

Memory Resource Allocation for File System Prefetching

From a Supply Chain Management Perspective

Zhe Zhang

Department of Computer Science
& Operations Research Program
North Carolina State University
zzhang3@ncsu.edu

Amit Kulkarni

Department of Computer Science
North Carolina State University
avkulkarni@gmail.com

Xiaosong Ma

Department of Computer Science
North Carolina State University
& Computer Science and
Mathematics Division
Oak Ridge National Laboratory
ma@csc.ncsu.edu

Yuanyuan Zhou

Department of Computer Science
University of Illinois at Urbana-Champaign
yyzhou@illinois.edu

Abstract

As an important technique to hide disk I/O latency, prefetching has been widely studied, and dynamic adaptive prefetching techniques have been deployed in diverse storage environments. However, two issues are not well addressed by previous research: (1) how to handle the prefetching resource allocation between concurrent sequential access streams with different request rates, and (2) how to coordinate prefetching at multiple levels in the data access path.

Interestingly, we found that these problems bear a strong resemblance to situations long studied in the field of *supply chain management (SCM)*, used by retailers such as Wal-Mart. In this paper, we demonstrate how to perform the problem mapping and then apply SCM principles in practice, particularly from the branch of *inventory theory*, to improve data prefetching performance in storage systems. More specifically, we applied (1) two SCM policies to dynamically configure the sequential prefetching parameters, and (2) an SCM solution to correct the access pattern information distortion in multi-level prefetching. We implemented these SCM-based strategies in the Linux kernel prefetching algorithm and a multi-level storage simulator, and evaluated the performance with three types of work-

loads. The results indicate that the SCM approaches are able to generate up to a 55.0% of performance improvement for a real-world server workload benchmark, and up to 33.3% for a combination of Linux I/O-intensive applications.

Categories and Subject Descriptors D.4.3 [Operating Systems]: File System Management

General Terms Design, Performance

Keywords Memory Cache, Prefetching, Supply Chain Management

1. Introduction

Prefetching is an important technique widely used in computer systems. In storage systems, single- or multi-level alike, it reduces the visible disk I/O costs by exploiting spatial localities in applications' access patterns.

Despite the plethora of studies done on data prefetching for storage systems, certain fundamental issues have not been well addressed. In particular, currently storage systems face the increased scale in two dimensions.

First, with the advances in server configurations, especially due to the emergence of multi-core/many-core nodes, the level of request concurrency is expected to continue to rise for both server and personal computing workloads. Existing adaptive prefetching algorithms focus on identifying the sequentiality versus randomness in the application access pattern, as well as identifying individual sequential access streams. Such analysis helps manage space allocation between prefetching and demand paging, and allows per-stream prefetching aggressiveness control [Gill 2005, Liang 2007].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04...\$5.00

However, existing approaches seldom explicitly consider the application request rates, in particular the relative access speed of multiple sequential streams, in their runtime decision making for dynamic prefetching. This may cause the loss of important timing information crucial to effective prefetching and balanced resource allocation. In most cases, the same prefetching settings will be applied to slow and fast streams, as long as they display the same sequentiality in accesses. Fast streams may not be able to have sufficient prefetched blocks to catch up with the application's pace, while slow streams may have many blocks unconsumed sitting in the buffer cache. To make it worse, the faster streams often have shorter computation periods to overlap with the I/O requests even when the program does I/O asynchronously, making the penalty directly visible to end users.

Second, the I/O stack is becoming increasingly deep with multiple layers of servers and their buffer caches, including database buffer caches, file system caches, storage server caches, etc. In multi-level storage systems, the caching/prefetching strategies working at an upper level may alter the application access behavior and mislead lower levels to prefetch too aggressively or conservatively [Yadgar 2008, Zhang 2008].

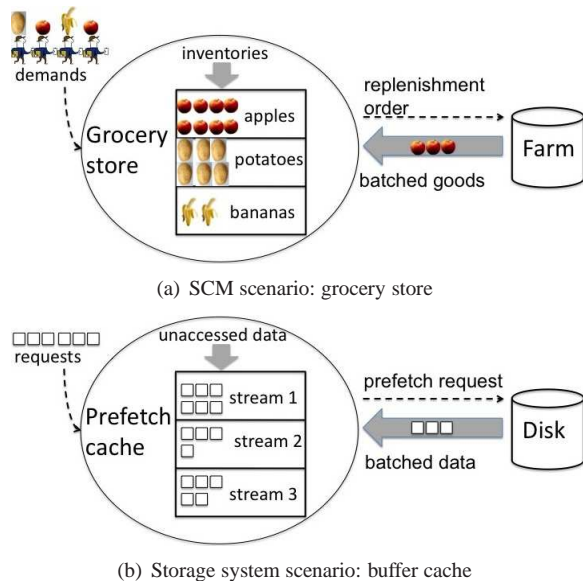


Figure 1. Similarities in SCM and storage system problems

Interestingly, we discovered that the above unaddressed data prefetching issues share many common properties with another field of study with a history of over half a century - *supply chain management (SCM)*. This is illustrated in Figure 1, where we see the one-to-one correspondence between SCM and dynamic data prefetching problems. Basically, in both contexts, items are retrieved to better serve *anticipated* customer requests, based on observations on the past and current request behavior. Correct and timely fetching of the predicted items will greatly enhance the cus-

tomers perceived system performance, while mistakes (such as stocking up the wrong items or performing the fetching too eagerly/leisurely) can waste precious resources in terms of storage/inventory and data/goods transportation capacity.

In this paper, we present a fresh approach to improving prefetching: borrowing and extending ideas from SCM theory and practice to enhance data prefetching algorithms deployed in modern storage systems. Our major contributions are summarized below:

1. We identified the mapping between data prefetching and supply chain management problems, and proposed the novel application of SCM principles to controlling adaptive prefetching parameters.
2. We performed dynamic prefetching by explicitly measuring level and deviation of per-stream access rates and conducting resource allocation accordingly, which to our best knowledge have not been done previously.¹
3. We implemented the SCM-based algorithms in the Linux kernel, as well as a verified multi-level storage server simulator, and evaluated their performance with Linux I/O intensive applications and real-world server workload traces. Our results indicate that SCM solutions generate considerable performance improvements over existing prefetching algorithms.

The rest of the paper is organized as follows. Section 2 provides background discussions in both SCM and prefetching. Section 3 presents the mapping of a prefetching problem in modern storage systems to an SCM problem, so that SCM solutions can be applied. In Section 4 we discuss how to use SCM policies in data prefetching. Experimental methods and results are presented in Section 5. Finally, Section 6 concludes the paper.

2. Background

In this section, we give an overview on relevant studies in both fields. Section 2.2 also serves as a survey of related work on sequential prefetching.

2.1 Inventory Theory Overview

A *supply chain* consists of “all the parties involved, directly or indirectly, in fulfilling a customer request” [Chopra 2001]. In particular, SCM research is still very active today, partly due to the challenges and opportunities brought by the advances in IT technology and supply chain globalization.

While many researchers have been renovating SCM for the web era, in this research we exploit technology transfer in the other direction. The specific SCM area of interest is *inventory theory*, which studies how to efficiently manage

¹Rate-aware resource allocation has been applied in other subfields of computer systems research, such as the proportional share scheduling in real time systems [Jeffay 1998]. However, those existing techniques can not be directly applied in the prefetching context due to significant differences in the problem spaces.

the inventory for goods and satisfy customer demands with low costs, by carefully distributing limited resources.

