Managing Prefetch Memory for Data-Intensive Online Servers*

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

Abstract

Data-intensive online servers may contain a significant amount of prefetched data in memory due to large-granularity I/O prefetching and high execution concurrency. Using a traditional access recency or frequency-based page reclamation policy, memory contention can cause a substantial number of prefetched pages to be prematurely evicted before being accessed. This paper presents a new memory management framework that handles prefetched (but not-yet-accessed) pages separately from the rest of the memory buffer cache. We examine three new heuristic policies when a victim page (among the prefetched pages) needs to be identified for eviction: 1) evict the last page of the longest prefetch stream; 2) evict the last page of the least recently accessed prefetch stream; and 3) evict the last page of the prefetch stream whose owner process has consumed the most amount of CPU since it last accessed the prefetch stream. These policies require no application changes or hints on their data access patterns.

We have implemented the proposed techniques in the Linux 2.6.10 kernel and conducted experiments based on microbenchmarks and two real application workloads (a trace-driven index searching server and the Apache Web server hosting media clips). Compared with access history-based policies, our memory management scheme can improve the server throughput of real workloads by 11–64% at high concurrency levels. Further, the proposed approach is 10–32% below an approximated optimal page reclamation policy that uses application-provided I/O access hints. The space overhead of our implementation is about 0.4% of the physical memory size.

1 Introduction

Emerging data-intensive online services access large disk-resident datasets while serving many clients simultaneously. Examples of such servers include Web-scale keyword search engines that support interactive search on terabytes of indexed Web pages and Web servers hosting large multimedia files. For data-intensive online servers, the disk I/O performance and memory utilization efficiency dominate 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. Due to the high storage device seeking overhead, large-granularity I/O prefetching is often employed to reduce the seek frequency and thus decrease its overhead. At high execution concurrency, there can be many memory pages containing prefetched but not-yet-accessed data, which we call *prefetched pages* in short.

The prefetched pages are traditionally managed together with the rest of the memory buffer cache. Existing memory management methods generally fall into two categories.

- Application-assisted techniques [6, 16, 22, 23, 29] achieve efficient memory utilization with application-supplied information or hints on their I/O access patterns. However, the reliance on such information affects the applicability of these approaches and their relatively slow adoption in production operating systems is a reflection of this problem. Our objective in this work is to provide *transparent* memory management that does not require any explicit application assistance.
- Existing transparent memory management methods typically use access history-based page reclamations, such as Working-Set [8], LRU/LFU [19], or CLOCK [27]. Because prefetched pages do not have any access history, an access recency or frequency-based memory management scheme tends to evict them earlier than pages that have been accessed before. Although MRU-like schemes may benefit prefetched pages, they perform poorly with general application workloads and in practice they are only used for workloads with exclusively sequential data access pattern. Among prefetched pages themselves, the eviction order based on traditional reclamation policies would be actually First-In-First-Out, again due to the lack of any access his-

*This work was supported in part by NSF grants CCR-0306473, ITR/IIS-0312925, and an NSF CAREER Award CCF-0448413.

tory. Under such management, memory contention at high execution concurrency may result in premature eviction of prefetched pages before they are accessed, which we call *page thrashing*. It could severely degrade the server performance.

One reason for the lack of specific attention on managing the prefetched (but not-yet-accessed) memory pages is that these pages only constitute a small portion of the memory in normal systems. However, their presence is substantial for data-intensive online servers at high execution concurrency. Further, it was recently argued that more aggressive prefetching should be supported in modern operating systems due to emerging application needs and the disk I/O energy efficiency [21]. Aggressive prefetching would also contribute to more substantial presence of prefetched memory pages.

In this paper, we propose a new memory management framework that handles prefetched pages separately from other pages in the memory system. Such a separation protects against excessive eviction of prefetched pages and allows a customized page reclamation policy for them. We explore heuristic page reclamation policies that can identify the prefetched pages least likely to be accessed in the near future. In addition to the page reclamation policy, the prefetch memory management must also address the memory allocation between caching and prefetching. We employ a gradient descent-based approach that dynamically adjusts the memory allocation for prefetched pages in order to minimize the overall page miss rate in the system.

A number of earlier studies have investigated I/O prefetching and its memory management in an integrated fashion [5, 14, 22, 23]. These approaches can adjust the prefetching strategy depending on the memory cache content and therefore achieve better memory utilization. Our work in this paper is exclusively focused on the prefetch memory management with a given prefetching strategy. Combining our results with adaptive prefetching may further improve the overall system performance but it is beyond the scope of this work.

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 presents the design of our proposed prefetch memory management techniques and its implementation in the Linux 2.6.10 kernel. Section 4 provides the performance results based on microbenchmarks and two real application workloads. Section 5 describes the related work and Section 6 concludes the paper.

2 Targeted Applications and Existing OS Support

Our work focuses on online servers supporting highly concurrent workloads that access a large amount of disk-resident data. We further assume that our targeted workloads involve mostly read-only I/O. In these servers, each incoming request is serviced by a request handler which can be a thread in a multi-threaded server or a series of event handlers in an event-driven server. 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 processing consumes both disk I/O and CPU resources, the overall server throughput is often dominated by the disk I/O performance and the memory utilization efficiency when the application data size far exceeds the available server memory. Figure 1 illustrates such an execution environment.

During concurrent execution, sequential data access of one request handler can be frequently interrupted by other active request handlers in the server. This may severely affect I/O efficiency due to long disk seek and rotational delays. Anticipatory disk I/O scheduling [11] 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 when the application exhibits some sequential I/O access patterns [26]. A larger prefetching depth results in less frequent I/O switching, and consequently yields fewer disk seeks per time unit. In practice, the default Linux and FreeBSD may prefetch up to 128 KB and 256 KB respectively in advance during sequential file accesses. We recently proposed a competitive prefetching strategy that can achieve at least half the optimal I/O throughput when there is no memory contention [17]. This strategy specifies that the prefetching depth should be equal to the amount of data that can be sequentially transfered within a single I/O switching period. The competitive prefetching depth is around 500 KB for several modern IBM and Seagate disks that were measured.

