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' Web-Search traces.*

1 Introduction

Motivation. The fast growth of data resources has made networked storage servers very useful for the ease of mass on-line data storage and management. According to a survey of 260 businesses with more than 500 employees disclosed in 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 servers to facilitate efficient data accesses in such distributed environments.

In a typical storage area network (SAN), file system and storage devices are separated by networks. The file system in each storage client coordinates requests from multiple clients and accesses data blocks in storage devices connected to shared storage servers. Storage servers are resource-rich systems equipped with powerful CPUs and large memory as storage caches (usually $0.05\% - 0.2\%$ of the storage capacity in commercial storage systems [10]). In this networked storage architecture, there are at least two levels of buffer 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 at 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 (the number of distinct accesses between two consecutive accesses to the same block) at the L2 level due to the existence of the L1 buffer caches in clients, weakening the caching effectiveness of the L2 cache. With a continuous increase of memory capacity in clients, this reuse distance is expected to keep growing accordingly at the L2 level, which can lead to a lower utilization of storage caches. Our simulation results[1] using production-level storage system traces from Storage Performance Council (SPC) also demonstrate that the storage cache miss rate would not be reduced by keeping increasing cache capacity beyond a relatively small size. For two OLTP traces and one Web search trace, the miss rates of the three workloads flatten at

[1]Due to the space limit, the graph illustrating the results cannot be fitted in here. Please refer to [15] for details.

28%, 52%, and 63%, respectively, with a cache size of 512 MBytes.

In face of this problem, previous studies on storage caches have either focused on improving the replacement algorithms to adapt to the weakened temporal locality, such as MQ algorithm [27], or focused on making multi-level caches exclusive, such as demotion based placement [25], eviction-based placement [27], ULC [11], Karma [26] and X-RAY [1]. The effectiveness of these caching-based studies can be limited in practice, because cache hit ratios in storage servers would continue to decrease for two main reasons. First, as buffer cache sizes in clients increase, the access locality in server caches is further weakened. Second, the increasingly large server cache capacity is not well utilized due to mainly conducting passive caching in storage servers. Effective prefetching is largely a missing component in storage servers, which can potentially improve data access performance and storage resource utilization. However, prefetching decisions must be made correctly and timely based on access patterns and data access costs. Specifically, we target two performance- and cost- sensitive access patterns: sequential and thrashing access patterns in disks. Sequential access pattern from a single request sequence has been well handled by existing prefetching technique. In contrast, we target on detecting and prefetching sequential accesses from multiple sequences of requests that are common execution patterns in practice. Prefetching to-be accessed data with thrashing access patterns in disks would significantly reduce the delay of data accesses in networked storage servers. We will address several technical issues in order to enable detection and prefetching of these two important types of data accesses.

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. First, very limited studies have been done on prefetching in storage servers. Through the implementation based study, we have shown that significant performance improvements can be achieved by our proposed storage server prefetching scheme – STEP. Second, we have proposed a new cost-benefit analysis model using access history, disk access costs, and data access locality. Third, we have proposed a method to detect disk thrashing – a common phenomenon that severely degrades storage system performance, yet is not explicitly detected and prevented in existing systems. Finally, we have proposed a method to detect sequential accesses from pseudo-random access patterns received by storage servers due to the intermingled accesses from multiple clients.

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 improvements can be achieved with STEP. The comparisons with Linux's default prefetching scheme and several heuristics show that our system improves the performance of the OLTP workload by up to 94% with an average of 25%. Improvements with frequently used Unix applications are up to 53% with an average of 12%. As we expected, random workload performance has remained roughly unchanged compared with other schemes including Linux. In addition, the results also show that in average only 2.8% of additional CPU cycles are used in our system compared with the original Linux system.

2 Background and Related Work

I/O prefetching has been an active research topic since the early days of computer systems. As the technology trend allows CPU speed and memory/storage capacity to scale up, leaving I/O speed far behind to keep up with, I/O prefetching will continue to be important [16, 7].

