Performance insulation: more predictable shared storage

MATTHEW WACHS

September 2011

CMU-CS-11-134

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This material is based on research sponsored in part by the National Science Foundation, via grants CNS-0326453, CCF-0621499, CNS-0509004, CCF-0621508, IIS-0429334, and CNS-0917012; by the Air Force Research Laboratory, under agreement number F496200-01-1-0433; by the Army Research Office, under agreement number DAAD19-02-1-0389; by a subcontract from the Southern California Earthquake Center's CME Project as part of NSF ITR EAR-01-22464; and by the Department of Energy under Award Number DE-FC02-06ER25767. Support was also provided by an NDSEG Fellowship, which was sponsored by the Air Force Office of Scientific Research, Department of Defense; and by the member companies of the Parallel Data Laboratory Consortium over the period the research was conducted (APC, Cisco, DataDomain, EMC, Engenio, Equallogic, Facebook, Google, Hewlett-Packard, HGST, Hitachi, IBM, Intel, LSI, Microsoft, NEC, Network Appliance / NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Sun, Symantec, Veritas, VMware, and Yahoo!).

Keywords: Storage systems, shared storage, clustered storage, efficiency, quality of service, performance isolation, performance insulation, bandwidth guarantees.

To my parents.

Abstract

Many storage workloads do not need the level of performance afforded by a dedicated storage system, but do need the degree of predictability and controllability that comes from one. The benefits of consolidation, such as reduced waste, motivate the move to shared storage; but these benefits can be lost if the storage system is not shared effectively and efficiently among workloads. Unfortunately, inter-workload interference, such as a reduction of locality when multiple request streams are interleaved, can result in dramatic loss of efficiency and performance.

Performance insulation is a system property where each workload sharing the system is assigned a fraction of resources (such as disk time) and receives nearly that fraction of its standalone (dedicated system) performance. Because there is usually some overhead caused by sharing, there could be a drop in efficiency; but a system providing performance insulation provides a bound on efficiency loss at all times, called the *R-value*. We have built a storage system server called *Argon* to confirm that performance insulation can be achieved in practice for R-values of 0.8–0.9. This means that, running together with other workloads on Argon, workloads lose, at most, only 10–20% of the efficiency they receive on a dedicated system.

When storage systems are built from a cluster of modest servers rather than a single, monolithic server, techniques used to maintain efficiency do not necessarily compose across the servers. The resulting efficiency may actually be *lower* than the level achieved if no effort were made to preserve efficiency. We identify the causes of this effect and identify the level of coordination among servers needed to avoid this degradation. With appropriate care,

efficiency can be maintained on a clustered storage system as well as it can be maintained on a single server.

While performance insulation provides a useful limit on loss of efficiency, many storage workloads also need performance guarantees. It may be significantly more straightforward to express a workload's requirements directly as performance guarantees rather than indirectly as efficiency guarantees. To ensure performance guarantees are consistently met, however, the appropriate allocation of resources needs to be determined and reserved, and later reevaluated if the workload changes in behavior or if the interference between workloads affects their ability to use resources effectively. If the resources assigned to a workload need to be increased to maintain its guarantee, but adequate resources are not available, violations will result.

Though intrinsic workload variability is fundamental, storage systems with the property of performance insulation strictly limit inter-workload interference, another source of variability in resource requirements. Such interference is the major source of “artificial” complexity in maintaining performance guarantees. We design and evaluate a storage system called *Cesium* that limits interference and thus avoids the class of guarantee violations arising from it. Workloads running on Cesium only suffer from those violations caused by their own variability and not those due to the activities of other workloads. Compared to other quality of service systems proposed in the literature that do not explicitly manage efficiency, realistic and challenging workloads may experience an order of magnitude fewer violations running under Cesium as a result. Performance insulation thus results in more reliable and efficient bandwidth guarantees.

Acknowledgements

First and foremost, I'd like to thank my advisor, Greg Ganger. His positivity and easygoing approach have made graduate school an uncommonly pleasant experience for me. He has always been there for us when we needed his help. It amazed me how I could be working on a problem for weeks, and have him say something deeply insightful within a couple minutes of describing it to him.

I would like to thank my other committee members, Garth Gibson, Arif Merchant, and Raj Rajkumar, for serving and for their feedback and suggestions for clarifications, extensions, and experiments.

Being a member of the Parallel Data Laboratory (PDL) has brought with it remarkable opportunities for interaction with leading storage researchers in industry. I would like to thank all those who make this environment possible, including Karen Lindenfelser, Bill Courtright, and Joan Digney.

I would like to thank my colleagues and friends for enjoyable interactions and collaboration, particularly Mike Mesnier, Mike Abd-El-Malek, Raja Sambasivan, Alexey Tumanov, Julio López, Chuck Cranor, Brandon Salmon, Eno Thereska, Bryan Parno, Deepti Chheda, Nuno Loureiro, Aditya Sethuraman, Chandramouli Rangarajan, Mikhail Chainani, Miray Kas, and many other members of the PDL.

I would like to thank the companies of the PDL Consortium over the time I was a member of the lab: APC, Cisco, DataDomain, EMC, Engenio, Equallogic, Facebook, Google, Hewlett-Packard, HGST, Hitachi, IBM, Intel, LSI, Microsoft, NEC, Network Appliance / NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Sun, Symantec, Veritas, VMware, and

Yahoo!. I would also like to thank Intel, IBM, Network Appliance / NetApp, Seagate, and Sun for hardware donations that enabled this work. Apart from the support provided by these companies, I've enjoyed the interactions of feedback and encouragement we've been able to have with them, and thank them for their investments of time and interest in our work.

My work has been sponsored in part by the National Science Foundation, via grants CNS-0326453, CCF-0621499, CNS-0509004, CCF-0621508, IIS-0429334, and CNS-0917012; by the Air Force Research Laboratory, under agreement number F496200-01-1-0433; by the Army Research Office, under agreement number DAAD19-02-1-0389; by a subcontract from the Southern California Earthquake Center's CME Project as part of NSF ITR EAR-01-22464; and by the Department of Energy under Award Number DE-FC02-06ER25767. I also appreciate the support of an NDSEG Fellowship, which was sponsored by the Air Force Office of Scientific Research, Department of Defense.

Contents

Figures

Tables

1 Introduction

Sharing a storage system among workloads is appealing for a number of reasons. Virtualization is potentially as beneficial for storage as it is for other resources, such as CPUs. For instance, spare resources can be allocated to whichever workload is experiencing a surge in demand at a particular point in time, a principle known as *statistical multiplexing*. This reduces the need for over-provisioning and avoids the need to configure static partitions.

Thus, a system that can accommodate multiple workloads is desirable. However, merely accepting requests from and storing data on behalf of different workloads is not enough. The resources provided to each of the workloads must be allocated and managed appropriately so that each workload is able to use the system effectively.

Unfortunately, many storage systems do not balance their performance-critical resources among concurrent workloads, or fail to do so in an adequate way. The benefits of virtualization are potentially negated when storage systems do not support robust quality of service guarantees or fair and efficient sharing for workloads. Without such support, the actions of one workload may unduly affect the performance of others. If the different types of interference between workloads are not carefully managed, then the performance each workload receives may be highly variable, unpredictable, or inefficient. Under such conditions, the storage system may be essentially useless to any workload that needs to make steady progress.

Controlling the fraction of resources allocated to specific workloads is necessary, but not sufficient to solve performance variability and unpredictability. The resources must be allocated in a way that allows workloads to make efficient use of their fractions. Unfortunately, many systems attempt to optimize for other metrics at the cost of such efficiency.

For robust quality of service guarantees, resources must be managed among workloads in both a controlled and efficient manner. However, from a workload- or client-centric perspective, resource allocations are generally not a first-class concern; workloads care about the performance they receive more so than the allocations that led to such performance. This motivates a system that can accept and fulfill performance requests for workloads.

Some workloads have clear performance requirements that can be quantified as bandwidth or throughput goals. For instance, a navigation system may need to be able to read map data at a certain rate in order to make control decisions in pace with the speed of the vehicle. Other workloads may not have an immediately obvious bandwidth requirement, but there may be an empirically observed rate of bandwidth above which an application tends to have acceptable performance. For instance, for a database, the administrator may have observed that users generally begin to complain when the storage bandwidth falls below a certain level, even if the derivation of that value from first principles is far from obvious. For various types of complex workloads, specifying guarantees in the form of a bandwidth guarantee may be the most convenient and straightforward way to maintain acceptable performance.

Thus, a storage system that can efficiently support performance objectives in the form of bandwidth guarantees, and internally make the appropriate control and tuning decisions to meet the guarantees, is desirable. This dissertation describes a system property, *performance insulation*, that bounds inter-workload interference to a low level (e.g., 10%). It demonstrates a combination of techniques that are necessary and sufficient to achieve such insulation for realistic workloads. It also shows that previous approaches do not maintain such insulation. With insulation, bandwidth guarantees are more robust and require less over-provisioning; systems without insulation may suffer dramatically more guarantee violations.

1.1 Limitations of current techniques

A number of techniques have been proposed to improve fairness and provide quality of service when workloads share a storage system. Generally, these techniques attempt to maintain minimum bandwidth, minimum throughput, or maximum latency levels for each individual workload in a combination. Many function by managing the number of requests that each workload is allowed to submit to disk in a given period of time, maintaining fractional allocations of total disk time or bandwidth among the workloads.

However, these techniques do not attempt to maintain efficiency for the workloads or characterize the total amount of bandwidth or throughput available at any point in time to distribute among the set of running workloads. Thus, it is not known *a priori* whether a combination of bandwidth guarantees can be supported at any point in time, because it is not known whether the sum of the guarantees exceeds the total available bandwidth from the disk.

For disks, total bandwidth is not a fixed value, but rather a complex and variable function of the access stream. One defining characteristic of this stream is the combination of active workloads and their activities; another is the precise fine-grained interleaving of individual requests among the workloads that is selected by the scheduler. The feasibility and robustness of a set of guarantees are a function of both. While variation from the workloads themselves is a fundamental, unavoidable factor affecting storage performance, variation in interference is highly dependent on the scheduler. Current schedulers do not systematically control or limit the efficiency loss or added variance caused by changes in inter-workload interactions and interference. As a result, bandwidth guarantees are less robust and over-provisioning needs to be unnecessarily high.

1.2 Maintaining efficiency

This work improves on past approaches by maintaining each workload's efficiency level, even as the activities of other workloads and potential inter-workload interference levels change over time. To accomplish this, we have identified the sources of such interference and demonstrated mitigation techniques for them. To maintain efficiency without manual tuning, we have determined how to measure workload and disk characteristics and automatically adjust system parameters as needed to limit efficiency loss to a specified maximum.

The two primary storage system resources are disks and caches; interference may occur at both. For disks, the main cause of interference is disruption in the locality of a workload's request stream due to interleaving of requests from other workloads. For caches, the main cause of interference is eviction of one workload's pages by another workload's accesses, particularly when the first workload relies heavily on the cache for its efficiency.

These two sources of interference require different mitigation techniques. For disks, we give each individual workload uninterrupted spans of exclusive access to the disk, during which a workload can maintain its intrinsic locality. For caches, we trace and analyze access patterns to determine the number of evictions (beyond the level incurred under a dedicated cache) each workload can tolerate before it experiences significant loss of overall efficiency; we then size dedicated cache space for each workload appropriately.

1.3 Thesis statement and validation

This dissertation confirms the effectiveness and utility of performance insulation, a storage system property that ensures efficiency will be maintained even when workloads could potentially interfere with each other (insulation is described in further detail in Section 3.4.1). In particular, this dissertation experimentally supports the following thesis statement:

Performance insulation is an important and feasible building block for storage systems, providing a quantified bound on efficiency loss when systems are shared between potentially-interfering workloads and enabling more robust performance

guarantees with less over-provisioning.

To substantiate these claims, we run benchmarks and real workloads against a storage system without support for performance insulation or quality of service, against storage systems with representative quality-of-service schemes from the literature, and against a storage system that provides the property of performance insulation. We confirm that our implementation provides insulation and maintains efficiency — for instance, a reliable bound on efficiency loss of 10% or less — when combinations of workloads run concurrently, while other approaches are not consistently efficient. Other techniques suffer from dramatically lower performance for some combinations of workloads. We then attempt to maintain bandwidth guarantees with each of these approaches and find that the performance-insulation–based quality-of-service system provides substantially more robust guarantees for a realistic set of workloads when the system is not over-provisioned or only moderately over-provisioned.

1.4 Contributions

This dissertation makes the following contributions:

(1) It identifies the key sources of inter-workload interference for storage systems.

(2) It proposes mitigation strategies for interference that are conducive to automatic tuning.

(3) It experimentally confirms the presence of significant interference in systems that do not support performance insulation and the practical effectiveness of performance insulation techniques in limiting it.

(4) It provides the first evaluation of storage QoS approaches with workloads that exhibit significant variability in locality and other characteristics, such as are prevalent in realistic workloads.

(5) It demonstrates that prior QoS approaches are ineffective when faced with such workloads.

(6) It proposes a technique for storage QoS based on the efficiency-preserving foundation of performance insulation and experimentally confirms that the resulting bandwidth guarantees are more robust and efficiency higher, even in the face of significant workload variability.

(7) It introduces an explicit differentiation between periods when bandwidth guarantees are achievable and when they are not, allowing QoS schedulers to be compared to a reasonable ideal.

(8) It confirms that the proposed QoS system approaches the ideal, not only improving dramatically on previous approaches but eliminating the vast majority of avoidable bandwidth guarantee violations, both on single disks and RAID arrays.

1.5 Roadmap

This dissertation starts by describing past approaches for storage QoS in Chapter 2. Then, it describes how to maintain efficiency in shared storage systems in Chapter 3. The described techniques are appropriate for single-node storage systems, but require extensions to work on clustered storage systems; the necessary additions and an explanation of why they are required forms Chapter 4. Chapter 5 then builds on the foundation of performance insulation, and the techniques used to achieve it, to provide more robust bandwidth guarantees.

2 Background and related work

This chapter describes background for this dissertation and prior work related to quality of service for storage systems.

2.1 Background

Sharing resources, such as time on mainframe computers, has been a common practice for decades. When individual workloads do not fully utilize a system, and when systems have high acquisition or operating costs, it is natural to want to multiplex the system among workloads.

Straightforward techniques for sharing many resources have been in use for dozens of years. For instance, round-robin–timeslicing–based approaches for sharing CPUs are simple and effective, as are fair queueing models for network flows. Such resources can generally be shared with minimal overhead or complication. One might expect that storage systems could be shared with similar success using similar techniques, but efficient sharing of storage has been elusive.

The difficulty of sharing resources is affected by the amount of state maintained in the resource and by the latency of changing state. Network links, for example, do not retain state between packets, and the latency of beginning a new packet transmission after the previous one completes is not significant. Techniques for sharing networks thus do not need to concern themselves with, for instance, how frequently they switch back and forth between different flows, because there is negligible cost to doing so. CPUs retain state in the form of registers and caches. Switching between processes involves storing and loading registers (fast) and rewarming caches (potentially slow). However, for many workloads, the cost of rewarming caches and the speed penalty while the caches are cold are low enough not to pose a serious problem.

Storage systems are not as fortunate, however. Hard disks are mechanical devices, with the location of the disk head a physical state. The cost of seeking to data residing at another location on the drive (i.e., changing state) is high: as much as 30 ms, a time period during which multiple accesses to locations closer to the head location could potentially have been executed. Thus, sharing policies must consider whether they are introducing unnecessary or excessively frequent seeks that would reduce efficiency.

Incurring unnecessary repositioning delays can, in fact, reduce the efficiency with which a workload is handled by an order of magnitude or more. When multiple workloads access the same disk concurrently, the combined request stream may contain interleaved requests from the different workloads. For many combinations of workloads, the locality of this combined stream is significantly lower than the locality within the individual workloads' streams. As a result, the disk head may need to frequently seek back and forth between the workloads' respective files. While disk firmware is locality-aware and can reorder a sequence of requests in the queue to reduce seek times, queue depths at the firmware level can be far too short to provide high efficiency levels for concurrent workloads. Moreover, disk firmware is not QoS-aware or likely to be conscious of which requests come from separate workloads.

Caches are also key storage system resources. Storage caches are larger and slower to load than CPU caches, and blocks generally remain in storage caches for much longer. Thus, beyond mere speed and size differences, storage caches also feature potentially–long-lived cache footprints for individual workloads, which introduces the concern that the relative footprints across the workloads may not be optimal or efficient.

2.2 Motivation

As described in Chapter 1, sharing a storage system is motivated by the benefits common in virtualization and consolidation, such as statistical multiplexing and reduced over-provisioning. Systems that do not manage their resources effectively among the set of concurrently-executing workloads may negate these benefits entirely.

While many systems have been proposed that attempt to control concurrent workloads' accesses to resources so that all of them can coexist and receive their desired levels of performance, none adequately controls efficiency to consistently maintain it at a high level. This dissertation focuses on techniques to maintain efficiency within storage systems and their application to quality of service techniques. Increased efficiency means that workloads complete their requests more quickly and use fewer resources. Performance and efficiency (lack of waste) are themselves worthy goals, but a reduction in resource consumption also reduces the provisioning required to achieve a desired level of performance. As a result, guarantees are more likely to be met (since fewer resources are needed to meet them) and more workloads may be able to coexist on the same system. In addition, a guarantee of consistent efficiency means one major source of variability — swings in efficiency caused by changes in inter-workload interference — is reduced to a negligible level.

2.3 Related work on storage QoS

This section describes past work on providing quality of service for storage. Other chapters of this dissertation include further related work specific to the scope of the corresponding chapter. This additional related work can be found in Sections 3.6.4, 4.4.3, and 4.6.

A significant body of work on storage QoS exists. Generally, papers discuss two aspects of quality of service: managing the request streams coming from the workloads in order to attempt to guarantee performance, and determining whether or not a given workload and its associated performance guarantee can "fit" on a particular system (known as admission control). These aspects will be discussed separately.

2.3.1 Managing request streams

There are two common approaches to request stream management: bandwidth-based request throttling and deadline-based scheduling. Some additional papers do not neatly fall into these categories.

Virtually all papers employ the strategy of deferring requests in some manner. Systems that defer requests maintain a queue or queues in which requests received from clients can be postponed from being sent to the disk. Requests are dispatched or scheduled (i.e., released from the queue and sent to the disk) when it is believed that doing so will not cause other workloads to miss their performance targets; further, requests are not intended to be deferred beyond the point at which they will miss guarantees [5, 9, 19, 22, 24, 32, 40, 46, 49, 56, 57, 59]. The biggest distinctions between different systems are the exact goals and the policies behind how this general mechanism is applied.

Bandwidth-based throttling

Systems that perform bandwidth-based request throttling manage the bandwidth that different workloads or classes of workloads receive, to attempt to ensure that no workload or class receives more than it needs at the expense of another workload or class underperforming. Generally (with some exceptions), these systems do not provide latency guarantees, because the techniques used for managing bandwidth are not conducive for directly controlling latency as well.

One system, YFQ [5], is a fair-queuing scheduler that controls the *proportions* of bandwidth each workload receives. Workloads are assigned *weights*, which specify their relative priorities. The total bandwidth that the mixed request stream will receive from the disk is not necessarily known in advance, but the fractions of that bandwidth received by each workload are established by the weights. Thus, the system allows for prioritization, but not strong bandwidth guarantees in terms of absolute quantities. While techniques for improving efficiency, and thus the total amount of bandwidth being divided up, are mentioned, they are not sufficient for achieving efficiency levels approaching 100%.

A more common approach is establishing absolute performance (specifically, bandwidth) targets for each workload using a *token bucket* or *leaky bucket* model [54]. These models establish a rate (bandwidth) for a workload, and permit a bounded amount of burstiness for that workload so long as it remains close to the specified rate. Permitting burstiness allows workloads that need occasional, high instantaneous throughput to receive it, so long as they also have periods of lower demand that allow the load to "average out" to the specified rate over the medium term. This is accomplished by analogy to a bucket and a liquid. Drops of liquid (*tokens*) represent requests. The liquid can be thought of as raining into the bucket (and thus adding to its contents) at the specified rate. The workload drains liquid out of the bucket whenever it wants to issue a request; if the bucket is empty, it must wait until liquid accumulates. The significance of the bucket is that it provides a buffer; if the workload draws from the bucket less quickly than it is being filled, the excess remains available for later when the workload may increase its demand. The size of the bucket represents the bounded amount of "burstiness" allowed for the workload. If the workload allows the bucket to fill by issuing fewer requests than its rate, then it can later "catch up" — but only up to the size of the bucket. If the bucket overflows, the workload forfeits the ability to catch up beyond one bucket's worth of deficit. This design is partially intended to prevent the scenario where a workload is idle for a long period of time, then becomes active and attempts to monopolize the system for a prolonged period in order to catch up.

Most *throttling*-based systems either pass through requests (if a workload is not exceeding its limits, i.e. liquid is available) or queue them (until a workload begins to fall below its limit, i.e. liquid becomes available). Some systems predominantly focus on a straight implementation of this policy, such as Zygaria [57] and SLEDS [9].

SLEDS [9] uses a leaky-bucket model to throttle workloads. The goal is to provide both bandwidth and latency guarantees to workloads, with the bandwidth guarantee specified using a leaky bucket. If at least one workload is failing to meet its guaranteed level of bandwidth or latency, workloads that are presenting more demand than they are guaranteed (i.e., their offered load exceeds the guaranteed level of bandwidth) are throttled to only receive their guarantee. Workloads that do not need to be throttled (i.e., they are naturally staying within their leaky-bucket bound) are also provided with their latency guarantee; workloads exceeding their limit are not guaranteed anything with respect to latency. Beyond this degree of control, no further optimization is provided; for instance, requests are not reordered to improve efficiency. This is not attempted because SLEDS is intended to be used as a pass-through performance manager that runs on top of an arbitrary storage system, possibly consisting of many resources. Thus, the underlying system is treated as a black box.

Triage [24] also uses a black-box approach. Workloads are throttled to their guaranteed request rates. Workloads are also given latency guarantees, which are managed adaptively. So long as all workloads are meeting their latency guarantees, Triage increases the queue depth at the storage device. If latency guarantees begin to be violated, the queue depth is decreased. This controller is based on the observation that longer queue depths may be beneficial because they allow for request reordering that can increase performance and efficiency. Longer queues can increase request latency, however; so there is some limit to how long they can be for a set of workloads and associated latency requirements. Additional complexity comes from the fact that the limit may change over time as the workloads and their interactions change; rather than attempting to model this behavior, feedback control is used to dynamically retune the system to the limit that is currently appropriate. In building such a control system, various questions about stability need to be addressed; Triage explores automatic tuning that chooses appropriate system parameters to promote stability. Similarly to YFQ, Triage observes that total system throughput is unpredictable due to differences in workload access patterns and inter-workload interference. Rather than attempting to strictly guarantee absolute quantities of throughput, Triage throughput guarantees are in the form of a piecewise function that ranges over different system operating regions. For instance, rather than guaranteeing workload A will receive 50 IOPS and workload B will receive 250 IOPS, the guarantee varies with the total number of IOPS the system is providing at a particular point in time. Guarantees are in the following format: If the system is providing ≤ 100 IOPS, then they will be split evenly between workload A and workload B. If the system is providing ≤ 300 IOPS, then split the first 100 IOPS as previously, then allocate the surplus to workload B. Thus, the following (*workload A IOPS*, *workload B IOPS*) pairs are all legitimate: $(25, 25)$, $(50, 50)$, $(50, 150)$, and $(50, 250)$. This is a novel approach for handling the unpredictability of total system throughput, by exposing the variability to the system administrator rather than attempting to suppress it. While such added flexibility may allow more expressive guarantees and more administrator control over the tradeoffs between workloads, it is arguable whether the significant increase in complexity is acceptable.

Zygaria [57] uses basic token-bucket scheduling to provide bandwidth guarantees to workloads. It differs in two ways: by accommodating sequential workloads better and by providing water-level fair sharing. Token buckets, by their nature, treat all requests as the same — a request consumes a token, or a request consumes a number of tokens corresponding to the number of bytes in the request. Ideally a token bucket, however, would capture the *cost* of a request to the underlying device. For sequential requests performed contiguously in larger blocks, cost does not scale linearly with size; positioning time at the beginning of the request dominates over incremental transfer time to read additional bytes. Zygaria allows sequential I/Os up to a fixed maximum size (the authors choose 32 KB) to be treated as a single request. This somewhat improves the amenability of throttling for sequential workloads, but does not maintain the high efficiency that such workloads should be capable of. A separate issue addressed by Zygaria is what policy should be applied when each workload is receiving its guaranteed level of throughput but there is excess bandwidth still available. Many systems provide "fair sharing" where each workload receives equal amounts of the excess. Zygaria adopts a novel alternative, *water-level fair sharing.* In this approach, excess capacity is used to approach the state where each workload receives the same performance. Like rising water, the workload with the lowest guaranteeed performance receives all of the surplus until its performance matches the workload with the second-lowest guaranteed performance (if there is enough surplus to even reach this state). Then, both workloads receive equal amounts of surplus until their bandwidths both match that of the third-lowest guarantee, and so on.

Deadline-based throttling

Systems that perform deadline-based scheduling attempt to provide latency guarantees by labeling each request with a deadline by which it must complete; generally, this is the arrival time plus the maximum latency. Based on expected service time at the disk (and the expected service time of concurrent requests), this establishes a time by which the request must be submitted to the disk; before that time, the system may perform other work. Alternatively, the system may send work to the disk early, but exploit the "slack" before deadlines to attempt to improve the efficiency with which requests are handled (e.g. by reordering or coalescing them to increase the locality of the request stream). Depending on the nature of a workload (e.g., open- or closed-loop), latency guarantees may indirectly correspond to throughput guarantees as well.

