Competitive Prefetching for Data-Intensive Online Servers*

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

Athanasios E. Papathanasiou
papathan@cs.rochester.edu
Dept. of Computer Science
University of Rochester

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

Abstract

In a disk I/O-intensive online server, sequential data accesses of one application instance can be frequently interrupted by other concurrent processes. Although aggressive I/O prefetching can improve the granularity of sequential data access, it must control the I/O bandwidth wasted on prefetching unneeded data. In this paper, we propose a *competitive* prefetching strategy that balances the overhead of disk I/O switching and that of unnecessary prefetching. Based on a simple model, we show that the performance of our strategy (in terms of I/O throughput) is at least half that of the optimal offline policy. We have implemented competitive prefetching in the Linux 2.6.3 kernel and conducted experiments based on microbenchmarks and two real applications (an index searching server and the Apache Web server). Our evaluation results demonstrate that competitive prefetching can improve the throughput of real applications by 15%–47%. The improvement is achieved without any application assistance or changes.

1 Introduction

Rapidly emerging Internet services allow interactive accesses from a large number of concurrent users. Earlier studies have identified a number of issues associated with server concurrency management, including TLB misses, scheduling overhead, and lock contention. Techniques, such as event-driven concurrency management [14, 19] and user-level threads [18], 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). In comparison, system 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 [3] that support interactive search on terabytes of indexed Web pages and the bioinformatics database GenBank [4] that allows online queries on over 100 GB genetic and protein sequence data.

For data-intensive online servers, the I/O efficiency typically dominates the overall system throughput when the dataset size far exceeds the available server memory. During concurrent execution, sequential 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 [10] 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. 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.

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 unneeded 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. This paper investigates I/O prefetching technique that maintains the balance between the overhead of I/O switching and that of unnecessary prefetching.

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

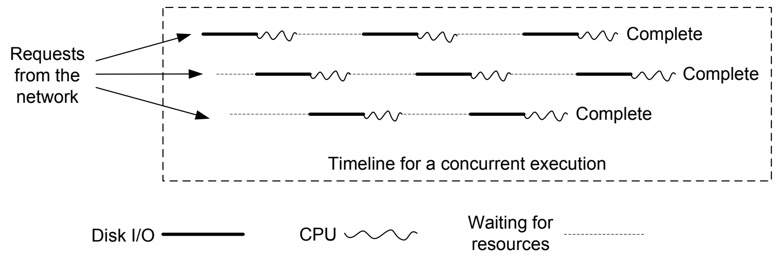

Figure 1: Concurrent application execution in a data-intensive online server.

The rest of the paper is organized as follows. Sections 1.1 and 1.2 describe the related work and the characteristics of targeted data-intensive online servers. Section 2 presents the design of our proposed competitive prefetching technique and its implementation in the Linux 2.6.3 kernel. Section 3 provides the performance results based on microbenchmarks and real applications. Section 4 discusses several remaining issues and Section 5 concludes the paper.

1.1 Related Work

I/O prefetching has long been studied by researchers. Cao *et al.* first explored application-controlled prefetching with complete knowledge of I/O access pattern [5]. Patterson *et al.* proposed a cost-benefit model for evaluating the benefit of prefetching based on application hints [16]. Their results suggested a limited prefetching depth up to a process' *prefetch horizon* under the assumption of no disk congestion. In order to maximize the applicability, our work in this paper focuses on transparent OS support that can improve the server throughput without any application assistance or changes.

Previous studies have examined ways to acquire or predict I/O access pattern information without direct application involvement, including modeling and analysis of interesting system events [13, 20]. Aggressive prefetching was also proposed to increase disk access burstiness and thus to save energy for mobile devices [15]. Fraser and Chang suggested that speculative I/O prefetching can reduce the running time for explicit I/O and swapping applications [8]. The above studies make no specific attempt to increase the granularity of sequential I/O accesses or to reduce the I/O switching frequency in a concurrent on-demand server.

The recent work by Carrera and Bianchini examined disk controller cache management and proposed two firmware-level techniques to improve the disk throughput for data-intensive servers [6]. 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 [1]. Our work provides transparent operating system level support for a wider scope of data-intensive workloads.

1.2 Targeted Application Characteristics

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.

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 FTP and Web servers, satisfy our assumptions. Many more applications are usually carefully constructed to follow mostly sequential access patterns in order to achieve efficient disk I/O.

