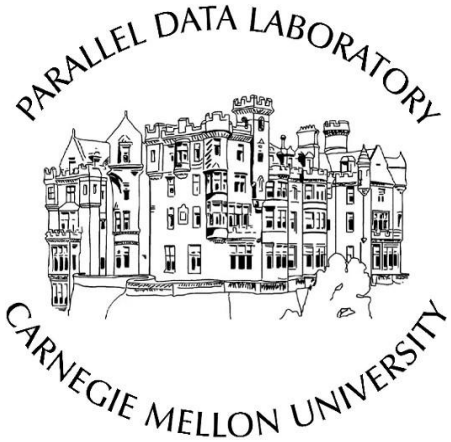

Argon: performance insulation for shared storage servers

Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, Gregory R. Ganger

CMU-PDL-06-106
May 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Services that share a storage system should realize the same efficiency, within their share of time, as when they have it to themselves. This technical report describes mechanisms for mitigating the inefficiency arising from inter-service disk and cache interference in traditional systems and their realization in Ursa Minor's storage server, Argon. Argon uses multi-MB prefetching and write-back to insulate sequential stream efficiency from the disk seeks introduced by competing workloads. It explicitly partitions the cache capacity among services to insulate the hit rate each enjoys from the access patterns of others. Experiments show that, combined, these mechanisms allow Argon to provide to each client a configurable fraction (e.g., 0.9) of its standalone efficiency. With fair-share scheduling, each of n clients approaches the ideal of $1/n$ *of its standalone throughput.*

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Symantec) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Matthew Wachs is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.

Keywords: performance isolation, quality of service, shared storage, performance

1 Introduction

Aggregating services onto shared infrastructures, rather than using separate physical resources for each, is a long-standing approach to mitigating administration costs. Doing so reduces the number of distinct systems that must be managed, allows excess resources to be shared among services that are bursty, and so on. Combined with virtualization, such aggregation strengthens notions such as hosting outsourced services and utility computing.

When multiple services use the same server, each will obviously get only a fraction of the server's resources and, if continuously busy, achieve a fraction of its peak throughput. The hope, however, is that each service is able to utilize its fraction of resources with the same efficiency as when it runs alone; that is, that there is no significant inefficiency-causing interference. For resources like CPU and network, time-sharing creates only minor interference concerns. For the two primary storage system resources—disk head time and cache space—this is not the case.

Disks involve physical motion in servicing requests, and moving the disk head from one region to another is very time-consuming. The worst-case scenario is when two sequential access patterns become interleaved such that the disk head bounces between two regions of the disk; performance goes from streaming disk bandwidth to being bound by positioning times. Likewise, cache misses are much less efficient than cache hits. Yet, it is easy for one data-intensive service to dominate the cache with a large footprint, significantly reducing the hit rates of others. The consequences of inefficient sharing of resources are significant performance degradation and lack of performance predictability (e.g., to predict the performance of a sequential workload, one must analyze all possible ways in which other workloads may interfere with it). Interference concerns drive many administrators to statically partition storage infrastructures among services.

This technical report describes mechanisms that together mitigate these interference issues, insulating[1] services that share a system from one another's presence. First, detecting sequential streams and using sufficiently large prefetching/write-back ranges can amortize positioning costs to achieve a configured fraction of streaming bandwidth. Second, explicit cache partitioning can prevent any one service from squeezing out others. Cache partitioning can be even more effective when the space allocated to a given service is reduced to only the amount it needs to achieve its best efficiency, if the service does not need a large amount of cache space to do so; for example, a service that streams large files and exhibits no reuse hits only needs enough to buffer its prefetched data. In addition to insulation, appropriate disk time scheduling is required if fairness (or weighted shares) or performance targets are desired, while still exploiting superior algorithms implemented within disk firmware.

The Argon storage server includes an implementation of these mechanisms together with policies for managing them. For example, Argon requires policies for detecting sequential streams and determining how much cache to use for each workload. The sequential access size is dictated by the efficiency desired and the re-positioning time, which can be measured.

Experiments with both Linux and pre-insulation Argon confirm the significant efficiency losses that can arise from inter-workload interference. With its insulation mechanisms enabled, measurements show that Argon mitigates these losses, often to a configurable percentage (e.g., 90%) of unshared efficiency. There are workload combinations that cannot be well-insulated, such as two workloads that need the entire cache capacity to perform well, but they are relatively pathological and easy to identify from their behavior.

This technical report makes three main contributions. First, it identifies, explains, and demonstrates experimentally the disk and cache interference issues that arise in shared storage. Second, it describes mechanisms that collectively mitigate them. Although each mechanism is known, we believe that their

[1] We use the term "insulate," rather than "isolate," because a service's performance will obviously depend largely on the fraction of resources it receives and, thus, on the presence of other services. But, ideally, its efficiency will not.

application to performance insulation and their inter-relationships have not been explored. Third, it demonstrates experimentally their effectiveness in providing performance insulation for shared storage.

The remainder of this report is organized as follows. Section 2 motivates the need for storage performance insulation and the challenges faced. Section 3 discusses three complementary mechanisms for doing so. Section 4 describes the implementation of these mechanisms. Section 5 evaluates Argon's efficacy in insulating performance. Section 6 discusses related work. Section 7 summarizes the report.

2 Shared storage and interference

Administration costs push for using shared storage infrastructures to support multiple activities/services, rather than having one for each. This section expands on why and describes the interference issues that arise when doing so. The next section discusses mechanisms for mitigating them.

2.1 Why shared storage?

Many IT organizations support multiple activities/services, such as financial databases, software development, and email. Although many maintain distinct storage infrastructures for each, using a single shared infrastructure can be much more efficient. Not only does it reduce the number of distinct systems that must be supported, it simplifies several aspects of administration. For example, a given amount of excess resources can be easily made available for growth or bursts in any one of the services, rather than having to be partitioned statically among their separate infrastructures (and then moved as needed by administrators). One service's bursts can use resources from others that are not currently operating at peak load, smoothing out burstiness across the shared infrastructure. Similarly, on-line spare components can also be shared rather than partitioned, reducing the speed with which replacements must be deployed to avoid possible outages.