I/O prefetching can create significant memory de-

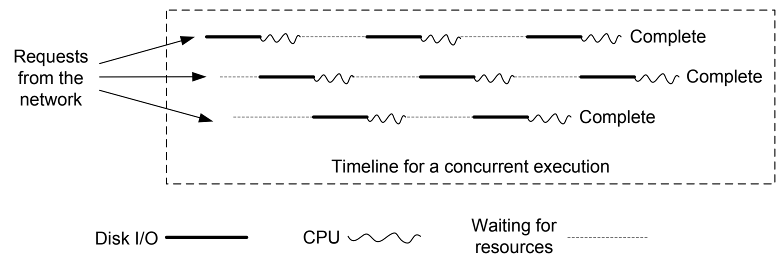

Figure 1: Concurrent application execution in a data-intensive online server.

mands. Such demands may result in severe contention at high concurrency levels, where many request handlers in the server are simultaneously competing for memory pages. Access recency or frequency-based replacement policies are misplaced for handling prefetched pages because they fail to give proper priority to pages that are prefetched but not yet accessed. We use the following two examples to illustrate the problems with access history-based replacement policies.

- *Priority between prefetched pages and accessed pages*: 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 the prefetching operation itself is not counted as an access, then p_2 will be evicted earlier under an access recency or frequency-based policy although it may be more useful in the near future. If the prefetching operation is counted as an access but t_1 occurs after t_2, then p_2 will still be evicted earlier than p_1.

- *Priority among prefetched pages*: Due to the lack of any access history, prefetched pages will follow a FIFO eviction order among themselves during memory contention. Assume page p_1 and p_2 are both prefetched but not yet accessed and p_1 is prefetched ahead of p_2. Then p_1 will always be evicted earlier under the FIFO order even if p_2's *owner request handler* (defined as the request handler that initiated p_2's prefetching) has completed its task and exited from the server, an strong indication that p_2 would not be accessed in the near future.

Under poor memory management, many prefetched pages may be evicted before their owner request handlers get the chance to access them. Prematurely evicted pages will have to be fetched again and we call such a phenomenon prefetch page thrashing. In addition to wasting I/O bandwidth on makeup fetches, page thrashing further reduces the I/O efficiency. This is because page thrashing destroys sequential access sequence crucial for large-granularity prefetching and consequently many of the makeup fetches are inefficient small-granularity I/O operations.

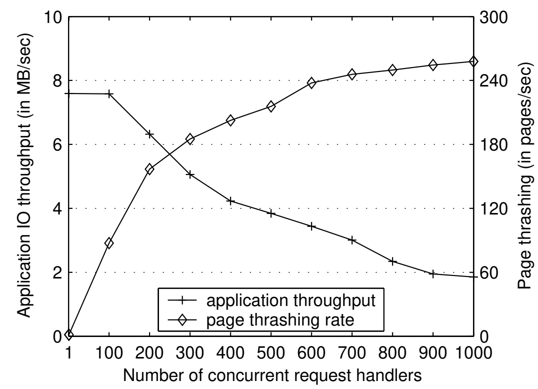

Figure 2: Application throughput and prefetch page thrashing rate of a trace-driven index searching server under the original Linux 2.6.10 kernel. A prefetch thrashing is counted when a prefetched page is evicted before the request handler that initiated the prefetching attempts to access it.

Measured performance We use the Linux kernel as an example to illustrate the performance of an access history-based memory management. Linux employs two LRU lists for memory management: 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. Frequently accessed pages are moved from the inactive list to the active 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.

Figure 2 shows the performance of a trace-driven index searching server under the Linux 2.6.10 kernel. This kernel is slightly modified to correct several performance anomalies during highly concurrent I/O [25]. The details about the benchmark and the experimental settings can be found in Section 4.1. We show the application throughput and page thrashing rate for up to 1000 concurrent request handlers in the server. At high execution concurrency, the I/O throughput drops significantly as a result of high page thrashing rate. Particularly at the concurrency level of 1000, the I/O throughput degrades to about 25% of its peak performance while the prefetch page thrashing rate increases to about 258 pages/sec.

3 Prefetch Memory Management

We propose a new memory management framework with the aim to reduce prefetch page thrashing and improve the performance for data-intensive online servers. Our framework is based on a separate *prefetch cache* to manage prefetched but not-yet-accessed pages (shown in Figure 3). The basic concept of prefetch cache was used earlier by Papathanasiou and Scott [22] to manage the prefetched memory for energy-efficient bursty disk I/O. Prefetched pages are initially placed in the prefetch cache. When a page is referenced, it is moved from the prefetch cache to the normal memory buffer cache and is thereafter controlled by the kernel's default page replacement policy. At the presence of high memory pressure, some not-yet-accessed pages in the prefetch cache may be moved to the normal buffer cache (called *reclamation*) for possible eviction.

The separation of prefetched pages from the rest of the memory system protects against excessive eviction of prefetched pages and allows a customized page reclamation policy for them. However, like many of the other previous studies [6, 16, 23, 29], Papathanasiou and Scott's prefetch cache management [22] requires application-supplied information on their I/O access patterns. The reliance on such information affects the applicability of these approaches and our objective is to provide transparent memory management that does not require any application assistance. The design of our prefetch memory management addresses the reclamation order among prefetched pages (Section 3.1) and the partitioning between prefetched pages and the rest of the memory buffer cache (Section 3.2). Section 3.3 describes our implementation in the Linux 2.6.10 kernel.