2 Competitive Prefetching

Under concurrent execution, I/O throughput mostly depends on the granularity of I/O switching, defined as the average amount of sequentially accessed data between consecutive I/O switches. Although an I/O prefetching strategy can control such granularity, it must address the following tradeoff in deciding the I/O prefetching depth: a conservative prefetching strategy may *produce high I/O switching overhead* while overly aggressive prefetching may *waste too much I/O bandwidth on fetching unneeded data*.

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:

$$\mathcal{C}_{min} = \frac{S_{tot}}{R_{tr}} + C_{switch} \tag{1}$$

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[1]. The total I/O switching cost for request processing can be derived as:

$$\begin{aligned} \mathcal{C}_{tot\_switch} &= \lceil \frac{S_{tot}}{S_p} \rceil \cdot C_{switch} \\ &\leq (\frac{S_{tot}}{S_p} + 1) \cdot C_{switch} \end{aligned} \tag{2}$$

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

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

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

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

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

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

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

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 [17] and multiprocessor synchronization [12] 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, our enhanced 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. 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).

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 2 shows the measured sequential read throughput and

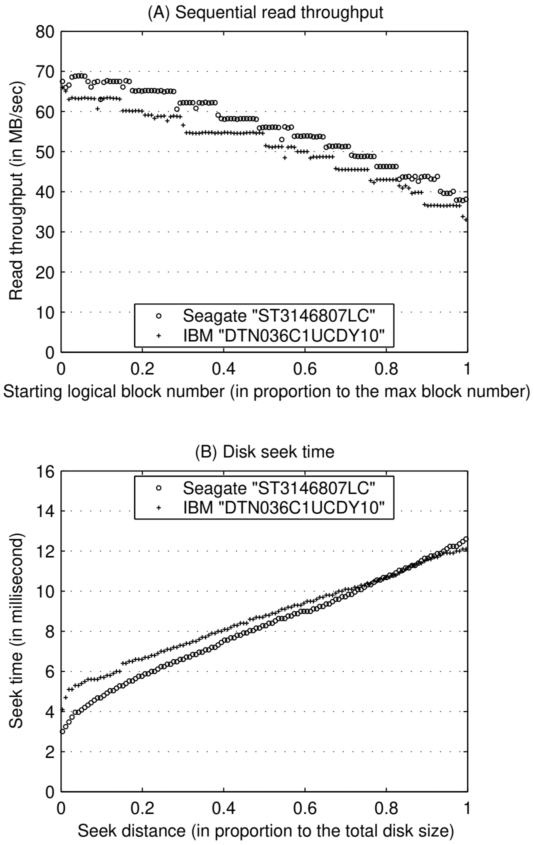

Figure 2: Sequential read throughput and seek time for two drives.

the seek time for these 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, the average seek and rotational delays justify transferring around 540 KB of data. In comparison, the default maximum prefetching depth in Linux 2.6.3 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.

3 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 by Gigabit Myrinet. Each node is equipped with two 2 GHz Xeon processors, 2 GB memory, and a 36.4 GB IBM 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).

3.1 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. Each thread reads disk files in certain pattern and access private memory of 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, or a 64 KB chunk), and a think time delay during the processing of each request (0 ms or 10 ms). We use 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 think time delay. 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

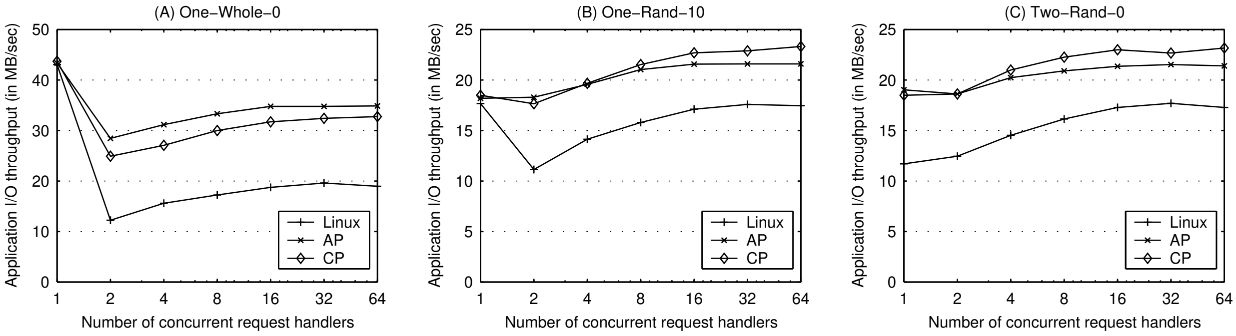

Figure 3: Microbenchmark performance at various concurrency levels. The OS disk scheduler does not use anticipatory scheduling.