2.2 Interference in shared storage

When services share an infrastructure, they naturally will each receive only a fraction of its resources. For non-storage resources like CPU time and network bandwidth, well-established resource management mechanisms can support time-sharing with minimal inefficiency from interference and context switching [5, 26]. For the two primary storage system resources, however, this is not the case. Traditional disk head and cache management policies can result in significant efficiency degradations when shared by multiple services. That is, interleaving multiple access patterns can result in much less efficient request processing for each. Such loss of efficiency results in poor performance for each workload and for the overall system—with fair sharing, for example, each of two services should each achieve at least half the performance they experience when not sharing, but efficiency losses can result in much lower performance for both.

Disk head interference: Disk head efficiency can be defined as the fraction of the average request's service time spent actually transferring data to or from the magnetic media. The best case, sequential streaming of data, achieves disk head efficiency of approximately 0.9, falling below 1.0 because no data is transferred when switching from one track to the next [28]. Non-sequential access patterns usually achieve efficiencies well below 0.1, as seek time and rotational latency dominate data transfer time. For example, a disk with an average seek time of 5 ms that rotates at 10,000 RPMs would provide an efficiency of $\approx$0.015 for random 8 KB requests (assuming 400 KB per track). Improved locality (e.g., cutting seek distances in half) might raise this value to $\approx$0.02.

Interleaving the access patterns of multiple services can reduce disk head efficiency dramatically, if doing so breaks up sequential streaming. This happens naturally to a sequential access pattern that shares storage with any other access pattern(s), sequential or otherwise. Almost all sequential patterns arrive one request at a time, leaving the disk scheduler with only other services' requests immediately after completing

one from the sequential pattern. The scheduler's choice for the other services' next access will incur a positioning delay and, more germane to this discussion, so will the next request from the sequential pattern. If this occurs repeatedly, the sequential pattern's disk head efficiency can drop by an order of magnitude or more.

Most systems use prefetching and write-back for sequential patterns. Not only can this serve to hide disk access times from applications, it can be used to convert sequences of small requests into fewer, larger requests. The larger requests amortize positioning delays over more data transfer, increasing disk head efficiency if the sequential pattern is interleaved with other requests. Although this helps, most systems do not prefetch aggressively enough to achieve performance insulation [24, 28] —for example, the 64 KB prefetch size common in many operating systems (e.g., for BSD and Linux) raises efficiency from ≈0.015 to ≈0.11 which is still far below the streaming bandwidth efficiency of ≈0.9. More aggressive use of prefetching and write-back aggregation is one tool used by Argon for performance insulation.

Cache interference: A crucial determinant of storage performance, for some applications, is the cache. Given the scale of mechanical positioning delays, cache hits are several orders of magnitude faster than misses. Also, a cache hit uses no disk head time, reducing disk head interference.

With traditional LRU cache replacement, it is easy for one service's workload to get an unfair share of the cache capacity, preventing others from achieving their appropriate cache hit rates. For a service that touches the disk, its cache footprint corresponds to its bandwidth relative to the total bandwidth delivered. So, a service that sequentially streams data will enjoy a much larger footprint than one with non-sequential requests, whether it exhibits cache hits or not. The result can be a significant reduction in cache hit rate for the second service's reads, and thus much lower efficiency if the workload depends upon the cache for its performance.

In addition to efficiency consequences for reads, unfairness can arise with write-back caching. A write-back cache decouples write requests from the subsequent disk writes. Since its writes go into the cache immediately, it is easy for a service that writes large quantities of data to fill the cache with its dirty blocks. In addition to reducing other services' read hit ratios, this can increase the visible work needed to complete each read—when the cache is full of dirty blocks, data must be written out to create free buffers before the next read or write can be serviced.

3 Mitigating interference

Argon is designed to reduce interference between workloads, allowing sharing without loss of efficiency. In many cases, fairness between workloads or weighted fair sharing between workloads is also desired. To accomplish the goals of insulation and fairness, Argon combines three techniques: aggressive amortization, cache partitioning, and quanta-based scheduling. In this section, we define our metrics and goals, then elaborate on Argon's design.

3.1 Goals and metrics

Argon is intended to insulate the *throughput* of competing workloads, within their share, with a secondary goal of keeping the *latency* for each workload reasonably low. This choice of priorities is made for two reasons: first, throughput is a measure of how much work is being accomplished in a unit of time, and of how efficiently a particular workload is using the disk; maximizing these values is often desirable. In fact, when the disk is being used more efficiently, average latency may be decreased as a secondary effect, even without direct focus. Second, other systems tend to optimize for low latency for each request, sometimes at the expense of throughput; thus, Argon explores a different point in the design space.

Insulation means that efficiency for each workload is maintained even when workloads are combined.

Efficiency, for disk access, is the fraction of time spent actually transferring data to or from the disk media. It is not the same as disk utilization — utilization may always be 100% in a busy system, but depending upon how much time is spent positioning the disk head, efficiency may be much less. For cache access, efficiency refers to the hit rate. In many cases, fairness is another goal that is complimentary to insulation. We define *fairness* to be the condition where each workload is receiving identical amounts of disk time and cache space. In the scope of this paper, we are concerned with providing fair shares of resources to each workload; the ability to provide controllable proportions to workloads is a straightforward enhancement.

The ideal for insulation is that a client's throughput when combined with $n-1$ other clients will be no worse than $1/n^{th}$ of its throughput when running alone, assuming fair sharing. For workloads that do not benefit from the cache, Argon approximates this result (for sufficiently large prefetch/coalescing sizes and scheduling quanta). Because the combination of disk sharing and cache sharing has a non-linear effect on performance [30], however, the bound is lower for workloads with significant hit rates, but can be analytically derived (and does not depend upon the details of the other workloads).

Argon's approach to fairness is to establish fair sharing of each individual resource, such as the disk and cache, and to ensure efficiency is retained for each workload, with the expectation that the overall system performance will be fairly distributed as a direct result. An alternative approach that many other systems employ is to allow administrators or clients to establish explicit performance guarantees or goals on a per-workload basis. To achieve these goals, they tune disk-sharing policies. Implementing support for guarantees in our system is future work and we do not expect it to be markedly different from implementing such support in other systems; our system improves the overall efficiency, an orthogonal goal (however, one that may allow better guarantees to be made).

