Competitive Prefetching for Data-Intensive Online Servers*

Chuanpeng Li, Athanasios E. Papathanasiou, and Kai Shen
{cli, papathan, kshen}@cs.rochester.edu
Department of Computer Science, University of Rochester

Technical Report #840

Abstract

Recent studies on operating system support for highly concurrent online servers mostly target CPU-intensive workloads with light disk I/O activities. However, an important class of online applications that access a large amount of disk-resident data, such as the index searching of large-scale Web search engines, has received limited attention. This paper examines operating system techniques to improve the throughput of data-intensive online servers under concurrent execution. Our contributions are two-fold. First, we propose a *competitive prefetching* strategy that balances the overhead of disk I/O switching and the wasted I/O bandwidth of prefetching unnecessary data. Second, we enhance the operating system memory management to reduce *prefetching-incurred page thrashing* at high concurrency levels. Our enhancements include the introduction of a new cache dedicated to prefetched pages and a novel page reclamation algorithm for it.

We have implemented the proposed techniques in the Linux 2.6.3 kernel and conducted experiments based on microbenchmarks and three real applications (an index searching server, the Apache Web server, and a genetic sequence database). Our evaluation results demonstrate that competitive prefetching can improve the throughput of real applications by up to 47% at low concurrency while the enhanced memory management scheme can produce up to a four-fold performance enhancement at high concurrency. The performance improvement is achieved without any application changes.

1 Introduction

Rapidly emerging Internet services allow interactive access from a large number of concurrent users. Inside these online servers, simultaneous requests are often managed by multiple processes or threads. Earlier studies have identified a number of overheads incurred by server concurrency management, including TLB misses, scheduling overhead, and lock contention. Techniques, such as event-driven concurrency management [26, 34] and user-level threads [33], have been proposed to avoid kernel-level context switching and to reduce the overhead of synchronization. However, such techniques mostly target CPU-intensive workloads with light disk I/O activities (*e.g.*, the typical Web server workload and application-level packet routing). In comparison, system-level support for concurrent online workloads that access a large amount of disk-resident data has received limited attention. Examples of such servers include large-scale Web search engines [4] that support interactive search on terabytes of indexed Web pages and the bioinformatics database GenBank [5] that allows online queries on over 100 GB genetic and protein sequence data.

For data-intensive online servers, disk I/O performance dominates the overall system throughput when the dataset size far exceeds the available server memory. During concurrent execution, data access of one request handler can be frequently interrupted by other active request handlers in the server. Such a phenomenon, which we call *disk I/O switching*, may severely affect I/O efficiency due to long disk seek and rotational delays. Anticipatory disk I/O scheduling [15] alleviates this problem by temporarily idling the disk so that consecutive I/O requests that belong to the same request handler are serviced without interruption. However, anticipatory scheduling may not be effective when substantial think time exists between consecutive I/O requests or when the request handler has to perform some interleaving synchronous I/O that does not exhibit strong locality.

In this paper, we investigate kernel-level techniques to improve the performance of data-intensive online servers. First, we examine the effectiveness of using a large I/O prefetching depth with the goal of reducing the disk I/O switching overhead. Due to insufficient knowledge of application data access patterns, aggressive prefetching must control the amount of I/O bandwidth wasted on prefetching unnecessary data. Using a simple model, the achieved I/O throughput of our strategy (called *competitive prefetching*) is shown to be at least half that of the optimal offline prefetching.

I/O prefetching can consume a significant amount of server memory, especially for highly concurrent online workloads. Such memory contention may result in the eviction of prefetched pages before they are accessed, or *prefetching-incurred page thrashing*. To minimize prefetching-incurred page thrashing, we manage prefetched but not-yet-referenced pages separately from other pages in the memory system and we propose a novel page reclamation policy for such pages. Our algorithm gives higher priority to pages whose prefetching processes[1] have had less

*This work was supported in part by the National Science Foundation grants CCR-0306473 and ITR/IIS-0312925.

[1]Unless stated otherwise, a process represents a process or a kernel thread in the rest of this paper.

chance to run since the time of the prefetch operation.

The rest of the paper is organized as follows. Section 2 discusses the characteristics of targeted data-intensive online servers and describes the existing OS support. Section 3 and Section 4 present the design of our proposed kernel-level techniques and their implementation in the Linux 2.6.3 kernel. Section 5 provides the performance results based on microbenchmarks and three real applications. Section 6 discusses several remaining issues. Section 7 describes the related work and Section 8 concludes the paper.

2 Background

2.1 Targeted Applications

Our work focuses on online servers supporting highly concurrent workloads that access a large amount of locally attached disk-resident data. In such servers, each incoming request is serviced by a request handler upon arriving from the network (shown in Figure 1). The request handler then repeatedly accesses disk data and consumes the CPU before completion. A request handler may block if the needed resource is unavailable. While request handlers consume both disk I/O and CPU resources, the overall server throughput is often dominated by the disk I/O performance when the application data size far exceeds the available server memory.

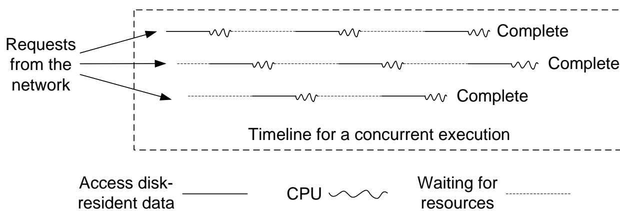

Figure 1: Concurrent application execution in a data-intensive online server.

We make the following assumptions on our targeted workloads: 1) request handlers perform mostly read-only I/O when accessing disk-resident data; and 2) a substantial fraction of the disk I/O follows a sequential access pattern. Note that we allow request handlers to perform some interleaving I/O that deviates from the sequential access pattern. Applications that retrieve files in request handlers, such as Web and FTP servers, satisfy our assumptions. More complex applications are usually carefully constructed to follow mostly sequential access patterns in order to achieve efficient disk I/O.

In many data-intensive online servers, request handlers read a portion of the desired data at a time instead of retrieving the complete data into application-level buffers at once. This is due to at least the following reasons. First, applications such as the index searching scan across the hosted dataset and look for a certain number of matches depending on the data content. Consequently, request handlers for these applications do not know a priori the exact amount of desired data. Second, applications that employ memory-mapped I/O do not directly initiate I/O operations while the OS kernel does not know the exact amount of data needed by the application. Third, retrieving the complete data into application-level buffers at once would result in significant double buffering and thus increase memory contention. For instance, the Apache Web server reads an 8 KB chunk at a time when transferring large Web objects [2].

2.2 Existing OS Support

Using Linux 2.6.3 kernel as an example, we briefly describe existing OS support that is mostly relevant to our work. This includes memory management, I/O prefetching, and the disk scheduling. LRU-style page cache replacement algorithms are commonly used in modern operating systems. Linux employs two LRU lists: an *active* list used to cache hot pages and an *inactive* list used to cache cold pages. Prefetched pages are initially attached to the tail of the inactive list. When the available memory falls below a reclamation threshold, an aging algorithm moves pages from the active list to the inactive list and cold pages at the head of the inactive list are considered first for eviction.

Linux supports a conservative prefetching algorithm that may read up to 32 pages (128 KB) in advance during sequential file accesses. For each sequential file stream, two access windows, the *current* and the *ahead* windows, are used to control prefetching. The application consumes data in the current window while the operating system prefetches data into the ahead window. When a page in the ahead window is accessed, a new I/O prefetch operation is initiated and the ahead window becomes the new current window. Prefetching starts with an initial depth of 16 pages (64 KB) and soon increases to 32 pages if the access pattern is deemed as sequential. Typical series of prefetching depths used for a sequential access stream are: 16 pages $\rightarrow$ 18 pages $\rightarrow$ 32 pages $\rightarrow$ 32 pages $\rightarrow$ ... The Free-BSD FFS performs prefetching somewhat differently, but its maximum prefetching depth defaults similarly to 32 file system blocks.

Linux supports an elevator-style disk scheduler that is similar to cyclic SCAN. In addition, it provides an implementation of *anticipatory scheduling* [15]. At the completion of an I/O request, the anticipatory disk scheduler may choose to keep the disk idle for a short period of time even when there are pending requests. The scheduler does so in anticipation of a new I/O request from the process that issued the just completed request, which often requires little or no seeking from the current disk head location. Anticipatory scheduling leads to significant performance improvement for concurrent workloads where all requests, issued by an individual process, exhibit strong locality.

[2]When the OS supports the `sendfile` system call, the Apache Web server utilizes this system call to directly transfer the content of a file into an outgoing network socket, thus removing the application-level buffering.

