Competitive Prefetching for Concurrent Sequential I/O*

Chuanpeng Li
Dept. of Computer Science
University of Rochester
cli@cs.rochester.edu

Kai Shen
Dept. of Computer Science
University of Rochester
kshen@cs.rochester.edu

Athanasios E. Papathanasiou
Intel Massachusetts
athanasios.papathanasiou@intel.com

ABSTRACT

During concurrent I/O workloads, sequential access to one I/O stream can be interrupted by accesses to other streams in the system. Frequent switching between multiple sequential I/O streams may severely affect I/O efficiency due to long disk seek and rotational delays of disk-based storage devices. Aggressive prefetching can improve the granularity of sequential data access in such cases, but it comes with a higher risk of retrieving unneeded data. This paper proposes a *competitive* prefetching strategy that controls the prefetching depth so that the overhead of disk I/O switch and unnecessary prefetching are balanced. The proposed strategy does not require *a-priori* information on the data access pattern, and achieves at least half the performance (in terms of I/O throughput) of the optimal offline policy. We also provide analysis on the optimality of our competitiveness result and extend the competitiveness result to capture prefetching in the case of random-access workloads.

We have implemented the proposed competitive prefetching policy in Linux 2.6.10 and evaluated its performance on both standalone disks and a disk array using a variety of workloads (including two common file utilities, Linux kernel compilation, the TPC-H benchmark, the Apache web server, and index searching). Compared to the original Linux kernel, our competitive prefetching system improves performance by up to 53%. At the same time, it trails the performance of an oracle prefetching strategy by no more than 42%.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management

General Terms

Design, Performance, Experimentation

Keywords

Competitive Prefetching, I/O, Performance Evaluation

*This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0615045, CCF-0621472, NSF CAREER Award CCF-0448413, and an IBM Faculty Award.

1. INTRODUCTION

Concurrent access to multiple sequential I/O streams is a common case in both server and desktop workloads. Multitasking systems allow the concurrent execution of programs that access different I/O streams, while common applications, such as the `diff` file utility, database operations and file merging operations, multiplex accesses to several files. For concurrent I/O workloads, continuous accesses to one sequential data stream can be interrupted by accesses to other streams in the system. Frequent switching between multiple sequential I/O streams may severely affect I/O efficiency due to long disk seek and rotational delays of disk-based storage devices. In the rest of the paper, we refer to any disk head movement that interrupts a sequential data transfer as an I/O switch. The cost of an I/O switch includes both seek and rotational delays.

The problem of unnecessary disk I/O switch during concurrent I/O workloads may be alleviated partially by non-work-conserving disk schedulers. In particular, the anticipatory scheduler [16] may temporarily idle the disk (even when there are outstanding I/O requests) so that consecutive I/O requests that belong to the same process/thread may be serviced without interruption. However, anticipatory scheduling is ineffective, in cases where a single process or thread synchronously accesses multiple sequential data streams in an interleaving fashion or when a substantial amount of processing or think time exists between consecutive sequential requests from a process. In addition, anticipatory scheduling may not function properly when it lacks the knowledge of I/O request-issuing process contexts, as in the cases of I/O scheduling within a virtual machine monitor [17] or within a parallel/distributed file system server [8].

Another technique to reduce I/O switch frequency for concurrent I/O workloads is to increase the granularity of sequential I/O operations. This can be accomplished by aggressive OS-level prefetching that prefetches deeper in each sequential I/O stream. However, under certain scenarios, aggressive prefetching may have a negative impact on performance due to buffer cache pollution and inefficient use of I/O bandwidth. Specifically, aggressive prefetching has a higher risk of retrieving data that are not needed by the application. This paper proposes an OS-level prefetching technique that balances the overhead of I/O switching and the disk bandwidth wasted on prefetching unnecessary data. We focus on the mechanism that controls the prefetching depth for sequential or partially sequential accesses. We do not consider other aspects or forms of prefetching.

The proposed prefetching technique is based on a fundamental *2-competitiveness* result: "when the prefetching depth is equal to the amount of data that can be sequentially transferred within the average time of a single I/O switch, the total disk resource consumption of an I/O workload is at most twice that of the optimal offline prefetching strategy". Our proposed prefetching technique does not require *a-priori* knowledge of an application's data access pattern or any other type of application support. We also show that the 2-competitiveness result is optimal, *i.e.*, no transparent OS-level online prefetching can achieve a competitiveness ratio lower than 2. Finally, we further extend the competitiveness result to capture prefetching in the case of random-access workloads.

The rest of this paper is structured as follows. Section 2 discusses previous work. Section 3 provides background on our targeted I/O workloads, existing OS support, and relevant storage device characteristics. Section 4 presents the analysis and design of our proposed competitive prefetching strategy. Section 5 discusses practical issues concerning storage device characterization. Section 6 evaluates the proposed strategy using several microbenchmarks and a variety of real application workloads. Section 7 concludes the paper.

2. RELATED WORK

The literature on prefetching is very rich. Cao *et al.* explored application-controlled prefetching and caching [6, 7]. They proposed a two-level page replacement scheme that allows applications to control their cache replacement policy, while the kernel controls the allocation of cache space among applications. Patterson *et al.* explored a cost-benefit model to guide prefetching decisions [26]. Their results suggest a prefetching depth up to a process' *prefetch horizon* under the assumption of no disk congestion. Tomkins *et al.* extended the proposed cost-benefit model for workloads consisting of multiple applications [30]. The above techniques require *a-priori* knowledge of the application I/O access patterns. Patterson *et al.* and Tomkins *et al.* depend on application disclosed hints to predict future data accesses. In contrast, our work proposes a transparent operating system prefetching technique that does not require any application support or modification.

Previous work has also examined ways to acquire or predict I/O access patterns without direct application involvement. Proposed techniques include modeling and analysis of important system events [20, 34], offline application profiling [25], and speculative program execution [10, 12]. The reliance on I/O access pattern prediction affects the applicability of these approaches in several ways. Particularly, the accuracy of predicted information is not guaranteed and no general method exists to assess the accuracy for different application workloads with certainty. In addition, some of these approaches still require offline profiling or application changes, which increases the barrier for deployment in real systems.

Shriver *et al.* studied performance factors in a disk-based file system [29] and suggested that aggressive prefetching should be employed when it is safe to assume the prefetched data will be used. More recently, Papathanasiou and Scott argued that aggressive prefetching has become more and more appropriate due to recent developments in I/O hardware and emerging needs of applications [24]. Our work builds on this idea by providing a systematic way to control prefetching depth.

Previous work has also explored performance issues associated with concurrent sequential I/O at various system levels. Carrera and Bianchini explored disk cache management and proposed two disk firmware-level techniques to improve the I/O throughput of data intensive servers [9]. Our work shares their objective while focusing on operating system-level techniques. Anastasiadis *et al.* explored an application-level block reordering technique that can reduce server disk traffic when large content files are shared by concurrent clients [1]. Our work provides transparent operating system level support for a wider scope of data-intensive workloads.

Barve *et al.* [5] have investigated competitive prefetching in parallel I/O systems with a given lookahead amount (*i.e.*, the knowledge of a certain number of upcoming I/O requests). Their problem model is different from ours on at least the following aspects. First, the target performance metric in their study (the number of parallel I/O operations to complete a given workload) is only meaningful in the context of parallel I/O systems. Second, the lookahead of upcoming of I/O requests is essential in their system model while our study only considers transparent OS-level prefetching techniques that require no information about application data access pattern.

Our previous position paper [21] introduced the competitiveness result for server workloads under the assumption of a constant I/O switch time and a constant sequential disk transfer rate. This paper substantially relaxes the above assumptions. Additionally, it shows the optimality of our competitiveness result; it extends the competitiveness result to capture prefetching in the case of random-access workloads; and evaluates the proposed prefetching strategy for significantly broader range of applications and storage devices.

3. BACKGROUND

3.1 Targeted I/O Workloads

Our work targets concurrent, read-mostly, sequential I/O workloads. We define a *sequential I/O stream* as a group of spatially contiguous data items that are accessed by a single program (process or thread). The program does not necessarily make use of the whole stream. It may also perform interleaving I/O that does not belong to the same sequential stream. Multiple sequential I/O streams may be accessed concurrently under the following situations.

First, in a multitasking system several applications access disk-resident data concurrently (Figure 1). This case is particularly common in data-intensive online servers where multiple user requests are serviced simultaneously. For example, a substantial fraction of disk operations in FTP and web servers, which tend to retrieve whole or partial files, is associated with sequential I/O streams. In addition, some applications employ data correlation techniques in order to layout their data sequentially on disk and increase I/O efficiency.

Concurrent sequential I/O may also be generated by a sin-

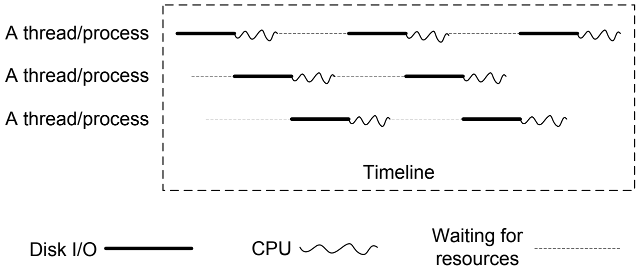

Figure 1: Concurrent I/O under concurrent program executions.