Prefetching based on sequential access patterns is a conventional wisdom in file systems and database systems [22, 21]. 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 [16]. At the storage level, sequential prefetching is mostly device oriented, which operates only according to the physical data layout. Recently, SARC and AMP were proposed to manage prefetching memory for sequential prefetching in storage servers, which focused on balancing cache allocation between random and sequential ac-

cesses and among multiple sequential streams [8, 9]. Most recently, system DiskSeen has been proposed to study history based block-level sequential prefetching in stand-alone systems [6]. Unlike these studies, STEP prefetches for sequential accesses by maintaining access history using prefetch contexts, which provides more accurate information to make prefetch decisions.

Enhanced I/O interface allows accurate semantic information to be passed from applications to systems for prefetching. Cao *et al.* [5] used application-controlled prefetching and caching for file systems. Patterson *et al.* [17] proposed an enhanced API to pass information (hints) to operating system for prefetching and caching cost-benefit analysis.

Predictive prefetching approach does not require application-provided hints, which allows an easier deployment for a wider range of applications. At logical level, Brown *et al.* [4] used locality analysis to generate prefetch requests with the assistance from compiler. By tracing file access relationship, Lei *et al.* [13] used file access tree and dynamic pattern matching for file prefetching. At block-level, Li *et al.* [14] used data mining techniques to find block correlation on storage server to direct prefetching.

Due to the semantic gap for storage servers [1], prefetching for storage servers must take a predictive approach using only block-level access history. Focusing on critical performance bottlenecks, STEP targets frequently used and high-cost access patterns: sequential access patterns and disk thrashing patterns. Based on a new cost-benefit analysis model, STEP issues aggressive yet cost-effective prefetch requests.

3 Detection-Based Prefetching Management

STEP is based on sequentiality and disk thrashing detection. In this paper, we define a block address range with good spatial locality on disk as a *neighborhood*. When accesses to a neighborhood are significantly interleaved with accesses to other neighborhoods, we call it a *thrashing* scenario. Sequentiality describes the contiguousness of accessed data. An access pattern has so-called high-confidence sequentiality if it has a long sequential access length and high *prefetch hit ratio* – the ratio at which prefetched blocks are actually demanded by clients and hit in the storage cache. In contrast, a sequential access pattern is of *low-confidence sequentiality* when it has short sequential access length or large length but low prefetch hit ratio. And we call the accesses within a sequential pattern a sequential stream.

3.1 Pattern Detections

Storage devices, including single disks and storage arrays, expose their storage capacity as a large linear block array to storage 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 [20, 12]. Therefore, file systems generally map logically sequential blocks to continuous LBAs to optimize access performance. This allows us to detect logical sequential accesses using LBA. At the same time, this spatial-locality-preserving property also enables us to use LBA to estimate disk access cost for thrashing detection.

3.1.1 Detecting Sequentiality

An effective way to detect 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.

In order to separate sequential streams from intermingled pseudo random request sequence, STEP uses a *Prefetch Context (PC)* to represent the run-time object associated with each detected sequential stream. Each PC includes attributes that describe this stream, such as the most recently requested and prefetched addresses, the total sequence length of the stream and the stream's recent prefetch hit ratio. A PC also keeps a small number of records for recent requests in 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. When a new request comes, the request address is compared against the most recent prefetch addresses of the existing streams to see if it extends any of them. If an extension 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. If the new request cannot extend any of the PCs, 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 number of PCs may be created and kept for non-sequential requests, which increases the overhead of locating relevant PCs and the memory consumption on data structures. To overcome the problem, we index the PCs 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 1, the algorithm for purging operates on three PC queues: *high-confidence*, *low-confidence* and *new* queues. 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 prefetch hit ratio of PCs in the *high-confidence* queue drops below a threshold, it is demoted to the *low-confidence* queue. Each queue is main-