2.3 Limitation of Existing OS Support

Existing OS mechanisms are inadequate for efficient support of data-intensive online servers. For example, frequent I/O switching among multiple sequential access streams combined with conservative I/O prefetching may result in a large number of piecemeal disk I/O accesses. This leads to severe performance penalties due to the high seek and rotational delays of modern disks. Although anticipatory scheduling can alleviate such a problem to a certain extent, its effectiveness is limited when substantial think time exists between consecutive I/O requests. The anticipation may also be rendered ineffective when a request handler has to perform some interleaving synchronous I/O that does not exhibit strong locality. Such a situation arises when a request handler simultaneously accesses multiple data streams. For example, the index searching server needs to produce the intersection of multiple sequential keyword indexes when answering multi-keyword queries. In addition, the locality of I/O requests from an individual process may also be interrupted by random page faults during high memory pressure.

I/O prefetching can create significant memory demands. Such demands can result in severe contention at high concurrency levels, where many request handlers in the server are simultaneously competing for memory pages for prefetching. LRU-style replacement policies are misplaced in such a context because they fail to give enough priority to pages that are prefetched but not yet referenced. When contention exceeds a certain threshold, a prefetched page may be evicted before its prefetching request handler gets a chance to access it. Such a page will have to be fetched again, wasting I/O bandwidth. For instance, assume page p_1 is prefetched and then accessed at t_1 while page p_2 is prefetched at t_2 and it has not yet been accessed. If t_1 occurs after t_2, then p_2 may be replaced first under the LRU though it is more likely to be useful in the near future. We call such a phenomenon *prefetching-incurred page thrashing*. In addition to wasting resources on repetitive fetching, page thrashing also creates many inefficient single-page fetches. When Linux encounters a page that has been prefetched earlier but then evicted, a single page read is initiated on that page and the request handler blocks. Such a disruption results in a high I/O switching rate and thus significantly affects the disk I/O efficiency. Figure 2 illustrates the performance of a trace-driven index searching benchmark under the original Linux 2.6.3 kernel. The details about the benchmark and the experimental settings can be found in Section 5.1. We show the I/O throughput and page thrashing rate for up to 512 concurrent request handlers in the server. Because of frequent I/O switching, the application I/O throughput is no more than 16.3 MB/sec, which is below one third of the sequential disk transfer rate. At high execution concurrency, the I/O throughput drops dramatically as a result of high page thrashing rate. Particularly at the concurrency level of 512, the I/O throughput degrades to 0.8 MB/sec while prefetching-incurred page thrashing rate increases to 256 pages/sec. We also note that the I/O throughput initially increases as the concurrency level climbs up to 32. This is due to at least two factors: 1) more overlapping of I/O and computation at higher concurrency levels; and 2) lower average seek distance when the disk scheduler can choose from more concurrent requests for seek reduction.

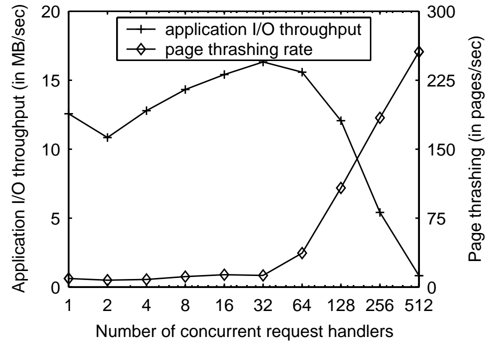

Figure 2: I/O throughput and prefetching-incurred page thrashing rate of a trace-driven index searching benchmark under the original Linux 2.6.3 kernel.

Considering the limitations of existing OS support, this paper proposes new techniques to improve the performance of data-intensive online servers, including balanced I/O prefetching (Section 3) and memory management against prefetching-incurred page thrashing (Section 4).

3 Competitive Prefetching

Under concurrent execution, I/O throughput depends mostly on the granularity of I/O switching, defined as the average amount of sequentially accessed data between consecutive I/O switches. Improving the I/O efficiency can be accomplished by employing a large I/O prefetching depth. A larger prefetching depth results in less frequent I/O switching, and consequently yields fewer disk seeks per time unit. Unfortunately, kernel-level prefetching may retrieve unnecessary data due to the lack of knowledge on how much data are desired by the application. Such a waste tends to be magnified by aggressive prefetching policies. Hence, there is a tradeoff in deciding the I/O prefetching depth: a conservative prefetching strategy may *produce high I/O switching overhead* while over-aggressive prefetching may *waste too much I/O bandwidth on fetching unnecessary data*. A good prefetching depth must maintain the balance between the above two costs.

We analyze the problem using a simple model. We assume a request handler sequentially accesses disk-resident data of total size S_{tot}. Let the disk transfer rate be R_{tr}. Also let the combined seek and rotational delay, or the I/O switching cost, be C_{switch}. The minimal disk resource consumption (in time) for request processing includes a single

I/O switch and the transfer time for S_{tot} data:

$$\mathcal{C}_{min} = S_{tot}/R_{tr} + C_{switch} \quad (1)$$

This minimal cost can only be achieved by the optimal offline strategy where S_{tot} is known. Kernel-level prefetching, however, does not have this knowledge. We assume the OS employs a fixed prefetching depth S_p[3]. The total I/O switching cost for request processing can be derived as:

$$\mathcal{C}_{tot_switch} = \lceil S_{tot}/S_p \rceil \cdot C_{switch} \leq (S_{tot}/S_p + 1) \cdot C_{switch} \quad (2)$$

The wasted time for fetching unnecessary data can be bounded by the cost of a single prefetch operation:

$$\mathcal{C}_{waste} \leq S_p/R_{tr} \quad (3)$$

Therefore, the total disk resource (in time) consumed by request processing is bounded by the following:

$$\begin{aligned} \mathcal{C}_{total} &= S_{tot}/R_{tr} + \mathcal{C}_{tot_switch} + \mathcal{C}_{waste} \\ &\leq S_{tot}/R_{tr} + (S_{tot}/S_p + 1) \cdot C_{switch} + S_p/R_{tr} \end{aligned} \quad (4)$$

Based on Equations (1) and (4), we find that:

$$\text{if } S_p = C_{switch} \cdot R_{tr}, \quad \text{then } \mathcal{C}_{total} \leq 2 \cdot \mathcal{C}_{min} \quad (5)$$

This result allows us to design a prefetching strategy with bounded worst-case performance, shown below:

> When the prefetching depth is equal to the amount of data that can be sequentially transfered within a single I/O switching period, the total disk resource consumption is at most twice that of the optimal offline strategy.

This also infers that the I/O throughput under this strategy is at least half the optimal performance. *Competitive* strategies have been used in the context of memory paging [31] and multiprocessor synchronization [19] to name algorithms whose performance can be shown to be no worse than some constant factor of an optimal offline strategy. Following their terminology, we name our prefetching strategy *competitive prefetching*.

Since the I/O switching time depends on the seek/rotational distance and the sequential transfer rate depends on the data location (due to zoning on modern disks), the competitive prefetching depth may dynamically change at runtime. In our strategy, the OS first calculates 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 $f_{transfer}$). This measurement is done offline, at disk installation time or OS boot time. During runtime, the OS maintains an exponentially-weighted moving average of the disk seek distance (denoted by D_{seek}). For each prefetch operation starting at disk location $L_{transfer}$, the predicted seek time and data transfer rate are $f_{seek}(D_{seek})$ and $f_{transfer}(L_{transfer})$ respectively. The rotational delay is much harder to predict at runtime. In particular, modern disks support out-of-order transfer for large I/O requests to hide the rotational delay. The intuition is that a read can start at any track location if the desired data spans across the whole track. Additionally, the rotational time may be partially hidden through techniques such as *freeblock scheduling* [23]. Since we do not need a very accurate estimation for our purpose, we simply use the average rotational delay between two random track locations (*i.e.*, the time it takes the disk to spin half a revolution).