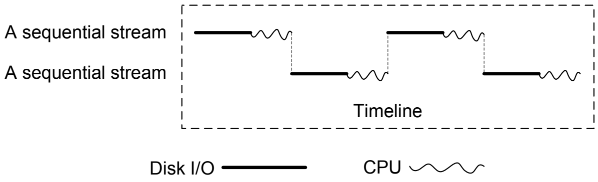

Figure 2: Concurrent sequential I/O under alternating accesses from a single program execution.

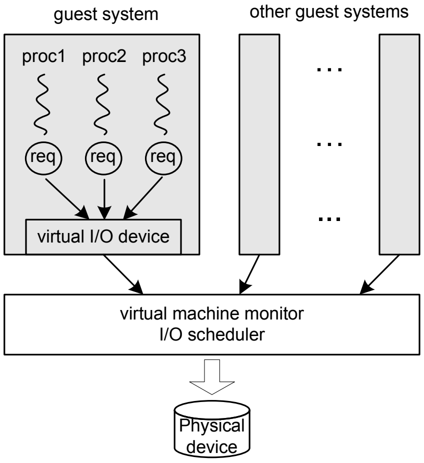

Figure 3: An illustration of I/O scheduling at the virtual machine monitor (VMM). Typically, the VMM I/O scheduler has no knowledge of the request issuing process contexts within the guest systems. As a result, the anticipatory I/O scheduling at the VMM may not function properly.

gle process or thread that multiplexes accesses among multiple sequential streams (Figure 2). Examples include the file comparison utility `diff` and several database workloads. For instance, web search engines [3] maintain an ordered list of matching web pages for each indexed keyword. For multi-keyword queries, the search engine has to compute the intersection of multiple such lists. In addition, SQL multi-table `join` queries require accessing a number of files sequentially.

Many applications read a portion of each data stream at a time instead of retrieving the complete data stream into application-level buffers at once. Reading complete data streams is generally avoided for the following reasons. First, some applications do not have prior knowledge of the exact amount of data required from a certain data stream. For instance, index search and SQL `join` queries will often terminate when a desired number of "matches" are found. Second, applications that employ memory-mapped I/O do not directly initiate I/O operations. The operating system loads into memory a small number of pages (through page faults) at a time on their behalf. Third, retrieving complete data streams into application-level buffers may result in double buffering and/or increase memory contention. For instance, while GNU `diff` [15] reads the complete files into memory at once, the BSD `diff` can read a portion of the files at a time and thus it consumes significantly less memory when comparing large files.

3.2 Existing Operating System Support

During concurrent I/O workloads, sequential access to a data stream can be interrupted by accesses to other streams, increasing the number of expensive disk seek operations. Anticipatory scheduling [16] can partially alleviate the problem. At the completion of an I/O request, anticipatory scheduling may keep the disk idle for a short period of time (even if there are outstanding requests) in anticipation of a new I/O request from the process/thread that issued the request that was just serviced. Consecutive requests generated from the same process/thread often involve data residing on neighboring disk locations and require little or no seeking. This technique results in significant performance improvement for concurrent I/O workloads, where all I/O requests, issued by an individual process, exhibit strong locality. However, anticipatory scheduling may be ineffective in several cases.

- Since anticipatory scheduling expects a process to issue requests with strong locality in a consecutive way, it cannot avoid disk seek operations when a single process multiplexes accesses among two or more data streams (Figure 2).

- The limited anticipation period introduced by anticipatory scheduling may not avoid I/O switches when a substantial amount of processing or think time exists between consecutive sequential requests from the same process. More specifically, the anticipation may timeout before the anticipated request actually arrives.

- Anticipatory scheduling needs to keep track of the I/O behavior of each process so that it can decide whether to wait for a process to issue a new request and how long to wait. It may not function properly when the I/O scheduler does not know the process contexts from which I/O requests are issued. In particular, Jones *et al.* [17] pointed out that I/O schedulers within virtual machine monitors typically do not know the request process contexts inside the guest systems (Figure 3). Another example is that the I/O scheduler at a parallel/distributed file system server [8] may not have the knowledge of remote process contexts for I/O requests.

An alternative to anticipatory scheduling is I/O prefetching, which can also increase the granularity of sequential data accesses and, consequently, decrease the frequency of disk I/O switch. Since sequential transfer rates improve significantly faster than seek and rotational latencies on modern disk drives, aggressive prefetching becomes increasingly appealing. In order to limit the amount of wasted I/O bandwidth and memory space, operating systems often employ an upper-bound on the prefetching depth, without a systematic understanding of the impact of the prefetching depth on performance.

Our analysis in this paper assumes that application-level logically sequential data blocks are mapped to physically contiguous blocks on storage devices. This is mostly true because file systems usually attempt to organize logically sequential data blocks in a contiguous way in order to achieve high performance for sequential access. Bad sector remapping on storage devices can disrupt sequential block allocation. However, such remapping does not occur for the majority of sectors in practice. We do not consider its impact on our analysis.

3.3 Disk Drive Characteristics

The disk service time of an I/O request consists of three main components: the head seek time, the rotational delay, and the data transfer time. Disk performance characteristics have been extensively studied in the past [19, 27, 28]. The seek time depends on the distance to be traveled by the disk head. It consists roughly of a fixed head initiation cost plus a cost linear to the seek distance. The rotational delay mainly depends on the rotation distance (fraction of a revolution). Techniques such as out-of-order transfer and freeblock scheduling [23] are available to reduce the rotational delay. The sequential data transfer rate varies at different data locations because of zoning on modern disks. Specifically, tracks on outer cylinders often contain more sectors per track and, consequently, exhibit higher data transfer rates.

Modern disks are equipped with read-ahead caches so they may continue to read data after the requested data have already been retrieved. However, disk cache sizes are relatively small (usually less than 16 MB) and, consequently, data stored in the disk cache are replaced frequently under concurrent I/O workloads. In addition, read-ahead is usually performed only when there are no pending requests, which is rare in the context of I/O-intensive workloads. Therefore, we do not consider the impact of disk cache read-ahead on our analysis of OS-level prefetching.

4. COMPETITIVE PREFETCHING

We investigate I/O prefetching strategies that address the following tradeoff in deciding the I/O prefetching depth: conservative prefetching may *lead to a high I/O switch overhead*, while aggressive prefetching may *waste too much I/O bandwidth on fetching unnecessary data*. Section 4.1 presents a prefetching policy that achieves at least half the performance (in terms of I/O throughput) of the optimal offline prefetching strategy without *a-priori* knowledge of the application data access pattern. In the context of this paper, we refer to the "optimal" prefetching as the best-performing sequential prefetching strategy that minimizes the combined cost of disk I/O switch and retrieving unnecessary data. We do not consider other issues such as prefetching-induced memory contention. Previous work has explored such issues [6, 18, 22, 26, 30]. In Section 4.2, we show that the 2-competitive prefetching is the best competitive strategy possible. Section 4.3 examines prefetching strategies adapted to accommodate mostly random-access workloads and their competitiveness results.

The competitiveness analysis in this section focuses on systems employing standalone disk drives. In later sections, we will discuss practical issues and present experimental results when extending our results for disk arrays.

4.1 2-Competitive Prefetching

Consider a concurrent I/O workload consisting of N sequential streams. For each stream i ($1 \leq i \leq N$), we assume the total amount of accessed data (or stream length) is $S_{tot}[i]$. Due to zoning on modern disks, the sequential data transfer rate on a disk can vary depending on the data location. Since each stream is likely localized to one disk zone, it is reasonable to assume that the transfer rate for any data access on stream i is a constant, denoted as $R_{tr}[i]$.