3.2 Overview of mechanisms

Figure 1 illustrates Argon's high-level architecture. Argon provides fairness by allocating disk time evenly across workloads (quanta-based scheduling) and by providing equal segments of the cache to each client (cache partitioning). Efficiency is retained by ensuring that disk time is allotted to clients in large enough quanta so that the majority of time is spent handling client requests, with comparatively minimal time spent at the beginning of a quantum seeking to the workload's first request. This works to insulate workloads. To ensure quanta are effectively used for sequential reads without requiring a queue of actual client requests long enough to fill the time, Argon performs aggressive prefetching; to ensure that sequential writes efficiently use the quanta, Argon coalesces them aggressively in write-back cache space.

While the individual techniques Argon uses are known, they have not been used in the context of performance insulation. There are four principles we follow when combining these techniques and applying them to this goal. First, no single technique is sufficient to solve all of the obstacles to fairness and efficiency; each technique only solves part of the problem. For instance, prefetching improves the performance of sequential workloads, but does not address unfairness at the cache level. Second, some of the techniques work best when they can assume properties that are guaranteed by other techniques. As an example, both read and write requests require cache space. If there are not enough clean buffers, dirty buffers must first be flushed before a request can proceed. If the dirty buffers belong to a different workload, then some of the first workload's time quantum must be spent performing writes on behalf of the second. Cache partitioning simplifies this situation by ensuring that latent flushes are on behalf of the same workload that triggers them, which makes scheduling easier. Third, a combination of mechanisms is needed to prevent unfairness from being introduced. For example, performing large disk accesses for sequential workloads must not starve random workloads, requiring a scheduler to balance the time spent on both types of workloads. Fourth, some of the techniques must be performed much more aggressively than is the case in current systems in order to effectively provide insulation. Sequential reads, for example, require multi-MB prefetches rather than the 64–256 KB prefetches that operating systems such as Linux perform.

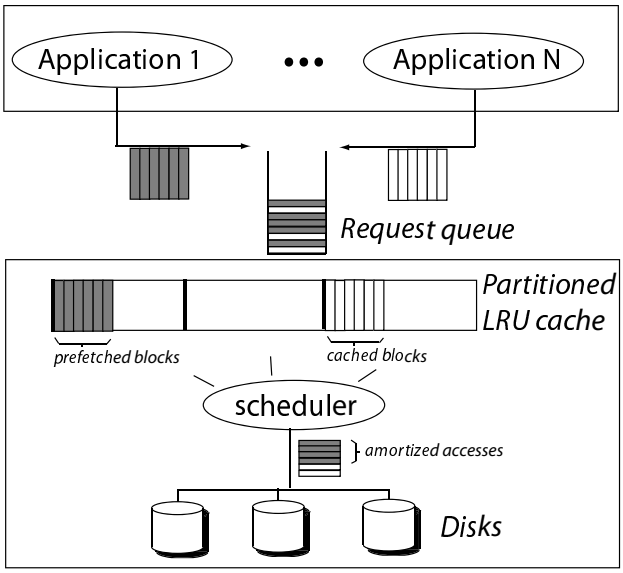

Figure 1: **Argon's high-level architecture.** Argon makes use of cache partitioning, request amortization and quanta-based disk time scheduling.

3.3 Amortization

As described earlier, amortization refers to performing large disk accesses for sequential workloads. Because of the relatively high cost of seek times and rotational latencies, amortization is necessary in order to approach the disk's streaming efficiency when sharing the disk with other workloads. However, as is commonly the case, there is a fundamental tradeoff between efficiency and responsiveness. Performing very large accesses for sequential workloads will achieve the disk's sequential bandwidth, but at the cost of larger response times for at least some requests. Because the disk is being used more efficiently, the average response time may actually improve; but, because blocking will occur as large prefetch or coalesced requests are processed, the maximum response time and the variance in response times may increase significantly.

In contrast to current file systems' tendency to use 64 KB to 256 KB disk accesses, Argon performs sequential accesses megabytes at a time. While this access size is a tunable parameter, our experience, as illustrated in §5.3, has been that such a large access size is required in order for disk efficiency to approach the efficiency of sequential access.

The access size for sequential workloads can be chosen and explained with a simple disk model. The average service time for a disk access not in the vicinity of the current head location can be modeled as:

$$\text{Service time} = T_{seek} + T_{rot}/2 + T_{transfer}$$

where T_{seek} is the average seek time, T_{rot} is the time for one disk rotation, and $T_{transfer}$ is the media transfer time for the data. T_{seek} is the time required to seek to the track holding the starting byte of the data stream. On average, once the disk head arrives at the appropriate track, a request will wait $T_{rot}/2$ before the first byte falls under the head.[2] In contrast to the previous two overhead terms, $T_{transfer}$ represents useful data transfer and depends on the transfer size. In order to achieve disk efficiency of 0.9 (an arbitrarily chosen but reasonable goal), $T_{transfer}$ must be 9 times larger than $T_{seek} + T_{rot}/2$. As shown in Table 1, modern SCSI disks have an average seek time of ≈5 ms, a rotation period of ≈6 ms, and a track size of ≈400 KB. Thus,

[2]Only a small minority of disks have the feature known as *Zero-Latency Access*, which allows them to start reading as soon as the appropriate track is reached and some part of the request is underneath the head (regardless of the position of the first byte) and then reorder the bytes later; this would reduce the $T_{rot}/2$ term.

Disk	Year	RPM	Head Switch	Avg. Seek	Avg. Sectors Per Track	Capacity	Req. Size for 0.9 Efficiency
IBM Ultrastar 18LZX (SCSI)	1999	10000	0.8 ms	5.9 ms	382	18 GB	2.2 MB
Seagate Cheetah X15 (SCSI)	2000	15000	0.8 ms	3.9 ms	386	18 GB	2.5 MB
Maxtor Atlas 10K III (SCSI)	2002	10000	0.6 ms	4.5 ms	686	36 GB	3.4 MB
Seagate Cheetah 10K.7 (SCSI)	2006	10000	0.5 ms	4.7 ms	566	146 GB	4.8 MB
Seagate Barracuda (SATA)	2006	7200	1.0 ms	8.2 ms	1863	250 GB	13 MB

Table 1: **SCSI/SATA disk characteristics.** Positioning times have not dropped significantly over the last 7 years, but disk density and capacity have grown rapidly. This trend calls for more aggressive amortization.

for the Cheetah 10K.7 SCSI disk to achieve a disk efficiency of 0.9 in a sequential access, $T_{transfer}$ must be $9 * (5\ ms + 6\ ms/2) = 72$ ms. Ignoring head switch time, $\approx 72\ ms/T_{rot} = 12$ tracks must be read, which is 4.8 MB. Each number is scaled up on a typical SATA drive.