3.1 Page Reclamation Policy

Previous studies pointed out that the offline optimal replacement rule for a prefetching system is that every prefetch should discard the page whose next reference is furthest in the future [4, 5]. Such a rule was introduced for minimizing the run time of a standalone application process. We adapt such a rule in the context of optimizing the throughput of data-intensive online servers, where a large number of request handlers execute concurrently and each request handler runs for a relatively short period of time. Since request handlers in an online server are independent from each other, the data prefetched by one request handler is unlikely to be accessed by others in the near future. Our adapted *offline optimal replacement rules* are the following:

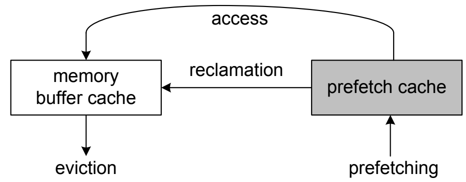

Figure 3: A separate prefetch cache for prefetched pages.

- $\mathcal{A}$. If there exist prefetched pages that will not be accessed by their respective owner request handlers, discard one of these pages first.

- $\mathcal{B}$. Otherwise, discard the page whose next reference is furthest in the future.

The optimal replacement rules can only be followed when the I/O access patterns for all request handlers are known. We explore heuristic page reclamation policies that can approximate the above rules without such information. We define a *prefetch stream* as a group of sequentially prefetched pages that have not yet been accessed. We examine the following online heuristics to identify a victim page for reclamation.

- #1. Discard any page whose owner request handler has completed its task and exited from the server.

- #2. Discard the last page from the longest prefetch stream.

- #3. Discard the last page from the prefetch stream whose last access is least recent. The recency can measured by either the elapsed wall clock time or the CPU time of each stream's owner request handler. Since in most cases only a page's owner request handler would reference it, its CPU time may be a better indicator of how much opportunity of access it had in the past.

Heuristic #1 is designed to follow the offline replacement rule $\mathcal{A}$ while heuristic #2 approximates rule $\mathcal{B}$. Heuristic #3 approximates both rules $\mathcal{A}$ and $\mathcal{B}$ since a prefetch stream that has not been referenced for a long time is not likely to be referenced soon and it may even not be referenced at all. Although heuristic #3 is based on access recency, it is different from traditional LRU in that it relies on the access recency of the prefetch stream it belongs to, not its own. Note that every page that was referenced before would have already been moved out of the prefetch cache.

Heuristic #1 can be combined with either heuristic #2 or #3 to form a page reclamation policy. In either case, heuristic #1 should take precedence since it guarantees to discard a page that is unlikely to be accessed in the near future. When applying heuristic #3, we have two different ways to identify least recently used prefetch stream. Putting these together, we have three page reclamation policies when a victim page needs to be identified for reclamation:

- **Longest**: If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page from the longest prefetch stream.
- **Coldest**: If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page of the prefetch stream which has not been accessed for the longest time.
- **Coldest+**: If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page of the prefetch stream whose owner request handler has consumed the most amount of CPU since it last accessed the prefetch stream.

3.2 Memory Allocation for Prefetched Pages

The memory allocation between prefetched pages and the rest of the memory buffer cache can also affect the memory utilization efficiency. Too small a prefetch cache would not sufficiently reduce the prefetch page thrashing. Too large a prefetch cache, on the other hand, would result in many page misses on the normal memory buffer cache. To achieve high performance, we aim to minimize the combined prefetch miss rate and the cache miss rate. A *prefetch miss* is defined as a miss on a page which was prefetched and then evicted without being accessed while a *cache miss* is a miss on a page that has already been accessed at the time of last eviction.

Previous studies [14, 28] have proposed to use hit histograms to adaptively partition the memory among prefetching and caching or among multiple processes. In particular, based on Thiébaut *et al.*'s work [28], the minimal overall miss rate can be achieved when the current miss-rate derivative of the prefetch cache as a function of its allocated size is the same as the current miss-rate derivative of the normal memory buffer cache. This indicates an allocation point where there is no benefit of either increasing or decreasing the prefetch cache allocation. These solutions have not been implemented in practice and implementations suggested in these studies require significant overhead in maintaining page hit histograms. In order to allow a relatively light-wight implementation, we approximate Thiébaut *et al.*'s result using a *gradient descent*-based greedy algorithm [24]. Given the curve of combined prefetch and cache miss rates as a function of the prefetch cache allocation, gradient descent takes steps proportional to the negative of the gradient at the current point and proceeds downhill toward the bottom.

We divide the runtime into epochs (or time windows) and the prefetch cache allocation is adjusted epoch-by-epoch based on gradient descending of the combined prefetch and cache miss rates. In the first epoch, we randomly increase or decrease the prefetch cache allocation by an *adjustment unit*. We calculate the change of combined miss rates in each subsequent epoch. This value is used to approximate the derivative of the combined miss rate. If the miss rate increases by a substantial margin, we reverse the adjustment of the previous epoch (*e.g.*, increase the prefetch cache allocation if the previous adjustment has decreased it). Otherwise, we further the adjustment made in the previous epoch (*e.g.*, increase the prefetch cache allocation further if the previous adjustment has increased it).

3.3 Implementation

We have implemented the proposed prefetch memory management framework in the Linux 2.6.10 kernel. 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 implementation, when a page in the prefetch cache is referenced, it is moved to the inactive LRU list of the Linux memory buffer cache. Prefetch cache reclamation is triggered at each invocation of the global Linux page reclamation. If the prefetch cache size exceeds its allocation, prefetched pages will be selected for reclamation until the desired allocation is reached.

