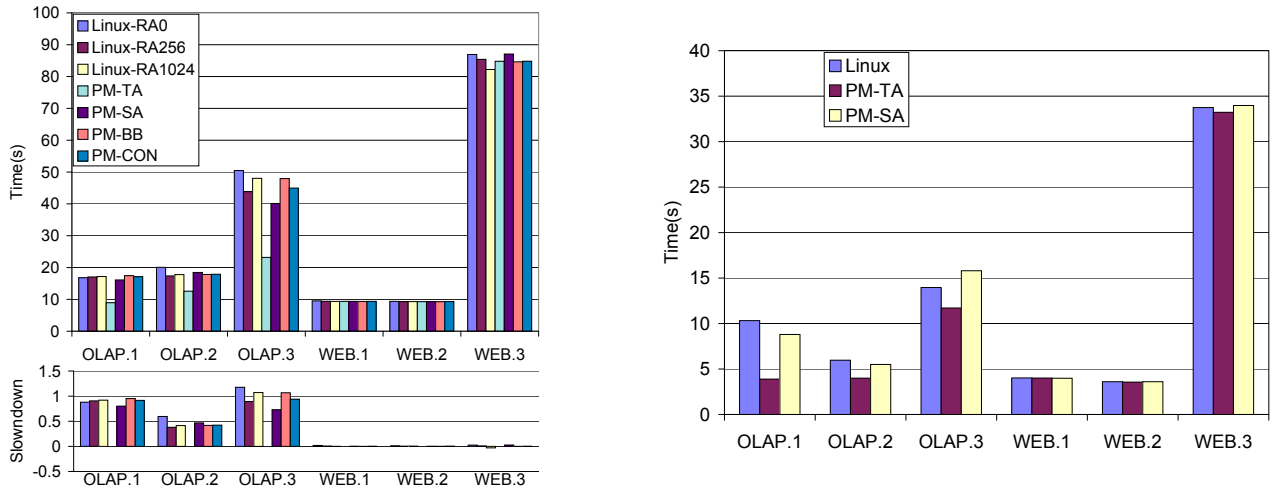
6.2 Experiment Platform

The experiments are conducted on an Intel Xeon 2.4GHz cluster. Each node has 1GB memory and 64 bit PCI-X 133 MHz bus. All nodes are connected to a 100Mbps fast Ethernet as well as an InfiniBand network. Each node is equipped with two 18GB 15K RPM SCSI disks and two 40GB 7.2K RPM, ATA/100 disk. We create a level one software RAID using the two SCSI disks for the experiments. To compare the technology impact, we also test with one of the ATA disk for some of the experiments. The operating system is Redhat Linux AS4.

We evaluate our design comprehensively using different system configurations to exclude device specific performance issues. We test our design with both local and networked storage scenario so as to illustrate the performance issues caused by network connections. For local test scenarios, we test with both IDE disk and SCSI disk array to demonstrate that the design is not device specific. For the network storage scenario, one node is set up as the storage server, which exposes the SCSI software RAID as block device using NBD.

6.3 Results

The following notations are used for simplicity of presentation in all test scenarios. *PM-TA* stands for STEP; *PM-SA*, *PM-BB*, and *PM-CON* stands for sequential aggressive, benefit bounded risk, and conservative schemes, respectively. For comparison with Linux's native prefetching scheme, we test Linux with different prefetching thresholds. *RA#* represents the block numbers Linux uses its prefetching threshold in the kernel. For SPC trace tests, we randomly choose three traces from each of the two workload types and use OLTP[1-3] and Web[1-3] as labels.



(a) IDE disk

(b) Software RAID

Figure 7. Local storage server performance test with different prefetching strategies and workloads using both IDE disk and SCSI Software RAID.