As disks' data densities improve at a much faster rate than seek times and rotational speeds, aggressive read prefetching and write coalescing grow increasingly important. In particular, the access size of sequential requests needed to insulate increases over time, and the gap between the appropriate access size and the insufficient size of systems such as Linux will grow over time.

So far, we have not distinguished between the two sequential access types: reads and writes (which are respectively amortized through read prefetching and write coalescing). From a disk-efficiency standpoint, it does not matter whether one is performing a large read or write request. Write coalescing is straightforward when a client sequentially writes a file into a write-back cache. The dirty cache blocks are sent in large groups (megabytes) to the disk. Read prefetching is appropriate when a client sequentially reads a file. As long as the client later reads the prefetched data before it is evicted from the cache, aggressive prefetching increases disk efficiency by amortizing the disk positioning costs. Of course, care must be taken to employ a sequential read detector that does not incorrectly predict large sequential access.

3.4 Cache partitioning

Cache partitioning refers to splitting up the buffer cache at the server among multiple services. Specifically, if Argon's buffer cache is split into n segments among services $C_1, ..., C_n$, then C_i's data is only stored in the server cache's i^{th} segment irrespective of the workloads' request pattern. Instead of allowing high cache occupancy for some workloads to arise as an artifact of LRU caching, cache partitioning preserves a fair share of cache space for each workload.

In Argon's default configuration, each workload (random or sequential) has $1/n^{th}$ of the cache space dedicated to it, where n is the number of workloads. In the absence of workload information or prioritization, this fair partitioning policy is the most appropriate. Partitioning ensures a workload's cache hit rate is insulated from other workloads' request patterns, though obviously not from the fact that the cache space is partitioned. §5 shows that insulating the cache is a crucial property in retaining fair performance when some workloads enjoy much higher throughput than others.

Although equally dividing the cache among workloads is adequate from the perspective of fairness, workload-aware cache partitioning schemes can lead to increased system efficiency. For example, consider two workloads running in a system: one consisting of sequential reads that have no cache reuse and another comprising random reads that benefit from the cache because of read reuse. Instead of splitting the cache equally among both workloads, it would be ideal to reserve most of the cache for the random workload and only dedicate enough cache space to the sequential workload to buffer the prefetched blocks (e.g., 4 MB). This cache partitioning scheme would give the random workload a cache hit rate close to the rate it would

achieve if it were the only workload (since there is no inherent contention for the cache in the workloads). Furthermore, the sequential workload would not suffer, since, in steady state, having more cache space than the prefetch access size would not provide any benefit.

3.5 Scheduling

Scheduling in the context of the design and implementation of Argon refers to controlling when each workload's requests are sent to the disk firmware (as opposed to "disk scheduling," such as SPTF, C-SCAN, or elevator scheduling, which reorders requests for efficiency rather than for insulation; disk scheduling occurs in the disk's queue and is implemented in its firmware). Scheduling is necessary to ensure that a workload receives exclusive disk access (as required for amortization), and that disk time is fairly divided among multiple services.

Current file systems do not throttle clients. As a result, an aggressive client (i.e., a client with many requests "in the pipeline") can starve other clients. In particular, some clients with non-sequential access maintain several outstanding requests at the disk to allow more efficient disk scheduling. Others may not be able to do this; for instance, if the location of the next request depends upon data returned from a preceding request (as when traversing an on-disk data structure), concurrency is limited. If a client in this latter category shares a disk with a client that is able to maintain a number of concurrent requests, the client with fewer requests may experience starvation. Argon reduces this problem by giving each client exclusive access to the disk during a scheduling quantum. Active clients' quanta are scheduled in a round-robin manner. In the absence of workload information or prioritization, this round-robin policy is the most appropriate.

4 Implementation

4.1 Overview

We have implemented the Argon storage server to test the efficacy of these performance insulation techniques. Argon is a component in Ursa Minor [2], a cluster-based storage system. Ursa Minor exposes an object-based interface [22] and, so, Argon also exposes an object-based interface. To focus on disk sharing, as opposed to the distributed system aspects of Ursa Minor, we use a single storage server and run benchmarks on the same node.

The techniques of amortization and quanta-based scheduling are implemented on a per-disk basis. Cache partitioning is done on a per-server basis, by default. The design of the system also allows per-disk caching.

Argon is implemented in C++ and runs on Linux and Mac OS X. For portability and ease-of-development, it is implemented entirely in user-space. Argon stores objects in any underlying POSIX filesystem, with each object stored as a file. Argon performs its own caching; the underlying file system cache is disabled (through `open()`'s `O_DIRECT` option in Linux and `fcntl()`'s `F_NOCACHE` option in Mac OS X).

Our servers are battery-backed. This enables Argon to perform write-back caching, by treating all of the memory as NVRAM.

4.2 Distinguishing among workloads

To distinguish among workloads, operations sent to Argon include a client identifier. "Client" refers to a service, not a user or a machine. In Ursa Minor, it is envisioned that clients will use sessions when communicating with a storage server; the identifier is an opaque integer provided by the system to the client on a new session. A client identifier can be shared among multiple nodes; a single node can also use multiple identifiers.

4.3 Amortization

To perform read prefetching, Argon must first detect the sequential access pattern to an object. For every object in the cache, Argon tracks a current *run count*: the number of consecutively read blocks. If a client reads a block that is neither the last read block nor one past that block, then the run count is reset to zero. During a read, if the run count is above a certain threshold (4), Argon reads "run count" number of blocks instead of just the requested one. For example, if a client has read 8 blocks sequentially, then the next client read that goes to disk will prompt Argon to read a total of 8 blocks (thus prefetching 7 blocks). Control returns to the client before the entire prefetch has been read; the rest of the blocks are read in the background. The prefetch size grows until the amount of data reaches the threshold necessary to achieve the desired level of disk efficiency (see §5.3); afterwards, even if the run count increases, the prefetch size remains at this threshold.