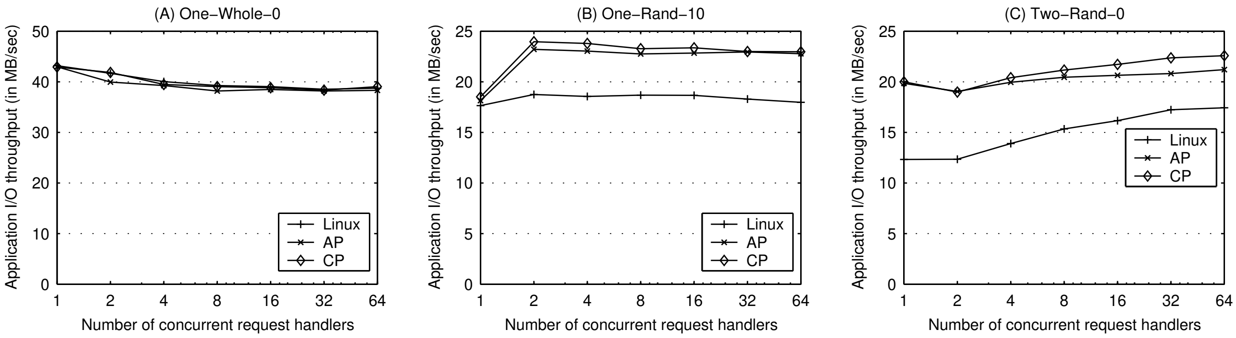

Figure 4: Microbenchmark performance at various concurrency levels. The OS disk scheduler employs anticipatory scheduling.

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 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 also include two 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 [3]. 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 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.

3.2 Microbenchmark Performance

We assess the effectiveness of the proposed technique 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.

#2. *AP:* Aggressive prefetching with a maximum prefetching depth of 256 pages (about twice that of competitive prefetching). This is a hypothetical approach included purely for the purpose of comparison.

#3. *CP:* Our proposed competitive prefetching strategy described in Section 2.

Figure 3 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. Competitive prefetching provides up to two-fold performance improvement compared with the original kernel. Aggressive prefetching provides very limited additional benefit (up to 14%) when application request handlers access whole files. Aggressive prefetching may even hurt performance as shown in Figures 3(B) and 3(C) because it wastes too much I/O bandwidth on fetching unneeded data. The wasted I/O bandwidth more than compensates the advantage of less frequent I/O switching.

Figure 4 illustrates the performance of the same three microbenchmarks when the OS disk scheduler employs anticipatory scheduling. Figure 4(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 4(B) and 4(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.

Figure 5 shows the performance of the random-access microbenchmark. We observe that all kernels perform

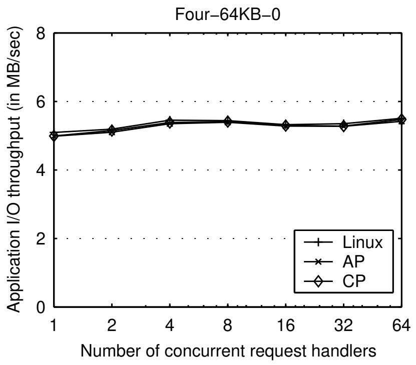

Figure 5: Performance of a random-access microbenchmark.

similarly. More 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 competitive prefetching. 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 four microbenchmarks are 45.3 MB/sec, 40.5 MB/sec, 40.5 MB/sec, and 5.5 MB/sec respectively. Our results show that the performance of competitive prefetching is almost always at least half the optimal performance for all benchmarks, which confirms its competitiveness. The only exception is at concurrency one for One-Rand-10. In this case, the long think time makes the disk underloaded, which violates our assumption that the workload is disk I/O-bound.