For stream i, the optimal prefetching strategy has *a-priori* knowledge of $S_{tot}[i]$ (the amount of data to be accessed). It performs a single I/O switch (including seek and rotation) to the head of the stream and then sequentially transfers data of size $S_{tot}[i]$. Let the I/O switch cost for stream i be $C_{switch}[i]$. The disk resource consumption (in time) for accessing all N streams under optimal prefetching is:

$$\mathcal{C}_{total}^{opt} = \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + C_{switch}[i] \right) \tag{1}$$

However, in practice the total amount of data to be accessed $S_{tot}[i]$ is not known *a-priori*. Consequently, OS-level prefetching tries to approximate $S_{tot}[i]$ using an appropriate prefetching depth. For simplicity, we assume that the OS employs a constant prefetching depth $S_p[i]$ for each stream.[1] Therefore, accessing a stream requires $\lceil \frac{S_{tot}[i]}{S_p[i]} \rceil$ prefetch operations. Each prefetch operation requires one I/O switch. Let the I/O switch cost be $C_{switch}[i, j]$ for the j-th prefetch operation of stream i. The total I/O switch cost for accessing stream i is represented by:

$$\mathcal{C}_{tot\_switch}[i] = \sum_{1 \leq j \leq \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil} C_{switch}[i, j] \tag{2}$$

The time wasted while fetching unnecessary data is bounded by the cost of the last prefetch operation:

$$\mathcal{C}_{waste}[i] \leq \frac{S_p[i]}{R_{tr}[i]} \tag{3}$$

Therefore, the total disk resource (in time) consumed to access all streams is bounded by:

$$\begin{aligned} &\mathcal{C}_{total} \\ &= \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \mathcal{C}_{waste}[i] + \mathcal{C}_{tot\_switch}[i] \right) \\ &\leq \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} + \sum_{1 \leq j \leq \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil} C_{switch}[i, j] \right) \end{aligned} \tag{4}$$

[1]The assumption of a constant prefetching depth constrains only the design of the prefetching policy. However, what can be achieved with a constrained policy can certainly be achieved with a non-constrained, broader policy. Thus, our simplification does not restrict the competitiveness result presented in this section.

The expected average I/O switch time (including seek and rotation) for a concurrent I/O workload can be derived from the disk drive characteristics and the workload concurrency. We consider the seek time and rotational delay below.

- I/O scheduling algorithms such as the Cyclic-SCAN reorder outstanding I/O requests based on the data location and schedule the I/O request closest to the current disk head location. When the disk scheduler can choose from n concurrent I/O requests at uniformly random disk locations, the inter-request seek distance D_{seek} follows the following distribution:

$$Pr[D_{seek} \geq x] = (1 - \frac{x}{C})^n \tag{5}$$

  Where C is the size of the contiguous portion of the disk where the application dataset resides. The expected seek time can then be acquired by combining Equation (5) and the seek distance to seek time mapping of the disk.

- The expected rotational delay of an I/O workload is the mean rotational time between two random track locations (*i.e.*, the time it takes the disk to spin half a revolution).

The above result suggests that the expected average I/O switch time of a concurrent workload is independent of the prefetching schemes (optimal or not) being employed. We denote such cost as C^{avg}_{switch}. Therefore, Equation (1) can be simplified to:

$$\mathcal{C}^{opt}_{total} = \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} \right) + N \cdot C^{avg}_{switch} \tag{6}$$

And Equation (4) can be simplified to:

$$\begin{aligned}
&\mathcal{C}_{total} \\
&\leq \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \sum_{1 \leq i \leq N} \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil \cdot C^{avg}_{switch} \\
&< \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \sum_{1 \leq i \leq N} \left(\frac{S_{tot}[i]}{S_p[i]} + 1 \right) \cdot C^{avg}_{switch}
\end{aligned} \tag{7}$$

Based on Equations (6) and (7), we find that:

$$\text{if } S_p[i] = C^{avg}_{switch} \cdot R_{tr}[i], \quad \text{then } \mathcal{C}_{total} < 2 \cdot \mathcal{C}^{opt}_{total} \tag{8}$$

Equation 8 allows us to design a prefetching strategy with bounded worst-case performance:

> When the prefetching depth is equal to the amount of data that can be sequentially transferred within the average time of a single I/O switch, the total disk resource consumption of an I/O workload is at most twice that of the optimal offline strategy.

Consequently, the I/O throughput of a prefetching strategy that follows the above guideline is at least half of the throughput of the optimal offline strategy. *Competitive* strategies commonly refer to solutions whose performance can be shown to be no worse than some constant factor of an optimal offline strategy. We follow such convention to name our prefetching strategy *competitive prefetching*.

4.2 Optimality of 2-Competitiveness

In this section, we show that no transparent OS-level online prefetching strategy (without prior knowledge of the sequential stream length $S_{tot}[i]$ or any other application-specific information) can achieve a competitiveness ratio lower than 2. Intuitively, we can reach the above conclusion since: 1) short sequential streams demand conservative prefetching, while long streams require aggressive prefetching; and 2) no online prefetching strategy can unify the above conflicting goals with a competitiveness ratio lower than 2.

We prove the above by contradiction. We assume there exists an α-competitive solution $\mathcal{X}$ (where $1 < \alpha < 2$). In the derivation of the contradiction, we assume the I/O switch cost C_{switch} is constant.[2] The discussion below is in the context of a single stream, therefore we do not specify the stream id for notational simplicity (*e.g.*, $S_{tot}[i] \rightarrow S_{tot}$). In $\mathcal{X}$, let S^n_p be the size of the n-th prefetch for a stream. Since $\mathcal{X}$ is α-competitive, we have $\mathcal{C}_{total} \leq \alpha \cdot \mathcal{C}^{opt}_{total}$.

First, we show that the size of the first prefetch operation S^1_p satisfies:

$$S^1_p < C_{switch} \cdot R_{tr} \tag{9}$$

Otherwise, we let $S_{tot} = \epsilon$ where $0 < \epsilon < (\frac{2-\alpha}{\alpha}) \cdot C_{switch} \cdot R_{tr}$. In this case, $\mathcal{C}^{opt}_{total} = C_{switch} + \frac{\epsilon}{R_{tr}}$ and $\mathcal{C}_{total} = C_{switch} + \frac{S^1_p}{R_{tr}} \geq 2 \cdot C_{switch}$. Therefore, we have $\mathcal{C}_{total} > \alpha \cdot \mathcal{C}^{opt}_{total}$. Contradiction!

We then show that:

$$\forall k \geq 2, \quad S^k_p < \left(\sum_{j=1}^{k-1} S^j_p \right) - (k-2) \cdot C_{switch} \cdot R_{tr} \tag{10}$$

Otherwise (if Equation (10) is not true for a particular k), we define $\mathcal{T}$ as below, and we know $\mathcal{T} > 0$.

$$\mathcal{T} = S^k_p + (k - \alpha) \cdot C_{switch} \cdot R_{tr} + (1 - \alpha) \cdot \left(\sum_{j=1}^{k-1} S^j_p \right)$$

We then let $S_{tot} = \left(\sum_{j=1}^{k-1} S^j_p \right) + \epsilon$ where $0 < \epsilon < \frac{\mathcal{T}}{\alpha}$. In this case, $\mathcal{C}^{opt}_{total} = C_{switch} + \frac{\left(\sum_{j=1}^{k-1} S^j_p\right) + \epsilon}{R_{tr}}$ and $\mathcal{C}_{total} \geq k \cdot C_{switch} + \frac{\left(\sum_{j=1}^{k} S^j_p\right)}{R_{tr}}$. Therefore, we have $\mathcal{C}_{total} > \alpha \cdot \mathcal{C}^{opt}_{total}$. Contradiction!

We next show:

$$\forall k \geq 2, \quad S^k_p < S^{k-1}_p \tag{11}$$

We prove Equation 11 by induction. Equation (11) is true for $k = 2$ as a direct result of Equation (10). Assume Equation (11) is true for all $2 \leq k \leq \hat{k} - 1$. This also means that

[2] This assumption is safe because contradiction under the constrained disk system model entails contradiction under the more general disk system model. Specifically, if an α-competitive solution does not exist under the constrained disk system model, then such solution certainly should not exist under the more general case.

$S_p^k < C_{switch} \cdot R_{tr}$ for all $2 \le k \le \hat{k} - 1$. Below we show Equation (11) is true for $k = \hat{k}$:

$$
\begin{aligned}
S_p^{\hat{k}} &< \left(\sum_{j=1}^{\hat{k}-1} S_p^j \right) - (\hat{k} - 2) \cdot C_{switch} \cdot R_{tr} \\
&= S_p^{\hat{k}-1} + \left(\sum_{j=1}^{\hat{k}-2} (S_p^j - C_{switch} \cdot R_{tr}) \right) \\
&< S_p^{\hat{k}-1}
\end{aligned} \tag{12}
$$

Our final contradiction is that for any $k \ge \frac{2 \cdot C_{switch} \cdot R_{tr} - S_p^1}{C_{switch} \cdot R_{tr} - S_p^1}$, we have $S_p^k < 0$:

$$
\begin{aligned}
S_p^k &< \left(\sum_{j=1}^{k-1} S_p^j \right) - (k - 2) \cdot C_{switch} \cdot R_{tr} \\
&< (k-1) \cdot S_p^1 - (k-2) \cdot C_{switch} \cdot R_{tr} \\
&= 2 \cdot C_{switch} \cdot R_{tr} - S_p^1 - k \cdot (C_{switch} \cdot R_{tr} - S_p^1) \\
&\le 2 \cdot C_{switch} \cdot R_{tr} - S_p^1 - (2 \cdot C_{switch} \cdot R_{tr} - S_p^1) \\
&= 0
\end{aligned} \tag{13}
$$

4.3 Accommodation for Random-Access Workloads

Competitive strategies address only the worst-case performance. A general-purpose operating system prefetching policy should deliver high average-case performance for common application workloads. In particular, many applications only exhibit fairly short sequential access streams or even completely random access patterns. In order to accommodate these workloads, our competitive prefetching policy employs a *slow-start* phase, commonly employed in modern operating systems. For a new data stream, prefetching takes place with a relatively small initial prefetching depth. Upon detection of a sequential access pattern, the depth of each additional prefetching operation is increased until it reaches the desired competitive prefetching depth.