Argon's sequentiality detector is simple, and sufficient for demonstrating the performance insulation techniques for sequential accesses and the tradeoffs involved. More complicated and effective detectors could be used in its place.

When Argon is about to flush a dirty block, it checks the cache for any contiguous blocks that are also dirty. In that case, Argon flushes these blocks together to amortize the disk positioning costs. As with prefetching, the write access size is bounded by the size required to achieve the desired level of disk efficiency. Client write operations complete as soon as the block(s) specified by the client are written to the cache; blocks are flushed to disk in the background.

4.4 Cache partitioning

In Argon, each workload client has $1/n^{th}$ of the cache space dedicated to it, where n is the number of workloads. A workload cannot intrude on other workloads' cache space, even if some workload is not using all of its share. The heuristics that Argon uses for detecting sequential streams for prefetching could also be used to differentiate cache share sizes between sequential and random workloads, dedicating most of the space to random workloads (under the assumption that random workloads are most likely to benefit from caching at the server). However, even in random workloads, short sequential runs can occur, and in sequential workloads there may be occasional out-of-order requests, resulting in somewhat fragile workload classifications. If this would cause frequent modifications to the cache space distribution, possibly back and forth as the detector temporarily misclassifies a workload, then it would be costly; thus, a strict $1/n$ policy is a safe, conservative choice. Also, Argon is focused on providing mechanisms for performance insulation; while each mechanism requires appropriate policies, we provide policies sufficient to show the benefits of the mechanisms and leave to future work the task of optimizing the policies.

4.5 Quanta-based scheduling

Scheduling is necessary to ensure fair access to the disk. Argon performs simple round-robin time quantum scheduling, with each workload receiving an equal time quantum. Requests from a particular workload are queued until that workload's time quantum begins. Then, queued requests from that workload are issued, and incoming requests from that workload are passed through to the disk until the workload has submitted what the scheduler has computed to be the maximum number of requests it can issue in the time quantum, or the quantum expires.

The scheduler must estimate how many requests can be performed in the time quantum for a given workload, since the average service times of requests may vary between workloads. This is estimated by initially granting a workload a statically-configured, reasonable number of requests. The scheduler then measures the amount of time these requests have taken to derive an average per-request service time for that workload. The statically-configured scheduling time quantum (chosen based on the desired level of

efficiency — see §5.3) is then divided by the calculated average service time to determine the maximum number of requests that will be allowed from that particular workload during its next quantum.

To provide both hysteresis and adaptability in this process, an exponentially weighted moving average is used on the number of requests for the next quantum. As a result of estimation error and changes in the workload over time, the intended time quanta are not always exactly achieved. Prefetches are treated by the scheduler as a single request that fills the scheduling quantum, and thus the prefetch length and the scheduling quantum should be configured to match.

There also needs to be a scheduling policy for when a time quantum begins but a client has no outstanding requests. On one hand, to achieve strict fair sharing, one might reserve the quantum even for an idle workload, because the client might be about to issue a request [15, 17]. On the other hand, to achieve maximum disk usage, one might skip the client's turn and give the scheduling quantum to the next client which is currently active; if the inactive client later issues a request, it could wait for its next turn or interrupt the current turn. Argon takes a middle approach — a client's scheduling quantum is skipped if the client has been idle in the last three scheduling quanta.

As an alternative to the type of scheduling that Argon performs, requests from multiple random workloads could be issued to the disk concurrently, allowing the disk scheduler to optimize seeks across a larger set of requests. Argon does not currently do this because it is difficult to maintain fairness with this policy when some workloads have many more outstanding requests than others; but it is an appropriate area for improvement because it could allow (in some cases) for efficiency to actually increase, not just stay the same, when workloads are combined.

5 Evaluation

5.1 Experimental setup

The machines hosting both the server and the clients have dual Pentium 4 Xeon 3.0 GHz processors with 2 GB of RAM; the disks are Seagate Barracuda SATA disks (see Table 1 for their characteristics). One disk stores the OS, and the other stores the objects (except in one experiment, which uses two disks to store objects to focus on only the effects of cache sharing). The drives are connected through a 3ware 9550SX controller, which exposes the disks to the OS through a SCSI interface. Both the disks and the controller support command queuing. All computers run the Debian "testing" distribution and use Linux kernel version 2.4.22. Micro-benchmarks running on the same node access the server using the object-based interface. Unless otherwise mentioned, all experiments are run three times, and the average and the standard deviation are reported.

5.2 Microbenchmarks

This section illustrates results obtained using both Linux and Argon. These experiments underscore the need for performance insulation and categorize the benefits that can be expected along the three axes of amortization, cache partitioning, and quanta-based scheduling.

In each experiment, two objects are stored on the server and are accessed by two clients running on the same server (to emphasize the effects of disk sharing, rather than networking effects). Each object is 56 GB in size, a value chosen so that all of the disk traffic will be contained in the highest-performance zone of the disk.[3] The objects are written such that each is fully contiguous on disk. While the system is configured so that no caching of data will occur at the operating system level, the experiments are performed in a way

[3]Disks have different zones, with only one zone experiencing the best streaming performance. To ensure that the effects of performance insulation are not conflated with such disk-level variations, it is necessary to contain experiments within a single zone of the disk.

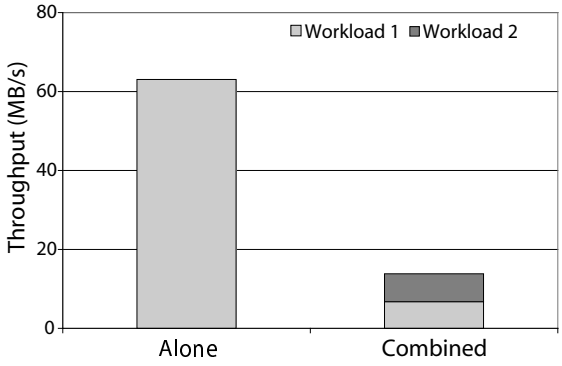

Figure 2: **Performace degradation in Linux due to inadequate amortization.** The standard deviation is at most 0.55 MB/s for these bars.