RT-FS [40] can either use slack to provide extra service to workloads or to handle best-effort requests for non-guaranteed-service workloads. In either case, requests not dispatched just-in-time for a deadline are scheduled in SCAN order — in other words, sorted by location on disk.

Cello [49] considers two classes of workloads: best-effort and real-time. A two-level scheduler design is proposed; a top-level scheduler balances between the two classes, and class-specific schedulers manage the types of requests and appropriate goals for their particular class. The top-level scheduler meters time or bytes to individual classes. A best-effort class scheduler will not attempt to provide guarantees, but will try to reorder requests to increase disk seek efficiency. A real-time class scheduler will attempt to provide guarantees to real-time workloads by sorting requests in order of completion deadlines and submitting them to the disk early enough to ensure meeting these commitments. The paper primarily concerns itself on coordinating the activities of multiple schedulers without them each needing to understand each others' inner workings or specific goals; the effectiveness and efficiency of the guarantee mechanisms is of secondary concern.

Wijayaratne and Reddy [56] describe a similar three-class system. In their system, however, all three classes receives guaranteed performance. The three classes are periodic workloads (e.g., open-loop workloads such as video playback), interactive workloads, and aperiodic ("normal") workloads. An overall timeline of the deadlines for each of pending requests is maintained, with the intent that requests will be sent to disk at the last possible point in time that will still allow their deadlines to be met (for periodic and aperiodic workloads). For those workloads, completing requests "early" may be of little benefit. For interactive workloads, though, this artificial postponement is not as acceptable. Thus, interactive requests are dispatched immediately, so long as doing so will not cause another request to miss its deadline. To increase disk scheduling efficiency, when interactive requests are sent to disk, other queued requests that are "nearby" on disk will be *promoted* (sent earlier than required) and sent at the same time so that they can be handled together with the interactive request for a lower total cost. The paper focuses on managing classes rather than individual workloads, and it does not explore the case where guarantees for different classes or workloads may conflict: how to detect this case is happening, how to handle this case, and how to do admission control to prevent incompatible workloads from being placed together on a disk in the first place.

Façade [33] provides latency guarantees that are a function of offered request rate, rather than in absolute terms. Each workload's guarantee is in the form of a pair of curves, one specifying read latency and the other specifying write latency. The intention is that latency is allowed to be higher if a workload has submitted more requests (since a request must queue behind other requests from the same workload, even before inter-workload interference is considered). Once the governing latency target (under the current request load) for each workload is known, standard deadline-based scheduling is used, alongside a novel feedback controller that manages queue depths. This feedback controller is based on the same efficiency / queue-depth / latency relationship exploited in Triage (note that Façade preceded Triage). Longer queues may allow more efficient request scheduling and disk seeks. The danger of longer queues, however, is the fact that requests cannot be cancelled once added to the disk queue; thus, an arriving request with a deadline that will occur before the current queue is drained cannot be assured of meeting its latency target. Therefore, the controller tries to lengthen the queue, which may (or may not) improve efficiency, so long as latency targets are being met. If the controller observes that one or more targets are being violated, queue depth is decreased until latency is back under control. The queue depth, and thus the overall efficiency of the system, is limited by the workload with the lowest latency requirement.

Stonehenge [22] attempts to virtualize bandwidth and latency for workloads using a standard deadline-based scheduler with fixed latency targets, but adds an additional optimization for locality. Two queues are maintained: one in deadline order and another in CSCAN order. When slack exists (i.e., requests do not need to be dispatched from the deadline queue immediately to avoid missing their deadlines), Stonehenge "sneaks ahead" requests from the CSCAN queue. This has the effect of explicitly managing disk scheduling during slack periods, rather than simply managing queue lengths and allowing disk scheduling to be performed in the disk's firmware.

AVATAR and SARC [59] are a pair of schedulers, used together, to provide latency and bandwidth guarantees. Taken together, they are intended to provide throughput guarantees over one second windows and latency guarantees to a specified percentile (e.g. 95^{th}-percentile latency). The AVATAR scheduler does standard earliest-deadline-first scheduling based on request arrival times and latency guarantees. In addition, AVATAR works similarly to Façade in that it determines whether to shorten (latency guarantees are being violated) or lengthen (there is available slack to exploit) the queue. However, AVATAR employs a queueing-theoretic approach to this decision rather than a control-theoretic approach. The SARC scheduler throttles workloads to their guaranteed request rates and provides fair sharing among "excess" requests in the case where AVATAR determines that the disk is underutilized even when meeting all guarantees, and, thus, spare bandwidth is available. The authors also identify an important control-theoretic "corner case" that they argue should be handled differently than it is in prior work: When deadlines are being violated *en masse*, other systems often shorten queue lengths to try to reduce latency. In general, this is the correct action to take when minor violations are occurring. When a significant backlog has developed, however, the authors argue that the correct action to take is to lengthen queues and attempt to clear out the backlog quickly. While every deadline may be violated during this period, the hope is that by taking aggressive action, the system will be returned to a normal, manageable operating point quickly. Were the system to throttle the workloads and continue to perform conservative scheduling, it may not be possible to "catch up," and the backlog — and its associated guarantee violations — may remain indefinitely.

pClock [19] performs a combination of leaky-bucket based throughput management and deadline-based scheduling. Workloads receive bandwidth and latency guarantees if they do not exceed the request intensity specified by the leaky bucket model. The authors introduce a novel method of performing token bucket calculations — the *arrival curve* — which is more computationally efficient. Spare capacity is used for background tasks, which are allocated contiguous batches of time up to the amount of slack available before the next deadline; this may improve the efficiency of background tasks, which are often sequential. (Alternatively, spare capacity can be directed to foreground tasks, providing service that is not accounted for under token-bucket limits.) Workloads are not necessarily throttled if they exceed their specified limit, but are no longer granted guarantees. This places the onus on a workload to determine if its performance is better over the limit or under it. Requests are assigned *start tags* and *finish tags*, with finish tags similar to deadlines in other systems. Start tags represent when a flow that is not exceeding the leaky-bucket limits would have been able to submit the request. Finish tags are computed using the start tags, maximum latency, and arrival time, and only workloads within their leaky-bucket limits are guaranteed that their finish tags will be no farther in the future than their maximum latency. Requests are dispatched in finish-tag order. A decentralized implementation of the algorithms for use in a storage cluster is presented as dClock [18].

Whether allocating contiguous spans of time to background workloads, performing SCAN / CSCAN dispatch, promoting nearby requests, or increasing parallelism, the goal of all of these variants of deadline-based scheduling is to try to improve throughput and efficiency without violating latency bounds. The exact amount of improvement is dependent upon the workloads and cannot be easily bounded.

Additional related work

Aqueduct [31] is a special-purpose scheduler intended to preserve the performance of foreground workloads while migration (the copying of data belonging to another workload to or from the server) is occurring in the background on the same disk. Foreground workloads are provided bounded average latency over 60-second sampling periods. Rather than attempting to determine up front the rate at which the background workload can proceed without causing foreground workloads to miss their latency targets, the system uses a feedback-driven controller to adapt to the levels of interference occurring at each point in time. If the latency guarantee is presently being violated, then migration needs to be throttled back; if not, the current rate is acceptable, and increasing the rate to provide better migration performance may be safe to attempt. This strategy has the advantage of avoiding overly-conservative up-front estimations of the appropriate migration rate by searching for the exact boundary between an acceptable and an unacceptable migration rate. The disadvantage is that, to “hug” this boundary tightly, the system will have to risk crossing it, resulting in transient violations. Also, if the boundary changes over time, violations will occur until the system reacts appropriately and sufficiently. Thus, in the system the latency was calculated over a 60-second window to allow violations to be averaged out. Workloads requiring much stricter latency bounds cannot be accommodated using this strategy. While this simple control system is effective for this special-purpose use, more complexity may be necessary when more than two workloads or workload classes are being managed. For instance, if deadline violations occur, it may not be immediately obvious which other workload should be scaled back from among the possible choices. In addition, in Aqueduct, the background migration workload is not given a guarantee, so there is no lower bound on its rate, which gives the system considerable flexibility to throttle it; in a system attempting to provide guarantees to all workloads, workloads cannot be rate-limited below their guarantee levels.

Fahrrad [46] uses Argon's approach (Argon is described in Chapter 3) of providing guarantees in terms of utilization rather than performance. Reservations specify desired throughput or latency, and expected I/O behavior (e.g., sequentiality and burstiness). A broker translates these reservations into the anticipated level of utilization that must be reserved. The standard deadline-based scheduler achieves guarantees over a period of time under the condition that all I/O requests that will be issued during that period are known at the beginning of the period, allowing for decisions that are theoretically optimal but unlikely to be practical in real systems.

Maestro [37] is a adaptive feedback controller, similar to Façade and Triage, that provides either latency or throughput targets for workloads, in this case for large disk arrays. Maestro works by varying concurrency levels for applications to meet targets and to enforce the relative priorities of workloads when insufficient performance is available to meet all the targets.

Limitations of these approaches

These approaches to providing performance guarantees suffer from a number of key limitations.

Systems that use token buckets or leaky buckets to "provide guarantees" use these models in a manner that is fundamentally "backwards" to their actual nature. The desired guarantee for a workload in these systems is specified by the rate and burst characteristics of a token bucket. Each workload is then throttled (i.e., limited) to the bucket parameters. The intent is to guarantee each workload will receive a *minimum* level of performance specified by the bucket, but, in fact, the bucket specifies a *maximum* level of performance: it is used to establish the level at which throttling occurs.

This "mismatch" is justified by the following argument. Suppose all the workloads and their corresponding guarantees *are* able to fit together on a particular system. Suppose further that they fit together under the scheduling policy being used (since the scheduling policy can affect system capacity). However, it is not necessarily known if the workloads will be able to coexist if one or more workloads *exceeds* its guarantee. Thus, the role of the system is to prevent any workload from receiving more than its guarantee, at least until it is certain that doing so will not cause another workload to receive less than its guarantee. (Workloads are not necessarily designed to send requests at exactly the guarantee rate, so the system itself is seen as assuming the role of controlling them.)

This "mismatch" is dangerous if the opposite situation holds, however. Suppose not all the workloads and their associated guarantees can fit on the system. Then, even limiting each workload to its desired level, at least one workload will still not reach its guaranteed level of performance. No change to the throttling can solve the problem because no workload is receiving excess resources; one can only potentially change which workload is the "victim."

Thus, throttling-based systems are not able to provide guarantees at all, unless they use a separate mechanism to test the feasibility of a particular set of guarantees. The real complexity of providing guarantees in these systems is transferred to that mechanism, but completely satisfactory solutions to the problem of determining feasibility of workload combinations under throttling do not yet exist. Furthermore, throttling-based systems use a particular class of scheduler that is not specifically designed to manage efficiency, and they often achieve poor efficiency levels in practice. Thus, even if the workloads are feasible on a system's hardware, they may not be feasible under a throttling scheduler. In addition, another consequence of throttling schedulers not explicitly managing efficiency is that there may be swings over time in efficiency and therefore, changes in the set of workloads that can fit on a particular system. Hence, the critical feasibility test is made significantly more complex due to this variance.

Systems that perform deadline scheduling provide latency guarantees for workloads. In particular, they attempt to ensure that the latency of requests never goes over a specified limit. In other words, their guarantees are for maximum latency. For workloads that truly need maximum latency never to exceed a particular value for even a single request, deadline-based schedulers with request reordering may provide the best possible performance.

It is not clear, however, what proportion of latency-sensitive workloads need guarantees in this form. It may be possible to provide slightly more flexible guarantees that still result in good latency for most requests; for example, guarantees on three values: mean latency (e.g., $\leq$ 25 ms), n^{th}-percentile latency (e.g., 95^{th}-percentile $\leq$ 50 ms), and maximum latency (e.g., $\leq$ 750 ms). By permitting occasional high latency while still preserving good latency for the majority of requests, it becomes possible in some cases to employ techniques like Argon's timeslicing (a main component of this dissertation, described in Chapter 3), which may provide dramatically better efficiency and bandwidth. Some deadline-based schedulers were evaluated, when published, with latency guarantees of approximately one second. Timeslicing can achieve very high levels of efficiency with similar or better maximum latencies, while also providing significantly better mean latency. One frequently-used example workload that *does* require hard maximum latency guarantees is video streaming, which is seemingly incompatible with timeslicing. This workload is highly sequential and predictable, however; thus, even a timeslice-based scheduler may be able to perform prefetching and provide workloads like this one consistently with "zero latency" (i.e., the latency of prefetch cache hits only).

In addition, even systems that perform strict deadline-based scheduling do not necessarily guarantee true hard latency maxima. This is because they must estimate the behavior of the disk in order to know the cutoff before which requests must be sent to the disk in order to finish on time. This entails more complexity than it might seem, and without perfect knowledge of disk characteristics, this estimate is itself imperfect. A deadline-based scheduler usually operates at a level above the disk scheduler and is not perfectly aware of disk geometry, a key determinant of positioning delays. Thus, request completion times at the disk are not known with certainty in advance; the last moment at which a request can be sent to the disk and complete on time can only be estimated, either with over-conservatism (which reduces opportunities to exploit slack and increase efficiency) or with the possibility of incurring latency violations at least some of the time. This problem is further compounded when more than one request is kept outstanding at the disk; the order of completions is not known in advance and there may be greater variance in completion times. Another problem is that disks occasionally have latencies well beyond the expected or advertised maximum latencies due to retries, thermal recalibration, and other effects. The level of conservatism needed to weather such occurrences without any latency violations may be either impractical or completely impossible (for latency guarantees that are shorter than the latency of a thermal recalibration, for instance). Therefore, to some extent, even strict deadline-based scheduling may not be providing true hard latency maxima, but only a guarantee of latency up to a certain percentile. Hence, it is not clear in what senses the guarantees offered by these systems are consistently better than those offered by the timeslicing-based approach described in Chapter 3, yet the efficiency these systems achieve may be significantly lower.

2.3.2 Admission control

Many papers that discuss policies for performance guarantees treat admission control as an orthogonal topic, which they do not explore in detail [5, 9, 24, 32, 59]. Among the papers that do discuss the topic further, there are three common approaches to admission control: making admission decisions based on current behavior (rather than possible future behavior); accounting for the worst possible case for each workload, and thus making correct (if very conservative) admission decisions; and performing admission control over utilization instead of bandwidth.

Making admission decisions based on current behavior is appealing, because it allows for a “trial-and-error” approach based on actual observations. This approach is adopted by Wijayaratne and Reddy [56], and Stonehenge [22]. The problem with this approach is the reliance on current behavior, which may not be predictive of future behavior. A set of workloads may fit together on a server now, but if even one of them were to change, at any point in the future, the reservations may no longer be adequate for any of the workloads (inadequate for the workload that changed because it changed; inadequate for the others because inter-workload interference may also have changed).

Accounting for the worst possible case for each workload is attractive, because it can provide guarantees that are highly likely to be met; after all, no case is worse than the worst case. This approach is adopted by Zygaria [57]. Usually the worst case is taken to mean highly random request streams with minimal locality, minimal reuse (no benefit to caching), and minimal request sizes (small amounts of data transferred per unit of work and time). The resources needed to provide the guaranteed level of bandwidth under these worst-case assumptions are reserved. Some workloads operate close to the worst case; for them, this approach is effective. Others, especially workloads with high locality, good cacheability, and / or large block accesses, are not handled well by this approach. For them, reservations would be made for a worst case that may be more than an order of magnitude more costly than their actual resource usage. For some workloads that could easily fit on a single disk, providing the same bandwidth to a worst-case workload is impossible because it exceeds the dedicated-disk performance for a worst-case workload. Thus, these policies avoid making guarantees that can’t be kept, but can result in extreme over-provisioning and the rejection of perfectly legitimate workloads, even running alone.

Performing admission control over utilization instead of bandwidth greatly simplifies the problem. This approach is taken by Argon. Utilization, which is essentially the same as the amount of resources needed for a workload, is a fraction. Implementing admission control on a fraction of resources is as simple as checking to make sure the fractions allocated to each workload do not sum to more than 100%. Admission control is usually done over externally-determined allocation requests. If the user or administrator determines and specifies a utilization goal, this entirely sidesteps the challenging problem of converting from a bandwidth goal into a resource level within the storage system. But, we anticipate that for most workloads, users and administrators have a better sense of what bandwidth (e.g., 6 MB/s) is required than what level of resources is required (e.g., one-tenth of a disk). Thus, while easier to implement, it is not clear that utilization-based guarantees and admission control are practically useful for many types of workloads.

3 Efficiently sharing a storage system (Argon)

A significant limitation in existing approaches for storage QoS is the lack of focus on efficiency. Efficiency is a worthy goal on its own, and it is desirable to provide better performance and efficiency to workloads sharing a storage system even if they do not require bandwidth guarantees. But, efficiency is also a valuable building block for workloads that *do* need minimum levels of bandwidth: Increasing efficiency may also significantly reduce variance in bandwidth due to interference, and reduce the amount of resources required to achieve the same level of performance for a workload. This reduces the over-provisioning needed to achieve the same guarantees for more workloads simultaneously. Less over-provisioning means more workloads can co-exist. And less variance means that guarantees are more robust.

By improving efficiency, systems that are currently already shared may perform better and more predictably and be able to host more workloads. But, the potential benefits extend beyond that; shared storage may become practical for a wider set of workloads, allowing them to be moved from dedicated or statically-partitioned systems to systems that are more cost-effective and whose slack is being shared among more workloads more flexibly and effectively.

This chapter describes the key sources of inefficiency when workloads share a storage system and the techniques that are necessary and sufficient to limit them. To eliminate other variables, it uses microbenchmarks to demonstrate the causes of interference and the effectiveness of the solutions which prevent it. To show which combination of techniques is adequate for more realistic workloads, it provides macrobenchmark results as well.

3.1 Introduction

When active workloads share a server, each will receive only a fraction of the server's resources. Thus, each should expect to receive less than the bandwidth it could achieve on a dedicated server. But, each workload should be able to use its fraction of resources with nearly the same efficiency as it receives when it runs alone. Unfortunately, when workloads share a storage server, they often interfere with each other, reducing each other's efficiency dramatically.

The causes of interference at a storage system can be placed into three categories. First, "aggressive" workloads can issue a disproportionate number of requests, and thus receive a disproportionate amount of attention and service from the storage system at the expense of other workloads. Second, fine-grained interleaving of requests at the disk among workloads can result in greatly diminished locality in the combined request stream; locality is a key determinant of hard disk performance due to the physical and mechanical aspects of hard drives. Third, one workload can suffer excessive evictions from the buffer cache (or equivalent) due to the requests of another workload.

Locality is important when using hard drives because they are physical devices that must perform mechanical motions (*seeks*) in order to service requests. Compared to the time scales at which other events (such as instruction execution, memory access, or network packet transmission) occur, these seeks take significantly (orders of magnitude) longer. Some workloads, such as so-called *sequential* or *streaming* workloads, perform better than others at a hard drive because the physical locations of consecutively-accessed blocks are nearby on the disk surface (thus, fewer or shorter seeks). For these workloads, if anything causes their intrinsic locality to be disrupted, such as the interleaving of requests from other workloads with data stored elsewhere on the surface, then their performance may drop by a factor of ten or more. In particular, their performance may deteriorate from that of a sequential application to that of a random application (in other words, from one extreme on the continuum of disk locality and performance to the other).

Similarly, caches can have a significant impact on workload performance. Cache misses are two orders of magnitude slower than cache hits. Without care, it is easy for one intensive workload to dominate the cache with a large footprint, significantly reducing the hit rates of other workloads.

In contrast to the unboundedly-low efficiency that may occur due to disk and cache interference, this chapter discusses the goal of maintaining the efficiency of each workload sharing a system to at least a configurable fraction (e.g., 0.9) of the full efficiency it achieves when it has the system entirely to itself, regardless of the number or activity of other workloads using the same system. We call this fraction the *R-value*, drawing on an analogy to the thermal resistance measure in building insulation. With a "perfect" R-value of 1.0, sharing would affect the portion of server time and resources dedicated to a workload, but not the efficiency with which the workload can use that portion. Thus, while a workload's performance would be scaled down as it contends for a bottleneck resource, the reduction in performance would be no more than should be expected given that it does not have full use of the resource. In a system with a realistic R-value of 0.9, workloads would suffer at most a slight (10%) further deterioration of performance due to sharing overhead, but no further loss than that. Not only does this increased efficiency result in increased performance, but also in increased predictability of performance and a much simpler and more predictable characterization of the effects of sharing.

The Argon storage server, described in this chapter, combines three mechanisms plus automated configuration to maintain efficiency under sharing. First, it detects sequential streams and uses sufficiently large prefetches and write requests to amortize positioning costs and achieve the configured R-value of streaming bandwidth. Second, it explicitly partitions the cache to prevent any one workload from squeezing out others. To maximize the value of available cache space, the space allocated to each workload is set to the minimum amount required to achieve the configured R-value of its full standalone efficiency. For example, a workload 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 for all workloads. Situations where not enough cache space is available to host a newly added workload are explicitly identified after a short period. Third, disk time quanta are used to separate the disk I/O of workloads, eliminating interference that arises from workload mixing except when one quantum is ending and another beginning. The length of each quantum is determined by Argon to achieve the configured R-value, and average response time is kept low by the improvement in overall server efficiency.

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 and consistently provides each workload with at least the configured level of 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.

3.2 Intended applications

This section describes the workloads and storage systems for which the techniques in this chapter are appropriate, and those for which the techniques are not suitable. Section 3.7 revisits this discussion at the end of the chapter, indicating why these limitations exist and to what extent they may be remediable with further refinement.

Argon is designed for single-server storage systems. (Accommodations for clustered storage systems are described in the next chapter.) We target storage systems where we can implement our techniques at a layer that is aware of request-to-disk mappings and that can control the caching policy.

Argon should work with a broad range of workloads. However, for best results, a workload should have enough activity that Argon can detect its access patterns reasonably soon after it begins executing. In addition, full insulation cannot be provided to sets of workloads whose access patterns are each too demanding of the cache to be able to co-exist well on a server. Finally, Argon's proportional allocations of server time are reasonable for many workloads, but may not be suitable for some, particularly those with significant idleness. While the resources provided to workloads with idleness are not improper, there is likely to be a more appropriate format of guarantee and a more appropriate type of resource allotment for such workloads.

Argon targets workloads who need efficiency in terms of bandwidth (or throughput). Argon's impact on latency varies, depending upon the specific metric of latency being considered; however, Argon is not suitable for workloads requiring hard real-time maximum latency guarantees.

3.3 Sources of interference

This section describes in greater detail the causes of low efficiency and high inter-workload interference in storage systems.

3.3.1 Request interference

A workload that submits a disproportionately large number of requests can receive a disproportionate level of attention from the storage system because attention is, in many systems, allocated on a request-by-request basis rather than in controlled fractions to workloads. Thus, workloads that maintain high concurrency of requests can starve other workloads. This type of interference does not directly reduce the efficiency other workloads receive in their fraction of server time, but rather reduces the fraction of server time they receive. Control over the allocation of time among workloads must be maintained to prevent one intense workload from dominating request queues.

3.3.2 Disk head interference

The fraction of "useful work" a workload accomplishes on a disk can be defined as the fraction of the average disk request's service time spent transferring data to or from the magnetic media (as opposed to waiting for positioning of the head or medium). The best case, sequential streaming, achieves useful work levels of approximately 0.9, falling below 1.0 because no data is transferred when switching from one track to the next [48] (the only positioning delay generally incurred by such workloads). Non-streaming access patterns can achieve useful work levels well below 0.1, as seek time and rotational latency are much greater than data transfer time. For example, a disk with an average seek time of 5 ms that rotates at 10,000 RPMs would provide useful work level of $\sim$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 $\sim$0.02.

Disk head efficiency under sharing can be defined as the reduction of useful work suffered by a workload due to sharing. For instance, if a workload's useful work level on a standalone disk is x and under sharing is also x, then the efficiency level is 1.0; if standalone useful work is x and under sharing is $0.5 \cdot x$, then the efficiency level is 0.5. Note that efficiency is not the same as performance, in part because it does not capture whether the number of requests or total amount of disk time allocated to a specific workload changes — only the efficiency with which its requests are handled, however many or few of them there may be.

Interleaving the access patterns of multiple workloads can reduce disk head efficiency dramatically if doing so breaks up sequential streaming. This often happens to a sequential access pattern that shares a disk 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 workloads' requests immediately after completing one from the sequential pattern. The scheduler's choice of another workload'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. Such larger requests are beneficial because they 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 [43, 48] — for example, the 64 KB prefetch size common in many operating systems (e.g., BSD and Linux) raises efficiency from ~0.015 to ~0.11 when sequential workloads share a disk. As shown later in this chapter, efficiency levels of 0.9 are achievable. More aggressive use of prefetching and write-back aggregation is one tool used by Argon for performance insulation.

3.3.3 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 services a user request without consuming any disk head time, reducing disk head contention.

With traditional cache eviction policies, it is easy for one 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 workload that writes large quantities of data to fill the cache with its dirty blocks. In addition to reducing other workloads' 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 on the critical path in order to create free buffers before the next read or write can be serviced.

3.4 Insulating from interference

Argon is designed to reduce interference between workloads, allowing fair or weighted sharing with bounded loss of efficiency. To improve efficiency, Argon combines three techniques: quanta-based scheduling, aggressive amortization, and cache partitioning. Argon automatically configures each mechanism to reach the configured fraction of full standalone efficiency for each workload. Quanta-based scheduling also serves to improve fairness or provide controllable weights for the fraction of resources each workload receives. This section describes Argon's goals and mechanisms in more detail.

