2014-01-01

2TL: A Raid I/O Scheduling Algorithm For Simultaneously Providing Latency And Throughput Guarantees

Yipkei Kwok
University of Texas at El Paso, ykwok2@miners.utep.edu

^{2}TL: A RAID I/O SCHEDULING ALGORITHM FOR SIMULTANEOUSLY PROVIDING LATENCY AND THROUGHPUT GUARANTEES

YIPKEI KWOK

Department of Computer Science

APPROVED:

Patricia J. Teller, Ph.D., Chair

Sarala Arunagiri, Ph.D.

Michael P. McGarry, Ph.D.

Shirley V. Moore, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

^{2}TL: A RAID I/O SCHEDULING ALGORITHM FOR SIMULTANEOUSLY PROVIDING LATENCY AND THROUGHPUT GUARANTEES

by

YIPKEI KWOK

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

August 2014

Abstract

It is increasingly common for applications to require that data read from and written to a shared storage system be delivered within a specified amount of time (usually milliseconds), called a latency requirement, or be delivered at a specified rate (measured in megabytes per second), called a throughput requirement. Given an input/output (I/O) workload, which consists of the streams of I/O requests of a set of applications, the storage system, via its I/O scheduler, is expected to simultaneously meet the workload's latency and throughput requirements. In addition, it is expected to provide performance guarantees, i.e., guarantees that it will meet a workload's latency and throughput requirements. Of course, these guarantees are provided under certain conditions associated with the storage system and the streams in the workload.

While providing throughput guarantees is a well-studied topic, providing latency guarantees requires further study. The vast majority of the existing schedulers that provide latency guarantees are "reactive" schedulers, which adjust request scheduling parameters based on stream performance. That is, when the latency requirement of a stream is not being met, a reactive scheduler adjusts its scheduling parameters, which may include increasing the service allocated to the stream, to meet the stream's latency requirement. The reactive nature of these schedulers makes it difficult, if not impossible, for them to meet latency requirements at high percentiles, e.g., for 99% of requests, to meet the latency requirement, for streams with bursty access characteristics.

This dissertation introduces ^{2}TL, an I/O scheduler for RAID storage systems that simultaneously provides latency guarantees at high percentiles as well as throughput guarantees. ^{2}TL was designed and implemented to continuously monitor the access characteristics of the latency-bound streams in a given workload and proactively adjust its scheduling parameters to meet their latency requirements. To the best of our knowledge, ^{2}TL is only the second scheduler in the literature that employs proactive scheduling to meet latency requirements

- Courier was introduced first. ^{2}TL differentiates itself from Courier in two significant ways: (1) ^{2}TL meets I/O stream latency requirements at high percentiles, and (2) it takes into consideration disk queuing, which is a technique that is employed in consolidated storage systems.

The effectiveness of ^{2}TL was evaluated through simulation using a set of synthetic and real workloads that cover a wide range of storage system access characteristics. The simulations are used to demonstrate essential properties of ^{2}TL and to compare its effectiveness with a reactive scheduler that functionally resembles ^{2}TL. When a set of conditions, which apply to schedulers that provide latency guarantees in general, are met, ^{2}TL's proactive scheduling of the requests of latency-bound streams, i.e., streams with latency requirements, allows ^{2}TL to meet the latency requirements of streams with bursty access characteristics that the reactive scheduler cannot meet. Most importantly, the simulations demonstrate that the more bursty the latency-bound streams are, the more pronounced the performance advantage of ^{2}TL. In contrast, when the latency-bound streams of a workload are not bursty or when all the streams in a workload have throughput requirements, ^{2}TL has similar performance to that of the reactive scheduler.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation introduces a new I/O scheduler for RAID storage systems, called ^{2}TL, which simultaneously provides latency guarantees at high percentiles as well as throughput guarantees. Such guarantees are essential for emerging applications with latency and throughput requirements that share a consolidated storage system and work in concert. In such an environment the I/O scheduler must simultaneously accommodate the needs of both of these types of applications, since failures in meeting their performance requirements, especially meeting their latency requirements at high percentiles, may lead to user dissatisfaction. To better understand the importance of this contribution to the state of the art, this chapter motivates our interest in the research problem that was addressed by this dissertation (Section 1.1.1), and provides the background (Section 1.1.2), including an overview of related work (Section 1.2), to understand the problem and the gap that is filled by ^{2}TL. In addition, in this chapter, we formally state the research problem and hypothesis (Section 1.3), and then briefly present ^{2}TL, the experimental methodology (Section 1.4.2) that was used to quantify and evaluate the effectiveness of ^{2}TL, and our experimental results (Section 1.4.3)). Finally Sections 1.5 and 1.6 describe the contributions made by this research to the state of the art and how the dissertation is organized, respectively.

1.1 Motivation and Background

Section 1.1.1 first motivates the need for simultaneously providing latency and throughput guarantees on storage systems that are concurrently accessed by the I/O streams of multiple applications. Since ^{2}TL is designed for RAID storage systems, Section 1.1.2 describes the

basic design and operation of these systems.

1.1.1 Motivation

It is increasingly common for applications to require that data read from and written to a shared storage system be delivered within a specified amount of time (usually milliseconds), called a latency requirement, or be delivered at a specified rate (measured in megabytes per second), called a throughput requirement. Applications that have latency requirements (latency-bound applications), e.g., online-transaction processing and rich web applications, are usually front-end, user-interacting applications. Applications that have throughput requirements (throughput-bound applications), e.g., data mining and business analytics, are usually back-end, data-intensive applications. Latency requirements need to be met at high percentiles, i.e., for a high percentage of requests, to ensure user satisfaction [20, 34]; some critical applications even have latency-requirements at above the 99th percentile [40]. Throughput-bound applications acquire massive amounts of data as input for processing and analysis. Therefore, they have throughput requirements. Since they often run in the background, their I/O performance is latency-agnostic.

Storage consolidation for latency-bound and throughput-bound applications is increasingly common for the following reasons. First, the growing popularity of storage consolidation inadvertently increases the chances that a storage system accommodates both latency-bound and throughput-bound applications. Second, latency- and throughput-bound applications are designed to work in concert in some realistic I/O environments. For example, in the data center of Facebook, front-end, user-interacting applications handle user activities, while the back-end of Facebook Insights analyzes user activities to help businesses and bloggers understand how people interact with their content [6].

A possible storage solution is to provide two storage systems, one for front-end applications and the other for back-end applications. This solution is relatively simple to configure and optimize for performance because each storage system accommodates only applications with similar I/O access characteristics and performance requirements. However, it requires

user-activity data in their analyses to be copied back-and-forth between the two storage systems. User-activity data are periodically copied from the front-end to the back-end storage system for analysis, while analysis results are copied from the back-end to the front-end storage system for users to access. As a result, when analytical results are available, they are often out-of-date due to dynamic user activities, data-analysis processing time, and data transfer overhead. Facebook concludes that user-activity data analyses conducted in any offline, periodic fashion yield poor user experiences [6].

In contrast, a consolidated storage solution that facilitates online, continuous analyses of user-activity data may potentially produce up-to-date analysis results. The analysis applications run in the background and continuously include the latest user-activity data into their analyses. Up-to-date analysis results are accessible to users immediately once available. And, of course, this solution eliminates the transfer overhead of user-activity data and analysis results. Depending on the scale of a data center, this overhead can be substantial (for instance, the Facebook data center supports at least a million user activities per second [6]). Such an I/O environment is a perfect example of one where, to produce up-to-date analysis results while maintaining the responsiveness of user-interacting applications, it is necessary for a storage system to simultaneously provide both latency and throughput guarantees to meet the I/O requirements of applications. Of course, such guarantees are subject to conditions associated with the storage system.

Accordingly, there are three reasons why it is essential to simultaneously meet the performance requirements of applications in terms of high-percentile latency and throughput that share and concurrently access a storage system:

1. Consolidated storage systems that simultaneously accommodate both latency-bound and throughput-bound applications are becoming increasingly popular.

2. Many latency-bound and throughput-bound applications are designed to work in concert.

3. Failures in meeting latency requirements at high percentiles may lead to user dissat-

isfaction.

Thus, it is essential to simultaneously meet the performance requirements of latency-bound and throughput-bound applications that share and concurrently access a storage system. To accomplish this, given an input/output (I/O) workload, which consists of the streams of I/O requests of a set of applications, the storage system, via its I/O scheduler, is expected to simultaneously meet the workload's latency and throughput requirements. In addition, it is expected to provide performance guarantees, i.e., guarantees that it will meet a workload's latency and throughput requirements. (Of course, these guarantees are provided under certain conditions.)

1.1.2 Background

A hard drive is a common non-volatile storage technology [36] that exists in many computers today. In medium- and large-scale storage systems, multiple hard drives are often organized as one or more RAIDs [31], where RAID stands for Redundant Array of Inexpensive Disks. Hard drives in a RAID are connected by a communication medium and a RAID controller. They are presented to I/O streams as a single, logical unit to enhance performance and/or data reliability.

There are different types of RAID configurations, known as RAID levels. RAID levels are designed to meet different objectives of storage systems in terms of performance, reliability, and capacity. Depending on the RAID level, data of an application may be striped and/or replicated across the disks of a RAID. Thus, to service an I/O request, which may read or write a variable amount of data, multiple disks may need to be accessed. Fortunately, they can be accessed in parallel, i.e., concurrently.

In such an environment, I/O performance can be measured by one of two metrics: latency and throughput. The latency performance of an application I/O stream is measured as the time (usually measured in milliseconds) it takes to service its requests after they are issued by the application. The throughput performance of a stream is measured as

the amount of its data the storage system services in a fixed unit of time. Throughput is usually measured in megabytes of data per second, i.e., MB/s. As mentioned earlier, because of the diverse performance requirements of I/O streams that concurrently access a storage system and the ubiquity of RAID storage systems [12], it is desirable that the I/O scheduler simultaneously meet the performance requirements of the I/O streams in a workload that concurrently access a RAID storage system.

1.2 Related Work

In the literature, providing throughput guarantees is a well-studied topic [45, 29, 46], while providing latency-guarantees requires further study. There are three categories of latency QoS (quality-of-service) schedulers. The first category achieves latency requirements by controlling the number of pending requests in the storage system [28, 24, 44, 40]. The second schedules requests in the earliest-deadline-first (EDF) order [30] according to their deadlines [28, 18, 44, 21, 45]. Finally, the third controls application latency performance by adjusting service allocations [40, 29, 41].

Schedulers in the first category exploit the trade-off between the latency and throughput performance of storage systems. These schedulers continuously monitor the performance of latency-bound streams and control the number of latency-bound requests in the storage system through a feedback mechanism. When the latency requirements of a workload are being met, these schedulers increase the number of pending latency-bound requests in the storage system for throughput performance. A large number of pending latency-bound requests in a storage system enhances throughput performance at the cost of latency performance, due to longer queuing delays. In contrast, when the latency requirements of the workload are not being met, they reduce the number of pending requests to optimize for latency performance. Except for Cake [40], these schedulers do not provide throughput guarantees. Also, although streams in a workload may have different latency requirements, Facade [28], Triage [24], and SARC+AVATAR [44] do not differentiate the latency require-

ments and strive to achieve a system-wide latency performance in order to meet all latency requirements in a workload. As a result, these schedulers may overly meet the latency requirements of some streams. Cake [40] provides performance guarantee to only one stream in a workload, be it latency-bound or throughput-bound.

Schedulers in the second category provide latency guarantees through EDF scheduling. These schedulers dynamically determine the deadlines of pending requests and schedule them in EDF order. A common disadvantage of these schedulers is that they do not provide latency guarantees at percentiles, which is necessary in practical environments.

Schedulers in the third category meet latency requirements by adjusting the storage service allocated to the I/O streams in a given workload. These schedulers continuously monitor the performance of each stream in the workload. If some streams are exceeding their performance requirements, while some are not meeting their performance requirements, these schedulers reduce the service allocated to the former for the benefit of the latter. When not all performance requirements in a workload are being met, some of these schedulers give priority to the servicing of some streams in a workload over the others. As explained below, in some scenarios it is difficult for these schedulers to meet latency requirements at high percentiles.

As discussed in Chapter 2, among the existing schedulers that provide latency guarantees, Cake [40], Courier [45], Fahrrad [32], Frosting [41], Maestro [29], and Stonehenge [21] provide throughput guarantees as well. Except for Courier, the rest of these schedulers are reactive in nature. They dynamically adjust their scheduling parameters based on stream performance. The reactive nature of these schedulers makes it difficult for them to meet latency requirements at high percentiles, especially for streams that issue request bursts. When the latency requirement of a stream is not being met, a reactive scheduler adjusts its scheduling parameters, which may include increasing the service allocated to the stream, in an attempt to meet its latency requirement. However, meeting a latency requirement at a high percentile means that very few requests can miss the stream's latency target. And, due to the reactive nature of these schedulers, by the time the scheduler notices that

the latency requirement of a stream is not being met, the number of requests that have already missed the stream's latency target may have exceeded the number allowed by the latency requirement (e.g., 99% of the requests), i.e., percentile rank of the stream's latency requirement (e.g., 99th percentile).

While we were developing our proactive scheduler, named ^{2}TL, to meet latency requirements at high percentiles, a similar scheduler, Courier [45], was published. Like ^{2}TL, Courier dynamically adjusts its scheduling parameters to avoid latency requirement violations. However, Courier has two significant disadvantages when compared with ^{2}TL. First, it does not provide percentile latency guarantees; instead it provides average latency guarantees. Therefore, Courier does not meet the needs of applications that have latency requirements at high percentiles. And, as mentioned earlier, recent research suggests that providing latency guarantees at high percentiles (i.e., at least at the 95th percentile) is crucial to many user-interacting applications [40, 41, 20]. Second, Courier does not take into consideration disk queuing. And, as indicated in the literature [43], disk queuing is ubiquitous and it is necessary to consider disk queuing because it enhances disk performance.

1.3 Research Problem and Hypothesis

The goal of this dissertation is to solve the following research problem.

> *Given a set of latency-bound and throughput-bound streams that concurrently access a RAID storage system with disk queues, how can latency guarantees at high percentiles be provided, while also providing throughput guarantees with best effort?*

In order to answer this research question, we need to answer the following research questions.

1. *Can latency guarantees at high percentiles be provided on a storage system with disk*

queues through proactive scheduling?

2. *What is the dynamic information required for a proactive scheduling method to meet latency requirements at high percentiles?*

3. *What is needed by a proactive scheduling method to meet throughput requirements with best effort, while not overly meeting latency requirements?*

To answer these research questions, we developed a hypothesis: *Given a set of latency-bound and throughput-bound streams that concurrently access a RAID storage system with disk queues, a scheduler can provide latency guarantees at high percentiles through proactive scheduling and throughput guarantees with best effort by taking into consideration the following dynamic information of each I/O stream in a workload: request storage latencies, request arrival rates, and request scheduling rates.*

1.4 ^{2}TL: Algorithm and Evaluation

This section provides an overview of ^{2}TL (Section 1.4.1), the experimental methodology that was employed to evaluate the effectiveness of ^{2}TL (Section 1.4.2), and the experimental results that demonstrate the properties of ^{2}TL, quantify the performance of ^{2}TL, and compare its performance with the reactive schedulers mentioned previously (Section 1.4.3).

However, before proceeding in this fashion, we briefly discuss other contributions made by this research, i.e., in addition to those presented in Section 1.5. We do this with the help of Figure 1.1, which illustrates the process that was followed in the development of this dissertation, from the design, development, and implementation of the ^{2}TL and SLAC scheduling algorithms to the collection of experimental results that provide evidence of ^{2}TL contributions to the state of the art. In the figure the process is divided into two stages: the Development Stage and the Experimental Stage. To highlight the aforementioned contributions, we focus mainly on the Development Stage. As shown in the top left part of the figure, during the Development Stage, we designed the ^{2}TL and SLAC scheduling

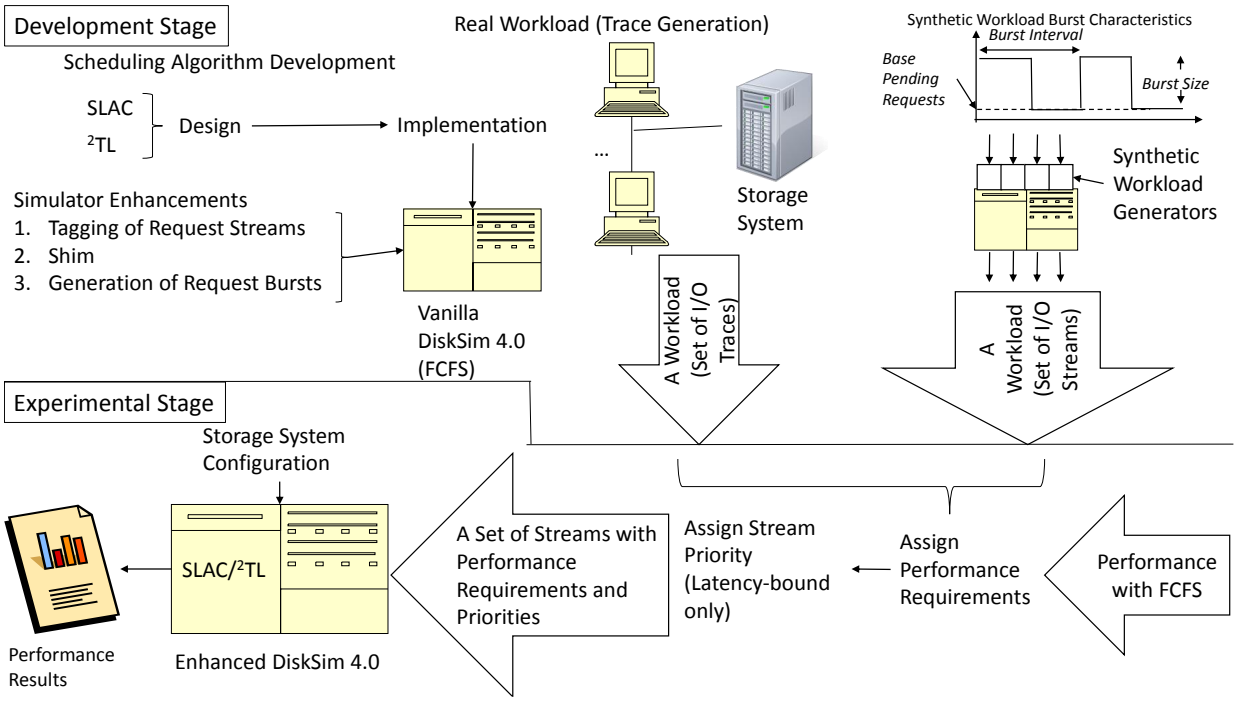


Figure 1.1: Experimental Methodology.

algorithms. Then, we implemented them in DiskSim 4.0. To use DiskSim 4.0 as our experimental platform, we had to enhance its capabilities (discussed further below and in Chapter 4). As shown in the top right part of the figure, this included modifying its synthetic workload generators to generate workloads comprised of I/O streams with request bursts, and designing and implementing a method to generate I/O streams of real applications. In addition, as depicted in the Experimental Stage, we developed a methodology to assign performance requirements to the I/O streams of a workload that demonstrate specific properties of ^{2}TL or that clearly exhibit ^{2}TL's superiority over reactive schedulers. Finally, as described in Section 5.1, we introduce the metric *Meet Rate*, which is used to measure the performance of latency-bound streams.

1.4.1 ^{2}TL Scheduling Algorithm

As discussed in Chapter 3, ^{2}TL provides performance guarantees subject to certain conditions, which apply universally to I/O schedulers that provide latency guarantees [28, 24, 44, 40, 30, 18, 21, 45, 29, 41] and throughput guarantees [45, 46, 29]. ^{2}TL provides performance guarantees to the latency-bound streams in a given workload if the following conditions are simultaneously met. And, when they are, in this dissertation we say that it is possible to provide latency guarantees.

1. The request arrival rate (in IOPS) of each latency-bound stream in the workload never exceeds the rate at which the storage system services the stream's requests, i.e., the stream's request service rate. Otherwise, the stream's number of pending requests in the I/O hierarchy (i.e., the Shim and storage system combined) will keep increasing, as will the end-to-end latencies of the stream's requests.

2. The number of pending requests in the I/O hierarchy of each latency-bound stream in the workload is never so large that the end-to-end latencies of the stream's requests exceed its latency target.

^{2}TL provides performance guarantees to each throughput-bound stream, $Stream_i$, in a workload if the following conditions are simultaneously met. And, when they are met for each $Stream_i$, in the dissertation we say that it is possible to provide throughput guarantees.

1. $Stream_i$ issues requests at a throughput (MB/s) that is high enough to consume the throughput service it requires.

2. The storage system services the requests of $Stream_i$ at a throughput (MB/s) at least as large as the throughput requirement of $Stream_i$.

The request service rate (in IOPS for latency-bound streams) and the request service throughput (in MB/s for throughput-bound streams) are subject to the performance capacity of the storage system, which is defined as how fast the storage system processes requests

issued by the streams in a workload. Thus, the performance capacity of the storage system is a limiting factor of the performance, for both latency and throughput, achievable by the streams in a workload on a storage system.

^{2}TL meets a stream's latency requirements at high percentiles by controlling the number of the stream's requests that expire in the storage system before being serviced. There are two challenges in providing latency guarantees: (1) varying request latencies in the storage system, and (2) the access characteristics of bursts of requests. ^{2}TL addresses these challenges by dynamically adjusting its scheduling parameters to: (1) reactively adapt to request latency variations in the storage system, and (2) proactively allocates the storage service to a latency-bound stream when it issues a request burst. To meet throughput requirements, ^{2}TL allocates storage service to throughput-bound streams in a workload proportional to their throughput requirements. To strive to simultaneously meet both the latency and throughput requirements of a workload, ^{2}TL is comprised of two scheduling components: a proactive scheduling component and a proportional service allocation component. When no latency-bound stream in a workload has requests that need to be scheduled immediately, the proportional service allocation component is used to meet the workload's throughput requirements. Otherwise, the proactive scheduling component is used to meet the workload's latency requirements. More details about ^{2}TL are provided in Chapter 3.