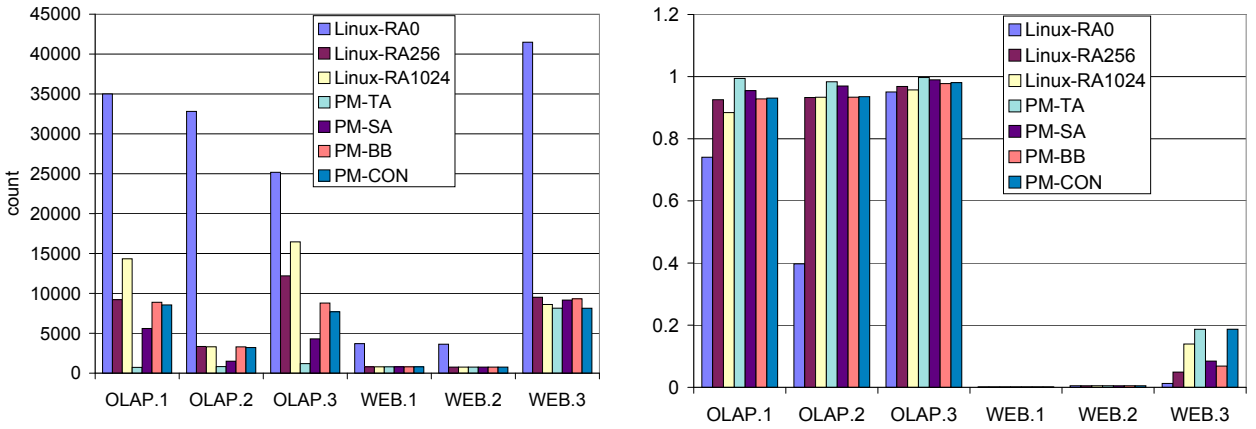
6.3.1 Local Storage Server Tests

Figure 7(a) illustrates the IDE disk results comparing the total execution time of different SPC traces and their slowdowns with respect to *PM-TA*. The results show that *PM-TA* performs best among all schemes for the sequential OLTP workloads. The performance improvements range from 38% to 117%. For random WebSearch workload, all the schemes perform similarly with slowdown ratios ranging from -3.0% to 2.6%. As expected, STEP significantly improves the performance of the interleaved sequential workload. At the same time, it has no noticeable effect on the random workload.

The results show that with Linux's default prefetching scheme, simply increasing prefetching threshold does not always lead to better performance for sequential workload, as it incautiously applies the aggressiveness to all sequential streams without cost-benefit analysis. For *PM-SA*, it prefetches aggressively only for *high confidence sequential stream*, which reduces the penalty for miss prefetching of *low confidence sequential streams*, thus improves the performance of Linux. With *PM-TA*, the incorporation of thrashing detection and cost-benefit analysis based prefetching, the performance is improved significantly. From the results of other heuristics, we also see prefetching conservatively does not give good performance and even degrade performance in some cases. Figure 8(a) and (b) provide the page miss count and the hit ratio for the above test scenarios. It shows that *PM-TA* achieves highest hit ratio for all sequential workloads and one of the random workload.

In Figure 7(b), we show that using SCSI software RAID – a faster storage technology, the benefit of STEP is consistent with an improvement ranging from 19% to 165% compared with the default Linux prefetching scheme.[4] And it also performs better than *PM-SA*. It shows that our sequentiality and thrashing detection scheme is applicable to both disk and RAID configuration.

[4]This set of result is not directly comparable with IDE disk results due to disk size difference and striping patterns of software RAID.



(a) Local IDE Total Page Miss Count

(b) Local IDE Hit Ratio

Figure 8. Local IDE Miss Count and Hit Ratio

6.3.2 Network Storage Server Tests

SPC Trace. For this set of tests, we compare Linux, *PM-SA* and the *PM-TA* scheme. All three schemes applies the default Linux RA value on the server side. On the client side, we vary the Linux RA value to compare the effects of client-side prefetching.

Figure 9 shows the execution time of the six SPC traces. In this figure, we also compare the performance with different network technologies, namely 100Mbps Fast Ethernet and 10Gbps InfiniBand network. InfiniBand supports different network protocol stacks with different performance and software compatibility trade-offs. In particular, the TCP compatible IPoIB protocol provides 1.3Gbps peak bandwidth on our testing platform.

The results show that *PM-TA* outperforms both Linux and *PM-SA* for all three sequential workloads in network scenarios. Among all test results, the OLTP1 workload improvement tops at 94% compared with default Linux prefetching scheme. And the average improvements across all the schemes using two different networks is 25.1% and with IPoIB only is 26.6%. For random workload, different schemes perform similarly as expected. The performance improvements vary from -1.3% to 2.4% with an average of 0.1%.