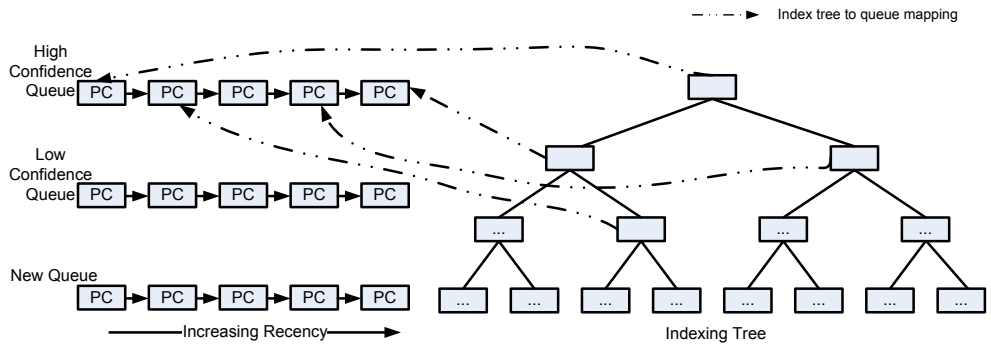


Figure 1. PC management data structures. Each PC is linked to one of the queues for purging based on recency and sequentiality. Meanwhile, it is also indexed by a balanced tree structure for efficient lookup.

tained 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*, so that PCs that are not sequential or not effective for prefetching are purged earlier than effective prefetch candidates.

To locate a PC, STEP uses a balanced index tree [24] 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 the number of PCs, the indexing and purging process involves only a small overhead.

3.1.2 Detecting Thrashing

For hard disks, disk seek operations between tracks are expensive. Ruemmler and Wilkes [19] described a model of modern disks, which breaks seek time into *speedup* time, *coast* time, *slowdown* time, and *settle* time. 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 of several milliseconds for current generation of technology [20]. Within this seek period, at least several tens of kilobytes sequential data can be read from a single drive. Therefore, if the 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 prefetch 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 for sequentiality detection is used by keeping track of sequential streams. In general, more flexible detection method other than the sequentiality detection can be used in this scenario.

For each neighborhood, we track the number of seeks and the sum of seek distances traveled to serve these accesses. In the aforementioned example, the distance traveled is $|46 - 20046| + |234546 - 46| + |47 - 234546| + |9848 - 48| = 498799$ blocks, and the number of seeks is 4. In real systems, the traced number of seeks and their distance do not strictly correspond to disk head movement due to low-level I/O scheduling and on-drive caching. However, since on-drive cache is small and I/O scheduler only queues a very small number of requests before issuing them, these tracked values are qualified estimations of the intensity of thrashing.

To formulate the intensity of thrashing in 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.

3.2 Prefetching Model

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

3.2.1 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 prefetch request includes the block transfer time, as well as, if any, the seek and rotation time for a disk head to commute to the desired prefetch 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 prefetch cost perceived by a client can be zero as long as the prefetch request is scheduled to be overlapped with the think time to get full prefetch benefit. Given the luxury of time in this case, cost-benefit analysis is less interesting, as we just need to predict 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 prefetched blocks follow fetched blocks. Second, disk drives perform internal prefetching into their small on-drive cache; immediate scheduling can pick up those blocks without accessing the disk media. Both of the above benefits are due to the locality benefits of immediacy. The disadvantage of immediate prefetching is the longer latency serving the fetch request due to possible request merges at disk level. However, this can be solved by issuing the prefetch request asynchronously after the fetch request is fulfilled. So we choose immediate prefetch scheduling in our model.

3.2.2 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 [10]; 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 (actual) request length.

I. 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 bytes for the next request, a seek is still required, if any, before reading the missing part due to requests interleaving on the storage server, thus prefetching only saves the time to transfer the already prefetched part, which is $RT \cdot Plen/Rlen$. In this case,

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

II. 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:

1. 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 usually is not wise in this case.