1.4.2 Experimental Methodology

As shown in Figure 1.1 and discussed in Chapter 4, the performance of ^{2}TL is assessed using simulations conducted on an enhanced version of DiskSim 4.0 [8]. We enhanced DiskSim 4.0 in three ways, i.e., we added: (1) a mechanism to tag each request with the ID of its stream; (2) a new I/O component, called Shim, where ^{2}TL and SLAC are implemented; and (3) a new feature of the synthetic workload generator that generates request bursts. As pictured at the top right of the figure, the synthetic workload generators in DiskSim are parameterized to generate synthetic I/O request streams with the characteristics specified

by the parameters. Thus, the third enhancement that we made to DiskSim allows the generation of I/O streams with bursts of requests, i.e., request bursts - such streams are called bursty streams. As shown in the top right part of the figure, to generate a synthetic workload that contains a bursty stream, we input the stream's burst parameters, along with other parameters, to one of DiskSim's enhanced synthetic workload generators - the output, which is dynamic, i.e., generated during the execution of DiskSim, is the I/O request stream of the bursty stream. Another way to drive a DiskSim simulation is with a real workload. As shown in the top right part of the figure, to use a real workload for an experiment, we first, a priori generate the I/O trace of each of the workload's streams of I/O requests. This is done by executing a parallel application on multiple computers that share a storage system. We capture the I/O requests of each stream at the storage system and export them in a trace. Finally we assemble a workload to drive a simulation. The workload consists of either a set of dynamically-generated I/O streams or a set of I/O traces. Finally, we assemble a workload to drive the simulations.

As shown in the Experimental Stage of the figure, given a workload to drive a simulation/experiment, using the methodology described in Section 4.4 of the dissertation, we next assign a performance requirement, either a latency requirement or a throughput requirement, to each stream of the workload (see middle right part of the figure). The performance requirements are assigned to create a specific scenario, for example, a scenario where the performance capacity of the storage system can or cannot meet the performance requirements of all of the streams in the workload. Next, we assign a unique priority, where Priority 1 is the highest, to each latency-bound stream in the workload. Finally, our enhanced version of DiskSim 4.0, given three inputs: (1) a workload in which each stream has a performance requirement: each latency-bound stream has a latency requirement and each throughput-bound stream has a throughput requirement, (2) a storage system configuration, and (3) an I/O request scheduler (^{2}TL, SLAC, or FCFS), simulates the scheduling of the I/O requests of the workload and outputs performance data. That data is used to analyze the performance of the specified scheduler in the created scenario.

To thoroughly evaluate the efficacy of ^{2}TL, we conducted experiments /simulations driven by workloads that are comprised of only latency-bound I/O streams, only throughput-bound streams, or a mix of both. Some workloads contain streams that issue request bursts, while others do not. The performance requirements of the streams of the different workloads are varied so that the experiments present different scenarios: when the performance requirements of none of the streams of a workload can be met, when only some of the performance requirements of the streams of a workload can be met, and when the performance requirements of all of the streams of a workload can be met. In this way we evaluate the performance of ^{2}TL in a wide range of scenarios.

Ideally, we would compare ^{2}TL with an existing scheduler in the literature. However, limitations of those schedulers, discussed in Section 4.3 in detail, do not allow such comparisons. Therefore, we designed and implemented the Storage Latency Adaptive Control (SLAC) scheduler, which does not have those limitations but embodies the reactive characteristics of many of the existing schedulers that we want to compare with ^{2}TL.

1.4.3 Experimental Results

As mentioned above, the efficacy of ^{2}TL was evaluated in simulations/experiments driven by synthetic and real workloads over a wide range of experimental settings. The experiments are organized into six sets, each of which demonstrates different properties of ^{2}TL. As discussed in Chapter 5, together these experiments provide evidence of the following five properties of ^{2}TL:

P1. Given a storage system with sufficient performance capacity to meet the performance requirements of all of the streams in a given workload, ^{2}TL will simultaneously meet the performance requirements of all of the streams, be they only latency-bound streams with or without request bursts, only throughput-bound streams, or a mix of both, providing that for each latency-bound stream (1) its request arrival rates (IOPS) does not exceed its request service rates, and (2) its number of pending re-

quests not large that request latencies are longer than the latency target, and for each throughput-bound stream (1) its request arrival throughput (MB/S) is higher than its throughput targets.

P2. Given a storage system with insufficient performance capacity to meet the performance requirements of all of the streams in a given workload, if the workload contains both latency-bound and throughput-bound streams, ^{2}TL will prioritize latency requirements over throughput requirements.

P3. Given a storage system with insufficient performance capacity to meet the performance requirements of all of the streams in a given workload, if the performance capacity is insufficient to meet the performance requirements of all of the latency-bound streams, ^{2}TL endeavors to meet the latency requirements of the streams based on the priorities that were assigned to the streams.

P4. Regardless of the performance capacity of the storage system, ^{2}TL allocates service to the throughput-bound streams in a given workload proportional to their throughput requirements.

P5. ^{2}TL's proactive scheduling of requests of the latency-bound streams in a given workload provides better performance, as compared to SLAC's reactive scheduling, when there exist one or more latency-bound streams in the workload that issue request bursts. As the burstiness of the latency-bound stream(s) increases, so does the comparative performance of ^{2}TL.

1.5 Contributions

The main research contribution of this dissertation is the design, development, implementation, and evaluation of ^{2}TL, a new I/O scheduler for RAID storage systems. ^{2}TL simultaneously provides latency guarantees at high percentiles as well as throughput guarantees,

which are essential for emerging applications with latency and throughput requirements that share a consolidated storage system and work in concert. In such an environment, the I/O scheduler must simultaneously accommodate the needs of both of these types of applications, since failures in meeting their performance requirements, especially meeting their latency requirements at high percentiles, may lead to user dissatisfaction.

The effectiveness of ^{2}TL was thoroughly evaluated through simulation with a set of synthetic and real workloads and was compared with the effectiveness of the Storage Latency Adaptive Control (SLAC) scheduler, which embodies the reactive characteristics of competitive reactive schedulers. This evaluation demonstrates the five properties of ^{2}TL listed above and shows that when certain conditions are met, ^{2}TL provides: (1) latency guarantees at high percentiles through proactive scheduling on a storage system with disk queues, and (2) throughput guarantees with best effort. In particular, this evaluation shows that ^{2}TL's proactive scheduling of requests of the latency-bound streams in a given workload provides better performance, as compared to SLAC's reactive scheduling, when there exist one or more latency-bound streams in the workload that issue request bursts. And, as the burstiness of the latency-bound stream(s) increases, so does the comparative performance of ^{2}TL. These experimental results show that ^{2}TL's performance is essential for emerging applications with latency and throughput requirements that share a consolidated storage system and work in concert.

1.6 Organization

This dissertation is comprised of six chapters. Chapter 2 discusses related work, comparing it with the contribution of this dissertation. In Chapter 3 we present the details of the ^{2}TL scheduling algorithm. The experimental methodology is explained in Chapter 4, while Chapter 5 presents and analyzes our experimental results. Finally, Chapter 6 concludes this dissertation and presents future work.

Chapter 2

Related Work

In the literature, I/O schedulers that provide Quality-of-Service (QoS) guarantees to a set of applications concurrently accessing a storage system can be partitioned into three categories: (1) schedulers that provide performance insulation, (2) schedulers that provide latency guarantees, and (3) schedulers that provide throughput guarantees. The first category, described in Section 2.1, strives to provide each application with a fraction of the storage service that it would receive if it had exclusive access to the storage system. The other two categories, discussed in Sections 2.2 and 2.3, respectively, strive to meet application-specific I/O performance requirements. In this chapter, we first summarize these schedulers and then in Section 2.4 we compare them with ^{2}TL.

2.1 Performance Insulation Schedulers

When applications share a storage system, they cannot utilize storage service as efficiently as when they have exclusive access to the system. To share storage service effectively among a set of applications, a scheduler has to mitigate, as much as is possible, the degradation of storage system performance that is due to sharing. This is known as *performance insulation*. Argon [39] and Fahrrad [32] provide performance insulation.

Argon addresses the performance degradation of the storage system caused by (in-memory) I/O cache sharing and excessive seeks at the disk level. To address performance degradation caused by I/O buffer sharing, Argon (a) uses an application's access patterns to deduce its prefetch or write-back size in the (in-memory) I/O cache, and (b) reserves for each application a partition of the I/O cache to prevent cache block evictions caused by

other applications. To reduce seeks caused by sharing the storage system, Argon time-slices I/O accesses among applications by exclusively allocating storage service to one application during each time slice or quantum. This reduces interference due to sharing. However, quantum-based scheduling can inadvertently increase worst-case request latency and, thus, Argon is not suitable for storage systems that service latency-bound applications.

Fahrrad reserves a target fraction of the storage service provided by a single-disk storage system, in terms of time, to each application. Each application is associated with two pieces of information: its target fraction of storage service and its period, which defines the frequency with which it must receive its service share. Unlike Argon, it does not time-slice accesses to the storage system. The scheduler ensures that, over time, each application receives an amount of service that is commensurate with its target fraction. To minimize the degradation of storage system performance due to sharing, Fahrrad avoids excessive disk seeks by buffering and re-ordering scheduled requests to minimize disk seeks.

The advantage of performance-insulation schedulers is that they reduce the performance degradation of the storage system due to sharing. This is achieved by minimizing interference caused by I/O accesses of different applications. However, a disadvantage of these schedulers is that it is not straight-forward to use them to meet application I/O performance requirements in terms of throughput and latency. In particular, performance-insulation schedulers must: (1) profile the I/O performance of an application when it has exclusive access to the storage system, which is not always possible, and then (2) deduce the target fraction of service, in terms of disk time, for each application. In addition, meeting the throughput performance requirement of an application with Argon requires that it determines the fraction of the throughput performance that the application achieves when it has exclusive access to the storage system, i.e., within its time slices. To meet the latency performance requirement of an application with Fahrrad, it also requires that both the target fraction and period of the application be deduced, which is not trivial.

2.2 Latency QoS Schedulers

There are three categories of latency QoS schedulers. The first category achieves latency requirements by controlling the number of pending requests in the storage system [28, 24, 44, 40]. The second category schedules requests in the earliest-deadline-first (EDF) order [30] according to their deadlines [28, 18, 44, 21, 45]. The third category controls application latency performance by adjusting service allocations [40, 29, 41].

Schedulers in the first category exploit the tradeoff between latency and throughput performance of storage systems; a large number of pending requests in a storage system enhances throughput performance at the cost of latency performance, due to larger queuing delays. Facade [28] strives to simultaneously meet application latency requirements by periodically adjusting the number of pending requests in the storage system. When all latency requirements are met, Facade increases the number of pending requests to enhance storage throughput performance. Otherwise, it reduces the number of pending requests until all latency requirements are met.

Triage [24] achieves latency targets while proportionally allocating storage throughput service to applications; the proportionality is specific to individual ranges of storage throughput. Triage does not explicitly limit the number of pending requests in the storage system. Rather, it periodically controls the request scheduling rates of the application I/O streams, in terms of IOPS, into the storage system. At the beginning of each period, Triage decides the request scheduling rate to use with a second-order system model, where the inputs are the streams' request latencies and scheduling rates of previous periods. SARC-AVATAR [44] achieves latency guarantees through two-level scheduling. The service-level agreement (SLA) of each application specifies its request arrival rate and latency requirement. The upper-level scheduler, SARC, isolates application performance by preventing each application from issuing requests into the low level at a rate higher than the rate specified in its SLA. Once scheduled by SARC, requests are stored in an EDF queue. The lower-level scheduler, AVATAR, uses a queuing theory-based controller to decide the num-

ber of pending requests in the storage system that maximizes storage system throughput, while satisfying application latency requirements. Unlike Facade, AVATAR does not make the decision solely based on application latency performance but also considers the rate at which requests enter into EDF queue. If it detects that request arrives at a rate higher than the request service rate of the storage system, it increases the number of pending requests in the storage system so to increase the request service rate. Doing so allows that storage system to promptly return to the state where all latency requirements in the workload are met.

The second category of schedulers provides latency guarantees through EDF scheduling. pClock [18] is designed to provide latency guarantees to applications with defined burst characteristics. In its Service Level Agreement (SLA), each application specifies its required average request issue rate (in terms of IOPS), maximum burst size (in terms of number of I/O operations), and latency requirement. pClock's analytical model determines the throughput performance requirement (in terms of IOPS) of the storage system based on the application's SLA. The authors analytically proved that a storage system that meets the throughput performance requirement will never miss request deadlines as long as the applications do not violate their SLAs. pClock uses request arrival curves to detect SLA violations. An application's request arrival curve is constructed based on its average request arrival rate and maximum burst size. It determines the maximum number of requests an application may have issued at any given point of time based on its SLA. pClock assigns deadlines upon request arrivals, and schedules requests in EDF order. If an application abides by its SLA, the deadlines of its requests are assigned based on their actual issue times and the latency requirement of the application. Otherwise, if an application is violating its SLA, pClock assigns later deadlines (i.e., deadlines that are later than the sum of actual issue times and the latency requirement) to the application's requests. pClock has two main limitations. First, it does not consider request latency in the storage system. Rather, it assumes that, once scheduled, requests are serviced within a small amount of time. However, because of the tradeoff between the latency and throughput performance

of a storage system, achieving short storage latency may cause throughput performance to suffer [28]. Second, pClock does not support percentile latency guarantees; it strives to service all requests before their deadlines. As a result, pClock may overly meet the percentile latency requirements of applications.

Stonehenge [21] is another EDF-based scheduler. Its goal is to virtualize a shared storage system into virtual disks, where each virtual disk meets a set of requirements in terms of space, latency, and throughput. Latency guarantees are achieved by translating the latency requirement of a virtual disk into a throughput requirement. The terminal throughput requirement of a virtual disk is the larger of its throughput requirement and its throughput requirement based on its latency requirement. Deadlines of requests are assigned based on the associated disk's terminal throughput requirement so that, if requests are serviced by their deadlines, the virtual disk will meet its terminal throughput requirement. Instead of addressing percentile latency requirements, Stonehenge strives to service all requests before their deadlines. Therefore, like pClock, Stonehenge may overly meet percentile latency performance requirements.

Courier [45] provides latency and absolute throughput guarantees to each application that simultaneously shares a storage system, where each application has a latency requirement and a throughput requirement. Section 2.3.2 discusses how Courier provides absolute throughput guarantees to applications. To provide latency guarantees, Courier considers varying storage system performance and request arrival rates. Courier continuously monitors the request service rate of each application and uses this information to estimate request service time and identify requests that may miss their deadlines. To provide both latency and throughput guarantees, Courier has two scheduling modes: the latency-constrained scheduling mode (similar to ^{2}TL's proactive scheduling component) and the throughput-allocation mode (similar to ^{2}TL's proportional service allocation component). In the latency-constrained scheduling mode, expired requests and requests that may miss their deadlines are scheduled into the storage system. Afterward, Courier switches to the throughput-allocation mode, which is discussed in Section 2.3.2. By default, Courier oper-

ates in the throughput-scheduling mode, it switches to the latency-constrained scheduling mode in one of the following two cases. First, every time when the scheduler attempts to schedule a request, it checks if any application has expired requests or requests that may miss their deadlines. If so, it switches to the latency-constrained scheduling mode to schedule those requests. Second, to avoid latency requirement violations due to bursty access characteristics, when an application issues a request, based on the estimated service time, if Courier expects that the new request will miss its deadline, Courier switches to the latency-constrained scheduling mode and schedules all of the pending requests of that application. However, like Maestro, Courier is designed to provide average latency guarantees, but not percentile latency guarantees. Therefore, it does not address the needs of applications that have I/O latency requirements at percentiles. Also, like pClock, Courier does not consider request latencies in the storage system since it assumes that the storage system does not have request queueing. ^{2}TL is more functionally advanced than Courier in two ways. First, ^{2}TL provides percentile latency guarantees at high percentiles, while Courier does not. Latency guarantees at high percentiles are essential to many user-interacting applications, such as rich web applications [40, 41]. Second, while Courier assumes no disk queueing in the disk array, ^{2}TL considers disk queueing and its effect on latencies when scheduling requests. Given the enhancement to disk performance [43], it is essential for an I/O scheduler to take disk queueing into consideration.

The last category of schedulers meet latency requirements by adjusting storage service allocation to applications. In SLEDS [9] each application's SLA specifies its request arrival rate and latency requirement. SLEDS assumes that the storage system is sufficiently provisioned to simultaneously meet the latency requirements of all applications. It periodically samples application I/O performance. If an application's latency requirement cannot be met, SLEDS throttles the request scheduling of applications that have exceeded their performance requirements. SLEDS considers each application that shares a storage system to have a latency requirement but not a throughput requirement. Therefore, it is not applicable on modern, large-scale storage systems where some applications have only latency

requirements, while others have only throughput requirements [41].

Maestro [29] simultaneously provides performance guarantees, either in terms of latency or throughput, to all applications. It continuously monitors application I/O performance and periodically adjusts the service allocation to each application to meet its performance requirement. In the case when the storage system is unable to simultaneously meet the performance requirements of all the I/O streams in a workload, Maestro degrades the performance of each stream, where the degree of degradation is inversely proportional to the application's priority. Although Maestro is designed for large-scale storage systems that are shared by both applications that have latency requirements and applications that have throughput requirements, published evaluations examine its effectiveness only in terms of meeting average latency requirements, not in terms of meeting latency requirements at high percentiles.

Frosting [41] provides either a latency or a throughput guarantee to each application that shares a storage system with a deep software/hardware stack. These performance requirements are high-level requirements expressed in terms of, for example, response time for each get/put operation (latency) or the number of scan operations per second (throughput). Frosting continuously monitors application I/O performance, compares it to application performance requirements, and adjusts application request admission rates into the storage system as is necessary. When the storage system is unable to simultaneously meet all of a workload's performance requirements, like Maestro, Frosting degrades the performance of each application, where the degree of degradation is inversely proportional to the application's priority. Experiments conducted on an HBase distributed storage system show that Frosting is able to provide to each application either a latency performance guarantee at the 99^{th} percentile or a throughput performance guarantee, both expressed in the aforementioned high-level I/O operations. Frosting assumes applications that share a storage system use a common storage API; HBase is the common API in the experiments. Therefore, Frosting is not applicable on storage systems where sharing applications use different storage API, such as MapReduce or POSIX.

Cake [40] also provides performance guarantees in terms of latency and throughput on storage systems with deep software/hardware stacks. It coordinates resource scheduling on various layers of a deep storage hierarchy through two levels of scheduling. There is a first-level scheduler on each selected software layer. It (1) controls resource sharing among applications on that layer, (2) splits large requests into small chunks to allow more controllable service allocations, and (3) limits the number of pending requests at lower levels to balance between latency and throughput performance. The second-level scheduler deduces scheduling parameters for the first-level schedulers and coordinates their activities in order to achieve the specified high-level performance requirements. Coordinated scheduling on multiple levels allows Cake to effectively handle "multicast" effects, where a single request to a layer generates multiple requests to lower layers. However, among a set of applications that simultaneously access a storage system, Cake may provide performance guarantees to only one of the applications. Therefore, Cake is not suitable for storage environments where multiple applications require performance guarantees.

Except for Stonehenge [21], Maestro [29], Frosting [41], Courier [45], and Cake [40], the other afore-mentioned latency QoS schedulers are not designed to meet application throughput requirements. Although some latency schedulers, such as SARC+AVATAR [44] and pClock [18], consider the request arrival rates of applications, it is not clear how these schedulers can be applied to meet throughput targets of latency-agnostic applications, such as data-mining. Therefore, they are not applicable in storage systems that are concurrently accessed by latency-bound and throughput-bound applications. For Maestro, Frosting, and Cake, an application is the granularity of performance guarantees. That is, either a latency or a throughput guarantee is provided to each application. In contrast, for Stonehenge, a virtual disk is the granularity, where a virtual disk may be shared by a set of applications. The assignment of applications to virtual disks requires the consideration of several factors including application performance requirements, virtual disk performance specifications, and the combination of applications that share each virtual disk. If these factors are not considered, either application performance requirements cannot be met or the utilization

of the storage system will be low. Despite the complications, virtual disks are suitable for virtual machine environments where each virtual machine hosts a guest operating system that has its own I/O scheduler to allocate storage service to each applications running on the virtual machine.

2.3 Throughput QoS Schedulers

Throughput schedulers can further be categorized in subcategories of relative-throughput and absolute-throughput QoS schedulers. Each subcategory is discussed below.

2.3.1 Relative-Throughput QoS Schedulers

Given a set of applications that simultaneously access a storage system, a relative-throughput scheduler [7, 22, 42, 4, 37] proportionally distributes storage system throughput among the applications. Proportional throughput sharing is usually enforced through weighted fair queuing [15], in which each application receives throughput performance in proportion to its weight. Weights are usually assigned by system administrators.

The majority of relative-throughput QoS schedulers allocate throughput performance to applications, either in terms of I/O operations per second (IOPS) or number of bytes per second (bytes/second). However, [35] shows the difficulties of isolating application performance based on these throughput metrics in single-disk systems, let alone multiple-disk storage systems. In addition, it is challenging to regulate disk usage via throughput shares because I/O requests are not preemptable and the time required to service them is partially non-deterministic and can vary by orders of magnitude [23]. In contrast, compared to bytes/second or IOPS, the use of disk time can produce greater control, more efficient use of disk resources, and better workload insulation [23]. Both FAIRIO [4] and QBox [37] proportionally share I/O services of a RAID storage system in terms of disk time among applications. FAIRIO assumes a feedback mechanism through which per-request disk-time usage is reported to the scheduler at the I/O driver, while QBox proposes a model to esti-

mate per-application disk-time usage. Although these schedulers are able to proportionally share storage service in terms of disk time among applications at high accuracies, proportional disk-time sharing does not necessarily lead to proportional throughput sharing at the same ratio if applications have different access characteristics.

To meet application throughput requirements using relative-throughput QoS schedulers, it is necessary to assign application weights. Since the throughput of each application is partially dependent on other applications' weights, application weights need to be re-assigned when the set of applications changes. In practice, the set of applications sharing a large storage array changes considerably over time. Therefore, relative-throughput schedulers cannot be practically applied in meeting application throughput requirements [29].

2.3.2 Absolute-Throughput QoS Schedulers

Absolute-throughput QoS schedulers strive to individually meet each application's throughput requirement. Examples of these schedulers are Courier [45], U-Shape [46], and Maestro [29]. As mentioned earlier, both Frosting and Maestro simultaneously provide latency and throughput guarantees to applications. When the storage system is overloaded, they degrade application I/O performance according to application priorities. This requires comparing application I/O performance in terms of two different metrics, latency and throughput. To do so, both scheduling algorithms represent application I/O performance and deduce target performance degradation in terms of a unified performance metric.