We initialize the prefetch cache allocation to be 25% of the total physical memory. Our gradient descent-based prefetch cache allocation requires the knowledge on several memory performance statistics (*e.g.*, the prefetch miss rate and the cache miss rate). In order to identify misses on recently evicted pages, the system maintains a bounded-length history of evicted pages along with flags that identify their status at the time of eviction (*e.g.*, whether they were prefetched but not-yet-accessed).

The prefetch cache data structure is organized as a list of stream descriptors. Each stream descriptor points to a list of ordered page descriptors, with newly prefetched page in the tail. A stream descriptor also maintains a link to its owner process (*i.e.*, the process that initiated the prefetching of pages in this stream) and the timestamp when the stream is last accessed. The timestamp is the wall clock time or the the CPU run time of the stream's owner process depending on the adopted re-

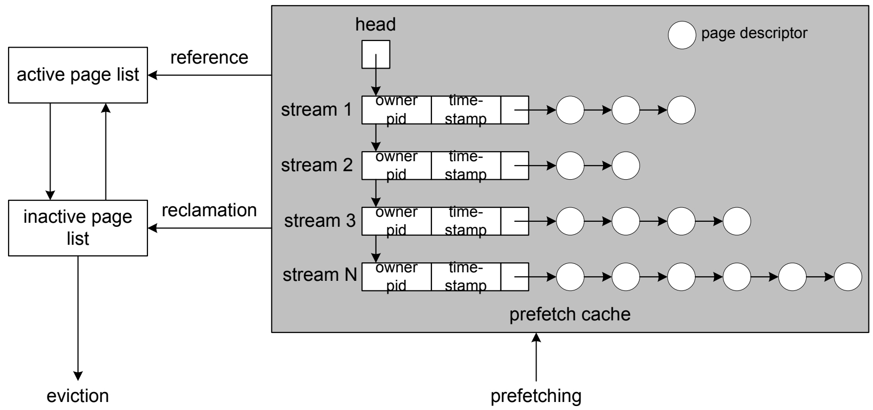

Figure 4: An illustration of the prefetch cache data structure and our implementation in Linux.

cency heuristic policy. Figure 4 provides an illustration of the prefetch cache data structure. In our current implementation, we assume a request handler is a thread (or process) in a multi-threaded (or multi-processed) server. Each prefetch stream can track the status of its owner process through the link in the stream descriptor. Due to its reliance on the process status, our current implementation cannot monitor the status of request handlers in event-driven servers that reuse each process for multiple request executions. In future implementation, we plan to associate each prefetch stream with its *open file* data structure directly. In this way, a file close operation will also cause the associated stream to be a reclamation candidate.

We assess the space overhead of our implementation. Since the number of active streams in the server is relatively small compared with the number of pages, the space consumption of stream headers is not a serious concern. The extra space overhead in each page includes two 4-byte pointers to form the stream page list. Therefore they incur 0.2% space overhead (8 bytes per 4 KB page). An additional space overhead is that used by the eviction history for identifying misses on recently evicted pages. 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. A 20-byte data structure is used to record information about each evicted page. When we limit the eviction history size to 40% of the total number of physical pages, the space overhead for the eviction history is about 0.2% of the total memory space.

4 Experimental Evaluation

We assess the effectiveness of our proposed prefetch memory management techniques on improving the performance of data-intensive online servers. Experiments were conducted on servers each with dual 2.0 GHz Xeon processors, 2 GB memory, a 36.4 GB IBM 10 KRPM SCSI drive, and a 146 GB Seagate 10 KRPM SCSI drive. Note that we lower the memory size used in some experiments to better illustrate the memory contention. 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.

The evaluation results are affected by several factors, including the I/O prefetching aggressiveness, server memory size, and the application dataset size. Our strategy is to first demonstrate the performance at a typical setting and then explicitly evaluate the impact of various factors. We perform experiments on several microbenchmarks and two real application workloads. Each experiment is run for 4 rounds and each round takes 120 seconds. The performance metric we use for all workloads is the I/O throughput observed at the application level. They are acquired by instrumenting the server applications with statistics-collection code. In addition to showing the server I/O throughput, we also provide some results on the prefetch page thrashing statistics. We summarize the evaluation results in the end of this section.

4.1 Evaluation Benchmarks

All microbenchmarks we use access a dataset of 6000 4 MB disk-resident files. On the arrival of each request, the server spawns a thread to process it. We explore the performance of four microbenchmarks with different I/O access patterns. They differ in the number of files accessed by each request handler (one to four) and the portion of each file accessed (the whole file, a random portion, or a 64 KB chunk). We use the following microbenchmarks in the evaluation:

- *One-Whole*: Each request handler randomly chooses a file and it repeatedly reads 64 KB data

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

Table 1: Benchmark statistics. The column "Whole-file access?" indicates whether a request handler would prefetch some data that it would never access. No such data exists if whole files are accessed by application request handlers. The column "Stream per request" indicates the number of prefetching streams each request handler initiates simultaneously. More prefetching streams per request handler would create higher memory contention.

blocks until the whole file is accessed.

- *One-Rand:* 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).
- *Two-Rand:* Each request handler alternates reading 64 KB data blocks from two randomly chosen files. For each file, the request handler accesses the same random number of blocks. This workload emulates applications that simultaneously access multiple sequential data streams.
- *Four-64KB:* Each request handler randomly chooses four files and reads a 64 KB random data block from each file.

We also include two real application workloads in the 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 trace recorded at the Ask Jeeves online site in early 2002.
- *Apache hosting media clips:* We include the Apache Web server in our evaluation. 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 [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. A random-size portion of each chosen file is accessed.

We summarizes our benchmark statistics in table 1.