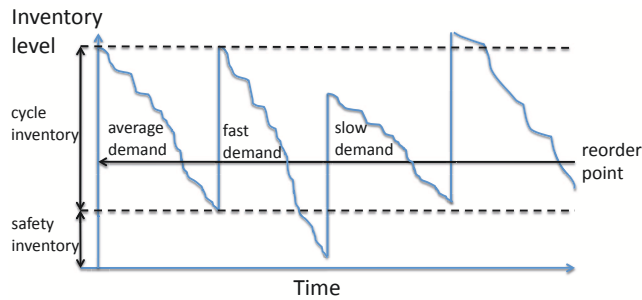


Figure 2. Sample inventory level fluctuation and associated terms

Let us give a brief introduction on inventory theory.² Figure 2 illustrates the periodic fluctuation in the inventory level for each type of goods in the grocery store example. The items are consumed by customers, so the inventory level will decrease. When the inventory level drops to the *reorder point*, another batch of items will be ordered, which will arrive after a certain amount of time, and boosts the inventory level. Much inventory research effort has been spent on configuring the two components of the desired inventory level: the *cycle inventory*, used to handle stable customer requests and achieve economic/efficient ordering and transportation, and the *safety inventory*, used to prepare for irregularity or burstiness in customer demands. Intuitively, if the customer demand for this type of goods is constant, then the safety inventory can be lowered to zero.

Similar to SCM and inventory theory, another mathematical tool originally developed for business management is *queuing theory* [Gross 1985]. Over the past decades, queuing theory has been applied in various fields in computer science with great success [Kleinrock 1976], especially in telecommunication, operating system scheduling and computer networks. While queueing models fit well with problems where multiple clients share certain critical resources (data channels or CPUs) in a time-splitting manner, inventory theory is able to model problems where a certain amount of space (such as CPU cache or memory) is the shared resource. In addition, the ordering operations in a supply chain and the prefetching operations in a storage system are system-level enhancements selectively placed on a subset of incoming requests, where queueing models do not fit as well. We believe that by following the successful example of queueing theory, SCM and inventory theory may also potentially become powerful tools in computer science research.

In Section 3, we describe detailed mapping of our target data prefetching problem to this inventory theory context.

²Throughout the paper, we adopt SCM terms and definitions from two widely used supply chain management textbooks [Chopra 2001, Silver 1998].

2.2 Dynamic Sequential Prefetching

Access pattern detection Many sophisticated prefetching algorithms have been proposed to prefetch data blocks in a non-sequential manner (e.g., with strided patterns) based on access pattern analysis and mining [Baer 1995, Dahlgren 1996, Fu 1991, Lee 1987, Li 2004, Soundararajan 2008]. However, most modern storage systems adopt *sequential prefetching* [Gill 2007; 2005, Smith 1982], which prefetches a number of blocks right next to the blocks requested, due to its simplicity and applicability to a wide range of application workloads [Gill 2007]. Therefore, the discussion in this paper is based on sequential prefetching, although our proposed approaches are portable to prefetching algorithms that establish correlations between data blocks.

Prefetching aggressiveness control For sequential prefetching, “what to prefetch” is given and the key problem is to decide “how much to prefetch” and “when to prefetch”, two metrics related to prefetching aggressiveness. In many systems today, this is determined at runtime through two prefetching parameters: *prefetch degree*, to control how much data to prefetch for each prefetching request, and *trigger distance*, to control how early to issue the next prefetch request. While there have been a number of studies on intelligent dynamic prefetching, by manipulating the trigger distance and/or the prefetch degree [Gill 2007, Li 2007], they focus more on identifying the sequentiality in accesses and do not measure or exploit the application request rate information.

One specific example of dynamic sequential prefetching algorithm used widely in systems today is the Linux kernel prefetching algorithm [Butt 2005], which we improved in this project using our SCM-based optimizations. This algorithm performs per-file access pattern analysis to determine whether a file is currently accessed sequentially, and doubles the prefetch degree when continuous sequential accesses are observed. With the 2.6 kernels, the prefetch degree and trigger distance are always set to equal values, with a default upper limit of 32 blocks. If the streak of sequential accesses is interrupted, the Linux algorithm will scale back the prefetching degree (by default to 3 blocks).

Another important related issue is how to allocate the shared memory cache space between prefetching and demand paging, both of which hide the I/O latency. There are a number of previous studies about managing prefetched data in a cache shared with demand paged data [Cao 1995, Kaplan 2002, Li 2005, Papathanasiou 2004, Patterson 1995]. Several other solutions alleviate the problem of *cache pollution* [P. Jain 2001, Reungsang 2001] by carefully bounding the space that prefetched data can use. As a recent example, the SARC algorithm deployed in IBM production systems [Gill 2005] uses two separate LRU queues for sequential and random data respectively, and adjusts their sizes according to the access pattern. Our proposed approach can be used in a complementary way, partitioning resource between

prefetching and demand paging through one of the above strategies, then among concurrent sequential access streams.

Resource allocation among multiple streams There has been a number of projects on caching/prefetching for workloads with multiple concurrent access streams [Barve 1997, Cao 1996, Gill 2007, Li 2007, Liang 2007, Tomkins 1997]. With two of them, namely LRU-SP [Cao 1996] and AMP [Gill 2007], although access rates are not measured explicitly, fast and slow streams will be treated differently through eviction and detection of request waiting events. However, LRU-SP is designed for cache replacement and does not change the prefetching behavior. The underlying model for AMP does consider the per-stream request rate, while its algorithm design and evaluation focus on throttling the overall prefetching aggressiveness without distinguishing between fast and slow streams. The TIPTOE algorithm [Tomkins 1997] estimates computation time between I/O requests. But it does so by relying on application hints, and at most two concurrent applications are used in the simulation-based evaluations. Our approach, in contrast, explicitly performs per-stream resource allocation based on level and deviation of access rates observed at runtime.

3. Problem Mapping

To apply SCM insights and techniques to data prefetching, we first need to map a prefetching problem in modern storage systems to an SCM problem. As mentioned in Section 1, we found striking similarities between the data prefetching and SCM scenarios, which allow us to construct the problem mapping. At the same time, there are challenging differences that prevent straight-forward or accurate mapping. These similarities and differences are discussed next, where we go through the key pairs of concepts.

3.1 Requests → Demand

The applications’ requests for data, mapped to blocks by the file systems, can be intuitively translated into customers’ demands for goods in an SCM context. Modern file and storage systems use dynamic algorithms that adjust the prefetching aggressiveness based on access pattern analysis. Similarly, in SCM, much effort has been spent on understanding and forecasting customer demands, where the demand patterns for a certain item are often modeled as time series of separate values. Advanced time series analysis techniques have been used to capture different characteristics of demand patterns [Silver 1998]. It is also natural for us to map each sequential access stream (*i.e.*, blocks requested by one client in a server system) to a type of goods in the SCM context, as shown in Figure 1. In both areas, access/demand pattern analysis and subsequent prefetching have been performed on per-stream or per-type basis.

The normal distribution and other distributions from the exponential family [Koopman 1936] are the mostly commonly observed customer demand patterns in SCM [Chopra

2001]. Most SCM techniques, including the algorithms to be discussed in section 4, are based on such demand models. In our analysis of real read-intensive applications’ I/O traces, we have found that their data access patterns share several important properties with the exponential family distributions, while at the same time possessing unique characteristics. Figure 3 shows the data access pattern of a representative stream from an SPC³ OLTP trace. We partitioned the access trace into many 10ms time windows. The *x* axis lists different demand rates (in terms of KBs of data requested in a window), while the height of the vertical bars illustrates the count of windows within the corresponding rate interval. In the figure we also show a fitting gamma distribution pattern. We can see that while the two patterns fit relatively well at and after gamma’s peak value at 7, the stream from the trace shows rather irregular patterns in the region with lower rates.