Unlike other absolute-throughput schedulers, U-Shape requires users to specify application execution-time requirements, but not I/O throughput requirements. Using a machine-learning model, it dynamically deduces the instantaneous I/O throughput requirement of each application in order to meet its execution-time requirement. U-Shape meets the instantaneous I/O throughput requirements of applications by time-slicing I/O accesses among applications and dedicating each slice to either one application or a set of applications. The size of each slice corresponds to the instantaneous throughput requirement(s) of the application(s) to which the slice is assigned. Since time-slicing inadvertently increases

request latency, like Argon, U-Shape is unable to provide latency guarantees.

When Courier is in its throughput-allocation mode, it operates on a period basis, where the period length is an input parameter. At the beginning of each period, it assigns credits to each application. For each request that an application schedules, it consumes a credit. The amount of credits for each application is calculated based on the period length and its throughput requirement. Each application takes a turn at exclusively accessing the storage system until it runs out of credits or Courier switches to its latency-constrained scheduling mode. Since an application may consume throughput service that exceeds its throughput requirement, during its latency-constrained scheduling mode Courier keeps track of the throughput service consumed by each application and when necessary reduces the credits assigned to an application for the next period. In the long run, this prevents an application from degrading another application's throughput performance. Courier assumes that throughput requirements are specified in I/O operations per second (IOPS). However, some applications, such as scientific applications, demonstrate phases of execution [25] that differ in terms of request sizes. Because storage systems usually take more time to service large requests than small requests [35], using MB/s, rather than IOPS, as the unit of throughput requirement better reflects an application's actual storage service consumption.

2.4 How is ^{2}TL Different?

^{2}TL differentiates itself from the schedulers discussed above in the six main ways:

1. Like Cake [40], Fahrrad [32], Frosting [41], Maestro [29], Courier [45], and Stonehenge [21], ^{2}TL provides both latency and absolute throughput guarantees. In contrast, most other schedulers in the literature provide either latency guarantees, such as Façade [28], pClock [18], SARC+AVATAR [44], and SLEDS [9], or throughput guarantees, such as FAIRIO [4], SFQ(D) [22], U-Shape [46], and YFQ [7].

2. The primary objective of performance-insulation schedulers is to minimize storage

system performance degradation due to sharing. In addition, as is exemplified by Argon [39] and Fahrrad [32], performance-insulation schedulers can be used to provide performance guarantees. In contrast, ^{2}TL does not provide performance guarantees through performance insulation, but if it did, this would require on-line evaluation of an application's I/O performance when it has exclusive access to the storage system, which is not always possible. For example, deploying a new application into a storage system would first require reserving the entire storage system for the new application in order to measure its performance when it is the only application accessing the storage system.

3. ^{2}TL provides percentile latency guarantees. Depending on an application's performance requirement (e.g., the response time of a web application to browser requests), application users may assign I/O latency requirements at various percentiles and, in this case, providing average latency guarantees [28, 24, 29, 45] may not meet latency requirements at high percentiles. On the other hand, providing latency guarantees at the 100^{th} percentile [21, 18] (i.e., ensuring that all requests are serviced before their deadlines) is unnecessary and can be expensive.

4. To the best of our knowledge, ^{2}TL and Courier are the only I/O schedulers in the literature that employ proactive scheduling to meet latency requirements. Existing latency-guarantee schedulers strive to meet latency requirements by *reacting* to application latency performance. Some latency-bound applications, such as OLTP, issue requests in bursts [29], which makes it difficult to meet their latency requirements at high percentiles through scheduling [40]. To address this challenge, ^{2}TL and Courier *avoid* latency requirement violations by continuously adjusting scheduling parameters to adapt to dynamic application access characteristics and storage system status. However, ^{2}TL, unlike Courier, is designed to meet latency requirements at high percentiles and takes disk queueing into consideration. In contrast, Courier's effectiveness in meeting latency requirements is measured using the average

latencies experienced by the requests of latency-bound streams in a workload.

5. ^{2}TL provides overload priority, i.e., unlike other schedulers [18, 44, 21], it does not assume a sufficiently provisioned storage system. When the performance capacity of the storage system cannot simultaneously meet all of the performance requirements of a workload (the storage system is overloaded), ^{2}TL gives priority to the highest-priority streams(s), with latency-bound streams being prioritized over throughput-bound streams. Latency-bound streams are given priority since they interact with end users [40, 13] and the satisfaction of latency requirements is crucial for user satisfaction and, thus, enterprise revenue [20]. Although storage provisioning is a well-studied area [1, 27, 33, 2, 3, 19, 38, 16], it is difficult to avoid overloading unless application access characteristics are thoroughly understood or the storage system is substantially over-provisioned. This is because storage system performance is highly dependent on application access characteristics [40].

6. ^{2}TL considers request latency in the storage system, while some existing schedulers, such as Courier [45], and pClock [18], do not.

Table 2.1 summarizes and compares the properties of the schedulers mentioned above. As can be seen, ^{2}TL has the same properties as only two of the other schedulers, i.e., Frosting and Maestro. However, in contrast to Frosting and Maestro, ^{2}TL is designed to provide latency guarantees at high percentiles, while providing throughput guarantees to throughput-bound applications. In particular, ^{2}TL is able to do this when latency-bound applications generate bursts of I/O requests and when the latencies experienced by the requests vary over time.

First, consider ^{2}TL's ability to provide latency guarantees to applications that generate bursts of I/O requests. Any latency-bound application, such as a decision-support system (DSS), online-transaction processing (OLTP), and mail hosting, has bursty access characteristics [17]. Failing to meet its latency requirement may lead to the loss of important

information, such as transaction data. Unfortunately, meeting the performance requirements of latency- bound applications through I/O scheduling is known to be difficult [40].

Second, consider ^{2}TL's ability to provide latency guarantees at high percentiles, e.g., from the 95th percentile to the 99th percentile. It is common for web applications to demonstrate a "request fan-out" pattern [40], where a user request for the rendering of a single page might spawn multiple, parallel I/O requests into the storage system. The overall turnaround time of the user request is decided by the completion time of the I/O request serviced last. Similarly, a database query may require multiple, parallel I/O accesses to complete [11]. The turnaround time of a database query is decided by the completion time of the I/O request serviced last. Because of the request fan-out pattern, a slight degradation in the 99th percentile latency performance can dramatically increase the overall turnaround time of the original user request [13, 40] (e.g., a web-service request or a database query) and, in turn, may lead to user dissatisfaction [20, 34].

Unlike Maestro, which provides average latency guarantees, ^{2}TL is designed to meet latency requirements at high percentiles. While Maestro and Cake strive to meet latency requirements by reacting to application latency requirement, ^{2}TL proactively adjusts scheduling parameters to avoid violating the latency requirements of applications that have bursty access characteristics. Due to the growing popularity of consolidated storage systems, it is common for a shared storage system to simultaneously service requests generated by latency-bound and throughput-bound applications. Because the latency performance of latency-bound applications is usually critical [20], while many throughput-bound applications are batch applications that run in the background [40], in cases when a storage system cannot meet the performance requirements of all of the applications in a workload, it gives priority to latency-bound applications over the throughput-bound applications.

Table 2.1: Summary of I/O Schedulers

I/O Scheduler	**Throughput Guarantee**	**Latency Guarantee**	**Performance Insulation**	**Overload Priority**
[2]TL	absolute	per-app latency		Y
Argon			Y	
Cake	absolute	per-app latency		
Courier	absolute	per-app latency		
Facade		global latency		
Fahrrad	absolute	per-app latency	Y	
FAIRIO	relative			
Frosting	absolute	per-app latency		Y
Maestro	absolute	per-app latency		Y
pClock		per-app latency		
SARC+AVATAR		per-app latency		
SFQ(D)	relative			
SLEDS		per-app latency		
Stonehenge	absolute	per-app latency		
Triage	relative	per-app latency		
U-Shape	absolute		Y	
YFQ	relative			

Chapter 3

^{2}TL I/O Scheduler

This chapter presents the ^{2}TL I/O scheduler, which provides latency guarantees and throughput guarantees to I/O streams concurrently sharing a RAID storage system. As mentioned in Chapter 1, an I/O stream is the granularity at which performance guarantees are enforced by ^{2}TL. And, in this context, an I/O stream is defined as a sequence of I/O requests issued by a process, a group of related processes, a virtual machine, or virtual machines running on the same physical host. An I/O stream has either a throughput or a latency requirement and, accordingly, is defined to be either throughput-bound or latency-bound, respectively. The requests of a latency-bound stream are known as latency-bound requests. Similarly, the requests of a throughput-bound stream are known as throughput-bound requests. The methodology used by ^{2}TL to provide performance guarantees to latency- and throughput-bound I/O streams is described in the sections of this chapter as follows. Section 3.1 describes the convention used to specify the performance requirements of I/O streams. A general description of ^{2}TL is provided in Section 3.2, while the algorithms employed by ^{2}TL to provide throughput and latency guarantees are described in Sections 3.3 and 3.4, respectively. To aid in the reading of this dissertation, Table 3.2 summarizes the terminology introduced in this chapter.

3.1 I/O Performance Requirement Specifications

For schedulers that provide performance guarantees, the convention used to specify I/O stream performance requirements needs to be stated clearly since there can be subtle differences in how this is done. For example, (a) throughput requirements are assumed to

be specified in terms of I/O operations per second (IOPS) in Triage [24] and in terms of MB/s in Maestro [29], and (b) latency requirements are assumed to be specified in terms of average latency in Maestro [29] and in terms of percentile latency in Cake [40]. This section presents ^{2}TL's convention and the terminology used to specify the throughput or latency requirement of an I/O stream.

In the context of ^{2}TL, a storage system that serves a given set of I/O streams is considered to be overloaded if it does not have the performance capacity to simultaneously meet the performance requirements of all of the I/O streams; otherwise, the system is defined as non-overloaded. To distinguish between different I/O streams each, I/O stream is denoted by $Stream_x$, where x is an alphanumeric character that is distinct for each I/O stream. The throughput requirement of $Stream_i$, which is called the target throughput of $Stream_i$, is denoted by $TputTarget_i$, and is specified in MB/s. Considering an I/O workload comprised of just throughput-bound I/O streams, ^{2}TL provides guaranteed throughput to an I/O stream only during certain time intervals described in Section 3.3.3 and for such a time interval, the average throughput is guaranteed to be at least as large as the I/O stream's target throughput if the storage system is non-overloaded. In an overloaded storage system the throughput achieved by an I/O stream is proportional to its target throughput.

Next consider a workload comprised of latency-bound I/O streams. The latency requirement of $Stream_j$ is described by the tuple: $< LatencyTarget_j, PercentileRank_j >$, where $LatencyTarget_j$ is its end-to-end latency target in milliseconds (ms) and $PercentileRank_j$ is the percentage of requests of the stream that is required to have an end-to-end latency less than or equal to $LatencyTarget_j$, i.e., $PercentileRank_j$ is the percentile rank of the I/O stream's latency target. For example, an I/O stream with a latency requirement of $< 10ms, 90\% >$ requires at least 90% of its requests to be serviced with end-to-end latencies less than or equal to 10ms. A request's end-to-end latency is measured as the time when the request arrives at the I/O driver to the time when it is serviced by the storage system and, therefore, is the sum of the latencies at the I/O driver and in the storage system. In a non-overloaded storage system with ^{2}TL each latency-bound I/O stream receives service

such that a percentage of its requests equal to its specified percentile is serviced in time less than or equal to its target latency. In contrast, in an overloaded storage system, there are no guarantees of any kind.

Now consider an I/O workload comprised of both throughput-bound and latency-bound I/O streams. When the storage system is non-overloaded every throughput-bound I/O stream is guaranteed its target throughput and each latency-bound I/O stream is guaranteed to meet its specified latency requirement. In an overloaded storage system, latency-bound I/O streams are prioritized over throughput-bound I/O streams and, therefore, ^{2}TL attempts to satisfy the performance requirements of latency-bound I/O streams first and the residual service is distributed among the throughput-bound I/O streams in proportion to their target throughput. However, if the storage system is overloaded to the extent that even the latency requirements of the latency-bound I/O streams cannot be met then no storage service will be allocated to throughput-bound I/O streams. This is justified by the fact that many latency-bound I/O streams are associated with interactive applications where the satisfaction of latency requirements is crucial for enterprise revenue [20] and in contrast, many applications that comprise throughput-bound I/O streams are background tasks that are less performance-critical [40]. Table 3.1 provides a summary of the performance guarantee provided by ^{2}TL.

3.2 ^{2}TL I/O Scheduling Algorithm: Overview and Terminology

Before we examine how ^{2}TL schedules requests, we need to understand the term scheduling opportunity, which is defined as the occurrence of an event from a predefined set of events that triggers the scheduling of one or more requests from the I/O driver into the storage system. While ^{2}TL defines the order in which requests are scheduled into the storage system, the events that comprise a scheduling opportunity are defined by the I/O component

where ^{2}TL is implemented. This dissertation assumes that ^{2}TL is implemented in an I/O component, named *Shim*. The description of Shim can be found in Section 4.1.2. For example, in Façade [28], pClock[18], and SARC+AVATAR [44] the occurrence of one of the following two types of events is considered a scheduling opportunity: the arrival of a request at the I/O driver and the completion of the servicing of a request by the storage system. ^{2}TL scheduling opportunities are defined in the next paragraph. When there is a scheduling opportunity, ^{2}TL schedules a request into the storage system. Also note that while in some operating systems, I/O streams share a unified queue, in the implementation of ^{2}TL, each I/O stream has its own FCFS-scheduled queue.

Assuming that the storage system is non-overloaded, ^{2}TL simultaneously provides latency guarantees and throughput guarantees to latency-bound and throughput-bound I/O streams, respectively. Figure 3.1 shows a schematic diagram of ^{2}TL with two I/O streams, $Stream_0$ and $Stream_1$, each with its own FCFS-scheduled queue. When a request arrives at the I/O driver, it is placed in the FCFS-scheduled queue of its I/O stream. A request from one of these FCFS-scheduled queues is scheduled by ^{2}TL into the storage system at every scheduling opportunity, which is defined as the fulfillment of the following two conditions:

(a) A request receives the requested service at the storage system or a request arrives at the I/O driver, and

(b) the number of requests in the storage system is less than its queue capacity, denoted by $QLength$.

To control request scheduling and provide performance guarantees ^{2}TL uses three types of data described below:

1. I/O stream performance requirements, which are the throughput requirement or the latency requirement of each I/O stream: These are depicted as solid lines in Figure 3.1.

2. Monitored data comprised of: (a) the storage latency experienced by each request of each latency-bound I/O stream and the stream's access characteristics, i.e., its

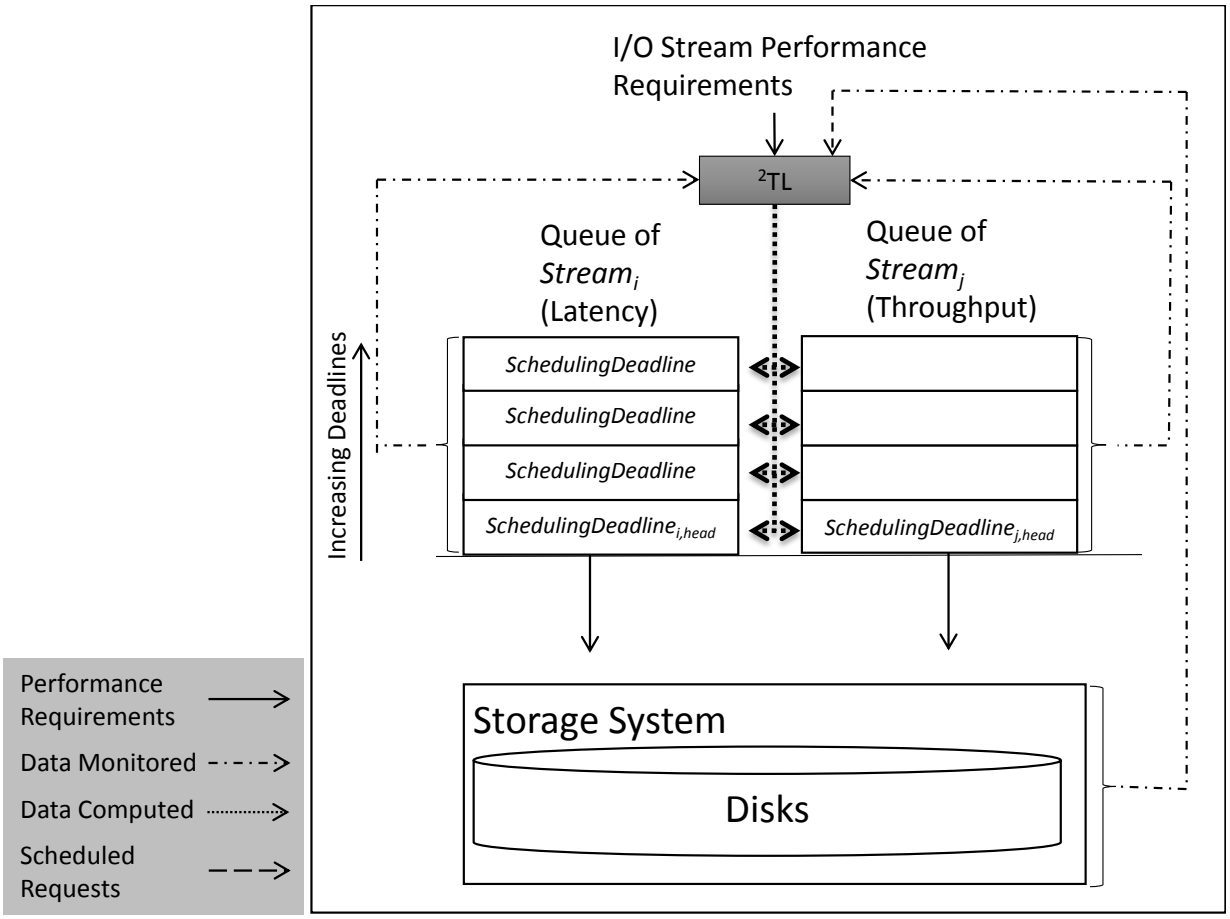
average request size and its average arrival rate; and (b) the occurrence of events defined as scheduling opportunities, which as mentioned before are request arrival and request completion: These are depicted as dash-dotted lines in Figure 3.1.

3. Derived data computed by ^{2}TL: These data are comprised of: (a) an estimated value of the storage latency experienced by the requests of each latency-bound I/O stream, which is updated every time a request from the I/O stream is serviced; (b) for each a throughput-bound stream, the scheduling deadline of each request, which is computed when the request arrives at the head of the stream's FCFS-scheduled queue; and (c) for each latency-bound stream, the scheduling deadline of each request, which is updated every time a request of the stream is serviced at the storage system. These derived data are depicted as dotted lines in Figure 3.1.

The crux of the ^{2}TL scheduling algorithm is in how it computes and uses the derived data and these details are presented in Sections 3.3 and 3.4. The algorithm, itself, is not presented until Section 3.4.5 because it would be difficult to understand without the context provided by the preceding sections. Again, Table 3.2 summarizes the terminology introduced in this chapter.

3.3 Throughput Guarantees

In this section we consider I/O workloads comprised of only throughput-bound I/O streams with throughput targets, in MB/s, that are provided to ^{2}TL as input parameters. Requests of I/O streams arrive at the I/O driver where they are placed in their respective queues before being scheduled into the storage system by ^{2}TL. To meet throughput requirements, ^{2}TL allocates storage service to the streams proportional to their throughput requirements. Proportional service allocation is achieved by dynamically deducing scheduling deadlines and scheduling requests in the Earliest-deadline-first (EDF) order. Therefore, in this section, we call the queue of each throughput-bound stream an EDF queue. In Section 3.3.2,



SchedulingDeadline$_{i,head}$: The scheduling deadline of the first pending request of Stream$_i$

Figure 3.1: ^{2}TL Scheduling Algorithm

we present three necessary conditions for the I/O stream throughput guarantees. The three conditions specify requirements on the request arrival rate, the rate at which requests are scheduled into the storage system, and the throughput realized, respectively. ^{2}TL controls the scheduling of requests into the storage system; that mechanism is described in Section 3.3.3. Finally, a discussion of the factors influencing I/O throughput guarantees is presented in Section 3.3.4.

3.3.1 Terminology

Several terms need to be defined to enable the description, analysis, and theoretical or empirical demonstration of the effectiveness of ^{2}TL and that is what we do next. Most ^{2}TL I/O stream-specific metrics are assumed to be measured during one-second intervals. For an I/O stream, $Stream_i$, the metrics $ArrivalRate(t)_i$, $ScheduledTput(t)_i$, and $RealizedTput(t)_i$ are the volume of requests that arrived at the EDF queue, the volume of requests scheduled into the storage system, and the volume of requests serviced, respectively, during the t^{th} second. All three of them have the unit MB/s. As shown below, these metrics are used to compute measurements over specified time intervals and in this dissertation we use the notation $< t0, t0 + T >$ to specify a time interval between $t0$ and $(t0 + T)$ seconds. For example, consider the metric $ArrivalRate(t)_i$ that is measured every second. We use it below to derive the aggregate arrival volume (of $Stream_i$'s requests) and the average arrival rate (of $Stream_i$'s requests) for a given interval $< t0, t0 + T >$ denoted $AggArrivalVol(t0, t0 + T)_i$ and $AvgArrivalRate(t0, t0 + T)_i$, respectively.

$$AggArrivalVol(t0, t0 + T)_i = \sum_{(t=1)}^{(t=T)} {}_iArrivalRate(t0 + t)$$

$$AvgArrivalRate(t0, t0 + T)_i = \frac{AggArrivalVol(t0, t0 + T)_i}{T}$$

The metrics $AggScheduledVol(t0, t0+T)_i$, $AvgScheduledTput(t0, t0+T)_i$, $AggRealizedVol(t0, t0+$

$T)_i$, and $AvgRealizedTput(t0, t0{+}T)_i$, which correspond to $ScheduledTput(t)_i$ and $RealizedTput(t)_i$ are derived similar to the way that the $ArrivalRate$ metrics are derived above. In addition, for $ArrivalRate(t)_i$ and $ScheduledTput(t)_i$ we define metrics for their minimum sustained average value over a given interval $< t0, t0 + T >$ as follows,

$$MinSusAvgArrivalRate(t0, t0 + T)_i = \min_{1 \leq T1 \leq T} ({}_iAvgArrivalRate(t0, t0 + T1)) \tag{3.1}$$

$$MinSusAvgScheduledTput(t0, t0 + T)_i = \min_{1 \leq T1 \leq T} (AvgScheduledTput(t0, t0 + T1)_i) \tag{3.2}$$

Note that the unit of all aggregate metrics is MB and for all other metrics it is MB/s.

