Improving Disk Throughput in Data-Intensive Servers *

Enrique V. Carrera and Ricardo Bianchini

Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019

{vinicio,ricardob}@cs.rutgers.edu

Technical Report DCS–TR–500, September 2002, Revised May 2003

Abstract

Low disk throughput is one of the main impediments to improving the performance of data-intensive servers. In this paper, we propose two management techniques for the disk controller cache that can significantly increase disk throughput. The first technique, called File-Oriented Read-ahead (FOR), adjusts the number of read-ahead blocks brought into the disk controller cache according to file system information. The second technique, called Host-guided Device Caching (HDC), gives the host control over part of the disk controller cache. As an example use of this mechanism, we keep the blocks that cause the most misses in the host buffer cache permanently cached in the disk controller. Our detailed simulations of real server workloads show that FOR and HDC can increase disk throughput by up to 34% and 24%, respectively, in comparison to conventional disk controller cache management techniques. When combined, the techniques can increase throughput by up to 47%.

1 Introduction

The continuous improvement in the processing power of servers is putting tremendous pressure on their I/O subsystems. In particular, disk drives are frequently the most critical I/O components, since disk performance is limited by mechanical delays. In order to reduce disk I/O overheads, several techniques and optimizations have been proposed. This literature includes work on optimizing the scheduling of disk requests [33, 3, 17, 15], disk arrays [10, 4, 34], and optimizing disk writes using logs [23, 13, 32].

Despite these techniques and optimizations, low disk throughput is still a serious problem for data-intensive servers, such as Web proxies, email and news servers, multimedia servers, and database servers. An important reason for this is that *current disk controller caches are not designed to handle server workloads well*. More specifically, current caches are optimized for the simultaneous sequential access of a relatively small number of large files. To amortize the cost of a disk access, the controller reads a fixed number of consecutive blocks ahead and stores them in its cache. The cache is divided into segments that correspond to sequential streams of data. An entire segment is the minimum unit of allocation and replacement.

This approach works well for several classes of systems, but does not for data-intensive servers for three reasons: (1) several servers, such as Web, proxy, and file servers frequently access small files; (2) all servers access a large number of files concurrently; and (3) server workloads exhibit a large number of disk blocks that frequently miss in the controller cache. Thus, our goal is to propose new management techniques for the disk controller that are efficient for servers as well. In this paper, we propose two such techniques.

Our techniques are implemented by the disk controller and are orthogonal to the caching and prefetching already implemented by the operating system and/or application. The two techniques exploit the processing and memory capacity of modern disk drives to implement file system-guided caching and read-ahead. In more detail, the first technique, which we call File-Oriented Read-ahead (FOR), employs a block-based (rather than segment-based) cache organization and adjusts the number of read-ahead blocks according to file system information that is produced by the disk controller itself. The idea here is to avoid reading ahead beyond the end of a set of consecutive blocks of a file. This strategy reduces the time wasted with reading useless blocks when the host processor accesses small files.

The second technique, which we call Host-guided Device Caching (HDC), is based on the observations that servers normally use disk arrays, and each disk has a reasonable amount of memory. The host file system can use those memories as an extension of its buffer cache. The idea is to give the host direct control over part of the disk controller cache. An an example use of this mechanism, we cache in

*This research was supported in part by NSF grant EIA-0203922.

the disk controller the blocks that produce the most misses in the buffer cache. This strategy reduces the number of disk accesses by increasing the overall cache hit rate.

We evaluate these techniques using detailed disk simulations and a combination of real and synthetic workloads. All aspects of real disk drives are simulated, including the costs of the new proposed functionality. Using synthetic workloads, we find that FOR and HDC outperform comparable techniques for a wide range of parameters. Using real Web, proxy, and file server workloads, we find that FOR can increase disk throughput by up to 34%, whereas HDC can increase disk throughput by up to 24%. The combination of the techniques increases disk throughput by up to 47%.

The remainder of this paper is organized as follows. In the next section we introduce some basic concepts about modern disk drives and operating system prefetching. Section 3 discusses previous works related to our proposed techniques. The FOR mechanism is presented in section 4. In section 5, we describe the HDC strategy. Section 6 presents our results, comparing our techniques against conventional approaches. Finally, section 7 draws our main conclusions.

2 Background

2.1 Modern Disk Drives

Modern disk drives have significant processing and memory capacity. For instance, the Cheetah X15-36LP disk drive [25] already includes an integrated Ultra160 SCSI controller with 8 MBytes of memory and an ARM966E-S 32-bit RISC core clocked at 200 MHz. The memory is mostly used for caching disk blocks, whereas the processor (the disk controller) implements device control, logical-to-physical block address translation, request scheduling and block remapping, and cache management.

Perhaps the most interesting characteristic of disk controller caches is that, differently from other caches, they have almost no temporal locality; the host processor caches the data accessed in its own much larger memory, i.e. in the file system buffer cache.

The management of disk controller caches is also different than that of other caches. Controller caches are normally divided into independent segments that correspond to sequential streams of data. Effectively, each I/O stream is treated as having its own cache. When the controller detects that there are more streams than segments, segment replacement takes place. The segment-replacement policy is usually LRU. However, FIFO, random, and round-robin policies have also been proposed [30, 11, 26]. Regardless of the specific policy, the whole victim segment is replaced to make room for the new stream.

Because disk workloads frequently access data sequentially, i.e. with good spatial locality, disk controllers normally read ahead several blocks (on each cache miss) and cache all blocks read [11]. Subsequent requests for the blocks that were read ahead can be served from the disk controller cache without incurring in extra overheads. The number of blocks read at a time is set to fill a segment segment by default. For example, the IBM Ultrastar 36Z15 disk can support up to 27 variable-sized segments in its 4 MBytes of memory. By default, as many as 128-KBytes are read at once. Nevertheless, the amount of data read-ahead does not have to match the segment size exactly. It can be fixed at a value smaller than the segment size or variable, according to a partial track buffering policy [26]. Even when segment sizes are allowed to vary, the size of all segments has to be the same at each point in time to reduce the complexity of space management.

The problem with large read-aheads is that some or all the read-ahead blocks may belong to a different file than the one that was accessed. Although the read-ahead mechanism does not affect the access time of the current request, it can certainly delay the next non-sequential accesses. The reason is that no other request can start before the disk head finishes reading all the blocks that had already been scheduled. The duration of a read operation of r blocks is captured by the following formula: $T(r) = seek\_time + rot\_latency + (r \times S)/xfer\_rate$, where $seek\_time$ is the time needed to move the disk head to the desired cylinder, $rot\_latency$ is the time that the disk head waits for the target block to rotate into position, $xfer\_rate$ is the speed of the media transfer as the target block rotates under the head, S is the block size, and r is the number of blocks read at once. The $seek\_time$ is a non-linear function that can be approximated by the equation [24]:

$$seek\_time(n) = \begin{cases} 0 & n = 0 \\ \alpha + \beta\sqrt{n} & 0 < n \leq \theta \\ \gamma + \delta n & n > \theta \end{cases}$$

where n is the number of cylinders to be traveled, and α, β, γ, δ, and θ are device-specific parameters. These latter parameters are not very intuitive; their values are obtained by performing regressions on actual seek times. We will use these definitions later.