The following example illustrates the slow-start phase. Let the initial prefetching depth be 16 pages (64 KB). Upon detection of a sequential access pattern, the depth of each consecutive prefetch operation increases by a factor of two until it reaches its maximum permitted value. For a sequential I/O stream with a competitive prefetching depth of 98 pages (392 KB), a typical series of prefetching depths should look like: 16 pages → 32 pages → 64 pages → 98 pages → 98 pages → ···

For our targeted sequential I/O workloads, the introduction of slow-start phase will affect the competitiveness of our proposed technique. In the rest of this subsection, we extend the previous competitiveness result when a slow-start phase is employed by the prefetching policy. Assume the slow-start phase consists of at most P prefetching operations (3 in the above example). Also let the total amount of data fetched during a slow-start phase be bounded by T (112 pages in the above example). The total disk resource consumed (Equation (7)) becomes:

$$
\begin{aligned}
&\mathcal{C}_{total} \\
&\le \sum_{1 \le i \le N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \sum_{1 \le i \le N} \left(\lceil \frac{S_{tot}[i] - T}{S_p[i]} \rceil + P \right) \cdot C_{switch}^{avg} \\
&< \sum_{1 \le i \le N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \sum_{1 \le i \le N} \left(\frac{S_{tot}[i]}{S_p[i]} + 1 + P \right) \cdot C_{switch}^{avg}
\end{aligned} \tag{14}
$$

Since the competitive prefetching depth $S_p[i]$ is equal to $C_{switch}^{avg} \cdot R_{tr}[i]$, Equation (14) becomes:

$$
\mathcal{C}_{total} < 2 \cdot \sum_{1 \le i \le N} \left(\frac{S_{tot}[i]}{R_{tr}[i]} \right) + (2 + P) \cdot N \cdot C_{switch}^{avg} \tag{15}
$$

Equations (6) and (15) lead to $\mathcal{C}_{total} < (2 + P) \cdot \mathcal{C}_{total}^{opt}$. Consequently, the prefetching strategy that employs a slow-start phase is still competitive, but with a weaker competitiveness factor ("$2+P$" instead of "2"). Despite this weaker analytical result, our experimental results in Section 6 show that the new strategy still delivers at least half the emulated optimal performance in practice.

5. IMPLEMENTATION ISSUES

We have implemented the proposed competitive prefetching technique in the Linux 2.6.10 kernel. In our implementation, we use our competitive prefetching policy to control the depth of prefetching operations. We have not modified other aspects of the prefetching algorithm used in the Linux kernel.

Our competitive prefetching policy depends on several storage device characteristics, including the functional mapping from seek distance to seek time (denoted by f_{seek}) and the mapping from the data location to the sequential transfer rate (denoted by $f_{transfer}$).

The disk drive characteristics can be evaluated offline, at disk installation time or OS boot time. Using the two mappings, we determine the runtime C_{switch}^{avg} and $R_{tr}[i]$ in the following way:

- During runtime, our system tracks the seek distances of past I/O operations and determines their seek time using f_{seek}. We use an exponentially-weighted moving average of the past disk seek times for estimating the seek component of the next I/O switch cost. The average rotational delay component of C_{switch}^{avg} is estimated as the average rotational delay between two random track locations (*i.e.*, the time it takes the disk to spin half a revolution).

- For each I/O request starting at logical disk location $L_{transfer}$, the data transfer rate is determined as $f_{transfer}(L_{transfer})$.

The competitiveness analysis in the previous section targets systems with standalone disk drives. However, the analysis applies also to more complex storage devices as long as they exhibit disk-like I/O switch and sequential data transfer characteristics. Hence, multi-disk storage devices can utilize the proposed competitive prefetching algorithm. Disk arrays allow simultaneous transfers out of multiple disks so they may offer higher aggregate I/O bandwidth. However,

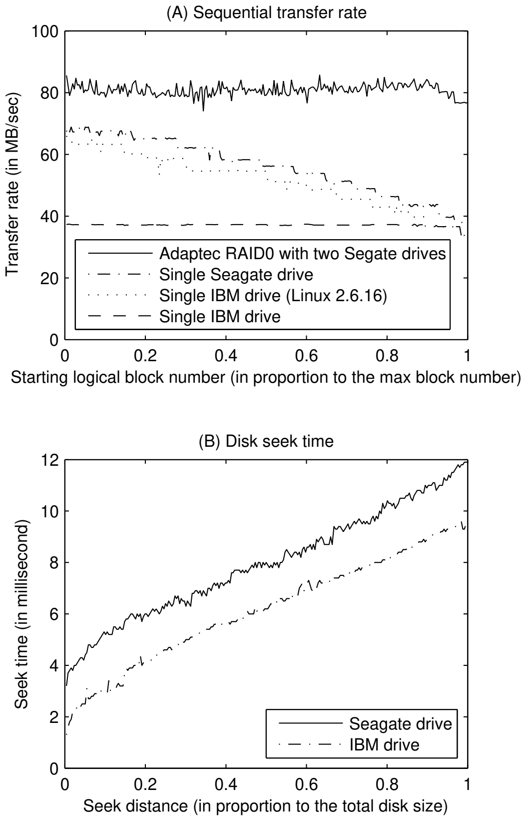

Figure 4: Sequential transfer rate and seek time for two disk drives and a RAID. Most results were acquired under the Linux 2.6.10 kernel. The transfer rate of the IBM drive appears to be affected by the operating system software. We also show its transfer rate on a Linux 2.6.16 kernel.

seek and rotational delays are still limited by individual disks. Overall, multi-disk storage devices often desire larger competitive prefetching depths.

We include three disk-based storage devices in our experimental evaluation: a 36.4 GB IBM 10 KRPM SCSI drive, a 146 GB Seagate 10 KRPM SCSI drive, and a RAID0 disk array with two Seagate drives using an Adaptec RAID controller. We have also measured a RAID5 disk array with five Seagate drives, which has a lower transfer rate than the RAID0 due to the overhead of relatively complex RAID5 processing. We keep the RAID0 device in our evaluation as a representative of high-bandwidth devices.

We measure the storage device properties by issuing direct SCSI commands through the Linux generic SCSI interface, which allows us to bypass the OS memory cache and selectively disable the disk controller cache. Our disk profiling takes less than two minutes to complete for each disk drive/array and it could be easily performed at disk installation time. Figure 4 shows the measured sequential transfer rate and the seek time. Most measurements were performed on a Linux 2.6.10 kernel. We speculate that the transfer rate of the IBM drive is constrained by a software-related bottleneck (at 37.5 MB/sec, or 300 Mbps). A more recent Linux kernel (2.6.16) does not have this problem.

For the IBM drive (under Linux 2.6.10 kernel), the transfer speed is around 37.3 MB/sec; the average seek time between two independently random disk locations is 7.53 ms; and the average rotational delay is 3.00 ms. The average competitive prefetching depth is the amount of data that can sequentially transferred within the average time of a single disk seek and rotation—393 KB. For the Seagate drive and the RAID0 disk array, measured average sequential transfer rates are 55.8 MB/sec and 80.9 MB/sec respectively. With the assumption that the average seek time of the disk array is the same as that of a single disk drive (7.98 ms), the average competitive prefetching depths for the Seagate drive and the disk array are 446 KB and 646 KB respectively. In comparison, the default maximum prefetching depth in Linux 2.6.10 is only 128 KB.

Note that the above average competitive prefetching derivation is based on the average seek time between two independently random disk locations (or when there are two concurrent processes). The average seek time is smaller at higher concurrency levels since the disk scheduler can choose from more concurrent requests for seek reduction. Thus, the runtime competitive prefetching depth should be lower in these cases. We use the average competitive prefetching depth as a reference for implementing aggressive prefetching (section 6.2).

6. EXPERIMENTAL EVALUATION

We assess the effectiveness and competitiveness of the proposed technique in this section. In our experimental evaluation, we compare a Linux kernel extended with our competitive prefetching policy against the original Linux kernel and a Linux kernel employing an aggressive prefetching algorithm. Anticipatory scheduling is enabled across all experiments and kernel configurations. Our main experimental platform is a machine equipped with two 2 GHz Intel Xeon processors, 2 GB memory, and three storage drives. Figure 4 presents sequential transfer rate and seek time characteristics of the three storage drives.

The performance of competitive prefetching is affected by several factors: 1) virtualized or distributed I/O architecture that may limit the effectiveness of anticipatory I/O scheduling; 2) the storage device characteristics; and 3) serial *vs.* concurrent workloads. Section 6.2 presents the base-case performance of concurrent workloads using the standalone IBM drive. Then we explicitly examine the impact of each of these factors (sections 6.3, 6.4, and 6.5). Finally, section 6.6 summarizes the experimental results.