3.3 Performance of Real Applications

In this section, we only show performance results when anticipatory scheduling is enabled since it generally performs no worse than the kernel without it for all our workloads. Figure 6 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% 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 unneeded data. Even

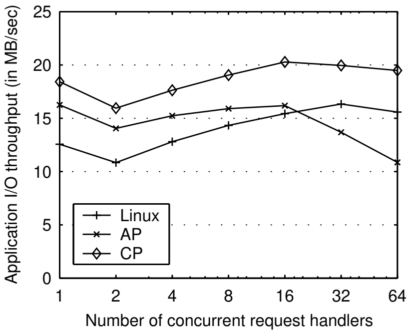

Figure 6: I/O throughput of the index searching server.

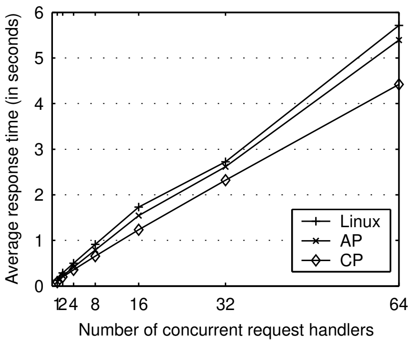

Figure 7: Request response time of the index searching server.

this improvement quickly diminishes as the concurrency level climbs up because of higher memory contention caused by aggressive prefetching. This issue will be discussed further in the next section. Figure 7 shows the request response time for the index searching server. Overall, our proposed techniques can reduce the average request response time by 28% when compared to the original kernel.

Figure 8 shows the performance of the Apache Web server hosting media clips. The improvement is 6% and 15% at concurrency levels of 2 and 64 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 quite good performance. The I/O throughputs increase as the concurrency level climbs up due to 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 disk I/O anticipation is more likely to be disrupted in these situations.

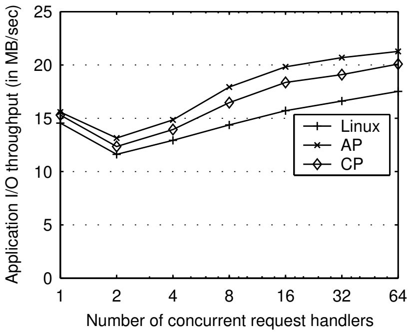

Figure 8: I/O throughput of Apache hosting media clips.

4 Discussion

Memory contention in high concurrency. I/O prefetching can consume a significant amount of server memory, especially for highly concurrent online workloads. The memory contention in high concurrency may lead to significant performance degradation for at least two reasons: 1) the eviction of prefetched pages before they are accessed; 2) increased cache miss rate due to memory used for prefetching. The competitiveness of our prefetching strategy would not hold at the presence of these problems.

Memory contention between caching and prefetching has been examined by Cao *et al.* [5] and Patterson *et al.* [16]. More recently, Kaplan *et al.* proposed a dynamic memory allocation strategy based on the maintenance of detailed cost and benefit statistics about candidate strategies [11]. Memory contention can also be avoided by limiting the server concurrency level with admission control and request queuing.

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 [9] and semi-preemptible I/O [7] 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 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 disk bandwidth. On the other hand, seek and rotational delays are inherently limited by individual disks. Consequently, multi-disk systems often require larger competitive prefetching depths.

Impact of non-sequential data layout. File system allocation algorithms can not guarantee all logically sequential data blocks are mapped to physically contiguous disk blocks. But in most cases, the file system attempts to organize them as contiguous as possible. Another issue that can disrupt sequential block allocation is bad sector remapping. However, we believe this does not occur frequent enough in practice to diminish the benefit of aggressive prefetching strategies such as competitive prefetching.

5 Conclusion

This paper presents the design and implementation of a competitive I/O prefetching strategy 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. We implemented the proposed technique in the Linux 2.6.3 kernel and conducted experiments using microbenchmarks and two real applications: an index searching server and the Apache Web server hosting media clips. Overall, our evaluation demonstrates that competitive prefetching can improve the throughput of real applications by 15%–47%.

Acknowledgment

We would like to thank Michael L. Scott and others in the URCS systems group for their valuable comments. We would also 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. Last but not least, we are indebted to Liudvikas Bukys for setting up the machines used in this study.

References

[1] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Circus: Opportunistic Block Reordering for Scalable Content Servers. In *Proc. of the 3rd USENIX Conf. on File and Storage Technologies*, pages 201–212, San Fancisco, CA, March 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] 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.

[5] 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.

[6] 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, February 2004.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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, November 2001.