Argon: performance insulation for shared storage servers

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

Abstract

Services that share a storage system should realize the same efficiency, within their share of time, as when they have the system to themselves. The Argon storage server explicitly manages its resources to bound the inefficiency arising from inter-service disk and cache interference in traditional systems. The goal is to provide each service with at least a configured fraction (e.g., 0.9) of the throughput it achieves when it has the storage server to itself, within its share of the server—a service allocated $1/n$th of a server should get nearly $1/n$th (or more) of the throughput it would get alone. Argon uses automatically-configured prefetch/write-back sizes to insulate streaming efficiency from disk seeks introduced by competing workloads. It uses explicit disk time quanta to do the same for non-streaming workloads with internal locality. It partitions the cache among services, based on their observed access patterns, to insulate the hit rate each achieves from the access patterns of others. Experiments show that, combined, these mechanisms and Argon's automatic configuration of each achieve the insulation goal.

1 Introduction

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

When multiple services use the same server, each obviously gets only a fraction of the server's resources and, if continuously busy, achieves a fraction of its peak throughput. But, each service should be able to use its fraction of resources with the same efficiency as when run alone; that is, there should be minimal interference. For resources like the CPU and network, time-sharing creates only minor interference. For the two primary storage system resources — disk head time and cache space — this is not the case.

Disks involve mechanical motion in servicing requests, and moving a disk head from one region to another is slow. The worst-case scenario is when two sequential access patterns become tightly interleaved causing the disk head to bounce between two regions of the disk; performance goes from streaming disk bandwidth to that of a random-access workload. Likewise, cache misses are two orders of magnitude less efficient than cache hits. Without proper cache partitioning, it is easy for one data-intensive service to dominate the cache with a large footprint, significantly reducing the hit rates of other services. Two consequences of disk and cache interference are significant performance degradation and lack of performance predictability. As a result, interference concerns compel many administrators to statically partition storage infrastructures among services.

This paper describes mechanisms that together mitigate these interference issues, insulating[1] services that share a storage system from one another's presence. The goal is to maintain each service's efficiency within a configurable fraction (e.g., 0.9) of the efficiency it achieves when it has the storage server to itself, regardless of what other services share the server. We call this fraction the *R-value*, drawing on the analogy of the thermal resistance measure in building insulation. With an R-value of 1.0, sharing affects the portion of server time dedicated to a service, but not the service's efficiency within that portion. Additionally, insulation increases the predictability of service performance in the face of sharing.

The Argon storage server combines three mechanisms plus automated configuration to achieve the above goal. First, detecting sequential streams and using sufficiently large prefetching/write-back ranges amortizes positioning costs to achieve the configured R-value of streaming bandwidth. Second, explicit cache partitioning prevents any one service from squeezing out others. To maximize the value of available cache space, the space allocated to each service is set to the minimum amount required to achieve the configured R-value of its standalone efficiency. For example, a service that streams large files and exhibits no reuse hits only requires enough cache space to buffer its prefetched data. On-line cache simulation is used to determine the required cache space. Third, disk time quanta are used to separate the disk I/O of services,

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

للاستشارات
www.manaraa.com

eliminating interference that arises from workload mixing. The length of each quantum is determined by Argon to achieve the configured R-value, and average response time is kept low by improving overall server efficiency.

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 and consistently provides each service with at least the configured R-value of unshared efficiency. For example, when configured with an R-value of 0.95 and simultaneously serving OLTP (TPC-C) and decision support (TPC-H Query 3) workloads, Argon's insulation more than doubles performance for both workloads. Workload combinations that cannot be sufficiently insulated, such as two workloads that require the entire cache capacity to perform well, can be identified soon after an unsupportable workload is added.

This paper makes four main contributions. First, it clarifies the importance of insulation in systems that desire efficient and predictable performance for services that share a storage server. Second, it identifies and experimentally demonstrates the disk and cache interference issues that arise in traditional shared storage. Third, it describes mechanisms that collectively mitigate them. Although each mechanism is known, their application to performance insulation, their inter-relationships, and automated configuration to insulation targets have not been previously explored. Fourth, it experimentally demonstrates their effectiveness in providing performance insulation for shared storage. Overall, the paper shows that Argon provides an important and effective foundation for predictable shared storage.

2 Motivation and related work

Administration costs push for using shared storage infrastructures to support multiple activities/services rather than having separate infrastructures. This section expands on the benefits of shared storage, describes the interference issues that arise during sharing, and discusses previous work on relevant mechanisms and problems. The next section discusses Argon's mechanisms for insulating against such interference.

2.1 Why shared storage?

Many IT organizations support multiple activities/services, such as financial databases, software development, and email. Although many organizations maintain distinct storage infrastructures for each activity/service, using a single shared infrastructure can be much more cost-effective. Not only does it reduce the number of distinct systems that must be purchased and supported, it simplifies several aspects of administration. For example, a given amount of excess resources can easily be made available for growth or bursts in any one of the services, rather than having to be partitioned statically among separate infrastructures (and then moved as needed by administrators). One service's bursts can use excess 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 [4, 26]. For the two primary storage system resources, however, this is not the case. Traditional free-for-all disk head and cache management policies can result in significant efficiency degradations when these resources are shared by multiple services. That is, interleaving multiple access patterns can result in considerably less efficient request processing for each access pattern. 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. Further, the service efficiency is determined by the activities of all workloads sharing a server, making performance unpredictable (even if proportional shares are ensured) and complicating dataset assignment tasks.

Disk head interference: Disk head efficiency can be defined as the fraction of the average disk request's service time spent transferring data to or from the magnetic media. The best case, sequential streaming, 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-streaming access patterns can 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-access 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.

المنارة للاستشارات
www.manaraa.com

Interleaving the access patterns of multiple services can reduce disk head efficiency dramatically if doing so breaks up sequential streaming. This often happens 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 of another service's 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., BSD and Linux) raises efficiency from $\approx$0.015 to $\approx$0.11 when sequential workloads share a disk, which is still far below the streaming bandwidth efficiency of $\approx$0.9. More aggressive use of prefetching and write-back aggregation is one tool used by Argon for performance insulation.

Cache interference: For some applications, a crucial determinant of storage performance 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 cache eviction policies, 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. Regardless of which cache eviction policy is used, there will exist certain workloads that fill the cache, due to their locality (recency- or frequency-based) or their request rate. The result can be a significant reduction in the cache hit rate for the other workloads' reads, and thus much lower efficiency if these workloads depend upon the cache for their 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 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' cache hit ratios, this can increase the visible work required to complete each miss — 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.