6.1 Evaluation Benchmarks

Our evaluation benchmarks consist of a variety of I/O-intensive workloads, including four microbenchmarks, two common file utilities, and four real applications.

Each of our microbenchmarks is a server workload involving a server and a client. The server provides access to a dataset of 6000 4 MB disk-resident files. Upon arrival of a new request from the client, the server spawns a thread, which we call a request handler, to process it. Each request handler reads one or more disk files in a certain pattern. The microbenchmark suite consists of four microbenchmarks that are characterized by different access patterns. Specifically, the microbenchmarks differ in the number of files accessed by each request handler (one to four), the portion of each file accessed (the whole file, a random portion, or a 64 KB chunk) and a processing delay (0 ms or 10 ms). We use

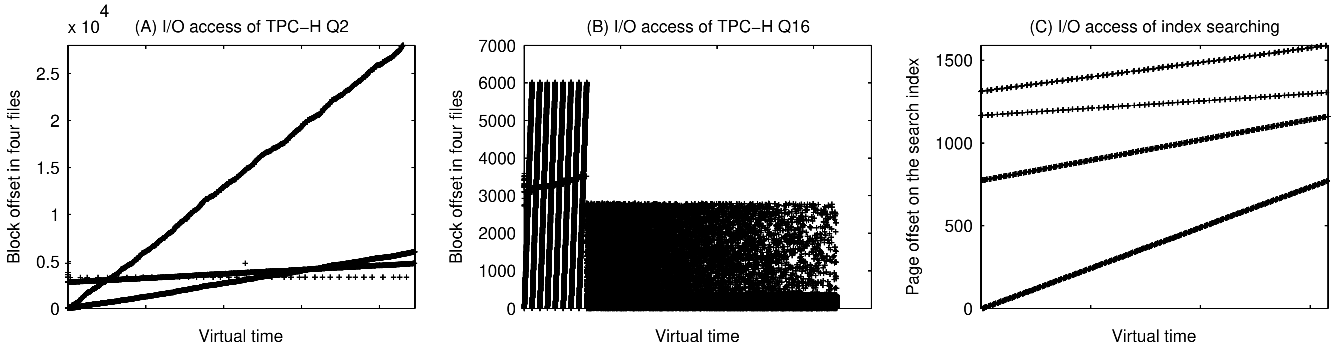

Figure 5: I/O access pattern of certain application workloads. Virtual time represents the sequence of discrete data accesses. In (A), each straight line implies sequential access pattern of a file. In (B), random and non-sequential accesses make the files indistinguishable.

a descriptive naming convention to refer to each benchmark. For example, <Two-Rand-0> describes a microbenchmark whose request handler accesses a random portion of two files with 0 ms processing delay. The random portion starts from the beginning of each file. Its size is evenly distributed between 64 KB and 4 MB. Accesses within a portion of a file are always sequential in nature. The specifications of the four microbenchmarks are:

- *One-Whole-0*: Each request handler randomly chooses a file and it repeatedly reads 64 KB data blocks until the whole file is accessed.
- *One-Rand-10:* Each request handler randomly chooses a file and it repeatedly reads 64 KB data blocks from the beginning of the file up to a random total size (evenly distributed between 64 KB and 4 MB). Additionally, we add a 10 ms processing delay at four random file access points during the processing of each request. The processing delays are used to emulate possible delays during request processing and may cause the anticipatory I/O scheduler to timeout.
- *Two-Rand-0:* Each request handler alternates reading 64 KB data blocks sequentially from two randomly chosen files. It accesses a random portion of each file from the beginning. This workload emulates applications that simultaneously access multiple sequential data streams.
- *Four-64KB-0:* Each request handler randomly chooses four files and reads a 64 KB random data block from each file.

We include two common file utilities and four real applications in our evaluation:

- *cp:* `cp` reads a source file chunk by chunk and copies the chunks into a destination file. We run `cp` on files of different sizes. We use `cp5` and `cp50` to represent test runs of `cp` on files of 5 MB and 50 MB respectively.
- *BSD diff:* `diff` is a utility that compares the content of two text files. BSD `diff` alternately accesses two files in making the comparison and it is more scalable than GNU `diff` [15]. We have ported BSD `diff` to our experimental platform. Again, we run it on files of different sizes. `diff5` works on 2 files of 5 MB each and `diff50` works on 2 files of 50 MB each.
- *Kernel build:* The kernel build workload includes compiling and linking a Linux 2.6.10 kernel. The kernel build involves reading from and writing to many small files.
- *TPC-H:* We evaluate a local implementation of the TPC-H decision support benchmark [31] running on the MySQL 5.0.17 database. The TPC-H workload consists of 22 SQL queries with 2 GB database size. The maximum database file size is 730 MB, while the minimum is 432 B. Each query accesses several database files with mixed random and sequential access patterns.
- *Apache hosting media clips:* We include the Apache web server in our evaluation. Typical web workloads often contain many small files. Since our work focuses on applications with substantially sequential access pattern, we use a workload containing a set of media clips, following the file size and access distribution of the video/audio clips portion of the 1998 World Cup workload [2]. About 9% of files in the workload are large video clips while the rest are small audio clips. The overall file size range is 24 KB–1418 KB with an average of 152 KB. The total dataset size is 20.4 GB. During the tests, individual media files are chosen in the client requests according to a Zipf distribution.
- *Index searching:* Another application we evaluate is a trace-driven index searching server using a dataset from the Ask Jeeves search engine [3]. The dataset supports search on 12.6 million web pages. The total dataset is approximately 18.5 GB and it includes a 522 MB index file. For each keyword in an input query, a lookup on the index file returns the location and size of this keyword's matching web page identifier list in the main dataset. The dataset is then accessed following a sequential access pattern. Multiple sequential I/O streams on the dataset are accessed alternately for each multi-keyword query. The search query words in our test workload are based on a trace recorded at the Ask Jeeves site in summer 2004.

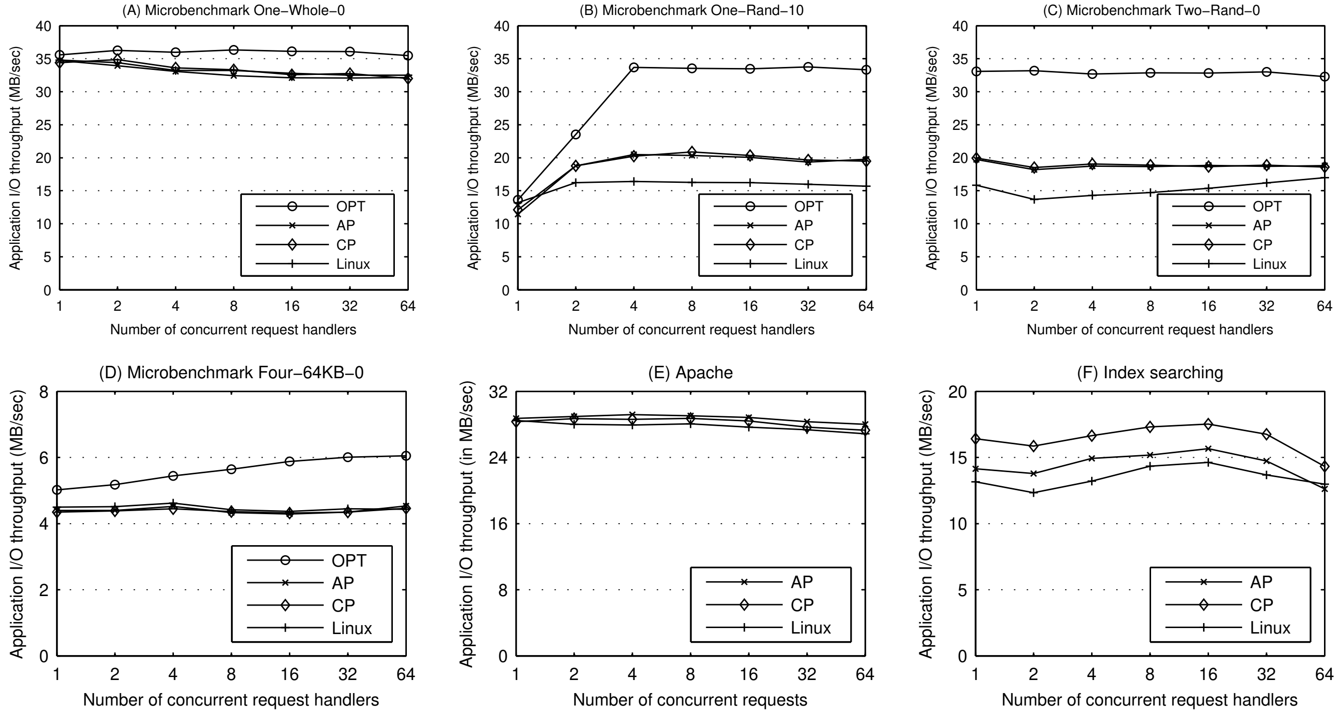

Figure 6: Performance of server-style workloads (four microbenchmarks and two real applications). The emulated optimal performance is presented only for the microbenchmarks.