4.2 Microbenchmark Performance

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

#1. *AccessHistory:* We use the original Linux 2.6.10 to represent kernels that manage the prefetched pages along with other memory pages using an access history-based LRU reclamation policy. Note that a more sophisticated access recency or frequency-based reclamation policy would not make much difference for managing prefetched pages. Although MRU-like schemes may benefit prefetched pages, they perform poorly with general application workloads and in practice they are only used for workloads with exclusively sequential data access pattern.

#2. *PC-AccessHistory:* The original kernel with a prefetch cache whose allocation is dynamically maintained using our gradient-descent algorithm described in Section 3.2. Pages in the prefetch cache is reclaimed in FIFO-order, which is the effective reclamation order for prefetched pages under any access history-based reclamation policy.

#3. *PC-Longest:* The original kernel with a prefetch cache managed by the Longest reclamation policy described in Section 3.1.

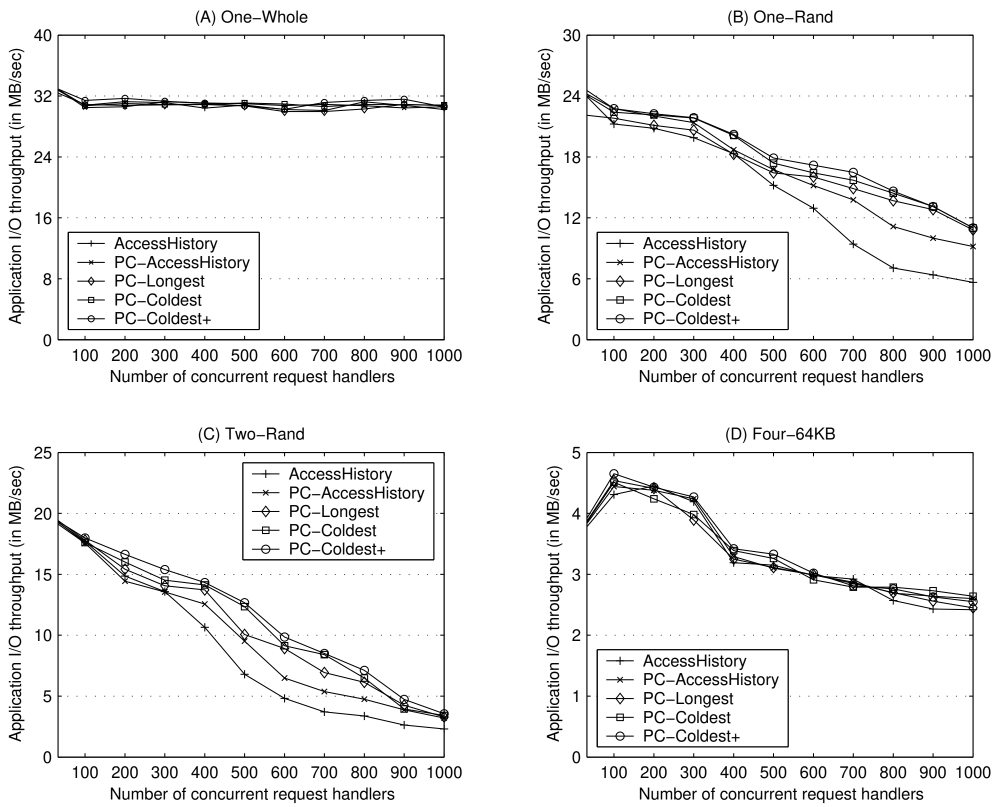

Figure 5: Microbenchmark I/O throughput at different concurrency levels. Our proposed prefetch memory management produces throughput improvement on One-Rand and Two-Rand.

#4. *PC-Coldest:* The original kernel with a prefetch cache managed by the Coldest reclamation policy described in Section 3.1.

#5. *PC-Coldest+:* The original kernel with a prefetch cache managed by the Coldest+ reclamation policy described in Section 3.1.

The original Linux 2.6.10 exhibits several performance anomalies when supporting data-intensive online servers [25]. They include a mis-management of prefetching during disk congestion and some lesser issues in the disk I/O scheduler. Details about these problems and some suggested fixes can be found in [25]. All experimental results in this paper are based on corrected kernels.

Figure 5 illustrates the I/O throughput of the four microbenchmarks under all concurrency levels. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 512 MB. Figure 5(A) shows consistently high performance attained by all memory management schemes for the first microbenchmark (One-Whole). This is because this microbenchmark accesses whole files and consequently all prefetched pages will be accessed. Further, for applications with strictly sequential access pattern, the anticipatory I/O scheduling allows each request handler to run without interruption. This results in a near-serial execution even at high concurrency levels, therefore little memory contention exists in the system.

Figure 5(B) shows the I/O throughput results for One-Rand. Since this microbenchmark does not access whole files, some prefetched pages will not be accessed. Results indicate that our proposed schemes can improve the performance of access history-based memory management at high concurrency. In particular, the improvement achieved by PC-Coldest+ is 8–97%.

Figures 5(C) shows the performance for Two-Rand. The anticipatory scheduling is not effective for this microbenchmark since a request handler in this case accesses two streams alternately. Results show that our proposed schemes can improve the performance of access history-based memory management at the concurrency level of 200 or higher. For instance, the improvement of PC-Coldest+ is 34% and two-fold at the concurrency levels of 400 and 500 respectively. This improvement is attributed to two factors: 1) the difference between PC-Coldest+ and PC-AccessHistory is attributed to the better page reclamation order inside the

