RICE UNIVERSITY

Workload Shaping for QoS and Power Efficiency of Storage Systems

by

Lanyue Lu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Peter J. Varman, Chair
Professor of Electrical and Computer
Engineering and Computer Science

S. Devika
Devika Subramanian
Professor of Computer Science and Electrical
and Computer Engineering

Lin Zhong
Assistant Professor of Electrical and
Computer Engineering and Computer Science

HOUSTON, TEXAS

FEB. 2009
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
Abstract

Workload Shaping for QoS and Power Efficiency of Storage Systems

by

Lanyue Lu

The growing popularity of hosted storage services and shared storage infrastructure in data centers is driving the recent interest in resource management and QoS in storage systems. The bursty nature of storage workloads raises significant performance and provisioning challenges, leading to increased resource requirements, management costs, and energy consumption. We present a novel dynamic workload shaping framework to handle bursty server workloads, where the arrival stream is dynamically decomposed to isolate its bursts, and then rescheduled to exploit available slack. An optimal decomposition algorithm RTT and a recombination algorithm Miser make up the scheduling framework. We evaluate this framework using several real world storage workloads traces. The results show that workload shaping: (i) reduces the server capacity requirements and power consumption dramatically while affecting QoS guarantees minimally, (ii) provides better response time distributions over non-decomposed traditional scheduling methods, and (iii) decomposition can be used to provide more accurate capacity estimates for multiplexing several clients on a shared server.
Acknowledgements

First, I would like to thank my advisor Professor Peter Varman. He gave me enough freedom to choose an interesting research topic, and provided insightful suggestions and discussions to help me to identify potential research problems. He was always patient, passionate and ready to listen and think about every research problem I encountered, no matter whether it involved high level directions or low level implementation details. Without his support, the thesis would not have been possible.

I would like to thank Professor Devika Subramanian and Professor Lin Zhong for being on my thesis committee, despite their heavy teaching and research schedules. Professor Subramanian taught me the beauty of algorithms and optimality, and the realistic importance of heuristic algorithms in COMP 440, which inspired me to propose the scheduling algorithms for IO workloads. Professor Lin asked me several intriguing questions since my ELEC 599 presentation, which helped me to identify the limitations of my past work and inspired me to explore alternative better solutions.

I am also grateful to many friends at Rice: Ajay, Bo, Ehysan, Jiayang, Mian, Mike, Yang, Yi and Zheng for their useful discussions, critical questions, working together for course projects, and enjoyable conversations. These precious experiences gave me a great opportunity to learn from them and improve myself and my research.

Last but not least, I am indebted to my parents for their unconditional love and support along this long way of study abroad.
## Contents

1 Introduction 1

2 Background and Related Work 7
   2.1 Architecture of Storage System 7
      2.1.1 DAS: Direct Attached Storage 8
      2.1.2 NAS: Network Attached Storage 9
      2.1.3 SAN: Storage Area Network 10
   2.2 IO Scheduler 11
   2.3 Performance of Storage System 13
   2.4 Related Work 15
      2.4.1 QoS-aware Scheduling 15
      2.4.2 Workload Shaping based Scheduling 18
      2.4.3 Power Efficiency 20

3 Workload Shaping 24
   3.1 Overview of Workload Shaping 24
      3.1.1 Architecture of Workload Shaper 25
      3.1.2 Workload is Bursty 26
      3.1.3 Decomposition and Recombining Methods 29
   3.2 Scheduling Framework Based on Workload Shaping 32
      3.2.1 Capacity Provisioning 33
      3.2.2 Power Efficiency 35
   3.3 Workload Shaping Algorithms 41
      3.3.1 RTT Decomposition 42
      3.3.2 Recombining Algorithms 45

4 Evaluation 48
   4.1 Experimental Setup 48
   4.2 Capacity-QoS Tradeoffs 49
   4.3 Power-QoS Tradeoffs 52
   4.4 Scheduling Performance 54
      4.4.1 Response Time Distribution of FCFS 54
      4.4.2 Response Time of Shaped Workload 58
   4.5 Multi-flow Consolidation 61
5 Conclusions and Future Work

5.1 Conclusions ......................................................... 67
5.2 Future Work ......................................................... 68
List of Figures

2.1 Network Attached Storage (NAS) exports the storage data at the file system level, which can be accessed using the network file system protocol NFS or CIFS. ................................................. 9
2.2 Storage Area Network (SAN) leverages Fibre Channel fabric to connect multiple storage devices (disk arrays, tape pools), and exports the storage data at the block device level, which can be shared by different operating system platforms .................................. 10

3.1 Architecture of workload shaper providing graduated QoS guarantees 25
3.2 Shaping the OpenMail trace by Decomposition and Recombination . 27
3.3 Illustrating the Decomposition and Recombination process .......... 30
3.4 Periodic workload in servers .................................................. 35
3.5 Architecture of Multi-queue Scheduler ..................................... 37
3.6 Capacity required for different percentages of the workload to meet a specified latency bound (Financial Transaction trace) ............. 38
3.7 Runtime scheduler for storage pools .......................... 40

4.1 Power consumption for OpenMail, TPC-D and WebSearch workload . 52
4.2 Response time CDF of FCFS scheduling for different response time targets ............................................. 55
4.3 Response time CDF of FCFS scheduling for different guaranteed fractions 57
4.4 Performance comparison of FCFS, Split, Fair Queuing and Miser: WebSearch workload ............................................. 60
4.5 Capacity required for the same workloads multiplexing .......... 63
4.6 Capacity required for the different workloads multiplexing ....... 65
List of Tables

4.1 Capacity (IOPS) required for specified Workload Fraction to meet the Response Time target ......................................................... 50
4.2 Response time performance comparison for OpenMail, TPC-D and WebSearch ................................................................. 53
Chapter 1

Introduction

The widespread deployment of Internet-based services and the growing interest in application hosting have fueled the growth of large data centers with tens of thousands of computing and storage nodes. The increasing complexity of managing huge amounts of data, providing high availability in the face of hardware or software failures, and the economic benefits of resource sharing and consolidation, are driving storage systems towards a service oriented paradigm. Service providers and enterprises are increasingly deploying application workloads on shared pools of computing and storage resources in modern data centers. By consolidating the computing and storage management in a data center, several benefits arise: the ease of sharing data among multiple applications, greater flexibility of data placement and maintenance, and lower operating costs due to consolidation and efficient resource multiplexing.

Storage data centers like Amazon Simple Storage Service (S3) [1], Microsoft Windows Live Sky Dive [3] and Apple Mac Mobile Me [2], already provide simple storage services for personal and corporate clients, who purchase storage space and access bandwidth to store and retrieve their data. Since the storage resource is shared by
multiple competing clients, it is important to provide some degree of performance guarantees or Quality of Service (QoS) for the clients according to the Service-Level Agreements (SLAs) between the clients and the service provider. A SLA is a general framework between the client and storage service provider that is used to negotiate the pricing and service guarantees based on different QoS models. Such a QoS model (e.g. see [17; 26; 41]) usually includes the storage space, access performance in terms of I/O bandwidth (req/sec, bytes/sec) or latency (average or maximum request response time), and reliability and availability levels. Thus, the shared infrastructures should effectively provide the performance isolation and differentiated service for diverse clients based on their needs and willingness to pay. Without proper resource scheduling and management, certain runaway or malicious clients may send a load surge, resulting in performance degradation of the other well-behaved clients. Isolation ensures that the effects of a client's bad behavior are confined to that client. Differentiated services allow different clients to receive different guarantees.

Besides the need for performance isolation for the clients, efficient resource provisioning can bring economic benefits to the service providers. The performance SLAs typically provide clients with minimum throughput [19; 29] guarantees, or response time bounds [21; 40] for rate-controlled clients. The server must provision sufficient resources (disk capacity: IOPS) to ensure that the clients receive their stipulated performance. A fundamental challenge in data center operations is the need to deal effectively with high-variance bursty workloads arising in the network and storage
server traffic [18; 28; 38]. These workloads are characterized by unpredictable bursty periods during which the instantaneous arrival rates can significantly exceed the average long-term rate. In the absence of explicit mechanisms to deal with it, the effects of these bursts are not confined to the localized regions where they occur, but spill over and affect otherwise well-behaved regions of the workload as well. As a consequence, although the bursty portion may be only a small fraction of the entire workload, it has a disproportionate effect on performance and provisioning decisions. This "tail wagging the dog" situation forces the server to make unduly conservative estimates of resource requirements, resulting in excess resource commitments with associated monetary and energy consumption costs, and unnecessary throttling of the number of the clients admitted into the system.

Efficient storage capacity provisioning not only saves the cost of purchase of hardware, but also the power consumption of the data center. The power density of modern servers in data center grows quickly, even up to 700 W/ft² [6], raising several challenging issues: the cost of energy consumption, the cost of cooling, environmental pollution, and secondary thermal effects. A typical data center with 1000 racks and 10MW total power consumption, costs $7M for power and $4M-$8M for cooling per year, with $2M-$4M of up-front costs for cooling equipment [39]. Among the different components of the data center, the storage system accounts for a significant percentage of the total power consumption [44]. All of these issues result in strong incentives and motivations for developing resource efficient storage systems for current
In this thesis, we present a novel approach to improve client performance and slim resource provisioning for data centers. In our approach we modify the characteristics of the arriving workload so that its behavior is dominated by the majority well-behaved portion of the request stream; the portions of the workload comprising the tail are identified and isolated so that their effects are localized. This results in more predictable behavior, and significantly lower resource requirements. The performance SLA consequently is specified by a distribution of response times rather than a single worst-case measure. By relaxing the performance guarantees for a small fraction, a significant reduction in server capacity can be achieved while maintaining stringent QoS guarantees for most of the workload. For instance, rather than specifying a single upper bound $r$ on the response time for all requests, a client may relax the requirements and instead require that 99% of the requests meet the bound $r$ and the remaining requests meet a more relaxed latency bound $r' > r$, or may be served in a best effort manner. This approach can provide significant benefits to the service provider since the worst-case requirements are usually determined by the tail of the workload. The server can pass on these savings by providing a variety of SLAs and pricing options to the client. Storage service subscribers that have highly streamlined request behavior, and who therefore require negligible surplus capacity in order to meet their deadlines, can be offered service on concessional terms as reward for their "well-behavedness".
This thesis makes the following specific contributions. (a) We present a new framework for run-time scheduling a client's workload based on decomposition and recombination of the request stream. This reshaped workload helps localize the effects of bursts so that a large percentage of the workload has superior response time guarantees, while keeping the behavior of the tail comparable to that achieved by traditional methods. (b) The resource requirements for the reshaped workloads are shown to be significantly lower than that for the original workload, since they are closer to the average rather than the worst-case requirements. This translates into reductions in provisioned capacity, and reduced energy consumption as well. (c) Finally, we show how the framework can be used to improve resource estimates of multiple concurrent clients. Due to statistical variations, the peak inputs of the workloads are unlikely to line up simultaneously. Estimates based on simple aggregation of the requirements of each client therefore tend to overestimate the requirements significantly, but estimating the benefits of multiplexing is difficult [27]. We show that aggregation based on the capacity of the reshaped workload provides more realistic estimates of resource requirements, compared to dealing with the unshaped workload.