These workloads contain a variety of different I/O access patterns. `cp`, kernel build, and the Apache web server request handler follow a simple sequential access pattern to access whole files. BSD `diff` accesses two files alternately and it also accesses whole files. TPC-H and the index searching server access partial files and they have more complex I/O access patterns. Figure 5 illustrates the access pattern of two TPC-H queries (Q2 and Q16) as well as a typical index searching request processing. We observe that the TPC-H Q2 and the index searching exhibit alternating sequential access patterns while TPC-H Q16 has a mixed pattern with a large amount of random accesses.

Among these workloads, the microbenchmarks, Apache web server, and index searching are considered as server-style concurrent applications. The other workloads are considered as non-server applications. Each server-style application may run at different concurrency levels depending on the input workload. In our experiments, each server workload involves a server application and a load generation client (running on a different machine) that can adjust the number of simultaneous requests to control the server concurrency level.

6.2 Base-Case Performance

We assess the effectiveness of the proposed technique by first showing the base-case performance. All the performance tests in this subsection are performed on the standalone IBM drive. We compare application performance under the following different kernel versions:

#1. *Linux:* The original Linux 2.6.10 kernel with a maximum prefetching depth of 128 KB.

#2. *CP:* Our proposed competitive prefetching strategy described in Section 4 (with the slow-start phase to accommodate random-access workloads).

#3. *AP:* Aggressive prefetching with a maximum prefetching depth of 800 KB (about twice that of average competitive prefetching).

#4. *OPT:* An oracle prefetching policy implemented using application disclosed hints. We have only implemented this policy for microbenchmarks. Each microbenchmark provides accurate data access pattern to the OS through an extended `open()` system call. The OS then prefetches exactly the amount of data needed for each sequential stream. The oracle prefetching policy is a hypothetical approach employed to approximate the performance of optimal prefetching.

We separately show the results for server-style and non-server workloads because server workloads naturally run at varying concurrency levels. Figure 6 illustrates the performance of server-style workloads (including four microbenchmarks, Apache hosting media clips, and index searching).

For the first microbenchmark (*One-Whole-0*), all practical policies (*Linux*, *CP*, and *AP*) perform equally well (Figure 6(A)). Anticipatory scheduling avoids I/O switching and files are accessed sequentially. However, Figures 6(B) and 6(C) demonstrate that anticipatory scheduling is not effective when significant processing time (think time) is present or when a request handler accesses multiple streams alternately. In such cases, our proposed competitive prefetching policy can significantly improve the overall I/O performance

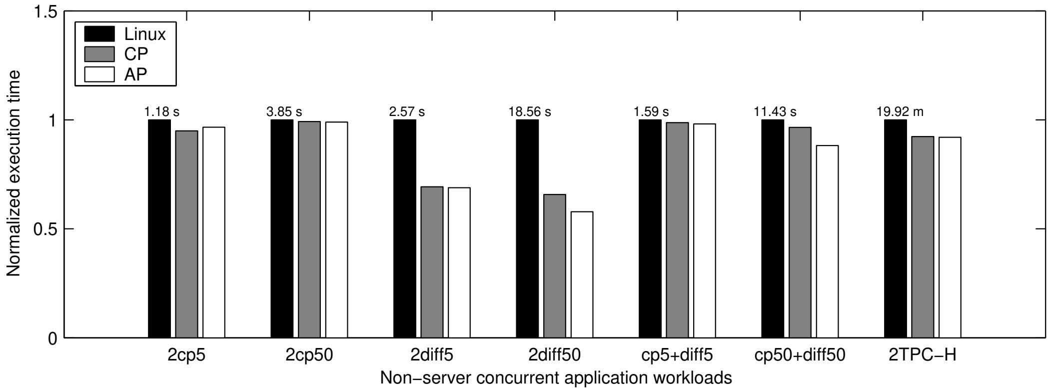

Figure 7: Normalized execution time of non-server concurrent applications. The values on top of the "Linux" bars represent the running time of each workload under the original Linux kernel (in minutes or seconds).

(16–24% and 10–35% for *One-Ran-10* and *Two-Rand-0* respectively). Figure 6(D) presents performance results for the random-access microbenchmark. We observe that all policies perform similarly. Competitive and aggressive prefetching do not exhibit degraded performance because the slow-start phase quickly identifies the non-sequential access pattern and avoids prefetching a significant amount of unnecessary data. Figures 6(A–D) also demonstrate that the performance of competitive prefetching is always at least half the performance of the oracle policy for all microbenchmarks.

Figure 6(E) shows the throughput of the Apache web server hosting media clips. Each request follows a strictly sequential data access pattern on a single file. Similarly to *One-Whole-0*, anticipatory scheduling minimizes the disk I/O switch cost and competitive prefetching does not lead to further performance improvement. Figure 6(F) presents the I/O throughput of the index search server at various concurrency levels. Each request handler for this application alternates among several sequential streams. Competitive prefetching can improve I/O throughput by 10–28% compared to the original Linux kernel.

We use the following concurrent workload scenarios for the non-server applications (`cp`, BSD `diff`, and TPC-H). `2cp5` represents the concurrent execution of two instances of `cp5` (copying a 5 MB file). `2cp50` represents two instances of `cp50` running concurrently. `2diff5` represents the concurrent execution of two instances of `diff5`. `2diff50` represents two instances of diff50 running concurrently. `cp5+diff5` represents the concurrent execution of `cp5` and `diff5`. `cp50+diff50` represents the concurrent execution of `cp50` and `diff50`. 2TPC-H represents the concurrent run of the TPC-H query series Q2→Q3→ ··· →Q11 and Q12→Q13→ ··· →Q22.

Figure 7 presents the execution time of the seven non-server concurrent workloads. There is relatively small performance variation between different prefetching policies for the concurrent execution of `cp`, because with anticipatory scheduling, files are read without almost any disk I/O switching even under concurrent execution. However, for `2diff5` and `2diff50` with alternating sequential access patterns, competitive prefetching reduces the execution time by 31% and 34% respectively. For 2TPC-H, the performance improvement achieved by competitive prefetching is 8%.

While competitive prefetching improves the performance of several workload scenarios, aggressive prefetching provides very little additional improvement. For index searching, aggressive prefetching performs significantly worse than competitive prefetching, since competitive prefetching maintains a good balance between the overhead of disk I/O switch and that of unnecessary prefetching. Furthermore, aggressive prefetching risks wasting too much I/O bandwidth and memory resources on fetching and storing unnecessary data.

6.3 Performance with Process-Context-Oblivious I/O Scheduling

The experimental results in the previous section show that during the concurrent execution of applications with strictly sequential access patterns, anticipatory scheduling avoids unnecessary I/O switch and competitive prefetching leads to little further performance improvement. Such workloads include *one-Whole-0*, Apache web server, `2cp5`, `2cp50`, `cp5+diff5`, and `cp50+diff50`. However, our discussion in Section 3.2 suggests that anticipatory scheduling is not effective in virtualized [17] or parallel/distributed I/O architecture due to the lack of knowledge on request issuing process contexts. In this subsection, we examine such performance effect under a virtual machine platform. Specifically, we use the Xen (version 3.0.2) [4] virtual machine monitor and the guest OS is a modified Linux 2.6.16 kernel. We have implemented our competitive prefetching algorithm in the guest OS.

Experiments are conducted using the standalone IBM hard drive. However, since the drive delivers higher I/O throughput under the Linux 2.6.16 kernel than under the Linux 2.6.10 kernel (as illustrated in Figure 4), the absolute workload performance shown here appears higher than that in the previous subsection.

Figures 8 and 9 show the performance of server-style and non-server workloads on a virtual machine platform. We present results only for the targeted workloads mentioned in the beginning of this subsection. In contrast to Figure 6(A), Figure 8(A) shows that competitive prefetching is very effective in improving the performance of the microbenchmark (up to 71% throughput enhancement compared to the original Linux kernel). Similarly, comparing Figure 8(B) with Figure 6(E) and Figure 9 with Figure 7 shows that competitive prefetching improves the performance of the Apache web server and the four non-server workloads. The perfor-

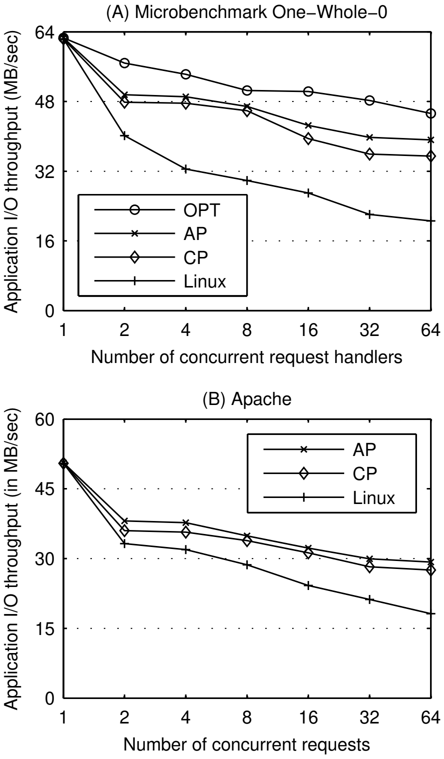

Figure 8: Performance of server-style workloads on a virtual machine platform where the I/O scheduling is oblivious to request-issuing process contexts. We present results only for workloads whose performance substantially differ from that in the base case (shown in Figure 6). For the microbenchmark, we also show an emulated optimal performance.

mance effects are due to ineffective anticipatory scheduling in a virtual machine environment. In addition, Figure 8(A) demonstrates that the performance of competitive prefetching is always at least half the performance of the oracle policy for the microbenchmark.