Fortunately, the medium and higher access rates are of more interest to most prefetching algorithms, since modern storage servers typically have high expected cache hit rates (around or above 50%) [Chen 2003, Zhou 2001], especially for sequential access patterns. More comprehensive and detailed analysis of real world applications’ data access patterns will potentially lead to enhanced algorithms and we consider it interesting future work.

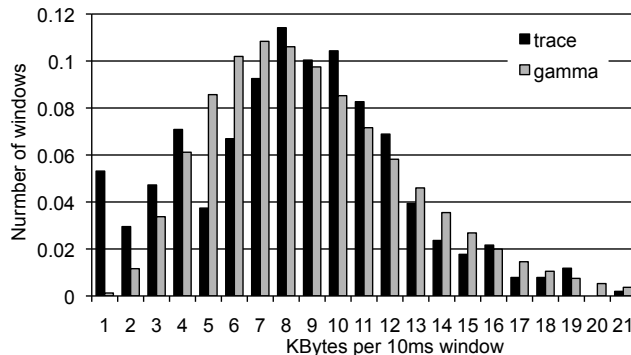


Figure 3. Distribution of application requested data per 10 milli-seconds

However, there is a fundamental challenge in mapping data requests to customer demands. In the SCM model, there are typically duplicate items of the same good. The customer demand specifies the type and quantity of goods needed (*e.g.*, “six apples”). In the data storage scenario, though, each data block is unique. However, when studying sequential prefetching we can achieve the mapping: assuming that correlations between data blocks have been correctly detected, any incoming data request can only refer to one possible block, which is “next to” the last accessed block in the detected logical correlation. Therefore, the unique data blocks belonging to the same access stream can be viewed as a sequence of “the next block”, hence becoming a group of identical items.

³ <http://traces.cs.umass.edu/index.php/Storage/Storage>

3.2 Access Time → Lead Time

In both storage and SCM scenarios, we need to consider the time to retrieve items, both to assess the potential benefit in hiding this latency, and to determine how early we should perform the fetch. In storage systems, this is the time to retrieve requested data from the disk, or a lower-level server in multi-layer architectures. Its SCM counterpart is the *lead time*, the time that elapses from the moment that an order is placed, till the requested items are physically on the shelf.

In general, we found the data access time in storage systems possess different characteristics than the lead time modeled in SCM studies. The data access cost depends on (1) the location (in terms of placement on disk) and the amount of data requested, and (2) the system load and disk head location. In contrast, SCM models typically assume the lead time as constant or i.i.d. (independent and identically-distributed) over time, insensitive to the demand volume or supplier workload. While bridging these differences for more accurate problem mapping makes interesting future work, in this paper, we report initial results achieved without extension of existing SCM theories to accommodate more sophisticated lead time modeling.

3.3 Batched Prefetching → Batched Replenishment

Contemporary file system prefetching algorithms fetch blocks in batches, rather than one at a time. Doing so takes advantage of the spatial locality in disk accesses, lowers the disk/network request overhead, and reduces energy consumption. In particular, due to the high start latency of disk accesses, a certain request size may be required to match the request rate. In SCM, intuitively, the same practice is adopted so that supply chain entities issue *batched replenishments* to lower entities or the factory. The motivation here is to exploit the *economics of scale*, examples of which include quantity discounts, sharing of per order costs, etc.

3.4 Prefetched Items → Inventory

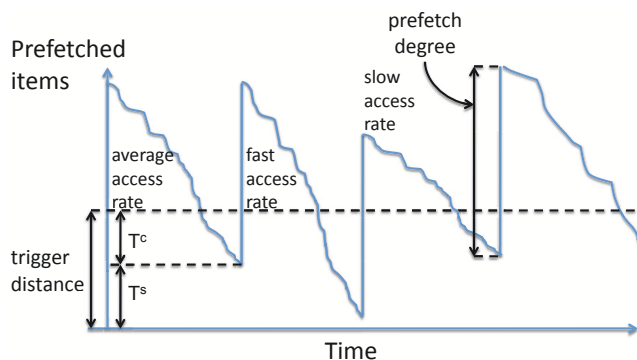


Figure 4. Inventory behavior in the prefetching context

Prefetching is about stocking up data blocks that are predicted to be soon accessed in the future. Many prefetching algorithms control the size of this set of fetched but not

accessed data by manipulating the trigger distance, which dictates how early the prefetch request should be issued. In SCM models, the set of prefetched goods corresponds to an extensively studied concept of *inventory*, the list of goods held available in stock.

We discovered that prefetching issues not studied explicitly in storage research have been discussed in inventory theory. More specifically, existing SCM literature provides insights on differentiating the functionality of the prefetched items in stock, into cycle inventory and safety inventory, as introduced in Section 2.1. We believe that the trigger distance is closely associated with the inventory notion and can be similarly decomposed.

Figure 4 shows how the dynamic prefetching parameters can be mapped to the inventory problem. Here the trigger distance, T , is equivalent to the reorder point in Figure 2. After each time the prefetch is triggered, the number of prefetched items unconsumed is expected to drop by a certain amount before the next batch of prefetched data arrives. This amount is determined by the average application request rate, as well as the fetch time (lead time) for this prefetch operation. It then corresponds to the cycle inventory and we denote this component of T as T^C . The other component, $T^S = T - T^C$, corresponds to the safety inventory. The prefetching degree can be intuitively mapped to the size of the batched replenishment. In Section 4, we describe how we leverage SCM policies to determine these prefetching parameters with multi-stream workloads.

One may argue that the prefetched items behave differently with grocery items in an inventory, in the sense that accessed data blocks still remain in the memory cache until replaced, while items bought by a customer will be removed from the inventory. Again, our mapping is achieved within the sequential prefetching context. When each strictly sequential stream is mapped to a type of goods, then after a prefetched block is accessed, this item can be viewed as “consumed” and will be taken off the inventory.

3.5 Multi-level Prefetching → Multi-echelon Inventory Control

With the rise of web-centric and service-oriented computing, storage systems nowadays are frequently composed of multiple levels, each equipped with a sizable memory cache. In such multi-level systems, prefetching is typically performed individually at each level to hide the latency of fetching data blocks from the lower layer. A supply chain normally has multiple levels of entities as well, such as *grocery store* → *distribution center* → *warehouse*, where each entity submits batched replenishment orders to its supplier. The branch of research in SCM on planning these inventory and replenishment behaviors is called *multi-echelon inventory control*. Our aforementioned problem mapping process can be repeated to construct a multi-echelon inventory problem.

In Section 4.2, we present an existing SCM analysis that lends us insight into a multi-level prefetching performance problem and inspires our solution.

4. SCM Algorithms for Data Prefetching

4.1 Access-Rate-Aware Prefetching

As mentioned earlier, existing prefetching algorithms, while identifying the concurrent sequential access streams in mixed or multi-request workloads, do not explicitly consider the individual access rates of these streams. With SCM practices, it is intuitive that if apples and bananas are consumed at very different speeds, it does not make sense to stock up the equal amount of them in store. The counterparts in replenishment policy for trigger distance and prefetch degree are *order point* and *order quantity*. A fixed quantity Q is ordered whenever the inventory level drops under the reorder point s . Inventory theory determines the reorder point s according to the rate of and the uncertainty in the incoming demands for a certain item. Next, we discuss the SCM optimization goal and introduce two commonly used policies for determining reorder points. Although there are more advanced SCM techniques, we choose these two relatively simple policies. This is due to their potential of being adopted in widely used operating systems such as Linux, considering several factors, including runtime overhead, implementation complexity, and capacity to work for a large range of applications. We then apply them to dynamically configuring the prefetching trigger distance. In particular, we focus on the problem of concurrent, long sequential access streams with different access speeds. Such scenarios are common in personal computing, web servers, multimedia servers, and scientific data centers.