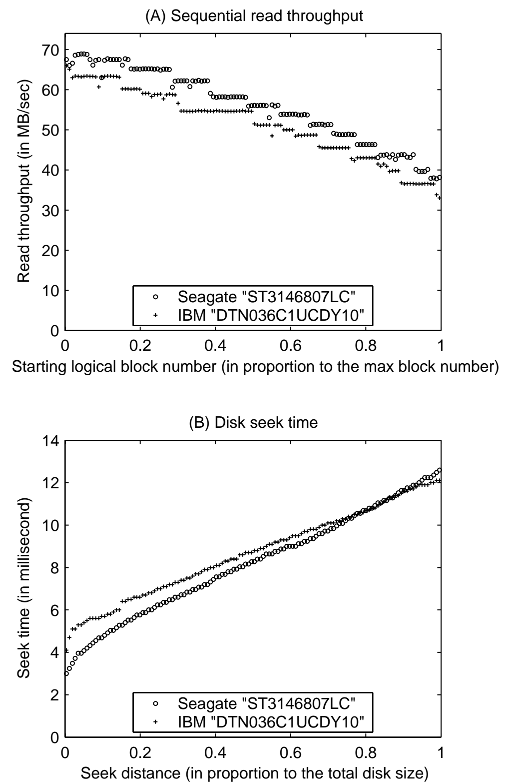

Figure 3: Sequential read throughput and seek time for two SCSI drives.

We quantitatively assess the competitive prefetching depth for two specific disk drives: a 36.4 GB IBM 10 KRPM SCSI drive and a 146 GB Seagate 10 KRPM SCSI drive behind an Adaptec RAID controller. We measure the disk 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 drive and it could be easily performed at disk installation time. Figure 3 shows the measured sequential read throughput and the seek time for these

[3]This is not strictly true for Linux 2.6.3 since prefetching starts with a relatively small initial depth, which yields more frequent I/O switching at the initial stage. However, the number of resulted additional I/O switches is bounded by a small constant for each request processing.

two drives. For the IBM drive, the average transfer speed is 51.3 MB/sec; the average seek time (between two independent random disk locations) is 7.53 ms; and the average rotational delay is 3.00 ms. Therefore about 540 KB of data can be transfered in an average seek and rotational delay. In comparison, the default maximum prefetching depth in Linux 2.6.3 and Free-BSD FFS is only 128 KB (32 pages). Note that the average seek distance is often smaller at higher concurrency levels since the disk scheduler can choose from more concurrent requests for seek reduction.

Competitive strategies only address the worst-case performance while a good prefetching policy should provide high average-case performance, or at least it must not decrease the performance of the original kernel. Since the prefetching depth in the proposed competitive prefetching policy is typically larger than that of the original Linux kernel, it may exhibit inferior performance for *short* access streams, including small-size random accesses. We remedy the problem by employing a relatively small initial prefetching depth as in the original kernel (*i.e.*, 16 pages). When the data access pattern is deemed as sequential by the OS, the depth for each additional prefetching operation is doubled until it reaches the desired competitive prefetching depth. We call this the *slow-start* phase.

4 Memory Management Against Page Thrashing

Prefetching-incurred page thrashing can occur during highly concurrent request execution, which often leads to significant performance degradation. Aggressive prefetching strategies such as our proposed competitive prefetching may further exacerbate the problem by inducing higher memory pressure. The intuitive solution for reducing prefetching-incurred page thrashing is to increase the priority of prefetched but not-yet-referenced pages. For instance, prefetched pages may be placed in the active list of the page cache. However, pages in the active list are still subject to LRU-style replacement policies, which are ill-equipped to handle prefetched pages (as explained in Section 2.3). A more radical approach would be to pin prefetched pages in memory until they are referenced. Unfortunately, this may create permanently-pinned ghost pages when a portion of the prefetched pages are not needed by the application.

We enhance the existing OS memory management to reduce prefetching-incurred page thrashing. Our modifications include the addition of a *prefetch cache* used to keep track of prefetched pages and the proposal of a novel page reclamation policy specific for this cache (Section 4.1). We also enforce an additional threshold of the maximum prefetching depth based on the available memory and the number of concurrent prefetching streams at high concurrency level. (Section 4.2).

4.1 Prefetch Cache

We use a new data structure, called *prefetch cache*, to manage prefetched but not-yet-referenced pages. Prefetched pages are initially placed in the prefetch cache. When a page is referenced, it is moved from the prefetch cache to the kernel's inactive LRU list and is thereafter controlled by the kernel's default page reclamation policy. The basic concept of the prefetch cache was used earlier by Papathanasiou and Scott [28] to control prefetching aggressiveness with the goal of improving energy efficiency. Our design in this paper targets at reducing prefetching-incurred page thrashing for highly concurrent data-intensive servers.

In Linux, page reclamation is initiated when the number of free pages falls below a certain threshold. Pages are first moved from the active list to the inactive list and then the OS scans the inactive list to find candidate pages for eviction. In our design, to avoid committing too much memory for the prefetch cache and leaving too little for the rest of the system, some not-yet-referenced pages in the prefetch cache would have to be moved to the inactive list at the presence of high memory pressure. We have augmented the Linux page reclamation logic to perform such a task. Our strategy contains two parts: an adaptive ceiling of the maximum prefetch cache size and a second-chance reclamation algorithm.

Prefetch cache size ceiling. In order to determine an appropriate size for the prefetch cache, the system maintains several memory performance statistics along with their exponentially weighted moving averages. Specifically, we keep track of the miss rate on pages that used to reside in the LRU cache (*LRU miss*) as well as the miss rate on pages that were prefetched and then evicted without being accessed (*prefetching miss*). Pages are evicted from the prefetch cache when the number of pages in the cache exceeds the the prefetch cache size ceiling. To monitor misses on recently evicted pages, the system maintains a new structure, called the *eviction history*, that records a short history of evicted pages along with flags that identify their status at the time of eviction (*e.g.*, whether they were part of the LRU cache or were prefetched but not-yet-accessed). When a page miss occurs, the eviction history is checked to identify its status.

To achieve high performance, we aim to minimize the overall miss rate by controlling both the LRU miss and the prefetching miss. Suppose variable x is the prefetch cache size ceiling, we use function $f_{pre}(x)$ and $f_{lru}(x)$ to represent prefetching miss rate and LRU miss rate respectively. Then our objective is to minimize the overall miss rate on the server:

$$f_{tot}(x) = f_{pre}(x) + f_{lru}(x)$$

The upper bound of x is limited by the maximum amount of memory that can be used for system cache x_{max}, which is roughly the physical memory size minus those memory occupied by the kernel.

$$0 \leq x \leq x_{max}$$

This is a linearly constrained nonlinear optimization problem. Technically, if the prefetch cache size increases, the prefetch miss rate $f_{pre}(x)$ will decrease. However, as prefetch cache size goes beyond the demand of the workload, the prefetch miss rate will level off and $f_{pre}(x)$ will approach zero. Since similar analysis can also be applied to $f_{lru}(x)$, it is acceptable for us to assume that $f_{tot}(x)$ is concave upward. With this assumption, it can be proved that the minimization problem has a global unique solution [14]. We make use of an incremental greedy algorithm called *gradient descent* [30] to address this problem. Given a curve of a cost function, gradient descent means going downhill toward the bottom and taking steps proportional to the gradient at the current point. In our implementation, we initialize the prefetch cache size ceiling, x, to be 25% of the total physical memory. It is then periodically (*e.g.*, once per 4 seconds) adjusted based on gradient descending of $f_{tot}(x)$. The time interval of each period is an epoch. In the first epoch, we randomly increase or decrease x by an *adjustment unit*. We calculate the change of overall miss rate in each subsequent epoch. If the overall miss rate increases obviously, we change x with an increase or decrease operation which is different from the operation in the previous epoch. Otherwise, we make the same adjustment to x as in the previous epoch. For example, if we previously increase x by Δx and it causes $f_{tot}(x+\Delta x) - f_{tot}(x) > threshold$, we will decrease x in the next epoch by an adjustment unit. Otherwise, we keep increasing x. This implementation is a simple approximation of gradient descent of $f_{tot}(x)$. Our experiment shows that when the adjustment unit is chosen properly, the prefetch cache size ceiling, x, will quickly converge to a narrow interval. The adjustment unit is currently set as the low free memory threshold used by the paging daemon to trigger reclamation.

Second-chance page reclamation. Pages in the prefetch cache are organized in a FIFO queue. Prefetched pages are initially attached to the tail of the queue. At the time of prefetch, the OS also marks each page with the `pid` of the prefetching process and a timestamp indicating the prefetching time. When the reclamation logic is triggered, we examine pages from the queue head. Pages that were prefetched by a currently inactive process are immediately moved to the inactive list. A process is considered inactive if it has exited or it is in the `ZOMBIE` state. If a page's prefetching process is still active, the page prefetching timestamp is compared against the last execution time of the prefetching process. If this comparison shows that the process has had significant opportunity to reference the prefetched page but it did not, it is likely that the prefetching process is not interested in the page at all, so we move the page to the inactive list. Otherwise, the page is given a second chance in the prefetch cache: its `pid` and timestamp fields are invalidated and it is re-attached to the tail of the prefetch cache queue. When it moves again to the queue head for reclamation, it will be immediately moved to the inactive list due to its invalid `pid` and timestamp fields. Figure 4 provides a high-level overview of our enhanced memory management.