2.2 Disk Arrays

Data-intensive servers often have to resort to multiple disks (disk arrays) for performance and/or storage capacity reasons. These disk arrays can have additional hardware support (e.g. a RAID controller), or can be directly managed by the operating system. Regardless of the type of implementation, data redundancy or replication may be necessary for servers that must be reliable.

Each disk of an array has its own controller, just as discussed in the previous section. In hardware RAID configurations, the RAID controller may also cache data. However,

even in this case, the caching at the different controllers is independent.

The throughput of a disk array is proportional to the number of individual disks, assuming perfect load balancing. The simplest and best-known technique for balancing the load is striping. Striping groups several sequential disk blocks in units of fixed size and lays those units out across the physical disks in round-robin fashion. Smaller striping units normally lead to better load balancing. However, very small units can hurt performance [28], as each large request gets fragmented into sub-requests to multiple disks. In the absence of contention, the response time of a request for r blocks can be defined as $T(r) = \gamma(D) \times T(r/D)$, where D is the number of sub-requests and $\gamma(D)$ is a factor that depends on the probability distribution function of $T(r/D)$. For instance, $\gamma(D) = 2D/(D+1)$ for a uniform distribution [28].

For the purposes of this paper, the most important aspect of striping is its relationship with disk read-ahead. Since striping spreads files across multiple disks, it increases the chance that consecutive blocks of a disk do not belong to the same file. Thus, read-aheads that are larger than the striping unit may read useless data.

2.3 Operating System Prefetching

General-purpose operating systems incorporate prefetching techniques in their file systems. *Operating system prefetching and controller cache read-ahead operate independently*; regardless of the type of request, the controller will read-ahead on a cache miss (unless read-ahead operations have been explicitly disabled).

The sequential prefetching algorithm is the most frequently used technique in UNIX-like operating systems [20]. The algorithm adapts the number of disk blocks to be prefetched according to the sequentiality of the accesses to a specific file. For a large sequential file, the amount of data prefetched can reach a maximum value, e.g. 64 KBytes in Linux. When accesses are random, the amount of data prefetched can be reduced to zero.

In the case of sequential accesses, the operating system or even the device driver may be able to coalesce requests for consecutive blocks into a single large request, depending on the timing of the requests. When coalescing is possible, the disk controller replies to the request with several blocks.

3 Related Work

There have been several works on improving disk I/O performance. Most of them discuss techniques that are external to the disk drive. Typical examples are techniques for improving the prefetching and caching of disk blocks [21, 6, 27], scheduling requests [3, 15], and for optimizing disk writes [13, 23, 32]. Our techniques deal with read-ahead (a form of prefetching) and caching of disk data, but their focus is on the disk controller cache. There are several important differences between the controller cache and other caches, such as the buffer cache, the Disk Caching Disk [13], or even a second-level buffer cache [35]: (1) accesses to the controller cache lack temporal locality; (2) the controller cache is orders of magnitude smaller than other caches; (3) controller cache read-ahead is restricted to sequential disk blocks; and (4) the management of the controller cache cannot be excessively complicated to avoid overheads. These differences mean that techniques proposed for other caches rarely apply to controller caches.

Perhaps the most closely related operating system-level technique is explicit grouping [12]. This technique groups small files that belong to the same directory into consecutive disk blocks, so that read-aheads fetch useful data when multiple files in the directory have to be accessed. Explicit grouping shares the same goal as FOR, but requires that a meaningful grouping of files be found and maintained. FOR does not rely on such grouping and can more easily adapt to changes in file access behavior.

A few works do discuss read-ahead and caching at the disk controller level. Shriver presents a good summary of the proposed read-ahead strategies in [26]. Unlike our approach however, these techniques do not rely on file system information to improve the read-ahead. In terms of controller cache replacement, a few algorithms [16, 30, 11] have been proposed for segment-based caches. None of these works considered host-guided caching or block-based cache organizations.

A few other papers discuss techniques for disk controllers. These papers include works on the scheduling of disk requests [33], utilizing free bandwidth [17], the replication of disk data [34], and disk arrays [10, 34, 4]. These techniques reduce seek and/or rotational overheads through sophisticated request scheduling or by accessing the closest copy of a block. Our techniques are orthogonal to these contributions. In particular, FOR reduces the transfer time by adjusting read-ahead to file size, whereas HDC reduces the number of disk accesses by increasing the cache hit rate at the disk controller.

Programmable disks have also been studied. However, the proposed approaches sought to reduce the amount of data transferred between disks and main memory [1, 22], to implement flexible RPC on disks [29], and/or to offload file system functionality from the host processor to the disks [9]. All of these works assumed significant processing capacity at the disks. Unlike these proposals, our focus in on improving disk throughput through more sophisticated cache management techniques. Furthermore, FOR and HDC require only minimal programmability to accommodate different file systems and workloads.

Finally, more loosely related are works on main memory

caching for servers, e.g. [19, 2], and application-controlled main memory caching [6]. Our work is complementary to the server caching studies in that we focus on the disk controller cache, which exhibits significantly different characteristics than a main memory cache, as mentioned above. Our work differs from application-controlled caching as well. In this approach, applications are allowed to influence the main memory caching decisions made by the operating system. Again, the sets of policies that are appropriate for a disk controller cache are quite different than those for main memories. Further, it does not seem reasonable to expect applications to understand the details of how disk controller caches operate. Nevertheless, HDC allows the operating system a similar ability to manage the disk controller cache.

4 File-Oriented Read-Ahead

Since the locality of the disk accesses is normally associated with files [27], the read-ahead mechanism implemented by the disk controller is very useful for files larger than a disk block. However, most of the read-ahead blocks can be useless when applications (e.g. Web servers, Web proxies, file servers) access relatively small files. The reason is that the disk controller has no notion of the layout of files on disk (which is determined by the host file system), so the read-ahead may fetch data from files other than the one actually accessed. When read-ahead operations bring useless data into the cache, the disk utilization due to these operations is not amortized and throughput decreases. Furthermore, the useless blocks pollute the controller cache.