2.3 Related work

Argon adapts, extends, and applies some 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 complements traditional prefetch/write-back with automated determination of sizes so as to achieve a tunable fraction (e.g., 0.9) of standalone streaming efficiency.

Schindler et al. [27, 28] show how to obtain and exploit underlying disk characteristics to achieve good performance with certain workload mixes. In particular, that work shows that, by accessing data in track-sized track-aligned extents, one can achieve a large fraction of streaming disk bandwidth even when interleaving a sequential workload with other workloads. Such disk-specific mechanisms are orthogonal and could be added to Argon to reduce prefetch/write-back sizes.

Most database systems explicitly manage their caches in order to maximize their effectiveness in the face of interleaved queries [11, 15, 27]. A query optimizer, for example, can use knowledge of query access patterns to allocate for each query just the number of cache pages that it estimates are needed to achieve the best performance for that query [15]. Cao et al. [7, 8] show how these ideas can also be applied 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 partitions that are used for read prefetching, caching hinted blocks for reuse, and caching unhinted blocks for reuse. Argon uses cache partitioning, but with a focus on performance insulation rather than overall performance and without assuming prior knowledge of access patterns. Instead, Argon automatically discovers the

www.manaraa.com

necessary cache partition size for each service based on its access pattern.

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[39] 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 ([10, 13, 34, 35]) allows services to share a server, but relies on orthogonal research for assistance with performance insulation.

Most previous QoS and proportional sharing research has focused on resources other than storage. For example, resource containers [4] and virtual services [26] provide mechanisms for controlling 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 [17, 36] or explicit performance goals [9, 18, 20, 38]. 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, 12, 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 automatically determining the lengths of time slices required and by adding appropriate and automatically configured cache partitioning and prefetch/write-back.

Rather than using time-slicing for disk head sharing, one can use a QoS-aware disk scheduler, such as YFQ [5] or Cello [29]. Such schedulers make low-level disk request scheduling decisions that reduce 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, for workloads whose access patterns would not interfere when combined.

3 Insulating from interference

Argon is designed to reduce interference between workloads, allowing sharing with bounded loss of efficiency. In many cases, fairness or weighted fair sharing between workloads is also desired. To accomplish the complementary goals of insulation and fairness, Argon combines three techniques: aggressive amortization, cache partitioning, and quanta-based scheduling. Argon automatically configures each mechanism to reach the configured fraction of standalone efficiency for each workload. This section describes Argon's goals and mechanisms.

3.1 Goals and metrics

Argon provides both insulation and weighted fair sharing. *Insulation* means that efficiency for each workload is maintained even when other workloads share the server. That is, the I/O throughput a service achieves, within the fraction of server time available to it, should be close to the throughput it achieves when it has the server to itself. Argon allows "how close?" to be specified by a tunable *R-value* parameter, analogous to the R-value of thermal insulation, that determines what fraction of standalone throughput each service should receive. So, if the R-value is set to 0.9, a service that gets 50% of a server's time should achieve at least 0.9 of 50% of the throughput it would achieve if not sharing the server. And, that efficiency should be achieved no matter what other services do within the other 50% of the server's time, providing predictability in addition to performance benefits.

Argon's insulation focus is on efficiency, as defined by throughput. While improving efficiency usually reduces average response times, Argon's use of aggressive amortization and quanta-based scheduling can increase variation and worst-case response times. We believe that this is an appropriate choice (Section 5 quantifies our experiences with response time), but the trade-off between efficiency and response time variation is fundamental and can be manipulated by the R-value choice.

Argon focuses on the two primary storage server resources, disk and cache, in insulating a service's efficiency. It assumes that network bandwidth and CPU time will not be bottleneck resources. Given that assumption, a service's share of server time maps to the share of disk time that it receives. And, within that share of server time, a service's efficiency will be determined by what fraction of its requests are absorbed by the cache and by the disk efficiency of those that are not.

Disk efficiency, as discussed earlier, is the fraction of a request's service time spent actually transferring data to or from the disk media. Note that this is not the same as disk utilization—utilization may always be 100% in a busy system, but efficiency may be high or low depending on how much time is spent positioning the disk head

www.manaraa.com

for each transfer. Idle time does not affect efficiency. So, a service's disk efficiency during its share of disk time should be within the R-value of its efficiency when not sharing the disk; for a given set of requests, disk efficiency determines disk throughput.

Cache efficiency can be viewed as the fraction of requests absorbed by the cache. Absorbed requests—read hits and dirty block overwrites—are handled by the cache without requiring any disk time. Unlike disk efficiency, cache efficiency cannot be maintained for every mix of services, because each service's cache efficiency is a non-linear function of how much cache space (a finite resource) it receives. To address this, a service will be in one of two states: trial and supported. When first added, a service receives spare cache space and is observed to see how much it needs to achieve the appropriate absorption rate (as described in Section 3.4). If the amount needed fits in that spare cache space, that amount is allocated to the service and the service becomes supported. If not, Argon reports the inability to support the service with full efficiency, allowing an administrator or tool to migrate the dataset to a different server, if desired, or leave it to receive best effort efficiency. Thus, new services may not be able to be supported, but supported services will not have necessary cache space taken from them, thereby maintaining their specified R-value of efficiency.

Argon's fairness focus is on providing explicit shares of server time. An alternative approach, employed in some other systems, is to focus on per-service performance guarantees. In storage systems, this is difficult when mixing workloads because different mixes provide very different efficiencies, which will confuse the feedback control algorithms used in such systems. Argon provides a predictable foundation on which such systems could build. Atop Argon, a control system could manipulate the share allocated to a service to change its performance, with much less concern about efficiency fluctuations caused by interactions with other workloads sharing the system. Exploring this approach is an area of future work.

3.2 Overview of mechanisms

Figure 1 illustrates Argon's high-level architecture. Argon provides weighted fair sharing by explicitly allocating disk time and by providing appropriately-sized cache partitions to each workload. Each workload's cache efficiency is insulated by sizing its cache partition to provide the specified R-value of the absorption rate it would get from using the entire cache. Each workload's disk efficiency is insulated 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. To ensure quanta are effectively used for streaming reads without requiring a queue of actual client requests long enough to fill the time, Argon performs aggressive prefetching; to ensure that streaming writes efficiently use the quanta, Argon coalesces them aggressively in write-back cache space.