Several metrics quantifying time are crucial for the analysis and theoretical proofs related to ^{2}TL scheduling and are introduced next. The wall clock time is referred to as true wall clock time and is denoted by $true_{wall-clock}$. The time at which ^{2}TL was invoked is a reference for all times related to ^{2}TL and we call it the reference time denoted $t_{reference}$. The ^{2}TL wall clock time denoted by $^2tl_{wall-clock}$ is time relative to the reference time and is given by $^2tl_{wall-clock} = true_{wall-clock} - t_{reference}$. At any given moment, the scheduler time, $t_{scheduler}$, refers to the ^{2}TL-assigned scheduling deadline of the latest scheduled request and all deadlines are specified with respect to $t_{reference}$. For example, in Figure 3.2 the scheduling deadline of the request at the head of the EDF queue of $Stream_0$ is 10ms; this means that its scheduling deadline is 10ms after ^{2}TL is invoked. In the situation depicted in Figure 3.2, $Stream_0$'s head request will be selected to be scheduled next since the head request of $Stream_1$ has a scheduling deadline of 11ms; accordingly, after that point in time until the next request is scheduled, $t_{scheduler}$ = 10ms, which is the scheduling deadline of $Stream_0$'s head request. $t_{scheduler}$ might not be the same as the wall-clock time, $t_{wall-clock}$. In an overloaded storage system, the scheduler is unable to always schedule requests prior to their respective scheduling deadlines and, thus, $t_{scheduler}$ is less than $^2tl_{wall-clock}$. On the other hand, in a non-overloaded storage system the scheduler can be expected to schedule requests before or at their respective scheduling deadlines and, thus, $t_{scheduler}$ is greater

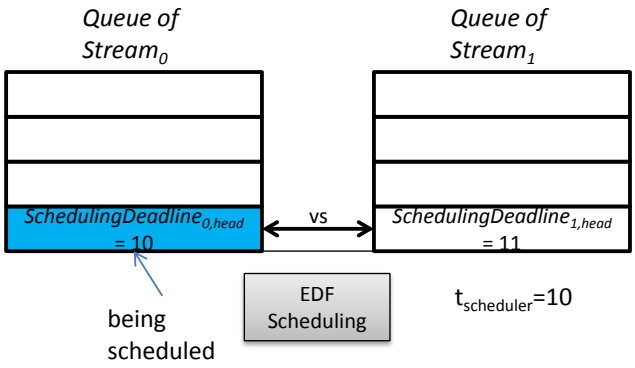


Figure 3.2: Scheduler Time.

than or equal to ${}^2tl_{wall-clock}$.

The remainder of this chapter describes how ^{2}TL computes and uses scheduling deadlines to schedule requests into the storage system with the goal of satisfying the performance requirements of all I/O streams.

3.3.2 Conditions for I/O Stream Throughput

The throughput achieved by a throughput-bound I/O stream depends on three conditions labeled C1, C2, and C3 described in this section.

We first present an intuitive explanation of condition $C1$ and the definition of an *I/O duration*. To achieve the target throughput of an I/O stream during any time interval $< t0, t0 + T >$, it is obvious that the stream needs to have a request arrival rate at least as large as the throughput target. However, it is not necessary for the arrival rate to be as large as the throughput target during every second of this time interval. An arrival rate during a one-second sub interval in this time interval can be smaller than the throughput target as long as other arrival rates at earlier one-second sub intervals in $< t0, t0+T >$ make up for the deficiency. For example, consider an illustrative 40-second duration $< 10, 50 >$. Let the arrival rate of $Stream_i$ in MB/s at time t be denoted by $ArrivalRate(t)_i$. If the arrival rate during every second in $< 10, 23 >$ is larger than or equal to $TputTarget_i$ and the arrival rate at the 24^{th} second, ${}_iArrivalRate(24)$, is less than $TputTarget_i$, then

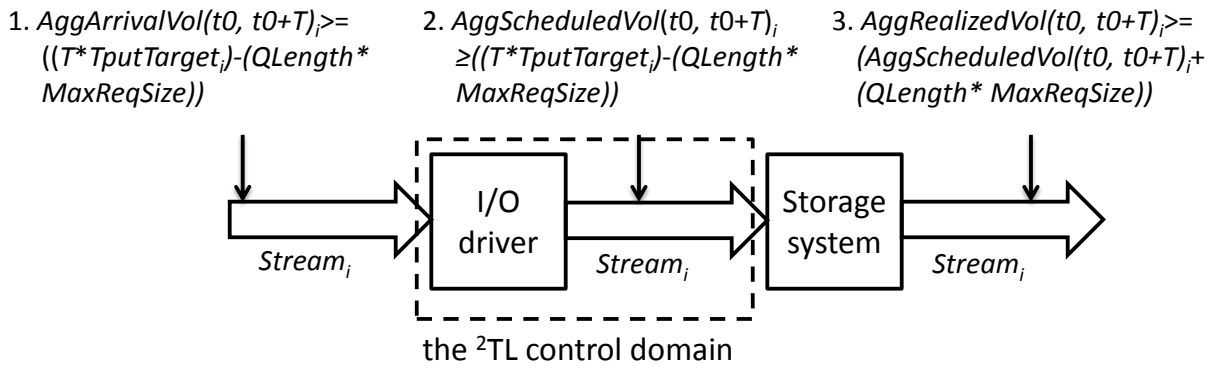


Figure 3.3: Conditions for ^{2}TL's Throughput Guarantees during a Time Interval $< t0,\ t0+T >$

if $AggArrivalVol(10, 24)_i \geq ((24 - 10) * TputTarget_i)$ then there is a sufficient volume of requests in the EDF queue for ^{2}TL to schedule and possibly meet the application's throughput target. Such a time interval is called an I/O duration. However, in the above example we did not take into consideration the requests that could potentially be in the storage system at the beginning of the time interval $t = 10$ seconds (secs) and that is captured in the following definition of an I/O duration. A time interval $< t0, t0 + T1 >$ is called an I/O duration of $Stream_i$ if

$$AggArrivalVol(t0, t0 + T1)_i \geq ((T1 * TputTarget_i) - (QLength * MaxReqSize)),$$

where $QLength$ and $MaxReqSize$ are respectively the queue length or queue capacity of the storage system and the maximum request size. Note that the definition of an I/O duration uses the I/O stream's throughput target and, therefore, the concept of an I/O duration holds only for throughput-bound streams and not for latency-bound streams.

Finally, the three conditions for an I/O stream's throughput guarantee during a time interval $< t0, t0+T >$ are expressed in terms of values of the metrics $AggArrivalVol(t0, t0+T)_i$, $AggScheduledVol(t0, t0 + T)_i$, $AggRealizedVol(t0, t0 + T)_i$, and $TputTarget_i$. The conditions are depicted in Figure 3.3.

$C1$: $< t0, t0 + T >$ is an I/O duration of $Stream_i$, i.e., $AggArrivalVol(t0, t0 + T)_i \geq ((T * TputTarget_i) - (QLength * MaxReqSize))$.

$C2$: During $< t0, t0+T >$, the I/O driver must schedule $Stream_i$'s requests at a throughput that is high enough to consume sufficient storage throughput capacity. Otherwise, the I/O driver will become a performance bottleneck. More specifically, the aggregate scheduled volume during the I/O duration along with the volume of requests that could potentially already be in the storage system must be at least as large as the product of T, the length of the time interval, and the throughput target, i.e.,

$$AggScheduledVol(t0, t0+T)_i \geq ((T * TputTarget_i) - (QLength * MaxReqSize))$$

$C3$: During $< t0, t0+T >$, the storage system must service $Stream_i$'s requests at a throughput at least as large as the sum of $Stream_i$'s average scheduled throughput and $(QLength * MaxReqSize)/T.$, i.e.,

$$AggRealizedVol(t0, t0+T)_i \geq AggScheduledVol(t0, t0+T)_i + (QLength * MaxReqSize).$$

Although, $C1$ and $C2$ are each a necessary condition when $C1$ and $C2$ are combined with $C3$, the three together, become a sufficient condition for an I/O stream's throughput guarantee. Whether or not a given time interval is an I/O duration and therefore condition $C1$ is satisfied is determined solely by the request arrival rate. However, given that a time interval $< t0, t0+T >$ is an I/O duration and that the storage system is sufficiently provisioned, ^{2}TL is designed to ensure that condition $C2$ is satisfied via computing and scheduling according to appropriate scheduling deadlines. This is described in Section 3.3.3.

Table 3.2: ^{2}TL Terminology

I/O streams	
$TputTarget_i$	Throughput target (in MB/s) of $Stream_i$
$ArrivalRate(t)_i$	The volume (in MB) of requests arrived at the I/O driver during the t^{th} second
Continued on next page	

Table 3.2 – continued from previous page

$AggArrivalVol(t0, t0+T)_i$	Aggregate size (in MB) of all requests of $Stream_i$ that arrived at the I/O driver during the interval $< t0, t0+T >$
$AvgArrivalRate(t0,\ t0+T)_i$	Request arrival rate (in MB/s) during the time interval $< t0,\ t0+T >$ averaged over T, the length of the interval
$MinSusAvgArrivalRate(t0,\ t0+T)_i$	Minimum sustained arrival rate (in MB/s) of $Stream_i$ during the time interval $< t0, t0+T >$ defined by Equation (3.1)
$ScheduledTput(t)_i$	The volume (in MB) of requests scheduled into the storage system during the t^{th} second
$AggScheduledVol(t0,\ t0+T)_i$	Aggregate size (in MB) of all requests of $Stream_i$ scheduled into the storage system during the interval $< t0, t0+T >$
$AvgScheduledTput(t0,\ t0+T)_i$	Request scheduled throughput (in MB/s) during the time interval $< t0,\ t0+T >$ averaged over T, the length of the interval
$MinSusAvgScheduledTput(t0,\ t0+T)_i$	Minimum sustained scheduled throughput (in MB/s) of $Stream_i$ during the time interval $< t0, t0+T >$ defined by Equation (3.2)
$RealizedTput(t)_i$	The volume (in MB) of requests serviced by the storage system during the t^{th} second
$AggRealizedVol(t0, t0+T)_i$	Aggregate size (in MB) of all requests of $Stream_i$ serviced by the storage system during the interval $< t0,\ t0+T >$

Continued on next page

Table 3.2 – continued from previous page

$AvgRealizedTput(t0, t0+T)_i$	Request service throughput (in MB/s) during the time interval $< t0,\ t0+T >$ averaged over T, the length of the interval
$LatencyTarget_i$	Latency target (in ms) of $Stream_i$. It is measured as the time when a request arrives at the I/O driver to the time when it is serviced by the storage system
$PercentileRank_i$	Percentile rank of the latency target of $Stream_i$
$StorLatThreshold_i$	Storage latency threshold of $Stream_i$
${}_iI/O\_duration_k$	Length of an I/O duration of $Stream_i$
Requests	
$Size_{i,\alpha}$	The size of $Request_{i,\alpha}$
$Position(Request_{i,\alpha})$	The position of $Request_{i,\alpha}$ in its EDF queue
$SchedulingDeadline_{i,\alpha}$	^{2}TL scheduling deadline of $Request_{i,\alpha}$ when it is at the head of a EDF queue
^{2}TL Scheduler	
$^2tl_{wall-clock}$	Elapsed time since ^{2}TL is invoked
$t_{scheduler}$	^{2}TL-assigned scheduling deadline of the latest scheduled request
$true_{wall-clock}$	True wall-clock time
$t_{reference}$	The time at which ^{2}TL was invoked. It is a reference for all times related to ^{2}TL.
Storage System	
$QLength$	Queue length or queue capacity of the storage system

Continued on next page

Table 3.2 – continued from previous page

$MaxReqSize$	Maximum request size of the storage system

3.3.3 Providing Sufficient Scheduled Throughput

^{2}TL controls scheduled throughput via computing and scheduling requests based on appropriate scheduling deadlines. There are two types of events that trigger the assignment of deadlines by ^{2}TL: (1) the arrival of a request into an empty queue and (2) the scheduling of a request from a queue, if the scheduled request was not the only request in the queue. Both these events place a new request at the head of the queue and ^{2}TL computes the scheduling deadline of that request. As shown below, ^{2}TL computes the scheduling deadline based on the I/O Stream's throughput target and the size of the request. If at time t, a request, $Request_{i,\alpha}$, of a throughput-bound stream, $Stream_i$, becomes the head request. ^{2}TL computes its deadline, based on Equation (3.3).

$$SchedulingDeadline_{i,\alpha} = t + \frac{Size_{i,\alpha}}{TputTarget_i} \tag{3.3}$$

where $Size_{i,\alpha}$ is the size of $Request_{i,\alpha}$. Observe that the scheduling deadline of a request is computed in a memory-less and load-unaware fashion. It is memory less in the sense that in computing the scheduling deadline of a request ^{2}TL does not use any information acquired by direct monitoring or deduced from historic performance data of the requests of the I/O stream. It is load-unaware in the sense that in computing the deadline of a request ^{2}TL does not consider the number of requests pending in the EDF queue. In contrast, as described in Section 3.4, the process of computing ^{2}TL scheduling deadlines for requests of a latency-bound I/O stream is load-aware and not memory less. Assuming that the request is scheduled at its scheduling deadline, $SchedulingDeadline_{i,\alpha}$, Equation (3.4) gives the average scheduled throughput of $Stream_i$ during the time interval

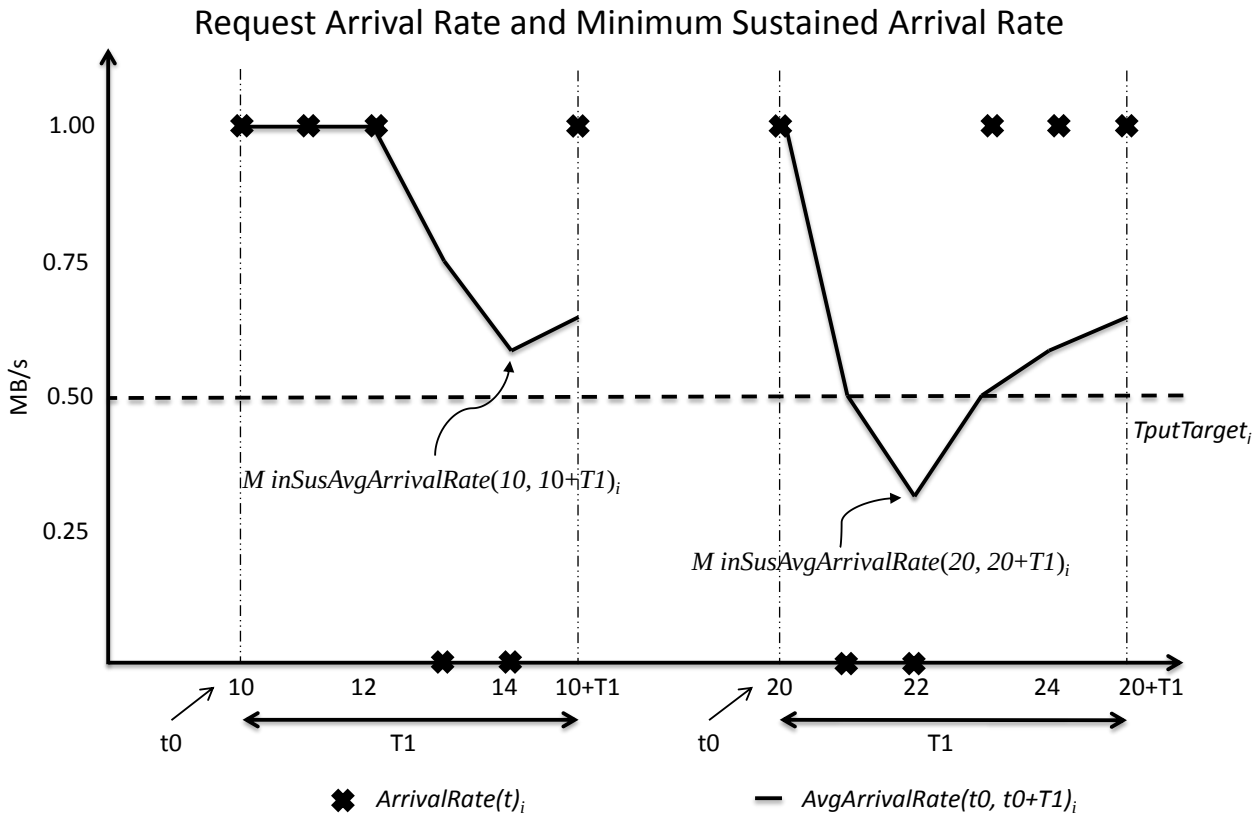


Figure 3.4: Minimum Sustained Arrival Rate.

$< t, SchedulingDeadline_{i,\alpha} >$ during which $Request_{i,\alpha}$ was at the head of the EDF queue.

$$AvgScheduledTput(t, SchedulingDeadline_{i,\alpha})_i = \frac{Size_{i,\alpha}}{SchedulingDeadline_{i,\alpha} - t} \quad (3.4)$$

From Equation (3.3), it is clear that the smaller the difference between $SchedulingDeadline_{i,\alpha}$ and t (i.e., the sooner the scheduling deadline of the request), the larger the average scheduled throughput. In Equation (3.4) substituting for $RecmSchDeadline_{i,\alpha}$ from Equation (3.3), we get $AvgScheduledTput(t, SchedulingDeadline_{i,\alpha}) = TputTarget_i$. If during $< t0, t0 + T >$ the head request is always scheduled at the scheduling deadline then $AvgScheduledTput(t0, t0+T)_i$ will be equal to the stream's throughput target, $TputTarget_i$. Thus, if all requests of $Stream_i$ are scheduled at or before (after) their scheduling deadlines, then $Stream_i$'s average scheduled throughput will be larger (smaller) than its throughput target.

3.3.4 Factors Affecting I/O Stream Throughput

The throughput achieved by an I/O stream is determined by the following three factors: rate and pattern in which the requests of the I/O stream arrive at the I/O driver, scheduling of requests into the storage system, the amount of service received at the storage system, which translates to the realized throughput of the I/O stream. Note that ^{2}TL has control over request scheduling, which is only one of the above mentioned three factors and therefore, in reality, there is always a degree of uncertainty about the final outcome, which is the throughput achieved by the I/O stream. In this subsection we discuss aspects of the three factors that influence the throughput performance of the I/O stream and enumerate factors that favor I/O stream throughput guarantee.

Arrival Rate

The arrival rate and pattern of requests of an I/O stream may vary over time. Examples of such I/O streams are those that are comprised of requests from a scientific application whose I/O activity depends on the phases of the application execution [46]. For a given

I/O stream, the length of I/O durations and the time interval between them are dependent on the nature of the applications and their dynamic execution. ^{2}TL does not make any assumptions about the length of I/O durations and the time that separates them. Note that although condition $C1$ presents a lower bound on the volume of requests in the EDF queue during the I/O duration it does not state anything about how these request arrivals need to be distributed over the time interval. Consider a scenario in which for a given I/O stream during an I/O duration $< t0, t0 + T >$ the entire volume of requests stipulated by condition $C1$ arrives all at once during the last second of $< t0, t0 + T >$. Although, in this scenario, the necessary condition $C1$ is satisfied, it is highly improbable, that the I/O stream will achieve the target throughput during this I/O duration. In this scenario, since the EDF queue of the I/O stream is empty during $< t0, t0 + T - 1 >$ ^{2}TL does not have any requests to schedule. At $t0 + T$ second the stipulated volume of requests arrive all at once and whether the requests can all be scheduled into the storage system and whether they can all be serviced by the storage system within that second depends on the storage latency and the load being offered by the other I/O streams. The uncertainty associated with such situations can be eliminated if the request arrival pattern for I/O streams is such ^{2}TL is able to schedule requests at a pace that is evenly spread out over the I/O duration $< t0, t0 + T >$. This is possible if for each I/O stream $Stream_i$, for every I/O duration $< t0, t0 + T >$, $MinSusAvgArrivalRate(t0, t0 + T)_i \geq TputTarget_i$. Let's call this Condition $C4$. Although, $C4$ is neither a necessary condition nor a sufficient condition for fulfillment of $Stream_i$'s throughput guarantee, intuitively it is more likely that the throughput guarantee of the stream is met if condition $C4$ is satisfied by all I/O durations of all streams and the storage system is non-overloaded. To understand how this works we need to explore the idea of $MinSusAvgArrivalRate(t0, t0 + T)_i$ defined as follows in Section 3.2.

$$MinSusAvgArrivalRate(t0, t0 + T)_i = \min_{1 \leq T1 \leq T} (AvgArrivalRate(t0, t0 + T1)_i)$$

Since $MinSusAvgArrivalRate(t0, t0+T)_i$ is the minimum of $AvgArrivalRate(t0, t0+T1)_i$ for all $T1$ such that $(1 \leq T1 \leq T)$, it gives us the uniform rate at which the EDF queue can be drained such that the queue will never be empty during $< t0, t0+T-1 >$. If $MinSusAvgArrivalRate(t0, t0+T)_i$ is larger than or equal to $TputTarget_i$ it means that during $< t0, t0+T >$ ^{2}TL can schedule requests of the stream into the storage system at the rate $TputTarget_i$ and be assured that the request arrival is such that the EDF queue will not be empty during $< t0, t0+T-1 >$.

Figure 3.4 presents an example scenario of two time intervals of the same length during which the same volume of requests arrive at the EDF queue but due to differences in the timing of the request arrivals the minimum sustained average arrival rates of the two intervals are different. The figure illustrates the average request arrival rate for two sets of four one-Mbyte requests of a stream arriving at the I/O driver during two time intervals of the same duration $T1$= 5 secs, $< 10, 10+T1 >$ and $< 20, 20+T1 >$. The volume of requests that arrive during each of these intervals are the same, 4 MB, and given that the throughput target of $Stream_i$ is 0.5 MB/s , $AggArrivalVol(10, 10+T1)_i = AggArrivalVol(20, 20+T1)_i$ = 4 MB, which is larger than 5*0.5 = 2.5 MB and therefore both $< 10, 10+T1 >$ and $< 20, 20+T1 >$ are I/O durations. However, the minimum sustained average arrival rates in the two intervals differ due to the way the request arrivals are distributed over time. During $< 10, 10+T1 >$ most requests arrive at the first half of the time interval and the minimum sustained arrival rate is 0.6 MB/s. On the other hand, during $< 20, 20+T1 >$ most requests arrive at the second half of the time interval and the minimum sustained arrival rate is 0.33 MB/s. Thus, $MinSusAvgArrivalRate(10, 10+T1)_i$ is larger than or equal to $TputTarget_i$ whereas $MinSusAvgArrivalRate(20, 20+T1)_i$ is not.