These test results reveal the network performance impact on networked storage I/O performance. Comparing the performance between 100Mbps Fast Ethernet and 1.3Gbps InfiniBand IPoIB, the overall performance improves much more for sequential workload than random workload. Therefore, our prefetching scheme sees a better performance improvement for sequential workload as storage's bottleneck effect becomes clearer with faster network technology. For example, with 100Mbps fast Ethernet, OLTP2 workload performs only 2.7% better than the default Linux scheme. However with 1.3Gbps InfiniBand IPoIB, the improvement increases to 9.7%.

Another observation from the results is that client-side prefetching is generally beneficial for all workloads. However, prefetching beyond the relatively small default upper bound of 256 blocks does not lead to better performance for all work-

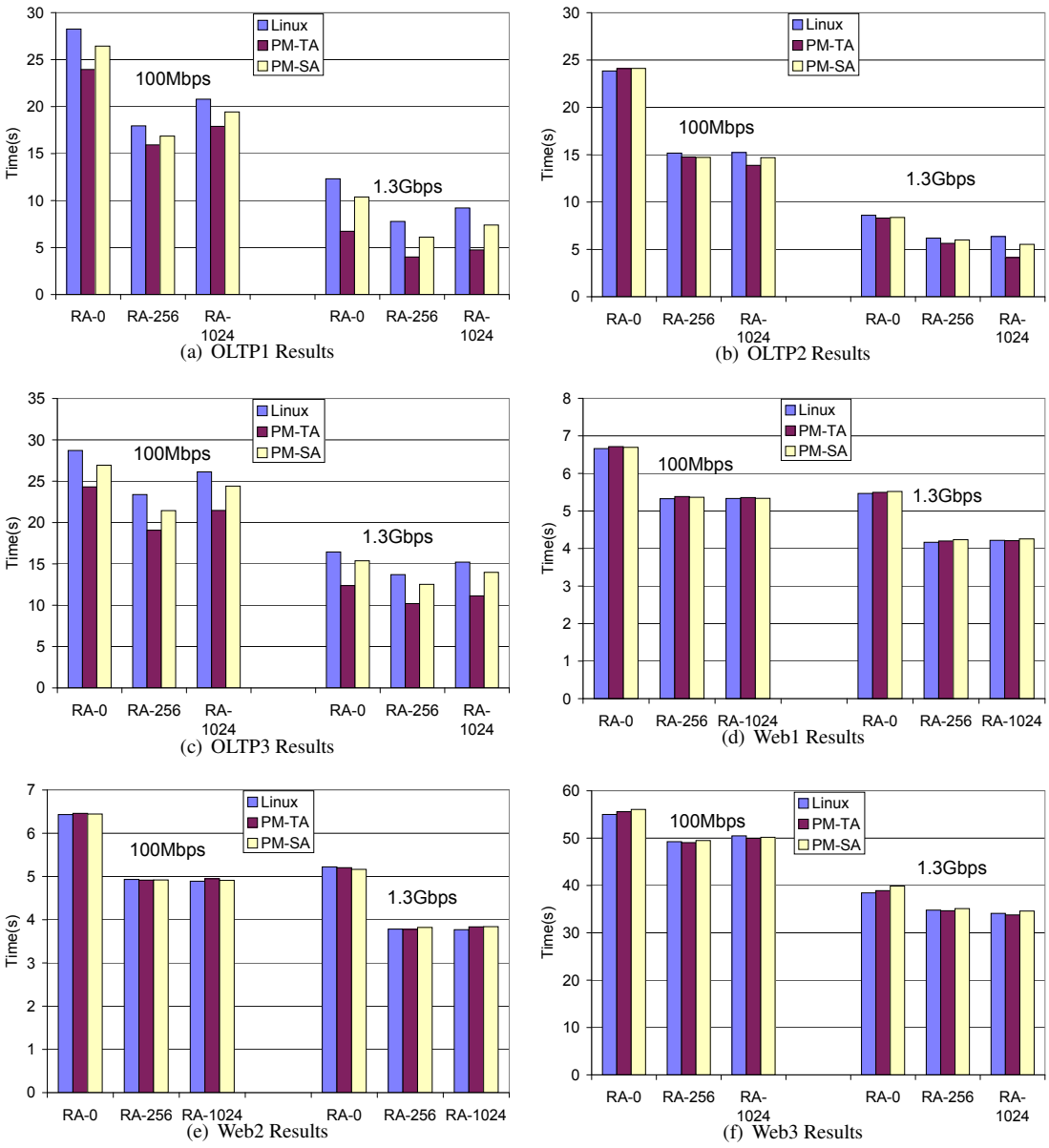


Figure 9. Network Storage Server Test Performance with Different Prefetching Strategies and Workloads.

loads. In two of the sequential workloads: OLTP1 and OLTP3, performance actually degrades noticeably ranging from 9% to 23%, which help corrobrate our arguements that block-level storage client prefetching cannot substitute storage server prefetching in Section 3.