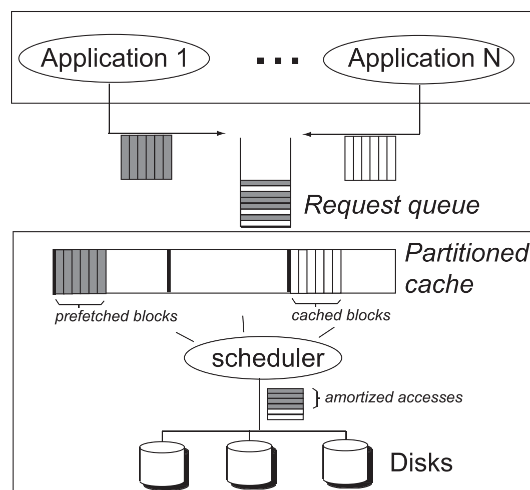

Figure 1: Argon's high-level architecture. Argon makes use of cache partitioning, request amortization, and quanta-based disk time scheduling.

There are four guidelines we follow when combining these mechanisms and applying them to the goals. First, no single mechanism is sufficient to solve all of the obstacles to fairness and efficiency; each mechanism only solves part of the problem. For instance, prefetching improves the performance of streaming workloads, but does not address unfairness at the cache level. Second, some of the mechanisms work best when they can assume properties that are guaranteed by other mechanisms. 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 required to prevent unfairness from being introduced. For example, performing large disk accesses for streaming workloads must not starve non-streaming workloads, requiring a scheduler to balance the time spent on each type of workload. Fourth, each mechanism automatically adapts to ensure sufficient insulation based on observed device and workload characteristics and to avoid misconfiguration. For example, the

www.manaraa.com

ratio between disk transfer rates and positioning times has changed over time, and full streaming efficiency requires multi-MB prefetches on modern disks instead of the 64–256 KB prefetches of OSes such as Linux and FreeBSD.

3.3 Amortization

Amortization refers to performing large disk accesses for streaming 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 trade-off between efficiency and responsiveness. Performing very large accesses for streaming workloads will achieve the disk's streaming bandwidth, but at the cost of larger variance in response time. Because the disk is being used more efficiently, the average response time actually improves, as we show in Section 5. But, because blocking will occur as large prefetch or coalesced requests are processed, the maximum response time and the variance in response times significantly increase. Thus, the prefetch size should only be as large as necessary to achieve the specified R-value.

In contrast to current file systems' tendency to use 64 KB to 256 KB disk accesses, Argon performs sequential accesses MBs at a time. As discussed in Section 4.2, care is taken to employ a sequential read detector that does not incorrectly predict large sequential access. The exact access size is automatically chosen based on disk characteristics and the configured R-value, using 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:

$$S = T_{seek} + T_{rot}/2 + T_{transfer}$$

where S stands for service time, 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, for example, 0.9, $T_{transfer}$ 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, 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 higher on a typical SATA drive.

[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.

As disks' data densities increase at a much faster rate than improvements in seek times and rotational speeds, aggressive read prefetching and write coalescing grow increasingly important. In particular, the access size of sequential requests required for insulation increases over time. Argon automatically determines the appropriate size for each disk to ensure that it is matched to a server's current devices.

Multiple-MB sequential accesses have two implications. Most significantly, the scheduling quantum for sequential workloads must be sufficiently long to permit large sequential accesses. Consequently, although the average request response time will decrease due to overall increased efficiency, the maximum and variance of the response time usually increases. In addition, the storage server must dedicate multiple-MB chunks of the cache space for use as speed-matching buffers. Both of these limitations are inescapeable if a system is to provide near-streaming disk bandwidth in the presence of multiple workloads, due to mechanical disk characteristics.

So far, we have not distinguished between reads and writes (which are amortized through read prefetching and write coalescing, respectively). 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 (MBs) 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.

3.4 Cache partitioning

Cache partitioning refers to explicitly dividing up a server's cache among multiple services. Specifically, if Argon's cache is split into n partitions among services $W_1, ..., W_n$, then W_i's data is only stored in the server's i^{th} cache partition, irrespective of the services' request patterns. Instead of allowing high cache occupancy for some services to arise as an artifact of access patterns and the cache eviction algorithm, cache partitioning preserves a specific fraction of cache space for each service.

المنارة للاستشارات www.manaraa.com

Disk	Year	RPM	Head Switch	Average Seek	Average 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.

It is often not appropriate to simply split the cache into equal-sized partitions. Workloads that depend on achieving a high cache absorption rate may require more than $1/n^{th}$ of the cache space to achieve the R-value of their standalone efficiency in their time quantum. Conversely, large streaming workloads only require a small amount of cache space to buffer prefetched data or dirty write-back data. Therefore, knowledge of the relationship between a workload's performance and its cache size is necessary in order to correctly assign it sufficient cache space to achieve the R-value of its standalone efficiency.

Argon uses a three-step process to discover the required cache partition size for each workload. First, a workload's request pattern is traced; this lets Argon deduce the relationship between a workload's cache space and its I/O absorption rate (i.e., the fraction of requests that do not go to disk). Second, a system model predicts the workload's throughput as a function of the I/O absorption rate. Third, Argon uses the specified R-value to compute the required I/O absorption rate (the relationship calculated in step 2), which is then used to select the required cache partition size (the relationship calculated in step 1).

In the first phase, Argon traces a workload's requests. A cache simulator uses these traces and the server's cache eviction policy to calculate the I/O absorption rate for different cache partition sizes. Figure 2 depicts some example cache profiles for commonly-used benchmarks. The total server cache size is 1024 MB. On one hand, the TPC-C cache profile shows that achieving a similar I/O absorption rate to the one achieved with the total cache requires most of the cache space to be dedicated to TPC-C. On the other hand, TPC-H Query 3 can achieve a similar I/O absorption rate to its standalone value with only a fraction of the full cache space. If both the TPC-C workload and TPC-H Query 3 use the same storage server, Argon will give most of the cache space to the TPC-C workload, yet both workloads will achieve similar I/O absorption rates to the ones they obtained in standalone operation.