Scheduled Throughput

In a non-overloaded storage system, during an I/O duration of $Stream_i$, as explained in Section 3.3 ^{2}TL is designed to satisfy Condition $C2$ by dynamically deducing and using appropriate request deadlines. Similar to Condition $C4$ for arrival rates of I/O streams we

can define Condition $C5$ for scheduling throughputs as follows: If each I/O stream $Stream_i$ has $MinSusAvgScheduledTput(t0, t0 + T)_i \geq TputTarget_i$ then Condition $C5$ is said to be satisfied during the time interval $< t0, t0 + T >$. Using the same explanation that was used to describe the utility of Condition $C4$, given that Condition $C4$ is satisfied in the time interval and that the storage system is non-overloaded, it is more likely that the throughput requirement of every I/O stream will be met if Condition $C5$ is met rather than when just Condition $C2$ is satisfied for each I/O stream. This new condition implies that, to avoid the I/O driver from becoming the bottleneck of throughput guarantees, the scheduler should not delay the scheduling of requests but should strive to schedule them in a timely fashion. Note that, as is the case with Condition $C4$, $C5$ is neither a necessary condition nor a sufficient condition for throughput guarantee of streams of the I/O workload.

Achieved Throughput

For a given I/O stream, $Stream_i$, and a time interval $< t0, t0 + T >$ even when Conditions $C1$ and $C2$ are satisfied, Condition $C3$ may not be met because there is non-determinism in the working of storage system and in that sense it is a 'gray box'. Although, we have some information about how the storage system might perform we certainly do not have a complete understanding or control of the processes in the storage system. The storage system has a queue with an upper bound on the length given by $QLength$, into which the scheduled requests are placed before they are serviced. In particular when requests from different I/O streams are scheduled into the storage system during a given interval $< t0, t0+T >$, it is not possible to control which of these get serviced and which ones don't. The maximum number of requests that can remain in the storage system at any time is bounded by $QLength$ and thus the volume of requests pending in the storage system is bounded by $(QLength * MaxReqSize)$, where $MaxReqSize$ is the upper bound on the request size. This is a reasonable assumption because RAID storage systems have limits on queue capacity and request size [8]. For an I/O stream, $Stream_i$, assuming that every request of the stream is scheduled by ^{2}TL at its scheduling deadline, consider the bounds

on the volume, $AggRealizedVol(t0, t0+T)_i$, of requests serviced during an I/O duration $< t0, t0+T >$. The largest value that $AggRealizedVol(t0, t0+T)_i$ can potentially have is $(T * TputTarget_i + (QLength * MaxReqSize))$ and this happens if (a) At the beginning of the I/O duration $< t0, t0+T >$, i.e., at time $t0$, every requests in the storage system belongs to $Stream_i$ and has a size equal to $MaxReqSize$. (b) At the end of the I/O duration $< t0, t0+T >$, i.e., at time $t0+T$, there are no requests of $Stream_i$ in the storage system. Likewise, the smallest value that $AggRealizedVol(t0, t0+T)_i$ can potentially have is $(T * TputTarget_i - (QLength * MaxReqSize))$ and this happens if (a) At the beginning of the I/O duration $< t0, t0+T >$, i.e., at time $t0$, there are no requests of $Stream_i$ in the storage system. (b) At the end of the I/O duration $< t0, t0+T >$, i.e., at time $t0+T$, every requests in the storage system belongs to $Stream_i$ and is of size equal to $MaxReqSize$. Thus,

$$\begin{aligned} & (T * TputTarget_i - (QLength * MaxReqSize)) \\ & \leq AggRealizedVol(t0, t0+T)_i \\ & \leq (T * TputTarget_i + (QLength * MaxReqSize)) \end{aligned} \tag{3.5}$$

From (3.5) one can deduce that,

(a) The smaller the $QLength$ and $MaxReqSize$ the tighter are the bounds for the throughput realized during the I/O duration. This increases the likelihood of the realized throughput being equal to the target throughput. For large-scale storage systems that have large queue capacities, $QLength$, this could be a problem. For example, the queue capacity of a RAID controller can range from 1 to 256 [10]. Small $QLength$ could also lead to performance problems because in a disk that uses SSF algorithm the throughput performance can be maximized by having a queue with a large capacity. These facts need to be factored into the analysis to determine the optimum queue length.

(b) Larger the length of the I/O duration, T, the smaller will be the relative contribution of the uncertainty due to the pending requests.

Thus, having a limited queue capacity in the RAID system, having a long I/O duration, and having limited request sizes favor the I/O stream's throughput guarantee.

3.4 Latency Guarantees

This section explains how ^{2}TL meets the performance requirements of the latency-bound streams in a workload and presents the ^{2}TL scheduling algorithm. The performance requirement of each latency-bound stream, $Stream_i$, is expressed as $< LatencyTarget_i, PercentileRank_i >$, where $LatencyTarget_i$ and $PercentileRank_i$ are the latency target of $Stream_i$ and a percentile rank, respectively. To meet the latency requirement of $Stream_i$, the percentage of $Stream_i$ requests with end-to-end latencies that do not exceed the $LatencyTarget_i$ must be at least $PercentileRank_i$. As shown in Figure 3.5, the end-to-end latency of a request is the sum of its latency in the Shim, *Shim Latency*, and its latency in the storage system, *Request Storage Latency*. ^{2}TL provides a latency guarantee for an I/O stream by attempting to control the Shim Latencies of its requests so that for a majority of them (where the magnitude of "majority" is specified by the percentile rank), the sum of a request's Shim Latency and its stream's estimated Request Storage Latency is not larger than the stream's target latency. As a result, the efficacy of ^{2}TL in doing so depends on the effectiveness of controlling the Shim Latencies and the accuracy of the estimated Request Storage Latencies.

This section is comprised of five parts. Section 3.4.1 describes the two conditions that are necessary in order for ^{2}TL to provide latency guarantees. The basic algorithm employed by ^{2}TL for providing latency guarantees is presented in Section 3.4.2. Section 3.4.3 discusses the challenges of providing latency guarantees, namely Request Storage Latency variations and bursty access characteristics, while Sections 3.4.4 and 3.4.5 describe how ^{2}TL addresses these two challenges, respectively. Given the necessary information to understand the ^{2}TL scheduling algorithm, which was presented in the first four sections of this chapter, Section 4.5 present the algorithm, itself.

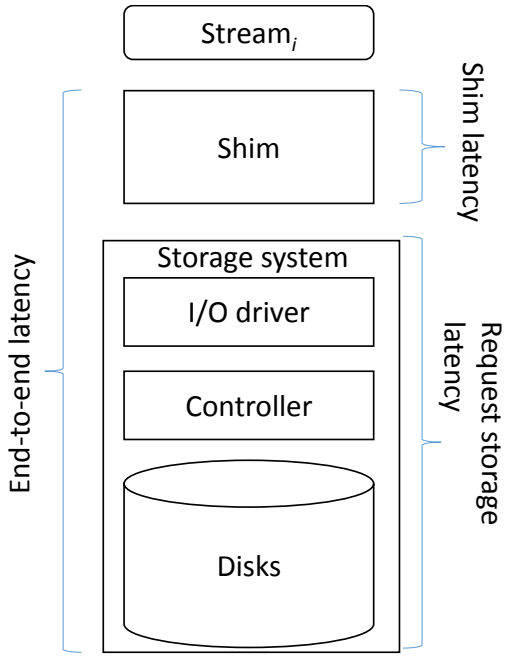


Figure 3.5: End-to-end Request Latency: Sum of Latencies in the Shim and Storage System.

3.4.1 Conditions for I/O Stream Latency Guarantees

[2]TL can provide performance guarantees to the latency-bound streams in a workload if the following conditions on the behavior of the streams are simultaneously met. And, when they are, in this dissertation, we say that it is possible to provide latency guarantees.

1. The request arrival rate (in IOPS) of each latency-bound stream does not exceed the rate at which the storage system services its requests, i.e., the request service rate. Otherwise, the number of the stream's pending requests in the I/O hierarchy (i.e., the Shim and storage system combined) will keep increasing, as will the end-to-end latencies of the requests.

2. The number of pending requests in the I/O hierarchy of each latency-bound stream is not so large that the end-to-end latencies of its requests exceed its latency target.

3.4.2 Basic Algorithm

To meet the percentile latency requirements of a workload, ^{2}TL endeavors to limit the percentage of each latency-bound stream's requests that expire in the storage system. A request of a stream expires when it is not serviced before the stream's latency target. To meet a stream's percentile latency requirement, the percentage of its requests that may expire is equal to 100% minus its percentile latency requirement. For example, if a stream's percentile latency requirement is 98%, ^{2}TL endeavors to limit the percentage of its requests that expire in the storage system to 2%.

To accomplish this, for each latency-bound I/O stream, ^{2}TL monitors and records the storage latency of a user-defined number of recently-serviced requests and constructs the cumulative distribution function (CDF) of the stream's Request Storage Latencies. Figure 3.6 shows an example of such a CDF. If the percentile rank of the stream is x, then its $RequestStorageLatencyThreshold$ is the value of the Request Storage Latency at the xth percentile in the CDF. To ensure that x% of requests are serviced before they expire, when scheduling requests, ^{2}TL strives to ensure that the slack of each request (i.e., the remaining time before it expires) is at least as long as its Request Storage Latency Threshold. ^{2}TL does this by computing the $SchedulingDeadline$ of each request of $Stream_i$, $Request_{i,\alpha}$, as shown in Equation 3.6, where $SchedulingDeadline_{i,\alpha}$ and $ServiceDeadline_{i,\alpha}$ are the scheduling and service deadlines of $Request_{i,\alpha}$, respectively, and $StorLatThreshold_i$ is the Request Storage Latency Threshold of $Stream_i$.

$$SchedulingDeadline_{i,\alpha} = ServiceDeadline_{i,\alpha} - StorLatThreshold_{i,\alpha} \tag{3.6}$$

If the scheduling deadlines of $Stream_i$'s requests are met, i.e., by scheduling them at or before their scheduling deadlines, when scheduled, the remaining time before they expire is at least as long as $Stream_i$'s Request Storage Latency Threshold. ^{2}TL's objective is to have the percentage of $Stream_i$'s requests that meet its latency target be at least $PercentileRank_i$. In this case, the latency requirement of $Stream_i$ is met. Note that, the

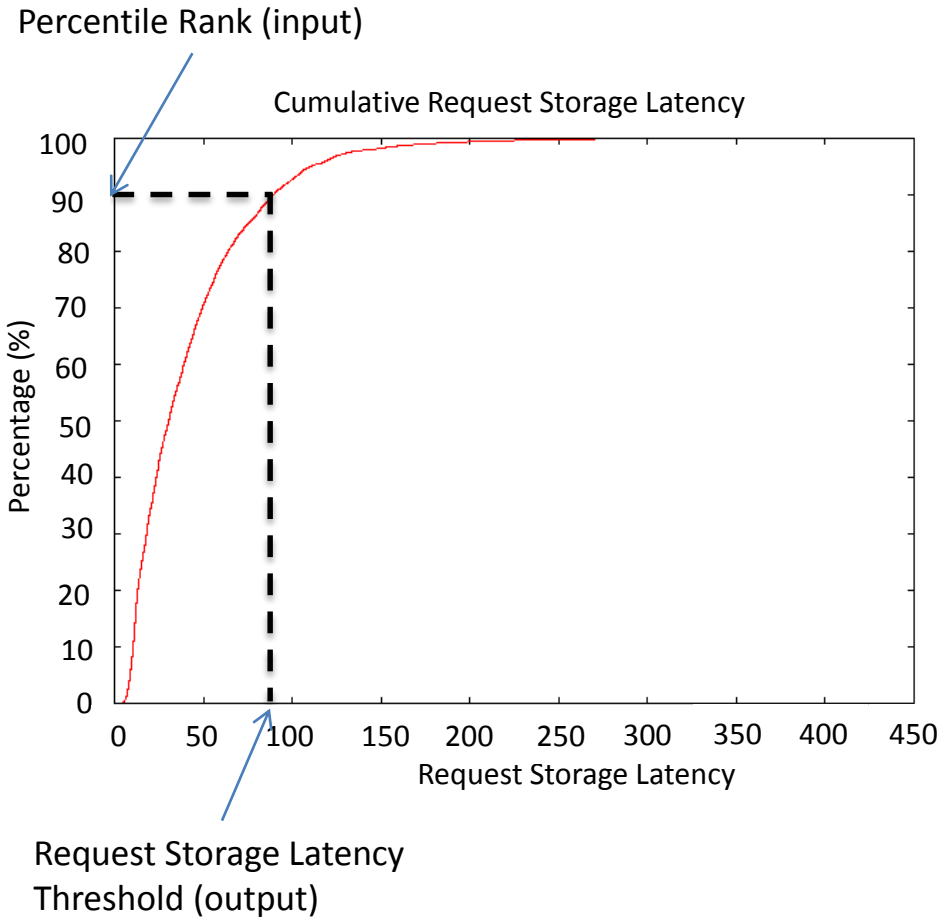


Figure 3.6: Estimating an I/O Stream's Request Storage Latency Threshold.

CDF of each latency-bound stream is updated every time a request of the stream is serviced at the storage system. When the CDF of a stream changes, its Request Storage Latency Threshold and the scheduling deadlines of its requests are updated accordingly.

3.4.3 Challenges

There are two challenges in meeting latency requirements, namely (1) Request Storage Latency fluctuation and (2) burst access characteristics. The request storage latencies of a stream vary widely over time on shared storage systems due to resource contention [14]. Variations in request storage latencies may cause ^{2}TL to be unable to meet the latency

requirement of a stream. For example, some latency-bound I/O streams have bursty access characteristics, such as OLTP [29], which make it difficult for an I/O scheduler to meet their latency requirements. When a stream issues a burst of requests, its number of requests in the Shim increases, as does their Shim Latencies. Sections 3.4.4 and 3.4.5 discuss how we address these challenges, respectively.

3.4.4 Reactive Adaptive Scheduling to Address Request Storage Latency Variations

To overcome variations in the latencies of the requests of a stream, ^{2}TL dynamically adjusts their scheduling deadlines. As the storage latencies of the requests of $Stream_i$ increase, ^{2}TL strives to schedule them earlier to overcome prolonged request storage latencies. Figure 3.7 presents an example of this situation. CDF-0 is the original cumulative distribution function (CDF) of $Stream_i$'s request storage latencies. As the storage latencies of its requests increase, $Stream_i$'s CDF-0 shifts right and becomes CDF-1. As a result, the Request Storage Latency Threshold of $Stream_i$ increases. Accordingly, as shown by Equation 3.6, ^{2}TL decreases the scheduling deadlines of the pending requests of $Stream_i$. To meet these earlier scheduling deadlines, ^{2}TL increases the scheduling rate of $Stream_i$'s requests.

3.4.5 Proactive Adaptive Scheduling to Address Request Bursts

To avoid violating latency requirements due to request bursts, when $Stream_i$ issues a request burst (the requests of which become pending requests in $Stream_i$'s queue), ^{2}TL proactively speeds up the scheduling of $Stream_i$'s requests to meet the scheduling deadlines of the requests in the burst. As shown in Figure 3.8, in order to meet the scheduling deadlines of such requests, ^{2}TL's scheduling rate must increase to a rate that is relatively high in comparison to the rate it was scheduling latency-bound requests prior to the arrival of the request burst. This is accomplished by using every scheduling opportunity to schedule a request of $Stream_i$ or any other latency-bound stream that includes pending requests of

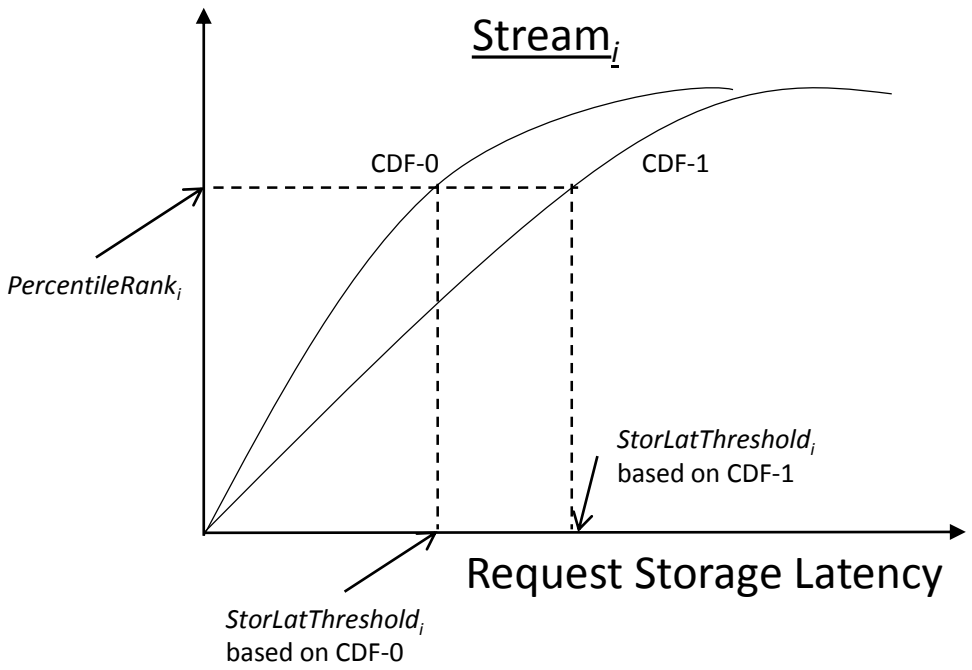


Figure 3.7: Request Storage Latency Threshold Increases with Request Storage Latencies.

bursts that may be in danger missing stream latency requirements. However, if the storage system is unable to service $Stream_i$'s requests fast enough, ^{2}TL will not be able to meet the scheduling deadlines of $Stream_i$'s requests that arrive in bursts. (Recall that the Shim limits the number of pending requests in the storage system (see Section 4.1.2) and, thus, the request scheduling rate in the Shim is limited by the request service rate in the storage system.)

As depicted in Figure 3.8, to prevent such a situation from happening, which in turn could cause possible violations of stream latency requirements, ^{2}TL attempts to push more requests into the storage system before the burst(s) must be scheduled. This is done by scheduling the leading requests in $Stream_i$'s queue, i.e., the head request and those between the head request and the requests of the burst, prior to their scheduling deadlines. This increases the time by which the requests in the burst (the trailing requests) must be scheduled and, thus, can prevent violations of stream latency requirements.

Next we discuss how ^{2}TL determines if it is necessary to schedule requests prior to their

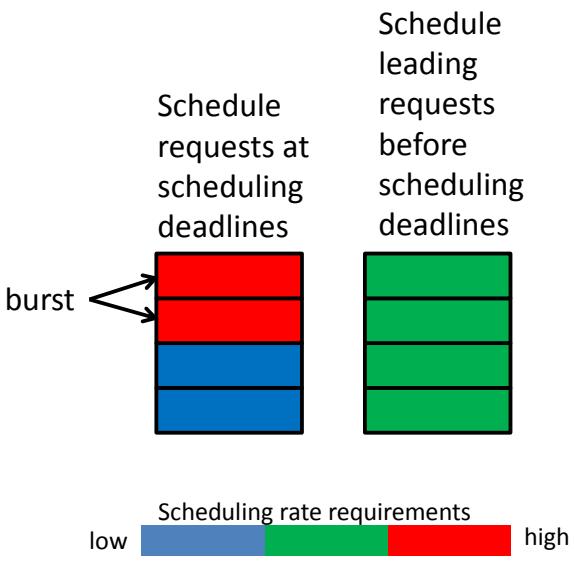


Figure 3.8: Schedule Leading Requests Early to Meet All Scheduling Deadlines.

scheduling deadlines and, if so, when. At every scheduling opportunity (i.e., when there is room in the storage system and there is at least one pending request in the Shim), ^{2}TL schedules a request into the storage system following the following general algorithm:

1. If the head request of the queue of a latency-bound stream will miss its Scheduling Deadline if not scheduled immediately, then schedule that request. If there is more than one queue in this situation, then schedule the head request of the queue of the stream with the highest priority.

2. Else, if there is a request pending (not the head request) in the queue of a latency-bound stream that will miss its Scheduling Deadline if scheduling of its stream's requests is not accelerated, then schedule the head request of that queue. If there is more than one queue in this situation, then schedule the head request of the queue of the stream with the highest priority. We refer to such streams as those that require immediate scheduling. And, below, we describe how ^{2}TL knows that a stream requires immediate scheduling.

3. Else, if there are pending requests in the queues of throughput-bound streams, sched-

ule the one with the earliest deadline.

4. Else, if there are pending requests in the queues of latency-bound streams, schedule the one with the earliest Scheduling Deadline. If there is more than one head request with the earliest Scheduling Deadline, schedule the one in the queue of the stream with the highest priority.

5. Else, no request is scheduled.

It is important to note that in step 1 of this algorithm that ^{2}TL is looking for "any" pending request that is in danger of missing its Scheduling Deadline. This is because ^{2}TL takes a proactive, yet procrastinating, approach to request scheduling in order to meet the scheduling deadlines of requests in a burst. It proactively schedules requests of a latency-bound stream prior to their scheduling deadlines in order to be able to schedule the requests of a burst by their scheduling deadlines and, thus, avoid missing the stream's latency requirement. Yet, it procrastinates scheduling the head request of a stream unless this request may miss its Scheduling Deadline. For example, when $Stream_i$ issues a request burst, assume that these pending requests are located in $Stream_i$'s queue behind a number of other previously issued requests (the leading requests in the queue, which includes the head request). In this case, if ^{2}TL does not schedule the head request of $Stream_i$ immediately, but rather at the next scheduling opportunity, the head request may still meet its Scheduling Deadline; however, the requests in the burst may not. If the latter is true, ^{2}TL proactively schedules the head request immediately.

At successive scheduling opportunities, one might assume that ^{2}TL would continue to service this stream until it scheduled the burst. However, at each scheduling opportunity, ^{2}TL must reevaluate the situation for three reasons: (1) the scheduling of some of the leading requests in the stream's queue may have alleviated "the pressure", i.e., now the requests in the burst are not in danger of expiring in the Shim; (2) now a higher-priority latency-bound stream may also require immediate scheduling and, thus, its head request must be scheduled at the next scheduling opportunity; or (3) the scheduling of some of the

leading requests in the queues of all of the latency-bound streams that required immediate scheduling may have alleviated "the pressure" on the requests in the bursts pending in the streams' queues, i.e., now the requests in the burst(s) are not in danger of expiring in the Shim.