4.1.1 Performance Metrics and Objective

In the prefetching and caching environment, the cache hit ratio is one of the most important factors contributing to the overall I/O performance, measured either in request response time or throughput.

In SCM the corresponding concept is *fill rate*, which is defined to be the fraction of demand that is satisfied from product in inventory [Chopra 2001]. Given the customer demand distribution and the replenishment policy, the fill rate, FR , can be estimated as:

$$FR = 1 - \frac{ESC}{Q}$$

In the above ESC denotes *Expected Shortage per Cycle* (the average number of unsatisfied customer demands in each replenishment cycle), and Q is the order quantity (prefetching degree in the storage context). SCM studies have shown that the optimization of FR is a very challenging problem [Silver 1998], and two heuristics policies have been widely adopted, as discussed in the following sections.

Notice that the ultimate objective of the policies is to optimize the overall system performance, a common goal in file

and storage system research, especially with adaptive techniques [Li 2005]. The policies can be extended in the future to consider other objectives or constraints, such as fairness across streams and real time requirements. Further, our current policies mainly focus on improving the performance of pure sequential access patterns, which are common and important in today's servers and data centers, running applications such as video-on-demand (VOD) and scientific data visualization.

4.1.2 Equal Time Supplies

The first policy, *Equal Time Supplies (ETS)*, is an intuitive, widely used approach: A large U.S.-based international consulting firm estimates that 80% ~ 90% of its clients use this policy for setting safety inventory levels [Silver 1998]. It is applied after the total amount of safety inventory for an SCM entity has been determined, e.g., by a grocery store's physical storage space. Given this total inventory level, with ETS, the individual safety inventories are set so that each type of goods has an equal time period's worth of supply. In other words, the safety inventory level for individual types of goods is set proportional to their demand rates.

The same observation can be applied to application data request rates. The "total inventory level" in this case, corresponds to the overall prefetching aggressiveness. There are existing algorithms, such as SARC [Gill 2005], that adjust the cache capacity and bandwidth resources between sequential accesses (prefetching) and random accesses (demand paging), according to runtime access pattern analysis. They can be leveraged to determine the total inventory level.

With the given total inventory level set, our prefetching algorithm will determine the desired trigger distance for each sequential stream. Its goal is to maintain a certain overall prefetching aggressiveness, in terms of T_{total} , the sum of the trigger distances for all concurrent sequential streams. Suppose at a given time, there are n concurrent access streams in the system, each with an access rate of R_i . Based on the ETS policy, the safety inventory component of the trigger distance, T_S , is proportional to the per-stream access rate:

$$\forall i, \frac{T_i^S}{R_i} = C_1$$

As can be seen in Figure 4, the cycle inventory component, T_C , is determined by how much of the prefetched items will be consumed during the time to complete the next prefetch order. As most adaptive prefetching algorithms have a fixed upper limit for the batch size (prefetching degree), all pure sequential streams will be able to reach this upper limit after a short ramp-up stage. So the time to prefetch each batch of data blocks can be assumed to be equal across all streams. Therefore, the expected amount of data to be requested while the batched prefetch request is in-flight, is

then proportional to the data request rate as well:

$$\forall i, \frac{T_i^C}{R_i} = C_2$$

Consequently, the individual trigger distance should be proportional to the request rate according to the ETS policy. We can calculate the ETS-based trigger distance T_i as

$$T_i = T_i^S + T_i^C = \frac{R_i}{\sum_{1 \leq i \leq n} R_i} \cdot T_{total}$$

4.1.3 Equal Safety Factors

As discovered by SCM researchers, one significant limitation of the ETS policy is that it does not consider the difference in the demand uncertainties of the streams. It works best when the customer demand for each type of good remains stable, which is often not the case in SCM or storage scenarios.

The *Equal Safety Factors (ESF)* policy addresses this limitation by setting the safety inventory level for each type of goods proportional to the standard deviations of their requests. The intuition is that those streams with the highest uncertainty in demands should receive a higher safety inventory level. With this approach, we first calculate the cycle inventory component of the per-stream trigger distance explicitly using the stream access rates. Suppose the time to fetch a batch of blocks using the maximum prefetch degree is $t_{prefetch}$, we have

$$T_i^C = R_i \cdot t_{prefetch}$$

Then the per-stream safety inventory component is allocated from the total safety inventory level proportionally, considering the standard deviation of access rate rather than the average rate. Suppose the standard deviation in access rate for stream i is σ_i , we have

$$T_i^S = \frac{\sigma_i}{\sum_{1 \leq i \leq n} \sigma_i} \cdot (T_{total} - \sum_{1 \leq i \leq n} T_i^C)$$

Finally, for each stream i , the trigger distance is again

$$T_i = T_i^S + T_i^C$$

4.1.4 Implementing ETS/ESF in the Linux Kernel

We implemented the runtime trigger distance configuration algorithms described in Section 4.1.2 and 4.1.3 in the Linux 2.6.18 kernel. As mentioned earlier, in Linux 2.6 kernels the default value for both the prefetch degree and the trigger distance is 32 pages. Since we assume that the overall inventory level for prefetching is determined using other algorithms and is beyond this scope of this research, in our implementation we try to maintain the same amount of memory consumption as the baseline Linux prefetching algorithm.

In Section 5, we show that by constraining the prefetching memory use, our algorithms have a rather small impact on concurrent random accesses though they target sequential streams.

With a trigger distance of T and a prefetching degree of P , the average per-stream inventory level (the total number of unconsumed prefetched blocks) is $((T + (T + P))/2 = T + P/2$. Therefore increasing the trigger distance by one is twice as expensive as doing the same with the prefetching degree, in terms of memory space. For our ETS/ESF algorithms, we start with a setting where $T = P/2$, for a balanced allocation between the two parameters (which is also adopted by another existing algorithm [Gill 2007]). To maintain the same average inventory level as the Linux prefetching default parameters, we get

$$T + P/2 = P/2 + P/2 = 32 + 32/2 = 48$$

This gives us the default ETS/ESF trigger distance, $T_{default}$, of 24, and the maximum prefetching degree of 48. A lower limit for trigger distances is set to 4. In our evaluation, we compared our SCM-based strategies with both the original Linux prefetching algorithm (32-32) and the algorithm with modified parameters (24-48).

4.2 Reducing Information Distortion in Multi-level Prefetching

Modern server systems often have a multi-level architecture, employing layered servers for application/web, database, and storage services [Chen 2005]. Meanwhile, access-pattern-driven optimization is also done at many levels, from the front-end server to the disk scheduler. The problem is that in these systems, the original access pattern is altered by layers of caching and prefetching, and the observed access pattern deviates from the true application request pattern more and more as requests go down the hierarchy. Compared with the cache replacement problem, where filtered accesses and exclusive caching [Wong 2002] are desirable features, for prefetching, the batched, long, sequential accesses induced by a higher-level prefetching operation is often misleading to lower-level prefetching algorithms.

While this information distortion problem has recently been addressed in storage systems [Zhang 2008], the similar situation in supply chains has been studied for much longer. The term *bullwhip effect* is used to describe the phenomenon of variations in demand getting amplified as we go down the supply chain, away from the end customers. One famous illustration of the bullwhip effect is the *Beer Game* [mit-beer], developed by the MIT Sloan School of Management in the 1960's, where players form a beer supply chain without communication other than receiving orders. It demonstrated the frustrating nature of individual supply chain entity management, with each entity trying to minimize back orders and inventory levels merely based on the demand observation.