Figure 3: **Amortization of two sequential workloads in Argon.** The standard deviation is at most 0.31 MB/s for these bars.

that ensures all of the metadata (e.g., inodes and indirect blocks) needed to access the objects is cached, to concentrate solely on the issue of data access. In experiments involving randomly-accessed data, the random block selection process is configured to choose only a uniformly distributed subset of the blocks across the file. The aggregate size of this subset is chosen relative to the cache size to achieve the desired hit rate.[4]

Amortization: Figure 2 shows the performance degradation due to insufficient request amortization in Linux. Two sequential read workloads, each of which receives a throughput of approximately 63 MB/s when running alone, do not utilize the disk efficiently when running together. Instead, each receives a ninth of its unshared performance, and the disk is providing, overall, only one quarter of its streaming throughput. Requests from each of the workloads are 64 KB in size, which is not sufficient to amortize the cost of disk head movement when switching between workloads, even though Linux does perform prefetching.

Figure 3 shows the effect of amortization in Argon. The version of Argon without performance isolation has similar problems to Linux. However, by performing aggressize amortization (in this case, using a prefetch size of 8 MB), sequential workloads better utilize the disk and achieve higher throughput — both workloads receive more than half of their performance when running alone, and the disk is providing nearly its full streaming bandwidth. Average response times also improve by the same rate throughput improved (by $\approx$2 times).

Cache partitioning: Figure 4 shows the performance degradation due to cache interference in Linux. A sequential workload, when run together with a random-access workload with a cache hit rate of 50%, degrades the performance of the random workload. To focus on only the cache partitioning problem, both workloads share the same cache, but go to separate disks. The sequential workload evicts enough blocks that belong to the random-access workload that the performance of the latter decreases to approximately what it would receive if it had no cache hits at all — its performance drops from 3.9 MB/s to 2.3 MB/s, even though only the cache, and not the disk, is being shared. We believe that the small decrease in the sequential workload's performance is an artifact of a system bottleneck.

Figure 5 shows the effect of server cache partitioning in Argon. Two workloads are running, a sequential one W_1 and a random one W_2. The graph only shows the random workload, which receives a 50% cache hit rate when running alone. The first three bars in the figure show random-access workloads with a 0%, 25%, and 50% cache hit rates, respectively, running alone, to establish baseline performance. The bar without performance isolation shows W_2 once combined with the sequential workload. In that case, the performance W_2 receives equals the performance of a random-access workload with a 0% hit rate. By

[4]One alternative would be to vary the file size to control hit rates, but this would also affect the disk seek distance, adding another variable to the experiments.

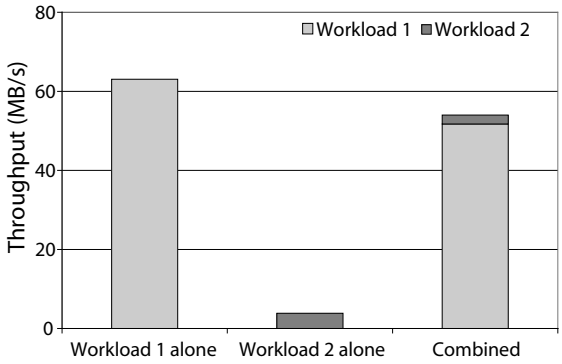

Figure 4: **Performace degradation in Linux due to lack of cache partitioning.** The standard deviation is at most 0.55 MB/s for these bars.

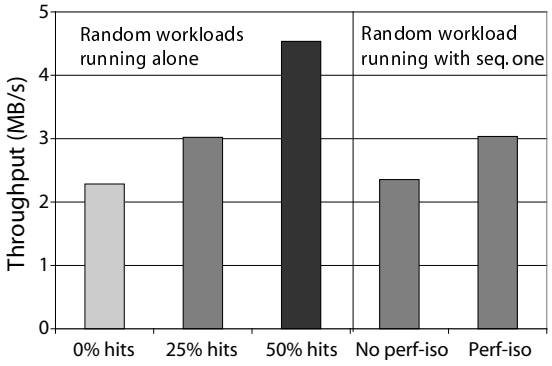

Figure 5: **Cache partitioning for a sequential and random workload.** The standard deviation is at most 0.04 MB/s for these bars.

adding cache partitioning, W_2 gets performance equivalent to that of a random-access workload with a 25% hit rate, which is the performance that fair sharing of cache space through cache partitioning should offer. Average response times also improved by the same rate throughput improved (by $\approx$1.3 times).

Quanta-based scheduling: Figure 6 shows the performance degradation due to unfair scheduling of requests in Linux. Two random-access workloads, one with 27 requests outstanding, and one with just 1 request outstanding, are competing for the disk. When run together, the first workload overwhelms the disk queue and starves the requests from the second workload. Hence, the second workload receives practically no service from the disk at all.

Figure 7 shows the effect of quanta-based disk time scheduling in Argon. The version of Argon with performance isolation disabled had similar problems to Linux. However, by adding quanta-based scheduling with 140 ms time quanta, the two random-access workloads each get a fair share of the disk. Average response times for workload 1 increased by $\approx$2.3 times and average response times for workload 2 decreased by $\approx$37.1 times. Both workloads received slightly less than 50% of their unshared throughput; with a larger quanta size, this loss would be reduced.

5.3 Tunable parameters

This section illustrates the effect of tuning the prefetch size and the scheduling quantum on performance and disk utilization.

Adjusting sequential access size: Figure 8 shows the effect of prefetch size on throughput. Two sequential workloads, each with an access size of 64 KB, were run with performance insulation. The performance each of them receives is similar, hence we only show the throughput of one of them. In isolation, each of these workloads receives approximately 62 MB/s, hence the ideal scenario would be to have them each receive 31 MB/s when run together. This graph shows that the desired throughput is achieved with a prefetch size of at least 32 MB, and that a reasonable throughput (one that achieves an efficiency of roughly 0.9) can be achieved with 8 MB prefetches. We observed that further increases in prefetch size do not improve, or degrade, performance significantly. The spike at 32 KB occurred repeatedly; we believe it is due to an internal disk optimization.