In order to eliminate useless read-ahead blocks, our FOR technique determines that, on each disk access, a block should only be read-ahead if it belongs to the same file as the block that was actually accessed. Thus, the disk controller should read consecutive blocks ahead, up to a block belonging to a different file or the read-ahead size.

To decide whether or not a block should be read-ahead, the controller requires information about the layout of files on disk. However, storing this information can be expensive in terms of space, given the size of current disks and the memory restrictions in disk drives. Thus, our solution is to create a bitmap with one bit for each block on disk. A bit in the bitmap is set to 1 only if the corresponding disk block is the logical continuation within a file of the previous physical block on disk. Otherwise, the bit is set to 0. This approach allows us to conserve disk controller memory, since we only need one bit for each 4 or 8-KByte disk block. This ratio of control to useful space means that the overhead of the bitmap is only 0.003% (i.e. 100%/(8×4096)) for 4-KByte blocks. In addition, using this bitmap, the decision of how many blocks to read-ahead is very simple: from the location of the block that missed in the cache, we only need to count the number of bits until a 0 bit is found. As we show in section 6, this bitmap is enough to increase the effectiveness of the disk usage and improve the performance of several server workloads.

There are a few possibilities for how to initialize and maintain the bitmap. One approach would be for the disk controller to understand the layout of the file system on disk. This scheme would obviate the need for programming the controller in the field, but would make the drive file system-dependent. Another approach would be for the controller to be programmed with manufacturer-provided routines for each file system. These routines could be part of the drive's installation package. This scheme would require programming the controller in the field, but programming would be under control of the manufacturer. Finally, the third approach would be for the controller to be programmed by the file system/device driver. This scheme would require more sophisticated drivers. Determining the best approach is beyond the scope of this paper; what is important to realize is that the disk controller is the only component of the system that has accurate information about the disk geometry. Given that FOR is implemented at the controller, it can rely on this accurate information.

With FOR's sophisticated read-ahead, segments associated with small files can be under-utilized. To avoid this effect, FOR could use variable-size segments. However, the use of segments with different sizes complicates the management of free space. Thus, we introduce a more flexible block-based organization for the controller cache. The idea behind this organization is to assign blocks to new streams from a pool of free blocks on demand. When FOR runs out of free blocks, it replaces blocks using an MRU policy. The overall effect of this scheme is that we can have segments with multiple sizes using simple cache management algorithms.

Thus, FOR has two key benefits with respect to conventional drives: lower disk utilization (due to its read-ahead mechanism) and higher controller cache hit rates (due to the combination of its read-ahead with a block-based cache organization). The lower utilization is clearly observed in the formula we mentioned in section 2: $T(r) = seek\_time + rot\_latency + (r \times S)/xfer\_rate$. FOR reduces r for small files; seek time, rotational time, and transfer rate are not affected. *Prefetching and large read-aheads, on the other hand, try to hide disk latency, not to reduce utilization.* Working out the numbers for the parameters of the IBM Ultrastar 36Z15 drive (15000 rpm, ∼440 sectors/track, 3.4 msecs average seek time) and a 4-KByte average file size, FOR reduces the disk utilization by 29% in comparison to a conventional 128-KByte read-ahead.

The higher hit rates of FOR can also be modeled formally. For that, assume that a server is sequentially reading t files, the size of the cache in blocks is c, the number of segments is s, p is the number of blocks requested by the file system/device driver on each access ($p \geq 1$ because of file system prefetching), and f is the average size of files in blocks.

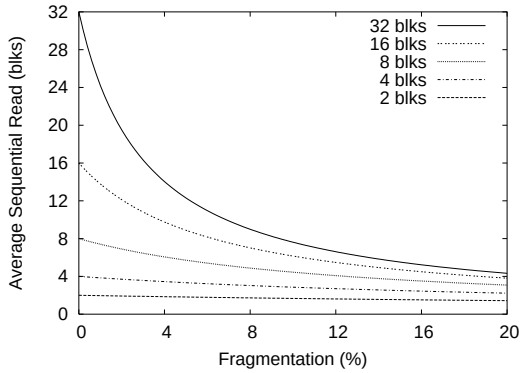

Figure 1: Average sequential read as function of fragmentation degree for different file sizes.

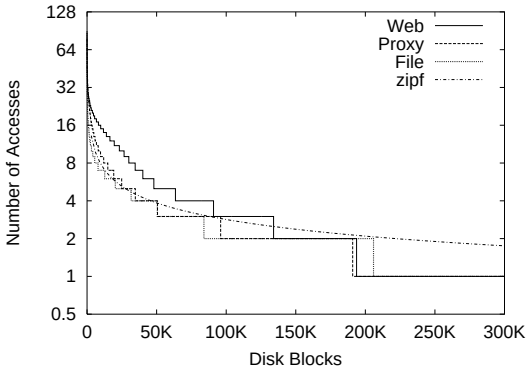

Figure 2: Distribution of disk block accesses in three real workloads. Note that the Y-axis is on a log scale.

The hit rate h for a conventional cache is:

$$h = \begin{cases} (min(f, c/s) - 1)/min(f, c/s) & t \leq s \\ (p-1)/p & t > s \end{cases}$$

The hit rate h_{for} for FOR is:

$$h_{for} = \begin{cases} (f-1)/f & t \leq c/f \\ (p-1)/p & t > c/f \end{cases}$$

Because $c/f > s$ for small files and $f \geq p$ (as the file system does not prefetch beyond the end of a file), FOR exhibits higher hit rates than a conventional read-ahead cache. Again, assuming the parameters of the IBM Ultrastar 36Z15 drive (4-MByte cache, maximum of 27 segments), we find that FOR should exhibit a higher hit rate for average file sizes that are smaller than 128 KBytes and $t > 27$. Both of these characteristics are common to several types of data-intensive servers.

The FOR benefits increase with smaller average file size or higher fragmentation, i.e. a higher rate of blocks that are consecutive logically, but not physically on disk. In fact, even a small amount of fragmentation can increase the benefits of FOR significantly. Figure 1 shows the average number of sequential blocks for different average files sizes (in disk blocks), as we varied the percentage of fragmentation. The figure shows that 5% of fragmentation can reduce the sequentiality of 32 and 8-block files by 62% (from 32 to about 12 sequential blocks) and 29% (from 8 to about 6 sequential blocks), respectively.

5 Host-Guided Device Caching

The disk controller cache is currently used solely as a speed-matching and read-ahead buffer, since it is typically small (between 4 and 16 MBytes) compared to a main memory/buffer cache. However, note that data-intensive servers frequently rely on disk arrays with per-disk caches. The combined size of all these controller caches is non-negligible, especially given the extremely high cost of each actual disk access. We argue that these caches can be made more useful if (at least a non-trivial fraction of) each of them can be managed directly by the host processor, hence the name Host-guided Device Caching (HDC). The host processor would then integrate the management of the host and disk controller caching. This direct management can be used to determine which blocks to cache, when to cache them, which replacement policy to use, when to flush dirty blocks to disk, etc. For example, the host file system can use part of the disk controller caches as an array-wide victim cache for its buffer cache with this type of caching control. We refer to the part of each controller cache that is managed by the host as the HDC cache.