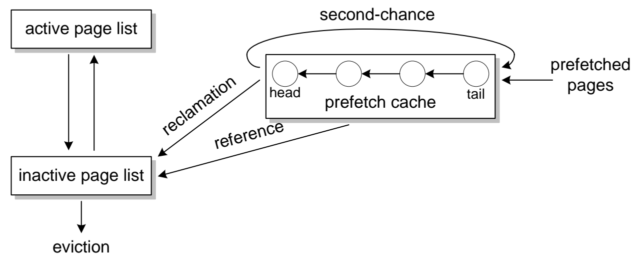

Figure 4: Overview of our enhanced memory management.

In our current implementation, the extra `pid` and timestamp fields occupy 4 bytes each in the page descriptor. Therefore they incur 0.2% space overhead (8 bytes per 4 KB page). This may be improved by using 2 bytes for each of the `pid` and timestamp fields. Such a scheme creates a problem of imprecise timing and further investigation is needed to assess its feasibility. An additional space overhead is that used by the eviction history. We can limit this overhead by controlling the number of entries in the eviction history. A smaller history size may be adopted at the expense of the accuracy of the memory performance statistics. If a 40-byte data structure is used to record information about each evicted page and we limit the eviction history size to 20% of the total number of physical pages, the space overhead for the eviction history would be 0.2% of the total memory space.

4.2 Constrained Prefetching at High Concurrency Levels

Simply adjusting the page cache reclamation policy may not be sufficient to avoid page thrashing at very high concurrency levels. When a large number of request handlers run in the server simultaneously, memory contention created by the active prefetching streams alone can create page thrashing. To deal with this problem, we enforce an additional maximum per-stream prefetching depth $\hat{S}_p = \frac{P_{mem}}{N_{stream}}$, where P_{mem} is the ceiling of the prefetch cache size and N_{stream} is the number of active prefetching streams in the server. In implementation, we approximate N_{stream} with the total number of opened files in user level processes. Our test shows this approximation is reasonable for applications we experimented.

This additional threshold limits the prefetching depth at high concurrency levels, which may increase the I/O switching rate and thus lower the I/O efficiency. However, this is necessary since the penalty of page thrashing far outweighs the lost I/O efficiency.

5 Experimental Evaluation

We assess the effectiveness of our proposed techniques on improving the performance of data-intensive online servers. Experiments were conducted on a Linux cluster connected

Benchmark/workload	Whole-file access?	Streams per request	Memory Footprint	Total data size	Min-mean-max size of sequential access streams
Microbenchmark: One-Whole-0	Yes	Single	1 MB/request	24.0 GB	4 MB-4 MB-4 MB
Microbenchmark: One-Rand-10	No	Single	1 MB/request	24.0 GB	64 KB-2 MB-4 MB
Microbenchmark: Two-Rand-0	No	Multiple	1 MB/request	24.0 GB	64 KB-2 MB-4 MB
Microbenchmark: Four-64KB-0	No	N/A	1 MB/request	24.0 GB	N/A
Index searching	No	Multiple	Unknown	19.0 GB	Unknown
Apache hosting media clips	Yes	Single	Unknown	20.4 GB	24 KB-152 KB-1418 KB
BLAST genetic sequence database	Yes	Unknown	Unknown	10.9 GB	Unknown

Table 1: Benchmark statistics.

by Gigabit Myrinet. Each node is equipped with two 2 GHz Xeon processors, 2 GB memory, a 36.4 GB IBM 10 KRPM SCSI drive, and a 146 GB Seagate 10 KRPM SCSI drive. Each experiment involves a server and a load generation client. The client can adjust the number of simultaneous requests to control the server concurrency level (*e.g.*, the number of concurrent processes or threads).

The evaluation results are affected by several factors, including the server memory size, the application dataset size, and the average size of prefetching streams. Our strategy is to first demonstrate the performance at a typical setting and then explicitly evaluate the impact of various factors. Section 5.1 describes the benchmarks used in our evaluation. Sections 5.2 and 5.3 illustrate the overall performance of examined benchmarks at a typical workload setting with 512 MB server memory. Section 5.4 presents profiling results on the I/O request size, prefetching-incurred page thrashing, and the CPU usage. Section 5.5 evaluates the performance impact of several factors in the evaluation setting. Section 5.6 summarizes the performance results.

5.1 Evaluation Benchmarks

Our evaluation benchmark suite contains both microbenchmarks and real applications. All microbenchmarks access a dataset of 6000 4 MB disk-resident files. At the arrival of each request, the server spawns a thread to process it. The user-level memory footprint of each request handler is 1 MB. We explore the performance of four microbenchmarks with different I/O access patterns. 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, a 64 KB chunk) and a think time delay during the processing of each request (0 and 10 ms). We use a descriptive naming convention to refer to each benchmark. For example, <Two-Rand-0> describes a microbenchmark with request handlers that access a random portion of two files with 0 ms think time delay during request processing. Accesses within a portion of the file are always sequential in nature. We use the following microbenchmarks in the evaluation:

- *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 file up to a random total size (evenly distributed between 64 KB and 4 MB). Additionally, we add a 10 ms think time at four random points during the processing of each request. The think times are used to emulate possible delays during request processing and may cause the anticipatory disk scheduler to timeout.
- *Two-Rand-0:* Each request handler alternates reading 64 KB data blocks from two randomly chosen files. It accesses a random portion of each file. 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 also include three real applications in our evaluation:

- *Index searching:* We acquired an earlier prototype of the index searching server and a dataset from the Web search engine Ask Jeeves [4]. The dataset contains the search index for 12.6 million Web pages. It includes a 522 MB mapping file that maps MD5-encoded keywords to proper locations in the search index. The search index itself is approximately 18.5 GB, divided into 8 partitions. For each keyword in an input query, a binary search is first performed on the mapping file and then the search index is accessed following a sequential access pattern. Multiple prefetching streams on the search index are accessed for each multi-keyword query. The search query words in our test workload are based on a one-week trace recorded at the Ask Jeeves online site in early 2002.
- *Apache hosting media clips:* We include the Apache Web server in our evaluation. Typical Web workloads often contain too small files. Since our work focuses on data-intensive applications, 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 [3]. About 9% of files in the workload are large video clips while the rest are small audio clips. The overall file size range is 24-1418 KB with an average of 152 KB. The total dataset size is 20.4 GB. During the tests, individual media files

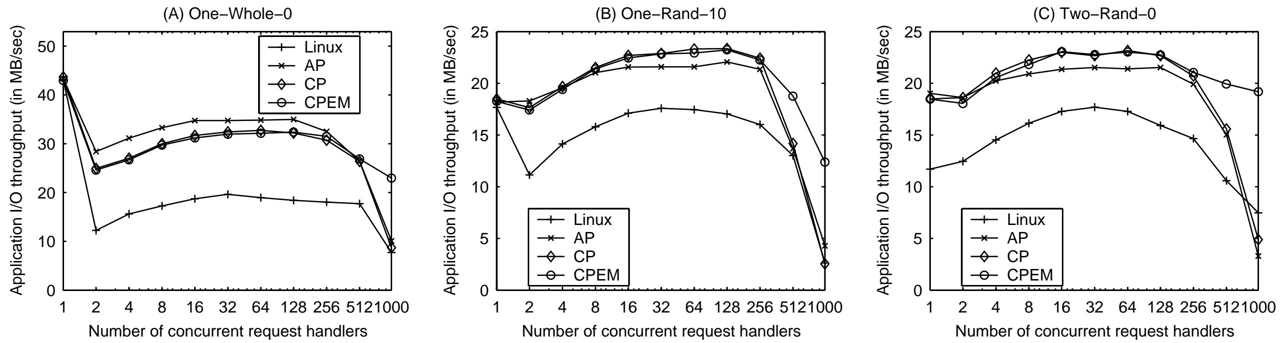

Figure 5: Microbenchmark performance at various concurrency levels. The OS disk scheduler does not use anticipatory scheduling.