In this paper, we exploit one straight-forward solution from SCM: extending the visibility of customer demand as far as possible. This approach has been most successfully adopted by the Wal-Mart distribution system [Andel 1996], generating better inventory positioning and lower costs by enhancing the customer demand visibility. At Wal-Mart, the point-of-sale (POS) data are collected from the cash registers, processed by corporate headquarters a few times a day, and used to guide its distribution center in interacting with both the stores and the suppliers.

We applied the above approach to attack the bullwhip effect in multi-level prefetching, by having an upper-level server explicitly label the appended prefetching requests in each request to its direct supplier, a lower-level server. Most dynamic prefetching algorithms will adjust the prefetch degree and trigger distance on each block access to the cache, hit or miss. With the SCM-based solution, these actions are only taken for those blocks in the original application request. While a prefetched block appended to the application request may still trigger other actions at the lower level, such as issuing another prefetch request, the lower-level trigger distance and prefetch degree will not be affected.

We implemented the above mechanism in a two-level storage simulator, where several existing prefetching algorithms have been implemented at each level. In Section 5.3, we present a case study where this simple approach is able to correct a severe performance problem caused by inter-layer information distortion.

Regarding the overhead of such a mechanism, we consider I/O interface change rather small and comparable to that of existing inter-layer coordination schemes (e.g., [Gill 2008]), and the size of the extra data added to the communication should not be a problem considering the disk system is typically the performance bottleneck [Chen 2005].

5. Performance Evaluation

We implemented our SCM-based optimizations in the Linux 2.6.18 kernel (see Section 2.2) and a multi-level storage simulator (described below in Section 5.1). In the rest of the section, we first give an overview of our test platforms and workloads, then report experiment results.

5.1 Platform and Workload Overview

Platforms We tested our modified Linux kernel for single-level prefetching on a Dell PowerEdge 2950 III server, with two 2.33GHz quad-core Intel Xeon E5410 processors, 16GB main memory, and integrated RAID 5 with six 146 GB 15K RPM SAS 3Gbps hard drives.

For multi-level prefetching, we used a simulator to perform trace-driven evaluation. This multi-level cache simulator has been used and verified by a number of previous studies [Chen 2005; 2003, Zhang 2008, Zhou 2001, Zhu 2005], and is connected to the widely used disk simulator DiskSim [Ganger 1999]. Four existing prefetching algo-

rithms (Fixed ReadAhead (RA) [Smith 1982], AMP [Gill 2007], SARC [Gill 2005], and Linux kernel prefetching [Butt 2005]) are implemented in this simulator to work independently at each level.

Workloads We evaluate our proposed approach with three types of workloads:

1. Synthetic benchmarks. These benchmarks are created to include a group of sequential access streams, where we can easily control the per-stream access rate and rate deviation, as well as the number of concurrent streams.
2. Real Linux file transfer applications (*cp* and *scp*). For *cp* we use both local and remote (NFS) destinations. For *scp* destinations we use workstations within the same campus of our server, as well as a set of geographically dispersed sites from the PlanetLab [Chun 2003].
3. Two types of server benchmarks. The first is an HTTP web server workload. At the server side we run Apache on our Dell server with modified Linux kernel. On the client side we use *httperf-0.9.0* [Mosberger 1998] to generate 20 clients (again running on PlanetLab nodes) that request file downloads with different bandwidth. The second is a SPC2-VOD-like benchmark that we implemented according to SPC specifications⁴ to simulate video-on-demand workloads. In SPC2-VOD, each sequential stream has a fixed think time between consecutive I/O requests. To assess our algorithms' impact on concurrent random accesses, we also run a TPC-H trace collected on workstations at Purdue University [Butt 2005], which has highly random accesses.

Unless otherwise noted, all experiments are repeated 7 times and for each result (from our proposed approaches or baseline algorithms for comparison) we report the average after eliminating the best 2 and worst 2 results. This allows us to filter out "outliers" caused by other concurrent system activities (periodic system daemons, login attempts, etc.), which are observed to happen in similar ways when we switch between prefetching algorithms.

5.2 Concurrent Streams with Varied Access Rates

5.2.1 Synthetic Benchmarks

As our research in prefetching for concurrent streams focus on sequential streams, we first use synthetic workloads to examine the behavior of the original and SCM-based Linux prefetching algorithm with controllable stream access characteristics. In the synthetic benchmarks used in this section, we create multiple concurrent streams which issue pure sequential 4KB read requests, each accessing a different file of 1GB size (which is larger than any per-stream access footprint). The streams are fully overlapped with each other, starting together and running for the same length of time.

In these experiments, the system response time under four different policies are evaluated: "Linux 32-32" for original

⁴ <http://www.storageperformance.org/specs>

Linux kernel prefetching algorithm with both prefetch degree and trigger distance being 32 pages, “Linux 24-48” for the policy which increases prefetch degree to 48 and decreases trigger distance to 24, plus the ETS and ESF algorithms.

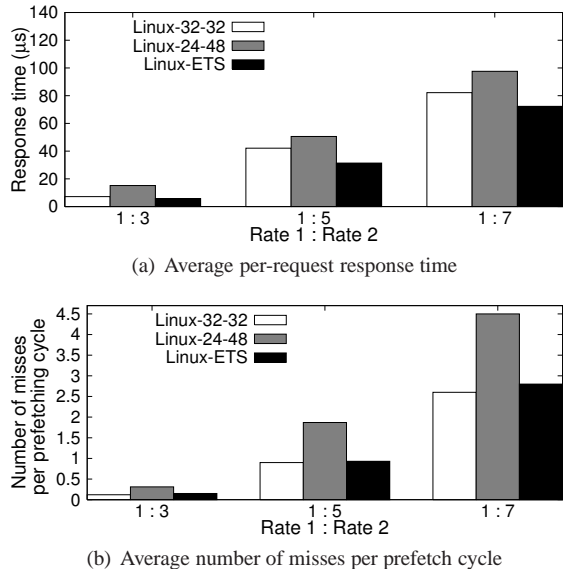


Figure 5. Two streams with growing rate differences

First, to evaluate how the ETS algorithm responds to different levels of access rate contrasts between streams, we generate a set of traces, each containing two streams with a pre-specified rate difference. The rate of the first stream is fixed at 1000 requests/s, while that of the second stream is set to 3000, 5000 and 7000 requests/s in 3 experiments respectively. Figure 5(a) shows the results in terms of the average request response time (for all requests in both streams). Figure 5(b) portrays the average number of cache misses per prefetching cycle (*ESC*).

With 24-48, the total number of prefetching cycles is decreased by a third from 32-32, by enlarging the prefetching degree limit from 32 to 48. However, the overall response time is worse due to the disproportionately higher number of misses per cycle, due to the shorter and uniformly set trigger distance. Although ETS has the same number of prefetching cycles as 24-48, it efficiently allocates the total amount of trigger distance between the two streams. Thus it is able to control the number of misses per cycle to be very close to 32-32 and improves the request response time by 19.0% on average, with a maximum enhancement of 25.4%. The trigger distances used by ETS for the two streams (averaged over all prefetching cycles) are (13,35), (8,40), and (7,41), for the three different rate combinations respectively.