In the second phase, Argon uses an analytic model to predict the workload's throughput for a specific I/O absorption rate. In the discussion below, we will only consider reads to avoid formula clutter (the details for writes are similar). Let S_i be the average service time, in seconds, of a read request from service i. S_i is modelled as $p_i * S_i^{BUF} + (1 - p_i) * S_i^{DISK}$. Read requests hit the cache with probability p_i and their service time is the cache access time, S_i^{BUF}. The other read requests miss in the cache with probability $1 - p_i$ and incur a service time S_i^{DISK} (write requests similarly can be overwritten in cache or eventually go to disk). p_i is estimated as a function of the workload's cache size, as described in the first step. S_i^{DISK} is continuously tracked per workload, as described in Section 4.4. The server throughput equals $1/S_i$, assuming no concurrency.

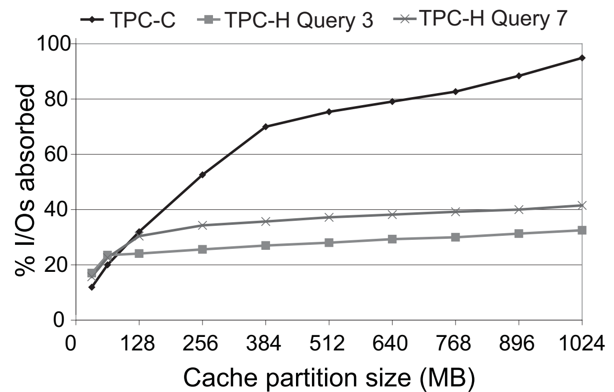

Figure 2: Cache profiles. Different workloads have different working set sizes and access patterns, and hence different cache profiles. The I/O absorption percentage is defined as the fraction of requests that do not go to disk. Such requests include read hits and overwrites of dirty cache blocks. The exact setup for these workloads is described in Section 5.1.

In the final phase, Argon uses the R-value to calculate the required workload throughput when sharing a server as follows:

$$\textit{Throughput required in share of time} = (\textit{Throughput alone}) \cdot (\textit{R-Value})$$

www.manaraa.com

A workload must realize nearly its full standalone throughput in its share of time in order to achieve high efficiency. Its actual throughput calculated over the time both it and other workloads are executing, however, may be much less. As an example, suppose a workload receives 10 MB/s of throughput when running alone, and that an R-value of 0.9 is desired. This formula says that the workload must receive at least 9 MB/s in its share of time. (If it is sharing the disk fairly with one other workload, then its overall throughput will be $9 \cdot 50\% =$ 4.5 MB/s.)

Using the second step's analytic model, Argon calculates the minimum I/O absorption rate required for the workload to achieve *Throughput required* during its share of disk time. Then, the minimum cache partition size necessary to achieve the required I/O absorption rate is looked up using the first step's cache profiles. If it is not possible to meet the R-value because of insufficient free cache space, the administrator (or automated management tool) is notified of the best efficiency it could achieve.

3.5 Quanta-based scheduling

Scheduling in 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 performance rather than for insulation; disk scheduling occurs in the disk's queue and is implemented in its firmware). Scheduling is necessary for three reasons. First, it ensures that a workload receives exclusive disk access, as required for amortization. Second, it ensures that disk time is appropriately divided among workloads. Third, it ensures that the R-value of standalone efficiency for a workload is achieved in its quantum, by ensuring that the quantum is large enough.

There are three phases in a workload's quantum. In the first phase, Argon issues requests that have been queued up waiting for their time slice to begin. If more requests are queued than the scheduler believes will be able to complete in the quantum, only enough to fill the quantum are issued. In the second phase, which only occurs if the queued requests are expected to complete before the quantum is over, the scheduler passes through new requests arriving from the application, if any. The third phase begins once the scheduler has determined that issuing additional requests would cause the workload to exceed its quantum. During this period, the outstanding requests are drained before the next quantum begins.

Inefficiency is introduced by a quanta-based scheduler in two ways. First, if a workload has many outstanding requests, the scheduler may need to throttle the workload and reduce its level of concurrency at the disk in order to ensure it does not exceed its quantum. It is well-known that, for non-streaming workloads, the disk scheduler is most efficient when the disk queue is large. Second, during the third phase, draining a workload's requests also reduces the efficiency of disk head scheduling. In order to automatically select an appropriate quantum size to meet efficiency goals, an analytical lower bound can be established on the efficiency for a given quantum size by modeling these effects for the details (concurrency level and average service time) of the specific workloads in the system. Once a quantum length is established, the number of requests that a particular workload can issue without exceeding its quantum is estimated based on the average service time of its requests, which the scheduler monitors.

Efficiency can also depend upon whether request *mixing* is allowed to happen for non-streaming workloads. Efficiency may be increased by mixing requests from multiple workloads at once, instead of adhering to strict time slices, because this lengthens disk queues. From an insulation standpoint, however, doing so is acceptable only if all clients receive a fair amount of disk time and efficient use of that time. This does not always occur — for instance, some workloads may have many requests "in the pipeline" while others may not. In particular, clients with non-sequential accesses often 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 for that workload is limited. If such workloads are mixed, starvation may occur for the less aggressive workload. Our current design decision has been biased in favor of fairness; we do not allow requests from different workloads to be mixed, instead using strict quanta-based scheduling. This ensures that each client gets exclusive access to the disk during a scheduling quantum, which avoids starvation because active clients' quanta are scheduled in a round-robin manner. In continuing work, we are investigating ways to maintain fairness and insulation while using a mixed-access scheduler.

4 Implementation

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

المنارة للاستشارات
www.manaraa.com

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 cache partitioning.

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.

4.1 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 our cluster-based storage system, 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.

4.2 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; afterwards, even if the run count increases, the prefetch size remains at this threshold.

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 stored in the cache; blocks are flushed to disk in the background (within the corresponding service's quanta).

4.3 Cache partitioning

Recall from Section 3.4 that the cache partitioning algorithm depends on knowledge of the cache profile for a workload. The cache profile provides a relationship between the cache size given to a workload and the expected I/O absorption rate. Argon collects traces of a workload's accesses during the trial phase (when the workload is first added). It then processes those traces using a simulator to predict the absorption rate with hypothetical cache sizes.