If the first case is true, then as long as there is only one stream that requires immediate scheduling, there is no danger associated with scheduling a throughput-bound request. But, note that, right after scheduling it, the queue may require immediate scheduling again. If the second case is true, ^{2}TL may "bounce" from one stream to another, scheduling requests in a set of latency-bound streams that require immediate scheduling. And, if the third case is true, as mentioned when discussing the first case, at this point there is no danger associated with scheduling a throughput-bound request; however, right after scheduling it, the queue may require immediate scheduling again.

When ^{2}TL is able to service a throughput-bound request, i.e., if there are no latency-bound requests that must be scheduled during a scheduling opportunity, then it schedules the one with the earliest deadline. But, if there are no throughput-bound requests pending in the Shim, then ^{2}TL schedules the head request from a latency-bound stream with pending requests' if there is more than one, then it services the stream with the highest priority.

When ^{2}TL services one or more latency-bound streams that require immediate scheduling, without servicing any throughput-bound streams, the rate at which it is scheduling latency-bound requests is called the *Exclusive Scheduling Rate of Latency-bound Requests*. The faster this scheduling rate, the earlier the latency-bound requests in the streams that require immediate scheduling are scheduled.

Next, we describe how ^{2}TL determines when each latency-bound stream requires immediate scheduling. For this purpose, ^{2}TL estimates and assigns to each pending request in every latency-bound stream a *Scheduling Timestamp*. If the Scheduling Timestamp of any pending request in a stream is larger than the request's Scheduling Deadline, then the head request of the stream's queue requires immediate scheduling, i.e., at the next possible scheduling opportunity.

As we explain below, to estimate a request's *Scheduling Timestamp*, ^{2}TL must estimate the *Exclusive Scheduling Rate of Latency-bound Requests* since this scheduling rate is not a parameter within the control of ^{2}TL. As ^{2}TL increases the rate at which it is scheduling latency-bound requests, that rate may be throttled, at least temporarily, if it is larger than the rate at which the storage system can service the requests. As mentioned earlier, this is because the number of pending requests in the storage system is limited and the scheduling rate of the Shim is limited by the request service rate in the storage system. The remainder of this section discusses (1) how ^{2}TL estimates the scheduling timestamps of latency- bound streams and (2) how ^{2}TL estimates the Exclusive Scheduling Rate of Latency-bound Requests.

Estimation of Scheduling Timestamps

To determine if a latency-bound stream requires immediate scheduling, ^{2}TL estimates the scheduling timestamps of each stream's pending requests using Equation 3.7. Assume that $Stream_i$ is such a stream and the timestamp of $Request_{i,\alpha}$ is denoted by $SchedulingStamp_{i,\alpha}$. Using Equation 3.7, $SchedulingStamp_{i,\alpha}$ is estimated based on the number of pending requests in front of $Request_{i,\alpha}$ in $Stream_i$'s FIFO queue in the Shim, denoted by $Pending_{i,\alpha}$, and the estimated scheduling rate (IOPS) of $Stream_i$'requests when the scheduler next exclusively schedules $Stream_i$'s requests, denoted by $EstSchedulingRate_i$.

$$SchedulingStamp_{i,\alpha} = t_{next-opport} + \frac{{}_{shim}Pending_{i,\alpha}}{EstSchedulingRate_i} \tag{3.7}$$

Next we explain how ^{2}TL computes $t_{next-opport}$, which is the time at which the next scheduling opportunity occurs, and $EstSchedulingRate_i$. The timestamp of the next scheduling opportunity, $t_{next-opport}$, is calculated using Equation 3.8, where $t_{wall-clock}$ and ${}_{shim}SchedulingRate$ are the wall-clock time and the current total scheduling rate (IOPS) of all queues in the Shim.

$$t_{next-opport} = t_{wall-clock} + \frac{1}{{}_{shim}SchedulingRate} \tag{3.8}$$

To estimate $EstSchedulingRate_i$, we assume that the latency-bound streams equally share the request scheduling rate when ^{2}TL next exclusively schedules latency-bound requests. Although this results in a rough estimate, it is a conservative estimate since it assumes that all latency-bound streams will require immediate scheduling. A conservative estimate is desirable as an over-estimated scheduling rate may lead to latency requirement violations. The estimated scheduling rate of $Stream_i$'s requests is calculated using Equation 3.9. ${}_{latency}EstSchedulingRate$ is the estimated scheduling rate (in IOPS) when the scheduler next exclusively schedules latency-bound requests from different streams. ${}_{nr}LatencyBound$ is the number of latency-bound streams in the workload.

$$EstSchedulingRate_i = \frac{{}_{latency}EstSchedulingRate}{{}_{nr}LatencyBound} \tag{3.9}$$

Next, we describe how we estimate ${}_{latency}EstSchedulingRate$, i.e., how ^{2}TL estimates the scheduling rate when it next exclusively schedules latency-bound requests from one or more streams.

Estimation of Exclusive Scheduling Rate of Latency-Bound Requests

To determine if there are one or more latency-bound streams in a workload that require immediate scheduling, ^{2}TL must predict the scheduling rate (in IOPS) of each latency-bound stream in the workload when it next will exclusively schedule latency-bound requests. ^{2}TL uses one of two prediction strategies for this purpose, which is selected by a ^{2}TL parameter. The first assumes a simple strategy, i.e., that the scheduling rate does not change when ^{2}TL switches to exclusively schedule latency-bound streams; while the second, which assumes that it does change, employs a prediction model to estimate the exclusive scheduling rate.

Strategy 1: Using this simple strategy, ^{2}TL sets the Exclusive Scheduling Rate of Latency-bound Requests to the *current scheduling rate*, i.e., the one it has been using to schedule the requests of all the streams in the workload during a number of recent one-second intervals, where the number of intervals is a user-specified parameter. Analogously,

a stream's current scheduling rate is the scheduling rate it has been using to schedule its requests during the same number of recent one-second intervals. Since ^{2}TL does not schedule latency-bound requests until there are some that may, otherwise, miss their scheduling deadlines, latency-bound requests are often scheduled in groups of multiple requests, or in bursts. When switching to exclusively schedule a group of latency-bound requests of a stream, ^{2}TL may schedule the requests of the stream at a rate that is lower than the stream's current scheduling rate. This is because latency-bound streams usually have poor spatial locality of reference [12], which may lead to a lower request service rate. In addition, since the Shim limits the number of pending requests in the storage system, the request scheduling rate may be throttled due to it being limited by the request service rate. Accordingly, using this strategy may cause ^{2}TL to over-estimate the Exclusive Scheduling Rate of Latency-bound Requests and fail to fulfill latency requirements.

Strategy 2: This strategy, which is more complex, is based on the scheduling rate history. The prediction is made based on the observation that whenever ^{2}TL schedules bursts of latency-bound requests, i.e., groups of requests of any number, the burst scheduling rates are similar. Given this observation, we use the scheduling rates of recent bursts to estimate the scheduling rate of the next burst. There are two parameters that are needed to make such a prediction: *history size* and the *estimation model*. History size indicates the scheduling rates of a specified number of the most recent bursts that will be used to estimate the Exclusive Scheduling Rate of Latency-Bound Requests. Note that for simplicity, the strategy does not consider burst size. In terms of the estimation model, we use a rudimentary moving-percentile model that takes a user-specified percentile of the scheduling rate history to obtain the estimate. This history size and percentile of the moving percentile model are used in the evaluation of ^{2}TL; they are presented in Section 4.5. Although we could use a more sophisticated estimation model, which is left for future work, our experimental results show that using our moving-percentile model does not hinder the ability of ^{2}TL to provide latency guarantees.

Table 3.1: Summary of ^{2}TL Performance Guarantee

Constituent of I/O Workload		**State of Storage System**	
		Non-Overloaded	**Overloaded**
Only throughput-bound I/O streams		Absolute throughput guarantee	Proportional throughput guarantee
Only latency-bound I/O streams		Latency requirements satisfied	The latency requirement(s) of some I/O streams may not be satisfied
Mixed	Throughput-bound I/O streams	Absolute throughput guarantee	Latency-bound streams are prioritized and after they are serviced, throughput-bound streams receive the left-over storage service in proportion to their target throughput
	Latency-bound I/O streams	Latency requirements satisfied	Depending on the degree of overloading, some latency requirements may not be satisfied.

Chapter 4

Experimental Methodology

The performance of ^{2}TL is assessed using simulations conducted on an enhanced version of DiskSim 4.0 [8]. In this chapter we describe the enhancements that we implemented in Section 4.1 and the simulated I/O subsystem that is used in our experiments in Section 4.2.

Ideally, we would compare ^{2}TL with an existing scheduler in the literature. However, limitations of those schedulers, discussed in Section 4.3 in detail, do not allow such comparisons. Therefore, we designed and implemented the Storage Latency Adaptive Control (SLAC) scheduler, which does not have those limitations but embodies the characteristics of many of the existing schedulers that we want to compare with ^{2}TL. Section 4.3 presents SLAC, as well as the reasons why its performance, which is also assessed using simulations, is compared with that of ^{2}TL.

Another scheduler that is used in this dissertation is First-Come-First-Serve (FCFS). Its performance is used as a baseline for comparison. Because its scheduling methodology is rather straight-forward, the performance results of our experiments with FCFS are used to show that the performance requirements of the workloads that drive the experiments cannot be met trivially, i.e., without a more sophisticated scheduling methodology. The performance results of the experiments with FCFS also are used in the methodology that we use to assign performance requirements to the streams of a workload, which is described in Section 4.4. To evaluate the efficacy of ^{2}TL, we assign performance requirements to the streams of a workload to create a range of desired scenarios. For example, these performance requirements might translate to a storage system with a performance capacity that is sufficient to meet the requirements of all of the streams of the workload. Or, they may translate to a storage system that has the performance capacity to meet only a subset

of the performance requirements of the workload.

Finally, the two methodologies we used in ^{2}TL to predict scheduling rates are discussed in Section 4.5. Recall that ^{2}TL's efficacy in providing latency guarantees relies on the accurate prediction of scheduling rates.

4.1 Enhancements to DiskSim

We enhanced DiskSim 4.0 in four ways. First, as described in Section 4.1.1, we implemented request streams, or simply streams, into the simulator. That is, we made it possible to identify requests of the same stream and individually quantify the performance of each stream. Second, as discussed in Section 4.1.2, we introduced a new I/O component to DiskSim 4.0, called "Shim", which is where the schedulers, i.e., ^{2}TL, SLAC, and FCFS, are implemented. Third, as mentioned in Section 4.1.3, we implemented the ^{2}TL, SLAC, and FCFS scheduling algorithms within DiskSim. And, finally, as outlined in Section 4.1.4, we enhanced DiskSim's synthetic workload generator to generate bursts of I/O requests (request bursts). We use the default DiskSim parameters to conduct our experiments. While there are over 200 parameters in Disksim [8], Table 4.1 lists some that could substantially affect the experimental results.

4.1.1 Tagging of Request Streams

The I/O requests that comprise an I/O stream, or request stream, in a DiskSim simulation emanate from one of two sources: (1) an externally generated I/O trace, or (2) a DiskSim synthetic workload generator. Although DiskSim allows multiple synthetic workload generators to independently generate I/O streams during a simulation, the performance results it provides do not distinguish between requests generated by different generators, i.e., associated with different I/O streams. This limitation prevents the analysis of the performance of individual I/O streams. To overcome this limitation, we introduced and implemented within DiskSim the concept of request streams, or simply streams. This was done by

adding: (1) a new request attribute, stream ID, to DiskSim's synthetic workload generator, which associates a unique numeric ID with each I/O stream and each of its requests; (2) a new field to DiskSim's ASCII request trace record format, which stores the new request attribute, stream ID; and (3) a new field to the description of each synthetic workload generator in the simulation configuration file, which defines the unique numeric ID assigned to each I/O stream. Given these additions to DiskSim, DiskSim's synthetic workload generator code was modified to cause each instantiation of a generator to read its stream ID from the simulation configuration file and include it in each request trace record that it generates. Given the stream ID of each serviced request, we are able to assess the individual performance of each stream.

4.1.2 Implementation of Schedulers in the Shim, a New I/O Component

DiskSim simulates an I/O driver as a striping device driver that has a queue associated with each disk in the disk array; this is provided in most operating systems. When a request arrives at the I/O driver, it is striped into multiple sub-requests according to the disk array organization. Then each sub-request is independently queued on its destination disk's I/O driver queue and scheduled into the storage controller.

If ^{2}TL were the scheduler in the I/O driver, we would have to implement an instance of ^{2}TL for each I/O driver queue, and these instances would have to coordinate their activities to meet a set of performance requirements. In order to avoid this complication in the implementation of ^{2}TL in DiskSim and the associated communication overhead between the ^{2}TL instances, we implemented ^{2}TL in the "Shim", a new component of the simulated I/O subsystem.

The Shim serves as a layer of abstraction between the I/O streams, which are the input to the simulation (along with DiskSim input parameters), and the I/O driver. As such, the Shim hides the disk array organization from the implementation of the scheduler, in

our case, ^{2}TL, SLAC, and FCFS. Since the Shim is not a striping device driver, it does not require multiple instances of ^{2}TL. In this dissertation, all levels of the storage hierarchy below the Shim, which include the I/O driver, the storage controller, and the disk array, are collectively represented as the storage system. More details about the Shim are discussed in Section 4.2.

4.1.3 Implementation of ^{2}TL, SLAC, and FCFS Schedulers

In order to compare the performance of ^{2}TL with that of SLAC and FCFS, simulation experiments were conducted using all three schedulers. Thus, each had to be implemented within DiskSim 4.0. The implementation of ^{2}TL is true to the description of the algorithm presented in Chapter 4. Similarly, the implementation of SLAC is true to its description in Section 4.3. In terms of FCFS, the implementation follows the description of the algorithm in Silbertchatz's operating system textbook [36].

4.1.4 Generation of Request Bursts

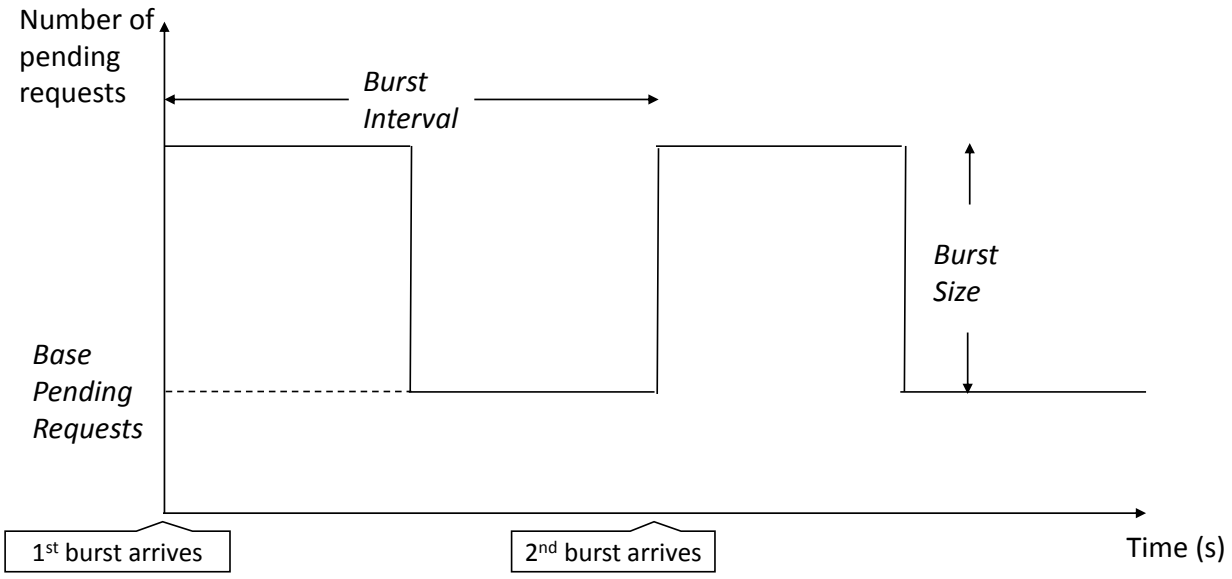
Streams with bursty behaviors switch back-and-forth between a higher request issue rate and a lower request issue rate. To cause the synthetic workload generators in DiskSim 4.0 to generate such bursty behavior, we view the time during which a burst of requests is generated as a sequence of one-second intervals. The length of this sequence, i.e., the number of seconds it encompasses, is defined by a new parameter called *Burst Interval*. Each interval is divided in half, the first half being associated with a higher request issue rate - a request burst - and the second half being associated with a lower rate. As described in the paper that introduces Maestro [29] and described briefly below, this vision is realized by controlling the number of pending requests a stream has in the I/O hierarchy.

To control the number of pending I/O requests a stream has in the I/O hierarchy, two additional new DiskSim parameters are introduced: *Base Pending Requests* and *Burst Size*. Base Pending Requests defines the smallest number of pending requests of a stream

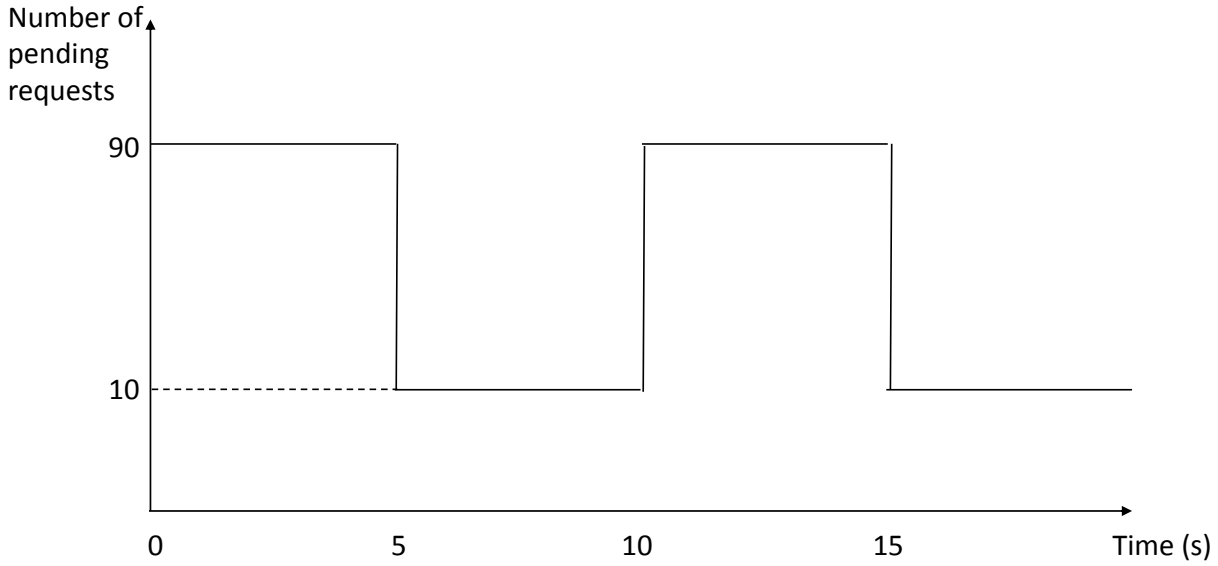
has in the I/O hierarchy. As shown in Figure 4.1a, DiskSim maintains this number of pending requests during the second half of each Burst Interval. Burst Size defines the size of a request burst, which increases the number of the stream's pending requests to Base Pending Requests + Burst Size. As shown in Figure 4.1a, DiskSim maintains this higher number of pending requests during the first half of each Burst Interval. Note that the three new parameters discussed above are new fields in the description of each synthetic workload generator in the simulation configuration file.

Figure 4.1a illustrates the variation in the number of pending requests a bursty stream has in the I/O hierarchy over time. If the stream's Burst Interval is x seconds, the number of its pending requests in the I/O hierarchy rises from the lower level to the higher level every x seconds. The number of its pending requests remains at the higher level for $x/2$ seconds before returning to the lower level for $x/2$ seconds.

Each synthetic workload generator continuously monitors the number of its pending requests (i.e., the number of a stream's pending requests) in the I/O hierarchy, which includes the Shim and the storage system. When generating a request burst, when the number of its pending requests reaches the specified level, the synthetic workload generator does not issue new requests; and, when the number of its pending requests is below the specified level, it issues requests at the highest rate possible until the number of its pending requests is at the specified level. Figure 4.1b illustrates a stream's bursty behavior when its Base Pending Requests is 10, its Burst Size is 80, and its Burst Interval is 10 seconds. At the beginning of a Burst Interval, the synthetic workload generator issues requests at the highest rate possible until the number of its pending requests reaches the high level, i.e., 90. After 5 seconds (1/2 the Burst Interval), it pauses issuing requests until the number of its pending requests drops to the low level, 10 (Base Pending Requests).



(a) Three parameters specify the bursty behavior of a stream: *Burst Interval*, *Burst Size*, and *Base Pending Requests*.



(b) Example with *Burst Interval*= 10 seconds, *Burst Size*= 80, *Base Pending Requests*= 10.

Figure 4.1: Bursty Stream Behavior.

4.2 Simulated I/O Hierarchy

The simulated I/O system is depicted in Figure 4.2. At the bottom level of the I/O system is a RAID-0 disk array made up of 8 Maxtor disk drives. Each disk drive has an Shortest Positioning Time First (SPTF)-scheduled queue that can hold up to eight requests. Above the disk array is the storage controller, which does not queue up requests for scheduling; rather, it simply passes requests scheduled from the I/O driver to the disk array. As mentioned in Section 4.1.2, the I/O driver is a striping device driver. For each disk in the disk array, there is an FCFS-scheduled I/O driver queue of unlimited capacity. When the disk queue of a disk is not full, the I/O driver schedules a request, if one is available, from the disk's I/O driver queue to the storage controller and, from there, to the disk.

Each stream has either a latency or throughput requirement. Between the streams and the I/O driver is the Shim, which has an FCFS-scheduled queue of unlimited capacity for each stream. The Shim's scheduler, i.e., FCFS, SLAC (Section 4.3), or ^{2}TL, determines the order in which requests in these queues are dispatched to the I/O driver. The Shim (upper-) bounds the number of pending requests in the storage system. For performance reasons, the disk queues of the disk array should be fully occupied. For each request that enters into the storage system, the I/O driver stripes it into at most eight sub-requests and dispatches each sub-request to the I/O driver queue of the destination disk. Thus, we set the upper-bound of the number of pending requests (before being striped) in the storage system (I/O driver, storage controller, and disk array combined) to 16, which is larger than the capacity of a disk queue (8) to ensure that all disk queues are fully occupied as long as the streams issue requests fast enough to occupy the disk queues.