2. As shown in Case 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 benefit, 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, as the cost of multiple seeks can be avoided.

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

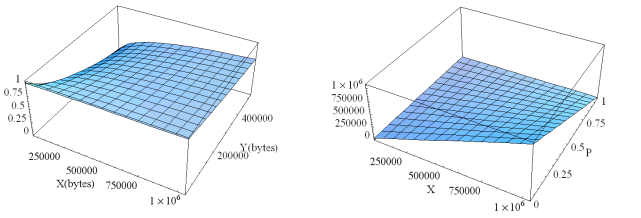
In summary, when the disk is busy serving requests from different locations, prefetching aggressively has a large potential for performance improvement. At the same time, successful prefetching requires a delicate balance between the probability of a future access and the prefetch length.

3.3 Prefetching Request Generation

With the formulas 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 prefetch length.

Estimation of Prefetch Request Length. The estimation of the next request is based on recent history kept in the relevant PC, which stores the last N demanded fetch request lengths (N is small, e.g. 4). Due to client-side prefetching/caching and the aggregate nature of block I/O, requests of a sequential stream received by storage servers are comparatively predictable. For example, requests are multiples of page size; the request length of long 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.

Prefetching Length Decision. Under two circumstances, we generate prefetching requests: a) detection of a high



(a)PDF for Sequential Workload (b)Reverse PDF for Random Workload

Figure 2. 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.

confidence sequential stream; b) detection of a thrashing condition. For a high confidence sequential stream, we use its PC's prefetch 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 3.2.2. 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 prefetch length y as inputs, 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 exponential functions to approximate *pdf* for mostly sequential workload, and linear functions to approximate *pdf* for mostly random workload. The coefficients of these functions are tunable and can be determined empirically and modified as run-time parameters when system workload changes. The *pdf*s used in our experiments are shown in Figure 2.

In order to generate the optimal prefetch length, we extend the single request formula in Section 3.2.2 to include multiple subsequent requests. We use PL to represent total expected prefetch 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 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 prefetched 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.

3.4 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 [5].

As shown earlier, cached storage server blocks are less likely to be reused in the near future. However, based on pattern detection and cost-benefit analysis, the prefetched data are of high probability to be beneficial. Considering a unified cache for both demanded and prefetched blocks, higher priority should be given to the prefetched blocks in the cache replacement decision. Instead of designing a completely new replacement algorithm to achieve the goal, we advise the cache management to adapt to the difference of demanded 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 3.3, we estimate the fetch time of the next request. For a request whose estimated fetched time is far into the future, we put the prefetch 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, in other words, 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.

4 Implementation in Linux Environment

Our prototype implementation is based on Linux 2.6.16 kernel. We have implemented a storage server using NBD – a network block protocol implementation distributed with Linux kernel and used in Redhat Global File System (GFS) [18].

The prefetching management system is implemented as a dynamic library on top of the raw disk device exposed by the NBD server. 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 user-space storage provider which exposes the storage capacity of the server. Our choice of dynamic library as the implementation layer makes the prefetching functionality transparent to any user-space storage servers for easy deployment.

On each block read request, our library creates or updates relevant PCs, then it generates corresponding prefetch

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 also implement a Red-Black tree [24] as the balance tree in our design to index the PCs. Each PC is linked to one of the three link lists representing the three PC queues: new, low-confidence and high-confidence. Each list is maintained in the LRU order to facilitate recency-based purging of victim PCs.

5 Performance Evaluation

5.1 Methodology

The evaluation is conducted by both 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. The OLTP trace is mainly sequential, while the web search trace is random. We filter the traces to retrieve read requests and divide resulting requests into smaller trace files based on the Application Specific Unit(ASU). For application benchmarks, we use three frequently used Linux applications: *cscope*, *tar*, and *diff*. *Cscope* is a source code browser. 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 for the kernel upon which STEP runs.