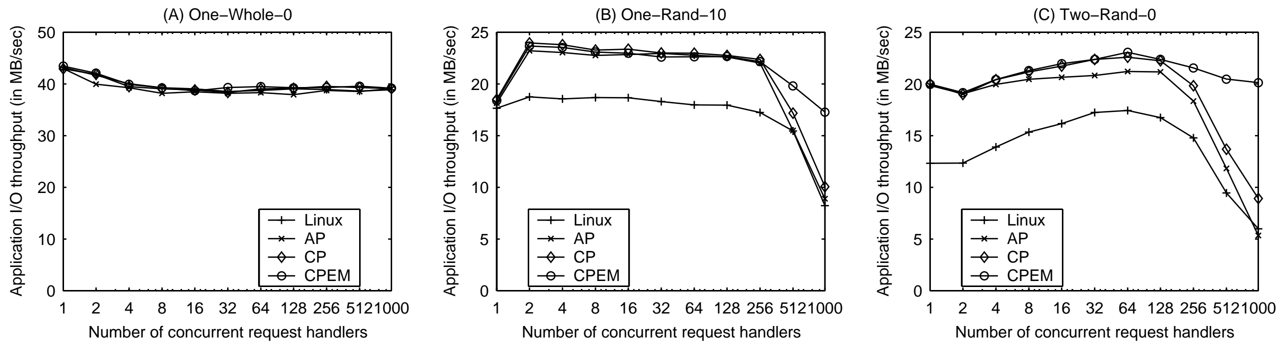

Figure 6: Microbenchmark performance at various concurrency levels. The OS disk scheduler employs anticipatory scheduling.

are chosen in the client requests according to a Zipf distribution.

- *BLAST genetic sequence database:* We ported the nucleotide and protein sequence similarity matching application *BLAST* [1] developed at the National Center for Biotechnology Information. Given an input sequence, BLAST searches hosted databases for similar sequences. In our BLAST server, we run each request handler as an OS process such that the application binaries can be used directly without much integration effort. Our tests involve around 10.9 GB nucleotide sequence data, based on a collection available at the GenBank [5]. We use 1000 synthetically generated queries in our tests.

Table 1 summarizes benchmark statistics that might affect the performance of the proposed strategies. The column "Whole-file access?" indicates potential waste on prefetching unnecessary data. No such waste exists if whole files are accessed by application request handlers.

5.2 Microbenchmark Performance

We assess the effectiveness of the proposed techniques by comparing the server performance under the following different kernel versions:

#1. *Linux:* The original Linux 2.6.3 kernel with a maximum prefetching depth of 32 pages. In fact the kernel we used in the experimentation is slightly different from the exact original. We discovered several performance anomalies in the original kernel (described in Section 6) for supporting data-intensive online servers. Our experimentation used a modified kernel that had these performance problems corrected.

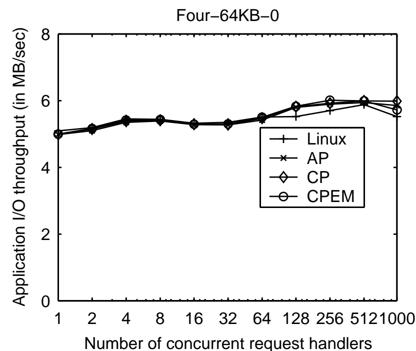

Figure 7: Performance of a random-access microbenchmark.

#2. *AP:* The original kernel with an aggressive prefetching policy. A maximum prefetching depth of 256 pages (about twice that of competitive prefetching) is used. This is a hypothetical approach included purely for the purpose of comparison.

#3. *CP:* Our proposed competitive prefetching strategy described in Section 3.

#4. *CPEM:* The competitive prefetching policy (approach #3) with the enhanced memory management system for prefetched pages, described in Section 4.

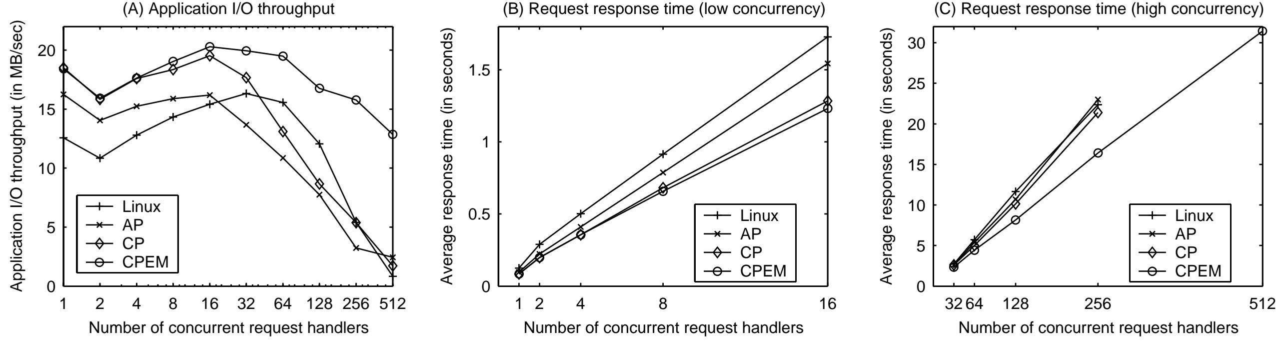

Figure 8: Performance (I/O throughput and request response time) of the index searching server at various concurrency levels.

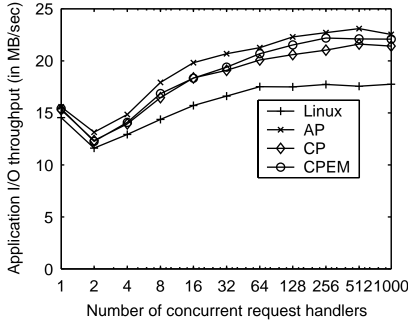

Figure 9: I/O throughput of Apache hosting media clips.

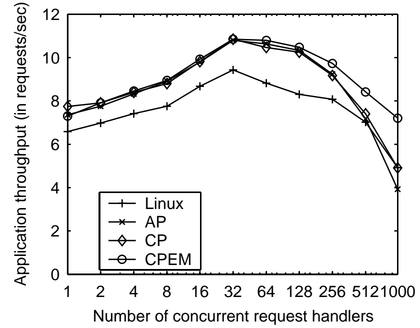

Figure 10: Application request throughput of the BLAST genetic sequence database.

Figure 5 shows the application-observed I/O throughput for the first three microbenchmarks that possess a significant amount of sequential data accesses. The results were produced without the use of anticipatory scheduling in the OS disk scheduler. We observe that our proposed techniques are very effective in improving the performance of all three microbenchmarks. At low concurrency levels, competitive prefetching provides up to two-fold performance improvement compared with the original kernel. At high concurrency levels, the *CPEM* policy achieves up to a four-fold performance improvement when compared to the original kernel. Aggressive prefetching provides very limited additional benefit (up to 14%) when application request handlers access whole files. This benefit is only available at low concurrency levels. Aggressive prefetching may even hurt performance as shown in Figures 5(B) and 5(C) because it wastes too much I/O bandwidth on fetching unnecessary data. The wasted I/O bandwidth more than compensates the advantage of less frequent I/O switching. We also note that the performance of *CPEM* still drops at high concurrency levels. This is due to constrained prefetching to avoid page thrashing (described in Section 4.2).

Figure 6 illustrates the performance of the same three microbenchmarks when the OS disk scheduler employs anticipatory scheduling. Figure 6(A) shows dramatically improved concurrent performance for the first microbenchmark (*One-Whole-0*). The improvement is an artifact of the reduction of I/O switching thanks to anticipatory scheduling. However, Figures 6(B) and 6(C) demonstrate that anticipatory scheduling is not very effective when significant think times are present in the request processing or when a request handler accesses multiple streams simultaneously. In such cases, our proposed strategies can significantly improve the application performance. The rest of this section only shows performance results when anticipatory scheduling is enabled since it always performs no worse than the kernel without it for all our workloads.

Figure 7 shows the performance of the random-access microbenchmark. We observe that all kernels perform similarly at all concurrency levels. Aggressive prefetching strategies do not suffer from fetching too much unnecessary data, because the slow-start phase used in all kernels quickly identifies the non-sequential access pattern and avoids large prefetch operations.

Competitiveness assessment. We estimate the optimal I/O throughput to assess the competitiveness of *CP*. We use a simple estimation $\frac{S_{tot}}{S_{tot}/R_{tr}+C_{switch}}$, where S_{tot} is the mean size of the benchmark sequential access streams, R_{tr} is the mean disk sequential transfer rate, and C_{switch} is the mean I/O switching time between two independent random disk head locations. Our estimated optimal throughputs for the four microbenchmarks are 45.3 MB/sec, 40.5 MB/sec, 40.5 MB/sec, and 5.5 MB/sec respectively. At low concurrency levels (1-32), the performance of *CP* is at least around half the optimal performance for all benchmarks, which confirms its competitiveness.