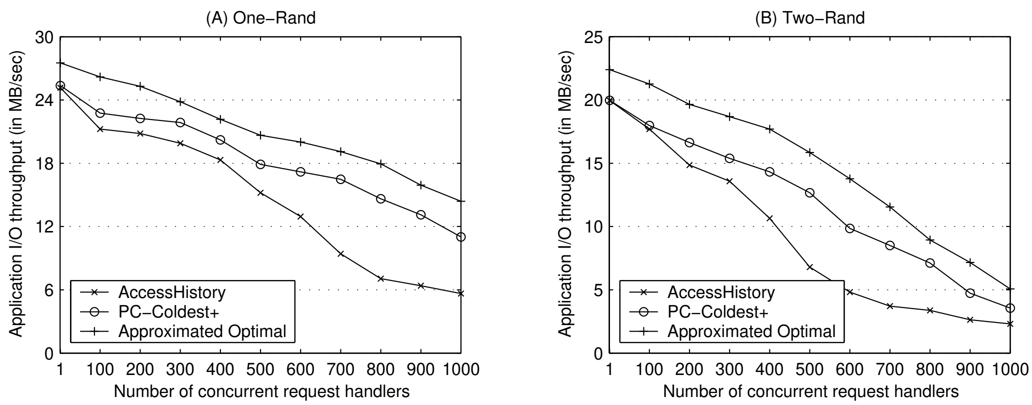

Figure 6: Comparison among AccessHistory, Coldest+, and an approximated optimal policy. The throughput of Coldest+ is 10–32% lower than that of the approximated optimal while it is 8–97% higher than that of AccessHistory in high concurrency.

prefetch cache; 2) the improvement of PC-AccessHistory over AccessHistory is due to the separation of prefetched pages from the rest of the memory buffer cache. Finally, we observe that the performance of PC-Coldest and PC-Longest are not as good as that of PC-Coldest+.

Figure 5(D) illustrates the performance of the random-access microbenchmark. We observe that all kernels perform similarly at all concurrency levels. Since prefetched pages in this microbenchmark are mostly not used, early eviction would not hurt the server performance. On the other hand, our memory partitioning scheme can quickly converge to small prefetch cache allocations through greedy adjustments and thus would not waste space on keeping prefetched pages.

Note that despite the substantial improvement on Two-Rand and One-Rand, our proposed prefetch memory management cannot eliminate memory contention and the resulted performance degradation at high concurrency. However, this is neither intended nor achievable. Consider a server with 1000 concurrent threads each prefetches up to 512 KB. Just the pages pinned for outstanding I/O operations alone occupies up to 512 MB memory space in this server.

To assess the remaining room for improvement, we approximate the offline optimal policy through direct application assistance. In our approximation, applications provide exact hints about their data access pattern through an augmented `open()` system call. With the hints, the approximated optimal policy can easily identify those prefetched but unneeded pages and evict them first (following the offline optimal replacement rule $\mathcal{A}$ in Section 3.1). The approximated optimal policy does not accurately embrace offline optimal replacement rule $\mathcal{B}$, because there is no way to determine the access order among pages from different processes without the process scheduling knowledge. For the eviction order among pages that would be accessed by their owner request handers, the approximated optimal policy follows that of Coldest+. Figure 6 shows that the performance of Coldest+ approach is 10–32% lower than the approximated optimal in high concurrency with the two tested microbenchmarks.

4.3 Performance of Real Applications

We examine the performance of two real application workloads. Figure 7 shows the I/O throughput of the index searching server at various concurrency levels. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 512 MB. All schemes perform similarly at low concurrency. We notice that the I/O throughput initially increases as the concurrency level climbs up. This is mainly due to lower average seek distance when the disk scheduler can choose from more concurrent requests for seek reduction. At high concurrency levels (100 and above), the PC-Coldest+ scheme outperforms access history-based management by 11–64%. Again, this improvement is attributed to two factors. The performance difference between PC-AccessHistory and AccessHistory (about 10%) is due to separated management of prefetched pages while the rest is attributed to better reclamation order in the prefetch cache.

In order to better understand the performance results, we examine the page thrashing rate in the server. Figure 8 shows the result for the index searching server. We observe that the page thrashing rate increases as the server concurrency level goes up. Particularly at the concurrency of 1000, the page thrashing rates are around 258, 238, 217, 207 and 200 pages/request for the five schemes respectively. We observe that the difference

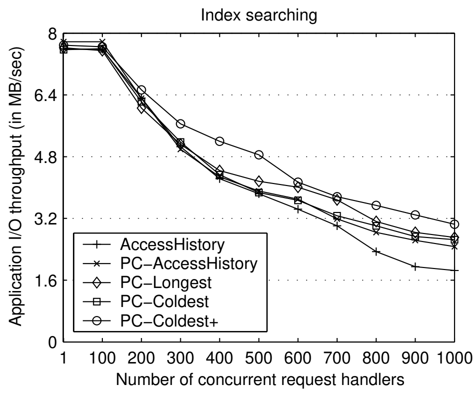

Figure 7: Throughput of the index searching server.

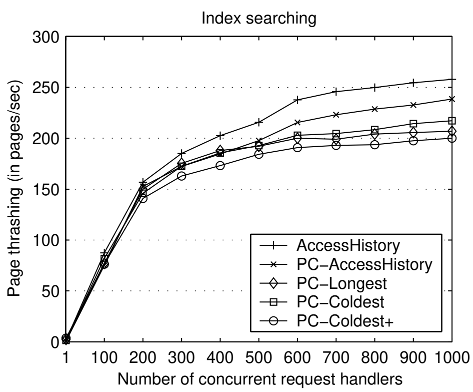

Figure 8: Prefetch page thrashing rate for the index searching server.

in page thrashing rates across the four schemes matches well with that of the throughput performance in Figure 7, indicating that the page thrashing is a main factor in the server performance degradation at high concurrency. Even under the improved management, the page thrashing rate still increases in high concurrency levels for the same reason we described earlier for the microbenchmarks.

Figure 9 shows the application IO throughput of the Apache Web server hosting media clips. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 256 MB. The four prefetch cache-based approaches perform better than access history-based approach in all concurrency levels. This improvement is due to separated management of prefetched pages. The difference among the four prefetch cache-based approaches is slight, but PC-Coldest+ persistently performs better than others. This is because of its effective reclamation management of prefetched pages.