To demonstrate the effectiveness of our design, we compare STEP with Linux's prefetching scheme, as well as several other heuristics to illustrate the benefits. Linux has a prefetching mechanism for raw devices. For sequential accesses, it adaptively grows the prefetching window to a tunable upper bound, which is $128K$ by default [3]. We also compare with several other heuristics: *Conservative* prefetches the next estimated request only; *Benefit Bounded Risk* bounds the prefetch risk according to the advantage a stream has taken. More specifically, it tracks the total prefetch hit length of each PC and generates prefetch length using a percentage of this total benefit length; *Sequential Aggressive* prefetches aggressively for high confidence sequential streams using prefetch upper bound times the past prefetch hit ratio on a cache hit.

None of the three schemes detect thrashing and prefetch based on cost-benefit analysis. Thus clients only have the advantage of prefetching for interleaved sequential streams with different degrees.

5.2 Experiment Platform

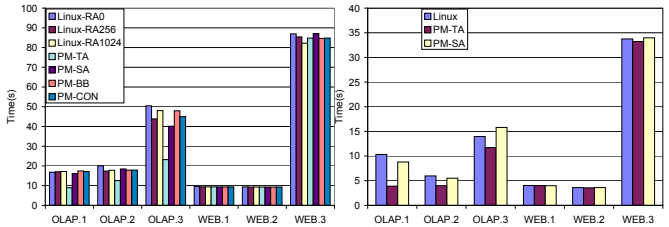
The experiments are conducted on an Intel Xeon 2.4GHz cluster. Each node has 1GB memory and 64 bit PCI-X 133 MHz bus. All nodes are connected to a 100Mbps fast Ethernet as well as an InfiniBand network. Each node is equipped with two 18GB 15K RPM SCSI disks and two 40GB 7.2K RPM, ATA/100 disks. We create 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 using different system configurations to exclude device-specific performance issues. The experiments are done with both local and networked storage scenarios 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. However, due to the space limit for the presentation, we can only show part of our experimental results, more details can be found at [15].

5.3 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 prefetching scheme, we test Linux with different prefetching thresholds. *RA#* represents the upper bound Linux uses for block-level prefetching in the kernel. For SPC trace tests, we randomly choose three of the processed trace files from each of the two workload types and use OLTP[1-3] and Web[1-3] as labels.

5.3.1 Local Storage Server Tests



(a) IDE disk (b) Local Software RAID

Figure 3. Local storage server performance test with different prefetching strategies and workloads

Figure 3 (a) illustrates the IDE disk results comparing the total execution time of different SPC traces. 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 the random Web-