Application Performance. For this set of tests, we compare Linux, *PM-SA*, and *PM-TA* schemes using application benchmarks on the client. Ext3 file system is built on the client and populated with files used frequently in this work including the Linux source tree and trace files. We use the *time* utility to measure the *real time*, *user time* and *system time*. In the figures, we report both the real execution time and "out of box" time, i.e. the difference of real time minus user and system time,

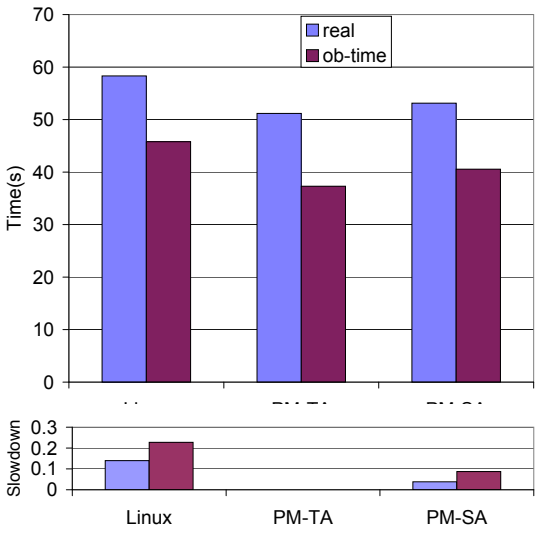


Figure 10. Tar performance results of archiving a built Linux source tree.

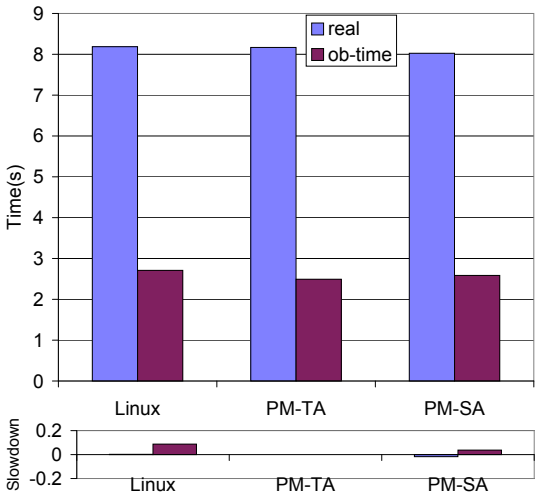


Figure 11. Cscope performance results of building a cross reference database.

which represents the storage server access time.

Figure 10 shows the results of *tar* creating an archive for a built Linux source tree. In this benchmark, the I/O pattern is sequential reads for files in the source tree and writes to the archive. The top part of the figure is the execution time in seconds, the lower part is the slowdown ratio. The results show that the *PM-TA* scheme performs best among the three. The real execution time improves 13.9% and 3.8% respectively compared with Linux and *PM-SA*. The "out of box" time improves 22.7% and 8.7%, respectively.

Figure 11 shows the result of *cscope* building a cross reference database for Linux source tree. In this benchmark, the major I/O operation is file meta-data access, such as *lstat* and *access* calls, so that the sequentiality is not as strong as the first benchmark. The results show that our *PM-TA* scheme performs only 2% better than Linux and 1.7% worse than *PM-SA*. However, for the "out of box" time, it improves the performance for 8.7% and 3.7%, respectively. We can see from the figure that the "out of box" time constitutes only around one third of the total execution time, which explains the small overall performance improvements.

Next, we test the performance of generating the patches to our customized kernel for supporting STEP. Figure 12 shows the result of *diff* comparing our modified Linux source tree and the original Linux source tree. In this benchmark, the major I/O operation is sequential file read for these two source trees. In order to serve this source comparison, disk head has to move back and forth to bring the data out of disk. Aggressive prefetching is very effective for this test scenario. The results show that our *PM-TA* scheme performs 43.7% better than Linux and 11.1% better than *PM-SA*. For the "out of box" time, it improves the performance for 52.7% and 12.5%, respectively. By detecting and prefetching for the thrashing patterns, our scheme has significantly improved the storage server access time which constitutes more than 80% of the total execution time.