The traces collected while a workload is running capture all aspects of its interactions with the cache (cache hits, misses, and prefetches). Such tracing is built in to the storage server, can be triggered on demand (e.g., when workloads change and models need to be updated), and has been shown to incur minimal overheads (5-6%) on foreground workloads in the system [31]. Once sufficient traces for a run are collected, a cache simulator derives the full cache profile for the workload. The simulator does so by replaying the original traces using hypothetical cache sizes and the server's eviction policy. Simulation is used, rather than an analytical model, because cache eviction policies are often complex and system-dependent; we found that they cannot be adequately captured using analytical formulas. We have observed that for cache hits the simulator and real cache manager need similar times to process a request. The simulator is on average three orders of magnitude faster than the real system when handling cache misses (the simulator spends at most 9,500 CPU cycles handling a miss, whereas, on a 3.0 GHz processor, the real system spends the equivalent of about 22,500,000 CPU cycles). The prediction accuracy of the simulator has also been shown to be within 5% [30].

Another implementation issue is dealing with *slack* cache space, the cache space left over after all workloads have taken their minimum share. Currently slack space is distributed evenly among workloads; if a new workload enters the system, the slack space is reclaimed from the other workloads and given to the new workload. This method is very similar to that described by Waldspurger [37] for space reclamation. Other choices are also reasonable, such as assigning the extra space to the workload that would benefit the most, or reserving it for incoming workloads.

للاستشارات المنارة
www.manaraa.com

4.4 Quanta-based scheduling

Scheduling is necessary to ensure fair, efficient access to the disk. Argon performs simple round-robin time quantum scheduling, with each workload receiving a scheduling 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 average service times of requests may vary between workloads. Initially, the scheduler assigns each request the average rotational plus seek time of the disk. The scheduler then measures the amount of time these requests have taken to derive an average per-request service time for that workload. The automatically-configured scheduling time quantum (chosen based on the desired level of efficiency) 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.

Argon does not terminate a quantum until the fixed time length expires. Consequently, workloads with few outstanding requests or with short periods of idle time do not lose the rest of their turn simply because their queue is temporarily empty. Argon does have a policy to deal with situations wherein a time quantum begins but a client has no outstanding requests, however. 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 [14, 16]. On the other hand, to achieve maximum disk utilization, 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 for the last k consecutive scheduling quanta. Argon currently leaves k as a manual configuration option (set to 3 by default). It may be possible to automatically select an optimal value for a given workload through trace analysis.

5 Evaluation

This section evaluates the Argon storage server prototype. First, we use micro-benchmarks to show the performance problems arising from storage server interference, and Argon's effectiveness in mitigating them. Micro-benchmarks allow precise control of the workload access patterns and system load. Second, macro-benchmarks illustrate the real-world efficacy of Argon.

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 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.

Unless otherwise mentioned, all experiments are run three times, and the average is reported. Except where noted, the standard deviation is less than 5% of the average.

5.2 Micro-benchmarks

This section illustrates micro-benchmark 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.

Micro-benchmarks are run on a single server, accessing Argon using the object-based interface. In each experiment, objects are stored on the server and are accessed by 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 that ensures all of the metadata (e.g., inodes and indirect blocks) needed to access the objects is

[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.

المنارة للاستشارات
www.manaraa.com

cached, to concentrate solely on the issue of data access. In experiments involving non-streaming workloads, unless otherwise noted, the block selection process is configured to choose 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 absorption rate.[4]

Amortization: Figure 3(a) shows the performance degradation due to insufficient request amortization in Linux. Two streaming 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. Disk accesses for each of the workloads end up being 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(b) shows the effect of amortization in Argon. The version of Argon without performance insulation has similar problems to Linux. However, by performing aggressive amortization (in this case, using a prefetch size of 8 MB, which corresponds, for the disk being used, to an R-value of 0.9), streaming workloads better utilize the disk and achieve higher throughput — both workloads receive nearly half of their performance when running alone, and the disk is providing nearly its full streaming bandwidth.

Figure 4 shows the CDF (cumulative distribution function) of response time for one of the streaming read workloads in each scenario. (The other workload exhibits a virtually identical distribution because the workloads are identical in this experiment.) The three curves depict response times for the cases of the sequential workloads running alone, together without prefetching, and together with prefetching. The value on the y-axis indicates what fraction of requests experienced, at most, the corresponding response time on the x-axis (which is shown in log scale). For instance, when running alone, approximately 80% of the requests had a response time not exceeding 2 ms. Without performance insulation, each sequential workload not only suffered a loss of throughput, but also an increase in average response times; approximately 85% of the requests waited for 25–29 ms. With prefetching enabled, more than 97% of the requests experienced a response time of less than 1 ms, with many much less.[5] Because some requests must wait in a queue for their workload's time slice to begin, however, a small number (≈2.4%) had response times above

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

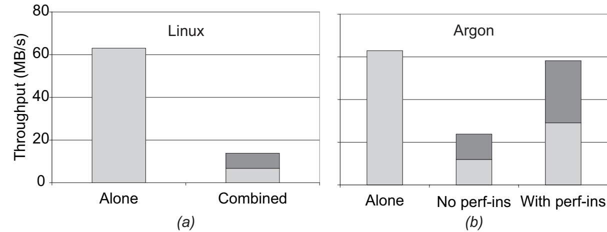

Figure 3: Throughput of two streaming read workloads in Linux and Argon.

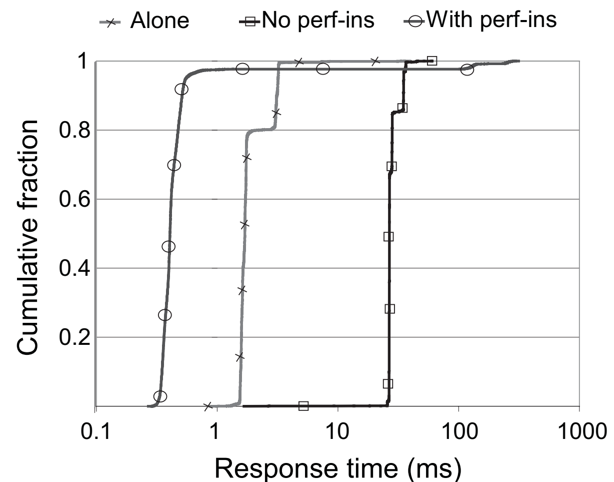

Figure 4: Response time CDFs. When running alone, the average of response times is 2.0 ms and the standard deviation is 1.17 ms. When the two workloads are mixed without performance insulation, the average for each is 28.2 ms and the standard deviation is 3.2 ms. When using performance insulation, the average is 4.5 ms and the standard deviation is 27 ms.