Next, we assess the effectiveness of the ESF algorithm by comparing it with ETS. with a different trace setting. Here again we have two streams, with the same mean access rates but different rate variances. The standard deviation of the

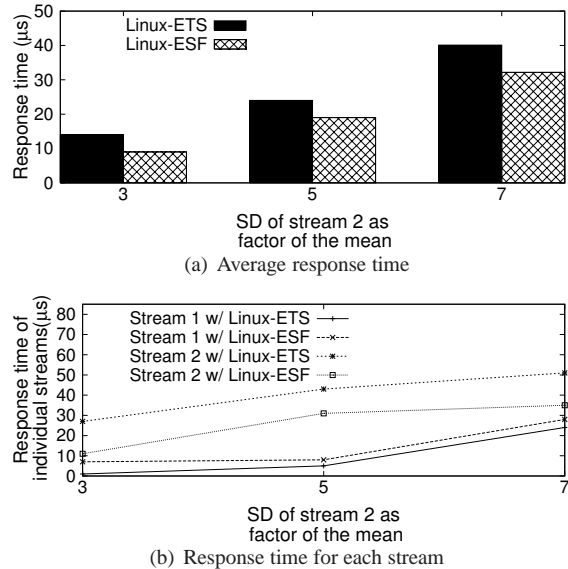


Figure 6. Two streams with growing rate variances

first stream is fixed at the square root of the mean rate, while for the second stream it is varied from the 3 times to 5 times and then to 7 times the mean value. Figure 6(a) shows the average per-request response time, while Figure 6(b) shows the response time of the two streams separately.

As expected, ESF does perform better than ETS with less stable access streams, winning by an average of 25.4% and up to 35.7%, due to its capability of adjusting the safety inventory level according to the observed variance in request rate. As can be seen from Figure 6(b), the unstable stream obtains a significant performance improvement under ESF, while the more stable one suffers small degradation.

However, in the rest of our results using real applications and server workloads, where the access rate behavior is less regular, we found that ESF is consistently outperformed by ETS. We believe the reason is related to our measurement of the rate deviation, which currently uses a simple sliding window average method. In our experiments we observe significant fluctuations of the measured deviation values. This will cause unstable trigger distance setting, hurt the performance, and possibly inject access rate variances.

5.2.2 Real Linux Applications

We then evaluate both ETS and ESF using a combination of local and remote Linux file transfer applications, each transferring multiple 128MB files. Among them, *cp-local* copies to a local file system, *cp-NFS* copies to an on-campus machine via NFS, and *scp1~scp6* copy to different PlanetLab nodes (with decreasing bandwidth). Here we measure the I/O throughput of individual streams as well as the aggregate system throughput, during a fixed time period where the concurrent streams fully overlap. This experimental environment mirrors a file system in a high performance com-

puting center where users frequently need to transfer job input/output files between the local and remote sites.

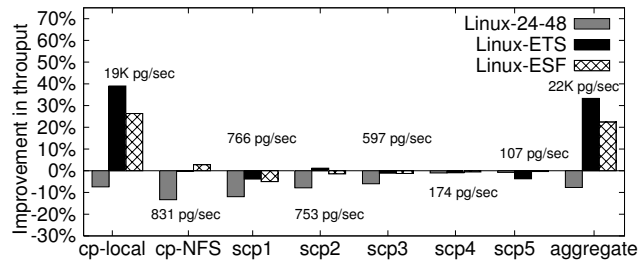


Figure 7. Throughput of file transfer applications

Figure 7 shows the improvement of three algorithms over the native 32-32 algorithm, whose per-stream and aggregate throughput numbers are marked above the bars. The aggregate throughput is improved by 33.3% with ETS and 22.4% with ESF, mainly due to the large performance gain for the fast *cp-local* stream (39.0% with ETS and 26.3% with ESF). Meanwhile, for streams *cp-NFS*~*scp6* the SCM algorithms do not have significant impact (within +/-5%). The much smaller trigger distances assigned to them by the SCM algorithms (close to lower limit) do not cause poor performance because in most cases they can satisfy the low access rates.

5.2.3 HTTP Web Server

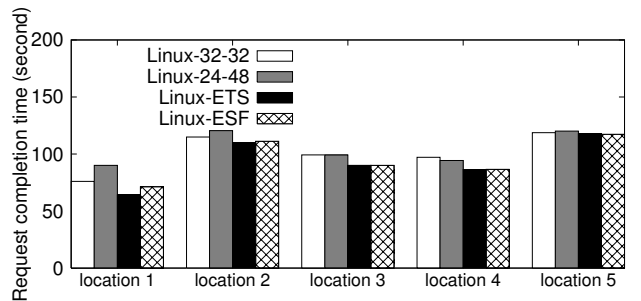


Figure 8. Request completion time for web clients

In this section we perform experiments on a HTTP web server handling file download requests from multiple clients. Our web clients are generated by *httperf* [Mosberger 1998], a web workload generator developed by HP labs and used by previous system research work [Abdelzaher 2002, Gupta 2008]. *Httpperf* is able to create a realistic environment where multiple concurrent clients connect to the server through the Internet and issue HTTP requests with specified patterns.

We use five PlanetLab nodes at different geographical locations, with decreasing connection bandwidth to the server. On each node we start four file download requests using *httperf*. Instead of system throughput, in this set of experiments we calculate the average request completion time for clients on each location, which is measured by *httperf* and is a more natural metric from a web client’s point of view. To make the download requests overlap with each other, we

let faster clients download larger files, so that all download requests have close execution time. This is a realistic scenario since web servers often recommend files with higher qualities to faster clients.

Figure 8 shows that the completion time of the fastest location is reduced by around 18% compared with the native 32-32 algorithm. The completion times for the slower clients are also reduced by 1% ~ 12% despite the shorter trigger distances assigned to them. This is because the earlier completion of the fast requests allow them to eventually work with more resources. The relative performance of ESF compared with ETS is similar as in section 5.2.2. For workloads where the access rates do not vary rapidly over time, ESF’s does not have an advantage in resource appropriation. On the other hand, the calculation overhead and unstable trigger distance caused by the measurement of standard deviation degrade its performance.

5.2.4 Server Benchmarks

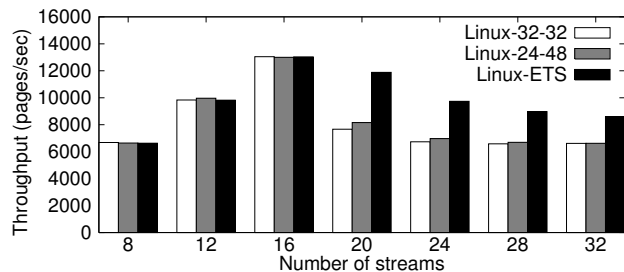


Figure 9. Throughput of SPC-2 VOD-like workload

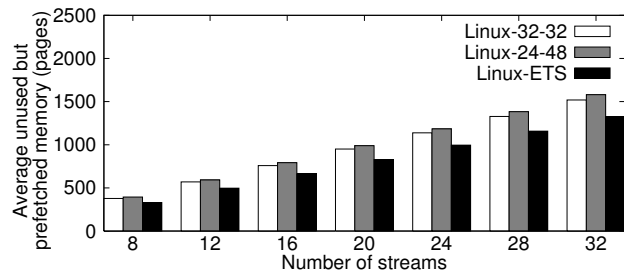


Figure 10. Memory usage of SPC-2 VOD-like workload

Figure 9 depicts the performance for an SPC2-VOD-like workload using different algorithms. Here a certain number of sequential streams access the server, each sending requests to a separate file. All streams have the same think time between consecutive I/O requests, which is set to 33 ms. To create streams with different access rates, we use four different per-stream request size levels (16KB, 32KB, 64KB, 128KB), simulating varying video qualities. In the figure we show the aggregate throughput with different numbers of streams, each randomly selects a request size level. Since the relative performance of ESF compared with ETS is similar to that observed in sections 5.2.2 and 5.2.3, we omit it in this