The rest of the thesis is organized as follows: Chapter 2 describes the background and discusses related work. Architecture and performance models of storage systems are introduced, and current approaches for QoS scheduling and improving the power efficiency of storage systems are described. Chapter 3 presents our workload shaping framework. Section 3.1 introduces the overview, including workload characteristics,
the architecture of the shaper, and the high level illustration of decomposition and re-
combining methods. Section 3.2 presents the potential applications of the framework
for efficient capacity planning, graduated QoS performance guarantees, and power
conservation of storage systems. Section 3.3 describes the detailed workload decom-
position and recombining algorithms, and proof of optimality of the decomposition
algorithm. Chapter 4 presents the evaluations of the effectiveness of workload shap-
ing for scheduling performance, capacity and power efficiency by using several real
storage workload traces. Finally, Chapter 5 concludes the thesis, and discusses the
limitations and future work.
Chapter 2

Background and Related Work

This chapter describes the background and the related work. The background introduces the architecture and performance models of storage systems. The related work presents different scheduling algorithms used for storage systems, including QoS-aware scheduling, size-aware scheduling and traffic shaping based scheduling; and various methods that have been proposed to improve the power efficiency of storage systems, involving both hardware and operating systems.

2.1 Architecture of Storage System

The rapid growth in the volume of data required by modern applications such as web search engines, online video and network games, has transformed the data storage industry, which is quickly becoming one of the most dynamic segments of the information technology infrastructure. Driven by the growing requirements of high performance, reliability, efficient and flexible sharing, and simpler management and maintenance, the architecture of storage systems is undergoing tremendous changes. Storage systems have evolved from the familiar direct attached disk (DAS storage)
to hundreds or thousands of disks in high-end storage array connected via high-end storage area networks (SAN storage). The storage system organizations can be categorized into three popular architectures: (a) Direct Attached Storage (DAS), (b) Network Attached Storage (NAS) and (c) Storage Area Network (SAN). An overview of these architectures is described as below.

2.1.1 DAS: Direct Attached Storage

As the most familiar storage system organization, DAS is widely used in personal computers and small companies or institutions servers. A commodity disk or disk array is directly connected to the local host computer or server using the standard interfaces (such as SCSI, IDE), which is different from networked storage system [12; 42]. Even in current days, DAS still consists of a large percentage of personal computers and enterprise infrastructure due to its low initial cost.

For consideration of economic factors, DAS is preferred for a small amount of data accessing or sharing. For example, the institution departments or startup companies may utilize DAS based servers for web and email services, with a good balance between the cost and performance. Beyond this cost efficiency benefit for personal and small scale usages, DAS also has several obvious limitations. In DAS, the clients can only access the storage data through the server. If the server crashes due to software or hardware errors, the underlying storage data will be unavailable to the clients. For scalability, the complexity of administration and maintenance grows rapidly when the data requirements increase, such as adding a new drive to the existing system,
removing a failed drive and so on. In addition, since DAS is not networked storage, the free storage space and capacity cannot be used by other peers. These limitations directly motivate the networked storage systems, such as NAS and SAN.

2.1.2 NAS: Network Attached Storage

![Diagram of Network Attached Storage (NAS)](image)

**Figure 2.1:** Network Attached Storage (NAS) exports the storage data at the file system level, which can be accessed using the network file system protocol NFS or CIFS.

NAS is a special storage device or box which is dedicated to exporting the data at the file level over a network [12; 37]. Internally, a NAS device consists of both file system and the underlying drives as shown in Figure 2.1. In terms of the hardware components, the NAS device contains CPU and memory for file transaction processing and caching purposes, and back end storage devices, such as a set of disk arrays. The clients can access the files in the NAS device via a network file system protocol, such as NFS (for Unix) or CIFS (for Windows), without the need of managing the file.
A NAS system is easy to manage, since it already implements lots of the file system features, such as checkpoint snapshotting, data backup, data integrity checking and so on, reducing the maintenance burden of the clients. Unlike DAS which only can support a single host operating system, NAS device can serve files across a mix of operating system platforms, including Unix, Mac and Windows clients over the network. The performance and reliability of NAS are also improved by leveraging high end disk arrays (such as RAID) as the underlying storage devices.

2.1.3 SAN: Storage Area Network

![Storage Area Network (SAN) diagram]

**Figure 2.2:** Storage Area Network (SAN) leverages Fibre Channel fabric to connect multiple storage devices (disk arrays, tape pools), and exports the storage data at the block device level, which can be shared by different operating system platforms.
SAN is a dedicated storage network that connect heterogeneous storage devices together and provide high performance data transfers for the front end servers as shown in Figure 2.2. The storage devices (such as disk arrays, tape pools) are connected to the servers using specific high speed interconnect technologies, such as Fibre Channel (FC) or iSCSI [12; 37]. SAN exports block I/O services rather than file access services of NAS. Fibre Channel fabric is a specially designed high speed interconnect technology for reliable communications of storage systems.

Compared with NAS which is the ideal choice for data processing at the file system level, SAN is faster at block level data transfer. It is a better choice for I/O intensive applications, such as online video and picture broadcast, database transactions, web search engines. Since SAN exports the raw device at the block level in a distributed environment, this is very useful for storage virtualization in data centers. In a single disk array, the underlying disks can be abstracted as different logical units (LUN), such as RAID 1, RAID 5 or JBOD. In multiple distributed disk arrays, a single logic unit (LUN) can be mapped to several disk arrays in different places where the remote disks can appear as the local disks for the clients. This will improve the efficiency of resource scheduling and consolidation.

2.2 IO Scheduler

Given a workload, it would be helpful to provide IO scheduler with both the application level and low block device information. Application level information, such
as client ID and request priority, can help the IO scheduler to provide differentiated services for different clients. Block level information, such as request size, address and type, will help the scheduler to improve the system throughput by exploring the locality among different outstanding requests. Thus, the IO scheduler should be implemented in the appropriate layer within the whole system stack, which is dependent on the storage system architecture adopted.

For single systems with DAS, the appropriate place for IO scheduler is the block device layer which sits under the file system and above the device driver layer in the I/O stack. In Linux systems, the IO scheduler is called elevator, which intercepts the IO request at the block device, uses the process information to classify the requests, and the block information to reorder them. The elevator scheduling algorithms have four options, Anticipatory, Complete Fair Queueing, Deadline and Noop. However, these I/O schedulers only provide limited or coarse grained QoS performance guarantees. For example, Deadline scheduler assigns a deadline of 500 ms for all read requests and 5 s for all the write requests, without consideration of performance guarantee from the client or process’s perspective. Thus, any new IO scheduler with different QoS performance targets for specific applications requirements can be implemented in block device layer instead of using the above four schedulers.

Since NAS box contains the file system implementation and low block device layer, the IO scheduler can be integrated into NAS’s block device layer. For the application level information, the IP address or ID of the clients can be attached to the requests.
Once the requests arrive at the block device layer, the scheduler can interpret the client information and schedule accordingly to performance requirements of different clients. Current commodity NAS devices only provide load balance in the scheduling layer, without support of fine grained QoS guarantees.

SAN usually uses the high-end disk arrays as the storage devices, which are connected by Fibre Channel switches. When the IO requests arrive at the storage array, the controller (with dedicated CPU and memory), does the IO processing and caching, issues the request to the underlying disks. Inside a single storage array, the scheduler can be implemented in the controller software layer, which has full knowledge of the IO requests. SAN uses Fibre Channel switches to connect multiple storage arrays. In such an environment, one possibility is that the IO scheduler can be implemented at the switches level, which schedules the requests across multiple distributed disk arrays. Alternatives include a separate meta server to schedule the request, or using a distributed feedback monitoring of the load on the array to regulate the request rate at the host [20].

2.3 Performance of Storage System

The two most common performance measures of a storage system are throughput (also known as IOPS) and response time (also known as latency). When the client application needs to access the data, it will issue IO requests through the file system interface to the underlying storage system, such as a disk or disk array. The storage
system queues the outstanding requests while they wait for service. When the underly­ing device is free, it will select a request from the queue and send it to disk for service.

Response Time is the time from the client sends the request to the queue until the storage server finishes the request. In other words, response time consists of two parts: queueing time and service time. Queueing time is the time spent in the queue waiting for service. Service time is the sum of mechanical delays and data transfer time. Mechanical delays include: (a) seeking time: the time to move the disk head to the desired track (average seek time of commodity disk is from 3ms to 12ms); (b) rotational time: the time to rotate the desired sector under the disk head (which is related to the RPM of the disk); (c) transfer time: the time to transfer data from the disk platters through the disk head which is dependent on the transfer rate (MB/s) of the disk and the delay in transfer data over the I/O bus.