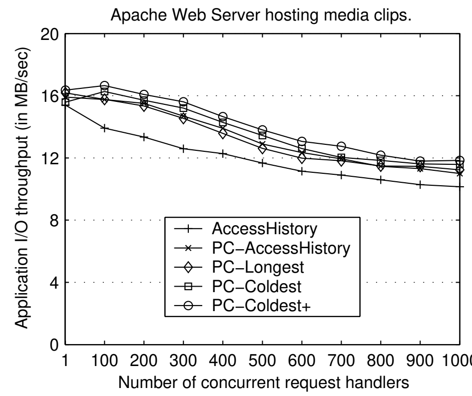

Figure 9: Throughput of the Apache Web server hosting media clips.

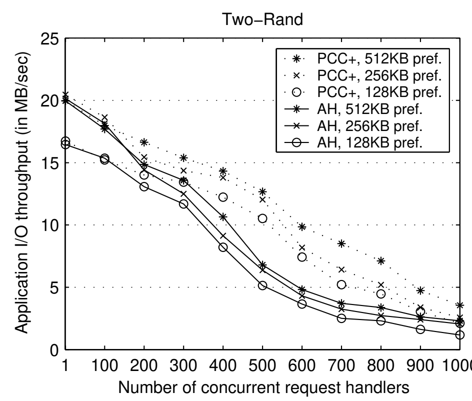

Figure 10: Performance impact of the maximum prefetching depth for the microbenchmark Two-Rand. "PCC+" stands for PC-Coldest+ and "AH" stands for AccessHistory.

4.4 Impact of Factors

In this section, we explore the performance impact of the I/O prefetching aggressiveness, server memory size, and the application dataset size. We use a microbenchmark in this evaluation because of its flexibility in adjusting the workload parameters. When we evaluate the impact of one factor, we set the other factors to default values: 512 KB maximum prefetching depth, 512 MB for the server memory size, and 24 GB for the application dataset size. We only show the performance of PC-Coldest+ and AccessHistory in the results.

Figure 10 shows the performance of the microbenchmark Two-Rand with different maximum prefetching depths: 128 KB, 256 KB, and 512 KB. Various maximum prefetching depths are achieved by adjusting the appropriate parameter in the OS kernel. We observe that our proposed prefetch memory management performs

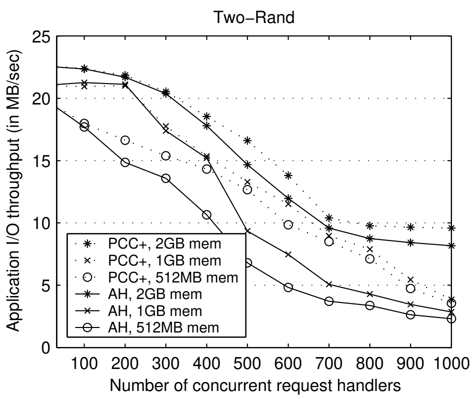

Figure 11: Performance impact of the server memory size for the microbenchmark Two-Rand. "PCC+" stands for PC-Coldest+ and "AH" stands for AccessHistory.

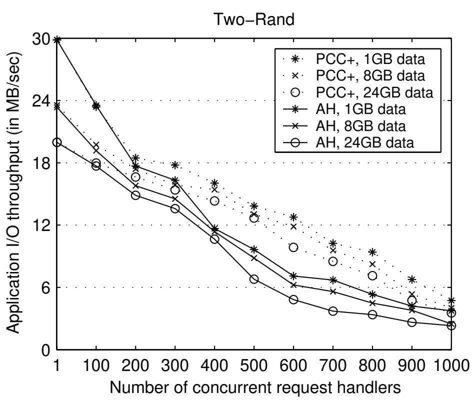

Figure 12: Performance impact of the workload data size for the microbenchmark Two-Rand. "PCC+" stands for PC-Coldest+ and "AH" stands for AccessHistory.

consistently better than AccessHistory at all configurations. The improvement tends to be more substantial for servers with higher prefetching depths due to higher memory contention (and therefore more room to improve).

Figure 11 shows the performance of the microbenchmark at different server memory sizes: 2 GB, 1 GB, and 512 MB. Our proposed prefetch memory management performs consistently better than AccessHistory at all memory sizes. The improvement tends to be less substantial for servers with large memory due to lower memory contention (and therefore less room to improve).

Figure 12 illustrates the performance of the microbenchmark at different workload data sizes: 1 GB, 8 GB, and 24 GB. Intuitively better performance is correlated with smaller workload data sizes. However, the change on the absolute performance does not significantly affect the performance advantage of our strategy over access history-based memory management.

4.5 Summary of Results

- Compared with access history-based memory management, our proposed prefetch memory management scheme (PC-Coldest+) can improve the performance of real workloads by 11–64% at high execution concurrency. The performance improvement can be attributed to two factors: the separated management of prefetched pages and enhanced reclamation policies for them.

- At high concurrency, the performance of PC-Coldest+ is 10–32% below an approximated optimal page reclamation policy that uses application-provided I/O access hints.

- The microbenchmark results suggest that our scheme does not provide performance enhancement for applications that follow strictly sequential access pattern to access whole files. This is because the anticipatory I/O scheduler executes request handlers serially for these applications, thus eliminating memory contention even at high concurrency. In addition, our scheme does not provide performance enhancement for applications with only small-size random data accesses since most prefetched pages will not be accessed for these applications (and thus their reclamation order does not matter).

- The performance benefit of the proposed prefetch memory management may be affected by various system parameters. In general, the benefit is higher for systems with more memory contention. Additionally, our scheme does not degrade the server performance on all the configurations that we have tested.