3.4.1 Goals and metrics

Argon provides both insulation and weighted fair sharing. *Performance insulation* means that efficiency for each workload is maintained even when other workloads share the server. That is, the I/O throughput a workload 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 the allowable loss in efficiency to be specified by a tunable *R-value* parameter, analogous to the R-value of thermal insulation. If the R-value is set to 0.9, a workload 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 (in other words, 45%). And, that efficiency (minimally) should be achieved no matter what other workloads 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 3.6 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 parameter.

Argon focuses on the two primary storage server resources, disk and cache, in insulating a workload's efficiency. It assumes that network bandwidth and CPU time will not be bottleneck resources. Given that assumption, a workload's share of server time maps to the share of disk time that it receives. And, within that share of server time, a workload'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 under sharing was defined earlier in Section 3.3.2 as the change, due to sharing, of the fraction of a request's service time spent actually transferring data to or from the disk media (useful work fraction). A workload's useful work level under sharing should be within the R-value of its level when not sharing the disk (in other words, its efficiency should be at least the R-value); for a given set of requests, the useful work fraction determines disk throughput.

Cache efficiency under sharing can be defined as the reduction, caused by sharing, of 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. When the cache is shared among workloads, no one workload is likely to be allocated the full cache; thus, its absorption rate may go down due to contention for cache pages. Many workloads' cache absorption rates are non-linear functions of how much cache space they receive, and different workloads may have dramatically different functions. As a result, some workloads may suffer a significant degradation in absorption rate, and thus efficiency and performance, with slight decreases in cache occupancy; while others may tolerate significant reductions in footprint with little or no effect. Because some workloads will suffer significantly when sharing the cache with others, cache efficiency cannot be maintained for every mix of workloads — unlike disk efficiency.

Argon's goal is to provide explicit shares (fractional allocations) of server time, together with an explicit guarantee of maintained efficiency. An alternative approach, employed in some other systems, is to focus on per-workload performance guarantees as the first-order goal instead of shares and efficiency. In storage systems, this is difficult when mixing workloads because different mixes provide very different efficiencies, confusing the feedback control algorithms used in such systems. Argon provides a predictable foundation that is significantly more conducive to stable control, as explored in Chapter 5. Atop Argon, a control system can manipulate the share allocated to a workload to change its performance, with much less concern about efficiency fluctuations caused by interactions with other workloads sharing the system. A control system based on this property is described in Chapter 5.

3.4.2 Overview of mechanisms

Figure 3.1 illustrates Argon's high-level architecture. Argon provides weighted fair sharing by explicitly allocating disk time among workloads in the appropriate proportions (relative time). Argon maintains efficiency by managing per-workload cache footprints and the combined disk request stream as needed to avoid the sources of efficiency-affecting interference mentioned above. In particular, each workload's disk efficiency is insulated by ensuring that disk time is allotted to clients in large enough quanta (absolute time) 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. Each workload's relationship between hit rate and cache size is then analyzed and appropriate cache sizes chosen for each so that the total efficiency loss for each workload, accounting for disk sharing overhead and the reduction in cache hit rates, is no more than allowed by the R-value. To ensure quanta are effectively used for streaming reads without requiring a queue of actual client requests long enough to fill

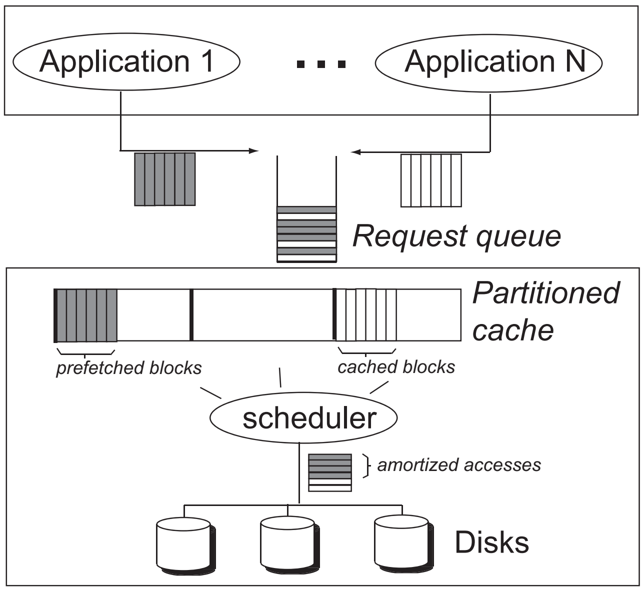


Figure 3.1: **Argon's high-level architecture.** Argon makes use of cache partitioning, request amortization, and quanta-based disk time scheduling. Requests from different applications are received over the network in a single queue, but with tags that differentiate the originating workload. The requests are first compared against entries in a partitioned cache for hits. If a request misses, then it proceeds to a scheduler which schedules requests from each workload in turn; the scheduler also performs prefetching into the appropriate cache partition for sequential workloads.

a full quantum at full utilization, Argon performs aggressive prefetching; to ensure that streaming writes efficiently use the quanta, Argon coalesces them aggressively in write-back cache space.

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, the mechanisms work best when they can assume properties that are guaranteed by the other mechanisms comprising Argon. For instance, timeslice-based scheduling is simplified when cache flushes on the critical path of a request x belong to the same workload as request x; cache partitioning guarantees this is the case. 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, to avoid misconfiguration and complexity, Argon automatically adapts each mechanism to ensure sufficient insulation, based on observed device and workload characteristics. For example, the ratio between disk transfer rates and positioning times has changed over time, and high streaming efficiency requires multi-MB prefetches on modern disks instead of the 64–256 KB prefetches of OSes such as Linux and FreeBSD. Argon is designed to automatically maintain efficiency on disks old and new, including disks made in the future, without administrator intervention.

3.4.3 Amortization

Amortization refers to performing large, uninterrupted disk accesses for streaming workloads. Because of the relatively high cost of seek times and rotational latencies whenever repositioning is required, amortization is necessary in order to approach the disk's streaming efficiency when sharing the disk with other workloads.

However, 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 3.6. But, because blocking will occur for all other pending requests as large prefetch or coalesced requests are processed, the maximum response time and the variance in response times may significantly increase. Thus, the prefetch and coalesced write sizes 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. The exact access size is automatically chosen based on disk characteristics and the

Disk	Year	RPM	Head Switch	Avg. Seek	Avg. Sectors/ 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 3.1: **SCSI/SATA disk characteristics.** Positioning times have not dropped significantly over recent years, but disk density and capacity have grown rapidly. This trend calls for more aggressive amortization.

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. The value T_{seek} is the expected 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.[1] In contrast to the other two terms which can be thought of as overhead, $T_{transfer}$ 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 3.1, 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 $9*(5\,\text{ms} + 6\,\text{ms}/2) = 72\,\text{ms}$. Ignoring head switch time, $\sim 72\,\text{ms}/T_{rot} = 12$ tracks must be read, which is 4.8 MB. Each number is higher on a typical SATA drive.

[1]Only a small minority of current disks appear to 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.

There is an interaction between disk request size policies and cache policies. In particular, multiple-MB sequential accesses require that the storage server dedicate multiple-MB chunks of the cache space for use as speed-matching buffers. Argon coordinates between the disk scheduler and the cache manager to make appropriate reservations when sequential accesses are detected.

Reads and writes 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; efficiency simply requires that request sizes be large in either case. However, the process of generating large requests differs for reading and writing. Write coalescing is straightforward because when a client sequentially writes a file, the blocks are staged into a write-back cache and when it is time to flush a block, the entire set of contiguous dirty cache blocks of which it is a member can simply be sent in a large group (MBs) to the disk. In contrast, read prefetching is speculatively performed when a client appears to be sequentially reading a file. However, because the blocks must be read before they are consumed, and future accesses are not known with certainty, the strategy for determining when continued sequential accesses are likely must be more sophisticated to minimize the useless prefetching of blocks that are ultimately not needed.

3.4.4 Cache partitioning

Cache partitioning refers to explicitly dividing up a server's cache among multiple workloads in a manner that provides complete isolation between the respective partitions. Specifically, if Argon's cache is split into n partitions among workloads $W_1, ..., W_n$, then W_i's data is only stored in the server's i^{th}

cache partition, irrespective of each workloads' request patterns. Instead of allowing excessive cache occupancy for some workloads to arise as an artifact of access patterns and the cache replacement algorithm, cache partitioning preserves a specific fraction of cache space for each workload.

It is often not appropriate simply to 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 full standalone efficiency. Conversely, large streaming workloads require only a small amount of cache space to buffer prefetched data or dirty write-back data, and workloads with little reuse do not benefit from cache beyond a small number of pages for staging. 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 full 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 desired R-value to compute the required I/O absorption rate (using 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 while it is running. A cache simulator replays these traces using the server's cache eviction policy to calculate the I/O absorption rate for different hypothetical cache partition sizes. Figure 3.2 depicts example cache profiles for three benchmark workloads. In the figure, the total server cache size is 1024 MB. 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. Not all workloads have this characteristic, however; TPC-H Query 3 can achieve a similar I/O absorption rate to its standalone value with only a small fraction of the full cache space. If both the TPC-C

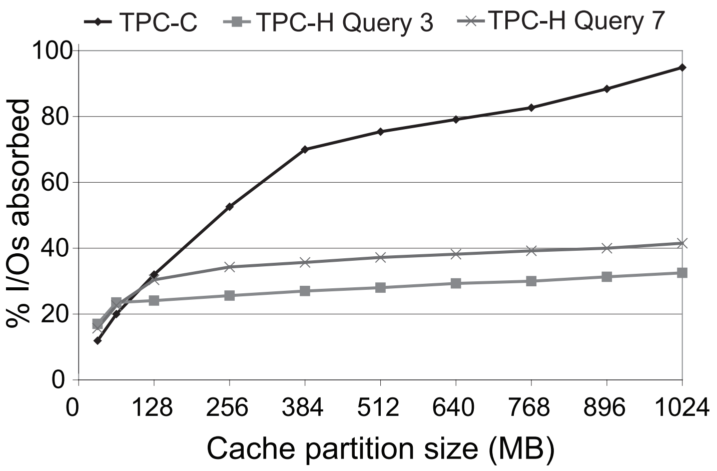


Figure 3.2: **Cache profiles.** Different workloads have different working set sizes and access patterns, and hence different cache profiles. For three database workloads, the relation between cache size and I/O absorption percentage (defined as the fraction of requests that do not go to disk) is shown. Requests that do not go to the disk include read hits and overwrites of dirty cache blocks.

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 following discussion, 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 workload i. The value S_i is modeled 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). The value p_i is estimated as a function of the workload's cache size, as described in the first step. The value S_i^{DISK} is continuously tracked per workload, as described in Section 3.5.4. The server throughput equals $1/S_i$, assuming no concurrency.

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

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

A workload must realize nearly its full standalone throughput in its share of time in order to maintain 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 equally 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.

3.4.5 Quanta-based scheduling

Argon's scheduler controls when each workload's requests are sent to the disk firmware (as opposed to performing "disk scheduling," such as SPTF, C-SCAN, or elevator scheduling, which reorders requests for performance rather than for insulation; Argon expects that this level of disk scheduling occurs underneath Argon's own scheduler in the disk's queue, implemented in the disk's firmware).

Scheduling is necessary for three reasons. First, it ensures that a workload receives exclusive disk access, which is required to maintain the large uninterrupted requests intended to be performed for sequential workloads. Second, it ensures that disk time is appropriately divided among workloads in the fractions desired. 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 (at the beginning of a timeslice), Argon issues requests that have been queued up waiting for that workload's quantum to begin. If more requests are queued than the scheduler believes will be able to complete in the quantum, however, 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 immediately 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.

A quanta-based scheduling regime can eliminate much of the inefficiency caused by sharing, but leaves two remaining sources. 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), the act of draining itself 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.

3.5 Implementation

We implemented the Argon storage server to test the efficacy of our performance insulation techniques, as a component in the Ursa Minor cluster-based storage system [2] which exposes an object-based interface [38]. 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.

The techniques of amortization and quanta-based scheduling are implemented on a per-disk basis. Cache partitioning is done on a per-server basis. 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.

3.5.1 Distinguishing among workloads

To distinguish among workloads, operations sent to Argon include a client identifier. "Client" refers to a workload, 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. The defining rule is that a client is the entity to which efficiency and proportional share guarantees are provided; these guarantees apply to the set of requests tagged with the corresponding client identifier, whatever host they may come from.

3.5.2 Amortization

To perform read prefetching, Argon must first detect that a client is performing a 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* many 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. If it finds any, 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; all blocks are flushed to disk in the background (within the corresponding workload's quanta).

3.5.3 Cache partitioning

Recall from Section 3.4.4 that the cache partitioning algorithm depends on knowledge of the cache profile for a workload. The cache profile captures the relationship between the cache size given to a workload and the I/O absorption rate. Argon collects traces of a workload's accesses when the workload is first added. It then processes those traces using a simulator, while the workload continues to run, to predict the absorption rate with different 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 overhead (5-6%) on foreground workloads in the system [51]. Once sufficient traces 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 under 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 we could not adequately capture them using analytical formulas. We have observed that for cache hits the simulator and real cache manager need similar times to process a request. But, 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% [50].

Another implementation issue is dealing with *slack* cache space, the cache space left over after all workloads have taken their minimum share. In our implementation, 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 [55] 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.

3.5.4 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 actual 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 for 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 [12, 23]. 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 (at the cost of introducing extra inefficiency). Argon takes a middle approach — a client's scheduling quantum is skipped if the client has been idle for its last k consecutive scheduling quanta. (Argon leaves k as a manual configuration option, set to 3 by default.)

3.6 Evaluation

This section evaluates the Argon storage server prototype. First, we use microbenchmarks as minimal test cases to demonstrate the causes of performance problems arising from storage server interference, and Argon's effectiveness in mitigating them. (Microbenchmarks allow precise control of the workload access patterns and system load.) Second, macrobenchmarks illustrate the real-world efficacy of Argon.

3.6.1 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 3.1 for their characteristics). One disk stores the OS, and the other stores the objects (except in one experiment which uses two separate disks to store two separate workload's objects to focus only 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.

Of course, more sophisticated storage systems are built using techniques such as clustering and RAID. Chapter 4 explores the efficacy of Argon's techniques in a clustered storage system; Section 5.6.2 of Chapter 5 demonstrates results on RAID arrays.

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.

3.6.2 Microbenchmarks

This section illustrates microbenchmark results obtained using both Linux and Argon. These experiments underscore the need for performance insulation. We show results from Argon because they allow us to compare performance before and after we add performance insulation techniques to our server. We show results from Linux to illustrate that the interference problems we observe in our system are not unique, but rather common in widely-deployed systems. Note that we have not exhaustively traced through each layer of the OS to determine exactly which implementation decisions in Linux detract from efficiency or whether they can be remedied easily or only with great difficulty. Our goal is to highlight the set of mechanisms and policies needed to provide performance insulation through an implementation in our storage system, while showing that the efficiency challenges we address are not an artifact of our system by confirming they also exist in Linux. The techniques we identify could also, in principle, be implemented in Linux.

Microbenchmarks 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.[2] 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 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 to achieve a desired working-set size.[3]

[2]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.

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

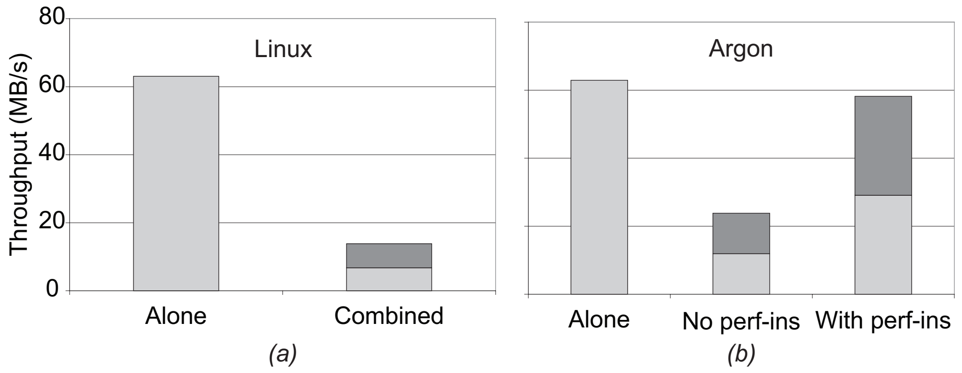


Figure 3.3: **Throughput of two streaming read workloads in Linux and Argon.**

Amortization

Figure 3.3(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 a degree of prefetching.

Figure 3.3(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 in aggregate.

Figure 3.4 shows, for each scenario running under Argon, the CDF (cumulative distribution function) of response times for one of the streaming read workloads. (The other workload exhibits a virtually identical distribution because the two workloads are essentially 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.[4] 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 95 ms. This increases the variance in response time, while the mean, 90^{th} percentile, and 95^{th} percentile response times decrease.

Cache partitioning

Figure 3.5(a) demonstrates performance degradation due to cache interference in Linux. A non-streaming workload (Workload 2) has a potential full-cache absorption rate of 50%. However, a streaming workload (Workload 1) has much higher throughput. Thus, more of its requests are handled by the system in a unit of time, and the recency-based cache replacement policies used by both Linux and Argon (which implements a form of LRU) allow it to consume most of the cache. As a result, Workload 2's performance is significantly degraded because it loses its cache occupancy and thus its hit rate. To focus on only the cache partitioning problem, both workloads share the same cache, but their misses are routed to separate disks. The throughput of Workload 2 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 its disk, is being shared. (We believe that

[4]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.

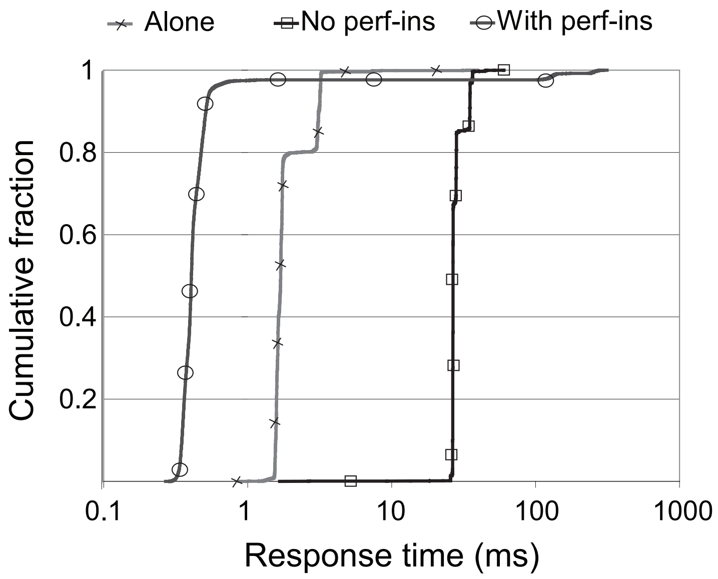


Figure 3.4: **Response time CDFs for two streaming workloads.** When running alone, the average of response times is 2.0 ms and the standard deviation is 1.17 ms. When the two workloads are mixed without performance insulation, the average for each is 28.2 ms and the standard deviation is 3.2 ms. With performance insulation, the average is 4.5 ms and the standard deviation is 27 ms.

the small decrease in the streaming workload's performance is an artifact of a system request handling bottleneck.)

Figure 3.5(b) shows the benefits 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 (when the desired R-value is set 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.

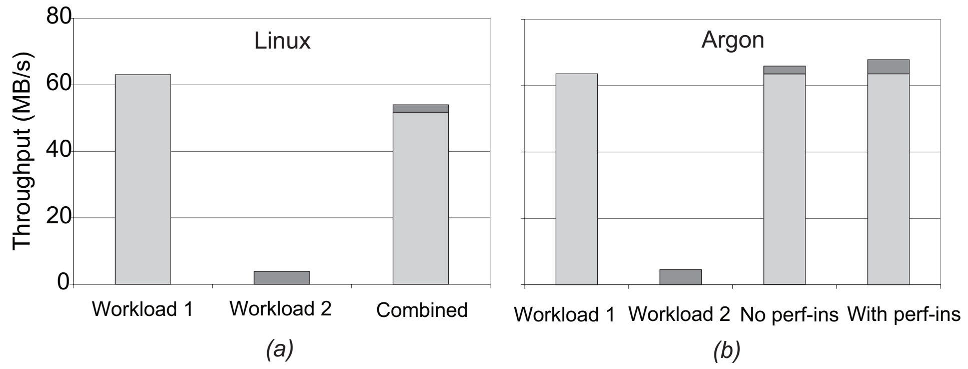


Figure 3.5: **Effects of cache interference in Linux and Argon.** The standard deviation is at most 0.55 MB/s for all Linux runs and less than 5% of the average for the Argon runs.

Quanta-based scheduling

Figure 3.6(a) shows the performance degradation due to unfair scheduling of requests in Linux. Two non-streaming workloads, one that maintains 27 requests outstanding at all times (Workload 1) and one with just a single request outstanding at a time (Workload 2), compete 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 3.6(b) shows a benefit of quanta-based disk time scheduling in Argon. The version of Argon with performance insulation disabled has 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 only slightly less than 50% of their unshared throughput, exceeding the R = 0.9 bound.

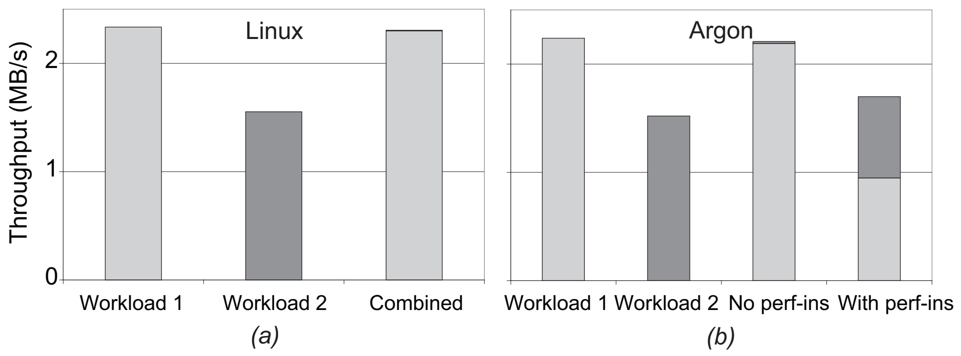


Figure 3.6: **Need for share-based scheduling in Linux and Argon.** The standard deviation is at most 0.01 MB/s for all Linux runs and at most 0.02 MB/s for all Argon runs.

Proportional scheduling

Figure 3.7 shows that the sharing of an Argon server need not be fair (i.e., equal across all workloads); the proportion of resources assigned to different workloads can be adjusted to meet higher-level goals. In this experiment, the same workloads as in Figure 3.6(b) are shown, but the system is configured so that the workload with one request outstanding (Workload 2) receives 75% of the server time, and the workload with 27 requests outstanding (Workload 1) receives only 25%; with this configuration from the administrator or management tool, quanta sizes are proportionally sized by Argon's scheduler to achieve this. Amortization and cache partitioning can similarly be adapted to use weighted priorities.

Combining sequential and random workloads

Table 3.2 shows the combination of Argon's 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 is sized so that it has a negligible hit rate. Without performance insulation, the workloads receive 2.2 MB/s and

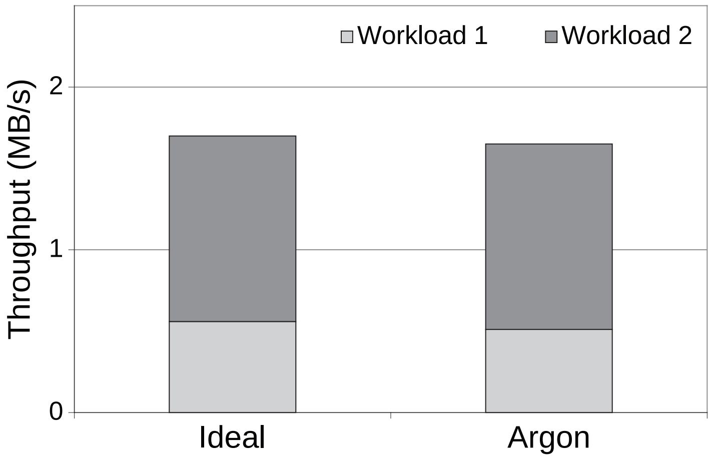


Figure 3.7: **Scheduling support for two random-access workloads.** With the same workloads as Figure 3.6(b), scheduling can be adjusted so that Workload 2 receives 75% of the server time.

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 full standalone efficiency, as desired.

Figure 3.8 shows the CDF of response times for both workloads. The sequential workload, shown in Figure 3.8(a), exhibits the same behavior shown in Figure 3.4 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 3.8(b). Running alone, it had a range of response times, with none exceeding 26 ms. The 90^{th} percentile was at 13.7 ms. Virtually all values were above 3 ms. Running together with the sequential workload, response times increased; they ranged from 6–60 ms with the 90^{th} 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.

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 3.2: **Amortization and scheduling effects in Argon.** When a sequential and a random workload run together, performance insulation results in much higher performance for both workloads. Standard deviation was less than 6% for all runs.

Adjusting sequential access size

Figure 3.9 shows the effect of prefetch size on efficiency. Two streaming workloads, each with an access size of 64 KB, were run with performance insulation. The performance each receives is similar, hence only the throughput of one of them is shown. In isolation, each of these workloads receives approximately 62 MB/s, thus 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 beyond 32 MB neither improve nor degrade performance significantly.