Throughput reflects the average number of requests completed by the storage server over a specific time interval. There are two common measures of throughput in a storage system: IOPS and MBPS. IOPS is the number of I/O requests finished per second. MBPS is the number of megabytes transferred from disk per second. IOPS depends on the average size of a request while MBPS depends on the size of a block. Many factors affect the throughput, such as sequentiality in the request pattern, synchronous or asynchronous write, ratio of read/write request and caching.
2.4 Related Work

In this thesis we focus on workload scheduling algorithms and power efficiency of shared storage systems. The related work in these two areas is presented in detail below.

2.4.1 QoS-aware Scheduling

In shared storage systems, multiple request from different applications or clients compete for the shared I/O bandwidth. Each client has its own performance requirements (SLA), such as throughput guarantees and bounds on the response time. For instance, a file transfer client may request a guaranteed throughput of 1 MB/sec or a transaction processing application may request 1000 IO/sec and maximal response time of 100 ms for each request. QoS-aware scheduling algorithms focus on providing a degree of performance guarantee for different clients in a shared environment.

**Throughput Guarantee:** Fair queuing algorithms have been developed for network bandwidth multiplexing. Lately, these have been adapted for use in storage and server environments. WFQ [14], WF²Q [7] and SFQ [19] allocate the shared disk bandwidth proportionally for the flows based on their assigned weights. The basic idea is that assign each request from each flow with time tags (reflecting priorities) which are used to determine the order of dispatching the requests. The fundamental results in this context is based on the simulation of the ideal fluid resource multiplexing, known as Generalized Processor Sharing(GPS) [34]. For example, suppose
two flows compete for a shared disk bandwidth of 100 IOPS with weights of 0.2 and 0.8. Then a fair queuing algorithm will give the two flows 20 IOPS and 80 IOPS accordingly, as long as they are both backlogged.

The limitation of a fair queuing scheduler is that there is no independent control or guarantees for the response time of the request flow. The response time incurred by a request is inversely related to its bandwidth allocation. However, with the same bandwidth allocation, different request arrival patterns can result in very different response time distributions. Thus fair queuing scheduling is not sufficient for the applications which requires response time guarantees, such as real-time applications.

**Response Time Guarantee:** Algorithms aim to provide statistical guarantee of the response time for the flows.

Facade [29] utilizes the earliest deadline first (EDF) algorithm to schedule the requests. At runtime, it monitors the workload, collects I/O performance statistics, and periodically adjusts the queue length accordingly to the request arrival rate. SLEDS [9] uses a leaky bucket filter to shape and throttle the I/O flows based on the performance feedback collected. Basically, if a flow sends requests too fast, it will throttle the flow so that it sends fewer requests and the existing resource is sufficient to provide the response time guarantees.

Although the above algorithms provide response time guarantees for requests, they usually need to over-provision the resource to guarantee that all the requests meet their deadlines even in the worst case. Also, throttling the flows is not work
conserving, even if surplus resources are available for free.

Guaranteeing both throughput and response time independently: For the applications requiring independent throughput and response time guarantees, only either guarantee may not be enough. For example, several research projects try to provide both the throughput and response time guarantees, provided the workload conforms to certain traffic model.

pClock [21] monitors the request arrival pattern and checks for its conformance to a leaky bucket model. The throughput target is guaranteed by leveraging fair queuing scheduling. Guaranteeing the response time requires provisioning enough bandwidth based on the worst case estimation of the traffic arrival in the leaky bucket model, and scheduling the requests in a latency sensitive manner. Fahrrad [36] proposes a disk utilization reservation based scheduler for the periodic real time application. Basically, it proportionally allocates the disk time to different flows based on their periodic arrival pattern or response time target. Thus each request has a deadline, and the scheduler uses earliest deadline first scheduling to guarantee response time.

The limitation of the above scheduling algorithms is the assumption that the traffic is constrained by a worst case model, such as a leaky bucket or periodic pattern. But the real workload is usually unpredictable and bursty [18; 38], resulting in significant amount of over-provisioning.

Our work differs from the QoS-aware scheduling above in the performance QoS model, which in turn determines the efficiency of resource provisioning and the
scheduling policy. Previous QoS models for servers provide a single performance target such as minimum throughput or maximal response time, for 100% of the workload (that is for all requests). Due to the bursty and unpredictable nature of real workloads, the service providers must provision sufficient resources for the worst case, such as peak request rate, to ensure that the clients receive their stipulated performance. This over-provisioning results in low resource utilization and efficiency. In contrast, we focus on improving capacity provisioning of the shared storage system by shaping storage workloads to provide graduated, distribution based QoS guarantees. Under this new QoS model, we decompose the workload to filter out bursts, and schedule the partitioned workload in a resource efficient way with distribution based QoS guarantees.

2.4.2 Workload Shaping based Scheduling

Traffic Shaping in Network: Considerate body of related work can be found in the literature on network QoS [15] where traffic shaping is used to tailor the workloads to fit QoS-based SLAs.

Typically, arriving network traffic is made to conform to a token-bucket model by monitoring the arrivals, and dropping requests that do not conform to the bucket parameters of the SLA. Alternatively, early detection of overload conditions is used to create back pressure to throttle the sources [16]. Techniques leveraging statistical envelopes have been proposed [27] to reshape inbound traffic and to allocate resources in network systems in order to achieve probabilistically bounded service delays, while
simultaneously multiplexing system resources among the requesters to achieve higher utilization.

In storage workloads, request dropping is not a viable option since the protocols do not support automatic retry mechanisms, and throttling is difficult in an open system and can lead to loss of throughput in disks and storage arrays. Thus, these traffic shaping methods cannot be used in storage systems.

Size-aware Scheduling: Considerable amount of previous work has been devoted to the designing optimal size-aware schedulers to improve performance [24; 43; 30] in Web servers.

The basic idea is to separate jobs in terms of their size to avoid having short jobs getting stuck behind long ones. The SRPT scheduler [24] gives preference to jobs or requests with short remaining processing times to improve mean response time of Web servers. In a clustered server environment, D_EQAL [43] utilizes the size-based policy to assign the jobs to different servers in terms of size distribution, and further enhances this by considering the autocorrelation property of the workload to deliberately unbalance the load to improve the performance. Swap [30] also leverages the size-autocorrelation property of the jobs to do an online simulate the Short Job First scheduler and delay the long jobs in preference to short ones.

Our scheduling framework is designed for storage systems, where the request sizes are not as diverse as Web applications. The big requests are already partitioned by the OS or storage device driver into smaller-sized block requests, such as up to 32KB.
Our work differs from the above works by considering the correlation of request rate rather than the request size correlation, and then propose decomposing the workload to different classes dynamically based on their burst characteristics to improve the resource efficiency and performance.

2.4.3 Power Efficiency

Many schemes for efficient power management of storage systems have been proposed in recent years. These approaches can be categorized as follows: (a) Disk Caching and Consolidation, (b) Hardware Based Solutions, (c) File System Based Solutions. We discuss each of them in detail as below.

**Disk Caching and Consolidation**: These solutions try to serve the workload from just a few disks in the storage system with appropriate caching strategies. The aim is to reduce the load on the remaining disks sufficiently so that they can spend longer periods of time in the low-power state for power conservation.

MAID [13] uses a set of caching disks to store the recently accessed data. These cache disks are kept active while the remaining disks are kept in the low power state until a miss occurs in the cache disks. On a cache miss the run-time system will spin up the appropriate back end disk and serve the request. PDC [35] also exploits the data lifetime cycle to segregate data into popular and unpopular categories. Popular data dynamically migrates to a few "hot" disks that are kept active, displacing less popular data that are moved to disks that can be kept in the inactive state.
The above schemes rely on temporal locality and skewed data access distributions of the workload, such as the popularity-based demand for clips of a large video collection. However, the consolidation of data on a subset of the disks can lead to bottlenecks and performance degradation even when all disks are powered on, and overheads for migration between the disks can be significant. Furthermore, the applications should be able to tolerate the high latencies of a miss in the caching disk.

**Hardware Based Solutions:** These approaches explore the use of new disk drives model with multiple speeds that trade off power consumption with performance. The underlying principle is to transition a disk to a lower speed when the load is predicted to be low, and spin it up to a higher speed under heavy load.

DRPM [23] leverages the multi-speed disk model which can dynamically modulate the speed at which the disk spins, thereby controlling the power expended in the disk. The speed of the disk is determined by the dynamic workload. It decreases the disk speed when the idle time of the disk is beyond a threshold for power conservation, and speeds up the disk when the load increases to avoid performance degradation. Hibernator [44] also deploys multi-speed disks and explores the tradeoffs between the power consumption and performance. It proposes constrained optimization algorithm to find the optimal power setting (disk speed) needed to minimize energy consumption while meeting the performance guarantees.

Unfortunately such multi-speed disks are not yet available as commodity devices, and are limited to just one extra intermediate power state [33].
**File System Based Solutions:** These approaches improve the power efficiency at the file system level, by changing the way of serving the requests.

To reduce the disk positioning latencies, FS$^2$ [25] dynamically places copies of data in file system free blocks according to the disk access patterns observed at runtime. As one or more replicas can now be accessed in addition to their original data block, choosing the nearest replica that provides fastest access can significantly improve performance for disk I/O operations, which indirectly reduces the power consumption of the disk. BlueFS [32] is a distributed file system, which adaptively decides when and where to access data based on the performance and the energy characteristics of each candidate device in the distributed system. This file system is focused on mobile storage systems and mobile computing.