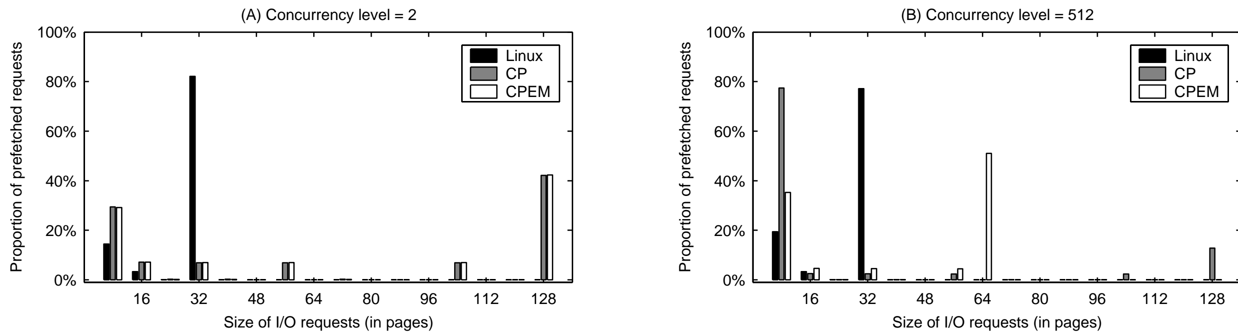

Figure 11: I/O request size histogram (8 page sizes per bucket) for index searching.

5.3 Performance of Real Applications

Figure 8(A) shows the I/O throughput of the index searching server at various concurrency levels. The results suggest that competitive prefetching can improve I/O throughput by around 47% at low concurrency when compared to the original kernel. In comparison, aggressive prefetching provides much less improvement (up to 29%) due to wasted I/O bandwidth on fetching unnecessary data. Even this improvement quickly diminishes as the concurrency level climbs up because of higher memory contention caused by aggressive prefetching. At high concurrency levels, the *CPEM* policy achieves an over four-fold performance improvement over all other kernels. Figures 8(B) and 8(C) show the request response time for the index searching server. For policies other than *CPEM*, we only show the request response time up to the concurrency level of 256 because their throughput is too low at higher concurrency levels to make the response time measurement meaningful. Overall, our proposed techniques can reduce the average request response time by 28% when compared to the original kernel.

Figure 9 shows the performance of the Apache Web server hosting media clips. The improvement is 6% and 24% at concurrency levels of 2 and 1000 respectively. Because each Web server request handler follows a strict sequential data access pattern on a single file, anticipatory disk scheduling in the original kernel helps to achieve very good performance. The I/O throughputs increase as the concurrency level climbs up. As we mentioned in section 2.3, this is due to at least two factors: 1) more overlapping of I/O and computation at higher concurrency levels; and 2) lower average seek distance when the disk scheduler can choose from more concurrent requests for seek reduction. The performance improvement due to competitive and aggressive prefetching becomes more evident at higher concurrency levels because the anticipation is more likely to be disrupted in these situations. We also observe that the server performance does not degrade significantly at high concurrency levels even without our enhanced prefetching memory management (*CPEM*). This is because the Apache Web server has a relatively small memory footprint.

Figure 10 shows the application request throughput of the BLAST database. Compared to the original kernel, the *CP* policy can improve the server throughput by around 18% at low concurrency while *CPEM* achieves 47% performance improvement at high concurrency.

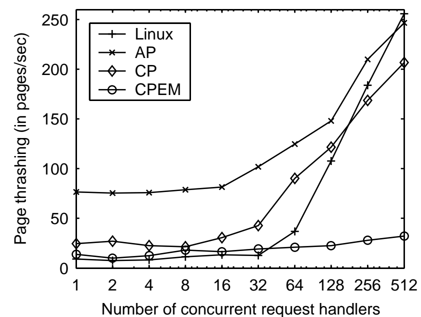

Figure 12: Prefetching-incurred page thrashing for index searching.

5.4 Per-resource Profiling

In order to better understand the performance results, we examine the server behavior in terms of the I/O request size, memory page thrashing, and CPU usage. We use a modified Linux Trace Toolkit (LTT) [35] for our experiments. LTT instruments the OS kernel with trace points that capture events of interest and log them using a user-level logging daemon. We modified LTT by adding trace points at kernel events that we are interested in.

Figure 11 shows the I/O request size distribution for the index searching application. The average I/O request size is an indicator of the I/O switching frequency, and thus the overall I/O efficiency. Figure 11(A) illustrates the I/O request size histogram at the low concurrency level of 2. Each bucket in the histogram contains 8 page sizes. The original Linux 2.6.3 has a maximum prefetching depth of 32 pages. Therefore, it produces much more frequent I/O switching than the competitive prefetching policy (*CP*). At high concurrency levels, shown in Figure 11(B), the two competitive prefetching approaches behave quite differently. First, *CPEM* has fewer small I/O requests (1-8 pages) because it leads to much less page thrashing. We also observe that it has a smaller maximum prefetching depth of 64 pages, due to constrained prefetching at high concurrency levels.

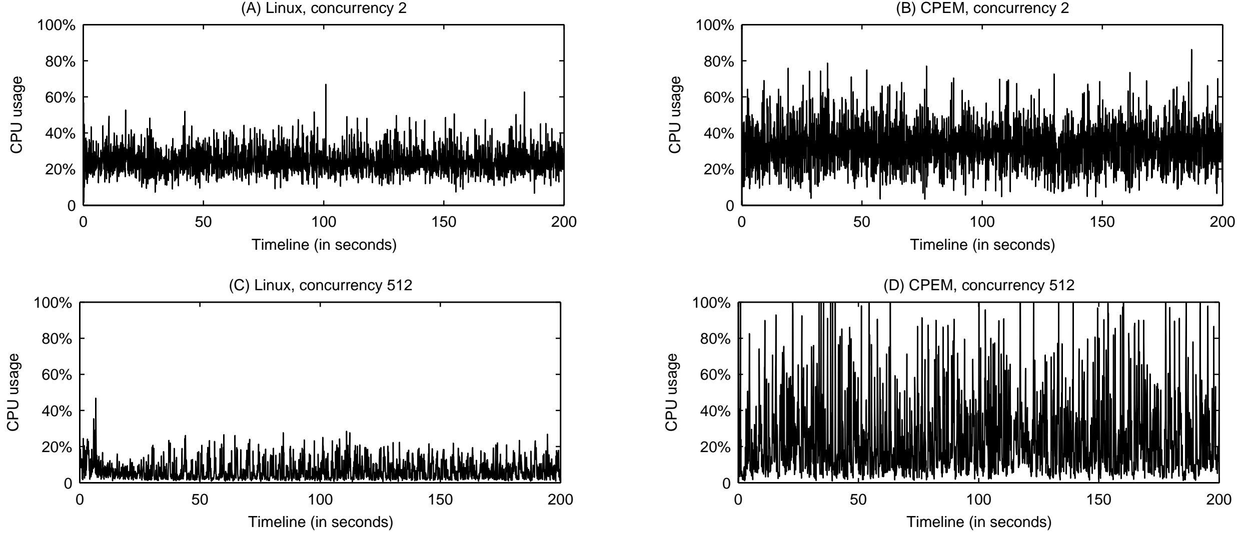

Figure 13: 200-second CPU usage snapshots (50ms running averages) for index searching.

Figure 12 shows the prefetching-incurred page thrashing rate for the index searching application. At low concurrency levels, all policies but *AP* exhibit a low page thrashing rate (below 25 pages/second). At high concurrency levels, all policies but *CPEM* suffer from significant page thrashing (207-256 pages/second at the concurrency of 512). In comparison, our proposed prefetching memory management (*CPEM*) maintains a low page thrashing rate (no more than 32 pages/second) at all measured concurrency levels.

Figure 13 illustrates 200-second CPU usage snapshots for the index searching benchmark under the original Linux 2.6.3 kernel and our proposed kernel. Results for a low concurrency level (2) and a high concurrency level (512) are shown. We observe that the CPU usage is consistently under 50% across all settings but *CPEM* at high concurrency. Figure 13(D) shows that the *CPEM* CPU usage at high concurrency is quite bursty. This is because the kernel paging daemon is frequently triggered to reclaim pages, an indication that the server free memory is constantly below the reclamation threshold. We have not discovered any performance problem associated with this burstiness and we plan to look into this further in the future. The CPU usage for the original Linux 2.6.3 kernel at the high concurrency level is very low. The substantial CPU idleness is caused by very inefficient I/O at this setting. Overall, the average CPU usage for the four test configurations are 19%, 28%, 6%, and 24% respectively.