Adjusting scheduling quantum

Figure 3.10 shows the result of a single-run experiment intended to measure the effect of the scheduling quantum (the amount of disk time scheduled for one workload at a time before moving on to the other workloads) on efficiency. For simplicity, we show quanta measured in terms of the number of requests in 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

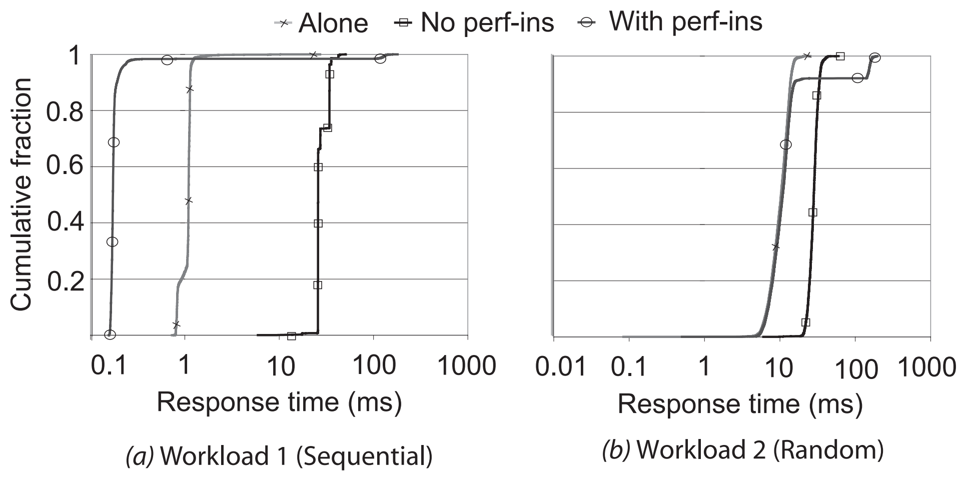


Figure 3.8: **Response time CDFs for a sequential workload and a random workload.** The standard deviations of response times for the sequential (a) and random-access workloads (b) when they run alone are 0.316 ms and 2.72 ms respectively. The random-access workload's average response time is 10.3 ms. When the two workloads are mixed without performance insulation, the standard deviations of their response times are 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 deviations are 15.87 ms and 39.3 ms respectively. The random-access workload's average response time is 21.9 ms.

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 a quantum of 32 requests. We observed that further increases in quantum size beyond 128 neither improve nor degrade performance significantly.

3.6.3 Macrobenchmarks

To explore Argon's techniques on more complex workloads, we ran two database benchmarks, TPC-C (an OLTP workload) and TPC-H (a deci-

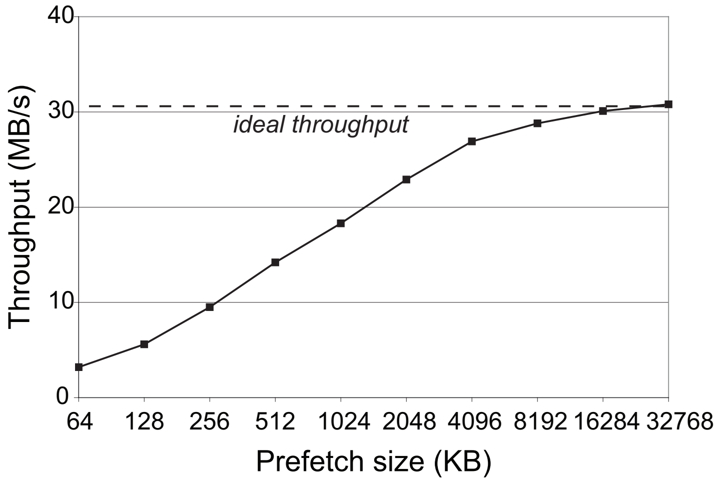


Figure 3.9: **Effect of prefetch size on throughput.**

sion support workload), using the same storage server. The combined load on the server 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.

The TPC-C workload mimics an on-line database performing transaction processing [52]. 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 3.2.

TPC-H is a decision-support benchmark [53]. 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 3.2.

Figure 3.11 shows that without performance insulation, the throughput of both benchmarks degrades significantly. With an R-value of 0.95, Argon's

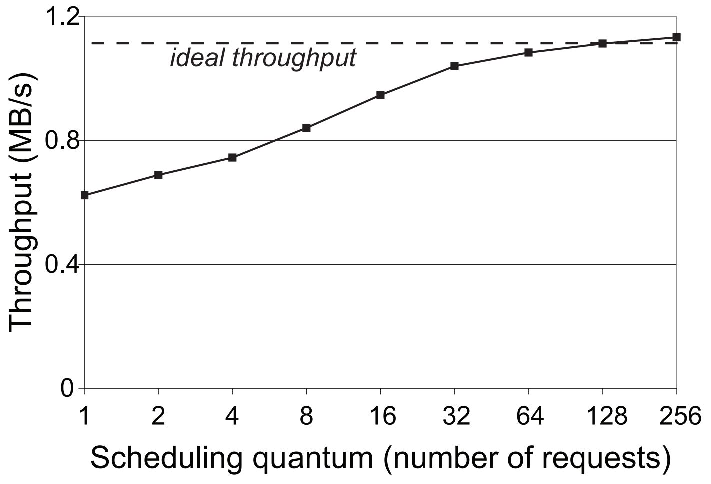


Figure 3.10: **Effect of scheduling quantum on throughput.**

insulation significantly improves the performance for both workloads. Figure 3.12 examines the run with TPC-H Query 3 more closely, showing how much each of the three techniques, scheduling (S), amortization (A), and cache partitioning (CP) contribute to maintaining the desired efficiency.

3.6.4 Related work

Argon adapts, extends, and applies mechanisms that have been used in the past for other purposes, to provide performance insulation for shared storage servers. This section discusses previous work on these mechanisms and on similar problems in related domains. It does not attempt to cover the general space of related work in storage QoS, as that is described in more detail in Chapter 2.

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

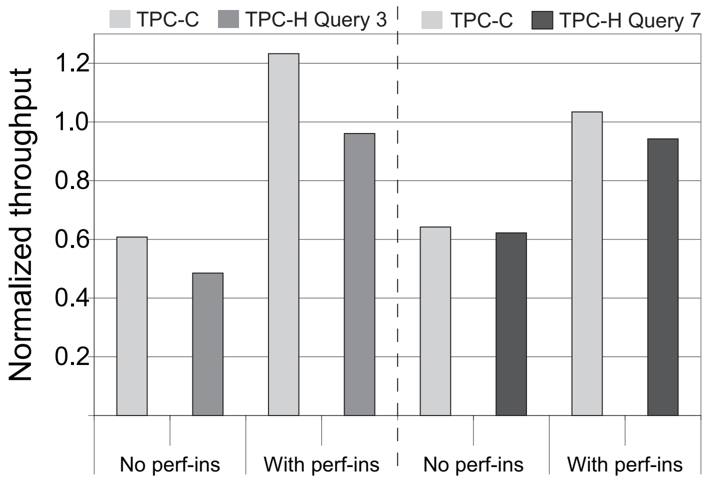


Figure 3.11: **TPC-C and TPC-H running together.** The throughput received by two database workloads is shown: on the left, TPC-C runs with TPC-H Query 3, on the right, it runs with TPC-H Query 7. The normalized throughput with and without performance insulation in shown. The normalization is done with respect to the throughput each workload receives when running alone, divided by two. Thus, a workload receiving a half share of time with perfect efficiency would receive a normalized throughput of one and, for an R-value of 0.9, at least 0.9.

prefetch [45]. The popular 64 KB prefetch size was appropriate more than a decade ago [36], but is now insufficient [43, 48]. 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. [47, 48] show how to obtain and exploit underlying disk characteristics to achieve good performance with certain workload mixes. In particular, they show 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.

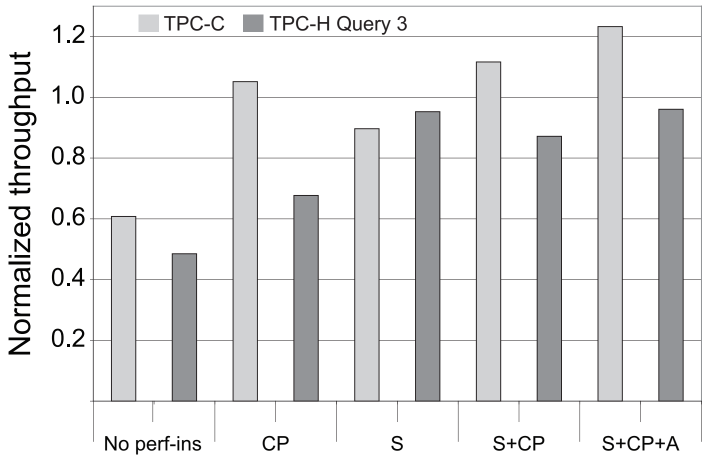


Figure 3.12: **All three mechanisms are needed to achieve performance insulation.** The different techniques are examined in combination. "CP" is cache partitioning, "S" is scheduling, "A" is amortization; the full version of Argon uses all of them. No combination short of all three is sufficient. Throughput is normalized with respect to the throughput each workload receives when running alone, divided by two; thus it matches achieved efficiency.

Most database systems explicitly manage their caches in order to maximize their effectiveness in the face of interleaved queries [10, 13, 47]. 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 [13]. Cao et al. [8, 7] show how these ideas can also be applied to file systems in their exploration of application-controlled file caching. In other work, the TIP [45] 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 necessary cache partition size for each workload based on its access pattern.

Timeslicing

Some prior work also performs timeslicing 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, 11, 30, 40] 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.

3.7 Discussion: Intended applications

This section revisits the target workloads and environments for this chapter.

Argon is designed for single-server storage systems; the next chapter describes and remedies the issues that preclude the use of Argon in a clustered storage system. As described in this chapter, Argon controls access to disk heads, thus we must implement our techniques at a layer that is aware of request-to-disk mappings. In addition, Argon's cache partitioning requires that we can control the caching policy. Hence, Argon is not suitable for implementation, for instance, external to a complex storage array.

Due to Argon's cache tracing, a workload that does not have enough activity for Argon to detect its access patterns quickly will not benefit from Argon's adaptive cache sizing approach. If such workloads are very low throughput, but expect a high cache hit rate, they will not be provided the expected insulation from Argon. In addition, as described, Argon detects cases where workloads cannot co-exist because they cumulatively require more cache than is available on the server; full insulation cannot be provided in this case, but the problem can be identified and communicated to the administrator or higher-level control system. Another limitation is on the number of workloads; if more sequential workloads are executing on a server than can be accommodated in the server's cache, given the prefetch size and the number of workloads, then full insulation cannot be provided for lack of memory.

Argon provides efficiency in terms of bandwidth and throughput. Timeslicing increases efficiency, and thus decreases mean latency compared to not providing quality of service, and increases mean latency for closed-loop workloads, compared to running alone, in a bounded manner. Timeslicing increases maximum latency (for a potentially small subset of requests) significantly, and similarly increases variance. Argon is not suitable for workloads requiring hard real-time maximum latency guarantees, or that are otherwise sensitive to these latency effects.

3.8 Conclusion

Storage performance insulation can be achieved when workloads share a storage server. Traditional disk and cache management policies do a poor job, allowing interference among workloads' 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 workload with a configurable fraction (the R-value; e.g., 0.9) of the full efficiency it would receive without competition. So, with fair sharing, each of n workloads will achieve no worse than R/n of its standalone throughput. This increases both efficiency and predictability when workloads share a storage server.

Argon's insulation allows one to reason about the throughput that a workload will achieve, within its share, without concern for what other workload do within their shares. Workloads 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 administrators the latter, so that a control system can place such workloads' datasets on distinct storage servers.

4 Coordinating among servers in a cluster

The previous chapter discussed how to maintain efficiency when a storage system composed of a disk and a cache is shared among workloads. One way to scale storage up, in terms of capacity and available bandwidth, is to cluster together a set of relatively modest nodes into a combined system. This approach has been gaining appeal over time. Such *clustered storage systems* potentially represent a simpler and more cost-effective way to build a large storage system, because with redundancy, no one server is essential to the functioning of the system. Thus, components need not be engineered to the same extreme level of reliability as would be required in a monolithic system. Clustered systems may also provide the benefits inherent in spreading out work, such as fewer central bottlenecks and easier growth.

Providing efficiency and effective performance insulation is desirable for such systems. This chapter describes why the approaches described in the previous chapter are not effective on their own when applied in this domain, and investigates the extensions need to Argon's base techniques to bring insulation to clustered storage.

4.1 Introduction

Cluster-based storage systems must be designed to spread the work associated with handling a request across a set of servers, rather than making it the exclusive responsibility of a single node. This approach is used to allow the performance and reliability of multiple modestly-performant and modestly-engineered nodes to be constructively added. There are different schemes for partitioning the work associated with a single request among a set of nodes. One simple scheme, known as striping, divides the request into equal-sized pieces (*fragments*) and sends each to a different server. More complicated schemes, such as erasure coding, add redundancy to this design.

Our goal is to construct a cluster-based storage system that provides the same guarantees as Argon, a standalone server. A naïve approach might be to just "run Argon" as each of the individual servers; each server would provide guaranteed efficiency for the fragments it is storing. The question that arises, however, is *how do these per-fragment efficiency guarantees compose into the block-level guarantees desired for a workload*?

Disk-head timeslicing is one of the primary techniques used by Argon to maintain efficiency and provide performance insulation. Unfortunately, however, it faces difficulties for cluster-based storage. When data is striped across multiple servers, a client read request requires a set of responses which must be reassembled into the complete block. Until all are received, the read request is not complete. Since each server only acts on a disk request within the associated workload's disk head timeslice, the client will end up waiting for the last (furthest-in-the-future) of the timeslices. If the relevant disk head timeslices are not scheduled simultaneously, the delay could be substantial (5–7x in our experiments). Worse, if the disk head timeslices do not all overlap, the throughput of a closed-loop one-request-at-a-time workload will be one request per round of timeslices.

Providing performance insulation for cluster-based storage requires *co-scheduling* of a workload's disk head timeslices across the servers over which its data is striped (arranging all timeslices so that those belonging to the same workload begin and end at the same wall-clock time across each of servers it uses). If all data is striped across all servers, the system needs to use an identical ordering of disk head timeslices across the servers, handle striped requests in the same timeslices, and synchronize the timing of timeslice switches. For cluster-based storage systems that allow different volumes to be striped differently (e.g., over more servers or over different servers), there is the additional challenge of finding a schedule for the cluster.

This chapter describes a cluster-based storage system that guarantees R-values to its workloads, extending the single-server mechanisms from the previous chapter with explicit co-scheduling of disk head timeslices. It uses standard network time synchronization and synchronizes "time zero" for the disk head time scheduler. It implicitly coordinates the work done by each server in co-scheduled timeslices, while allowing local request ordering decisions within timeslices. It assigns workloads to subsets of servers and organizes their timeslices to ensure that each striped workload's disk head timeslices are co-scheduled. Finding an assignment that works is an NP-complete problem, but there exist heuristics that allow solutions to often be found quickly. Although each of the heuristics yields quick answers in only 40–80% of cases, we find that running several in parallel yields a quick solution in over 95% of cases. When a quick solution cannot be found, introducing a small co-scheduling efficiency reduction helps. In such cases, the system compensates by increasing the per-server R-value used in automatically configuring per-server resource allocations. Our experiments confirm that, by adopting a global timeslice schedule found with heuristics and ensuring the servers are synchronized as they follow it, workloads receive the insulated efficiency expected.

4.2 Intended applications

This section describes the workloads and storage systems for which the techniques in this chapter are appropriate, and those for which the techniques are not suitable. Section 4.7 revisits this discussion at the end of the chapter, indicating why these limitations exist and to what extent they may be remediable with further refinement.

The approaches discussed in this chapter are effective for storage clusters sized to accommodate at least dozens of workloads. Growth of processor speeds through frequency increases or more cores per processor will scale this limit over time. If the storage servers in a cluster have significant computational power available during the workload provisioning process, then this resource can be used to accommodate significantly more workloads as well.

The approaches described are most suitable for long-lived workloads that require a temporally consistent resource allocation on their respective servers, and, further, for environments with generally the same set of workloads over time. An example of such an environment would be enterprise / data center applications.

Since each server in the cluster runs Argon, the limitations of Argon also apply.

4.3 Issues with timeslicing

With striping, a client request for a block translates to a fragment request at each server. The client request is not fulfilled until all its fragments are read and combined back into the original block. Unfortunately, the throughput for a block is not simply the sum of the throughputs for its fragments. While that sum is an upper bound, the response time observed by a client is that of the slowest server to respond.[1] For closed-loop workloads, slower response times directly reduce throughput.

Timeslicing can cause significant response time differences across servers in two ways. First, the list of workloads, ordering and length of timeslices, and overall round length (time before the schedule of timeslices repeats) may not match across the servers. Without arranging the schedules for each of the servers that a workload uses, its timeslices may not consistently “line up” across the servers (Figure 4.1). Second, even if the schedules are conducive to co-scheduling the timeslices of each workload, the phase of the servers (i.e., the point in wall-clock time when the round-robin order restarts) may not match (see Figure 4.2). This can occur either because no attempt was made to synchronize the servers, or because they diverged over time.

The effects of non-synchronization are drastic and immediate. Figure 4.3 shows that timeslicing alone is effective on a single server, but becomes detrimental (even worse than not performing any performance insulation techniques at all) when a workload is striped across two or more servers. A single-threaded workload only completes one request per round because

[1]With a more flexible data distribution scheme — such as erasure coding — the client can read from any m of the n servers storing its data. While the request response time may no longer be that of the slowest server, it is still limited by the $n - m + 1^{st}$ slowest server.

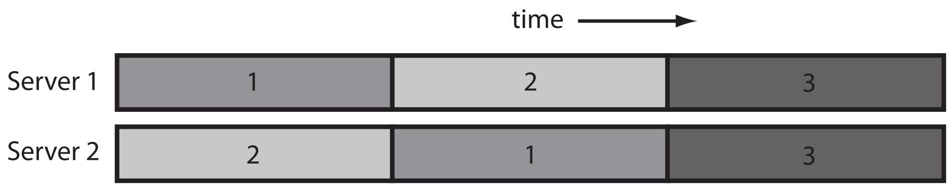


Figure 4.1: **Mismatched schedules.** If the ordering of workloads does not match across the servers, then a client must wait for the latest occurrence of its timeslice across all the servers before its request completes.

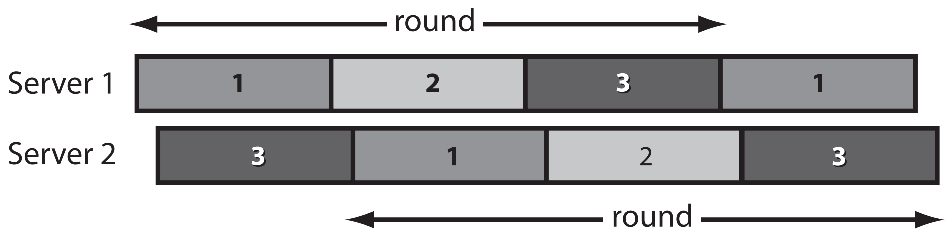


Figure 4.2: **Out-of-phase servers.** If the phase of identical rounds does not match across the servers, then a client must wait for the most-in-the-future occurrence of its timeslice across all the servers before its request completes.

there is no overlap between the corresponding timeslices at the servers — by the time the request completes at the second server, the timeslice is over at the first server.

To realize the insulation benefits of timeslicing in a cluster, it is necessary to find a cluster-wide schedule that co-schedules timeslices for each workload across all servers it uses. In addition, the servers must adhere to that schedule in a synchronized fashion — that is, timeslices must begin and end at the same time on the servers. Note that timeslice synchronization does not require that individual request scheduling decisions within timeslices be synchronized; thus, only minimal coordination is needed, and traditional local control of request ordering is acceptable. Sections 4.4 and 4.5 describe and evaluate our solutions to finding schedules and coordinating servers, respectively.

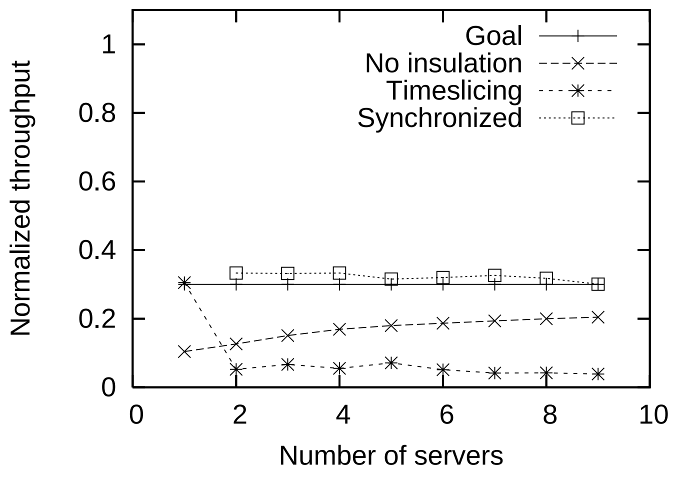


Figure 4.3: **Moving beyond one server.** Three workloads share a server and the throughput of one, which has been allocated a one-third share of disk time, is graphed. Without performance insulation, the other workloads interfere and the goal is not met. Timeslicing disk head time solves the problem on one server. But when the workloads are striped across servers, timeslicing becomes ineffective unless the timeslices are synchronized. (The y-axis is normalized against the throughput the depicted workload receives running alone on the corresponding number of servers.)

4.4 Designing a schedule

To minimize administrator effort, we wish to have an automated means of creating timeslice schedules. This section describes and evaluates algorithms for schedule design.

4.4.1 Problem specification

In formulating algorithms, we assume a cluster of homogenous storage servers with one disk each.[2] Machines that have significantly different re-

[2]Machines that have multiple, independent disks can be thought of as separate machines in the context of this problem. From our perspective, machines with disk arrays can be treated as having a single, bigger disk. Experiments shown in Section 5.6.2 confirm that our techniques work as expected on arrays.

sources (i.e., disrupt homogeneity) should be placed in a separate cluster with like machines.[3]

Workloads are specified using two numbers: number of servers and share of servers. For instance, a workload may be striped across five servers and, to achieve the desired level of performance, need at least a one-third share of time on each of the five servers.

Finding a suitable schedule for the cluster amounts to finding round-robin timeslice orderings for each of the servers so that each workload receives its share of total time at the required number of servers, and each workload's timeslices are co-scheduled. The system may select whichever servers are convenient for a given workload.[4]

Finding a solution should be relatively efficient (e.g., a few minutes or less), but need not be extremely fast, in practice. Timeslice schedules in Argon are long-lived because they are a function of the set of workloads, not of specific requests. A new schedule need be found only when workloads enter the cluster, or when the fraction of server time assigned to a workload is changed. Adding a new workload involves the substantial task of adding a new dataset, which makes it a time-consuming operation already. Rescheduling due to workload removal need not be fast, since the remaining workloads will remain appropriately scheduled. Thus, our target is algorithms that can find solutions within a few minutes. Nonetheless, our algorithms parallelize very well and can be completed in seconds when spread over multiple CPUs.

Figure 4.4 shows an example input list of workloads and an example solution to the problem.

[3]If a set of machines has minor variations, however, they may be clustered together and treated as all having the "lowest common denominator" of performance. Thus, for example, minor differences in disk characteristics (e.g., due to in-field replacement) are fine.

[4]There may be additional constraints, such as not exceeding the storage capacity available at a server, or preferring certain placements due to network topology, that we do not consider here.

Total servers = 7

Num servers	**Proportion**
3	1/2
3	1/2
1	1/2
1	1/6
1	1/3
5	1/3
7	1/6

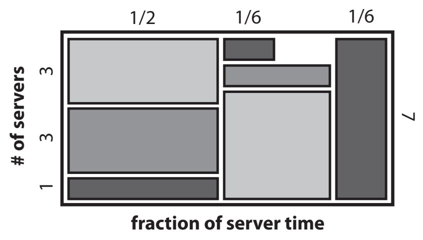


Figure 4.4: **Example problem instance and solution.** On the left is an example input to the scheduling algorithm; on the right is one possible solution. Rectangles correspond to workloads, with their height corresponding to the number of servers and their vertical location corresponding to which servers to use; their width corresponds to share of time, and their horizontal location corresponds to the span of time during which their timeslices are scheduled. The enclosing rectangle represents a single round in the cluster; the schedule is repeated indefinitely.

4.4.2 Geometric interpretation

This problem may be recast as a geometric problem, as suggested by Figure 4.4. Consider each workload as a rectangle whose height is the number of servers it needs, and whose width is the fraction of time it needs on each of those servers. Consider a larger rectangle, whose height is the total number of homogenous servers in the cluster, and whose width is one (corresponding to the full share of time on a server). Can the set of smaller rectangles be placed into the larger rectangle without overlapping or rotating any of the smaller rectangles, or exceeding the boundaries of the larger rectangle? If so, then an appropriate schedule exists, and the rectangle placements can be directly translated into a suitable timeslice schedule.

This geometric problem is known as the *strip-packing problem* or the *cutting-stock problem* [17]. It has been studied in industrial settings where a larger piece of material, such as wood or glass, must be cut into smaller pieces to manufacture a product. Unfortunately, it is known to be NP-complete (for instance, it is a more general version of the bin-packing problem). The above formulation is the *decision* version of the problem, which asks *can the rect-*

angles all fit in a larger rectangle of a given size? The *optimization* version asks *what is the minimum width of the larger rectangle, for a fixed height, such that all the rectangles fit?*[5] An *optimal* solution of a given problem (in our choice of dimensions) is a packing of rectangles that uses the minimal possible width.

For our purposes, a solution that has a width of at most one is sufficient to provide all workloads their requested share of the server. It may be preferable to find a solution that is even narrower, if one exists, because it would give the workloads extra server time; but this is not necessarily to meet the basic insulation goals.