The above three class of power management for general storage systems are mainly based on exploiting the temporal locality and periodicity observed in workloads, where the fluctuations occur in daily, weekly or even monthly cycles. The periodic pattern provides opportunities for predicting the future traffic, thus improving the caching effects and varying the number of active servers (or storage pools) for different time periods accordingly to meet the performance requirements. By keeping the inactive storage servers in the low power (or powered down) state power is conserved during periods of low utilization. Although the longer term trends of the workload are predictable, the workload tends to be very bursty at a finer granularity, meaning that the instantaneous arrival rates in some time intervals can be higher than the long-term
rate by an order or two in magnitude. Thus estimates based on worst-case patterns
still result in significant over provisioning of capacity and increased power consump-
tion. We improve the power consumption of the storage systems by decreasing the
required capacity needed as described earlier for QoS-aware scheduling. Also, our
methods can work on commodity systems without specific file system support.
Chapter 3

Workload Shaping

This chapter describes our workload shaping framework in detail. We first present an overview of workload shaping, including the architecture of the shaper, workload characteristics, and a high-level illustration of decomposition and recombining methods. Then we present the potential applications of the framework for efficient capacity planning with graduated QoS performance guarantees, and power conservation of storage systems. Finally, we describe the detailed workload decomposition and recombining algorithms, and proof of optimality of the decomposition algorithm.

3.1 Overview of Workload Shaping

In this section, we motivate the idea behind workload shaping. The goal is to smoothen the workload to reduce the unpredictability caused by the bursty arrival patterns, which makes capacity planning difficult and degrades performance. Although the average utilization of the system tends to be low, the unpredictable bursts of high activity overwhelm server resources resulting in unacceptable performance. With traditional scheduling the effects of these bursts are not confined to the
localized regions where they occur, but spill over and affect otherwise well-behaved regions of the workload as well. Consequently, a small fraction of bursty behavior has a disproportionate effect on overall performance, as well as on provisioning and admission control decisions.

3.1.1 Architecture of Workload Shaper

By workload shaping we refer to dynamically modifying the characteristics of the arriving workload so that its behavior is dominated by the majority well-behaved portion of the workload; the portions of the workload comprising the tail are identified and isolated so that their effects are localized. This results in more predictable behavior, and significantly lower resource requirements. The shaping procedure consists of two complementary operations: \textit{decomposition} and \textit{recombination}, as shown schematically in Figure 3.1.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{diagram}\caption{Architecture of workload shaper providing graduated QoS guarantees}
\end{figure}
In the decomposition phase, the workload of a single application (or client) is partitioned into two (or more in general) classes with different performance guarantees. The requests belonging to the different classes are directed to separate queues. In the scheme shown in Figure 3.1 there are two classes, identified by queues Q1 and Q2 respectively. In this example, requests belonging to Q1 will be guaranteed a response time or delay $R_1$ and requests in Q2 are served in a best-effort fashion. In the recombination phase the requests of the two classes are multiplexed in a suitable manner to satisfy the individual performance constraints. Different scheduling algorithms which provide different response time distributions for the tail of the distribution can be used in this phase. These will be discussed and evaluated later in Sections 3.3 and Section 4 respectively.

3.1.2 Workload is Bursty

A major challenge in data center operations is the need to deal effectively with high-variance bursty workloads arising in the network and storage server traffic [18; 28]. Since the instantaneous arrival rates can be significantly higher than the average long-term rate, provisioning based on worst-case traffic patterns result in onerous capacity and power consumption requirements. Furthermore, these local bursts can spill over and affect the whole workload's performance. Here, we use a real storage workload trace from an HP email server as an example to show how our workload shaping framework deals with the bursty workload.

Figure 3.2(a) shows a portion of an OpenMail trace of I/O requests (displayed
Figure 3.2: Shaping the OpenMail trace by Decomposition and Recombination
using aggregated requests in a time window of 100 ms). Note that the peak request rate is about 4440 IOPS while the average request rate is only about 534 IOPS. Figure 3.2(b) shows the class Q1 containing 90% of the requests after decomposing the workload using our decomposition algorithm RTT (described later). The capacity of the server is chosen so that all requests in Q1 meet a response time of 10 ms. RTT is optimal in the sense that with the same capacity, RTT maximizes the fraction of requests that will meet the response time bound. As may be seen Q1 is relatively even at this granularity; experimentally we find that this 90% of the original workload can be served to meet the response time bound with a capacity of only 1080 IOPS, compared to 9241 IOPS for the original workload. Finally, Figure 3.2(c) shows the workload following recombination of Q1 and Q2 using the Miser algorithm (described later). This algorithm monitors the slack in the arrivals where it can schedule a request of Q2 without causing any of the requests of Q1 to miss their deadline and schedules a request from Q1 at the earliest such time. Due to the online nature of the recombination process, one can argue (see Section 3.3.2) that guaranteeing all requests of Q1 when interleaving requests of Q2 is not possible in the worst case by any on-line method, without either placing restrictions on the arrival pattern of Q1 or by increasing the server capacity a small amount. We choose the latter strategy since it is under the control of the resource allocator; in Section 3.3 we quantify the amount of excess capacity required to guarantee all requests of Q1 when serving both Q1 and Q2 together.
### 3.1.3 Decomposition and Recombining Methods

Having introduced the architecture of the workload shaper and workload characteristics, we now describe a formal model for the workload and illustrate the operations of decomposition and recombining with detailed examples.

The workload is characterized by its arrival sequence that specifies the number of I/O requests \( n_i \) arriving at time \( a_i, i = 1, \cdots, N \). The Cumulative Arrival Curve (abbreviated AC) \( A(t) \) is the total number of I/O requests that arrive during the interval \([0,t]\); i.e. \( A(t) = \sum_{j=1}^{i} n_j \), where \( a_i \leq t < a_{i+1} \). Figure 3.3 (a) shows the AC as a staircase function with jumps corresponding to the arrival instants. The server provides service at a constant rate of \( C \) IOPS as long as there are unfinished requests. The Service Curve (SC) is shown by a line of slope \( C \) beginning at the origin during a busy period when the server is continuously busy. At any time, the vertical distance between SC and AC is the number of pending requests (either queued or at the server). Each request has a response time requirement of \( \delta \), so that requests arriving at \( a_i \) have a deadline of \( d_i = a_i + \delta \). If the number of pending requests exceeds \( C \times \delta \) it signals an overload condition. Since at most \( C \times \delta \) requests can be completed in time \( \delta \), some of the requests pending at an overflow instant must necessarily miss their deadlines. In Figure 3.3 (a) the line above and parallel to the Service Curve is an upper bound on the amount of pending service that can meet their deadlines \( (C \times \delta = 3 \) in this example). We call this the Service Curve Limit (SCL).

The operation of a decomposition algorithm can be described easily with respect to
Figure 3.3: Illustrating the Decomposition and Recombination process

the Service Curve Limit. The goal is to identify requests to drop from the workload (in actuality dropped requests are merely moved to Q2 and served from there). Consider time instants like 2 and 3 in Figure 3.3 (a) where the AC exceeds SCL. From the previous discussion, requests exceeding the SCL limits cause an overload condition and some requests must be dropped in order for the rest of the requests to meet their deadline. If requests are dropped from the workload, the AC shifts down by
an amount equal to that removed. This is shown in Figure 3.3 (b) which shows the situation following the removal of 1 request at time 1 and another at time 2. As can be seen the modified AC lies below the SCL which means that all requests in the new AC will meet their deadlines. A different choice of removing two requests is shown in Figure 3.3 (c), where one request each at times 2 and 3 are removed. One can argue that for the given capacity and response time requirements, at least two requests in this workload will miss their deadlines (as in the two choices mentioned above). On the other hand dropping two requests at time 1 is a poor choice, since a request arriving at time 3 will still miss its deadline. Note also that the decomposition method needs to be online in that it needs to make a decision on whether or not to drop a request based on the past inputs only, without knowing the future patterns of requests. We shown in Section 3.3 that our decomposition algorithm RTT satisfies these properties: it is online and minimizes the number of dropped requests for a given capacity and deadline.

We now describe the operation of a recombination algorithm. The goal is to serve the overflowing requests that have been placed in Q2 concurrently with the guaranteed requests in Q1. For instance, in Figure 3.3 (d) the two requests that were dropped at times 2 and 3 are scheduled from $Q_2$ at times 4 and 5 when there is slack in the server. Several alternative strategies with different tradeoffs can be employed for the recombination. One simple approach is to offload the overflowing requests to a separate physical server where they can be serviced without interfering.
with the guaranteed traffic (this is similar in principle to the write offloading strategy in [31] where bursts of write requests are distributed to a number of low-utilization disks for service). In cases where this offloading is not feasible, perhaps due to lack of a suitable off-load server or the need for dedicated resources on the main server, a good strategy is to treat the two parts of the workload as independent and multiplex them on the same server using a Fair Queuing scheduler to keep them isolated. This approach actually has significant capacity benefits over the dedicated offload server approach as we show in Section 4, due to the benefits of statistical multiplexing. In particular the overflow workload is active only during bursts and the capacity during idle periods can be profitably used by the guaranteed portion of the workload to improve its response time profile. We also propose a new slack-based scheduling algorithm to combine the two portions of the workload. This method called Miser, allows better shaping of the tail of the workload than a Fair Queuing Scheduler, but may in the worst-case slightly increase the fraction of requests missing their deadlines. We provide a theoretical upper bound on the amount of additional capacity required by Miser to prevent this from ever occurring.

3.2 Scheduling Framework Based on Workload Shaping

This section presents the potential applications of our scheduling framework based on workload shaping, including the efficient capacity provisioning for single client and multiple clients, and power conservation under response time distribution based QoS
3.2.1 Capacity Provisioning

In this section we address the issue of how much server capacity needs to be reserved in order to meet a client's requirements. We consider both the cases of provisioning for a single client and for multiple, concurrent clients.

**Single Client:** We profile the workload to determine the capacity reservation needed to meet a stipulated QoS requirement; i.e. Given a response time bound δ, find the minimum server capacity C required to guarantee that a specified fraction f of the requests of the given workload meets their deadlines. Decomposing the workload in this way results in a much smaller server capacity requirement albeit less than 100%, while still maintaining a high QoS.