Adjusting scheduling quantum: Figure 9 shows the result of a single-run experiment intended to measure the effect of the scheduling quantum (or the amount of disk time scheduled for one workload before moving on to the other workloads) on throughput. For simplicity, we show quanta measured in

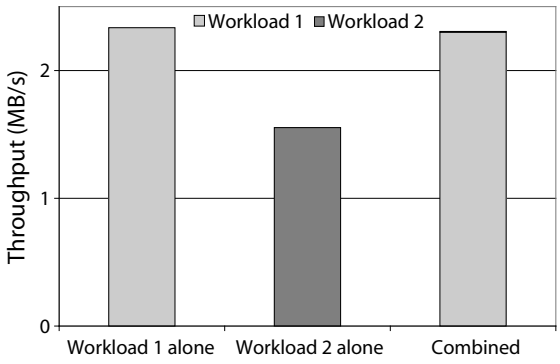

Figure 6: **Performace degradation in Linux due to lack of explicit fair sharing.** The standard deviation is at most 0.01 MB/s for these bars.

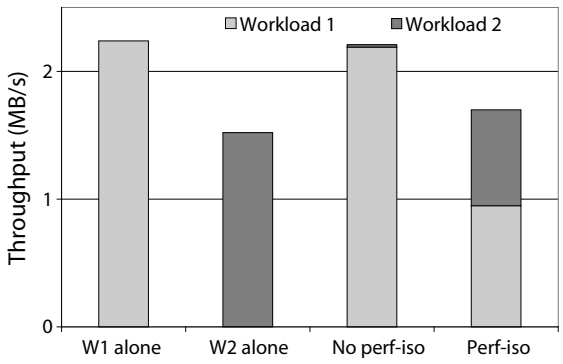

Figure 7: **Quanta-based scheduling support for two random-access workloads.** The standard deviation is at most 0.02 MB/s for these bars.

number of requests for this figure, rather than in terms of time — since different workloads may have different average service times, the scheduler actually schedules in terms of time, not number of requests. Two random workloads are running insulated from each other. We only show the throughput of one of them. In isolation, the workload shown receives approximately 2.23 MB/s, hence the ideal scenario would be to have it receive 1.11 MB/s when run together with the other. This graph shows that the desired throughput is achieved with a scheduling quantum of at least 128 requests, and that a reasonable throughput (one that results in performance within 90% of 1.11 MB/s) can be achieved with one of 32. We observed that further increases in quantum size do not improve, or degrade, performance significantly.

6 Related work

Argon adapts and applies various existing mechanisms to provide performance insulation for shared storage servers. This section discusses previous work on these mechanisms and on similar problems in related domains.

Storage resource management: Most file systems prefetch data for sequentially-accessed files. In addition to hiding some disk access delays from applications, accessing data in larger chunks amortizes seeks over larger data transfers when the sequential access pattern is interleaved with others. A key decision is how much data to prefetch [25]. The popular 64 KB prefetch size was appropriate more than a decade ago [21], but is now insufficient [24, 28]. Similar issues are involved in syncing data from the write-back cache, but without the uncertainty of prefetching. Argon uses much larger prefetch and write-back requests than traditional filesystems, configured to bound inefficiency to a certain percentage (e.g., 90%) of streaming bandwidth.

Schindler et al. [27, 28] show how to obtain and exploit underlying disk characteristics to maximize efficiency for medium-sized disk accesses. In particular, by accessing data in track-sized, track-aligned extents, one can achieve a large fraction of streaming disk bandwidth even when interleaving multiple sequential streams. Such use of disk-specific details, when possible, would allow Argon to mitigate disk head inefficiency with smaller prefetch/write-back requests.

Unlike most file systems, most database systems explicitly manage their caches in order to maximize their effectiveness in the face of interleaved queries [12, 27]. A query optimizer uses its knowledge of query access patterns to allocate for each query just the number of cache pages that it estimates are needed to

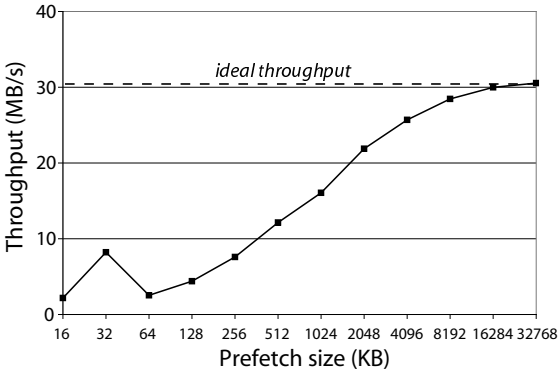

Figure 8: **Effect of prefetch size on throughput.**

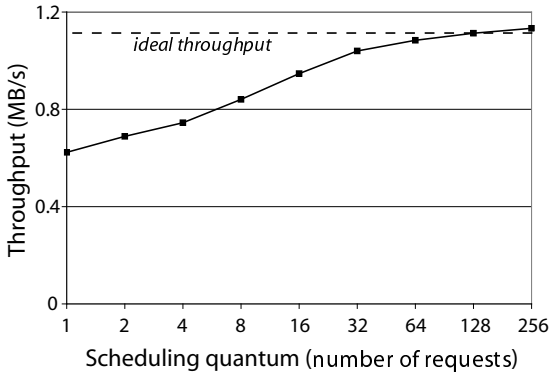

Figure 9: **Effect of scheduling quantum on throughput.**

achieve the best performance for that query. Cao et al. [8, 9] applied these ideas to file systems in their exploration of application-controlled file caching. In other work, the TIP [25] system assumes application-provided hints about future accesses and divides the filesystem cache into three segments that are used for read prefetching, caching hinted blocks for reuse, and caching unhinted blocks for reuse. Argon uses cache partitioning, with a focus on performance insulation rather than overall performance, but works well even without prior knowledge of access patterns, unlike databases and these research efforts.

Resource provisioning in shared infrastructures: Deploying multiple services in a shared infrastructure is a popular concept, being developed and utilized by many. For example, DDSD[35] and Oceano[3] are systems that dynamically assign resources to services as demand fluctuates, based on SLAs and static administrator-set priorities, respectively. Resource assignment is done at the server granularity: at any time, only one service is assigned to any server. Subsequent resource provisioning research ([11, 14, 31, 32]) allows services to share a server, but relies on orthogonal research for assistance with performance insulation.

Most previous performance insulation research has focused on resources other than storage. For example, resource containers [5] and virtual services [26] provide mechanisms for controlling the OS resource usage for CPU and kernel resources.