4.4.3 Related work

This chapter discusses the application of the strip-packing problem to cluster-based performance insulation. This section provides theoretical background on the problem in more detail; Argon, on which this chapter builds, is covered in the previous chapter, and the general space of storage QoS is covered in Chapter 2. A later section (Section 4.6) discusses other related work specific to this chapter.

Strip packing has been explored by theoreticians seeking ways to accelerate exhaustive searches for the optimal solution, for approximation algorithms that can provide solutions within a guaranteed distance from the optimal, solution, and for heuristics that may find "acceptable" solutions quickly.[6]

Exhaustive search

As with any NP-complete problem, one can enumerate all possible solutions and test whether they meet the requirements or whether they are the best solution seen up to that point (yielding the best possible solution at the end

[5]The problem is usually specified for a fixed width and variable height, but we reverse the dimensions because it works more naturally for our problem.

[6]By "acceptable," we mean solutions that are not even guaranteed to be necessarily close to optimal, but might be subjectively good enough.

of the exhaustive search). However, this approach results in an exponential run time that will be prohibitive for sufficiently large problems.

In the case of strip packing, a naïve search might consider placing a rectangle at each possible coordinate location.[7] However, Fernandez de la Vega and Zissimopoulos [16] show that it suffices to consider a smaller (but still exponential) set of possible solutions. If there is a solution, then there exists a *left-bottom justified* solution — essentially, one in which there are no gaps. Without loss of generality, a solution with no gaps can be built by placing rectangles one after the other in some order, such that the next rectangle in the list is placed with its left side touching another rectangle or the edge of the enclosing rectangle, and its bottom side touching a rectangle or the bottom of the enclosing rectangle. The authors show that, for a fixed order of placing rectangles, there is no strategic advantage to leaving a "gap" to fill later. Thus, it suffices to examine all possible orderings of the rectangles, and for each of those orderings, only the placements of successive rectangles that are at *active corners*, i.e. touching other rectangles or the boundaries. This strategy still results in an exponential search space, however.

We have used the exhaustive method to solve synthetic problem instances described later in Section 4.4.6. The amount of time taken to find a solution or determine that one does not exist is shown in Figure 4.5 and in log scale in Figure 4.6. Each line represents initializing the exhaustive search with a different initial guess, and then successively permuting that initial guess to eventually explore the entire solution space, as explained later. Search times grows exponentially with the number of workloads, regardless of which starting point is used. Beyond thirteen workloads, exhaustive search is impractical. For instance, one fourteen-workload instance took about 6 hours. This motivates the need to find more efficient and effective ways to create schedules, even for relatively modest numbers of workloads.

Another technique for reducing the size of the search space further is the branch-and-bound method. Martello *et al.* [35] used branch-and-bound to solve or approximate solutions to a set of benchmark problems with around

[7]When we refer to *placing* a rectangle at a coordinate location, we mean placing a specific corner at that location.

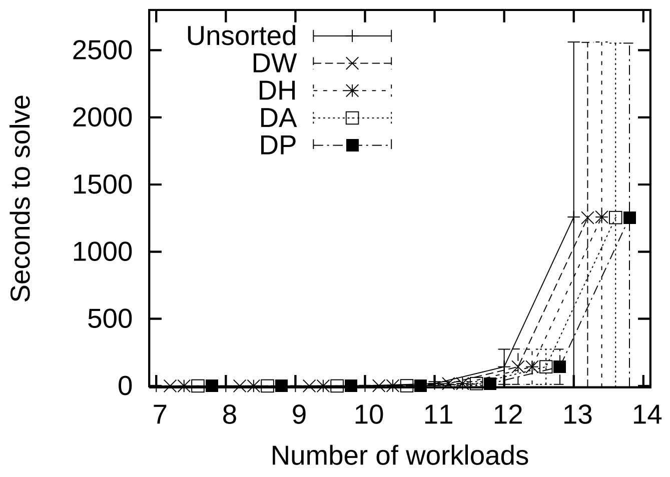


Figure 4.5: **Exhaustive search and heuristics.** Exhaustive search of the solution space takes impractically long when there are more than a few workloads. Initializing the search with one of the four heuristics (Decreasing Width, Decreasing Height, Decreasing Area, or Decreasing Perimeter) does not remedy this problem; growth is exponential throughout the range plotted. In this graph, four of the lines are slightly offset in the horizontal direction to make them distinguishable. Error bars show one standard deviation in either direction.

two dozen rectangles on the order of minutes to an hour on an 800 MHz Pentium III.

Approximation algorithms

Approximation algorithms exist [16, 26] that are able to find solutions within $(1 + \epsilon)$ of optimal in time that is polynomial in the number of rectangles but exponential in other variables. Unfortunately, for our problem sizes, we believe that no point along the runtime-vs.-ϵ tradeoff will be acceptable (too much error is introduced and runtime remains high). Approximation algorithms for strip packing work by subdividing the large rectangle into smaller rectangles, packing subsets of the rectangles into each of these smaller problem instances, and then "gluing" the smaller instances back into a larger

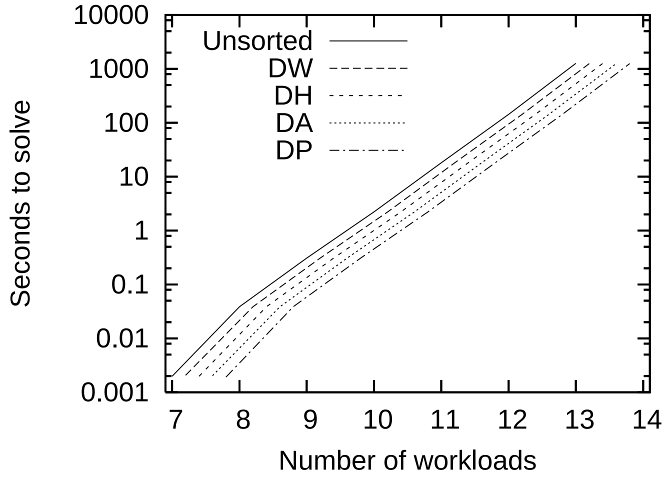


Figure 4.6: **Exhaustive search and heuristics.** Exhaustive search of the solution space takes impractically long when there are more than a few workloads. Initializing the search with one of the four heuristics (Decreasing Width, Decreasing Height, Decreasing Area, or Decreasing Perimeter) does not remedy this problem; growth is exponential throughout the range plotted. In this graph, the lines are slightly offset horizontally to make them distinguishable. The y-axis is in log scale.

instance. The loss of optimality comes from the fact that solutions of this form may have a small amount of wasted space in each smaller instance. Had the entire problem been considered at once, however, there may have been a way to coalesce these voids into a larger space and fit a rectangle into it (see Figure 4.7).

There are two issues that make even the approximation forms of these theoretical algorithms potentially impractical for our use. First, while the algorithms are polynomial in the number of rectangles, the number of possible ways to fit rectangles into a subproblem appears as a constant in the running time analysis. This value is exponential in the size of the subproblems and the number of distinct sizes of rectangles. With small subproblems, running time will be fast but distance from optimality may be high because too many subproblems are being "glued together"; loss can be introduced

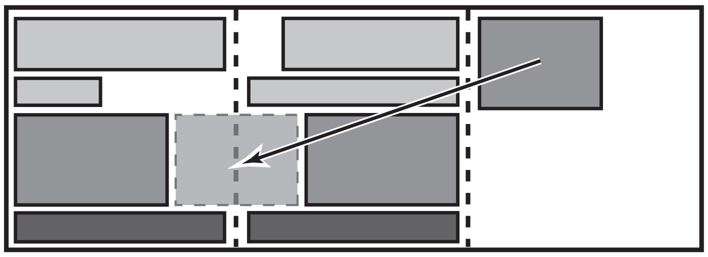


Figure 4.7: **Approximation algorithms.** These algorithms divide the larger problem into smaller subproblems that are solved optimally, but there can be significant loss in "gluing" them together into a larger solution. Here, the indicated rectangle wasn't placed in the more appropriate location, because it spanned two subproblems that were considered independently.

at each "joint." With large subproblems, loss will be reduced, but at the expense of a dramatic increase in time required to consider subproblems. Second, loss cannot necessarily be made arbitrarily small; it is limited by the size of the problem instance. In other words, when finding a solution within $(1+\epsilon)$ of optimal, the best ϵ achievable is a function of the problem size. For the size of the problems we expect to encounter, ϵ is expected to be too high.

Heuristics

There are various heuristic techniques that are sometimes able to produce a solution quickly. They do not guarantee the optimality of the solution, nor whether their inability to find a solution indicates there is none. However, they can often be very effective in practice. Furthermore, despite not providing guaranteed bounds, they may do significantly better than the $(1+\epsilon)$ approximation bound for our problem sizes. In our case, we only need to find a solution with a width of one or less; if a heuristic method finds any such solution quickly, how close it is to optimal may not be important.

In a similar vein to Fernandez de la Vega and Zissimopoulos's use of left-bottom justified solutions, a series of heuristic methods [4, 21] will try simply placing rectangles in a fixed order one after the other as far to the bottom and left as possible. The first such method is called *BL*, or *Bottom-Left*. The rectangles are placed in the same order as they appear in the problem specification. BL places the first rectangle at the bottom-left corner of the enclosing rectangle. For each of the remaining rectangles, BL places them one after the other in order as follows: introduce the next rectangle at the top right, then slide it downwards until it hits another rectangle or the bottom, then slide it leftwards until it hits another rectangle or the side. If it can now be moved down farther, continue moving it alternatively down and then left until it can no longer move in either direction. An improvement on this method is called *BLF*, or *Bottom-Left-Fill*. This scheme places each rectangle in turn at the lowest, leftmost position that can fit it; lower locations are (arbitrarily) preferred over more-to-the-left locations. The difference between BLF and BL is that BLF can place rectangles into locations that have space, but cannot be reached by sliding actions because they have been "sealed off" by earlier rectangles.

Variations on these algorithms change the order in which rectangles are placed, rather than following the arbitrary order in which the rectangles appear in the problem instance. BL-DW and BLF-DW sort the rectangles in order of decreasing width first. BL-DH and BLF-DH sort by decreasing height. Lesh *et al.* [28] suggest sorting by decreasing area (BLF-DA) or perimeter (BLF-DP).

Lesh *et al.* [28, 29] further suggest that, if one of these heuristic methods does not work, it may still be possible to find a solution quickly by spending a limited amount of time randomly searching small perturbations to the initial heuristic orderings, a technique they name BLD*.

Finally, Hopper and Turton [21] discuss combining these heuristics with meta-heuristic techniques such as simulated annealing, genetic algorithms, or hill-climbing. These techniques can start with an initial placement made with, for instance, BLF-DW, and refine the solution over a sequence of modifications in an attempt to approach the optimum.

4.4.4 Relaxing the problem

Argon maintains a guaranteed level of efficiency, expressed by the R-value, when a single server is shared; we extend that concept to the clustered setting. We refer to the R-value being enforced at a particular server as R_{server}. If clustering does not introduce any further loss in efficiency and each server is operating at at least R_{server}, then the overall efficiency seen by the client is still R_{server}. However, just as it may not be possible or practical to share a single server with perfect efficiency, it may not be reasonable to cluster with perfect efficiency. Thus, we introduce a second R-value, $R_{clustering}$, which represents the minimum efficiency maintained by the clustering scheme. The R-value observed at the client, then, is

$$\geq R_{clustering} \times \min_{servers} R_{server}$$

The R-value at the client is the ultimate indication of whether insulation has been achieved; $R_{clustering}$ and R_{server} are not externally visible and can be manipulated for the convenience of the storage system.

4.4.5 Our approach

We perform strip-packing to generate a schedule for the cluster as follows. We start by attempting to achieve $R_{clustering} = 1.0$. We create four parallel threads to attempt four different heuristics (BLF-DW, -DH, -DA, -DP). If the heuristic orderings do not lead to a solution, then we try nearby points in the solution space by permuting the heuristic sort orders for a limited period of time.[8] Each of the threads explores orderings near its initial sort order. In our experience, no specific sort order is always a good starting point for this exploration, but one of them usually is. When a thread finds an acceptable solution, the other threads are halted and the solution is used. Figure 4.8 shows pseudocode specifying the exact sequence in which we search the solution space near a heuristic ordering. If a solution cannot quickly be

[8]For instance, sorting the rectangles in decreasing width order may yield the heuristic ordering {A, B, C, D, E}. If placing rectangles in this order does not yield a satisfactory solution, we might next try the ordering {B, A, C, D, E}.

```
sched = blank schedule
for i = 1 to number of distinct types of workloads do
  remaining[i] = number of workloads of type i
call place_next_workload(sched, remaining)
exit with "No schedule found"

place_next_workload(sched, remaining)
  if remaining[i] = 0 for all i then
    exit with "Schedule found", sched
  for i = 1 to number of distinct types of workloads do /* L */
    if remaining[i] > 0 then
      /* Build a schedule by placing a workload of type i next */
      for s = 1 to number of servers do
        for t = beginning of round to end of round do
          if (s, t) is an active corner in sched
          and a workload of type i fits at location (s, t) in sched then
            sched2 = sched
            remain2 = remaining
            Place a workload of type i at location (s, t) in sched2
            decrement remain2[i]
            call place_next_workload(sched2, remain2)
            /* Only consider the first placement that fits */
            skip to next iteration of loop L
```

Figure 4.8: **Search ordering.** If the heuristic placement methods do not immediately find a solution, nearby solutions are searched as specified by this algorithm. The particular heuristic being used affects the ordering of the types of workloads in the above code. For instance, if Decreasing Width is used, "workloads of type 1" refers to those with the greatest width.

found, we relax $R_{clustering}$ (for instance, we might next try $R_{clustering} = 0.95$) and repeat. If the minimum acceptable R-value is known up-front, we can alternatively start with a value of $R_{clustering} < 1.0$ rather than initially considering the best possible value.

Initial setup

The inputs to our algorithm are the number of homogenous servers in the cluster and the list of workloads, each of which is described by a number of servers and a fractional share for each server. We create small rectangles corresponding to each of the workloads, as described in Section 4.4.2, with the height equal to the number of servers needed by that workload and the width equal to the proportion of the servers' time it needs (e.g., 1/3). We create a large rectangle to represent the cluster with the height equal to the number of servers in the cluster and the width equal to $1/R_{clustering}$.

The width of the larger rectangle represents the round length (the period over which the schedule repeats); each workload's rectangle consumes a fraction of that round length. If $R_{clustering} = 1.0$, representing no loss of efficiency due to clustering, then when we pack the workloads, a workload requesting a particular share of a server will receive that proportion of time. If $R_{clustering} < 1.0$, however, then the round length will be scaled up without scaling up the workloads. This creates more space into which to pack the rectangles, which may make the problem faster to solve or possible to solve where the original one was not. But, the workloads receive a lower proportional share of the overall round, resulting in a fractional decrease in performance equal to $R_{clustering}$.

4.4.6 Evaluation

To evaluate the efficacy of creating a timeslice schedule using our approach, we created a number of random problem instances representing sets of storage workloads. Our results show that exhaustive search is impractical; that starting exhaustive search with any one of the heuristic orderings does not improve mean solution time significantly; but that our approach of trying multiple heuristics in parallel and exploring nearby solutions does result in much faster solutions in the mean. Furthermore, for problems too large to solve even with this approach, relaxing the value of $R_{clustering}$ can create an easier-to-solve version of the problem.

Experimental setup

Problem instances To create a large number of workload sets, we generate lists of storage workloads randomly. A range of list sizes is generated to evaluate the growth of computation time as the number of workloads grows.

For each workload, we choose the number of servers on which to store the workload uniformly at random from among the numbers 1, 3, 5, 7, and 9.[9] For each workload, we also choose uniformly at random the proportion of the servers' time it needs from among the fractions 1/2, 1/3, 2/3, 1/4, 1/5, and 1/6.

An appropriate cluster size is calculated by summing the areas of the workloads' rectangles and choosing a number of servers so that the cluster's rectangle has that area, rounded up (before adjusting for $R_{clustering}$). This corresponds to the cluster size that would have the least wasted resources for that set of workloads. If one or more of the workloads needs more than the computed number of servers, however, the cluster size is increased to match.

We used this procedure to generate 1000 random problem instances of each size depicted in the figures, with the exception of some of the $R_{clustering} = 0.9$ cases to keep experiment time manageable; for 14, 20, and 30 workloads, we used 500 random problem instances and for 40 workloads we generated 200 problem instances.

Hardware All experiments were performed on machines with Pentium 4 Xeon 3.0 GHz processors running Linux 2.6.24. The memory footprint of strip packing is small, so memory and storage were not bottleneck resources.

Results

Individual heuristics Figure 4.6, introduced earlier, shows the decreasing width (DW), decreasing height (DH), decreasing area (DA), and decreasing perimeter (DP) heuristics applied to the problem. If a solution was not found by one of the heuristic orderings, we continued to explore the solution space, starting near the heuristic orderings, until a solution was found or the entire solution space had been exhausted, as described in Section 4.4.5. No single heuristic improved the mean time to solution; search time averages over the

[9]These values are typical of threshold quorum schemes used for erasure coding, where an odd number ensures a majority exists for a bipartition of servers.

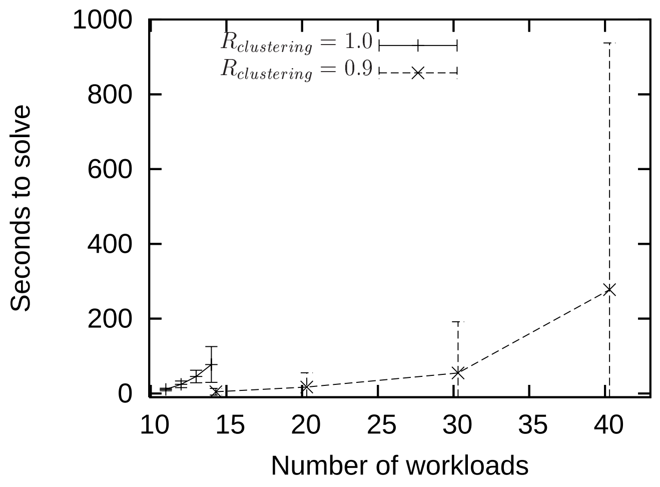


Figure 4.9: **Parallel heuristic search.** Running searches initialized with different heuristics in parallel allows the best-performing heuristic for a particular problem to find a solution faster. For the $R_{clustering} = 1.0$ case, however, an adequate sample size could not be achieved for problems of size 14 and greater due to growing run time. Relaxing $R_{clustering}$ further accelerates the search and makes larger problems tractable. The error bars show one standard deviation.

sets of workloads of each size for each heuristic are indistinguishable from the unsorted exhaustive search.

Our approaches Figure 4.9 shows solution speed with our approach — parallel execution of the four heuristics, continuing to search the nearby solution space if a solution is not immediately found. The two lines represent $R_{clustering} = 1.0$ and $R_{clustering} = 0.9$. A timeout value, described in the next section, is used to terminate the search and return "no solution found" after a period of time. The plotted times correspond to running the four threads on a single CPU, each at quarter-speed. This represents a pessimistic run time; one might use CPUs in the storage cluster itself to perform this search in parallel, or a multi-core machine.

Figure 4.10 shows, for the $R_{clustering} = 0.9$ case, the proportion of problems solved for each of the four heuristics (and continued search of nearby

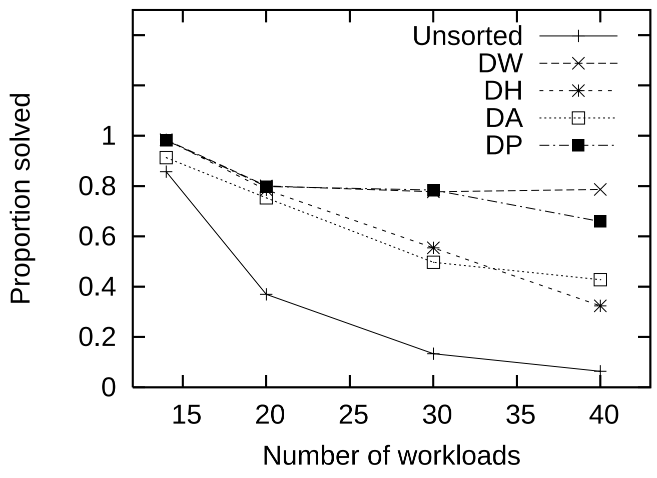


Figure 4.10: **Proportion of problems solved per heuristic.** In a limited amount of time (one minute for 14 and 20 workloads, two minutes for 30 workloads, and nine minutes for 40 workloads), initializing a search with one heuristic is not able to solve all of the problems that trying the combination of heuristics in parallel is able to solve. The proportion of problems solved is normalized against the number of instances solved using the parallel approach. While the performance of DW and DP is similar and both do well, the two alone are not sufficient.

orderings for a limited period of time) normalized against the number of problems solved using our approach. No one heuristic is sufficient to solve all of the problems in a reasonably short period of time. Our approach improves solvability by allowing whichever of the four is best for a particular problem to be used without knowing which it will be.

Figure 4.11 shows the tradeoff between $R_{clustering}$ and runtime for problems of size 13. Relaxing $R_{clustering}$ not only reduces run time, but also makes more of the randomly-generated problem instances solvable. To maintain the desired overall R-value, however, this approach would then incur the cost of increasing R_{server} to compensate.

Solvability and timeouts We observed that our approach either finds a solution relatively quickly, or not at all, suggesting that we should halt the

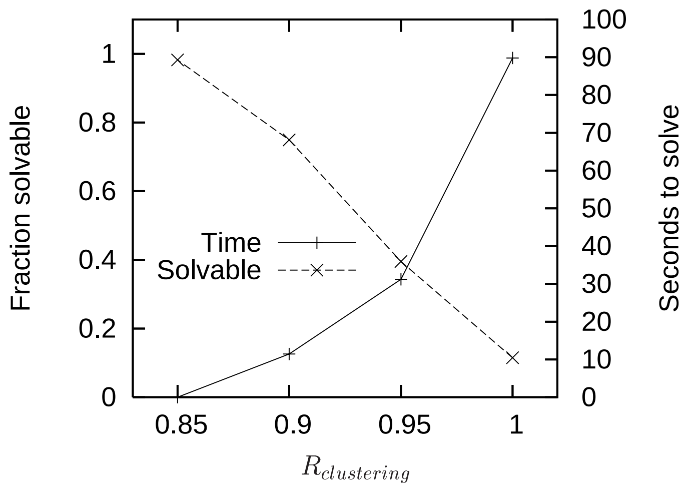


Figure 4.11: **R-value tradeoffs for 13 workloads.** Relaxing $R_{clustering}$ can accelerate solution speed and increase the number of solvable instances.

search after a period of time. To determine appropriate timeout values for each problem size, we ran all of the randomly-generated problem instances of that size without a timeout. Initially, many runs find a solution and terminate. Over time, we observed a decline in "completions" until a negligible or zero number of completions occurred per minute, at which point we halted the experiment. We then used the 90^{th} percentile of these completion times (rounded up to the nearest second) for the timeout value for that size problem instance. This has the effect of sacrificing the ability to find a small number of solutions with outlying run times.

Since many of these problem instances are too large for us to exhaustively solve, we cannot determine for certain that solutions do not exist beyond those we found. However, our experience with smaller problem instances, where we *can* compare the number of solutions found by our technique to exhaustive search, suggests that this strategy results in finding the vast majority of solutions. For instance, no further solutions were found by running to exhaustion the thirteen-workload problem instances that did not find a solution before we halted the original experiment; the percentile used to set

Num. workloads	Timeout (sec.)
12	2
13	5
14	6
20	30
30	114
40	524

Table 4.1: **Timeouts.** Timeouts used for $R_{clustering} = 0.9$, calculated as described in Section 4.4.6. Only problem sizes large enough to have a timeout greater than one second are shown.

the timeout directly determines what percent of problems will be solved in this case.

Parallelization in the cluster Our experiments show that the single-CPU parallel heuristic search achieves our "few minutes or less" target. But, searching for a solution to the scheduling problem is an "embarrassingly parallel" problem which can be split up across a virtually unlimited number of CPUs. Thus, any CPUs available in the storage cluster for which the schedule is being computed can be exploited for further speedup. For $R_{clustering} = 0.9$, if one CPU per server in the cluster is used to parallelize the search, then on average problems with twenty workloads take less than half a second, problems with thirty workloads take about a second, and problems with forty workloads take under four seconds.[10]

4.5 Coordination among servers

Once an appropriate schedule has been determined, the servers must follow it in a synchronized fashion. This section describes the requirements associated with such coordination and the solution we adopted.

[10]Recall that the number of servers varies across problem instances.

4.5.1 Requirements

The servers must begin following the schedule of timeslices at approximately the same moment of wall-clock time.[11] Once begun, the servers must start each subsequent timeslice in the schedule at the same approximate time. Any offset or error must not compound over time unless it accumulates identically across the servers. For instance, if a timeslice begins late at one server for some reason, the next timeslice must either begin on time, or (if it is desired to give the late-starting workload a full timeslice) the servers must all retard the beginning of the next timeslice by the same amount.

In addition to coordinating the timeslices across the servers, there are other decisions made by the servers that impact a workload's performance if not also coordinated. For example, if a workload has more requests queued than can be handled in a single timeslice, the servers must choose which subset of requests to send to disk. Only those requests chosen by all the servers will complete at the client that round. But, notice that the issue here is the set of requests completed rather than the order in which they are completed — request scheduling within each timeslice does not need to be synchronized and can be local to each server, allowing each server's low-level disk scheduling to remain unchanged.