In this paper we examine another example of host-guided caching. Based on the observation that the workloads of data-intensive servers often exhibit widely different file popularity (i.e. some files are accessed much more frequently than others), we propose that the set of disk controller caches permanently cache the files (or blocks) that cause the most misses in the buffer cache.

To motivate this idea, figure 2 shows the number of accesses for the 300000 most accessed disk blocks according to real workloads for a Web server, a proxy server, and a file server. Note that these are disk access statistics resulting from misses in the application and file system caches. (The details of the workloads and servers that led to theses results are discussed later.) The figure also plots a Zipf distribution [5] with an α coefficient of 0.43. We can see that a non-trivial fraction of these disk accesses can be transformed into controller cache hits, if we can cache the 10000 blocks that miss the most, for example, across the disk array. More generally, for an array-wide cache of H HDC blocks, the expected hit rate can be approximated as $h = z_\alpha(H, N)$, where N is the total number of blocks and $z_\alpha(H, N)$ represents the accumulated probability of requesting the H most accessed blocks in a Zipf distribution of requests to N blocks.

In the particular strategy we study here, each disk controller only caches blocks that are stored on its respective disk. Our goal is to simplify the controller cache manage-

ment, especially for disk writes. More complex caching policies could be implemented (e.g. cooperative caching between controllers), but our simple strategy already provides significant gains and requires very little support on the disk controller side. The support involves three simple commands that can be sent by the host processor to a disk controller: *pin_blk()*, *unpin_blk()*, and *flush_hdc()*. *pin_blk()* reads a disk block and marks it as non-replaceable in the disk controller cache. A non-replaceable block cannot be dropped from the cache while the non-replaceable flag is set. The non-replaceable flag can only be cleared by the *unpin_blk()* call. By default, HDC dirty blocks are not automatically updated on disk. The operating system (i.e. file system or device driver) must call *flush_hdc()* in order to write all the dirty blocks in the HDC cache to disk. The operating system should call *flush_hdc()* as a result of an application that wants to sync its writes to disk or periodically (e.g. when it wants to sync the dirty blocks of the buffer cache every 30 seconds, as in Unix). Other additional calls could also be useful. For instance, calls for changing the dirty block policy and/or keeping track of the amount of memory used with these non-replaceable blocks. However, we can assume that the operating system is able to manage the coherence of dirty blocks and the number of non-replaceable blocks in each device.

To manage the HDC cache, the host needs to determine: (i) how many blocks are going to be cached; (ii) which blocks those are; and (iii) when the blocks are going to be cached. To answer these questions, we propose to divide the server's execution into fixed periods of time. Each period can correspond to a meaningful cycle of the server's life (e.g. a day, a week). The blocks that cause the most buffer cache misses during a period can be cached in the beginning of the period, based on the history of one or more previous periods. These blocks should remain cached throughout the period. The size of each HDC cache can be pre-established, based on the overall size of the disk controller memory and experiments with the particular server, since there is a tradeoff between the amount of memory used by HDC and the amount of memory left as a read-ahead cache. Both the HDC hit rate and the read-ahead requirements depend on specific characteristics of the workload.

The tradeoff between the space dedicated to HDC and read-ahead caching can be described formally. The maximum array-wide amount of memory allocated to HDC (in blocks) should be $H_{max} = D \times c - R_{min}$, where D is the number of disks, c is the size of the disk controller cache in blocks, and R_{min} is the minimum size of the read-ahead cache. In a blind read-ahead implementation $R_{min} = t \times (c/s)$ blocks, whereas FOR only needs $R_{min} = t \times f$ blocks, such that t is the number of simultaneous I/O streams, s is the number of segments in each cache, and f ($c/s > f$ for small files) is the average size of the streams in blocks.

Finally, note that the evaluation of HDC presented in section 6 assumes a block-based organization of the disk controller cache like that used for the FOR technique. However, nothing in HDC depends on the cache organization. The same idea and mechanisms can be applied to a segment or block-based caches.

Parameter	Default Value
Number of disks	8
Disk size	18 GBytes
Average disk seek time	3.4 msecs
Average rotational latency	2.0 msecs
Raw disk transfer rate	54 MB/sec
Disk controller interface	Ultra160
Disk controller cache size	4 MBytes
Disk block size	4 KBytes
Segment size	128, 256, or 512 KBytes
Number of segments	27, 13, or 6
Disk-resident bitmap	546 KBytes

Table 1: Main parameters and their default values.

6 Experimental Results

In this section we evaluate the two techniques we propose. For that, we have built a simulator that is described in the next subsection. To study the behavior of the techniques in a controlled manner, we use synthetic workloads in subsection 6.2. Subsection 6.3 presents results for real server workloads, quantifying the benefits of our techniques in practice.

6.1 Simulator

We have developed a detailed event-driven disk simulator based on the MINT [31] front-end. The simulator mimics the behavior of an array of SCSI disks attached to a single Ultra160 SCSI card. Logical disk blocks are striped along the array of physical disks. Each disk has its own controller with a programmable processor and some amount of memory. Contention for buses, memories, and other components is simulated in detail. For request scheduling, each disk controller has a queue that implements the LOOK algorithm [26]. Before queuing a new request, the disk controller checks the cache to see if the block is already present in its cache.

The main architectural parameters of our simulations are listed in table 1. We have modeled the disk parameters after our own 18GB IBM Ultrastar 36Z15 drive [14]. The seek time and rotational latency are simulated in detail. For the seek time we have approximated the nominal values of our IBM drive using $\alpha = 0.9336$, $\beta = 0.0364$, $\gamma = 1.5503$, $\delta = 0.00054$, and $\theta = 1150$ (subsection 2.1).

The FOR technique is simulated exactly as it was described. The HDC simulations assume perfect knowledge of the future, i.e. the blocks to be placed in the HDC cache are determined based on statistics generated by the correspond-

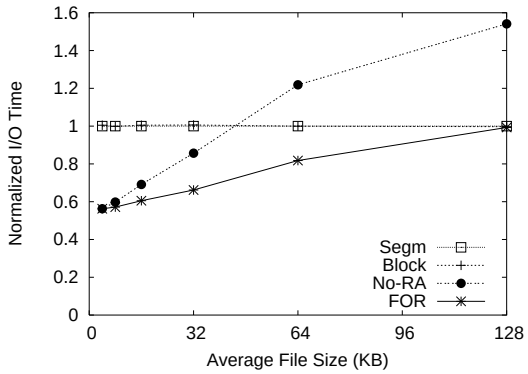

Figure 3: Normalized I/O time as function of average file size. Number of simultaneous streams = 128.