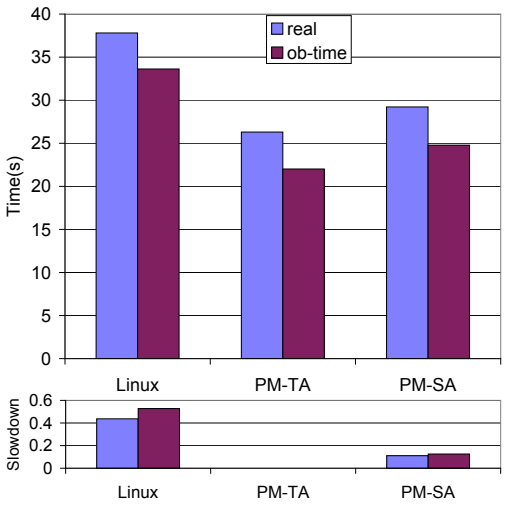


Figure 12. Diff performance results of comparing two different Linux source trees.

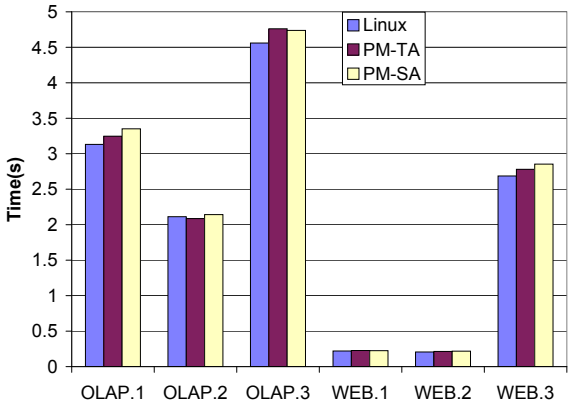


Figure 13. Cached storage performance results of SPC traces

Finally, we also test the case when multiple applications are executing concurrently, which generates more intermingled traffic patterns. However, due to the hardware environment change during the study, unlike the other tests, these experiments were done on Dell PowerEdge SC440 servers with 7.2K RPM SATA disk connected by Gigabit Ethernet. Therefore, we report normalized performance results. As shown in Figure 14, compared with Linux, *PM-TA* improves performance by 11.7% for *diff* and 15.8% for *tar*; compared with *PM-SA*, *PM-TA* improves performance by 7.7% for *diff* and 5.9% for *tar*.

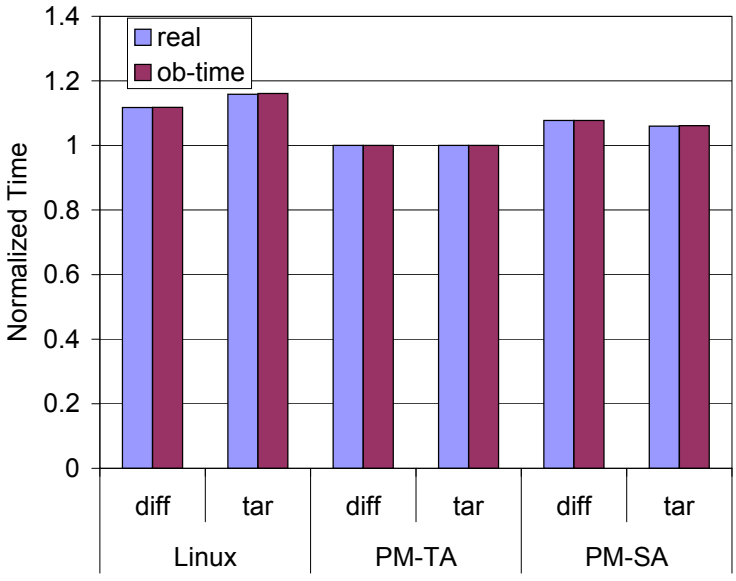


Figure 14. Performance results by running diff and tar concurrently.

Server CPU Usage Impact. The above experiments are all conducted with cold storage cache, i.e. disk blocks need to be brought in from disk. Although we argue that storage servers are increasingly facing *cold misses*, it is always interesting to

see how our scheme impacts the case when most of the accesses hit in the storage cache. Since our scheme uses more CPU cycles for prefetching management, we expect some performance degradation for this case because of the extra processing.