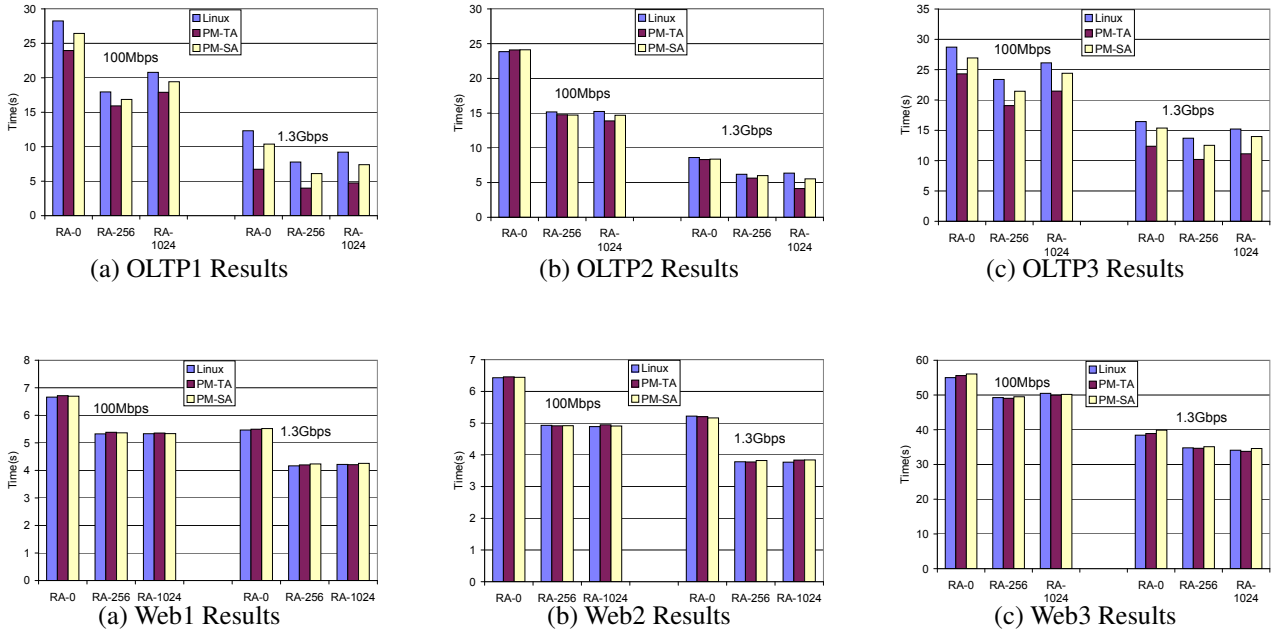


Figure 4. Network Storage Server Test Performance with Different Prefetching Strategies and Workloads

Search workload, all the schemes perform similarly with slowdown ratios ranging from -3.0% to 2.6%. As we expected, STEP significantly improves the performance of the interleaved sequential workload. At the same time, it has no noticeable effect on the random workload. Figure 3 (b) compares Linux, *PM-TA* and *PM-SA* (the best of the three heuristics) using SCSI software RAID – a faster storage technology. STEP's improvements range from 19% to 165% compared with the default Linux prefetching scheme in this case.

5.3.2 Network Storage Server Tests

SPC Trace. For this set of tests, we compare Linux, *PM-SA* and *PM-TA*. All three schemes use 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.

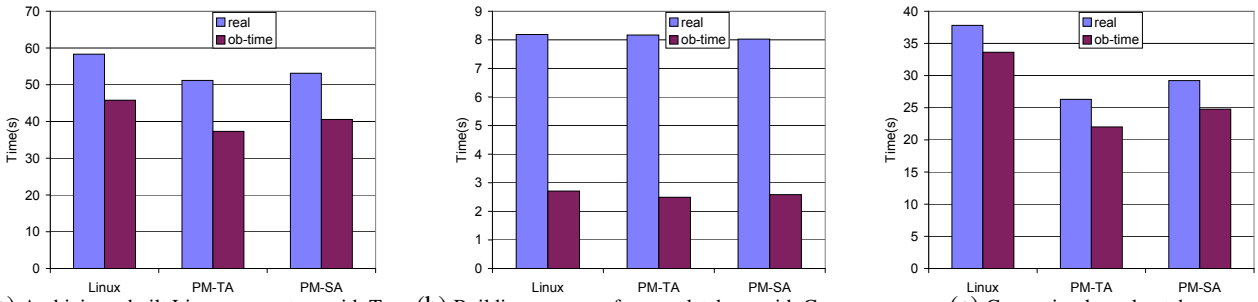
Figure 4 shows the execution time of the six SPC traces. In this figure, we also compare the performance with different network technologies, namely 100Mbps Fast Ethernet and 10Gbps InfiniBand network. InfiniBand supports different network protocol stacks with different performance and software compatibility 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 the network scenarios. Among the test results, the OLTP1 workload improvement with IPoIB and RA256 tops at 94% compared with Linux. The average improvements across all the schemes using two different networks is 25%, and average improvements with IPoIB only is 27%. 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. Compared with the performance improvements of 100Mbps Fast Ethernet, the overall performance of 1.3Gbps InfiniBand IPoIB improves more. Therefore, our prefetching scheme is expected to see a better performance improvement for sequential workload as the storage's bottleneck effect becomes more serious with faster network technologies. 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 kilobytes does not lead to better performance for all workloads. In two of the sequential workloads: OLTP1 and OLTP3, performance actually degrades noticeably ranging from 9% to 23%.