Although it is possible to find direct methods for the optimization problem stated above, we found that a deterministic search of the solution space provided the answers with low computational overhead even for very large traces. We search the space as follows. For a given C and δ we use the RTT algorithm (detailed in Section 3.3) to find a decomposition that maximizes the number of requests meeting their deadlines. If the fraction meeting the deadline is higher than the required fraction f we reduce the capacity and try again; else we increase the capacity and retry. By performing a binary-search we converge rapidly (within $O(\log C)$ iterations) to the desired minimum capacity $C_{\text{min}}$ required to guarantee the specified response time for a fraction f of the workload.
We provision a capacity of $C_{\text{min}} + \Delta C$, where the latter is used to prevent starvation of the requests in Q2. In our experiments an additional capacity of $\Delta C = 1/\delta$ was found to be sufficient to obtain good performance for the entire workload.

Multiple Concurrent Clients: In a data center environment, the service provider needs to provision sufficient resources for several clients simultaneously sharing the system. Accurate provisioning is an extremely difficult problem and several approaches have been proposed [27]. A brute-force approach is to estimate the worst-case capacity required for each client and then reserve at least that much capacity for each of the clients. This approach results in poor server utilization and overly cautious admission control policies. There are two main problems: first, as we noted earlier, the worst-case capacity requirements of a client are usually several times of that required for the average workload; secondly, adding the individual capacity requirements presumes that the worst-cases of all the individual workloads line up simultaneously, an extremely unlikely situation in practice. Different statistical QoS approaches have been proposed to address this issue usually based on statistical assumptions of the arrival process, to analyze the overload probabilities.

We argue that using the capacity estimate of the reshaped workload not only reduces the capacity provisioning for a single client, but also can provide a good estimate of the capacity required for multiple clients as the sum of these individual capacities. Intuitively this is because the variance in the individual workloads have been reduced by reshaping, and worst and average cases have become closer to each
other. We evaluate this in Section 4 and show that using the aggregated requirements of the reshaped workloads provides a very good estimate of the capacity needed for multiplexing multiple concurrent clients.

3.2.2 Power Efficiency

Many proposals for power management of general servers and storage systems are based on exploiting the temporal periodicity in workloads [10; 11; 44]. A general stylized form is shown in Figure 3.4. The load fluctuates in daily, weekly or even monthly cycles, with periods of high load and periods of low load. The periodic pattern provides opportunities for predicting the future traffic and varying the number of active servers (or storage pools) accordingly, to meet the performance requirements in different time periods. By keeping the inactive servers in the low power (or powered down) state, power consumption is reduced during periods of low utilization.

![Periodic Workload](image)

**Figure 3.4:** Periodic workload in servers

At the end of a time epoch, a prediction of the load in the next epoch is made,
and enough servers are spun up (or down if transitioning to a low power epoch) to satisfy the performance QoS in that interval. Although the longer term trends of the workload are predictable, the workload tends to be very bursty at a finer granularity, meaning that the instantaneous arrival rates in some time intervals can be higher than the long-term rate by an order or two in magnitude. Thus during the high load period, estimates based on worst-case patterns result in significant over provisioning of capacity and increased power consumption. On the other hand, spinning up a powered-down commodity disk can take tens of seconds, and starting a server from the sleeping state needs up to several minutes to power on and warm up; hence, changing the number of active servers dynamically at a fine granularity is not a viable option. Consequently, to meet the QoS performance requirements, a large number of servers, (in some cases maybe all the servers), are always kept powered on, although they are critical only for short periods of bursty or worst-case activity. This results in significant power consumption even if most of the time the workload is relatively low. By shaping the workload one can keep the number of active servers small, while providing an improved and quantifiable performance profile.

The scenario above motivates our performance models, which explore a new trade-off between the performance and power consumption by shaping the workload to account for bursty traffic.
Performance Model

A general form of graduated QoS requirements for a storage server is described by its Response Time Distribution (RTD), a statistical distribution of its response time requirements. RTD is the general extended version of QoS model in Figure 3.1 of Section 3.1. An $n$-tier RTD is a set of $n$ pairs $\{(f_i, R_i) : 1 \leq i \leq n\}$, which specifies that a fraction $f_i$ of the workload's requests must have a response time $R_i$ or less. An RTD specifies a lower bound on the cumulative response time distribution achieved by that workload. A simple SLA in which 100% of the requests are guaranteed a maximum response time $R_1$, corresponds to an RTD with $n = 1$. A 3-tier RTD $\{(0.9, 20\text{ms}), (0.99, 50\text{ms}), (1.0, 500\text{ms})\}$ indicates that no more than 10% of the requests can exceed 20 ms latency, and no more than 1% can exceed 50 ms, while all request must be served within a 500 ms response time.

Figure 3.5: Architecture of Multi-queue Scheduler

Figure 3.5 shows the system organization of an $n$-tier RTD. The request stream of an application arriving at the workload shaper is partitioned into different classes $W_i$.
through \( W_n \), and directed to separate queues. Then the queues are multiplexed on the storage pools, that serve the requests with response time guarantees to each class. In this thesis we concentrate on a two-tier RTD architecture as shown in Figure 3.1 of Section 3.1; the extension to multiple QoS tiers can be implemented recursively. In this case, the workload \( W \) is partitioned into two classes \( W_1 \) and \( W_2 \) that will be referred to as primary and secondary classes respectively.

![Figure 3.6](image)

**Figure 3.6**: Capacity required for different percentages of the workload to meet a specified latency bound (Financial Transaction trace)

Figures 3.6(a) and 3.6(b) show the QoS variation of the Financial Transaction workload from UMass Storage Repository [5] as the capacity is varied. They show the server capacity in IOs/sec (IOPS) needed for a fraction \( f \) of the requests in the workload, to meet response time bounds of 50 ms, 20 ms and 10 ms, for \( f \) between 90\% and 100\%, and 99\% to 100\% respectively. As can be seen, the capacity required falls off significantly by exempting between 1\% and 10\% of the workload from the response time guarantee. For a 10 ms latency, the capacity increases 7.5 times (from 200 IOPS to 1500 IOPS) when \( f \) increases from 90\% to 100\%, and by a factor of 4.2
in going from 99% to 100%. Corresponding capacity increases by factors of 5.0 (10%) and 3.5 (1%) for a response time of 20 ms can be observed. In fact, for response times of 10 ms, 20 ms and 50 ms, the capacity increases in going from 99.9% to 100% guarantees are by factors of 3.0, 2.7, and 1.6 respectively. Similar trends for other storage workload are noted in our experiments, and presented in detail in Chapter 4.

These experiments provide strong empirical evidence of the bursty nature of storage traces, and show the significant potential for optimizing capacity and power consumption using a graded QoS policy. Exempting even a small fraction of the workload from the response time guarantees can substantially reduce the capacity required, especially for aggressive service with low response time requirements. Motivated by this, we apply this model for reducing power consumption.

Power Model

Figure 3.7 shows the basic architecture of the target storage system logically organized as multiple storage pools. A pool may be considered as a logical volume that stores the entire data set. For reliability and performance, data is replicated and stored multiply in several pools. A pool is simply assumed to be made up of commodity disks connected by a high speed SAN. A disk may be in any of three states: sleep, idle or active. In the idle state the disk is powered on but is not actively seeking, while in the active state it is performing a read or write operation. When in sleep mode, the disk is assumed to consume negligible power. The energy consumption of a single disk $E_{disk}$ is calculated by weighting its power consumption in a particular
mode by the time spent in that mode. The total energy is the sum of all the disks,
\[ \sum_i E_{disk}(i). \]

\[ E_{disk} = t_{active} \times P_{active} + t_{idle} \times P_{idle} + t_{sleep} \times P_{sleep} \quad (3.1) \]

![Diagram](image)

**Figure 3.7:** Runtime scheduler for storage pools

At the start of an epoch, a subset of the storage pools are placed in the powered on state (ON pools) and the rest are powered down in the sleep mode (OFF pools). The number of ON pools is estimated by analyzing the workload using our workload decomposition algorithm RTT (explained in section 3.3) and the statistical QoS performance profile (RTD) discussed in Section 3.2.2. We first use RTT to statically profile the workload to get the capacity requirement \( C_{total} \) for providing the QoS guarantees for the workload during this epoch. If the capacity of each pool is \( C_{pool} \), then a conservative estimate of the number of pools that must be powered ON during this epoch is \( \lceil C_{total} / C_{pool} \rceil \). Using the example in Figure 3.6, it requires a server capacity of \( C_{total} = 1500 \) IOPS to guarantee a 10 ms response time for 100% of the workload, while satisfying 90% of workload with a 10 ms deadline only requires
a capacity of 200 IOPS. The remaining 10% is provided a much larger deadline (or is classified as best effort), and uses either the spare capacity from that provisioned for the 90% or a small additional amount. Suppose, for instance, we assume each pool has capacity \( C_{pool} = 200 \) IOPS, then provisioning 100% of workload needs 8 pools while provisioning 90% of workload only needs 1 active pool while keeping the remaining 7 pools in the OFF states for power conservation.

The RTT decomposition algorithm partitions the workload to different classes at runtime to obtain the required 90%-10% split, and issues them to the underlying storage pools. Within the collection of ON pools, requests are sent to the disks in one pool as long as it can handle the workload. This allows the disks in the remaining pool to stay in the lower power idle state, until forced to become active to serve a request from an overloaded disk among the currently active disks.

### 3.3 Workload Shaping Algorithms

The system model is shown in Figure 3.1 of Section 3.1. The workload shaper maintains two queues \( Q_1 \) and \( Q_2 \). The primary queue \( Q_1 \) has bounded length to control the latencies of requests accepted into it. The overflow queue \( Q_2 \) acts as the overflow buffer for requests that are not accepted into \( Q_1 \) because their latency cannot be guaranteed. The server has a capacity \( C \) and the response time bounds for the requests in the primary queue is \( \delta \). Section 3.3.1 presents the details and theoretical properties of the decomposition algorithm RTT. Methods for recombining the split
stream are described in Section 3.3.2.