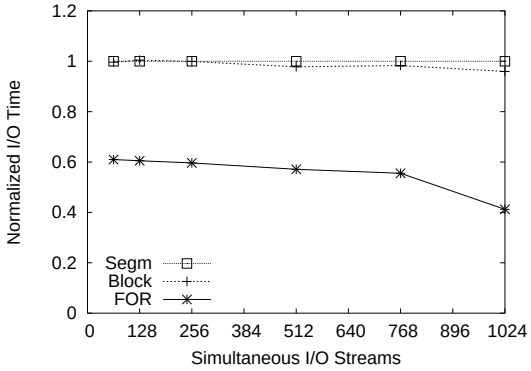

Figure 4: Normalized I/O time as function of number of simultaneous I/O streams. Average file size = 16 KBytes.

ing trace. The dirty HDC blocks are only updated to disk at the end of each simulated execution, which is a fine assumption for Web and proxy servers. For file servers, a periodic update policy would have been more realistic. However, we have determined the effect of such periodic syncs on overall throughput to be negligible ($< 1\%$), assuming periods of 30 seconds, for all our simulations.

Besides the FOR and HDC techniques, we simulate a conventional disk drive with blind read-ahead and variable-sized segments, just like our IBM drive. However, our servers never require segments of more than 128 KBytes, the smallest possible segment size for our disk. Thus, the read-ahead size we simulate is always equal to this segment size. This conventional system is used as a basis for comparison.

We have validated our simulator by comparing its results for the conventional system against real executions of two micro-benchmarks, a read-only and a write-only benchmark, on our drive. The micro-benchmarks mimic the disk access patterns of the data-intensive servers we study in this paper by reading/writing small files located randomly on a disk. The simulation results are within 8% and 3% of the real measurements for reads and writes, respectively. These very positive results give us confidence about the accuracy of our simulations.

Note that we simulate disk arrays because each disk delivers very low bandwidth compared to modern server-grade host microprocessors, and *not* because the workloads we study require a significant amount of storage space.

6.2 Evaluation with Synthetic Workloads

In this subsection, we seek to observe the main trends exhibited by our techniques and to compare them against other approaches in a simple and controlled manner. To expose their behavior, we use a synthetic trace containing 10000 requests. Each request reads a set of sequential blocks, representing accesses to complete files. All requests are of the same size, but we run simulations for different sizes. The starting block of each request is chosen according to a Bradford Zipf distribution [18]. The α coefficient of such distribution has a default value of 0.4. The requested blocks are striped across the 8 disks and the size of the striping unit is equal to 128 KBytes. We assume this value to be equal to largest sequential access to avoid fragmentation that could increase the FOR gains. We have also set the number of concurrent I/O data streams to 128. Most of these parameters are varied below.

Since prefetching and coalescing of requests to sequential blocks are important for getting realistic results, we assume perfect operating system prefetching, i.e. the system prefetches all the blocks of each file, and a request coalescing probability of 87%. This value is the average coalescing probability that we have found for our real workloads.

FOR. Figure 3 plots the normalized I/O time for our synthetic trace as a function of file size. Besides the results for the FOR technique, the figure also shows results for a system with blind read-ahead using both segment-based ('Segm') and block-based ('Block') controller cache organizations, and a system with the read-ahead mechanism disabled ('No-RA'). Like FOR and Block, No-RA assumes a block-based organization for the cache. All times are normalized with respect to the conventional system (Segm).

We can see that blind read-ahead performs the same regardless of the organization of the cache. No-RA performs better than blind read-ahead for files smaller than 48 KBytes, but it is very expensive for larger files. No-RA can be improved by an increased probability of coalescing. However, No-RA does not outperform FOR even for an unrealistic coalescing probability of 100%, i.e. with perfect coalescing. Due to its poor results for a wide range of file sizes, we do not consider No-RA further in this paper.

FOR performs at least as well as blind read-ahead and No-RA in all cases. In particular, FOR can reduce the I/O time with respect to blind read-ahead by 40% for 16-KByte files, a common average size of requested files in several types of data-intensive servers, such as those we study in this paper. The reductions in I/O time progressively decrease as

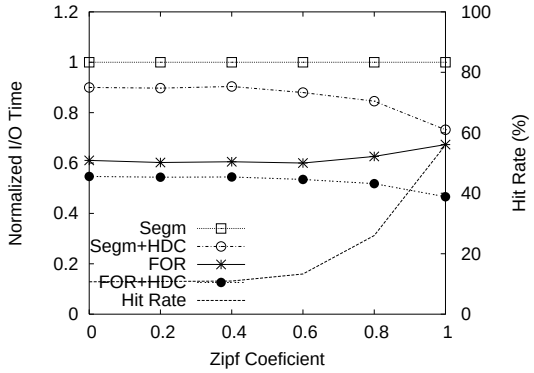

Figure 5: Normalized I/O time as function of access frequency distribution. HDC caches = 2 MB. Writes = 0%.

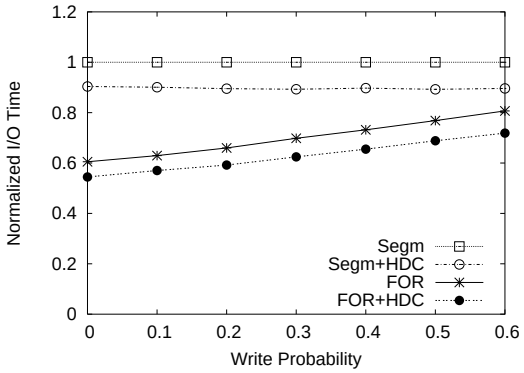

Figure 6: Normalized I/O time as function of percentage of writes. HDC caches = 2 MB. Zipf α = 0.4.

we increase the average file size. These results illustrate FOR's ability to improve performance significantly by reducing disk utilization for small file accesses.

Because FOR frees up cache space that can be used to support more I/O streams, it is also important to understand the impact of the number of simultaneous streams utilized by the server. Figure 4 shows the normalized I/O time for 10000 16-KByte requests, as a function of the number of simultaneous streams. Again, the figure plots results for Segm, Block, and FOR. All times are normalized to the Segm results. Note that the number of segments for Segm is 216 (8 disks $\times$ 27 segments per disk).