5.5 Impact of Factors

In this section, we explore the performance impact of the server memory size, the application dataset size, and the average size of sequential access streams. We use the second microbenchmark (*One-Rand-10*) because of its flexibility in adjusting the workload parameters. When we evaluate the impact of one factor, we set the other factors to

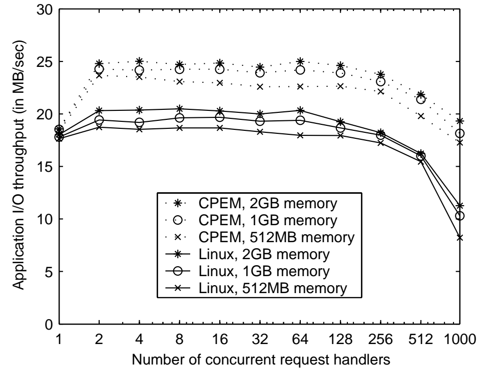

Figure 14: Performance impact of the server memory size for the microbenchmark *One-Rand-10*.

default values: 512 MB for the server memory size, 24 GB for the application dataset size, and 2 MB for the average size of sequential access streams. Note that the average size of sequential access streams is half of the file size because each request handler in this microbenchmark reads a random amount (evenly distributed between 64 KB and the total file size).

Figure 14 shows the performance of the microbenchmark at different server memory sizes: 2 GB, 1 GB, and 512 MB. Intuitively a larger memory size can improve the server throughput. However, this does not affect the performance advantage of our proposed techniques against the original Linux kernel.

Figure 15 illustrates the performance of the microbenchmark at different workload sizes: 0.4 GB, 2 GB, 8 GB, and 24 GB. Intuitively better performance is correlated with smaller workload sizes. Note that the 0.4 GB workload can completely fit into the server memory, therefore it has much better performance than others. Figure 15(B) shows that our proposed techniques do not negatively affect the server

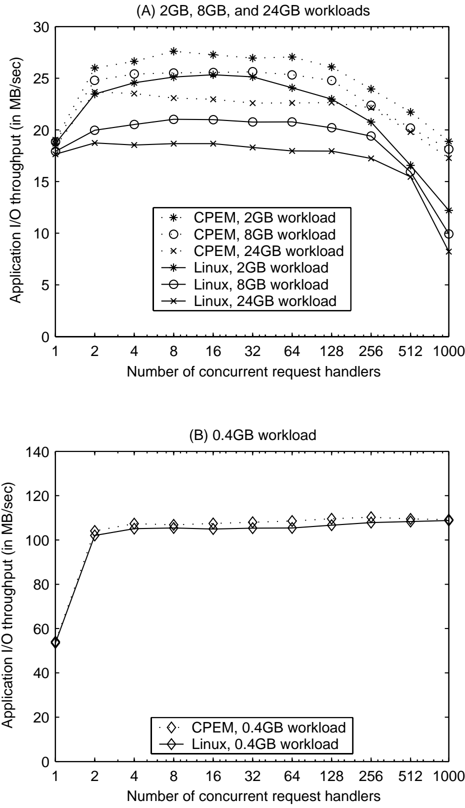

Figure 15: Performance impact of the workload size for the microbenchmark *One-Rand-10*.

performance when the application data can completely fit into the server memory. We also observe that the performance during concurrent execution is better than that of the serial execution for the 0.4 GB workload. This is because this workload is CPU-bound and concurrent execution allows better utilization of the two processors in the server.

Figure 16 shows the performance of the microbenchmark with different average sequential access stream sizes: 4 MB, 2 MB, and 1 MB. Various sequential access stream sizes are achieved by adjusting the file size in the dataset. We observe that our proposed techniques perform consistently better than the original Linux 2.6.3 at all configurations. We also observe that the improvement of competitive prefetching (*CP*) is more substantial with longer sequential access streams (around 26% for 4 MB stream and 9% for 1 MB stream at the concurrency of 2). This is because the small initial prefetching depths, which reduce the effectiveness of competitive prefetching, constitute a more significant portion for shorter sequential access streams. In comparison, the improvement achieved by *CPEM* is substantial (over 57%) for all settings at high concurrency.

5.6 Summary of Results

- At low concurrency levels, competitive prefetching can improve the server throughput by up to 62% for microbenchmarks and by up to 47% for real applications.

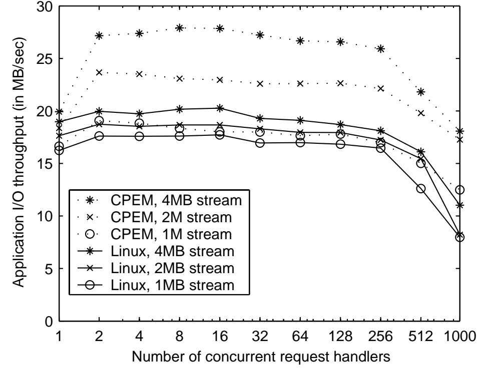

Figure 16: Performance impact of the sequential access stream size for the microbenchmark *One-Rand-10*.

The improvement over the original kernel is small when request handlers access disk data in a strictly sequential fashion (without any interleaving I/O) because anticipatory scheduling in the original kernel already helps to achieve very good performance for these workloads. Our results on microbenchmarks also show that the performance of competitive prefetching is at least half of the optimal offline performance.

- Aggressive prefetching provides limited additional improvement (up to 14%) when application request handlers access whole files. However, it may hurt the server performance when request handlers access only portions of files and it also leads to increased memory contention at high concurrency levels.
- The proposed memory management techniques can effectively control prefetching-incurred page thrashing at high concurrency levels. As a result, they can improve the server throughput by up to four times for both microbenchmarks and real applications.
- Competitive prefetching provides more performance improvements for applications with longer sequential data access streams. The improvement is relatively small for streams of 1 MB or smaller.
- The proposed techniques do not degrade the performance of applications with small data sizes (that can completely fit into memory) or those that access disk-resident data in a random fashion.

6 Discussion

Impact of concurrency control. The concurrency level of an online server can be controlled by employing a fixed-size process/thread pool and a request waiting queue. Additionally, admission control can be used to control the server load. Such techniques do not eliminate the benefit of competitive prefetching, which is shown to produce significant performance improvement even at low concurrency levels. On the other hand, controlling the execution concurrency may mitigate the memory contention due to aggressive prefetching. Keeping the concurrency level too

low, however, may reduce the server resource utilization efficiency and lower the server responsiveness.

Buy more memory to solve the problem? Larger memory sizes can improve the throughput of data-intensive online servers by allowing a larger portion of the dataset to remain in the main memory. However, many applications require a significant amount of additional memory to achieve noticeable performance improvements. For example, the index searching application used in the evaluation would not acquire significant performance enhancement unless a substantial fraction of the total dataset is cached. Provisioning enough memory such that the complete dataset can fit into memory would solve the problem. Although this practice is possible for the most critical data partitions or server components at online service sites, many other parts of the system still have to function with limited amount of memory for economical reasons [36]. Consequently, it is desirable to provide efficient support for highly concurrent online servers whose dataset size far exceeds the available server memory.

Impact on other workloads. Our design takes measures to minimize negative impact on applications not directly targeted in this work. However, such impact may still exist. In particular, increased prefetching depths can lead to reduced interactive responsiveness from the I/O subsystem. Techniques such as priority-based disk queues [12] and semi-preemptible I/O [10] can be employed to alleviate this problem. Additional investigation is needed to address the integration of such techniques. Further to our relief, it is often the case that an online server (*e.g.*, those hosting Internet services) is dedicated for supporting a single workload. Therefore, efficient OS support for other types of workloads may be unnecessary on those servers.

Impact of multi-disk systems. Our current work focuses on single-disk storage devices. We believe our competitive prefetching strategy can be similarly employed to guide prefetching depths for multi-disk systems, such as disk arrays (RAID). These systems often allow simultaneous transfers out of multiple disks and thus offer much higher aggregate throughput. However, seek and rotational delays are inherently limited by individual disks. Consequently, multi-disk systems often require larger competitive prefetching depths.

Performance anomalies in the original Linux kernel. We have discovered several performance anomalies in the original Linux 2.6.3 when supporting data-intensive online servers. One such problem appears in the implementation of anticipatory disk scheduling. Very large I/O requests from the file system are usually split into smaller pieces up to a certain maximum size (*e.g.*, 320 disk sectors) before being forwarded to the disk drive. The completion of each one of these pieces will trigger an I/O interrupt. The original anticipatory scheduler would start the anticipation timer right after the first such interrupt, which often causes premature timeout. We corrected the problem by starting the anticipation timer only after all pieces of a file system I/O request have completed.