5 Related Work

Concurrency management. 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 [20]. 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 provided a user-level thread package that can scale to support hundreds of thousands of threads [32]. These studies mostly targeted CPU-intensive workloads with light disk I/O activities (*e.g.*, the typical Web server workload and application-level packet routing). They did not directly address the memory contention problem for data-intensive online servers at high concurrency.

Admission control. The concurrency level of an online server can be controlled by employing a fixed-size process/thread pool and a request waiting queue. QoS-oriented admission control strategies [18, 30, 31, 33] can adjust the server concurrency according to a desired performance level. Controlling the execution concurrency may mitigate the memory contention caused by I/O prefetching. However, these strategies may not always identify the optimal concurrency threshold when the server load fluctuates. Additionally, keeping the concurrency level too low may reduce the server resource utilization efficiency (due to longer average disk seek distance) and lower the server responsiveness (due to higher probability for short requests to be blocked by long requests). Our work complements the concurrency control management by inducing slower performance degradation when the server load increases and thus making it less critical to choose the optimal concurrency threshold.

Memory replacement. A large body of previous work has explored efficient memory buffer page replacement policies for data-intensive applications, such as Working-Set [8], LRU/LFU [19], CLOCK [27], 2Q [13], Unified Buffer Management [15], and LIRS [12]. All these memory replacement strategies are based on the page reference history such as reference recency or frequency. In data-intensive online servers, a large number of prefetched but not-yet-accessed pages may emerge due to aggressive I/O prefetching and high execution concurrency. Due to a lack of any access history, these pages may not be properly managed by access recency or frequency-based replacement policies.

Memory management for Caching and Prefetching. 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]. Patterson *et al.* examined memory management based on application-disclosed hints on application I/O access pattern [23]. Such hints were used to construct a cost-benefit model for deciding the prefetching and caching strategy. Kimbrel *et al.* explored the performance of application-assisted memory management for multi-disk systems [16]. Tomkins *et al.* further expanded the study for multi-programed execution of several processes [29]. Hand studied application-assisted paging techniques with the goal of performance isolation among multiple concurrent processes [10]. 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]. Memory management with application assistance or application-level information can achieve high (or optimal) performance. However, the reliance on such assistance may affect the applicability of these approaches and the goal of our work is to provide transparent memory management that does not require any application assistance.

Chang *et al.* studied the automatic generation of I/O access hints through OS-supported speculative execution [7, 9]. The purpose of their work is to improve the prefetching accuracy. They do not address the management of prefetched memory pages during memory contention.

Several researchers have addressed the problem of memory allocation between prefetched and cached pages. Many such studies required application assistance or hints [5, 16, 23, 29]. Among those that did not require such assistance, Thiébaut *et al.* [28] and Kaplan *et al.* [14] employed hit histogram-based cost-benefit models to dynamically control the prefetch memory allocation with the goal of minimizing the overall page fault rate. Their solutions require expensive tracking of page hits and they have not been implemented in practice. In this work, we employ a greedy partitioning algorithm that requires much less state maintenance and thus is easier to implement.

6 Conclusion

This paper presents the design and implementation of a prefetch memory management scheme supporting data-intensive online servers. Our main contribution is the proposal of novel page reclamation policies for prefetched but not-yet-accessed pages. Previously proposed page reclamation policies are principally based on the page access history, which are not well suited for pages that have no such history. Additionally, we employ a gradient descent-based greedy algorithm that dynamically adjusts the memory allocation for prefetched pages. Our work is guided by previous results on optimal memory partitioning while we focus on a design with low implementation overhead. We have implemented the proposed techniques in the Linux 2.6.10 kernel and conducted experiments using microbenchmarks and two real application workloads. Results suggest that our approach can substantially reduce prefetch page thrashing and improve application performance at high concurrency levels.

Acknowledgments The implementation of the statistics for evicted pages (mentioned in section 3.3) is based on a prototype by Athanasios Papathanasiou. We also would like to thank Athanasios Papathanasiou, Michael Scott, and the anonymous reviewers for their valuable

comments that greatly helped to improve this work.

References

[1] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Circus: Opportunistic Block Reordering for Scalable Content Servers. In *Proc. of the Third USENIX Conf. on File and Storage Technologies (FAST'04)*, 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] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[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] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation (OSDI'94)*, Monterey, CA, November 1994.

[7] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, February 1999.

[8] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11:323–333, May 1968.

[9] 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.

[10] S. M. Hand. Self-paging in the Nemesis Operating System. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, February 1999.

[11] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP'01)*, pages 117 – 130, Banff, Canada, October 2001.

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

[13] 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.

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

[15] 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 Symp. on Operating Systems Design and Implementation (OSDI'00)*, San Diego, CA, October 2000.

[16] 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 Second USENIX Symp. on Operating Systems Design and Implementation (OSDI'96)*, Seattle, WA, October 1996.

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

[18] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proc. of the IEEE INFOCOM*, pages 651–659, Tel-Aviv, Israel, March 2000.

[19] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems J.*, 9(2):78–117, 1970.

[20] 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.

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

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

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

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

[25] K. Shen, M. Zhong, and C. Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proc. of the 4th USENIX Conf. on File and Storage Technologies (FAST'05)*, San Francisco, CA, December 2005.

[26] E. Shriver, C. Small, and K. Smith. Why Does File System Prefetching Work ? In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[27] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[28] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.

[29] 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.

[30] T. Voigt and P. Gunningberg. Dealing with Memory-Intensive Web Requests. Technical Report 2001-010, Uppsala University, May 2001.

[31] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[32] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, pages 268–281, Bolton Landing, NY, October 2003.

[33] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems*, Seattle, WA, March 2003.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP'01)*, pages 230–243, Banff, Canada, October 2001.