We can see that Block exhibits slightly better performance than Segm for the larger numbers of simultaneous streams; the difference reaches 3% for 1024 simultaneous streams. The advantage of Block is that the eviction of one disk block does not imply the eviction of all other blocks of the corresponding stream. However, the number of simultaneous streams has to be extremely large for Block to provide performance benefits. For a number of streams $\leq$ 256, blind read-ahead performs the same regardless of the organization of the cache. In fact, we observe this behavior in all our other comparisons, so we do not present further results for Block in this paper.

The FOR gains start out at 39% for 64 simultaneous streams and slowly increase with an increase in the number of streams. The gains eventually reach 59% for 1024 streams. The reason for this behavior is that, as the number of streams is increased, there is a decreasing chance that a segment or block is used more than once by the blind read-ahead schemes. In contrast, replacements are not an important issue for FOR even with 1024 streams, since 1024 16-KByte files only consume 16 MBytes out of a total 32 MBytes (8 disks $\times$ 4 MBytes) of available cache space. For fewer than 216 streams, FOR improves performance by reducing disk utilization; for larger numbers of streams, FOR improves performance by reducing utilization and increasing cache hit rates. These results show that FOR is able to serve more simultaneous streams without interference among them.

HDC. The main parameters that influence the performance gains achievable by HDC are the size of the HDC cache, the request frequency distribution of disk blocks, and the percentage of block writes. In this subsection, we study the latter two parameters. The size of the HDC cache is studied in the next subsection.

Figure 5 shows the normalized I/O time for our trace of 16-KByte requests, as a function of the frequency distribution of accesses to different blocks. We vary the Zipf coefficient (α) from 0 to 1; larger coefficients mean that accesses are more concentrated on a smaller number of blocks. We assume that 2 MBytes of cache in each disk controller are allocated to HDC. Each HDC cache holds the blocks (belonging to the corresponding disk) that are accessed on disk most frequently. The figure plots I/O time results for Segm and FOR, as well as their combinations with HDC. All times are normalized with respect to the Segm results. The figure also plots HDC cache hit rate results. The hit rates are computed as the number of accesses (reads and writes) that hit in the HDC caches divided by the total number of accesses.

Comparing the Segm and Segm+HDC curves, we can see that the gains provided by HDC are fairly stable around 10% for values of α between 0 (uniform distribution) and 0.6 for our synthetic workload. Values greater than 0.6 improve the HDC gains up to 28%. When we compare the FOR and FOR+HDC curves, HDC can produce gains of up to 31%.

With respect to the HDC hit rates, the figure shows that, as expected, the hit rate increases as we increase the Zipf coefficient, reaching 56% for $\alpha = 1$. These results illustrate the fact that HDC can increase performance even for workloads with uniform access frequency distributions ($\alpha = 0$). With high α, the HDC gains become significant.

So far, we have assumed workloads without disk writes. Figure 6 shows the normalized I/O time as a function of the percentage of writes in the workload. The figure plots results for Segm and FOR, as well as their combinations with HDC. Again, we assume that each HDC cache can hold 2 MBytes

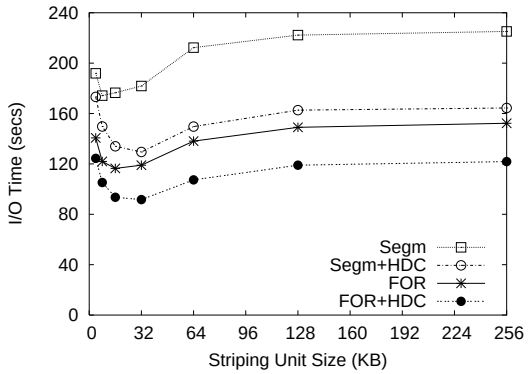

Figure 7: Web server – I/O time as function of striping unit size. HDC caches = 2 MB.

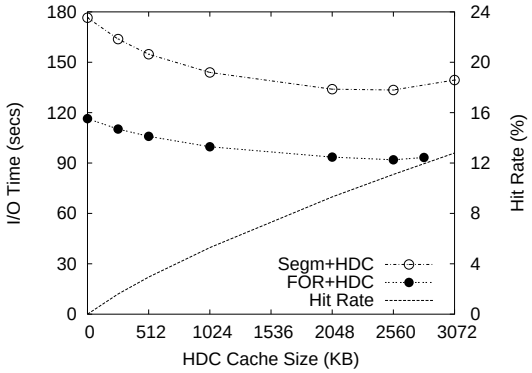

Figure 8: Web server – I/O time as function of HDC memory size. Striping unit = 16 KBytes.

of data. All times are normalized to the Segm results.

We can see that the performance improvements provided by HDC are fairly constant, regardless of the technique it is combined with, for the range of write percentages we consider. It is also interesting to note that the FOR improvements drop from 39 to 19% as we increase the percentage of writes from 0 to 60%. The improvements due to FOR+HDC also drop significantly, from 46 to 28%, in the same range. These performance drops are expected, since FOR is targeted at read operations, which are usually substantially more common than write operations.

Summary. These results clearly demonstrate that FOR and HDC are superior to comparable techniques for a wide range of parameters. The FOR gains increase as we decrease file sizes and/or increase the number of simultaneous I/O streams, whereas the HDC gains increase as the requests are shifted toward a small number of blocks. The gains of FOR decrease as the percentage of writes increases, but significant improvements still remain for write-mostly workloads.

6.3 Evaluation with Real Workloads

Having isolated the performance trends associated with our techniques, our goal now is to quantify their impact in practice. This subsection analyzes the impact of our two proposed techniques on the performance of real servers and their workloads. In particular, we present results for a Web server, a Web proxy server, and a file server. These servers have been implemented by ourselves and include all the functionality required by their workloads.

In order to generate the traces for our simulator, we have used a Linux-based machine with a 1.9 GHz Pentium-4 processor and 512 MBytes of memory to run our servers. The machine stores all files required by our workloads on our IBM Ultrastar 35Z15 SCSI disk. In all cases, the server workloads are generated by a pool of client PCs connected to the server machine by a Gigabit Ethernet switch. The server workloads are determined by real traces that we describe below, along with the results for each server.

We consider the entire cache hierarchy in our simulations. We have instrumented the Linux 2.4.18 kernel to log all disk accesses as our server workloads are executed on our real server machine. Thus, the disk accesses are for data that was not found in the application server's cache (if there is one) and in the file system buffer cache. Two accesses to consecutive blocks are coalesced if the difference in time between the accesses is less than 2 msecs. The logs are later used as input to our simulator. The logs are replayed in the simulator as fast as possible to determine the maximum throughput achievable by each system.