95 ms. This increases the variance in response time, while the mean and median response times decrease.

Cache partitioning: Figure 5(a) shows the performance degradation due to cache interference in Linux. A streaming workload (Workload 1), when run together with a non-streaming workload (Workload 2) with a cache absorption rate of 50%, degrades the performance of the non-streaming workload. To focus on only the cache partitioning problem, both workloads share the same cache, but go to separate disks. Because of its much higher throughput, the streaming workload evicts nearly all of the blocks belonging to the non-streaming workload. This causes the performance of the latter to decrease 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

[5]In fact, the response times for many requests improve beyond the standalone case because no prefetching was being performed in the original version of Argon.

المنارة للاستشارات
www.manaraa.com

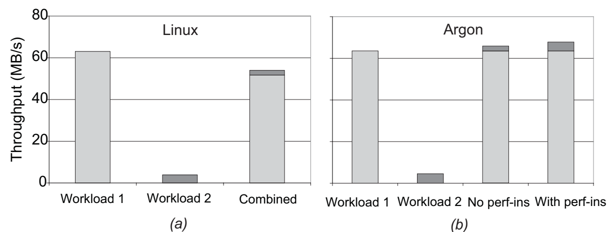

Figure 5: Effects of cache interference in Linux and Argon. The standard deviation is at most 0.55 MB/s for all Linux runs and less than 5% of the average for the Argon runs.

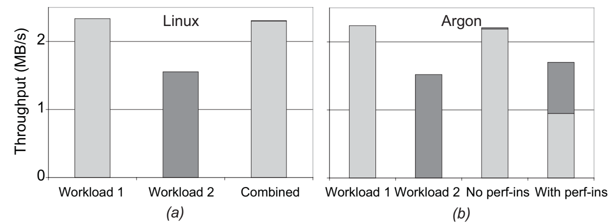

Figure 6: Need for request scheduling in Linux and Argon. The standard deviation is at most 0.01 MB/s for all Linux runs and at most 0.02 MB/s for all Argon runs.

in the streaming workload's performance is an artifact of a system bottleneck.

Figure 5(b) shows the effect when the same workloads run on Argon. The bar without performance insulation shows the non-streaming workload combined with the streaming workload. In that case, the performance the non-streaming workload receives equals the performance of a non-streaming workload with a 0% absorption rate. By adding cache partitioning and using the cache simulator to balance cache allocations (setting the desired R-value to 0.9, the simulator decides to give nearly all of the cache to the non-streaming workload), Workload 2 gets nearly all of its standalone performance.

Quanta-based scheduling: Figure 6(a) shows the performance degradation due to unfair scheduling of requests in Linux. Two non-streaming workloads, one with 27 requests outstanding (Workload 1) and one with just 1 request outstanding (Workload 2), 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 6(b) shows the effect of quanta-based disk time scheduling in Argon. The version of Argon with performance insulation disabled had similar problems to Linux. However, by adding quanta-based scheduling with 140 ms time quanta (which achieves an R-value of 0.9 for the disk and workloads being used), the two non-streaming workloads each get a fair share of the disk. Average response times for Workload 1 increased by ≈2.3 times and average response times for Workload 2 decreased by ≈37.1 times compared to their uninsulated performance. Both workloads received slightly less than 50% of their unshared throughput, exceeding the $R = 0.9$ bound.

Proportional scheduling: Figure 7 shows that the sharing of an Argon server need not be fair; the proportion of performance assigned to different workloads can be adjusted to meet higher-level goals. In the experiment, the same workloads as in Figure 6(b) are shown, but the

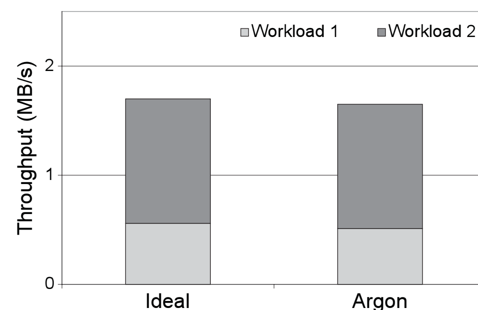

Figure 7: Scheduling support for two random-access workloads. With the same workloads as Figure 6(b), scheduling can be adjusted so that Workload 2 gets 75% of the server time.

requirement is that the workload with one request outstanding (Workload 2) receive 75% of the server time, and the workload with 27 requests outstanding (Workload 1) receive only 25%; quanta sizes are proportionally sized to achieve this. Amortization and cache partitioning can similarly be adapted to use weighted priorities.

Combining sequential and random workloads: Table 2 shows the combination of the amortization and scheduling mechanisms when a streaming workload shares the storage server with a non-streaming workload. To focus on just the amortization and scheduling effects, the non-sequential workload does not hit in cache at all. Without performance insulation, the workloads receive 2.2 MB/s and 0.55 MB/s respectively. With performance insulation they receive 31.5 MB/s and 0.68 MB/s, well within $R = 0.9$ of standalone efficiency, as desired.

Figure 8 shows the CDF of response times for both workloads. The sequential workload, shown in Figure 8(a), exhibits the same behavior shown in Figure 4 and discussed earlier. As before, the variance and maximum of response times increase while the mean and median decrease. The random workload is shown in Figure 8(b). Running alone, it had a range of response times, with none exceeding 26 ms. The 90th percentile was at 13.7 ms. Virtually all values were above 3 ms. Running together with the sequential workload, response times in-

www.manaraa.com

Scenario		Throughput
Alone	Workload 1 (S)	63.5 MB/s
	Workload 2 (R)	1.5 MB/s
Combined (no perf-ins.)	Workload 1 (S)	2.2 MB/s
	Workload 2 (R)	0.55 MB/s
Combined (with perf-ins.)	Workload 1 (S)	31.5 MB/s
	Workload 2 (R)	0.68 MB/s

Table 2: Amortization and scheduling effects in Argon. Performance insulation results in much higher efficiency for both workloads. Standard deviation was less than 6% for all runs.