The number of requests per timeslice Requests from clients arrive at our storage server via a custom protocol before being turned into I/O system calls. We perform timeslicing between workloads by queueing requests in the user-level storage server, then issuing the corresponding system calls once a workload's timeslice begins. However, we are not able to cancel a request that has been sent to the kernel or disk; this complicates the implementation of our scheduler. Timeslices establish a period of time during which a workload's requests can execute, but the number of requests that can be executed during that period is not known with certainty in advance. If we send more requests than would fill the timeslice, we would delay the beginning of the next timeslice and penalize the subsequent workload. If we send

[11] Because timeslices are long, e.g. 140 ms, small offsets among the servers — say, 0.5 ms — will not be significant.

fewer requests than could fit, we would have unnecessary idle time at the end of the timeslice.[12] Thus, we estimate how many requests would fill the timeslice based on historical observations of a given workload. We then send exactly that many to the disk, provided enough requests have been queued by the client.

The servers may not all compute the same number of requests to issue for a given workload. Variations in disk service times may make the historical observations that drive this decision different across the servers. If one server issues more requests in a timeslice than the others, this does not improve client performance because the client will still have to wait for the other servers to complete the corresponding fragment requests in later timeslices. If one server issues fewer requests than the others, on the other hand, it impedes client performance. Thus, care must be taken so that the servers issue approximately the same number in a given timeslice.

4.5.2 Initial solution: central coordination

One approach to keeping all the servers in sync is to centrally coordinate their actions. We implemented a central coordinator and added commands to our storage protocol to communicate between the coordinator and the servers. The coordinator is the "timekeeper" that monitors wall-clock time and determines when timeslices should begin and end. It sends `begin_timeslice` messages to servers at the appropriate times. The payload indicates the workload to execute next, the length of the next timeslice in milliseconds, and the number of requests the server should allow to be issued. Servers locally track the time that has elapsed since receiving these messages to know when to prepare to switch between timeslices; however, this information is not considered authoritative. A server that finishes a timeslice late, because a request was still in flight when the next `begin_timeslice` message arrives, begins the next timeslice as soon as possible and abbreviates the new timeslice (by issuing fewer requests) to return to sync.

[12] We could "trickle" additional requests to the disk until the timeslice ends, but this conservative strategy can hurt workloads that benefit from disk scheduling optimizations for concurrent requests.

At the end of a timeslice, each server sends a `report_requestsissued` message back to the coordinator. The payload indicates how many requests were actually issued to the disk, the actual length of time that the timeslice lasted (in case it started late), and the amount of idle time during the timeslice (in case the client did not keep the server busy the entire time). The central coordinator is able to calculate, for each of the servers, a per-request service time based on these three values and arrive at a common suggested number of requests to issue for that workload in the next round.

4.5.3 Symmetric operation

Although it worked, we prefer not to need a central coordinator. Our method for achieving this is to make decisions independently at each of the servers in such a way that they usually will agree across the servers without explicit coordination. We call this approach *symmetric operation*. We designed our servers to avoid the need for central coordination of the beginning and end of timeslices, the number of requests to issue in a timeslice, and which specific requests to issue in a timeslice if there are more requests queued than can be issued.

Timeslices are co-scheduled by using `ntp` [39] to keep wall-clock time synchronized across the cluster.[13] The management tool that determines the overall schedule of timeslices in a cluster also assigns a fixed wall-clock time for when the schedule should begin. Using this "time zero" and the schedule, the start and end times for each timeslice can be calculated. Once a server receives a schedule, it waits until the zero time and then uses its own clock to follow the schedule. If a server receives a schedule after the indicated time has passed, it joins the schedule in progress. If a workload overruns its timeslice, the server abbreviates the following timeslice to fall back in sync by the end of that second timeslice.

When more requests are queued than can be issued in a timeslice, the servers independently choose the same set of requests to issue. In our pro-

[13]Because timeslices are large — e.g. 140 ms — neither small inaccuracies in clock synchronization nor typical intra-data-center network latencies will affect performance significantly.

tocol each request is labeled with a unique ID by the client. Relative to a specific client, newer requests have a numerically greater ID. Thus, even if requests are received in different orders at different servers, the request IDs can be used to establish a consistent temporal ordering of requests. When choosing which subset of requests to issue in a given timeslice, then, each server can choose the oldest requests in the queue and be in agreement without explicit coordination. This approach also has the desirable effect of avoiding starvation.

Our system chooses to issue the same, or almost the same, number of requests in co-scheduled timeslices across the set of servers as long as the disks are performing similarly. Each server independently determines how many requests to issue in a given timeslice based on an exponentially-weighted moving average (EWMA) of the service times in previous timeslices for the associated workload at that server. This approach is not overly sensitive to occasional discrepancies, resulting in closely matching values across the cluster. Should a particular workload not be achieving its expected performance, there are two possible remedies. First, the α parameter for the EWMA, which represents the desired amount of smoothing, could be adjusted for that workload to promote more stable behavior. Alternatively, central coordination could selectively be provided for that workload.

4.5.4 Evaluation

We ran a series of experiments to confirm that (1) timeslicing without synchronization results in poor performance and (2) that our approach of symmetric operation results in coordination between the servers and the desired property of performance insulation.

Experiments were run on a cluster of Pentium 4 Xeon 3.0 GHz machines running Linux 2.6.16.11. The machines had 2 GB of RAM, but the storage servers were directed to use only 1 MB of RAM for caching to avoid confounding cache effects with disk performance. Each server used two Seagate Barracuda ST3250824AS 250 GB 7200 RPM SATA drives, one as a boot drive and one as the volume exported by our storage server. The drives were connected through a 3ware 9550SX controller, which exposes the disks to the OS through a SCSI interface. Both the disks and controller support command queueing. The machines were connected over Gigabit Ethernet using Intel 82546 NICs. Clients use the same hardware and do not perform local caching. The software used was Ursa Minor [2], with the Argon storage server described in the previous chapter.

Varying the number of servers

The first experiment shows that, while simple timeslicing provides performance insulation on a single server, striping data across two or more servers requires coordination between the servers to provide acceptable performance. We store two files, each of size 100 GB, contiguously starting at the beginning of the disks. Three closed, read-only, random workloads with no think time are run on three separate client machines. The first and third workloads use the first file and have one outstanding request at a time. The second workload uses the second file and has four outstanding requests at a time. We assign each workload one-third shares of each of the servers and request an R-value of 0.9. We run the workloads for a period of eight minutes and monitor the throughput over the last five minutes. The block size is chosen so that each server must supply a fragment of size 4 KB per request. This holds the disk activity constant as we increase the number of servers, to emphasize clustering effects rather than disk factors.

Figure 4.3 on page 68 shows the throughput of the first workload as we vary the number of servers under three scenarios.[14] The performance insulation goal is that the workload will receive a normalized throughput of $0.9 \cdot \frac{1}{3}$. Without timeslicing, the workload receives significantly less, regardless of the number of servers (in this case because Workload 2 — not shown — crowds it out with its higher degree of concurrency). Timeslicing solves the interference problem for the single-server case. But, with data striped over two or more servers, timeslicing results in worse performance than with no insulation at all because of the lack of coordination. Many requests must wait

[14]Standalone performance ranged from 305 KB/s for one server — within 7% of the datasheet performance for 4 KB random reads on this drive — to 2.2 MB/s for nine servers.

nearly an entire round to complete, because they are held up by the server with the farthest-in-the-future timeslice. Synchronizing timeslices achieves the goal for each cluster size tested.

The other workloads are not shown in the figure for readability, but they behave similarly. In the *Timeslicing* case, they also miss their goal performance by a wide margin. In the *Synchronized* case, each achieves its goal.

Putting it all together

The next experiment confirms that the entire process — starting with a list of workloads and a cluster, finding a schedule and disseminating it to the servers, and synchronizing their execution — results in the expected performance insulation. We use one of the random problem instances of size five and the schedule that was found when we solved it using our algorithm. We augment the problem instance with choices for file size and number of outstanding requests and ask the cluster to provide an R-value of 0.9. Expressed as tuples of (*num servers*, *share of time*, *file size in GB*, *outstanding*), the workloads are {(7, 1/6, 28, 2), (5, 1/3, 120, 3), (3, 1/2, 96, 1), (5, 1/2, 120, 4), (5, 1/2, 120, 5)}. We choose to make the workloads closed, uniformly random, and read-only with 4 KB fragments. We use a cluster of size ten, the minimum for this set of workloads. Each server runs with 1 GB of cache. The first workload benefits from a significant cache hit rate because it stores only 4 GB of data per server. If it receives insulated performance, this confirms the guarantees from cache partitioning compose as expected in the cluster as well. Figure 4.12 shows that the expected R-value is provided.

Macrobenchmarks

The next experiment confirms that our approach works for more realistic workloads. We run Postmark, TPC-C (an OLTP workload), and a specific query from TPC-H (a decision support workload) against a cluster. We ask the cluster to maintain an R-value of 0.85.

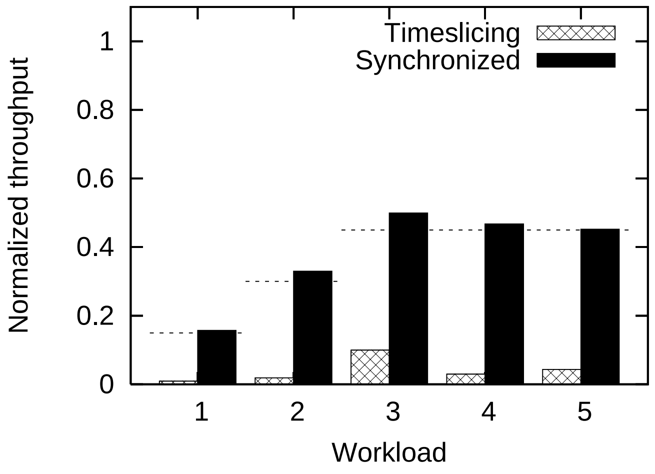


Figure 4.12: **Putting it all together.** Five workloads with different numbers of servers and shares of server time run on a cluster. The techniques described in Section 4.4 are used to generate a schedule of timeslices across the ten servers. Timeslices are coordinated across the cluster as described in Section 4.5. The y-axis shows throughput normalized against the performance each workload receives running alone on its set of servers. The lines show the $R = 0.9$ throughput for the share of server time each workload was assigned in the problem instance.

Postmark [25] is a benchmark designed to measure performance for small file workloads, such as on an email or newsgroup server. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append.

The TPC-C workload mimics an on-line database performing transaction processing [52]. 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).

TPC-H is a decision-support benchmark [53]. 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. Performance is measured by the elapsed time a query takes to complete. We run query 3 in this experiment (its runtime was closest to the other two benchmarks').

The workloads run on a cluster of six servers. Postmark stripes its data across five servers and is allocated $\frac{1}{3}$ of time on each. TPC-C and TPC-H stripe their data across three servers, and each is allocated $\frac{2}{3}$ of time on its respective set of servers. Table 4.2 shows the performance each receives, normalized to its standalone performance. Again, uncoordinated timeslicing is inadequate because workloads must wait for the slowest server to respond. Creating a schedule for the cluster and coordinating the servers provides the desired performance for each workload.

4.6 Other related work

This section discusses additional related work specific to this chapter, in particular, gang- and co-scheduling for high-performance computing, and spindle synchronization in disk arrays. Work related to the theoretical results on strip packing was described earlier in Section 4.4.3 and general quality-of-service literature was described in Chapter 2.

Timeslicing is employed to share CPUs among workloads in most common operating systems. When pooling numerous CPUs to create high-performance computing (HPC) clusters, the same concerns about composability of performance across individual nodes are raised. *Co-scheduling* [42] and *gang scheduling* [15] enforce synchronized CPU timeslices in HPC clus-

Table 4.2: Macrobenchmarks

Benchmark	Shared goal	Timeslicing	Synchronized
Postmark	$0.85 \times 1/3 = 0.28$	< 0.05	0.29
TPC-C	$0.85 \times 2/3 = 0.57$	0.21	0.58
TPC-H	$0.85 \times 2/3 = 0.57$	0.50	0.62

Timeslicing is unable to achieve the desired efficiency for three macrobenchmarks sharing a cluster. Only with a synchronized timeslice schedule do the workloads perform as expected. Performance is normalized against what each workload receives when it has exclusive use of its machines; bigger is better. For TPC-H, the metric is inverse runtime; for the others, transactions per second.

ters in the same spirit as our synchronization of disk timeslices. Feitelson [14] provides a survey of the various scheduling issues in the HPC environment. Despite the similarities, the approaches used to create process schedules for an HPC cluster are not directly applicable to storage systems. First, the strategies employed in the HPC setting may be constrained by communication topologies that do not apply to storage systems. Second, the cost of "context switches" can be significantly greater in storage systems than for CPUs; processor sharing techniques need not take as much care to minimize the occurrence of context switches or provide workloads with long contiguous spans of time. On a theoretical level, "zero-cost preemption" of a resource is compatible with linear programming formulations (which assume arbitrary divisibility of the modeled variables) whereas "non-preemption" of a resource is only compatible with integer programming formulations (which do not) [14]. Thus, the lower cost of preemption in processor scheduling is more likely to be conducive to polynomial-time approaches, and indeed polynomial-time algorithms are used for gang scheduling [14].

A similar coordination problem to ours is that of spindle synchronization in disk arrays [27, 41]. Without explicit synchronization among disks that store units of the same striped block, the rotational speeds and phases of the disks may differ. This can result in rotational latencies approaching the worst case (one whole rotation) instead of the average case as the number of disks increases. Some disks have physical inputs or use command-set extensions to match speed and phase in an array.

4.7 Discussion: Intended applications

This section revisits the target workloads and environments for this chapter.

The results in this chapter demonstrate that acceptable schedule calculation times can be achieved for dozens of workloads. As the bottleneck of this task is computational, improvements to processors will scale this limit over time, and parallelism at the servers will scale it now.

The approaches described are most suitable for long-lived sets of workloads with stable resource allocations on their respective servers, because schedule generation is a relatively slow process (fast enough to do occasionally, slow enough not to do dynamically every few seconds or more frequently). In addition, we have not exhibited a fast incremental algorithm that preserves most of the placement decisions. Thus, arriving workloads may require a drastic reconfiguration of the cluster, which could necessitate migration of all existing workloads to different servers. Such a task may be impractically costly and slow.

Sets of workloads that leave sufficient unassigned time in the schedule may allow transient workloads to be fit as they arrive. Therefore, some degree of fluctuation in the set of workloads may be acceptable, but it depends on the availability of exploitable free time (both in quantity and in "shape").

Our schedules assume that workloads stripe across each server. Workloads that use erasure coding would still work with our approach, but if they mostly read, then idle time will be introduced at the servers not receiving specific read requests. It is not clear to what extent this idle time could be put to more productive use.

For other types of cluster workloads, an entirely different approach may be necessary. For instance, individual files may be independently stored on separate servers, and a workload might access several of these files to complete an application-level task. There may be no consistent correlation between the servers accessed over time by a particular workload, and in fact, deliberate randomization of the access pattern may be the norm. Some form of central coordination (or distributed equivalent) and credit scheme may be necessary to track and maintain fairness of the consumption of each workload; the type of scheduling needed to provide efficiency while also handling the unpredictability of such workloads is not clear.

Since each server in the cluster runs Argon, the limitations of Argon also apply.

4.8 Conclusion

Performance insulation can be realized in cluster-based storage by co-scheduling timeslices for each striped workload. Parallel execution of several heuristics enables quick discovery of global schedules in most cases. Explicit time synchronization and implicit work coordination enable the system to provide 2–3x higher throughput than without performance insulation. With the appropriate care, performance insulation can scale across servers in a clustered storage system.

5 Providing bandwidth guarantees (Cesium)

Chapter 3 showed how to maintain efficiency while providing proportional sharing of a storage system. Consistent efficiency has significant benefits: reduced variability, less over-provisioning, and the ability to get better performance with fewer resources and host more workloads on the same system. Proportional sharing, however, is not directly able to provide bandwidth guarantees, because the proportion of a system needed to provide a given level of performance is highly workload-dependent and can be highly variable even for a specific workload. This chapter discusses how to build on the efficiency foundation of Argon, but provide bandwidth guarantees with efficiency instead of proportional shares with efficiency.

5.1 Introduction

For storage, performance SLO goals are commonly specified in terms of I/O bandwidth. Meeting such SLO metrics can be challenging, because the fraction of a disk needed to provide a particular bandwidth or response time varies dramatically, depending on I/O locality in the workload. A modern disk might provide over 100 MB/s for sequential access, but less than 1 MB/s for random 4 KB requests. Moreover, mixing the I/O of multiple workloads can disrupt locality for each, potentially making the resources required a complex function of each workload's individual I/O locality and how their requests are interleaved during servicing.

This chapter describes Cesium, a system that provides for SLOs specified in terms of storage bandwidth using explicit performance insulation (via disk head timeslicing) to bound the effects of such interference. Timeslices are dynamically resized, in each round, to adapt to the changes in I/O characteristics that workloads exhibit.

The approaches for storage QoS described in Chapter 2 have not been evaluated in the past with workloads that experience high variability and extensive disruption to locality from inter-workload interference. Our experiments with Cesium and with representatives of the other predominant approaches confirm the challenges of workload interference, the shortcomings of previous approaches in coping with it, and the success of our new approach. Perhaps surprisingly, the most popular traditional approach, based on token-bucket throttling, increases the frequency of violations that could have been avoided with a more efficient scheme (but reduces the worst magnitudes of violations compared to not providing QoS). In addition, throttling can dramatically reduce efficiency (e.g., by a factor of 6–8x in our experiments) compared to not providing QoS at all, because inter-workload interference is not managed and can even be increased as an artifact of how throttling works. In contrast, Cesium maintains each workload's efficiency and prevents almost all avoidable violations. (We present a characterization of which violations are avoidable by an effective scheduler, and which are fundamental and should not be expected to be avoided by any scheduler unless slack is available.) For realistic workload mixes that experience high variability and sensitivity to locality, Cesium achieves an order-of-magnitude reduction in the number of guarantee violations, compared to prior techniques. Cesium allows workloads to receive this predictable and controllable performance while also maintaining efficiency and reducing the average and 95^{th} percentile response times. Evaluation on RAID arrays confirms that Cesium's approach scales to these more complex systems as well.

5.2 Intended applications

This section describes the workloads and storage systems for which the techniques in this chapter are appropriate, and those for which the techniques are not suitable. Section 5.7 revisits this discussion at the end of the chapter, indicating why these limitations exist and to what extent they may be remediable with further refinement.

Cesium is suitable for single storage servers and for storage clusters for which the set of workloads and requests are the same across each node (in other words, striping across the entire cluster). Its performance guarantees are tailored for workloads that do not exhibit excessive idle time. As a bandwidth guarantee system, it is intended for use with workloads that desire bandwidth guarantees; those requiring latency guarantees may not be compatible with its effects on latency.

5.3 Background

This section describes background for providing bandwidth guarantees and efficiency to workloads sharing a storage system. Chapter 2 discusses prior storage QoS work, and Chapter 3 discusses the Argon storage server on which this chapter builds.

Service level objectives for storage performance are often expressed as bandwidth requirements; a large portion of the literature discussed in Chapter 2 attempts to accommodate such guarantees. Suppose Workloads $A, B, \ldots$ are sharing a disk and each requires guaranteed performance. SLOs might be specified as, for instance, "Workload A needs 6 MB/s." A controller that is able to accept and achieve such guarantees must be provided. If the controller is unfair, then Workload A might suffer because B is allocated too many resources, leaving too few for A. Separately, if the controller does not maintain efficiency, then both workloads might suffer as the overhead (i.e., interference and context switching) of sharing consumes too many resources, leaving too few for the workloads themselves.

A workload that typically must meet its SLO may also have periods of non-essential activity with a different access pattern. For example, the workload might perform garbage collection during idle periods. It may not be expected nor required that the SLO be met for this "unusual" activity, and achieving the same performance as for other types of activity may not even be possible. While the guarantee may not be applicable during such periods, the performance should not be unnecessarily impeded, and the behavior of the system during these periods should be well-understood, fair to the other workloads, and reasonable.

Efficiency and interference are also not static; they are affected by changes in workloads which can perturb the entire system. Interference levels can vary dramatically — by an order of magnitude or more — over time. Even if Workload A has highly regular behavior and would receive stable performance on a dedicated system, the effects of Workload B can cause unexpected efficiency swings. This occurs when the intensity or access patterns of B change over time, resulting in periods when the combined request stream is especially unfavorable. Systems, such as previous QoS schedulers, that do not manage interference allow the variance in one workload to cause unpredictability for all workloads. If Workload A suffers guarantee violations as a result of this variance, such violations should be considered avoidable, because they were caused by the failure of the sharing policy to maintain acceptable insulation from B. This type of violation is distinct from one caused by a workload's guarantee fundamentally being unsupportable for its access pattern when limited to its assigned resources (irrespective of interference).

By strictly controlling interference and efficiency, Argon eliminates avoidable violations. But, Argon does not provide bandwidth or throughput guarantees in absolute terms, leaving unclear how to use it with SLOs of this format. Under Argon, workloads are assigned a fraction of disk time, e.g. 1/3, and the system is assigned a minimum efficiency level (the R-*value*, e.g. 0.9). By managing interference at the disk and cache, Argon then provides a bandwidth guarantee to a given workload A that is a function of the workload's dedicated-disk bandwidth, $standalone_A$. For instance, for the example $fraction_A$ and system efficiency level given above, Workload A would receive $1/3 \times 0.9 \times standalone_A$. The other workloads $B, C, \ldots$ in the system would receive similar levels of bandwidth corresponding to their respective values of $fraction_{B,C,\ldots}$ and $standalone_{B,C,\ldots}$.

Because *standalone* may vary as the access pattern of a workload changes, we believe an effective scheduler that is designed to provide bandwidth guarantees should be able to vary $fraction$ appropriately and automatically over time, and we wish to address the shortcoming of fixed$fractions$ in Argon. For example, at one point in time the workload may have a particular access pattern, and *standalone* may be 60 MB/s. Later, the workload may have a different access pattern, and *standalone* may be 45 MB/s. If the workload has been assigned 1/3 of disk time and the system is operating at an efficiency level of 0.9, then, when the workload is in the first operating region, its bandwidth will be at least 18 MB/s; but, when the workload is in the second operating region, its bandwidth will only be guaranteed to be at least 13.5 MB/s. For workloads needing consistent bandwidth levels, choosing appropriate fractions and efficiency levels already adds administrative complexity. Having to select a constant fraction for each workload is also a significant constraint, because the appropriate fraction may vary over time. Choosing the worst-case fraction and providing that amount even when the workload needs less wastes slack that might be put to better use for other workloads. Thus, we believe that an ideal QoS system should achieve Argon's efficiency levels while also dynamically allocating the proper amount of resources to workloads to meet SLOs.

5.4 Storage QoS scheduler design

In contrast to previous bandwidth guarantee systems, which largely ignore the effect of inter-workload interference on efficiency, we wish to design a scheduler that provides bandwidth guarantees while maintaining efficiency. With better efficiency, we expect better guarantee adherence. We do not target workloads that are highly sensitive to every single request's latency, but Section 5.6.2 shows that higher efficiency results in better latency than even "latency-optimized" but inefficient schedulers, for e.g. the 99^{th} percentile of requests, and in the mean. This section describes the concepts behind our scheduler, which itself is detailed in Section 5.5.

5.4.1 Maintaining efficiency

To maintain efficiency, we utilize the core Argon mechanisms (e.g., diskhead timeslicing) described in Chapter 3. For streaming workloads (workloads that read or write blocks sequentially from the disk), high efficiency (e.g., $> 80\%$) cannot be provided unless they are able to access the disk for long periods of time without having their spatial locality interrupted by any other workload. The efficiency level achieved for these workloads, in fact, can be directly computed using the amount of uninterrupted time they are assigned and the cost of "context switching" (seeking the disk head to the appropriate position within a streaming workload's file) at the beginning of their access periods. If a disk seek takes 10 ms and 90% efficiency is desired, for instance, then such a workload must be provided uninterrupted access for 90 ms at a time. Increasing the queue depth presented to a non-timeslicing scheduler may also potentially increase efficiency, but care must be taken to avoid starvation. Timeslicing is effectively a way to avoid starvation in a systematic manner. While other methods to avoid starvation may exist, dedicated spans of time at least as long as those calculated using the above method must be allocated to workloads to reach a given efficiency level. Thus, we argue that a scheduler must incorporate some degree of timeslicing in order to be efficient and avoid starvation, and believe it is appropriate to continue use timeslicing in this follow-on work to Argon.[1]

5.4.2 Providing bandwidth guarantees

However, in addition to the efficiency goal of the Argon storage server, we wish to provide bandwidth guarantees. Thus, instead of sizing timeslices to

[1]Disk-geometry–aware schedulers, such as the Freeblock scheduler [34], may be able to significantly reduce positioning times, thus also reducing the uninterrupted span of useful work over which seek costs must be amortized. While such schedulers can provide a decrease in necessary timeslice lengths, they do not obviate the need for timeslices for locality-sensitive foreground workloads.

apportion total disk time fractionally across workloads, Cesium's timeslice scheduler will size timeslices to meet bandwidth guarantees. Let a *round* be the period of time during which each workload's timeslice executes once; our timeslicing-based scheduler will execute round after round indefinitely. We choose an appropriate round length (in milliseconds) that will be divided among the workloads. Then, for each workload, we estimate what fraction of the round it will need to achieve its bandwidth guarantee over that period. For instance, if the round is 2000 ms long, and a particular workload is assigned a guarantee of 1 MB/s, then it will be assigned a timeslice that is predicted to be long enough for it to transfer 2 MB. Since its timeslices are scheduled once every 2 seconds, this will yield an average bandwidth of 1 MB/s.