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*. We report both the real execution time and "out of box" time, i.e. the difference of real time minus user and system time, which represents the storage server access time.



(a) Archiving a built Linux source tree with Tar (b) Building a cross reference database with Cscope (c) Generating kernel patch

Figure 5. Application Performance with STEP

Figure 5 (a) 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 results show that the *PM-TA* scheme performs best among the three. The real execution time improves 14% and 3.8%, respectively compared with Linux and *PM-SA*. The "out of box" time improves 23% and 8.7%, respectively.

Figure 5 (b) shows the result of *cscope* building a cross reference database for a 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 the *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 5 (c) shows the result of *diff* comparing our modified Linux source tree and the original Linux source tree. In this benchmark, the major I/O operations are sequential file reads for these two source trees. In order to serve this source comparison, the disk head has to move back and forth to get the data from the disk. Aggressive prefetching is very effective for this test scenario. The results show that our *PM-TA* scheme performs 44% better than Linux and 11% better than *PM-SA*. For the "out of box" time, it improves the performance for 53% and 13%, 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.

Finally, we test the case in which 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. Compared with Linux, *PM-TA* improves performance by 12% for *diff* and 16% for *tar*; compared with *PM-SA*, *PM-TA* improves performance by 7.7% for *diff* and 5.9% for *tar*.

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 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.

The performance results of the six SPC traces results show that the performance degradation of *PM-TA* is within 2.7% 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 STEP library is 1383 lines.

5.3.3 Summary

From the detailed experiment results, we show that STEP consistently outperforms the default Linux prefetching scheme and 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 Web-Search traces. The CPU usage for the prefetching management is also tested to be minimal.

6 Conclusions

The technology trend has significantly increased DRAM memory's capacity with falling price. The increased memory capacity on the storage clients makes the large storage cache on networked storage servers increasingly underutilized due to the weakened caching effectiveness for prolonged block reuse distance of data blocks. The powerful computing capability and available disk bandwidth in addition to the large cache resources have provided a unique op-

portunity to improve storage system performance through prefetching.

In this study, we have made a strong case for aggressively prefetching critical access patterns in networked storage servers. Leveraging client access pattern detection and server's internal knowledge of expensive storage operations such as disk seeks, we propose a new prefetching scheme – STEP, which generates prefetch 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.

We plan to extend our work to study the interaction of probability distribution function and different storage workloads to enhance the categories of applications that can benefit from our scheme.

7 Acknowledgment

We thank the appreciations and constructive comments from the anonymous referees. This work is partially supported by the National Science Foundation under grants CNS-0405909 and CCF-0602152.

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] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[4] 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.

[5] 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.

[6] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of USENIX 2007 Annual Technical Conference*, Santa Clara, CA, 2007.

[7] 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.

[8] B. S. Gill and L. A. D. Bathen. AMP: Adaptive Multi-stream Prefetching in a Shared Cache. In *Proceedings of the Fifth USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 185–198, San Jose, CA, 2007.

[9] 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.

[10] 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.

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

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 101–114, 2005.

[13] 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.

[14] 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.

[15] S. Liang, S. Jiang, and X. Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. Technical Report OSU-CISRC-3/07-TR23, 2007.

[16] 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.

[17] 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.

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

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

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

[21] 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.

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

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

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

[25] 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.

[26] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel Cache . In *Proceedings of the Fifth USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 169–184, San Jose, CA, 2007.

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