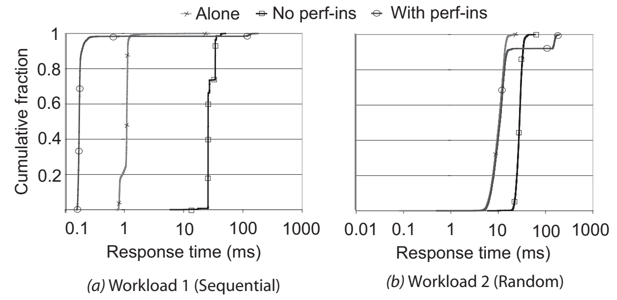

Figure 8: Response time CDFs. The standard deviation of response time for the sequential (a) and random-access workloads (b) when they run alone is 0.316 ms and 2.72 ms respectively. The random-access workload's average response time is 10.3 ms. When the two workloads are mixed without performance insulation, the standard deviation of their response times is 4.16 ms and 4.01 ms respectively. The random-access workload's average response time is 28.2 ms. When using performance insulation the standard deviation is 15.87 ms and 39.3 ms respectively. The random-access workload's average response time is 21.9 ms.

creased; they ranged from 6–60 ms with the 90th percentile at 33 ms. Once aggressive prefetching was enabled for the sequential workload, the bottom 92% of response times for the random workload ranged from 3–24.5 ms. The remainder were above 139 ms, resulting in a lower mean and median, but higher variance.

Scaling number of workloads: Figure 9 shows the combined effect of all three techniques on four workloads sharing a storage server. Workload 1 is the streaming workload used in the previous experiments. Workload 2 is a uniformly random workload with a standalone cache absorption rate of 12.5%. Workload 3 is a microbenchmark that mimics the behavior of TPC-C (with a non-linear cache profile similar to that shown in Figure 2). Workload 4 is a uniformly random workload with zero cache absorption rate. All four workloads get within the desired R-value (0.9) of standalone efficiency when sharing the storage server.

Adjusting sequential access size: Figure 10 shows the effect of prefetch size on throughput. Two streaming 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 R = 0.9 can be achieved with 8 MB prefetches. We observed that further increases in prefetch size do not improve, or degrade, performance significantly.

Adjusting scheduling quantum: Figure 11 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 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 non-streaming 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 R = 0.9 can be achieved with one of 32. We observed that further increases in quantum size do not improve, or degrade, performance significantly.

5.3 Macro-benchmarks

To explore Argon's techniques on more complex workloads, we ran TPC-C (an OLTP workload) and TPC-H (a decision support workload) using the same storage server. The combined workload is representative of realistic scenarios when data mining queries are run on a database while transactions are being executed. The goal of the experiment is to measure the benefit each workload gets from performance insulation when sharing the disk.

Each workload is run on a separate machine, and communicates with the Argon storage server through an NFS server that is physically co-located with Argon and uses its object-based access protocol.

المنارة للاستشارات
www.manaraa.com

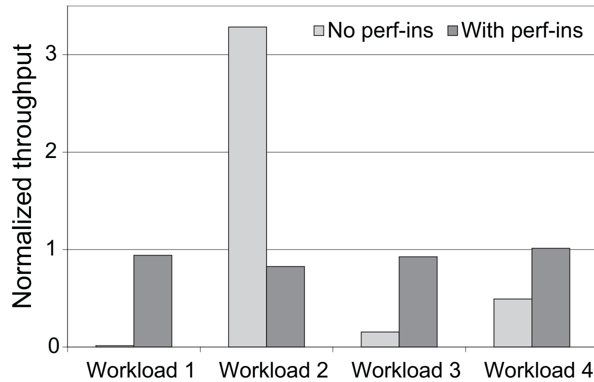

Figure 9: Four workloads sharing a storage server. The normalization is done with respect to the throughput each workload receives when running alone, divided by four.

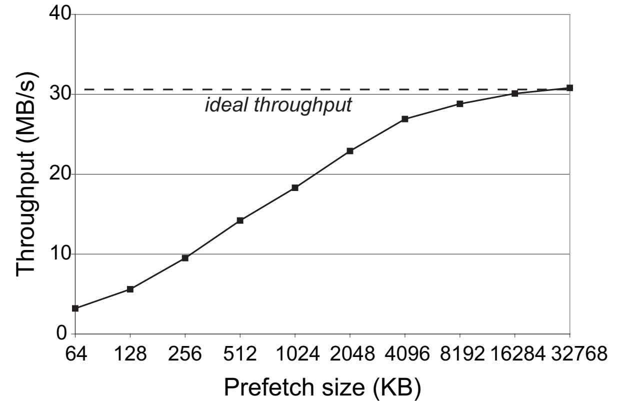

Figure 10: Effect of prefetch size on throughput.

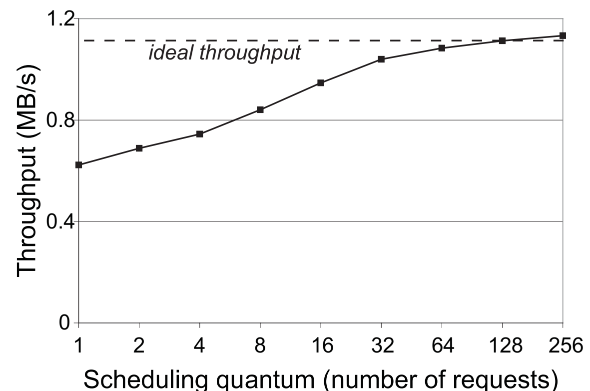

Figure 11: Effect of scheduling quantum on throughput.

TPC-C workload: The TPC-C workload mimics an on-line database performing transaction processing [32]. Transactions invoke 8 KB read-modify-write operations to a small number of records in a 5 GB database. The performance of this workload is reported in transactions per minute (tpm). The cache profile of this workload is shown in Figure 2.

TPC-H workload: TPC-H is a decision-support benchmark [33]. It consists of 22 different queries, and two batch update statements. Each query processes a large portion of the data in streaming fashion in a 1 GB database. The cache profile of two (arbitrarily) chosen queries from this workload are shown in Figure 2.

Figure 12 shows the results: without performance insulation, the throughput of both benchmarks degrades significantly. With an R-value of 0.95, Argon's insulation significantly improves the performance for both workloads. Figure 13 examines the run with TPC-H Query 3 more closely. This figure shows how much each of the three techniques, scheduling (S), amortization (A), and cache partitioning (CP) contribute to maintaining the desired efficiency.

6 Conclusions and future work

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 combines and automatically configures prefetch/write-back, cache partitioning, and quanta-based disk time scheduling to provide each service with a configurable fraction (the R-value; e.g., 0.9) of the efficiency it would receive without competition. So, with fair sharing, each of n services will achieve no worse than R/n of its standalone throughput. This increases both efficiency and predictability when services share a storage server.