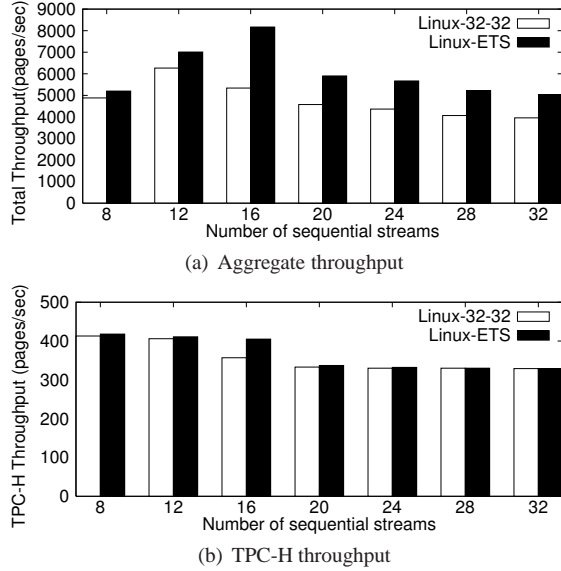


Figure 11. Throughput of mixed SPC-2 VOD-like and TPC-H workloads

section. From the results we can see that when the number of streams is small, all three algorithms have nearly the same performance. That is because with relatively idle disks and short disk access time, all streams can be satisfied. However, ETS is performing significantly better than the other two algorithms when there are more than 16 streams, and therefore fast streams need more aggressive prefetching. The average improvement of ETS over the native Linux algorithm is 20.1% and the peak improvement is 55.0%. The performance of 24-48, on the other hand, is almost the same with 32-32, due to the same reason as discussed in Section 5.2.1.

Also, in Figure 9 we plot the average per-stream memory usage by the three algorithms. As expected, the 32-32 and 24-48 algorithms have approximately the same memory usage level. ETS, in contrast, settles to a lower number although it is designed to maintain the same memory usage as the base algorithms. This is due to that the trigger distance needs to be bounded by the prefetching degree, and the fastest streams are not able to grow their trigger distance as indicated their access rates. Overall, this shows that ETS is able to generate a 20% performance enhancement while using about 13% less memory space.

Finally, we examine the impact of our SCM algorithms on random accesses that co-exist with sequential access streams, by running the above VOD workload concurrently with a TPC-H workload, which is composed of mostly random accesses. Figure 11 shows the aggregate throughput of both workloads and Figure 9 shows the throughput of TPC-H accesses. Since the relative performance of 24-48 compared with the other two algorithms is similar to that in Figure 9, we focus on the comparison between 32-32 and ETS here.

We can see that with concurrent random accesses, ETS still has significantly better aggregate throughput than the native 32-32 algorithm, with an average improvement of 26.1% and a maximum of 53.0%. The average improvement is slightly higher than that with SPC2-VOD alone, mainly because that with the additional TPC-H workload, the disk access time increases, which enables ETS to show performance benefits even for small number of streams.

Finally, we observe that the TPC-H throughput is increased slightly with ETS, by up to 13% and on average 2.6%. The impact of ETS on the performance of random accesses is twofold. On one hand, with a larger prefetch degree compared with 32-32, its disk traffic pattern is more favorable to random accesses. In addition, it uses a smaller amount of memory as we observe in Figure 10. On the other hand, the increased sequential access throughput causes a higher workload to the disks which are shared with random accesses. The observed change in TPC-H throughput is the result of these two counter acting factors.

5.3 Information Distortion in Multi-level Prefetching

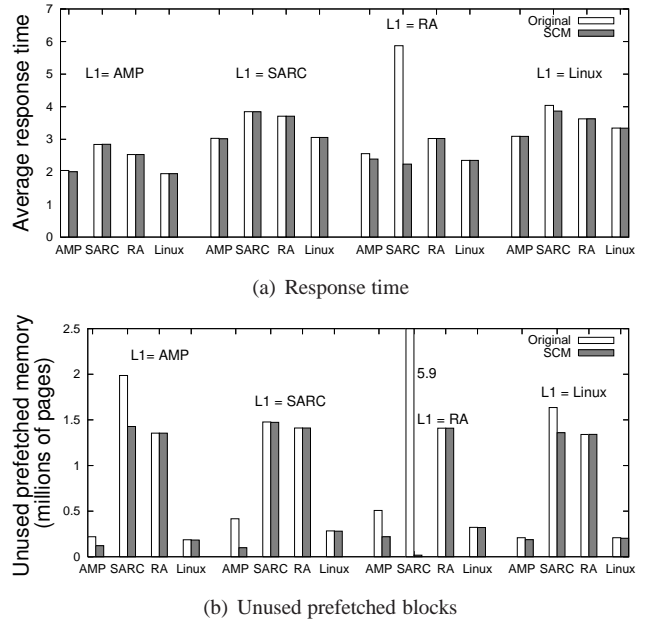


Figure 12. Multi-level prefetching with and without SCM approach

Here we present a case study where the SCM-based approach in extending customer demand visibility helped us to correct the bullwhip effect in multi-level data prefetching.

In this experiment, we tested all 16 combinations of the four prefetching algorithms implemented in the two-level simulator that we used, on the multi-application I/O trace collected by Purdue researchers [Butt 2005]. Figures 12(a) shows the average response time of trace requests resulted from the algorithm combinations, where the bars

are grouped by the L1 (upper-level) policy. Meanwhile, Figure 12(b) shows the number of data blocks that are fetched into the L2 (lower-level) cache but are evicted later without being used.

The white bars in the figures show the original algorithm performance and clearly one algorithm combination stands out: RA-SARC. After looking into the interplay between these two algorithms, we found the problems lies in the information distortion between the two levels. With this test, RA, the fixed read-ahead prefetching approach, uses a constant prefetching degree of 4. This means with every block request, the next 4 blocks will be prefetched, regardless of whether the access pattern is sequential. SARC, meanwhile, happens to detect the sequentiality of accesses with a default threshold of 4 contiguous blocks. Therefore, any application request, sequential or random alike, after passing through L1 will be perceived as sequential by L2 and encourage unnecessary prefetching.

SARC sequential threshold	3	4	5	6	7
Average response time (ms)	6.48	6.6	6.64	4.65	2.95

Table 1. RA-SARC performance under different SARC sequentiality detection thresholds

Table 1 further confirms this analysis. With the same RA configuration, the average response time drops back significantly when the SARC sequential access detection threshold is increased beyond 5. The reason that the threshold 6 produces a half-way improvement is because some of the random accesses cross the block border and access two blocks, and could still be detected as sequential with 4 additional blocks appended by L1 prefetching.

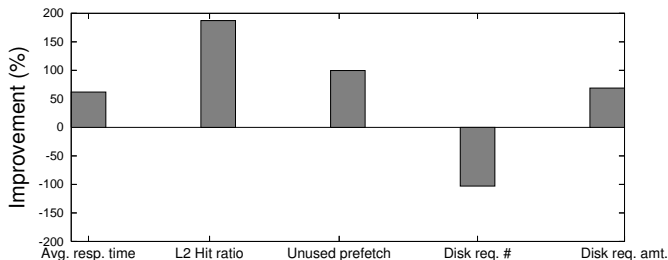


Figure 13. Zoomed-in study of the performance gain of SCM technique in the RA-SARC combination

Back in Figure 12(a), we illustrate the difference made by our SCM-based solution, where the original application requested blocks are labeled in the L1 requests to L2. The bullwhip effect in the RA-SARC combination has been removed, and SARC actually outperforms all other L2 algorithms when RA is employed at L1. To give a more detailed view into this 61.9% performance improvement, Figure 13 portrays the improvement generated by the SCM solution for the RA-SARC combination in four additional metrics.

The L2 cache hit ratio is improved by more than 180%, as a result of the reduced cache pollution, and the amount of unused prefetch is cut in half. Also, the total amount of data requested from the disk has also been significantly lowered. However, the total number of disk request roughly doubles, due to the much reduced L2 prefetching aggressiveness. We found that one drawback of the block-labeling approach here is that when L2 prefetching parameter adjustment is limited to “real requests” from the application, the L2 decision making will suffer from the loss of information when parts of application requests are masked from L2 due to L1 cache hits. In our future work, we plan to study the benefit-cost tradeoff in demand information propagation.