5.4.3 Handling different access patterns

Unfortunately, estimating appropriate timeslice lengths is not straightforward. For a streaming workload, it may take 33 ms to transfer 2 MB, while for a random workload, it may take 500–1000 ms. Other factors also affect performance, such as cache hit rate; and, while sequential and random access patterns represent extreme examples, there is a continuum of spatial locality in between. Making matters worse, workloads may exhibit constant change in access patterns; what seemed to be the proper timeslice length at the beginning of a timeslice may prove to be highly inaccurate.

However, simple computations take a negligible amount of time when compared to disk accesses. This allows us to monitor the performance a workload is receiving even while its timeslice is progressing and make adjustments to our scheduling decisions proactively rather than reactively. In particular, if it becomes clear that a workload will finish enough requests to satisfy its bandwidth guarantee for a round much earlier than expected, we can terminate the timeslice before it was initially scheduled to complete. Similarly, if a workload is performing worse than expected, we may be able to extend its timeslice beyond the original target. If a previous workload's timeslice was shortened because its guarantee has already been satisfied, this creates slack that could be assigned to later workloads without compromising any guarantees; the extra time may allow a workload to meet its guarantee for the round.

5.4.4 Fundamental vs. avoidable guarantee violations

When is it safe to assign more time to Workload A (decreasing its chances of a violation) at the expense of B (increasing its chances of a violation)? To make the proper tradeoffs, we must consider if and when guarantee violations are "acceptable."

To address this question, we focus on an aspect of bandwidth guarantees that has not been adequately considered in prior systems: the issue of changing access patterns. Suppose a workload is sequential and is assigned a guarantee of 30 MB/s. Such a guarantee may present no challenge to the system. Later, however, the workload may become significantly more random; so much so that 30 MB/s may no longer be possible, even with full use of a dedicated disk. (Many disks cannot provide above 10 MB/s for random workloads.) This would result in a bandwidth guarantee violation no matter how sophisticated the bandwidth guarantee algorithm.

One way to avoid this scenario is to assume each workload has a worst-case random access pattern when making admission decisions [58]. The problem of changing workloads can be avoided by extreme conservatism, but the set of workloads that can be accommodated will also be extremely conservative (i.e., much smaller than necessary). The 30 MB/s guarantee would never be accepted in the first place, even if the workload never wavers from sequentiality and could easily achieve that level of performance. Most other systems leave the issue of workload access pattern changes unaddressed; if the workload becomes unsupportable because of a change in access pattern, both it and the other workloads may begin to suffer guarantee violations. It will suffer, because it cannot be accommodated with the available resources; and other workloads will suffer, because in futilely trying to provide it with its performance requirement, resources will be directed away from other, supportable workloads. Note that "punishing" the unsupportable workload is not necessarily viable, because there is no notion of whether a particular access pattern is reasonable or unreasonable.

We provide a solution to this ambiguity by incorporating into our bandwidth guarantees the notion of *maximum fraction*. A workload is assigned both a bandwidth guarantee, such as 6 MB/s, and a maximum fraction, such as 1/3. Cesium maintains the bandwidth guarantee so long as doing so does not result in the workload consuming more than the specified fraction of server time. In addition, if there is additional time available once other workloads are satisfied, a workload that has reached its resource limit without meeting its guarantee may receive additional slack[2] beyond its maximum fraction, and may thus salvage its bandwidth guarantee. But, the system makes no promises that slack will be available.

This style of guarantee allows us to divide the set of guarantee violations into two classes: *fundamental* and *avoidable*. Fundamental violations are guarantee violations that occur when a workload's bandwidth cannot be accommodated within the fraction of time assumed when it was admitted; this occurs when the workload's access pattern has too little locality to achieve its bandwidth within its assigned resource fraction. Avoidable violations are the remaining violations, which are caused by failures of the sharing policy. Avoidable violations can occur because the sharing policy does not manage the set of workloads properly (for instance, it fails to maintain fairness among the workloads) or because the sharing policy does not maintain efficiency (it allows one workload's behavior to affect the efficiency another workload receives from the disk). These violations can be regarded as "artificial" sources of bandwidth guarantee violations.

Because fundamental violations are a characteristic of the workload and the physical limitations of disks, we reluctantly accept them as inevitable. Our goal is to minimize or eliminate the remaining sources of guarantee

[2]Note that we use the term *slack* to refer to free time that has accumulated when workloads end their timeslices early or rounds were not fully reserved, while deadline-based schedulers use the term to refer to exploitable free time before the request with the earliest deadline must be issued. Our slack accumulates when workloads receive better-than-expected performance or the system is undersubscribed; it is not directly related to issues such as workload idleness.

violations, the avoidable violations. These violations can be reduced by improving the scheduling policy. Therefore, we choose to design a system to monitor the behavior of workloads and make the appropriate tradeoffs among the workloads to provide guarantees to workloads whose accesses can be supported within their assigned resource allocations. We avoid diverting resources away from supportable workloads to workloads that are experiencing fundamental violations, because doing so could trigger even more violations (these ones, avoidable). But, we also avoid diverting excessive resources *away* from workloads undergoing fundamental violations. They continue to receive their maximum resources, to allow them to move past the period of unsupportable behavior quickly.

5.5 Implementation

This section details our algorithm and its implementation in a user- (as opposed to kernel-) level scheduler.

5.5.1 Workloads and requests

As requests arrive at our system, they are tagged with a workload identifier. A workload is the unit to which a guarantee is assigned; the guarantee applies to the set of requests received from that workload. When a new workload enters the system, it requests an unused identifier and an administrator specifies its bandwidth requirement in MB/s and its maximum fraction of resources. These values can be changed later if desired.

5.5.2 Round-robin timeslicing

The scheduler executes rounds of timeslices, where each round includes a single timeslice for each active workload. As a system property, a target round length is selected and represents the total time period over which each round executes. The appropriate round length can be calculated using the number of workloads and the "context switching cost" (cost of switching between workloads on a particular disk) to achieve a desired efficiency level, such as 90%. Rounds are repeated indefinitely while the storage system is running. Each round, however, need not be identical to the others.

Beginning a round

At the beginning of a round, the system determines whether the previous round ended early. If so, this represents slack caused by workloads receiving their guarantees in the previous round before using their maximum fractions of resources, and this slack can be exploited in future periods to potentially help workloads that are not meeting their guarantees. However, allowing slack to accumulate without bound and be assigned to a single workload could cause starvation for the others, because the one workload's timeslice could be lengthened excessively. Therefore, we apply a slack-aging algorithm; in particular, we only permit slack from the last two rounds to be carried forward. (This number empirically worked best for our test workloads.)

Next, we scan the list of workloads and determine which (if any) failed to meet their bandwidth requirements in the previous period despite having sufficient requests ready to issue to the disk. These workloads are marked as *unhappy*, and the remaining workloads are marked as *happy*. We then place the happy workloads into one set and the unhappy workloads into another, and randomly permute the orderings of workloads within the sets. This avoids consistently subjecting any one workload to the effects of executing first or last in a round. Then, we concatenate the permuted lists into an ordering of timeslices, where all the unhappy workloads execute first.

The next step is to choose appropriate timeslice lengths for each workload in the round. These initial allocations are predicted to allow each workload to meet its guarantee, but as described later, the algorithm may refine these allocations after timeslices are underway based on real-time observations of workload performance. For each workload, we calculate the amount of data it must transfer in one round to fulfill its guarantee. For instance, if the round length is 2 s, and a workload is guaranteed 1 MB/s, then it must transfer 2 MB in a round. We then reserve for each workload the projected amount of time needed to transfer the appropriate amount of data, based on estimated per-request time. Next, we sum the reservations across the workloads.

The total reservation for the round may exceed the configured standard round length plus any slack carried forward from the previous round; if so, we must start reducing reservations. We start with the last workload scheduled in the round (typically, the happy workload chosen last) and reduce its reservation until the total of all reservations matches the target round length, or the workload has been reduced to its maximum fraction of the round, whichever allows for a longer timeslice[3]. If the total reservation still exceeds the target round length, we continue the process with the $n-1^{st}$ workload, and so on.

Finally, if there is at least one unhappy workload, we attempt to reserve extra time for it, in the hopes of making it happy this period. This may allow for brief periods of lower bandwidth to be averaged out. We choose the first timeslice (belonging to one of the unhappy workloads), and increase its reservation to use any time in the round that has not been reserved. This extra time is available only if the initial reservations for the workloads summed to less than the target round size; if it was necessary to reduce the reservations as described in the previous paragraph, such extra time is not available. Last, if there is slack being carried forward from previous rounds, we also allocate this slack to the first (unhappy) workload to maximize its likelihood of returning to guarantee adherence.

During a timeslice

When a workload's timeslice begins, all requests from the preceding workload have completed and the disk is ready to be dedicated exclusively to the new workload. During the timeslice, available requests (received directly from the workload, or dispatched from a queue of postponed requests from the workload) are sent to the disk until we determine it is time to *drain* the timeslice. When we drain a timeslice, we stop issuing further requests and let any requests pending at the disk complete. Once the disk has completed

[3]Alternatively, we could reduce the last several workloads cumulatively to the same extent. These policies have different fairness on a micro level, but not a macro level.

all pending requests, the timeslice ends. We drain a timeslice when either of the following becomes true:

(1) The workload has issued enough requests to transfer its required amount of data for the round (e.g., the 2 MB given in the earlier example).

(2) The time reserved for the workload has been exhausted. However, if preceding workloads in the round ended their timeslices early, this surplus is made available to the current timeslice, increasing its reservation from the initial calculation. If it also ends early, the surplus is carried forward again. If the surplus is not used up by the last timeslice, it becomes slack in the next round.

When issuing requests to the disk, we do not constrain the number that can be outstanding at a time, with one exception: We do not allow more requests to be in flight than we project can complete by the end of the timeslice. However, if these requests complete sooner than expected, the workload is permitted to send additional requests.

Prediction

The prediction of how long a timeslice a workload needs is based on a prediction of the time it takes to complete a request at the disk. We have found that, because our algorithm can recover from imperfect predictions by changing timeslice lengths in flight, it is not highly sensitive to prediction accuracy. The workloads we use for testing and evaluation have high variability, with some requests completing in 0.1 ms and others in 30 ms. Predicting the wrong type of request would result in a two-order-of-magnitude error, which we found does result in a relatively high number of guarantee violations. Using a simple adaptive predictor resulted in such errors frequently, and hence in unacceptable violation levels. However, using a constant value (1.5 ms) that represents an estimate of workload behavior in between the extremes bounds error closer to one order of magnitude. Somewhat surprisingly, this guess is close enough to allow the adaptive timeslice length algorithm to size timeslices appropriately, resulting in much better performance and guarantee adherence than what we had believed would be a more sophisticated predictor. The ability to carry forward slack from timeslice to timeslice and across rounds further enhances the ability to compensate for imperfect predictions in practice.

Summary

Our algorithm must handle various scenarios and implementation issues, but all of its details center around meeting a primary set of goals. First, a workload's timeslice should end when its guarantee for the round is met or it has exceeded its resource limit. Second, all workloads should be treated fairly in the sense that, for instance, no one workload should consistently be first or last in a round. Third, if slack is available, it should be used appropriately.

5.6 Evaluation

To evaluate guarantee adherence for Cesium's algorithm and for other proposed bandwidth guarantee algorithms, we ran a series of experiments with combinations of realistic workloads and appropriate guarantees. We observed when guarantees were and were not being met and quantified which violations were fundamental. Results show that Cesium avoids nearly all guarantee violations other than fundamentally unavoidable ones, while the other algorithms did not deliver consistent guarantee adherence because of poor efficiency.

5.6.1 Experimental setup

This section describes the hardware, software, and traces used in our experiments.

Hardware

Each experiment used a single machine with a Pentium 4 Xeon processor running at 3.0 GHz. A dedicated disk was used to host the data being accessed by the workloads; it was a Seagate Barracuda 7200 RPM SATA drive, 250 GB in size. Two other, identical disks stored the OS and traces. All the drives were connected through a 3ware 9550SX controller, and both the disks and the controller support command queuing. The machines had 2 GB of RAM and ran Linux kernel version 2.6.26.

Software

A baseline that does not provide guarantees or manage workloads at all; our scheduler, Cesium; and alternatives from the literature were implemented in a disk time scheduler that manages requests coming from a trace replayer. The trace replayer performs as-fast-as-possible trace replay from trace files against the standard POSIX filesystem interface. For instance, a read in the trace will result in a standard POSIX `read()` call for the specified region of the local hard drive. The trace files recorded NFS-level read and write accesses to file systems. Files are specified as filehandles; thus, for each accessed filehandle in the traces, we assigned a corresponding, appropriately-sized region on the disk. The replayer maintains a specified level of concurrency while replaying the traces; we used 16 threads per workload.

The scheduler receives requests from individual workloads, queues them, and dispatches them to the disk (by making standard I/O system calls) when appropriate. While our scheduler runs at user level and sends requests to the underlying OS scheduler, this added layer did not appear to affect our results: they were not sensitive to the choice of OS scheduler. We chose a user-level implementation for simplicity of experimentation and debugging. (We have also implemented this scheduler as one of the selectable block I/O schedulers in the Linux kernel. Similarly, it could be incorporated into the kernel of another OS, or used to manage requests being handled by a user-level storage server.)

Sharing policies

We implemented four algorithms in our disk time scheduler to compare in our evaluation. The first, *Ignore SLOs*, does not manage the workloads or requests in any way. *Token bucket* implements a classic token-bucket–based throttling system attempting to maintain bandwidth guarantees. The bucket size (which represents the maximum surplus a workload can "save up" for later bursts) was 5 seconds' worth of bandwidth at each workload's configured rate. p*Clock* is an implementation of the *p*Clock algorithm [19], which uses both token-based metering and deadlines for latency control; we configure it to use a maximum latency of 2400 ms (to match the configuration of Cesium). *Cesium* uses the algorithm described in Sections 5.4–5.5 with a fixed round length of 2400 ms. We use a fixed round length in this evaluation that worked across the range of workload tuple sizes for comparability across experiments, but violations do not increase significantly with round lengths of $num\_workloads \times 300$ ms.

Workload traces

For simple workloads with minimal locality and no variability in characteristics, each scheduler is effective at maintaining bandwidth guarantees. But, the schedulers have not been previously evaluated with workloads that experience significant variation in behavior, suffer from significant inter-workload interference, and require efficient handling to maintain the benefits of locality. Unfortunately, previous schedulers can fail under the stress of these realistic challenges.

To evaluate the performance guarantee techniques with realistic workloads, we used traces collected from the day-to-day operations of a feature-film animation company and released by Anderson [3]. To our knowledge, these are the most intense NFS traces that are publicly available today. The traces contain the storage accesses that occur from different machines in a "render farm" as they perform the 3D rendering necessary to generate an animated film. The traces are read-heavy but do contain a significant number of writes [3].

Each activity record in the traces specifies the IP address of the machine making the request. Thus, we are able to separate accesses from different machines in the render farm, which are presumed to be performing unrelated jobs at a given point in time. The set of requests issued from a particular source machine is, therefore, treated as a workload for the sake of insulation from other workloads and as the unit to which performance guarantees are assigned. We chose ten of the most active machines from the traces to use as our workloads. We then selected a suitable span of time from the traces that is short enough to be able to run a series of experiments but long enough to capture interesting variability in behavior.

Because workloads in a datacenter often use storage systems built from more than one disk, for those experiments performed on a single disk we scale down the traces by a factor of four in a manner similar to striping. Since we are evaluating the performance of a shared server, we also suppress from the traces those accesses that would be absorbed by a client buffer cache that is 512 MB in size with an LRU replacement policy.

Run individually without any SLOs, the resulting trace excerpts each take approximately thirty minutes to run on our system. We derived appropriate performance guarantees by running each workload separately and examining the full-system bandwidth achieved for that workload's access patterns over time. We then chose a level of bandwidth that would be generally achievable while sharing the system efficiently with a few other workloads. Each workload, however, exhibits dramatic variability in access patterns, and for each workload we chose a guarantee that was reasonable for the majority of its run, but could not be met all the time, even on a dedicated disk. We believe it is important to include such behavior in our experiments, to evaluate how the techniques handle fundamental violations and periods of infeasible guarantees.

All but one workload is assigned a 0.2 MB/s bandwidth guarantee; the other, a 0.1 MB/s guarantee. While these bandwidth levels may seem undemanding, recall that we have filtered out requests that would hit in the cache from our traces, and note that the data sheet for our disks implies an average throughput of about 82 IOPS for random requests (based on average seek time and rotational latency). For an example request size of 4 KB, single workloads using a dedicated disk are therefore expected to receive 0.328 MB/s. Hence, our guarantee levels are appropriate and well-proportioned for our disks.

Workload combinations

For our experiments, we ran combinations of workloads consisting of between 2–10 workloads chosen from among the ten trace excerpts we generated. For two-workload sets, there are 45 combinations (not including pairs of the same workload), and we ran experiments with all of them. For nine-workload sets, there are ten combinations; and there is only one ten-workload set. We ran experiments with all of these combinations as well. To keep total experiment time manageable, however, we sampled the > 45 combinations of workloads sized between 3–8, and randomly chose a subset of 45 combinations for each size. The same random selections were used for each of the sharing policies we evaluated.

Measurements

We ran each of the ten workloads alone to measure the performance they receive on a dedicated disk. This allows us to identify the parts of the traces where fundamental violations occur. For a window of time, if a workload is able to receive x MB/s on a dedicated disk, and when it shares a disk it is assigned a maximum fraction f, then a system operating at efficiency R should be able to provide bandwidth $\geq f \times R \times x$ in the same window. Bandwidth can be measured over an arbitrarily configurable window; for our experiments, we measure bandwidth during 15-second periods of time.

Next, we run each of the different combinations of workloads. If, when a workload is sharing the disk, we detect a bandwidth violation, we can identify the corresponding window from the dedicated-disk run of that workload. By plugging in the appropriate values of x and f and expecting a system efficiency of $R = 0.9$ (as we showed in Chapter 3 is generally achievable), we can discern whether the violation in the shared run is fundamental or avoidable.

5.6.2 Results

This section presents several results. It illustrates the bandwidth achieved when a single combination of workloads runs to demonstrate the difference between fundamental and avoidable violations. It examines the violations experienced over the entire set of workload combinations. For example workload combinations, it quantifies total disk bandwidth, achieved efficiency, and request latency effects for each scheduling policy. Last, it confirms that Cesium provides good guarantee adherence on a RAID-5 configuration.

Example timelines

We plot the bandwidth received by one workload over time as it shares the disk in one of the randomly-chosen combinations of five workloads. The most straightforward way to depict bandwidth over time is to place successive windows of time on the x-axis and bandwidth on the y-axis. Unfortunately, in this format, it would be difficult to locate corresponding periods in a trace across different runs, because the x-axis would represent time rather than location in the trace — and thus is affected by performance. Instead, in Figure 5.1, we graph successive portions of the trace on the x-axis, and the amount of time it took to replay that part of the trace on the y-axis. If a workload has been assigned a guarantee of 1 MB/s, then we plot successive 15 MB windows of the trace (corresponding to an expected 15 second window of time, the same as our chosen measuring window) on the x-axis, and the average number of seconds it took to transfer 1 MB during each of the windows as the y-values. In this representation, a violation occurred whenever the plot is above the line $y = 1$ second (rather than below the line $y = 1$ MB/s, as would be the case in the conventional way). The advantage to this format is that points on the x-axis line up from run to run, regardless of performance.

Figure 5.1(a) shows the performance the workload receives with a dedicated disk, but the y-values of the data have been scaled to represent the performance a system running at 90% efficiency and allocating a maximum fraction of 1/5 should be able to provide that workload, as described in Section 5.6.1. This is accomplished by taking the number of seconds required to transfer the data with a dedicated disk and multiplying by 5 (the reciprocal of the fraction) and then dividing by 0.9 (the efficiency level). By inspecting Figure 5.1(a), we see that an efficient system should be able to meet the guarantee (points below the line $y = 1$) most of the time, but there are several periods where fundamental violations will occur.

Figure 5.1(b) shows the performance the workload receives when sharing the disk under the Ignore SLOs policy, where no attempt is made to manage the interference between the workloads. Note that many of its violations occur during the periods where fundamental violations are expected. Some, however, such as the ones at approximately $x = 750$, are avoidable violations because there are no corresponding violations in Figure 5.1(a).

Figure 5.1(c) shows the performance the same workload receives under Cesium. Note that because each "spike" in Figure 5.1(c) matches a spike in Figure 5.1(a), each violation is a fundamental violation in this particular run — as expected for an efficient system. Thus, in this specific experiment, our system has succeeded in maintaining efficiency for the workload and eliminated all the artificial or avoidable sources of guarantee violations. In fact, many of the fundamental violations predicted by Figure 5.1(a) do not occur with the timeslicing scheduler. This is because the fundamental model predicts which violations occur when the fraction of time available to the depicted workload is limited to 1/5. But, as described in Section 5.5, if another workload meets its guarantee for a round in less than its maximum time, this slack is made available to other workloads. Hence, the windows where we avoid predicted violations are windows where statistical multiplexing works to the favor of the workload shown.

While only one of the five workloads is plotted due to space constraints, the others experience similar behavior.

Fundamental violations

Using the method described in Section 5.6.1, we calculate the expected number of fundamental violations for each workload when assigned 1/2–1/10 of the system. These values correspond to the maximum fraction given to the workloads in our combinations of 2–10 workloads. (Note that neither our design nor implementation preclude other non-matching fractions among the workloads.) The results are shown in Figure 5.2, with each point in the scatterplot corresponding to one of the workloads. Sets of two or three workloads should fit easily on the disk. Larger sets will have violations in up to 20% of the periods because the guarantee levels are infeasible that frequently. An increase in such fundamental violations is how Cesium degenerates when facing infeasible guarantees.

Figure 5.3 shows conservative bounds on statistical multiplexing derived from Hoeffding's inequality [20]. This formula indicates an upper bound on how many violations should result from the failure of statistical multiplexing — in other words, how often the combination of workloads would, in total, require more disk time than 100%. The bound is based on the mean resource requirements and the range of resources needed by the workloads (e.g., average of 5% but as much as 20% in some periods). The two lines represent the violations predicted using the average range among our workloads and the worst-case among our workloads.

The inequality predicts that there may be periods of over-subscription when the number of workloads exceeds five. This is of significance because when there is such a period, no slack will be available in our system. This will prevent our algorithm from carrying forward extra time between rounds. During periods where our algorithm loses the ability to exploit slack, it loses much of its ability to correct for mispredictions in estimated request times and timeslice lengths.

Combinations of workloads

Figures 5.4(a)–5.4(d) show violations for each of the schedulers we implemented, running with the sets of workload combinations we generated. In these scatterplots, each point corresponds to the percentage of periods with violations experienced by one of the workloads when running in one of the combinations. The number of periods with fundamental violations, calculated by the method described in Section 5.6.1, is subtracted from the number of observed violations to yield the depicted values.

When ignoring SLOs (Figure 5.4(a)), avoidable violations occur even for the two-workload combinations. When ten workloads share the system, some experience nearly constant violations. Without management of the requests coming from each workload, proportional sharing cannot be ensured. In addition, the scheduler makes no effort to maintain the efficiency of the system. The benefits of locality that some workloads enjoy when running alone may be lost when their requests are interleaved with those from other workloads at a fine-grained level.

With token-bucket throttling (Figure 5.4(b)), the situation counter-intuitively becomes worse instead of better. Even for two workloads, avoidable violations occur in nearly half of all periods. While token-bucket schedulers attempt to manage fairness among the workloads, they do so by strictly limiting the number of requests each can send to the system. As described in Section 2.3.1, token bucket-schedulers work by limiting each workload to an upper bound: its bandwidth guarantee. This is fundamentally backwards from the intended goal of maintaining a lower bound. Consequently, even if workloads can fit together on a system, they are at best limited to exactly their guarantee level of bandwidth. Thus, the best-case scenario finds workloads constantly on the edge between making their guarantee and suffering a violation. But, token-bucket schedulers finely interleave requests coming from different workloads, disrupting any locality they may have; no attempt is made to preserve their efficiency. This may result in them requiring substantially more time to complete their requests, and there may not be enough time available in the system to service all the workloads at their reduced efficiency. Note that only a limited number of combinations of workloads were tested for this graph, because the run time of experiments with this scheduler was prohibitive.

pClock also counter-intuitively makes matters worse (Figure 5.4(c)).

While we have received acceptable performance using our implementation of pClock with other workloads, and while it outperforms token bucket for some combinations, it is not generally effective on these workloads. Like token-bucket throttling, it does not fully preserve efficiency when workloads share a system because it makes no effort to maintain locality or avoid fine-grained interleaving.

Cesium eliminates most non-fundamental violations (Figure 5.4(d)). It does this by automatically managing both the proportion of the system a workload receives, and the efficiency with which it operates during its assigned fraction of time. This allows workloads that benefit from locality to continue operating at close to their full efficiency, avoiding the increase in required resources suffered by the other systems. In some instances, the number of violations is actually less than the expected number; there are no avoidable violations and fewer fundamental violations than predicted. This is the result of statistical multiplexing working to the benefit of workloads; sometimes, one workload will meet its guarantee in less than its assigned fraction, and this slack allows another workload to use resources beyond its assigned fraction. Since the number of fundamental violations is calculated for a workload's assigned fraction, some violations may not occur when surplus is available.

Unfortunately, a low level of avoidable violations does remain in some cases. This begins at the point where Hoeffding's inequality predicts that the system will be oversubscribed in some periods (Figure 5.3). When this occurs, no free resources are available in the system, and the scheduler does not have slack available to make up for imperfect control decisions. The scheduler sizes the timeslices for each workload, and determines when to start draining requests and end a timeslice, based on projections of how long a workload's requests will take. The workloads used in these experiments are highly variable (regardless of scheduler), with some requests completing in tens of milliseconds and some in a tenth of a millisecond. The simple prediction algorithms used in our scheduler, chosen to keep implementation complexity low, do not make consistently accurate predictions.