4.3 Storage Latency Adaptive Control (SLAC)

To evaluate the effectiveness of ^{2}TL, it is desirable to compare ^{2}TL's performance with that of existing, competitive schedulers. In the literature, Frosting [41], Cake [40], Stone-

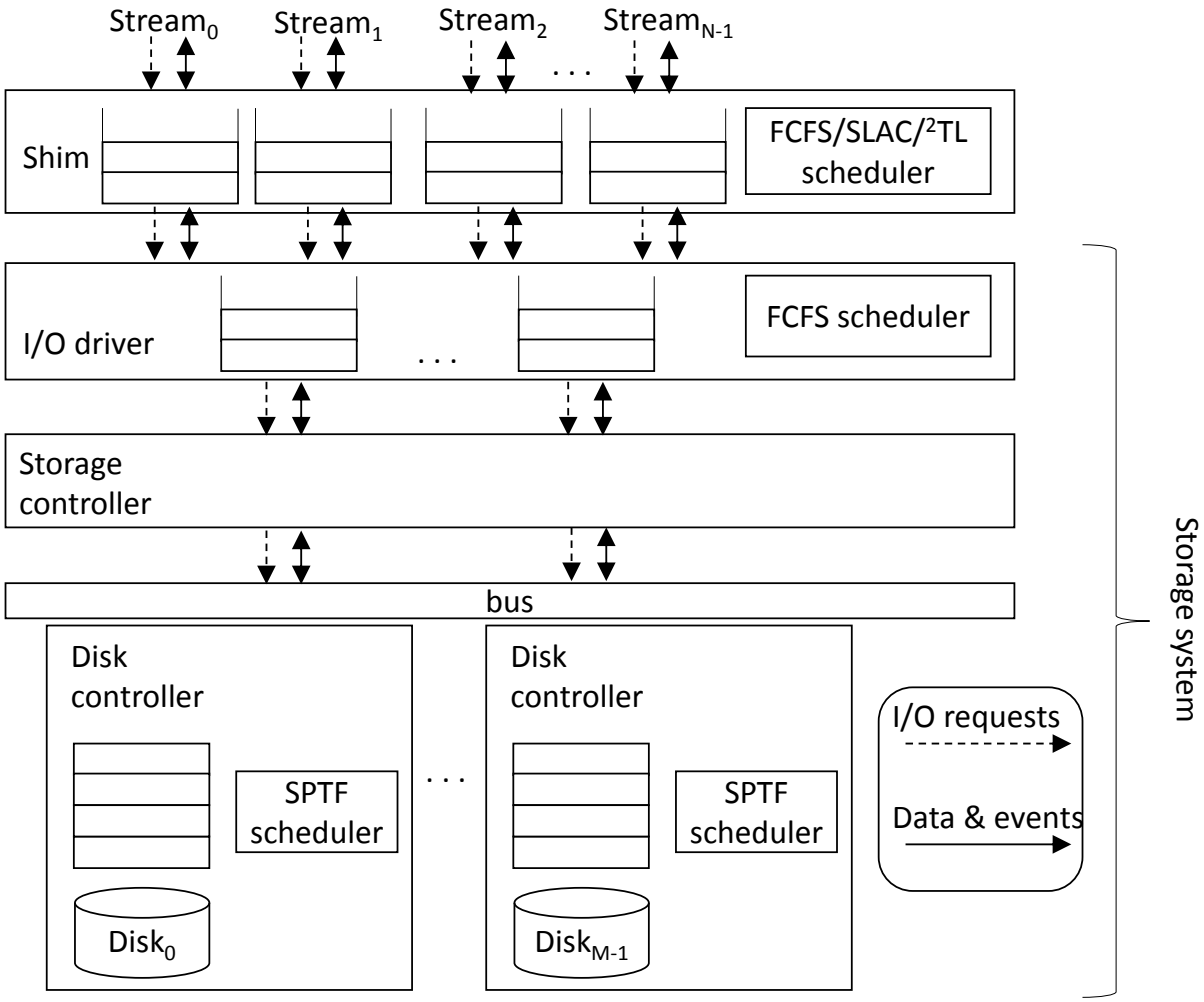


Figure 4.2: Simulated I/O Hierarchy.

henge [21], Courier [45], Maestro [29], and Fahrrad [32] are those schedulers, i.e., the only schedulers that simultaneously provide both latency and throughput guarantees. Among these schedulers, only Courier encompasses proactive scheduling of latency-bound requests and, to the best of our knowledge, Courier is the only existing I/O scheduler that includes proactive scheduling of this kind. Like ^{2}TL, it continuously monitors latency-bound stream request arrival rates, and dynamically adjusts request scheduling in order to avoid latency requirement violations. Courier has two scheduling mode: latency-constrained scheduling mode and throughput-allocation mode. But default, Courier operates in the throughput-allocation mode to meet throughput requirements of concurrent streams. Based on the request arrival rate and request service rate of each stream, Courier identifies requests that may miss their deadlines in the throughput-allocation mode. Once identified, Courier switches to the latency-constrained mode and schedules those requests. Unlike ^{2}TL, when identifying requests that may miss their deadlines, Courier assumes that requests will get serviced by the disks immediately once scheduled. Also, since Courier strives to meet the average latency requirements, it is unable to confine the amount of requests that miss their deadlines to small percentages. Failing to do so makes it unable to meet latency requirements at high percentiles. More details about these two difference with ^{2}TL is to be discussed below.

Unfortunately, we are unable to directly compare, via simulations, the performance of ^{2}TL with that of Courier for two reasons. First, while ^{2}TL assumes that the latency requirement of a stream consists of a latency target and a percentile rank, Courier assumes that the latency requirement of a stream is only a latency target – Courier does not provide percentile latency guarantees. Note that in the experiments that compared Courier with WFQ [5] and SARV+AVATAR [44], average latency is the performance metric. In addition, given a percentile latency requirement in ^{2}TL, it is not known how to translate it into an equivalent average latency requirement in Courier. Vice versa, given an average latency requirement in Courier, it is not known how to translate it into a percentile latency requirement for ^{2}TL. Note that, because of the request latency distribution, an average

latency requirement may not be equivalent to a 50th percentile latency requirement.

Second, Courier assumes no disk queuing, while ^{2}TL does consider disk queuing as well as its effect on latencies when scheduling requests. Not only do these two differences between ^{2}TL and Courier prevent us from directly comparing them, they also indicate that ^{2}TL is functionally more advanced than Courier. Since Courier does not provide percentile latency guarantees, it is unlikely to meet the stringent latency requirements of user-interacting applications [20, 40, 41]. For example, recent research suggests that providing latency guarantees at high percentiles (i.e., at least at the 95th percentile) is crucial to many user-interacting applications [40, 41, 20]. Also, Courier's exclusion of disk queuing in the disk array is an unrealistic assumption. As indicated in the literature, disk queuing is ubiquitous and, as indicated in [43], it is necessary to consider disk queuing because it enhances disk performance. In contrast, ^{2}TL assumes disk queuing and considers its effect on latencies when scheduling requests.

Since we cannot directly compare the performance of ^{2}TL and Courier via simulations, to demonstrate the advantage of ^{2}TL's proactive scheduling component, we could compare it with the reactive schedulers mentioned above, i.e., Frosting [41], Cake [40], Stonehenge [21], Maestro [29], and Fahrrad [32]. Among them, Frosting and Cake are the latest. However, we did not compare ^{2}TL with Frosting and Cake because they are designed for HBase clusters where performance requirements are expressed in terms of application-level operations such as get, put, and scan. ^{2}TL and almost all other schedulers in the literature do not support these operations, but rather block-level operations. Thus, remaining on the list are Stonehenge, Maestro, and Fahrrad. Like Courier, these schedulers do not provide percentile latency guarantees. Among them, Maestro is the most appropriate to compare against ^{2}TL because (1) it is the latest scheduler among the three, and (2) it assumes disk queuing and considers its effect on latencies when scheduling requests. However, we cannot fairly compare ^{2}TL and Maestro because Maestro does not provide percentile latency guarantees.

Accordingly, in order to evaluate ^{2}TL's performance, we designed a reactive scheduling

algorithm called *Storage Latency Adaptive Control* (SLAC) that is comparable in many ways to Maestro and the other reactive I/O schedulers that have goals similar to those of ^{2}TL. SLAC is similar to ^{2}TL and Maestro in the following three ways. First, SLAC dynamically deduces the scheduling deadlines of the requests of latency-bound streams based on the stream's request storage latency threshold and strives to meet them. As mentioned in Chapter 3, a stream's request storage latency threshold is the percentile latency in the storage system of its recently serviced requests at its percentile rank. By default, SLAC schedules requests, latency-bound and throughput-bound, according to their scheduling deadlines in the EDF order. To meet a stream's latency requirement, it exclusively schedules latency-bound streams that have missed their scheduling deadlines, if there is any. Second, SLAC prioritizes the performance requirements of the latency-bound streams of a workload over those of its throughput streams. Third, SLAC endeavors to fulfill throughput requirements and fulfills throughput guarantees through proportional sharing. However, due to its reactive nature, unlike ^{2}TL, SLAC does not proactively increase service allocation to a latency-bound stream when a request burst arrives.

SLAC is designed to be Maestro but with the capability to meet latency requirements at percentiles. It resembles Maestro functionally and embodies the following characteristics of Maestro. First, SLAC is a reactive scheduler. Second, when scheduling requests, SLAC considers latencies due to disk queuing. By comparing the performance of ^{2}TL with that of SLAC, we are able to evaluate the benefit of ^{2}TL's proactive scheduling component with respect to over the reactive scheduling on which both SLAC and Maestro are based. Chapter 5 presents the results of experiments that are used to conduct this evaluation/comparison.

To describe the details of SLAC, we use the decision tree depicted in Figure 4.3. At each scheduling opportunity, SLAC uses this decision tree to decide whether to schedule a request from a latency-bound stream or a throughput-bound stream. If a request from a latency-bound stream is available for scheduling, i.e., is pending in the Shim, then if there is one that has missed its scheduling deadline, SLAC schedules it. However, if there are

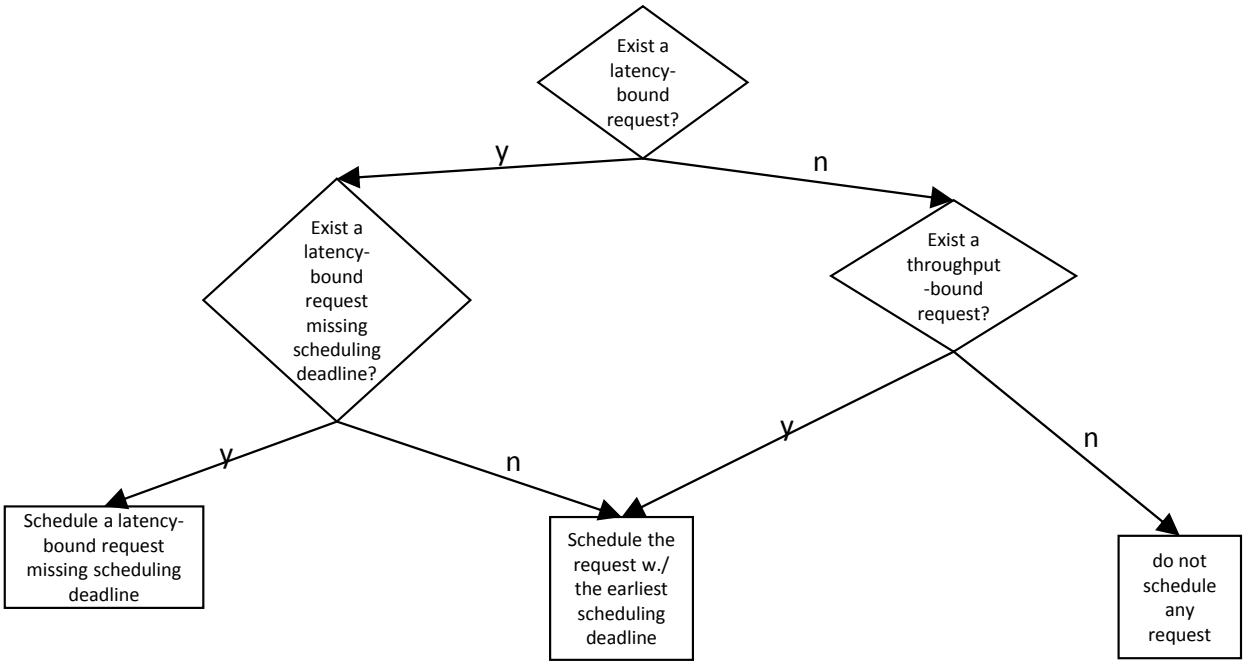


Figure 4.3: SLAC's Decision Tree.

multiple requests of latency-bound streams at the head of the streams' queues that have missed their scheduling deadlines, SLAC schedules the head request of the stream with the highest priority. On the other hand, if there is no requests of a latency-bound stream available for scheduling or if there are but they has not missed their scheduling deadlines, then SLAC schedules the request in the Shim with the earliest deadline - it can be from a latency-bound or a throughput-bound stream. Table 4.2 summarizes the parameters of SLAC used in our experiments.

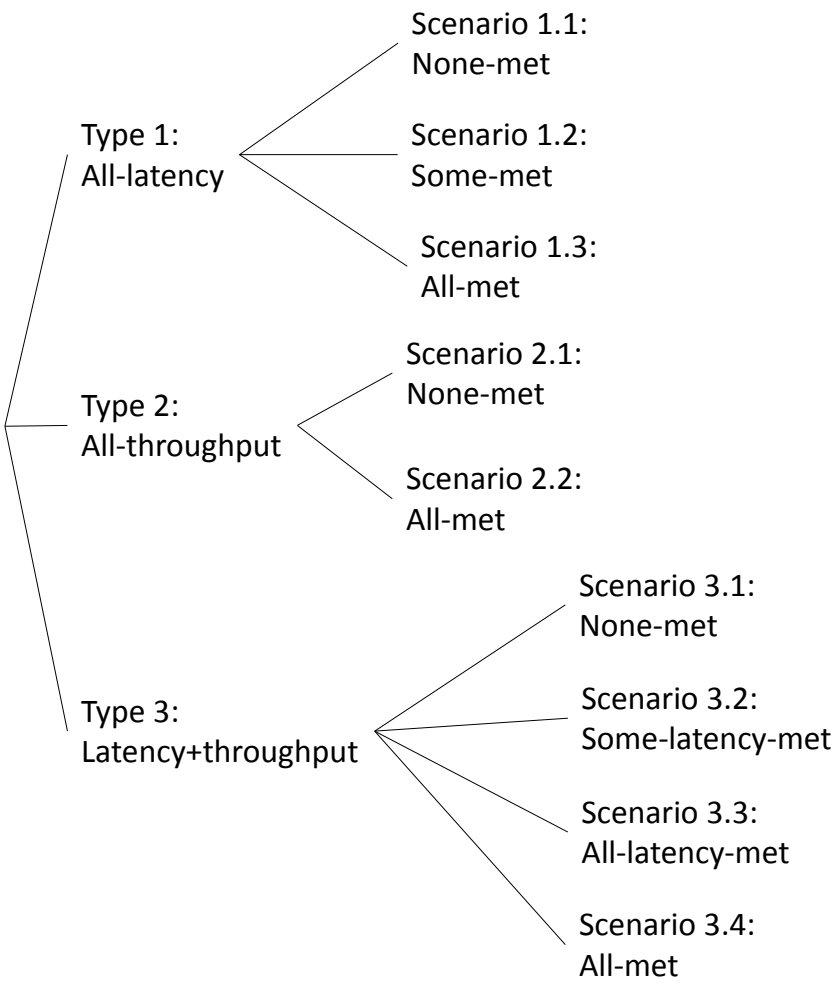


Figure 4.4: Possible Scenarios for each Type of Workload.

4.4 Performance Requirements

We evaluated the effectiveness of ^{2}TL with the experiments presented in Chapter 5. The experiments cover a wide range of scenarios and demonstrate the various aspects of ^{2}TL. In this context, a scenario is a workload and a storage system. The workloads in the experiments differ in terms of their composition (number and types of streams in the workload) and the performance requirements of their streams. Although the storage system is the same for each experiment, the performance requirements of the streams in a workload

dictate whether or not the storage system can meet all, some, or none of them. The performance requirements of the streams in the workloads that drive the experiments presented in Chapter 5 are set to create the scenarios pictured in Figure 4.4, which is discussed next. The methodology for determining what performance requirements to assign to the streams of a workload to create a particular scenario is described in the next section.

Figure 4.4 presents the three major types of workloads in our experiments and the scenarios created with each type. A workload of Type 1 consists of only latency-bound streams, while a workload of Type 2 consists of only throughput-bound streams and a Type 3 workload consists of a mix of latency- and throughput-bound streams. As shown in the figure, three scenarios were created with Type 1 workloads: (1.1) none of the performance requirements of the workload (all latency requirements) can be met (insufficient storage system performance capacity); (1.2) only some of them can be met (partially sufficient performance capacity); and (1.3) all of them can be met (sufficient performance capacity). Because SLAC and ^{2}TL proportionally share the storage service among the throughput-bound streams of a workload, only two scenarios were created with Type 2 workloads: (2.1) none of the performance requirements of the workload (all throughput requirements) can be met; and (2.2) all of them can be met. Four scenarios were created with Type 3 workloads: (3.1) none of the performance requirements of the workload can be met; (3.2) only some of the latency requirements can be met; (3.3) only the latency requirements can be met; and (3.4) all of the performance requirements of the workload can be met.

An overview of how we assign performance requirements to the streams of a workload in order to create a specific scenario is provided in Section 4.4.1. Sections 4.4.2, 4.4.3, and 4.4.4 discuss the specifics in terms of how we assign them given a Type 1, Type 2, and Type 3 workload, respectively.

4.4.1 Performance Requirement Assignment

As mentioned in Chapter 3, the performance capacity of a storage system is defined as how fast the storage system can process requests issued by the streams in a workload. This is a

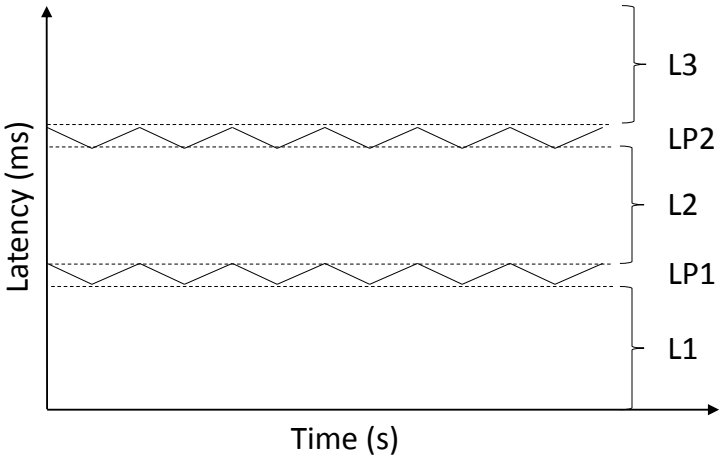


Figure 4.5: Latency Domain Demarcated into Regions L1, L2 and L3 by Latency Profiles LP1 and LP2. Table 4.3 indicates the set of streams used to obtain each Latency Profile.

limiting factor for both the latency and throughput performance achievable by the streams in the workload and, thus, for whether or not the storage system can meet their performance requirements. Thus, calculated assignment of performance requirements to the streams of a given workload determine whether or not the storage system's performance capacity is sufficient to meet them and, thus, creates a specific scenario. Next, given a workload and the desired scenario, we describe the general method that we use to determine the performance requirements to assign to the streams of a given workload and use in the related experiment.

For any scenario/experiment, the latency domain, which is a two-dimensional space with simulation time (in seconds) as the x-axis and measured request latencies (in ms) as the y-axis, is divided into three regions. These regions, called Regions L1, L2, and L3, are separated by smaller regions that are defined by the shortest and longest request latencies experienced by a set of streams of the given workload simulated with FCFS. The sets of streams of the workload used to demarcate Regions L1 and L2, and those used to demarcate Regions L2 and L3 depend on the type of the given workload and the specified scenario. As shown in Table 4.3 and Figure 4.5, Regions L1 and L2 are separated by LP1, the latency profile of the following set of streams in the workload (set

LP1) during their concurrent (isolated) access of the storage system with FCFS: (1) the highest-priority latency-bound stream if the specified scenario is Scenario 1.1 or 3.1; (2) the higher-priority latency-bound streams in the workload, i.e., those streams that are meant to have a performance requirement met by ^{2}TL, if the specified scenario is Scenario 1.2, 1.3, or 3.2; and (3) all of the latency-bound streams in the workload if the scenario is Scenario 3.3 or 3.4. Similarly, as shown in Table 4.3 and Figure 4.5, for all scenarios but 3.2, Regions L2 and L3 are separated by LP2, the latency profile of all of the latency-bound streams in the workload (set LP2) during their concurrent (shared) access of the storage system with the throughput-bound streams (if any) in the workload with FCFS. In contrast, for scenario 3.2, Regions L2 and L3 are separated by the latency profile of all of the latency-bound streams in the workload during their concurrent (isolated) access of the storage system with FCFS, i.e., the throughput-bound streams are not included in this simulation. As will be described below, to create Scenario 1.3, an additional simulation is required, i.e., a simulation with ^{2}TL.

FCFS does not distinguish between the requests of different streams and, thus, gives no preference to latency-bound or throughput-bound streams, and does not support stream priorities. It follows a straight-forward algorithm, which is described in Silbertchatz's operating system textbook [39]. Thus, when an I/O workload is scheduled (via DiskSim) with FCFS, the streams in the workload equally share access to the storage system.

Accordingly, the LP1 latency profile defines the performance capacity of the storage system. Thus, if the streams in set LP1 had performance requirements less than the shortest request latency in the LP1 latency profile, i.e., in Region 1, then the storage system would not have sufficient performance capacity to fulfill them with ^{2}TL. Analogously, if the streams in set LP1 had performance requirements equal to or greater than the longest request latency in the LP1 latency profile (with FCFS), i.e., in Region 2 or Region 3, then the storage system would have sufficient performance capacity to fulfill them with ^{2}TL assuming that two conditions are true. First, the request arrival rate (in IOPS) of each latency-bound stream in the workload never exceeds the rate at which the storage system

services the stream's requests, i.e., the stream's request service rate. Second, the number of pending requests in the I/O hierarchy of each latency-bound stream in the workload is never so large that the end-to-end latencies of the stream's requests exceed its latency target.