6.4 Performance on Different Storage Devices

The experiments of the previous sections are all performed on a standalone IBM disk drive. In this section, we assess the impact of different storage devices on the effectiveness of competitive prefetching. We compare results from three different storage devices characterized in Section 5: a standalone IBM drive, a standalone Seagate drive, and a RAID0 with two Seagate drives using an Adaptec RAID controller. Experiments in this section are performed on a normal I/O architecture (not on a virtual machine platform).

We choose the workloads that already exhibit significant performance variation among different prefetching policies on the standalone IBM drive. Figure 10 shows the normalized execution time of non-server workloads `2diff5` and `2diff50`. We observe that the competitive prefetching is effective for all three storage devices (up to 53% for `2diff50` on RAID0). In addition, we find that the performance improvement is more pronounced on the disk array than on standalone disks, because the disk array's competitive pre-

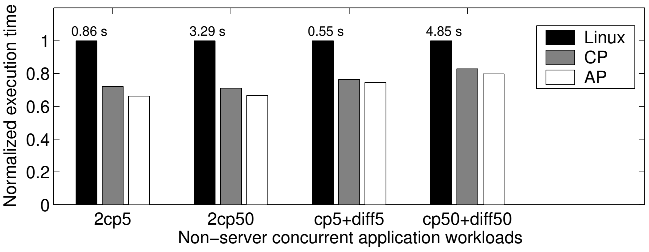

Figure 9: Normalized execution time of non-server concurrent applications on a virtual machine platform where the I/O scheduling is oblivious of request-issuing process contexts. We present results only for workloads whose performance substantially differ from that in the base case (shown in Figure 7). The values on top of the "Linux" bars represent the running time of each workload under the original Linux kernel (in seconds).

fetching depth is larger than that of standalone disks due to the array's larger sequential transfer rate.

Figure 11 shows the normalized *inverse of throughput* for two server applications (microbenchmark *Two-Rand-0* and index searching) on the three different storage devices. We use the inverse of throughput (as opposed to throughput itself) so its illustration is more comparable to that in Figure 10. Figure 11(A) shows that competitive prefetching can improve the microbenchmark I/O throughput by up to 52% on RAID0. In addition, the throughput of competitive prefetching is at least half the throughput of the oracle policy for all storage devices. Figure 11(B) shows that competitive prefetching can improve I/O throughput by 19–26% compared to the original Linux kernel. In contrast, aggressive prefetching provides significantly less improvement (up to 15%) due to wasted I/O bandwidth on fetching unnecessary data. For index searching, the performance improvement ratio on the disk array is not larger than that on standalone disks, since the sequential access streams for this application are typically not very long and consequently they do not benefit from the larger competitive prefetching depth on the disk array.

6.5 Performance of Serial Workloads

Though our competitive prefetching policy targets concurrent sequential I/O, we also assess its performance for standalone serially executed workloads. For the server applications (Apache web server and index search), we show their average request response time when each request runs individually.

Figure 12 presents the normalized execution time of the non-concurrent workloads. Competitive prefetching has little or no performance impact on `cp`, kernel build, TPC-H, and the Apache web server, since serial execution of the above applications exhibits no concurrent I/O accesses. However, performance is improved for `diff` and the index search server by 33% and 18% respectively. Both `diff` and the index search server exhibit concurrent sequential I/O (in the form of alternation among multiple streams). Query Q2 of TPC-H exhibits similar access patterns (as illustrated in Figure 5(A)). Our evaluation (Figure 12) suggests a 25% performance improvement due to competitive prefetching for

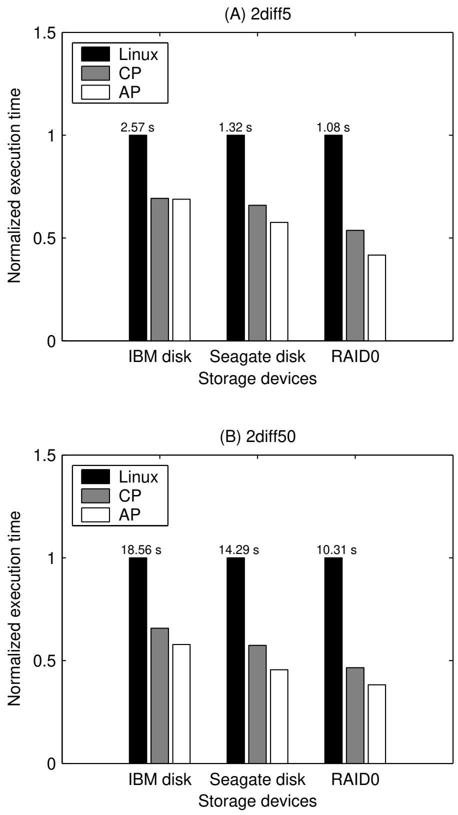

Figure 10: Normalized execution time of two non-server concurrent workloads on three different storage devices.

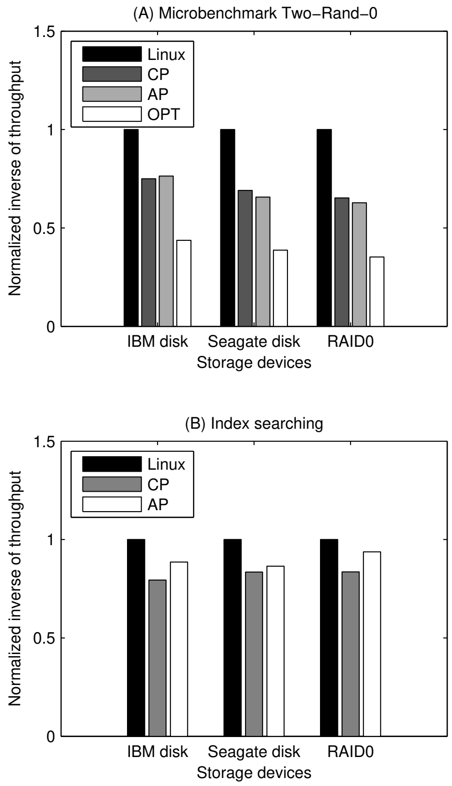

Figure 11: Normalized inverse of throughput for two server-style concurrent applications on three different storage devices. We use the inverse of throughput (as opposed to throughput itself) so that the illustration is more comparable to that in Figure 10. For the microbenchmark, we also show an emulated optimal performance. For clear illustration, we only show results at concurrency level 4. The performance at other concurrency levels are in similar pattern.

query Q2. Competitive prefetching did not lead to performance degradation for any of our experimental workloads.

6.6 Summary of Results

- Compared to the original Linux kernel, competitive prefetching can improve application performance by up to 53% for applications with significant concurrent sequential I/O. Our experiments show no negative side effects for applications without such I/O access patterns. In addition, competitive prefetching trails the performance of the oracle prefetching policy by no more than 42% across all microbenchmarks.

- Overly aggressive prefetching does not perform significantly better than competitive prefetching. In certain cases (as demonstrated for index search), it hurts performance due to I/O bandwidth and memory space wasted on fetching and storing unnecessary data.

- Since anticipatory scheduling can minimize the disk I/O switch cost during concurrent application execution, it diminishes the performance benefit of competitive prefetching. However, anticipatory scheduling may not be effective when: 1) applications exhibit alternating sequential access patterns (as in BSD `diff`, the microbenchmark Two-Rand-0, and index searching); 2) substantial processing time exists between consecutive sequential requests from the same process (as in the microbenchmark One-Rand-10); 3) I/O scheduling in the system architecture is oblivious to request-issuing process contexts (as in a virtual machine environment or parallel/distributed file system server). In such scenarios, competitive prefetching can yield significant performance improvement.

- Competitive prefetching can be applied to both standalone disk drives and disk arrays that exhibit disk-like characteristics on I/O switch and sequential data transfer. Disk arrays often offer higher aggregate I/O bandwidth than standalone disks and consequently their competitive prefetching depths are larger.