Fortunately, slack allows the scheduler to make up for poor decisions.

	Ignore SLOs			**Token bucket**		
Num. of workloads	Avg. total disk MB/s	Achieved efficiency	Latency ms mean / 95% / 99%	Avg. total disk MB/s	Ach. eff.	Lat. ms
2	3.5	92%	9.7 / 49 / 210	0.3	9%	87 / 963 / 1013
5	3.1	77%	38 / 224 / 720	0.6	16%	87 / 949 / 1016
	p**Clock**			**Cesium**		
2	0.6	15%	53 / 169 / 360	3.3	88%	9.4 / 15 / 118
5	0.5	11%	181 / 184 / 1023	3.5	89%	25 / 17 / 466

Table 5.1: Example efficiency and latency measurements for each of the scheduling policies

But, if slack is not available, the effects of imperfect predictions start to show up as "avoidable" violations. We called any violations that would not occur under a fair and efficient scheduler *avoidable*. While the results show that our scheduler is significantly more effective than the others, it is not perfectly fair and efficient because of the uncertainty it faces with highly variable workloads.

Efficiency and response time effects

Cesium results in better guarantee adherence in part because it maintains significantly higher efficiency. Table 5.1 shows average total disk bandwidth, efficiency, and average and $95^{th}/99^{th}$ percentile response times (latencies) for representative example combinations of two and five workloads. Without throttling (ignoring SLOs), workloads may receive relatively high efficiency because the disk scheduler is able to perform seek optimizations within and across workloads. But, there is no explicit management of efficiency, and efficiency deteriorates as the number of workloads increases. The token bucket and pClock schedulers provide significantly worse efficiency levels. Their approach to balancing the streams of requests from workloads results in requests from different workloads becoming interleaved, resulting in poor locality. When operating less efficiently, requests cannot be handled as quickly, and thus they suffer from increased response times as well.

Cesium consistently maintains efficiency and bandwidth, while simultaneously reducing most response times. Although a few requests may block for nearly an entire round waiting for the appropriate timeslice, and there-

Num	**Ignore SLOs**		**Cesium**	
work-	Achieved	Violations >	Ach	Viols >
loads	efficiency	fundamental	eff	fund
2	75%	1%	92%	0%
3	57%	8%	90%	0.5%
4	51%	10%	86%	0.5%

Table 5.2: Cesium on a four-disk RAID-5 configuration

fore experience high latency, this happens to only a very small fraction of requests. Thanks to higher disk efficiency, Cesium provides the vast majority of requests with response times lower than those provided by any other tested scheduler — both the average and the 99^{th} percentiles are significantly lower. To show the relative latency effects in more detail, Figure 5.5 gives the distribution of latencies for each scheduler for an example workload combination. Figure 5.6 zooms in on the "tails" of the same distributions. Timeslicing does cause the occasional request to wait much longer than it would under the other schedulers, which perform fine-grained request-by-request scheduling, but response times are significantly reduced on the whole.

RAID

RAID [44] can increase performance, capacity, and fault tolerance above the levels offered by a single disk. Our timeslicing-based scheduler works without modification on simple RAID arrays (i.e., those with striping and redundancy, but without internal caches or adaptive algorithms). Table 5.2 shows the efficiency and guarantee adherence achieved on a four-disk RAID-5 array with example workload combinations. The guarantees used in this experiment began to saturate the array with four workloads. Cesium maintained efficiency and suffered only a negligible number of avoidable violations.

The Cesium scheduling algorithms are designed to work at the disk scheduling level. Thus, they are unlikely to work outside of a file server or high-end disk array controller with large quantities of cache and adaptive mappings of LBNs to disks. Instead, they should be implemented within the firmware of such devices, allowing them to provide efficient SLO support.

One specific level of complexity introduced by such systems is the multi-level aggregation of disks. RAID 5 can tolerate only the failure of a single disk in a group without data loss. Thus, it is beneficial to limit the size of groups to minimize the probability of a two-disk failure. However, limiting the number of disks that can be allocated to a volume limits the amount of parallel bandwidth available to that volume. A solution to this issue is to stripe across multiple smaller RAID 5 groups for parallelism. For instance, a volume may be created by striping across two RAID 5 groups. A given byte would then be stored on one of these groups. Within the specific RAID 5 group, the byte will reside on one of the disks, and contribute to a parity value on another.

While this approach does avoid a disadvantage of the more straightforward all-disks-in-one-group approach, it raises the question of load balancing. If bytes are divided evenly across two RAID 5 groups, for example, is it safe to assume each group will receive the same level of load? Will both groups exhibit the same performance? If the answer to either of these questions is no, then "hot spots" will result — imbalances between the groups resulting in sub-optimal performance.

The possibility of hot spots could be exacerbated by timeslicing, which increases the latency of some requests by intentionally backlogging requests coming from workloads. We ran a series of experiments to determine whether this concern is justified in practice. We created two RAID 5 groups of four disks each, each with a stripe size of 16 KB. We then configured our system to perform striping over these two groups, so that half of all requests go to one group and half to the other, and perform co-scheduling similar to that described in Chapter 4.

We ran combinations of the feature-film animation workloads with 2–5 workloads[4]. Like the previous experiments, we constrained the number of concurrent I/Os issued by each workload (in this case, to 32). For these ex-

[4]Five pairs, three combinations of three workloads, two combinations of four workloads, and two combinations of five workloads.

periments, however, we added an additional constraint; we do not allow the newest request to be more than 32 requests in the trace beyond the oldest outstanding request. This "windowing" restriction makes performance particularly susceptible to hot spots. However, the postulated performance deterioration due to hot spots does not occur, as Cesium effectively maintains efficiency and performance even under these situations. Each combination met or exceeded the desired R-value of 0.9.

5.7 Discussion: Intended applications

This section revisits the target workloads and environments for this chapter.

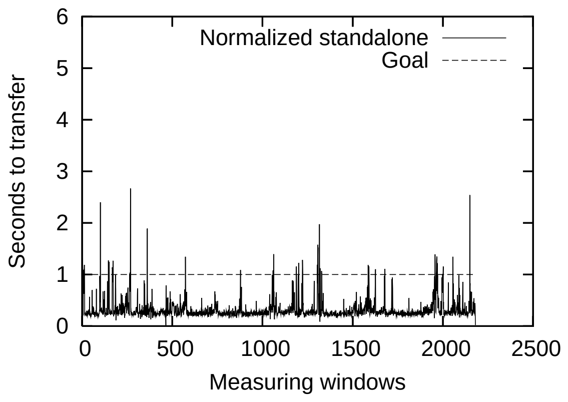
Cesium has been evaluated and shown to be effective on single storage servers. For storage clusters for which the set of workloads and requests are the same across the servers, the same fractions would likely be appropriate across the cluster, and thus Cesium could appropriately co-schedule timesslices across the servers. For clusters with workloads that use subsets of the servers, the clustering approach described in the previous chapter would not be effective. Cesium's reactive sizing of timeslices precludes static scheduling in advance of a round and makes consistent alignment of timeslices unlikely.

Cesium's performance guarantees are tailored for workloads that do not exhibit excessive idle time. It is not straightforward to interpret a bandwidth guarantee for a workload that is idle; such a workload will inevitably receive no bandwidth for the idle period. While the expected or desired behavior in this scenario could be defined more clearly, Cesium's timeslicing might still not be appropriate. If a workload is idle during its timeslice, the idle time cannot be used for either its benefit or for the benefit of other workloads. Its timeslice may be extended up to its maximum fraction, however, if it has not met its guarantee for the round. If it issues requests later after its timeslice has ended, they will be deferred to the next round because the workload's maximum fraction of time has already been exhausted, albeit unproductively.

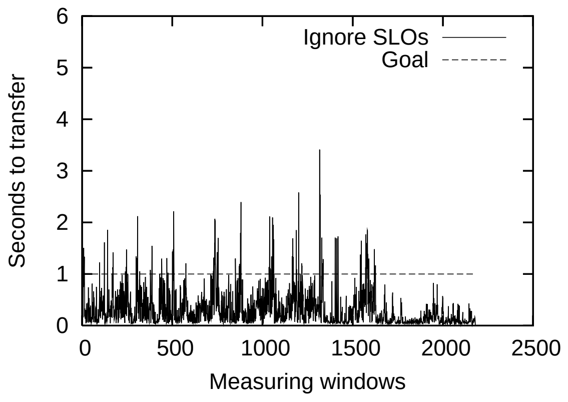
Cesium provides bandwidth guarantees. Similarly to Argon, Cesium's timeslicing increases efficiency, and thus decreases mean latency compared to not providing quality of service, and increases mean latency for closed-loop workloads in a bounded manner when compared to isolation. Timeslicing increases maximum latency (for a potentially small subset of requests) significantly, and similarly increases variance. Thus, Cesium is not suitable for workloads requiring hard real-time maximum latency guarantees, or that are otherwise sensitive to these latency effects.

5.8 Conclusion

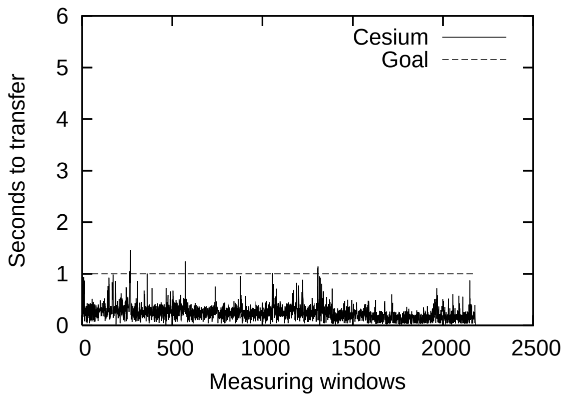
Bandwidth guarantee violations can be dramatically reduced by using a scheduler that explicitly manages interference between workloads. By considering efficiency, violations can be differentiated between those that are fundamental and those that are avoidable. Cesium, our timeslicing-based system, strictly limits interference while adapting timeslice lengths to control bandwidth. As a result, it can achieve $> 85\%$ efficiency where other schedulers may achieve $< 20\%$. In experiments on realistic workloads, Cesium uses its efficiency to greatly increase overall throughput and also to eliminate almost all avoidable violations, and is able to use slack to reduce the number of "fundamental" violations as well. Compared to traditional techniques, workloads running under Cesium may experience an order of magnitude fewer violations. Thus, Cesium provides predictable, controllable performance to workloads sharing a storage system.



(a) Normalized standalone



(b) Ignore SLOs



(c) Cesium

Figure 5.1: Bandwidth guarantee adherence over time for a dedicated disk and two of the schedulers

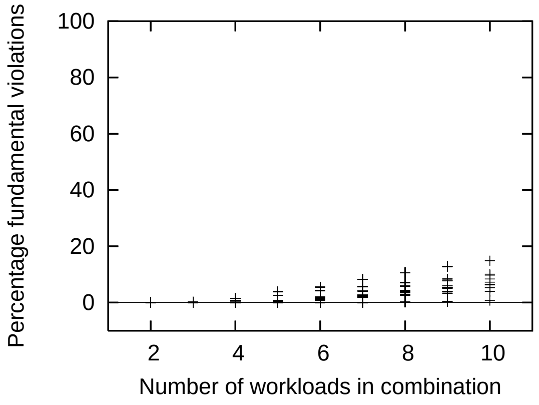


Figure 5.2: Percentage of fundamental violations

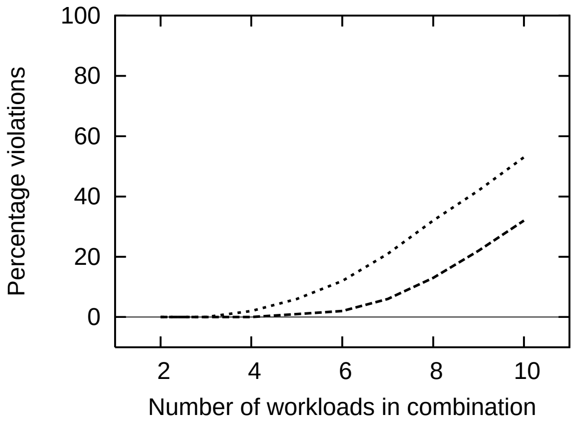
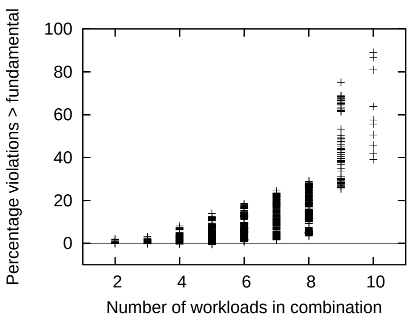
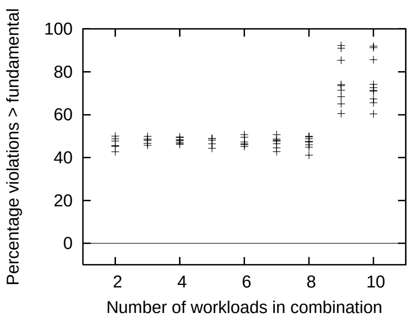


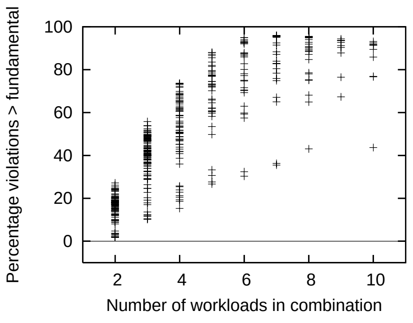
Figure 5.3: Percentage of violations predicted by Hoeffding's inequality



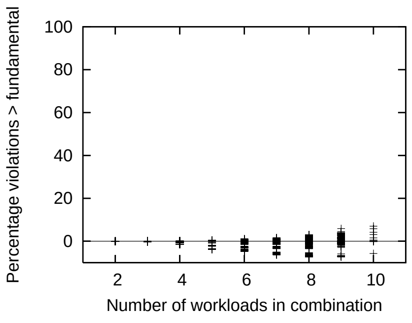
(a) Ignore SLOs — Average = 9.9%



(b) Token bucket — Average = 55.5%



(c) *p*Clock — Average = 49.2%



(d) Cesium — Average = –0.5%

Figure 5.4: Percentage of violations above or below the fundamental violations for each of the scheduling policies

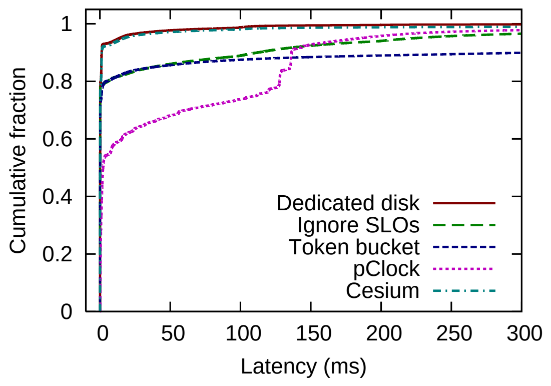


Figure 5.5: Latency for each of the scheduling policies: For an example combination of five workloads, the distributions of latencies experienced by a representative workload are shown as cumulative distribution functions. Each y-value shows the fraction of requests experiencing latency of at most the corresponding x-value.

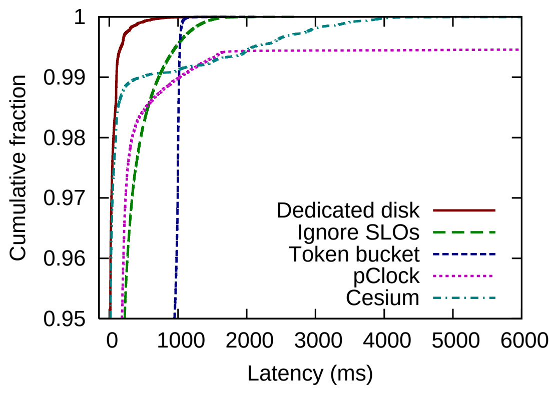


Figure 5.6: Latency "tails" for each of the scheduling policies. The y-axis begins at the 95^{th} percentile.

6 Conclusion

Performance insulation is a system property that bounds efficiency loss when workloads share a system. Insulation is achieved by controlling the sources of inter-workload interference. In a storage system, such interference comes from locality disruption at the disk head and from competition for cache space. Storage system interference can be mitigated with round-robin disk-head timeslicing, prefetching and write coalescing, and cache partitioning. Each of these techniques is necessary to provide performance insulation, and taken together, a system implementing them can maintain efficiency at 90% of a workload's dedicated-disk efficiency. To achieve such efficiency, however, casually applying the mechanisms is not enough; they must be guided by efficiency-aware policies to, for instance, size timeslices and cache partitions appropriately for efficiency goals.

In cluster-based storage, timeslicing can be detrimental because a client must wait for the farthest-in-the-future timeslice among the set of servers it is using. Its request cannot complete until the "slowest" server responds, a complication introduced by clustering. However, if the servers each schedule a workload's timeslice at the same time, this issue can be avoided entirely. Doing so requires synchronization among the servers, and the creation of a cluster-wide schedule of workloads across the servers. Finding an appropriate schedule is an NP-complete problem. Fortunately, effective heuristics exist that can solve most problem instances in a reasonable amount of time.

Providing bandwidth guarantees to storage workloads is challenging for two reasons: workloads may have intrinsic variation in demand; and inter-workload interference may change over time, resulting in further swings in resource requirements. To address workload variation, Cesium monitors the progress a workload is making toward meeting its bandwidth goal and adjusts its fraction of the system accordingly. To address inter-workload interference, Cesium adopts the same principles that Argon uses to maintain efficiency, such as performing timeslicing with sufficiently large quanta. By strictly limiting interference and maintaining efficiency, Cesium eliminates one major source of bandwidth guarantee violations. The same challenging workloads suffer from many avoidable violations when running under other representative schedulers from the literature. Virtually all violations experienced by workloads running under Cesium are caused by a workload's own intrinsic variability exceeding the level of resources available on the system.

Performance insulation is a useful system property when a system is shared among workloads. It provides a quantified bound on efficiency loss despite the potential for significant interference to occur. Performance insulation can be achieved on storage systems, single and clustered, by applying practical techniques with appropriate efficiency-aware policies. Insulation can be used to provide direct efficiency guarantees. But, performance guarantees on metrics such as bandwidth are appealing for many workloads. For systems providing bandwidth guarantees, insulation is a key building block to allow guarantees to be maintained more robustly with fewer resources. Performance insulation allows a higher degree of predictability and control when workloads share a storage system.

6.1 Future work

Various avenues for further exploration are suggested by the research described in this dissertation. This section lists some of the areas we have explored or identified.

6.1.1 Improving the accuracy of Cesium's predictor

A small number of avoidable (non-fundamental) violations occur under Cesium despite the care it takes to maintain efficiency. These result from imperfect predictions of how long each request will take, predictions that are used for the key decision of how long a timeslice should be. Instead of being considered fundamental or avoidable violations, these violations might best be placed in a third class, *prediction* violations. A more sophisticated prediction algorithm or a more resilient scheduler might be able to reduce this type of violation.

6.1.2 Implementing Argon or Cesium in complex arrays

High-end storage systems are often built with at least hundreds of disks and dozens of arrays. In front of these components is a sophisticated controller that routes requests from workloads to the appropriate arrays and disks, performs caching, executes algorithms that adapt the system's behavior to best match the needs of workloads, and performs various maintenance tasks. We have not evaluated Argon or Cesium in such an environment, because the modifications would need to understand the complex and potentially dynamic mappings of workloads to disks, and also alter the behavior of the caching subsystem. Both of these aspects would require modifications to the controller software or firmware.

Because of the proprietary nature of such storage systems, it was not practical for us to pursue this line of research. However, if we could implement an extended version of Argon or Cesium in such a system, or simulate much of the complexity of such a system in an environment that we could modify appropriately, it would allow us to identify the new challenges posed by expanding performance insulation to these complex systems and confirm that our techniques scale to this domain.

6.1.3 Improving response times with timeslice sharing

Some workloads, such as streaming workloads, must be strictly separated from others. Many workloads, however, may be able to coexist well. Argon and Cesium's rigid one-workload-per-timeslice scheduler does not permit such "coexistence." Workloads that do not need protected locality do not need to be given dedicated timeslices and can share them with non-interfering workloads. Appropriately chosen workloads could benefit from sharing timeslices in two ways. First, the timeslice can be made longer and each workload sharing the timeslice can have its requests serviced during that longer window, improving their response times. Second, some workloads constructively, rather than destructively, interfere with each other. For instance, if their files are nearby on disk, seek optimizations over their combined request stream may actually result in improved performance when compared to isolation.

Preliminary results have shown the benefit of allowing appropriate mixing. Speculatively mixing workloads and observing whether they suffer or not allows compatible workloads to be identified automatically. Speculative combinations that result in reduced performance can be quickly returned to separate timeslices, while beneficial combinations can be maintained in shared timeslices.

Bibliography

[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. 2005. San Francisco, CA, 13–16 December 2005.

[3] E. Anderson. Capture, conversion, and analysis of an intense nfs workload. FAST '09. 2009.

[4] B. S. Baker, J. E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–55.

[5] J. Bruno, J. Brustoloni, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. ICMCS '99. 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. 1998. New Orleans, LA, 15–19 June 1998.

[7] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. Summer USENIX Technical Conference. 6–10 June 1994. Boston, MA.

[8] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. Symposium on Operating Systems Design and Implementation. 14–17 November 1994. Monterey, CA.

[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. 2003. Florence, Italy, 06–08 October 2003.

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

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

[12] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals. ACM Symposium on Operating System Principles. 2005. Brighton, United Kingdom, 23–26 October 2005.

[13] 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.

[14] D. Feitelson. Job scheduling in multiprogrammed parallel systems. IBM Research Report RC 19790 (87657), October 1994, Second Revision, August 1997.

[15] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–18, 1992.

[16] W. Fernandez de la Vega and V. Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, 82:93–101, 1998.

[17] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9:849–59, 1961.

[18] A. Gulati, A. Merchant, and P. Varman. d-clock: distributed qos in heterogeneous resource environments. PODC '07. 2007.

[19] A. Gulati, A. Merchant, and P. J. Varman. pclock: An arrival curve based approach for qos guarantees in shared storage systems. SIGMETRICS '07. 2007.

[20] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[21] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128:34–57, 2001.

[22] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. SIGMETRICS/Performance '04. 2004.

[23] 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. 2001. Chateau Lake Louise, Canada, 21–24 October 2001.

[24] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. International Workshop on Quality of Service. 2004. Montreal, Canada, 07–09 June 2004.

[25] J. Katcher. Postmark: a new file system benchmark. Technical report TR3022, Network Appliance, 1997.

[26] C. Kenyon and E. Remila. Approximate strip packing. Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS). October 1996.

[27] M. Y. Kim. Synchronized disk interleaving. *IEEE Trans. on Computers*, C–35:978–988, November 1986.

[28] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. New exhaustive, heuristic, and interactive approaches to 2d rectangular strip packing. Technical report TR2003-05, Mitsubishi Electric Research Laboratories, 2003.

[29] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. New heuristic and interactive approaches to 2d rectangular strip packing. Technical report TR2005-113, Mitsubishi Electric Research Laboratories, 2005.

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

[31] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. FAST. 2002. Monterey, CA, 28–30 January 2002.

[32] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: virtual storage devices with performance guarantees. Conference on File and Storage Technologies. 2003. San Francisco, CA, 31 March–02 April 2003.

[33] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. FAST '03. 2003.

[34] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Reidel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. OSDI. 2000.

[35] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, 15(3):310–19, 2003.

[36] L. W. McVoy and S. R. Kleiman. Extent-like performance from a unix file system. USENIX Annual Technical Conference. 1991. Dallas, TX, January 1991.

[37] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in large disk arrays. ICAC. 2011.

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

[39] D. L. Mills. *RFC–1305: Network time protocol (version 3)*, March 1992.

[40] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. Proceedings Real-Time Systems Symposium. 1997. San Francisco, CA, 2–5 December 1997.

[41] S. Ng. Some design issues of disk arrays. COMPCONSpring. 1989.

[42] J. K. Ousterhout. Scheduling techniques for concurrent systems. Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS). October 1982.

[43] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. Hot Topics in Operating Systems. 2005. Santa Fe, NM, 12–15 June 2005.

[44] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). SIGMOD '88. 1988.

[45] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. ACM Symposium on Operating System Principles. 1995. Copper Mountain Resort, CO, 3–6 December 1995.

[46] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. Eurosys '08. 2008.

[47] 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. 2003. Berlin, Germany, 9–12 September 2003.

[48] 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. 2002. Monterey, CA, 28–30 January 2002.

[49] 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. 1998. Madison, WI, June 1998.

[50] 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. Dublin, Ireland, 12–16 June 2006.

[51] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. 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. Saint-Malo, France, 26–30 June 2006.

[52] Transaction Processing Performance Council. Tpc benchmark c. `http://www.tpc.org/tpcc/`.

[53] Transaction Processing Performance Council. Tpc benchmark h. `http://www.tpc.org/tpch/`.

[54] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10):8–15, 1986.

[55] C. A. Waldspurger. Memory resource management in vmware esx server. Symposium on Operating Systems Design and Implementation. 2002. Boston, MA, 09–11 December 2002.

[56] R. Wijayaratne and A. L. N. Reddy. Providing qos guarantees for disk i/o. *Multimedia Syst.*, 8:57–68, January 2000.

[57] 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. 2006. San Jose, CA, 04–07 April 2006.

[58] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: storage performance as a managed resource. RTAS '06. 2006.

[59] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Reidel. Storage performance virtualization via throughput and latency control. MASCOTS '05. 2005.