3.3.1 RTT Decomposition

Algorithm 1: RTT Decomposition

\begin{algorithm}
\begin{algorithmic}
\Function{RTT-Decompose}{ }
\State $\max Q_1 = C \times \delta$
\If {$len Q_1 \leq \max Q_1 - 1$}
\State Add request to $Q_1$
\State Increment $len Q_1$
\EndIf
\Else
\State Add request to $Q_2$
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

The primary queue $Q_1$ has bounded length $C \times \delta$, to control the latencies of requests accepted into it. The decomposition algorithm RTT (Response Time Threshold), shown in Algorithm 1, is used to partition the requests dynamically into the two queues. The algorithm is extremely simple. If the arriving request will cause the length of the primary queue $Q_1$ ($len Q_1$) to exceed its maximum length ($\max Q_1$), the request is diverted to the overflow queue; else it joins the end of the primary queue. Despite its simplicity, we will prove below that RTT satisfies the following optimality property.

RTT Optimality Property: For a given workload, capacity and response time bound, RTT correctly identifies a maximal-sized set of requests that can meet the deadline, among all online or offline partitioning algorithms.
To show the RTT optimality, we first show that in any period that RTT is continuously busy, the number of requests it drops is the minimum possible. Lemma 1 shows a lower bound on the number of dropped requests in any interval, and Lemma 2 shows that RTT matches that bound in a busy period. Following this, we consider an arbitrary period of operation in which RTT may alternate between idle and busy periods. We show inductively in Lemma 3, that RTT cumulatively drops no more than a hypothetical optimal algorithm OPT at the end of any busy period.

Recall from Section 3.1 that $a_t$ represents a request arrival instant, and $A(t)$ and $S(t)$ represent the cumulative arrivals and service up to some time $t$. Also, define the function $sgn(x) = \lfloor x \rfloor$ for $x \geq 0$, and $sgn(x) = 0$ for $x < 0$.

**Lemma 1:** Given server capacity $C$, a lower bound on the number of requests that cannot meet their deadlines is given by $\max_{1 \leq k \leq N}\{sgn(A(a_k) - S(a_k + \delta))\}$.

**Proof:** By definition, the number of requests with deadline less than or equal to $a_k + \delta$ equals the number of requests arriving at or before time $a_k$, which equals $A(a_k)$. Similarly the maximum amount of service that can be completed by time $a_k + \delta$ is $S(a_k + \delta)$. Hence, if $A(a_k) > S(a_k + \delta)$ then $\lfloor A(a_k) - S(a_k + \delta) \rfloor$ of the $A(a_k)$ requests that arrive in the interval $[0, a_k]$ will miss their deadlines. Hence at least $sgn(A(a_k) - S(a_k + \delta))$ requests will need to be dropped in the interval $[0, a_k]$. The largest of these values over all times $a_k, k = 1, \cdots N$ is a lower bound on the number of requests that need to be dropped. □

**Lemma 2:** In any busy period $[0, a_N]$, the number of requests that RTT will drop
is no more than $\max_{1 \leq i \leq N} \{ sgn(A(a_i) - S(a_i + \delta)) \}.$

**Proof:** Let $a_k$ be the last arrival instant in the busy period at which RTT drops a request. The total service done by RTT in the interval $[0, a_k]$ is $C \times a_k$. Let the total number of requests dropped by RTT prior to $a_k$ be $\Delta$. Now $n_k$ requests arrive at $a_k$, and any requests which result in a queue length over $\max Q_1$ must be dropped at $a_k$. That is service to be dropped at $a_k$ is given by $A(a_k) - \Delta - C \times a_k - \max Q_1$. Hence the total service that cannot be completed in $[0, a_k]$ is the sum of the requests dropped at $a_k$ plus the number dropped before $a_k$ (i.e. $\Delta$), and equals $A(a_k) - C \times a_k - \max Q_1 = A(a_k) - C \times (a_k + \delta) = A(a_k) - S(a_k + \delta)$, since RTT is continuously busy in this period. The number of dropped is therefore at most $sgn(A(a_k) - S(a_k + \delta)).$ □

Let intervals $I_1, I_2, \cdots I_m$ be successive busy periods of RTT during the time $[0, T]$. In particular $I_1 = [a_1, b_1], I_2 = [a_2, b_2] \cdots I_k = [a_{j_k}, b_k], I_m = [a_{j_m}, b_m]$; RTT is continuously busy from time $a_{j_k}$ (the start of an interval $I_k$) till some time $b_k$, $b_k < a_{j_{k+1}},$ when it becomes idle; it remains idle till the start of the next interval equal to the arrival time $a_{j_{k+1}}$. The following Lemma will be proved by Induction.

**Lemma 3:** Let OPT be an optimal algorithm that drops the minimal number of requests in $[0, T]$. Then \( \forall k, 1 \leq k \leq m, \) OPT drops at least $\Delta_k$ requests in $I_k$ and incurs an idle period of at least $\eta_k$, where $\Delta_k$ is the number of requests dropped by RTT in $I_k$ and $\eta_k$ is the amount of idle time of RTT in $I_k$.

**Proof:** We prove the Lemma by induction on the interval number $k$.

**Base Case:** For the base case consider the interval $I_1$ corresponding to $k = 1$. 44
Now RTT server is continuously busy in the interval $I_1$ and the initial amount of service done by RTT at the start of the interval is zero. Now by Lemma 2 the number of requests dropped by RTT in $I_1$ equals the lower bound of the number of requests that must miss their deadline in that interval, and hence both OPT and RTT will drop $A_1$ requests. Now RTT is continuously busy throughout $I_1$ and no further work arrives till the start of interval $I_2$; the idle time cannot be reduced further.

**Inductive Step:** For the Induction Hypothesis we assume the Lemma is true for all intervals up to $I_k$ and show it holds in the interval $I_k$. The proof is similar to the base case, additionally noting that by the Induction Hypothesis, OPT has incurred no less idle time than RTT till the start of $I_k$, and hence cannot have done more service till this time. Then by Lemmas 1 and 2, OPT will need to drop at least $A_k$ requests in $I_k$ as well. □

### 3.3.2 Recombining Algorithms

We now describe several strategies for combining the workload spilt by RTT and scheduling them at the server. We describe four scheduling methods to combine the two parts of the workload. Their performance evaluation is described in Section 4.

- **FCFS:** The requests are not partitioned and serviced in an FCFS manner. This serves as a base case for the evaluation.

- **Split:** The requests are partitioned by RTT and the overflow requests in $Q_2$ are served by a separate physical server. The primary server's capacity $C_{min}$ is
Algorithm 2: Miser Scheduling

On a request arrival:
begin
  RTT_Decompose( );
  /* Compute Slack*/
  if request r in Q1 then
    \[ r_{i, slack} = \max_{Q_1} - \text{len}_{Q_1} \]
    \[ \text{minSlack} = \min\{\text{minSlack}, r_{i, slack}\} \]
end

On a request departure:
begin
  /*Dispatch a request*/
  if \( \text{minSlack} \geq 1 \) then
    Issue request from \( Q_2 \)
  else
    Issue request from \( Q_1 \)
  /*Update Slack*/
  if scheduled request \( r_i \) is from \( Q_1 \) then
    if \( r_{i, slack} = \text{minSlack} \) then
      \[ \text{minSlack} = \min_{i \in Q_1}\{r_{i, slack}\} \]
    else
      for \( \forall i \in Q_1 \) do
        \[ r_{i, slack} = r_{i, slack} - 1 \]
      \[ \text{minSlack} = \text{minSlack} - 1 \]
end

Based on profiling the workload, and a small additional amount \( \Delta C \) is provided to the secondary server.

- **Fair Queuing**: The requests are partitioned by RTT and the two queues \( Q_1 \) and \( Q_2 \) are served using a proportional share bandwidth allocator (like WF2Q [7], SFQ [19], RFQ [22]) that divides the server capacity in the specified ratio. The total capacity of the server is \( C_{\text{min}} + \Delta C \), but by sharing a single physical server we hope to leverage the benefits of statistical multiplexing.
• **Miser:** The scheduler uses slack in the scheduling of the primary queue to schedule requests in $Q_2$ as early as possible. Unlike the previous two methods, where the additional capacity $\Delta C$ only affected the performance of the requests in $Q_2$, here the two queues are more closely coupled. Due to its online nature the composite algorithm (RTT + Miser), could sometimes drop more than the theoretical minimum number of requests. We can show theoretically that if $\Delta C = C_{\text{min}}$, then this can never occur. Our simulations show that even with a small amount of additional service $\Delta C$, very few (if any) requests in $Q_1$ are delayed beyond the deadline in practice, and the tail distribution of $Q_2$ is much nicer.

Algorithm 2 shows the actions taken on request arrival and request completion at the server for the scheduler Miser. On a request arrival the routine $RTT\_\text{Decompose}$ is first invoked to classify the request. If placed in the primary queue it is assigned a slack value equal to the number of places still available in $Q_1$. A request in the overflow queue $Q_2$ is scheduled when the smallest slack value is at least 1.
Chapter 4

Evaluation

In this Chapter, we evaluate the workload shaping based scheduling framework using DiskSim [8], an efficient and highly-configurable storage system simulation tool. We implemented the RTT decomposition algorithm at the device driver level which catches all the incoming requests before they reach the underlying disks. The workload is decomposed by RTT and requests are assigned to separate queues. When the disk driver needs to dispatch a new request to the disk, our recombining scheduler is called to choose the next request for service.

4.1 Experimental Setup

We use traces of three different storage applications for our evaluation: Web Search Engine (WebSearch), OLTP application (FinTrans) and Email service (OpenMail). The traces are obtained from UMass Storage Repository [5] and HP Research Labs [4]. All of these are block level storage I/O traces. The WebSearch traces are from a popular search engine and consist of user web search requests. The FinTrans
traces are generated by financial transactions in an OLTP application running at two large financial institutions. OpenMail traces are collected from HP email servers during the servers’ busy periods.