Web Server. Our Web server evaluation uses PRESS [8] and a Web request trace from our department at Rutgers to collect the logs of disk accesses. PRESS is an efficient event-driven Web server that uses 16 helper threads for disk access. This means that the maximum number of concurrent I/O streams in our Web server is 16. The Rutgers trace has 1.7M requests directed to $\sim$70K different files. The files occupy 1.7 GBytes of disk space, and the requested files have an average size of 21.5 KBytes. The most frequently requested disk block is accessed 88 times. Writes are 2% of the accesses in our Web server log.

Figure 7 shows the absolute I/O time (for the entire workload) of our Web server as a function of the size of the striping unit. The figure plots results for Segm and FOR, as well as their combinations with HDC. Again, we assume 2 MBytes of controller memory for each HDC cache.

We can make several interesting observations from this figure. First, we find that the best striping unit size for this server is between 16 and 32 KBytes. Larger striping units lead to disk load unbalances, whereas smaller striping units produce too many individual accesses. Second, we find that when blind read-ahead is combined with HDC, I/O times are reduced by 10, 30, and 27% for 4, 64, and 256-KByte striping units, respectively. In all these configurations, the HDC cache hit rate is 9%. This means that the read-ahead cache is so underutilized in Segm that it is better to dedicate some

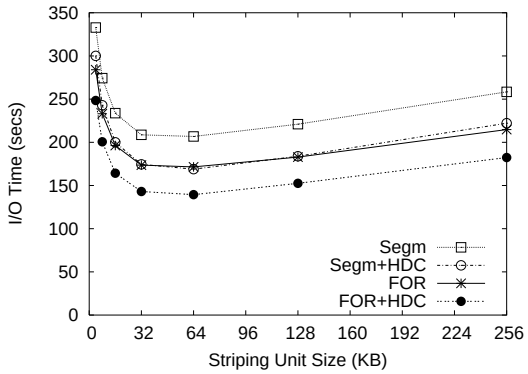

Figure 9: Proxy server – I/O time as function of striping unit size. HDC caches = 2 MB.

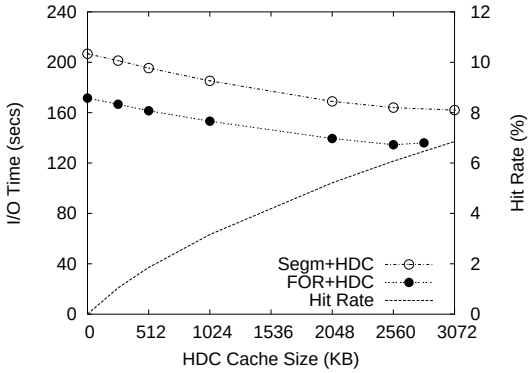

Figure 10: Proxy server – I/O time as function of HDC memory size. Striping unit = 64 KBytes.

of the read-ahead cache to HDC, even if the HDC hit rate is low. Third, FOR reduces I/O times between 27 and 34% compared to Segm. When we combine FOR and HDC, the I/O times are reduced by up to 47%, again in comparison to blind read-ahead. These I/O time reductions are consistent and stable for all the configurations. More importantly, these reductions translate directly to improvements in server throughput, as all the servers we consider are I/O-bound.

Finally, it also interesting to observe that HDC improves performance less when it is combined with FOR than when it is combined with blind read-ahead. The reason is that controller cache misses are more expensive under a blind read-ahead approach; eliminating even a small fraction of these misses is very useful.

For us to understand the effect of the size of the HDC caches, figure 8 plots the absolute I/O time (for the entire workload) of our Web server as a function of the memory allocated to HDC. The figure only shows results for the systems that use HDC and assumes a 16-KByte striping unit. Note that we do consider the space overhead associated with the FOR bitmap; that is the reason why the FOR+HDC curve does not touch the right side of the graph.

We can see that the HDC improvements increase with cache size, until a point (2.5 MBytes) at which the read-ahead cache starts becoming too small. At this point, the HDC gains (compared to not using HDC caches, i.e. the leftmost point of the graph) are around 24% for Segm+HDC and 21% for FOR+HDC.

Figure 8 also shows the overall HDC hit rate. For the Web server and its workload, the HDC hit rate can be as high as 13% for 3-MByte HDC caches. However, as we just discussed, improvements to the HDC hit rate are not always translated into performance improvements.

Proxy Server. Our proxy server evaluation uses a trace from the Hummingbird project at AT&T. The trace has approximately 750K requests for 440K different URLs with a 43% proxy miss rate. Requested files have an average size of 8.3 KBytes and occupy 4.9 GBytes of disk space. The proxy server was configured to satisfy up to 128 simultaneous requests, so there can be no more than 128 concurrent I/O streams. The logged disk accesses have 19% of writes and the most frequently requested block has 78 references.

Figure 9 plots the absolute I/O time of our proxy server as a function of the striping unit size. The figure shows that the performance gains that we achieve for the proxy server are smaller than those for the Web server, due to the larger data footprint and the larger fraction of disk writes in our proxy workload. When HDC is combined with Segm, it reduces I/O times by 10, 18, and 14% for 4, 64, and 256-KByte striping units, respectively. In contrast, FOR reduces I/O times between 15 and 17%, again in comparison to Segm. When FOR and HDC are combined, I/O time reductions reach 33%.

We can also see that the best striping unit size for our proxy server and workload is between 32 and 64 KBytes. This ideal unit size is larger than the one we found for the Web server. This increase is due to the smaller average requested file size and larger footprint of the proxy workload. These two characteristics imply a better load balancing of the accesses across the different disks, mitigating the problem of larger striping units.

Figure 10 shows the I/O times and HDC cache hit rates of the proxy server as a function of the amount of memory allocated to HDC. This figure is similar to that of the Web server. The HDC gains (compared to not using HDC caches) are around 22% for both Segm+HDC and FOR+HDC when 2.5 MBytes are allocated to HDC.

File server. Our file server evaluation uses a trace from HP labs. The trace has 9.5M requests for approximately 30K different files. Each disk request only accesses a fraction of the entire file. The average access size is 3.1 KBytes and 34% of the requests are writes. However, the logged disk accesses only involve 20% of writes, since the file system buffer cache merges writes for the same disk block. The overall footprint of the accessed files is 16 GBytes and the most frequently requested block has 90 accesses. The file server was config-

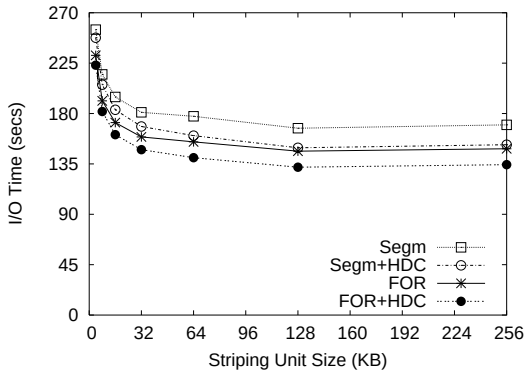

Figure 11: File server – I/O time as function of striping unit size. HDC caches = 2 MB.