An additional problem was discovered in the prefetching algorithm of Linux 2.6.3. The kernel checks for disk congestion when each prefetch operation is initiated. If the number of pending requests in the disk driver queue exceeds a certain threshold (113 in Linux 2.6.3)[4], the prefetch operation is canceled. However, the read-ahead window descriptors are not updated to reflect the cancellation of the prefetch request, leading to an interruption of prefetching and performance degradation. All experimental results in the paper are based on corrected kernels.

7 Related Work

The subject of highly-concurrent online servers has been investigated by several researchers. Pai *et al.* showed that the HTTP server workload with moderate disk I/O activities can be efficiently managed by incorporating asynchronous I/O or helper threads into an event-driven HTTP server [26]. A later study by Welsh *et al.* further improved the performance by balancing the load of multiple event-driven stages in an HTTP request handler [34]. The more recent Capriccio work provides a user-level thread package that can scale to support hundreds of thousands of threads [33]. However, these studies mainly focus on the CPU utilization efficiency while the throughput of online servers is bounded by the I/O performance when the application data size far exceeds the available server memory.

Previous work has explored modeling data access patterns and supporting efficient memory buffer page replacement for data-intensive applications and database systems, including LRU-K [25], 2Q [17], Unified Buffer Management [20], Multi-Queue [38], and LIRS [16]. However, these studies do not explicitly address memory buffer management at high execution concurrency, as seen in popular online services. Data-intensive online servers may produce significant prefetching-incurred page thrashing at high concurrency, which can severely degrade the server performance. We reduce such page thrashing by managing prefetched pages separately from the rest of the memory cache and by using a novel second-chance reclamation algorithm for these pages.

Previous work has suggested the use of application hints to increase prefetching accuracy for workloads consisting of both single [29] and multiple [32] applications. Cao *et al.* proposed a two-level page replacement scheme that allows applications to control their own cache replacement while the kernel controls the allocation of cache space among processes [6, 7]. Self-paging is used in the Nemesis operating system to provide QoS support for multimedia applications [13]. Recent studies also examined how heuristics-based predication of application data access pattern can improve the performance of file prefetching [22, 37]. Our

[4]The threshold is calculated as: $\frac{7}{8}Q + 1$, where Q is the maximum disk driver queue size (default to 128).

proposed strategy does not require application cooperation. However, the idea of prefetching with application-supplied information or predication can be incorporated into our work. For instance, application information can be used to reduce the overhead of prefetching unnecessary data.

Several researchers have addressed the problem of memory allocation between prefetched and cached pages. Cao *et al.* devised memory allocation algorithms to reduce application running time when facing memory contention between aggressive prefetching and page caching [6]. Kimbrel *et al.* further investigated this problem for multi-disk systems [21]. Patterson *et al.* [29] provide a cost-benefit model for evaluating the benefit of prefetching and hint a limited prefetching depth up to a process's *prefetch horizon*. Their cost-benefit model also focuses on reducing the program completion time. These techniques may not be effective in optimizing the throughput of data-intensive online servers, which mainly depends on the I/O efficiency and the rate of disk I/O switching in particular. Our memory management strategy tends to lower the number of memory misses, which can in turn lower disk I/O switching rate when there is severe memory contention. Kaplan *et al.* further explored cost-benefit model and they use histograms in dynamically controlling the prefetch memory allocation with the goal of minimizing the number of page faults [18]. Our strategy also aims to minimize the number of page faults. But instead of using a cost-benefit model, we treat this problem as linearly constrained nonlinear optimization. And the overhead of their implementation using histogram is much more expensive compared with our light-weight gradient descent solution.

Aggressive prefetching was also proposed to increase disk access burstiness and thus to save energy for mobile devices [27, 28]. Fraser and Chang suggested that speculative I/O prefetching can reduce the running time for explicit I/O and swapping applications [11]. These studies do not address the performance problem of frequent I/O switching exhibited in highly concurrent online servers.

Application-specific techniques have been proposed to support out-of-core execution of data-intensive applications. For instance, Chiang *et al.* employed I/O-efficient data structures to support interactive disk-resident isosurface extraction [9]. Compiler techniques have also been investigated to improve the performance of out-of-core applications. Mowry *et al.* proposed automatic compiler-inserted I/O prefetching techniques that resulted in significant performance improvements for the tested applications [24]. Our work provides system-level support that can transparently benefit data-intensive online applications.

More recently, Carrera and Bianchini studied disk controller cache management and proposed two firmware-level techniques to improve the disk throughput for data-intensive servers [8]. Our work shares their objective while focusing on the 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 [2]. Our work provides transparent operating system level support for much wider scope of data-intensive workloads.

8 Conclusion

This paper presents the design and implementation of a competitive I/O prefetching strategy and an enhanced memory management system supporting data-intensive online servers. Based on a simple model, it is shown that the performance of competitive prefetching (in terms of I/O throughput) is at least half that of the optimal offline policy. Our enhanced memory management includes the addition of a prefetch cache dedicated to prefetched memory pages and a novel second-chance page reclamation algorithm for it. Our approach is design to reduce prefetching-incurred page thrashing at high concurrency.

We implemented the proposed techniques in the Linux 2.6.3 kernel and conducted experiments using microbenchmarks and three real applications: an index searching server, the Apache Web server hosting media clips, and a genetic sequence database. Overall, our evaluation demonstrates that competitive prefetching can improve the throughput of real applications by up to 47% at low concurrency while the enhanced memory management scheme can produce up to a four-fold performance enhancement at high concurrency.

Acknowledgment: We would like to thank Lingkun Chu, Tao Yang, and Apostolos Gerasoulis at Ask Jeeves Inc. for providing us the index searching server and traces used in the experimentation. We would like to thank Peter DeRosa for porting the BLAST application in our experimentation. We would also like to thank Michael L. Scott and others in the URCS systems group for their valuable comments. Last but not least, we are indebted to Liudvikas Bukys for setting up the machines used in this study.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] /S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Circus: Opportunistic Block Reordering for Scalable Content Servers. In *Proc. of the 3rd USENIX Conference on File And Storage Technologies*, San Fancisco, CA, March 2004.

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

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

[5] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. GenBank. *Nucleic Acids Research*, 30(1):17–20, 2002.

[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 1st USENIX OSDI*, Monterey, CA, November 1994.

[8] E. Carrera and R. Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proc. of the 10th International Symp. on High Performance Computer Architecture*, Madrid,Spain, February 2004.

[9] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive Out-of-core Isosurface Extraction. In *Proc. of the Conference on Visualization*, pages 167–174, Research Triangle Park, NC, October 1998.

[10] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Design and Implementation of Semi-preemptible IO. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies*, pages 145–158, San Francisco, CA, March 2003.

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

[12] 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.

[13] S. M. Hand. Self-paging in the Nemesis Operating System. In *Proc. of the 3rd USENIX OSDI*, New Orleans, LA, February 1999.

[14] T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, 1988.

[15] 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 SOSP*, pages 117 – 130, Banff, Canada, October 2001.

[16] S. Jiang and X. Zhang. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proc. of the ACM SIGMETRICS*, pages 31–42, Marina Del Rey, CA, June 2002.

[17] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th VLDB Conference*, pages 439–450, Santiago, Chile, September 1994.

[18] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive Caching for Demand Prepaging. In *Proc. of the 3rd International Symp. on Memory Management*, pages 114–126, Berlin, Germany, June 2002.

[19] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proc. of the 13th ACM SOSP*, pages 41–55, Pacific Grove, CA, October 1991.

[20] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. of the 4th USENIX OSDI*, San Diego, CA, October 2000.

[21] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proc. of the 2nd USENIX OSDI*, Seattle, WA, October 1996.

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

[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 OSDI*, San Diego, CA, October 2000.

[24] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proc. of the 2nd USENIX OSDI*, Seattle, WA, October 1996.

[25] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proc. of the ACM SIGMOD*, pages 297–306, Washington, DC, May 1993.

[26] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[27] A. E. Papathanasiou and M. L. Scott. Energy Efficiency through Burstiness. In *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, October 2003.

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

[29] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM SOSP*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[30] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[31] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[32] 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.

[33] 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 SOSP*, pages 268–281, Bolton Landing, NY, October 2003.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM SOSP*, pages 230–243, Banff, Canada, October 2001.

[35] K. Yaghmour and M. R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proc. of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[36] T. Yang. Ask Jeeves, Inc., 2003. Personal communication.

[37] T. Yeh, D. Long, and S. A. Brandt. Using Program and User Information to Improve File Prediction Performance. In *Proc. of the Intl. Symposium on Performance Analysis of Systems and Software*, pages 111–119, Tucson, AZ, November 2001.

[38] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, June 2001.