7. CONCLUSION AND DISCUSSIONS

To conclude, this paper presents the design and implementation of a competitive I/O prefetching technique targeting concurrent sequential I/O workloads. Our model and analysis shows that the performance of competitive prefetching (in terms of I/O throughput) is at least half the performance of the optimal offline policy on prefetching depth selection. In practice, the new prefetching scheme is most beneficial

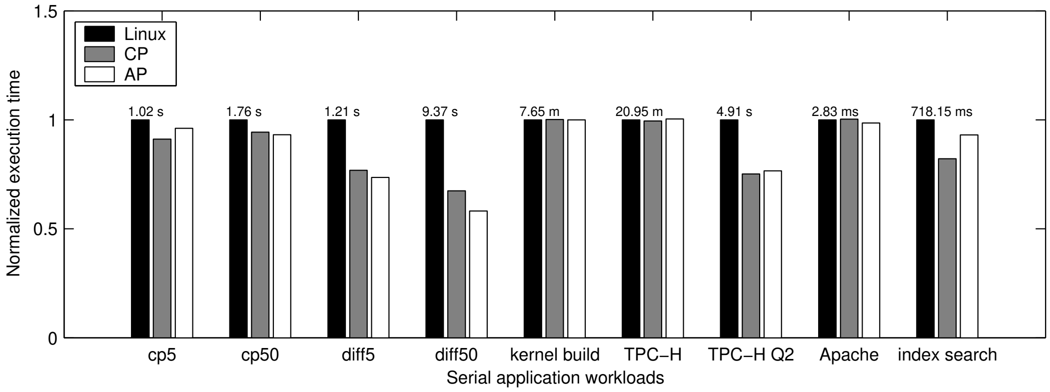

Figure 12: Normalized execution time of serial application workloads. The values on top of the "Linux" bars represent the running time of each workload under the original Linux prefetching (in minutes, seconds, or milliseconds).

when anticipatory I/O scheduling is not effective (*e.g.*, when the application workload has an alternating sequential data access pattern or possesses substantial think time between consecutive I/O accesses, or when the system's I/O scheduler is oblivious to I/O request process contexts).

We implemented the proposed prefetching technique in the Linux 2.6.10 kernel and conducted experiments using four microbenchmarks, two common file utilities, and four real applications on several disk-based storage devices. Overall, our evaluation demonstrates that competitive prefetching can improve the throughput of real applications by up to 53% and it does not incur noticeable performance degradation on a variety of workloads not directly targeted in this work. We also show that the performance of competitive prefetching trails that of an oracle offline prefetching policy by no more than 42%, affirming its competitiveness.

In the remainder of this section, we discuss implications related to our competitive prefetching policy. Prefetching and caching are complemental OS techniques that both use data locality to hide disk access latency. Unfortunately, they compete for memory resources against each other. Aggressive prefetching can consume a significant amount of memory during highly concurrent workloads. In extreme cases, memory contention may lead to significant performance degradation because previously cached or prefetched data may be evicted before they are accessed. The competitiveness of our prefetching strategy would not hold in the presence of high memory contention. A possible method to avoid performance degradation due to prefetching during periods of memory contention is to adaptively reduce the prefetching depth when increased memory contention is detected and reinstate competitive prefetching when memory pressure is released. The topic of adaptively controlling the amount of memory allocated for caching and prefetching has been previously addressed in the literature by Cao *et al.* [6] and Patterson *et al.* [26]. More recently, Kaplan *et al.* proposed a dynamic memory allocation method based on detailed bookkeeping of cost and benefit statistics [18]. Gill *et al.* proposed a cache management policy that dynamically partitions the cache space amongst sequential and random streams in order to reduce read misses [14]. In addition, in our previous work [22] we proposed a set of OS-level techniques that can be used to manage the amount of memory

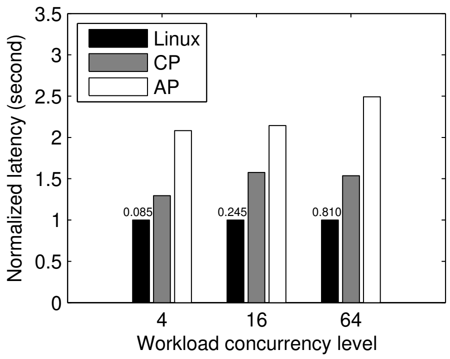

Figure 13: Normalized latency of individual I/O request under concurrent workload. The latency of an individual small I/O request is measured with microbenchmark Two-Rand-0 running in the background. The values on top of the "Linux" bars represent the request latency under the original Linux prefetching (in seconds).

dedicated to prefetching independently of the memory used by the buffer cache. Memory contention can also be avoided by limiting the server concurrency level with admission control or load conditioning [32, 33].

The proposed prefetching technique has been designed to minimize possible negative impact on applications not directly targeted by our work. However, such impact may still exist. In particular, large prefetching depths can lead to increased latencies for individual I/O requests under concurrent workloads. Figure 13 presents a quantitative illustration of this impact. Techniques such as priority-based disk queues [13] and semi-preemptible I/O [11] can be employed to alleviate this problem. Additional investigation is needed to address the integration of such techniques.

Acknowledgments

This work benefited from discussions with our colleagues at the University of Rochester (particularly Chen Ding, Michael Huang, and Michael L. Scott). We thank Pin Lu for helping us set up virtual machine test environment. We would also like to thank the anonymous EuroSys reviewers for their valuable comments that helped improve this work.

8. REFERENCES

[1] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Circus: Opportunistic Block Reordering for Scalable Content Servers. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pages 201–212, San Fancisco, CA, Mar. 2004.

[2] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35, HP Laboratories Palo Alto, 1999.

[3] Ask Jeeves Search. http://www.ask.com.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[5] R. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive Parallel Disk Prefetching and Buffer Management. *Journal of Algorithms*, 38(2):152–181, Aug. 2000.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the ACM SIGMETRICS*, pages 188–197, Ottawa, Canada, June 1995.

[7] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[8] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conf.*, pages 317–327, Atlanta, GA, Oct. 2000.

[9] E. Carrera and R. Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proc. of the 10th Int'l Symp. on High Performance Computer Architecture*, pages 130–141, Madrid,Spain, Feb. 2004.

[10] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[11] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design and Implementation of Semi-preemptible IO. In *Proc. of the Second USENIX Conf. on File and Storage Technologies*, pages 145–158, San Francisco, CA, Mar. 2003.

[12] K. Fraser and F. Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proc. of the USENIX Annual Technical Conf.*, pages 325–338, San Antonio, TX, June 2003.

[13] G. R. Ganger and Y. N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Transactions on Computers*, 47(6):667–678, June 1998.

[14] B. S. Gill and D. S. Modha. SARC: Sequential Prefetching in Adaptive Replacement Cache. In *Proc. of the USENIX Annual Technical Conf.*, pages 293–308, Anaheim, CA, Apr. 2005.

[15] GNU Diffutils Project, 2006. http://www.gnu.org /software/diffutils/diffutils.html.

[16] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 117 – 130, Banff, Canada, Oct. 2001.

[17] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of the USENIX Annual Technical Conf.*, pages 1–14, Boston, MA, June 2006.

[18] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive Caching for Demand Prepaging. In *Proc. of the Third Int'l Symp. on Memory Management*, pages 114–126, Berlin, Germany, June 2002.

[19] D. Kotz, S. B. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[20] H. Lei and D. Duchamp. An Analytical Approach to File Prefetching. In *Proc. of the USENIX Annual Technical Conf.*, Anaheim, CA, Jan. 1997.

[21] C. Li, A. Papathanasiou, and K. Shen. Competitive Prefetching for Data-Intensive Online Servers. In *the First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, Boston, MA, Oct. 2004.

[22] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. In *Proc. of the 4th USENIX Conf. on File and Storage Technologies*, pages 253–266, San Fancisco, CA, Dec. 2005.

[23] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. In *Proc. of the 4th USENIX Symp. on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[24] A. Papathanasiou and M. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

[25] A. E. Papathanasiou and M. L. Scott. Energy Efficient Prefetching and Caching. In *Proc. of the USENIX Annual Technical Conf.*, Boston, MA, June 2004.

[26] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, Dec. 1995.

[27] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, Mar. 1994.

[28] E. Shriver, A. Merchant, and J. Wilkes. An Analytical Behavior Model for Disk Drives with Readahead Caches and Request Reordering. In *Proc. of the ACM SIGMETRICS*, pages 182–192, Madison, WI, June 1998.

[29] E. Shriver, C. Small, and K. A. Smith. Why Does File System Prefetching Work? In *Proc. of the USENIX Annual Technical Conf.*, pages 71–84, Monterey, CA, June 1999.

[30] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed Multi-Process Prefetching and Caching. In *Proc. of the ACM SIGMETRICS*, pages 100–114, Seattle, WA, June 1997.

[31] TPC-H Benchmark. http://www.tpc.org.

[32] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, Oct. 2003.

[33] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 230–243, Banff, Canada, Oct. 2001.

[34] T. Yeh, D. Long, and S. A. Brandt. Using Program and User Information to Improve File Prediction Performance. In *Proc. of the Int'l Symposium on Performance Analysis of Systems and Software*, pages 111–119, Tucson, AZ, Nov. 2001.