Figure 13 shows the performance results of the six SPC traces. The results show that the performance degradation of *PM-TA* is within 3% [5] in average compared with Linux.

Code Size The total engineering effort for implementing our scheme is small. With our current prototype, the patch file for Linux-2.6.16 stock kernel is only 224 lines. Reused library code from Linux source is 695 lines. Our PM library is only 1383 lines.

6.3.3 Summary

From the detailed experiment results, we show that STEP consistently outperforms the default Linux prefetching scheme and all the other common heuristics by both replaying SPC OLTP sequential traces and running frequently used I/O intensity applications. For random workload, our scheme has no noticeable effect, as shown by the SPC WebSearch traces. The CPU usage for the prefetching management is also tested to be minimal.

7 Conclusions

The technology trend has significantly increased DRAM memory's capacity with rapid falling price. The increased memory capacity on storage clients makes the large storage cache on emerging networked storage servers increasingly underutilized due to the weakened caching effectiveness for prolonged block reuse distance of data blocks. While research efforts have been focused on improving caching, the powerful computing capability and available disk bandwidth in addition to the large cache resources actually provide us good opportunities to improve storage system performance through prefetching.

In this implementation-based study, we have made a strong case for aggressive and intelligent prefetching in networked storage servers to improve application performance. Based on client access pattern detection and server internal knowledge of expensive storage operations such as disk seeks, we propose a new prefetching scheme – STEP, which generates prefetching requests aggressively for sequential access patterns and thrashing patterns using a new cost-benefit analysis model. Our implementation and in-depth evaluation of the design by replaying Storage Performance Council's (SPC) I/O traces and widely used Unix applications demonstrate that significant performance improvements are achieved with STEP.

In the future, we plan to extend our work to study the interaction of probability distribution function and different storage workload to enhance the categories of applications that can benefit from our scheme.

[5] Please note the OLTP SPC trace in Figure 3 is composed of lots of small sequence stream instead of seemingly a few due to scale of the graph.

References

[1] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *Proceedings of the 31st annual international symposium on Computer architecture (ISCA)*, page 176, 2004.

[2] J. Band. *The Storage Outlook: Managing to maintain growth*. Business Insights, 2003.

[3] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[5] Z. Chen, Y. Zhou, and K. Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the General Track: USENIX 2003 Annual Technical Conference (USENIX)*, pages 269–281, 2003.

[6] G. A. Gibson, J. S. Vitter, and J. Wilkes. Strategic directions in storage i/o issues in large-scale computing. *ACM Computing Survey*, 28(4):779–793, 1996.

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

[8] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 395–408, 2004.

[9] W. Hsu and A. J. Smith. The performance impact of i/o optimizations and disk improvements. *IBM J. Res. Dev.*, 48(2):255–289, 2004.

[10] S. Jiang, K. Davis, and X. Zhang. Coordinated multi-level buffer cache management with consistent access locality quantification. *IEEE Transactions on Computers*, 2007.

[11] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of the USENIX 1997 Annual Technical Conference (USENIX)*, pages 275–288, 1997.

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

[13] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: An idea whose time has come. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.

[14] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM symposium on Operating systems principles (SOSP)*, pages 79–95, 1995.

[15] Rea Hat Inc. Red Hat GFS Documentation. http://www.redhat.com/.

[16] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[17] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 225–238, 2005.

[18] E. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *Proceedings of the USENIX 1999 Annual Technical Conference (USENIX)*, pages 71–84, 1999.

[19] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[20] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, 1978.

[21] Storage Networking Industry Association. http://www.snia.org.

[22] Storage Networking Industry Association. iSCSI/iSER and SRP Protocols. http://www.snia.org.

[23] Storage Performance Council. SPC I/O Traces. http://www.storageperformance.org.

[24] M. Swift, Muthukaruppan, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2004.

[25] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[26] H. Wedekind and G. Zoerntlein. Prefetching in realtime database applications. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 215–226, 1986.

[27] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long, and R. C. Burns. Using multiple predictors to improve the accuracy of file access predictions. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS)*, pages 230–240, 2003.

[28] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track: USENIX 2002 Annual Technical Conference (USENIX)*, pages 161–175, 2002.

[29] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.