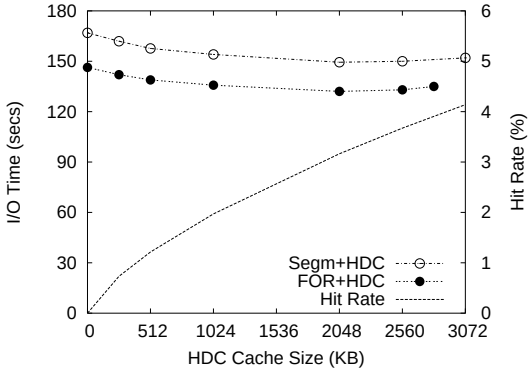

Figure 12: File server – I/O time as function of HDC memory size. Striping unit = 128 KBytes.

Server	Best Striping Unit	FOR	Segm+HDC	FOR+HDC
Web	16 KBytes	34%	24%	47%
Proxy	64 KBytes	17%	18%	33%
File	128 KBytes	12%	10%	21%

Table 2: Disk throughput improvements produced by our techniques.

ured to satisfy up to 128 simultaneous requests.

Figure 11 plots the absolute I/O time of our file server as a function of the striping unit size. The figure shows that the file server has almost the same behavior as the proxy server. FOR reduces the I/O times by up to 12% compared to Segm. These FOR performance gains are a little lower than those for the proxy server, since the file server does not necessarily access entire files. When combined with Segm, HDC reduces I/O times by 10% for striping units larger than 32 KBytes. When combined with FOR, HDC reduces I/O times by up to 21%. The best striping unit size for this server and workload is 128 KBytes. Again, the reason for such large ideal striping unit is the small average requested file size and large footprint.

Finally, figure 12 shows the I/O times and HDC cache hit rates of the file server as a function of the amount of memory allocated to HDC. This figure is also very similar to that of the proxy server. The maximum performance gains achievable by HDC are around 10% for both Segm+HDC and FOR+HDC, in comparison to a system with no HDC caches. The HDC cache hit rates are very low across the board, reaching 4% for a 3-MByte cache.

Summary. Our results with real workloads quantified the benefits of our techniques in practice. We find that FOR and HDC can induce consistent and significant performance gains. The combination of the two techniques achieves the best overall performance for a wide range of parameters, types of servers, and workloads. Table 2 summarizes the disk throughput improvements achieved by our techniques (in comparison to a conventional disk controller) for the different servers and their best striping unit sizes.

7 Conclusions

In this paper we showed that the cache management done by current disk controllers is inappropriate for servers. We proposed two novel techniques that improve this management by taking advantage of the processing and memory capacities of modern disk drives. Combining the two techniques achieves disk throughput that is at least as high as that of conventional controllers. More importantly, for a wide range of parameters and real server workloads, the combination of our techniques can achieve significant and consistent increases in server throughput. For the best striping unit sizes of each server/workload combination, the combination of our techniques improves disk throughput by 47%, 33%, and 21%, in comparison to a conventional disk controller. Given these results, our techniques should prove extremely useful, especially since it will be a long while before new storage technologies (e.g. [7]) become widely available.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, Cambridge, MA, October 1998.

[2] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):95–107, 1999.

[3] S. Akyürek and K. Salem. Adaptive Block Rearrangement Under UNIX. *Software – Practice and Experience*, 27(1):1–23, January 1997.

[4] R. Barve, E. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter. Modeling and Optimizing I/O Throughput of Multiple Disks on a Bus. In *Proceedings of ACM SIGMETRICS Conference*, pages 83–92, Atlanta, GA, May 1999.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM 99*, pages 126–134, New York, NY, March 1999.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[7] L. R. Carley, G. R. Ganger, and D. F. Nagle. MEMS-Based Integrated-Circuit Mass-Storage Systems. *Communications of the ACM*, 43(11):73–80, November 2000.

[8] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming*, pages 113–122, Snowbird, UT, June 2001.

[9] E. V. Carrera, M. Rangarajan, R. Bianchini, and L. Iftode. Impact of Next-Generation I/O Architectures on the Design and Performance of Network Servers. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks*, pages 27–34, Cambridge, MA, February 2002.

[10] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2), June 1994.

[11] G. R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, Department of Computer Science, University of Michigan, Ann Arbor, MI, June 1995.

[12] Gregory R. Ganger and M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–17, January 1997.

[13] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 169–178, Philadelphia, PA, May 1996.

[14] IBM. *IBM Ultrastar 36Z15 Hard Disk Drive*, July 2001. http://www.storage.ibm.com/.

[15] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the 18th Symposium on Operating Systems Principles*, October 2001.

[16] R. Karedla, J. Love, and B. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, 27(3):38–46, March 1994.

[17] 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 *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[18] S. Majumdar. Locality and File Referencing Behaviour: Principles and Applications. Technical Report TR 84-14, Department of Computer Science, University of Saskatchewan, August 1984.

[19] E. P. Markatos. Main Memory Caching of Web Documents. In *Proceedings of 5th International World Wide Web Conference*, pages 893–906, Paris, France, May 1996.

[20] K. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

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

[22] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Data Bases, VLDB*, pages 62–73, August 1998.

[23] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(2):26–52, February 1992.

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

[25] Seagate. *Product Manual: Cheetah X15 36LP Disc Drive*, June 2001. http://www.seagate.com/.

[26] E. Shriver. *Performance modeling for realistic storage devices*. PhD thesis, Department of Computer Science, New York University, New York, NY, May 1997.

[27] E. Shriver, C. Small, and K. A. Smith. Why Does File System Prefetch Work? In *Proceedings of the USENIX Annual Technical Conference*, pages 71–83, Monterrey, CA, June 1999.

[28] H. Simitci and D. A. Reed. Adaptive disk striping for parallel input/output. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 88–102, San Diego, CA, 1999. IEEE Computer Society Press.

[29] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 264–276, San Jose, CA, October 2002.

[30] V. Soloviev. MPL-Adaptive Algorithms for Multisegmented Disk Caches. Technical Report NDSU-CSOR-TR-9407, Department of Computer Science, North Dakota State University, June 1994.

[31] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, Durham, NC, January 1994.

[32] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log Based File Systems for a Programmable Disk. In *Proceedings of the 3th Symposium on Operating Systems Design and Implementation*, pages 29–44, New Orleans, LA, February 1999.

[33] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the ACM SIGMETRICS Conference*, pages 241–251, Nashville, TN, May 1994.

[34] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[35] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–104, June 2001.