We conducted four types of experiments: (a) measuring server capacity requirements as a function of the fraction $f$ of requests that are guaranteed a response time $\delta$; (b) the tradeoff between the power consumption and the performance guarantees under the distribution-based QoS model; (c) response time distribution obtained by a traditional FCFS scheduler that does not decompose the workload, and comparison of the response time distribution of recombination algorithms Split, Fair Queuing Schedule and Miser with FCFS; (d) capacity estimation for multiple concurrent clients using the decomposition framework.

4.2 Capacity-QoS Tradeoffs

Avoiding resource over-provisioning is a difficult problem due to the unpredictable bursty behavior of real workloads. This set of experiments explores the tradeoffs between the fraction $f$ of the workload that is guaranteed to meet a specified response time bound $\delta$, and the minimum server capacity $C_{\text{min}}$ required. The case $f = 100\%$, gives the minimum capacity required for all the requests to meet the latency bound. As $f$ is relaxed, a smaller capacity should be sufficient. Our results confirm the existence of a sharp knee in the $C_{\text{min}}$ versus $f$ relation, that shows that a very
small percentage of the workload necessitates an overwhelming capacity to meet its guarantees.

Table 4.1 shows capacity required for different fractions to meet a specified response time target for the three different workloads. Response time bounds of 5, 10, 20, 50 ms and \( f \) between 90% to 100% of the workload are considered. As can be seen in Table 4.1, the capacity required falls off significantly by exempting between 1% and 10% of the workload from the response time guarantees. For instance, with a 5 ms response time, extending the response time guarantee from 90% to 100% of the workload requires large capacity increases: almost 4 times (from 590 to 2325 IOPS) for the WebSearch workload, 7.5 times (from 400 to 3000 IOPS) for FinTrans workload, and more than 10 times (from 1350 to 13990 IOPS) for OpenMail workload. Even going from 99% to 100% the capacity required increases by a
factor of 2.4 (from 960 to 2325 IOPS) for WebSearch, a factor of 5 (from 600 to 3000 IOPS) for FinTrans and a factor of 3.5 (from 3950 to 13990 IOPS) for OpenMail. For higher response times, the capacity required also increases by significant, though smaller factors, as can be seen in the Table. For instance, for OpenMail workload, the required capacity for 100% guarantees is still several times that required to guarantee a reduced fraction: specifically, for response time bounds of 10 ms, 20 ms and 50 ms respectively, the capacity required increases 8.6, 6.4 and 4.9 times in going from 90% to 100%, and 3.1, 2.4 and 2 times in going from from 99% to 100%. The extent of burstiness (and potential for capacity savings) that can be present in the workload can be gauged by looking at the range from 99% and 100% of FinTrans workload, where increasing \( f \) from 99.9% to 100% required capacity increases by factors of 3.0, 3.0, 2.7 and 1.6 respectively for different response times.

Summarizing, the experiments clearly indicate that exempting even a small fraction of the workload from the response time guarantees can substantially reduce the capacity that needs to be provisioned. The more aggressive the QoS specifications (lower response time requirements), the greater the savings in relaxing the fraction meeting the guarantee. Even a small percentage of burst in the workload (such as 0.1%) can require a large amount of resources to guarantee the response time.
4.3 Power-QoS Tradeoffs

We evaluate the power efficiency of the workload shaping framework with OpenMail, TPC-D and WebSearch traces in this section. The test system consists of several storage pools, in which each pool contains several IBM Ultrastar 36Z15 disks as shown in Figure 3.7. For this disk model, the active power is 13.5 W and idle power is 10.2 W. In this experiment, the baseline system provisions enough capacity and power resources to serve the entire workload (100%) with a 20 ms response time guarantee. By decomposing the workload using RTT, we filter out the burstiest 1% of the workload and serve the remaining 99% with the same performance requirement as the baseline system.

![Graph showing power consumption comparison between baseline and decomposition for OpenMail, TPC-D, and WebSearch workloads.]

**Figure 4.1:** Power consumption for OpenMail, TPC-D and WebSearch workload

In Figure 4.1, we compare the power consumption of the baseline system and that obtained by the decomposition of the workloads. For OpenMail, TPC-D and WebSearch workloads, the power consumption of the baseline system is 1.93, 2.88
and 1.90 times of that obtained by decomposing the workload respectively while only serving 1% additional requests within 20 ms. Although the removed bursty part only account for 1% of the total workload in this experiment, guaranteeing these bursts with the response time bound requires several times increasing capacity as already shown in Table 4.1. Since greater capacity means that more power is needed to keeping the disks active, cutting down the capacity requirements by workload decomposition can also reduce the power consumption indirectly, as shown in Figure 4.1.

<table>
<thead>
<tr>
<th>Trace</th>
<th>&lt; 10 ms</th>
<th>&lt; 15 ms</th>
<th>&lt; 20 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>OM(base)</td>
<td>99.87%</td>
<td>99.98%</td>
<td>100.0%</td>
</tr>
<tr>
<td>OM(decom)</td>
<td>94.53%</td>
<td>98.38%</td>
<td>99.00%</td>
</tr>
<tr>
<td>TPC-D(base)</td>
<td>99.51%</td>
<td>99.94%</td>
<td>100.0%</td>
</tr>
<tr>
<td>TPC-D(decom)</td>
<td>97.78%</td>
<td>98.85%</td>
<td>99.0%</td>
</tr>
<tr>
<td>WS(base)</td>
<td>99.92%</td>
<td>99.99%</td>
<td>100.0%</td>
</tr>
<tr>
<td>WS(decom)</td>
<td>92.24%</td>
<td>98.58%</td>
<td>99.0%</td>
</tr>
</tbody>
</table>

Table 4.2: Response time performance comparison for OpenMail, TPC-D and WebSearch

The measured response times using the baseline system and the decomposed workload are shown in Table 4.2. We note that both finish their specified percentage within the guaranteed 20 ms bound. From the CDF of the response time, we can see that for response time categories of smaller than 10 ms, 15 ms and 20 ms, the results of decomposition method are very close to that of the baseline.

By employing a distribution-based QoS model, we identify a new space to optimize the tradeoff between power and performance, by noting that even a small relaxation in performance guarantees reduces not only the capacity but also the power consumption. This method is complementary with current techniques based on predicting and
exploiting periodicities in the workload patterns as illustrated in Section 2.4.3.

4.4 Scheduling Performance

4.4.1 Response Time Distribution of FCFS

The results of Section 4.2 show that meeting the guarantees of a relatively small fraction of the workload accounts for a large share of the server capacity requirement. We now investigate the effects of the bursts on the response time of the workload. In a shared data center, scheduling across clients may be done using a fair queuing scheduler or other isolating mechanisms, and scheduling at the low level of storage array uses throughput maximizing ordering among the requests in the low-level queue. However, requests of a single client are usually handled in a simple FCFS manner. The following experiments show that in the presence of bursty traffic within a single client workload, this can result in poor response time profiles. That is, the bursts in the workloads are not sufficiently isolated to prevent them from affecting the behavior of the non-bursty part of the workload, and isolation needs to be enforced specifically by a scheduler.

The cumulative response time distribution obtained for the unpartitioned workloads using FCFS scheduling is shown in Figure 4.2. Figures 4.2(a), 4.2(b) and 4.2(c) show the response time distribution for the three workloads assuming target response times of 10 ms, 20 ms and 50 ms respectively. In each case the capacity (C in the figure) is chosen so that 90% of the workload can meet the response time target (P
Figure 4.2: Response time CDF of FCFS scheduling for different response time targets
in the figure) if it were optimally decomposed using RTT.

In Figure 4.2(a), at a capacity of 417 IOPS, only 54% of the unpartitioned WebSearch workload meets a 10 ms latency bound. In contrast, in the partitioned workload 90% of the workload would meet the response time bound (see Table 4.1). The unpartitioned workload reaches 90% compliance only for a response time around 200ms. A similar behavior is shown by the OpenMail workload for a 10 ms response time bound and a capacity of 1080 IOPS. In the unpartitioned workload, only 71% of the requests meet the response time bound, and the system reaches a 90% compliance at around 90 ms, In contrast, the decomposed workload achieves 90% compliance with the 10ms latency (see Table 4.1). For the FinTrans workload, a capacity of 200 IOPS resulted in 64% of the unpartitioned workload, and 90% of the partitioned workload meeting the 10ms response time bound. In Figure 4.2(b), the response time target is 20 ms. At a capacity of 345 IOPS, only 8% of the unpartitioned WebSearch workload meets the 20 ms deadline, compared to 90% of the partitioned workload. For FinTrans and OpenMail workloads, the corresponding percentages of guarantees are 57% and 66% respectively. In Figure 4.2(c), the response time target is relaxed to 50 ms. In this case, for WebSearch only a tiny 5% of the requests meet the 50 ms deadline, compared to 90% of the partitioned workload. For FinTrans and OpenMail the corresponding numbers are still a low 29% and 55% respectively. The reason for this drop in FCFS performance is in stark contrast to the improvement in performance of the decomposed workload. With a more relaxed response time (50ms instead of
10ms), the partitioned workload can meet the same 90% compliance with a smaller capacity; however, for FCFS the smaller capacity results in the queues built up during the burst to drain slower, increasing the response time for the well behaved part of the workload as well. Thus, when the capacity provided is smaller from Figure 4.2(a) to Figure 4.2(c), the performance of FCFS becomes worse.

![Response time CDF of FCFS scheduling for different guaranteed fractions](image)

**Figure 4.3:** Response time CDF of FCFS scheduling for different guaranteed fractions

When the guaranteed percentage of the workload increases to 95% or 99%, the cor-
responding capacity needed also increases, which will improve FCFS's performance. In Figure 4.3, the performance of FCFS at a capacity for which RTT can guarantee 95% and 99% of a workload with 50 ms deadline is shown. In Figure 4.3(a), the corresponding percentages of guarantees of FCFS for WebSearch, FinTrans and OpenMail are still low: 30%, 57% and 85% respectively. In Figure 4.3(b), when the target increases to 99%, the corresponding percentages of guarantees of FCFS for WebSearch, FinTrans and OpenMail are 81%, 90% and 97% respectively.