Even with this limitation, as can be seen in Figure 12(a) the SCM solution does not generate any negative performance impact to any algorithm combination, besides dramatically improving the RA-SARC performance. For several other algorithm combinations, it lowered the L2 prefetching aggressiveness significantly (thus saving disk bandwidth and energy), without hurting the request response time. When the L2 algorithm is RA or Linux, neither the average response time nor the unused prefetch is affected noticeably by the SCM solution. For RA, it is easy to explain as RA is oblivious to L1 access pattern. For Linux, it is because the sequentiality detection uses a more flexible window-based method, therefore the L2 prefetching is more resistant to L1 access pattern distortion or correction.

Finally, our results reveals that RA, a seemingly “dumb” algorithm not adopted by today’s commercial systems, actually does a better job as an L1 algorithm than more sophisticated strategies such as SARC and Linux. From an SCM perspective, we believe this is because in most cases, its fixed and rather small prefetching degree generates less information distortion at L2 with sequential access streams.

6. Conclusion

In this paper, we revealed that the data prefetching problem is closely mirrored in the business discipline of supply chain management (SCM). In particular, the inventory theory models can be used to draw insights into the prefetching resource allocation problem when concurrent application request streams possess different accessing speeds, as well as the access pattern information propagation problem in multi-level prefetching. Our experiments demonstrated that the corresponding inventory theory solutions, when applied to data prefetching, can successfully improve multi-stream I/O performance and correct certain information distortion problems.

Meanwhile, we have found that the complexity in disk access time modeling currently prevents certain SCM policies, such as Equal Safety Factor, from working well consistently. The simpler Equal Time Supplies policy, on the other hand, generates rather stable improvement and can easily be plugged into existing dynamic sequential prefetching

algorithms. In our future work, we plan to investigate more sophisticated SCM models, as well as efficient online I/O performance monitoring techniques, to better understand the inventory phenomena in prefetching scenarios and develop more effective performance solutions.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. Also, we sincerely appreciate the wonderful help and guidance provided by our shepherd, George Candea, in the process of revising this paper. We are thankful to Kai Shen and Chuanpeng Li for sharing their Linux kernel patch, which gave us helpful insights on our own implementations, as well as Kyuhyung Lee for helping us set up the multi-level prefetching experiments. We also would like to thank Frank Mueller for his helpful comments regarding related work.

This research is sponsored by a collaborative NSF HECURA grant (CCF-0621470). It was also sponsored in part by a DOE ECPI Award (DE-FG02-05ER25685), two NSF CAREER Awards (CNS-0546301 and CNS-0347854), and NSF grant CNS-0615372. In addition, the work is supported by Xiaosong Ma's joint appointment between NCSU and ORNL.

References

- [Abdelzaher 2002] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002. ISSN 1045-9219.
- [Andel 1996] T. Andel. Manage inventory, own information. *Transportation and Distribution*, 1996.
- [Baer 1995] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995. ISSN 0018-9340.
- [Barve 1997] Rakesh Barve, Mahesh Kallahalla, Peter J. Varman, and Jeffrey Scott Vitter. Competitive parallel disk prefetching and buffer management. In *IOPADS '97: Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 47–56, New York, NY, USA, 1997. ACM. ISBN 0-89791-966-1.
- [Butt 2005] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, 2005.
- [Cao 1996] Pei Cao. *Application-controlled file caching and prefetching*. PhD thesis, Princeton, NJ, USA, 1996.
- [Cao 1995] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, 1995. ISSN 0163-5999.
- [Chen 2005] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005.
- [Chen 2003] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, Jun 2003. ISBN 1-931971-10-2.
- [Chopra 2001] S. Chopra and P. Meindl. *Inventory Management and Production Planning and Scheduling*. Prentice-Hall, 2001.
- [Chun 2003] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003. ISSN 0146-4833.
- [Dahlgren 1996] Fredrik Dahlgren and Per Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(4):385–398, 1996. ISSN 1045-9219.
- [Fu 1991] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 54–63, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-394-9.
- [Ganger 1999] G. Ganger, B. Worthington, and Y. Patt. The disksim simulation environment version 2.0, Dec. 1999.
- [Gill 2007] B. Gill and L. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 185–198, 2007.
- [Gill 2005] B. Gill and D. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX '05)*, pages 293–308, 2005.
- [Gill 2008] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.
- [Gross 1985] D. Gross and C. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, 1985.
- [Gupta 2008] Diwaker Gupta, Sangmin Lee, and Michael Vrable. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, Berkeley, CA, USA, 2008. USENIX Association.
- [Jeffay 1998] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 480–491, 1998.
- [Kaplan 2002] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepagging. In *Proceedings of the third international symposium on Memory management*, 2002. ISBN 1-58113-539-4.
- [Kleinrock 1976] L. Kleinrock. *Queueing Systems: Volume 2: Computer Applications*. John Wiley & Sons, 1976.

- [Koopman 1936] B.O. Koopman. On distributions admitting a sufficient statistic. *Transaction of American Mathematics Society*, 1936.
- [Lee 1987] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the International conference on parallel processing*, pages 28–31, 1987.
- [Li 2005] Chuanpeng Li and Kai Shen. Managing prefetch memory for data-intensive online servers. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.
- [Li 2007] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential i/o. *SIGOPS Oper. Syst. Rev.*, 41(3):189–202, 2007. ISSN 0163-5980.
- [Li 2004] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 173–186, 2004.
- [Liang 2007] S. Liang, S. Jiang, and X. Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS'07)*, page 64, 2007.
- [mit-beer] mit-beer. The mit beer game. <http://beergame.mit.edu>.
- [Mosberger 1998] David Mosberger and Tai Jin. httpperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998. ISSN 0163-5999.
- [P. Jain 2001] S. Devadas P. Jain and L. Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. In *Tech. Rep. CSG-462, M.I.T.*, 2001.
- [Papathanasiou 2004] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference (USENIX '04)*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.
- [Patterson 1995] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.
- [Reungsang 2001] P. Reungsang, S. K. Park, S.-W. Jeong, H.-L. Roh, and G. Lee. Reducing cache pollution of prefetching in a small data cache. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, page 530, Washington, DC, USA, 2001. IEEE Computer Society.
- [Silver 1998] Edward Silver, David Pyke, and Rein Peterson. *Inventory Management and Production Planning and Scheduling*. John Wiley & Sons, 1998. ISBN 978-0471119470.
- [Smith 1982] A. Smith. Cache memories. In *ACM Computing Surveys (CSUR)*, volume 14, pages 473–530. ACM Press, 1982.
- [Soundararajan 2008] Gokul Soundararajan, Madalin Mihalescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 377–390, Berkeley, CA, USA, 2008. USENIX Association.
- [Tomkins 1997] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, 1997. URL citeseer.nj.nec.com/tomkins97informed.html.
- [Wong 2002] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.
- [Yadgar 2008] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Mc2: Multiple clients on a multilevel cache. In *ICDCS '08: Proceedings of the 28th International Conference on Distributed Computing Systems*, Beijing, China, 2008.
- [Zhang 2008] Zhe Zhang, Kyuhyung Lee, Xiaosong Ma, and Yuanyuan Zhou. Pfc: Transparent optimization of existing prefetching strategies for multi-level storage systems. In *ICDCS '08: Proceedings of the 28th International Conference on Distributed Computing Systems*, Beijing, China, 2008.
- [Zhou 2001] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.
- [Zhu 2005] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 177–190, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5.