Argon provides a strong foundation on which one could build a shared storage utility with performance guarantees. Argon's insulation allows one to reason about the throughput that a service will achieve, within its share, without concern for what other services do within their share. Achieving performance guarantees also requires an admission control algorithm for allocating shares of server resources, which can build on the Argon foundation. In addition, services that cannot be insulated from one another (e.g., because they need the entire cache) or that have stringent latency requirements must be separated. Argon's configuration algorithms can identify the former and predict latency impacts so that the control system can place such services' datasets on distinct storage servers. Our continuing work is exploring the design of such a control system, as well as approaches for handling workload changes over time.

المنارة للاستشارات
www.manaraa.com

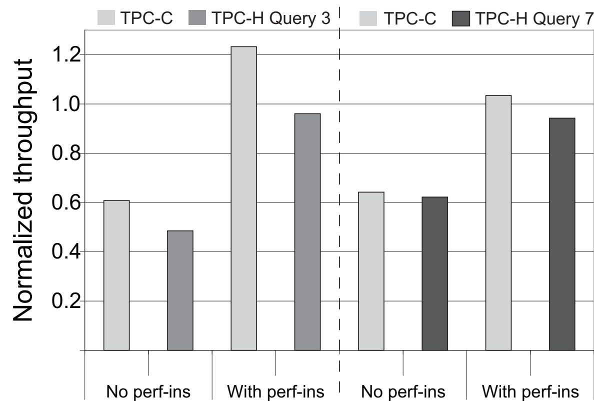

Figure 12: TPC-C and TPC-H running together. TPC-C shown running with TPC-H Query 3 and then with TPC-H Query 7. The normalized throughput with and without performance insulation in shown. The normalization is done with respect to the throughput each workload receives when running alone, divided by two.

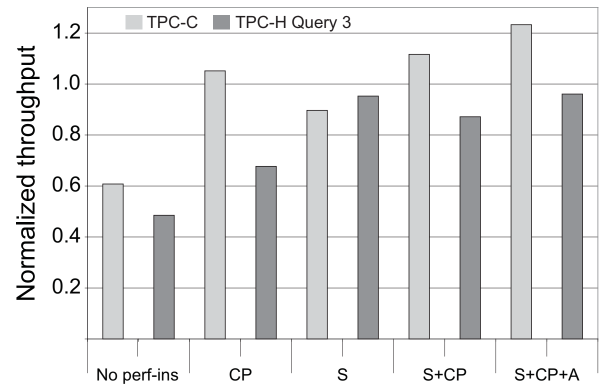

Figure 13: All three mechanisms are needed to achieve performance insulation. The different techniques are examined in combination. “CP” is cache partitioning, “S” is scheduling, “A” is amortization. Argon uses all of them in combination. The normalization is done with respect to the throughput each workload receives when running alone, divided by two.

Acknowledgements

We thank Gregg Economou, Michael Stroucken, Chuck Cranor, and Bill Courtright for assistance in configuring hardware, Craig Soules, John Strunk, Amin Vahdat (our shepherd) and the many anonymous reviewers for their feedback. We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, 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 grants #CNS-0326453 and #CCF-0621499, 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.

References

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

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies, 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, pages 855–868. IFIP/IEEE, 2001.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. Symposium on Operating Systems Design and Implementation, pages 45–58. ACM, Winter 1998.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. IEEE International Conference on Multimedia Computing and Systems, pages 400–405. IEEE, 1999.

[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, pages 235–246. USENIX Association, 1998.

[7] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. Symposium on Operating Systems Design and Implementation, pages 165–177. Usenix Association, 14–17 November 1994.

[8] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. Summer USENIX Technical Conference, pages 171–182, 6–10 June 1994.

[9] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. Symposium on Reliable Distributed Systems, pages 109–118. IEEE, 2003.

[10] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. ACM Symposium on Operating Sys-

المنارة للاستشارات
www.manaraa.com

tem Principles. Published as *Operating Systems Review*, **35**(5):103–116, 2001.

[11] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. International Conference on Very Large Databases, pages 127–141, 21–23 August 1985.

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

[13] 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. USENIX Association, 2003.

[14] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals. ACM Symposium on Operating System Principles, pages 249–262. ACM Press, 2005.

[15] C. Faloutsos, R. Ng, and T. Sellis. Flexible and adaptable buffer management-techniques for database-management systems. *IEEE Transactions on Computers*, **44**(4):546–560, April 1995.

[16] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **35**(5):117–130. ACM, 2001.

[17] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 37–48. ACM Press, 2004.

[18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. International Workshop on Quality of Service, 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] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: virtual storage devices with performance guarantees. Conference on File and Storage Technologies, 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, pages 33–43. USENIX, 1991.

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

[23] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. Proceedings Real-Time Systems Symposium, pages 155–165. IEEE Comp. Soc., 1997.

[24] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. Hot Topics in Operating Systems, 2005.

[25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[26] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual services: a new abstraction for server consolidation. USENIX Annual Technical Conference, pages 117–130. USENIX Association, 2000.

[27] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. International Conference on Very Large Databases. Morgan Kaufmann Publishing, Inc., 2003.

[28] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. Conference on File and Storage Technologies, pages 259–274. USENIX Association, 2002.

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

[30] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. International conference on autonomic computing, 2006.

[31] E. Thereska, B. Salmon, J. Strunk, M. Wachs, Michael-Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 2006.

[32] Transaction Processing Performance Council. TPC Benchmark C, December 2002. http://www.tpc.org/tpcc/.

[33] Transaction Processing Performance Council. TPC Benchmark H, December 2002. http://www.tpc.org/tpch/.

[34] B. Urgaonkar and P. 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.

[35] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. Symposium on Operating Systems Design and Implementation, pages 239–254. ACM Press, 2002.

[36] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared memory multiprocessors. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):181–192, November 1998.

[37] C. A. Waldspurger. Memory resource management in VMWare ESX server. Symposium on Operating Systems Design and Implementation, 2002.

[38] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. RTAS – IEEE Real-Time and Embedded Technology and Applications Symposium, pages 125–134, 2006.

[39] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation in Cluster-based Network Servers. IEEE INFOCOM, pages 679–688. IEEE, 2001.

المنارة للاستشارات
www.manaraa.com