4.4.2 Response Time of Shaped Workload

In this section, we evaluate the recombination methods discussed in Section 3.3.2, Split, Fair Queuing and Miser, and compare them with the performance of FCFS. In each case the total amount of capacity provided for the workload is held constant, equal to $C_{min} + \Delta C$; $C_{min}$ is the capacity required to guarantee the chosen fraction $f$ of the workload (as obtained from Table 4.1), and $\Delta C$ was chosen to be a small amount $1/\delta$. FCFS uses the entire capacity for the unpartitioned workload. For Split and Fair Queuing the capacity is divided in the ratio $C_{min}$ to $\Delta C$ for the primary and overflow portions of the workload respectively. In Split, the servers cannot be shared and consequently if either the main or overflow server becomes idle, the capacity is wasted even if the other part of the workload has pending requests. On the other hand, Fair Queuing multiplexes the capacity of a single server so that excess capacity can be flexibly moved from one part to the other, while guaranteeing a minimum reservation to each. Miser opportunistically uses the capacity to schedule the overflow requests.
depending on the amount of available slack.

In Figure 4.4, we evaluate the scheduling performance for WebSearch workload with the response time target of 50 ms. We can see that Split and Fair Queuing achieves the 90% target of 50 ms response time following decomposition of the workload. Miser, as noted previously, may incur some additional misses, but is still very close to the 90% target, even with just $\Delta C = 20$ IOPS additional capacity. However, FCFS can only finish 14% of the requests within 50 ms. Furthermore, FCFS has 74% of requests with response time bigger than 1000 ms, while Split, Fair Queuing and Miser have about 10%. Figure 4.4(b) shows the performance of these schedulers with percentage target 95% and $\delta = 50$ ms. Split, Fair Queuing and Miser still outperform FCFS with 95% guarantees of 50 ms response time, while FCFS finishes only 51% within 50 ms. For the response time larger than 1000 ms, Split has 4.9%, Fair Queuing has 4.1% and Miser has 4.6% of the requests respectively, while FCFS has 17.7%.

Figures 4.4(a) and 4.4(b) show that Split, Fair Queuing and Miser are better able to guarantee a higher percentage of requests with small deadlines. But Split, Fair Queuing and Miser have larger maximum response time than FCFS, because a decomposition-based scheduler will delay the burst in the workload to give good performance to other well behaved requests, leading to larger delays of the overflowing requests. But as the above figures show, the total number of long delayed requests (greater than 1s in the Figures) is less than in FCFS, even though the largest value
Figure 4.4: Performance comparison of FCFS, Split, Fair Queuing and Miser: WebSearch workload
may be higher.

Finally we compare the performance of Split, Fair Queuing and Miser. For Split, the capacity is partitioned without any sharing between the two classes, leading to very bad performance of the secondary class. In this experiment, both the average and maximal response time of secondary class in Split is an order of magnitude bigger than that of Fair Queuing and Miser. Fair Queuing assigns the weighted capacity to the two classes without any preference. The overflow class can only use the spare capacity of the primary when the latter has no requests. However, for Miser, it dynamically monitors the slack of the primary class, and uses it to improve the performance of the secondary class requests. Figure 4.4(c) shows the average and maximal response time of the secondary class of Miser normalized to that of Fair Queuing in the above experiments. We can see that for WebSearch workload, the average response time of secondary class of Miser is about 85% - 90% of Fair Queuing, while maximal response time is roughly 85% compared to Fair Queuing.

4.5 Multi-flow Consolidation

In a shared server environment, resource provisioning is usually hard to predict because of the bursty nature of the workloads. A straightforward aggregation of the reservation requirements of each client provides a simple estimate of the capacity requirements, but tends to severely overestimate the capacity, since it assumes strong correlation between the bursts of different clients. We evaluate the resource require-
ments for combinations of the same (Figure 4.5) and different (Figure 4.6) workloads based on a maximum response time of 10 ms, and compare it with the estimated value which is the sum of the individual capacities of the workloads.

Figure 4.5(a) shows the capacity needed for combining two identical workloads, for example, two copies of WebSearch workloads (same for FinTrans and OpenMail workloads). The estimated capacity for the pair of workloads is twice the capacity needed by each individual workload, because in the worst case their bursts or peaks overlap exactly. Shift-1s and shift-100s means that one workload is shifted in time by 1 second or 100 seconds, then merged with the other workload, to reflect a real multiplexing of the workloads. In Figure 4.5(a), we can see that for WebSearch, FinTrans and OpenMail, the capacity needed respectively for Shift-1s is 63%, 50% and 51% of the estimate. For Shift-100s, the capacity needed is 56%, 53% and 66% of the estimate. So, if the bursts or peaks of the two workloads are not overlapped exactly as would be, the worst case provisioning is much more than actually needed.

To avoid over-provisioning and provide a good estimate for the required capacity, we argue that capacity provisioning based on workload decomposition works well in real cases. In Figure 4.5(b) and 4.5(c), we show the capacity requirements based on decompositions of 90% and 95%, with the response time guarantee 10 ms, for the same workloads combining as in Figure 4.5(a). After decomposition, the actual capacity needed by shift-1s and shift-100s is very near the estimated capacity, with an error of 1% for WebSearch, an error of 0.1% for FinTrans and an error of 0.2% for
Figure 4.5: Capacity required for the same workloads multiplexing
OpenMail. Similar results can be found for 95%, with relative errors of 3%, 12.5% and 1% for WebSearch, FinTrans and OpenMail respectively. The decomposition process removes the most bursty part from the workload, thus the remaining part is more peaceful than the original workload as shown earlier in Figure 3.2 of Section 3.1. Thus, the estimate results based on aggregation after decomposition are very close to the real values. Since decomposition based on 90% removes more bursts than decomposition based on 95%, the errors of aggregation for 90% decomposition are smaller than 95% decomposition.

Figure 4.6(a) shows the results when combining different pairs of the three workloads. For WebSearch and FinTrans, the actual capacity needed is only 53% of the estimate, indicating considerable multiplexing gains in the combination. For FinTrans and OpenMail, OpenMail and WebSearch, the actual capacity needed is 86% and 87% of the estimate. The reason of this high real value is that the capacity needed individually by OpenMail (9241 IOPS) is much higher than WebSearch (1538 IOPS) and FinTrans (1500 IOPS), thus the resulting combined workload at least needs the amount of 9241 IOPS. The capacity provisioning based on workload decomposition also works well for combining different workloads. In Figure 4.6(b) and 4.6(c), we report the capacity requirements based on decompositions of 90% and 95%, with the response time guarantee 10 ms, for the same workload combinations as in Figure 4.6(a). We can see that after decomposition, the capacity estimate based on adding the individual capacity requirements is very close to the actual capacity.
Figure 4.6: Capacity required for the different workloads multiplexing
needed, with error of 0.3% for WebSearch + FinTrans, error of 0.05% for FinTrans + OpenMail, and error of 0.7% for OpenMail + WebSearch. Similar results can be found for 95%, with the relative errors 6.2%, 2.6% and 0.1% for WebSearch + FinTrans, FinTrans + OpenMail and OpenMail + WebSearch respectively. By removing the high variance portion of the individual workloads, the simple aggregation of the decomposed workloads provides a very good estimate for the combined workload.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we addressed the problem of resource provisioning (capacity and power) and performance degradation in storage servers caused by the bursty nature of many storage workloads. Since the arrival rates during a burst can be an order of magnitude or more than the steady state arrival rate, providing worst-case guarantees requires significant over provisioning of server resources. Furthermore, even though the bursts make up only a small fraction of the requests, their effects are not isolated but affect even the well-behaved portions of the workload.

We presented a workload shaping framework to address this problem. In our approach, the workload is dynamically decomposed into its bursty and non-bursty portions based on the response time and capacity parameters. By recombining the bursty portions to exploit available slack in the rest of the workload, the entire workload can be scheduled with much smaller capacity and superior response time distribution. We presented an optimal decomposition algorithm RTT and a slack-scheduling recombination method Miser to do the workload shaping, and evaluated it on several
real-world storage traces.

The evaluation results show significant capacity and power consumption reductions can be achieved by exempting just a small fraction of the workload from the response time guarantees. Our scheduling framework also can get better response time distributions over non-decomposed traditional scheduling methods for the same workloads. Finally, we showed how the decomposition could be used to provide more accurate capacity estimates for multiplexing several clients on a shared server, thereby improving admission control decisions.

5.2 Future Work

We have shown our workload shaping based scheduling framework in detail. However, there are still some limitations and open problems that need to be solved in the future.

First, we have not considered request dependencies (such as read after write, write after read, write after write) when the workload is decomposed and put on different queues. However, honoring the request dependencies is important for the data consistency. We plan to analyze the dependency among the requests as a constraint in the decomposition process, and schedule the requests in a dependency-aware manner to guarantee data consistency.

Second, different workloads have varying degrees of burstiness. Modeling the burst in the workload is a hot research area which can bridge the connection between
the workload and the capacity provisioning. An appropriate model may predict the capacity needed or performance results more efficiently than offline profiling.

In the future, we will implement our scheduling framework in the Linux kernel as a block device scheduler. This will help us to test our approach in real systems, and provide useful feedback and uncover new issues. The results in this thesis indicate this is a promising direction of further study.
Bibliography


[38] RISKA, A., AND RIEDEL, E. Long-Range Dependence at The Disk Drive Level.

[39] RIVOIRE, S., SHAH, M. A., RANGANATHAN, P., AND KOZYRAKIS, C. Joule-

[40] SARİOWAN, H., CRUZ, R. L., AND POLYzos, G. C. Scheduling for Quality
of Service Guarantees via Service Curves. In Proc. of the Intl. Conf. on Comp.

Systemics, Cybernetics and Informatics.

[42] Wilkes, J., Hoover, C., Keer, B., Mehra, P., AND Veitch, A. Storage,

[43] Zhang, Q., Mi, N., Riska, A., AND Smirni, E. Load Unbalancing to Improve