Several have considered disk time as a resource to be managed, with two high-level approaches. One approach is to use admission control to admit requests into the storage system according to fair-sharing [33] or explicit performance goals [10, 18, 20, 34]. These systems use feedback control to manage the request rates of each service. They do not, however, do anything to insulate the workloads from one another. Argon complements such approaches by mitigating inefficiency from interference. A second approach is time-slicing of disk head time. For example, the Eclipse operating system [6] allocates access to the disk in 1/2-second time intervals. Many real-time file systems [1, 13, 19, 23] use a similar approach. With large time slices, applications will be completely performance insulated with respect to their disk head efficiency, but very high latency can result. Argon goes beyond this approach by attempting to minimize the lengths of time slices required and by adding appropriate cache partitioning and aggressive prefetch/write-back.

Rather than using time-slicing for disk head sharing, one can use a QoS-aware disk scheduler, such as YFQ [7] or Cello [29]. Such schedulers make low-level disk request scheduling decisions that minimize seek times and also maintain per-service throughput balance. Argon would benefit from such a QoS-aware disk scheduler in place of strict time-slicing.

Assuming performance insulation is implemented in each individual server, systems like Cluster Reserves [4] and Sharc [31] provide performance insulation in a distributed setting. Argon focuses on single-server performance insulation.

7 Summary

Storage performance insulation can be achieved when services share a storage server. Traditional disk and cache management policies do a poor job, allowing interference among services' access patterns to significantly reduce efficiency (e.g., by factor of four or more). Argon uses a combination of multi-MB prefetch/write-back and cache partitioning to provide each service with a configurable fraction (e.g., 0.9) of the efficiency it would receive without competition. So, with fair sharing, each of *n* services will approach $1/n$ of its standalone throughput. This increases both efficiency and predictability when services share a storage infrastructure.

Acknolwedgements

We thank Gregg Economou, Michael Stroucken, Chuck Cranor, and Bill Courtright for assistance in configuring hardware, and Craig Soules for his feedback.

References

[1] Robert Abbott and Hector Garcia-Molina. *Scheduling real-time transactions with disk-resident data*. CS–TR–207–89. Department of Computer Science, Princeton University, February 1989.

[2] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA Based Management of a Computing Utility. *IM – IFIP/IEEE International Symposium on Integrated Network Management* (Seattle, WA, 14–18 May 2001), pages 855–868. IFIP/IEEE, 2001.

[4] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *ACM SIGMETRICS Performance Evaluation Review*, **28**(1):90–101. ACM Press, 2000.

[5] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 45–58. ACM, Winter 1998.

[6] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. *USENIX Annual Technical Conference* (New Orleans, LA, 15–19 June 1998), pages 235–246. USENIX Association, 1998.

[7] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. *IEEE International Conference on Multimedia Computing and Systems* (Florence, Italy, 07–11 June 1999), pages 400–405. IEEE, 1999.

[8] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 165–177. Usenix Association, 14–17 November 1994.

[9] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. *Summer USENIX Technical Conference* (Boston, MA), pages 171–182, 6–10 June 1994.

[10] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P. Lee. Performance virtualization for large-scale storage systems. *Symposium on Reliable Distributed Systems* (Florence, Italy, 06–08 October 2003), pages 109–118. IEEE, 2003.

[11] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, AB, Canada, 21–24 September 2001). Published as *Operating Systems Review*, **35**(5):103–116, 2001.

[12] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. *International Conference on Very Large Databases* (Stockholm, Sweden), pages 127–141, 21–23 August 1985.

[13] Steve J. Daigle and Jay K. Strosnider. Disk scheduling for multimedia data streams. *SPIE Conference on High-Speed Networking and Multimedia Computing*, February 1994.

[14] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. *USITS - USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003). USENIX Association, 2003.

[15] Lars Eggert and Joseph D. Touch. Idletime scheduling with preemption intervals. *ACM Symposium on Operating System Principles* (Brighton, United Kingdom, 23–26 October 2005), pages 249–262. ACM Press, 2005.

[16] Steven M. Hand. Self-paging in the Nemesis operating system. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 73–86. ACM, Winter 1998.

[17] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. *ACM Symposium on Operating System Principles* (Chateau Lake Louise, Canada, 21–24 October 2001). Published as *Operating System Review*, **35**(5):117–130. ACM, 2001.

[18] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. *International Workshop on Quality of Service* (Montreal, Canada, 07–09 June 2004), pages 67–74. IEEE, 2004.

[19] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *Computer Journal*, **36**(1):32–42. IEEE, 1993.

[20] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Facade: virtual storage devices with performance guarantees. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 131–144. USENIX Association, 2003.

[21] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *USENIX Annual Technical Conference* (Dallas, TX, January 1991), pages 33–43. USENIX, 1991.

[22] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[23] Anastasio Molano, Kanaka Juvva, and Ragunathan Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings Real-Time Systems Symposium* (San Francisco, CA, 2–5 December 1997), pages 155–165. IEEE Comp. Soc., 1997.

[24] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: an idea whose time has come. *Hot Topics in Operating Systems* (Santa Fe, NM, 12–15 June 2005), 2005.

[25] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[26] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual services: a new abstraction for server consolidation. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 117–130. USENIX Association, 2000.

[27] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. *International Conference on Very Large Databases* (Berlin, Germany, 9–12 September 2003). Morgan Kaufmann Publishing, Inc., 2003.

[28] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[29] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI, June 1998). Published as *Performance Evaluation Review*, **26**(1):44–55, 1998.

[30] Eno Thereska, Michael Abd-El-Malek, Jay J. Wylie, Dushyanth Narayanan, and Gregory R. Ganger. Informed data distribution selection in a self-predicting storage system. *International conference on autonomic computing* (Dublin, Ireland, 12–16 June 2006), 2006.

[31] Bhuvan Urgaonkar and Prashant Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Transactions on Parallel and Distributed Systems*, **15**(1):2–17. IEEE, 01–01 January 2004.

[32] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–12 December 2002), pages 239–254. ACM Press, 2002.

[33] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared memory multiprocessors. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):181–192, November 1998.

[34] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. *RTAS – IEEE Real-Time and Embedded Technology and Applications Symposium* (San Jose, CA, 04–07 April 2006), pages 125–134, 2006.

[35] Huican Zhu, Hong Tang, and Tao Yang. Demand-driven Service Differentiation in Cluster-based Network Servers. *IEEE INFOCOM* (Anchorage, AK, 22–26 April 2001), pages 679–688. IEEE, 2001.