Since we desire scenarios in which the performance requirements of the streams can only be met with sophisticated scheduling, Region 2 is the region from which the performance requirements are selected. This is because the latencies in the LP2 latency profile indicate that if the streams in set LP2 had performance requirements equal to the longest request latency in the LP2 latency profile, i.e., in Region 3, then they could be met by FCFS, as well as ^{2}TL.

When a workload contains throughput-bound streams, we need to work with the throughput domain as well (or solely if the workload contains only throughput-bound streams) to determine the performance requirements to assign to the throughput-bound streams of the workload. The throughput domain is also a two-dimensional space but with simulation time (in seconds) as the x-axis and measured throughput (in MB/s) as the y-axis. For the purpose of determining the performance requirements to assign to the streams, as shown in Figure 4.6, the throughput domain is divided into only two regions, Regions T1 and T2. These two regions are separated by a smaller region that is defined by the Throughput Profile (TP1), i.e., the shortest and longest throughput achieved by the throughput-bound streams in the workload during their concurrent (isolated) access (with no other streams) of the storage system simulated with FCFS. The reason only two regions are needed to determine throughput targets for the throughput-bound streams of a workload is because SLAC and ^{2}TL proportionally share storage service among these streams according to their throughput requirements. Thus, if the scheduler is not able to fulfill the throughput requirement of one of the throughput-bound streams of a workload, it is not able to fulfill any of the throughput requirements of the throughput-bound streams of the workload.

Accordingly, the general algorithm that we follow to assign performance requirements to the streams of a given workload and, thus, create a specific scenario follows. Note that,

as mentioned above, to create Scenario 1.3, an additional simulation is required, i.e., a simulation with ^{2}TL. Also, note that the assumptions that are listed after the algorithm must be taken into consideration when assigning performance requirements to the streams.

1. If the workload contains latency-bound streams,

   a. Run the simulations with FCFS that are required to collect the Latency Profiles LP1 and LP2 (the workloads for these simulations are specified in Table 4.3).

   b. Divide the latency domain into three regions, Regions L1, L2, and L3, using the Latency Profiles (LP1 and LP2).

   c. Using latency targets in Region L2, assign latency requirements to the streams in set LP1 with consideration of the assumptions mentioned below.

   d. Using latency targets in Region L2 or LP2, assign latency requirements to the streams in set LP2 with consideration of the assumptions mentioned below.

2. If the workload contains throughput-bound streams,

   a. Run a simulation with FCFS with a workload that contains only the throughput-bound streams in the workload in order to collect the Throughput Profile (TP1).

   b. Divide the throughput domain into two regions, Regions T1 and T2, using the TP1.

   c. Using throughput targets in Region T2, assign latency requirements to the throughput-bound streams with consideration of the assumptions mentioned below.

When assigning performance requirements to the streams of a workload, the following assumptions must be taken into consideration:

1. Given a Type 3 workload, i.e., one that contains both latency- and throughput-bound streams, service to latency-bound streams is prioritized over that provided to throughput-bound streams.

2. Given a Type 1 workload (one that contains all latency-bound streams) or a Type 3 workload that has latency-bound streams, each latency-bound stream has a distinct priority. The highest-priority latency-bound stream has Priority 1.

3. Given a workload that has throughput-bound streams (Type 2 or Type 3), throughput-bound streams have the same priority.

4. Given a workload that has latency-bound streams (Type 1 or Type 3), if the performance requirements of only some of the latency-bound streams are met, these must have higher priorities than the other latency-bound streams in the workload.

Below we describe how after steps 1a and 1b, and if applicable, 2a and 2b, of the general algorithm, performance requirements are assigned to the streams in the workloads that drive the experiments presented in Chapter 5, with consideration of the above-mentioned assumptions. In Chapter 5, for each experiment we present the latency profiles (LP1 and LP2) and provide the reasoning for the performance requirements assigned to the streams in the workload that drives the experiment.

4.4.2 Type 1: All Streams are Latency-bound

For Type 1 workloads, three scenarios are possible: none of the performance (latency) requirements of the streams in the given workload is met (Scenario 1.1: None-met); some are met (Scenario 1.2: Some-met); and all are met (Scenario 1.3: All-met). The assignment of performance requirements to the streams of the workload for these three scenarios are discussed below.

Scenario 1.1 None-met: In this scenario, where none of the performance (latency) requirements of the workload is to be met, first we assign a latency target in Region L1 to the highest-priority latency-bound stream because in this region there is insufficient performance capacity to fulfill it. Since the latency requirement of the highest-priority stream in the workload cannot be met by ^{2}TL, the latency requirements of the rest of

the streams in the workload cannot be met by ^{2}TL either. In fact, they will receive no service. Thus, it does not matter what performance requirements we assign to the rest of the streams. Nonetheless, we assign to each a latency target in LP2.

Scenario 1.2 Some-met: In this scenario, where only the performance (latency) requirements of some of the streams in the workload are to be met, we divide the streams into two groups according to their priorities: the high-priority and low-priority streams.

In order to make it possible for ^{2}TL to fulfill the latency requirements of the high-priority streams, we assign them latency targets in Region L2, where there is sufficient performance capacity to meet their latency requirements but only with careful scheduling. Since we do not want it to be possible for ^{2}TL to fulfill the latency requirements of the low-priority streams and since to meet the latency requirements of the high-priority streams, ^{2}TL will allocate less service to the low-priority streams than will FCFS, we assign latency targets in LP2.

Scenario 1.3 All-met: In this scenario, where the performance (latency) requirements of all of the streams in the workload are to be met, we take this opportunity to demonstrate the effectiveness of ^{2}TL in simultaneously meeting the performance requirements of streams with different latency targets. Again, we divide the streams into two groups, according to their priorities: the high-priority and low-priority streams. After running the simulations (one with the high-priority streams to obtain LP1 and one with all the streams to obtain LP2) with FCFS, in this case, a test simulation with ^{2}TL is required to finalize the assignment of the latency requirements to the low-priority streams. This is because the simulations with FCFS do not reveal the performance capacity available to the low-priority streams after ^{2}TL meets the performance requirements of the high-priority streams. Without this information, we are unable to assign performance requirements to the low-priority streams that can be met by ^{2}TL. In this test simulation, as we did above in Scenario 1.2, we assign latency targets in Region L2 to the high-priority streams and latency targets in LP2 to the low-priority streams. However, note that the latency requirements of the low-priority streams will be changed as a result of this test simulation. The performance

capacity is sufficient for ^{2}TL to meet the latency requirements of the high-priority streams (in Region L2), but only with careful scheduling. But to meet them, ^{2}TL will allocate less service to the low-priority streams than will FCFS. This is why the low-priority streams will achieve longer latencies with ^{2}TL (test simulation) than they will with FCFS, and their originally assigned latency requirements in LP2 will not be met by ^{2}TL. Based on the latencies achieved by the low-priority streams in the test simulation (with ^{2}TL), we assign them performance requirements for the experiments/simulations with SLAC and ^{2}TL. For each low-priority stream, we set its latency target above its longest achieve latency during the test simulation (with ^{2}TL). The latency requirements of the high-priority streams assigned for the test simulation do not change. This ensures that, in the experiments, after meeting the performance requirements of the high-priority streams, there is sufficient performance capacity to meet the performance requirements of the low-priority streams. As a result, the performance capacity is sufficient to simultaneously meet all latency requirements.

4.4.3 Type 2: All Streams are Throughput-bound

Because both SLAC and ^{2}TL allocate service to throughput-bound streams proportional to their throughput targets, either all performance requirements in a Type 2 workload are met (Scenario 2.2) or none is met (Scenario 2.1). The assignment of performance requirements to the streams of the workload for these four scenarios are discussed below.

Scenario 2.1 None-met: In this scenario, the performance (throughput) requirements of all of the streams in the workload are not to be met. Thus, we assign throughput targets to the streams such that their sum is larger than TP1. Since we know that FCFS was able to provide aggregate throughput service to these streams that is in the range of TP1, if the target aggregate throughput service is larger than this, i.e., in Region T2, then the storage system, ^{2}TL will not be able to meet the aggregate throughput requirement. And, therefore, the individual stream throughput requirements of the workload will not be met by ^{2}TL.

Scenario 2.2 All-met: In this scenario, the performance (throughput) requirements of

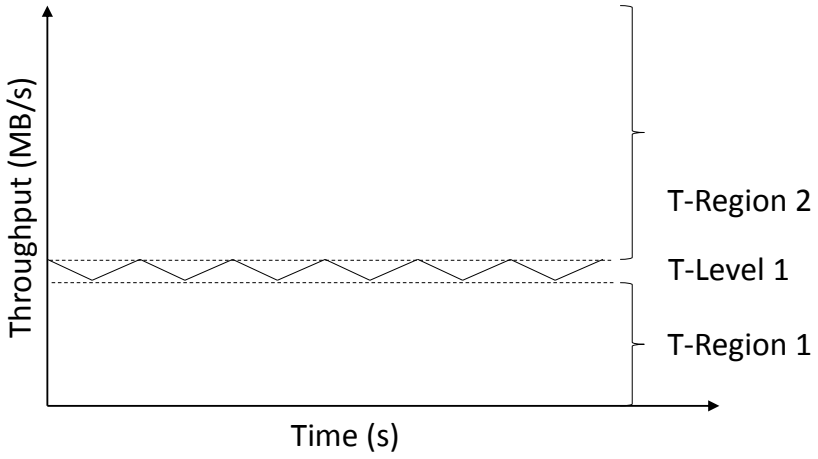


Figure 4.6: Throughput Domain Divided into Regions T1 and T2 by the Latency Profile of All of the Throughput-bound Streams Simulated in Isolation with FCFS.

all of the streams in the workload are to be met. Thus, we assign throughput targets to the streams such that their sum is smaller than TP1, i.e., in Region T1. Since we know that FCFS was able to provide aggregate throughput service to these streams that is in the range of TP1, if the target aggregate throughput service is smaller than this, i.e., in Region T1, then the storage system, ^{2}TL will be able to meet the aggregate throughput requirement. Therefore, the individual stream throughput requirements of the workload will be met.

4.4.4 Type 3: Mix of Latency- and Throughput-bound Streams

For Type 3 workloads, four scenarios are possible: none of the performance requirements of the streams in the given workload is met (Scenario 3.1: None-met); some are met, in particular the performance requirements of the higher-priority latency streams in the workload (Scenario 3.2: Some-latency-met); some are met, in particular the performance requirements of all the latency streams in the workload (Scenario 3.3: All-latency-met);

and all are met (Scenario 3.4: All-met). The assignment of performance requirements to the streams of the workload for these four scenarios are discussed below.

Scenario 3.1 None-met: In this scenario, where none of the performance requirements of workload is to be met, we first assign a latency target in Region L1 to the highest-priority latency-bound stream. Then we assign to the other latency-bound streams in the workload latency targets in LP2 (latencies during shared access by all the streams in the workload with FCFS). Since the performance requirement of the highest-priority latency-bound stream cannot be met if it is in Region L1, the other streams (both latency- and throughput-bound) will not receive any service from ^{2}TL and, therefore, their performance requirements cannot be met as well.

Scenario 3.2 Some-latency-met: In this scenario, only the performance requirements of the specified set of higher-priority latency-bound streams are to be met. Thus, we assign latency targets in Region L2 to the higher-priority latency-bound streams and latency targets in LP2 to the other latency-bound streams in the workload. In order to meet the performance requirements of the higher-priority latency-bound streams, SLAC and ^{2}TL will allocate less service to the lower-priority latency-bound streams than will FCFS. Thus, assigning the lower-priority streams latency requirements in LP2, will result in ^{2}TL not being able to meet them. Since not all latency requirements are met, the throughput-bound streams will not receive any service from ^{2}TL and, therefore, their throughput requirements will not be met. Thus, we inadvertently assign the throughput-bound streams throughput targets in TP1.

Scenario 3.3 All-latency-met: In this scenario, only the performance requirements of all of latency-bound streams are to be met. As in Scenario 3.2 (and for the same reasons) we assign the throughput-bound streams throughput targets in TP1. Also as in Scenario 3.2 (and for the same reasons but now for all the latency-bound streams, rather than just for the higher-priority latency-bound streams), we assign to each latency-bound stream a latency target in Region L2. The performance capacity is sufficient to simultaneously meet latency requirements in Region L2 only if careful scheduling is employed, i.e., not with

FCFS.

Scenario 3.4 All-met: In this scenario, all performance requirements (latency- and throughput-bound) of the workload are to be met. To create such a scenario, as in Scenario 3.3, we assign latency targets in Region L2 to the latency-bound streams that can all be simultaneously met by ^{2}TL. Note that the simulations with FCFC for obtaining LP1, LP2, and TP1 do not reveal the performance capacity available to the throughput-bound streams after ^{2}TL meets the performance requirements of the latency-bound streams. It only indicates that the total performance of the throughput-bound streams with SLAC or ^{2}TL must be below TP1. Without this information, we are unable to assign performance requirements to the throughput-bound streams that can be met by ^{2}TL. To assign throughput targets that can be met, we conservatively estimate the performance capacity available to throughput-bound streams, which is significantly below TP1. Then, we divide it among the throughput-bound streams according to the ratio of the throughput targets specified for the experiment. If the experiment with ^{2}TL does not result in the performance requirements of all of the streams being met, this indicates that we over-estimated the performance capacity available to the throughput-bound streams. In this case, we lower the estimate of the performance capacity available to the throughput-bound streams and re-run the experiment, iterating until we have assigned appropriate throughput requirements to the throughput-bound streams of the workload.

Table 4.1: Default DiskSim Parameters.

Simulation Parameters	
Seed used during random number generator initialization	42
Seed used after random number generator initialization	42
Synthetic Workload Generator	
Blocking factor (all generated request starting addresses and sizes are multiple of this value)	8
Probability of sequential access	0%
Request Size	Exponential Distribution with base=0 and mean=8
I/O Driver	
Using queuing in subsystem	True
Scheduling algorithm	FCFS
RAID	
RAID Level	1
Stripe unit	1
No. of disks	8
Disk type	Maxtor Atlas 10K5
Disk	
Max. queue length	8
Scheduling algorithm	SPTF
No. of buffer segments	33
Max. no. of write segments	11
Segment size	1200 blocks

Table 4.2: SLAC Parameters.

Deduction of Request Storage Latency Threshold	
No. of recently serviced requests in history	200
Scheduling	
Scheduling criteria used when multiple head requests has missed scheduling deadlines	Stream priority
Scheduling algorithm used when no request has missed scheduling deadlines	EDF

Table 4.3: Set of Streams used to Obtain Latency Profile LP1 and LP2 in each Scenario.

Stream Set	1.1	1.2	1.3	3.1	3.2	3.3	3.4
LP1	Highest-priority	High-priority	High-priority	Highest-priority	High-priority	All latency-bound	All latency-bound
LP2	All	All	All	All	All latency-bound	All	All

4.5 Exclusive Scheduling Rate of Latency-bound Requests

As mentioned in Chapter 3, in order to meet a stream's latency requirement, ^{2}TL must meet the scheduling deadlines of the stream's requests. Meanwhile, ^{2}TL has to avoid overly meeting latency requirements in order to meet throughput requirements. Therefore, ^{2}TL does not schedule latency-bound requests until they will, otherwise, miss their scheduling deadlines. When ^{2}TL schedules latency-bound requests, it exclusively schedules the requests in all the latency-bound streams in the workload that need to be scheduled to avoid missing their scheduling deadlines. As mentioned in Chapter 3, the rate at which ^{2}TL schedules these requests is called the Exclusive Scheduling Rate of Latency-bound Requests.

^{2}TL predicts this scheduling rate (in IOPS), used for each latency-bound stream in a workload, to use the next time it exclusively schedules latency-bound requests. We experimented with two prediction strategies discussed in Section 3.4.5. For Strategy 1, ^{2}TL sets the next Exclusive Scheduling Rate of Latency-bound Requests to the scheduling rate it was using to schedule the requests of all the streams in the workload in the recent past. In our experiments, the recent past is the last one-second interval, i.e., ^{2}TL uses the scheduling rate that was employed since the beginning of the last one-second interval. For Strategy 2, ^{2}TL uses a moving-percentile model to estimate the next Exclusive Scheduling Rate of Latency-bound Requests based on recent exclusive scheduling rates. The performance of ^{2}TL relies on the appropriate selection of parameters, namely the number of recent exclusive scheduling rates (i.e., history size) and the percentile of the moving-percentile model. In general, we recommend using low percentiles (e.g., the 10th percentile) to estimate the next Exclusive Scheduling Rate of Latency-bound Requests. A low percentile leads to a conservative estimate and, therefore, avoids over-estimating the scheduling rate and, thus, ^{2}TL missing request scheduling deadlines. If one or more latency-bound streams in a workload have fluctuating scheduling rates, the moving-percentile model may need to

estimate the next Exclusive Scheduling Rate of Latency-bound Requests based on more recent exclusive scheduling rates. Therefore, a large history size (e.g., 500 to 1,000) may be necessary. Otherwise, a small history size (e.g., 10) is sufficient. If a set of parameters for Strategy 2 does not lead to satisfactory results, parameter tuning may be necessary. In Chapter 5, the parameter values used in each experiment that employed Strategy 2 are presented.

We used both strategies for predicting a stream's exclusive scheduling rate in the experiments presented in Chapter 5. For the experiments driven by synthetic workloads, except one (Experiment 6c), the choice of prediction strategy does not lead to noticeable changes in the results. However, the history-based prediction strategy was required for both Experiment 6c and Experiment 5a, which is driven by a real workload. Although the history-based prediction strategy may work well for any workload, we hesitate to recommend it in general because its performance depends on the values of the model's input parameters and, thus, parameter tuning may be necessary. Future work will attempt to provide heuristics for setting the model's parameters appropriately.

In Chapter 5, for the experiments driven by synthetic workloads, except Experiment 6c, we present the results of simulations that use the simple prediction strategy, while for Experiment 5a and 6c, we present the results of simulations that use the history-based prediction strategy. However, given the above results concerning the strategies for predicting a streams exclusive scheduling rate, it is clear that the history-based prediction strategy should be employed and that future work should explore more sophisticated means of prediction. Table 4.4 summarizes the parameters of ^{2}TL used in our experiments.

Table 4.4: ^{2}TL Parameters.

Exclusive Scheduling Rate Prediction: Strategy 1		
No. of past intervals used for calculating current scheduling rate	1	
Exclusive Scheduling Rate Prediction: Strategy 2		
	Experiment 5a	Experiment 6c
No. of recent exclusive scheduling rates used for prediction	10	1000
Percentile used for prediction	10th	50th ($Stream_0$) 1st ($Stream_1$)
Deduction of Request Storage Latency Threshold		
No. of recently serviced requests in history	200	

Chapter 5

Experimental Results

The efficacy of ^{2}TL was evaluated in experiments driven by synthetic and real workloads. Section 5.1 introduces the performance metrics used for the evaluation. The experiments, discussed in Section 5.2, are organized into six sets, each of which demonstrates different properties of ^{2}TL. Section 5.3 briefly describes the synthetic and real workloads used in the experiments, while Section 5.4 presents the details of each set of experiments, including the properties of ^{2}TL that are demonstrated, a detailed description of the workloads that drive the experiments in the set, the experimental results, and analyses of the results.

We evaluated the efficacy of ^{2}TL over a range of experimental settings. The experiments are driven by workloads that are comprised of only latency-bound I/O streams, only throughput-bound streams, or a mix of both. Some workloads contain streams that issue request bursts, while the others do not. The performance requirements of the streams of the different workloads are varied so that the experiments present different scenarios: when the performance requirements of none of the streams of a workload can be met, when only some of the performance requirements of the streams of a workload can be met, and when the performance requirements of all of the streams of a workload can be met. Together these experiments provide evidence of the following five properties of ^{2}TL:

P1. Given a storage system with sufficient performance capacity to meet the performance requirements of all of the streams in a given workload, ^{2}TL will simultaneously meet the performance requirements of all of the streams, be they only latency-bound streams with or without request bursts, only throughput-bound streams, or a mix of both. (Recall that, in Chapter 3, the performance capacity of a storage system is defined as its sustainable performance in handling I/O requests.)

P2. Given a storage system with insufficient performance capacity to meet the performance requirements of all of the streams in a given workload, if the workload contains both latency-bound and throughput-bound streams, ^{2}TL will prioritize latency requirements over throughput requirements.

P3. Given a storage system with insufficient performance capacity to meet the performance requirements of all of the streams in a given workload, if the performance capacity is insufficient to meet the performance requirements of all of the latency-bound streams, ^{2}TL endeavors to meet the latency requirements based on the priorities that were assigned to the streams.

P4. Regardless of the performance capacity of the storage system, ^{2}TL allocates service to the throughput-bound streams in a given workload proportional to their throughput requirements.

P5. ^{2}TL's proactive scheduling of requests of the latency-bound streams in a given workload provides better performance, as compared to SLAC's reactive scheduling, when there exist one or more latency-bound streams in the workload that issue request bursts. As the burstiness of the latency-bound stream(s) increases, so does the comparative performance of ^{2}TL.

5.1 Performance Metrics

This section presents the metrics that we use to evaluate if the performance delivered to I/O streams in a workload by a particular scheduler fulfills the streams' performance requirements. As explained in Section 3.1, the performance requirements of an I/O stream are specified in terms of either latency or throughput. In an experiment, the delivered performance is evaluated at the end of each one-second interval. Thus, for a latency-bound stream, we compute the percentile latency performance (specified by the percentile rank of the stream's latency requirement) of the end-to-end latencies (in the Shim and storage

system combined) of all requests serviced during each one-second interval. If this number does not exceed the stream's latency target, its latency requirement is met during the interval.

While percentile latency is a well-accepted latency metric in the literature [41, 40, 44], it has a shortcoming when it comes to analysis. When the performance delivered to a stream during an interval does not meet the stream's latency requirement, its percentile latency does not reveal the percentage of its requests that met the latency target. Therefore, we introduce another latency metric, which we call the *meet rate*, denoted by $MeetRate_i$ for $Stream_i$. It is defined as the percentage of $Stream_i$'s serviced requests that met its latency target during a time interval. If $MeetRate_i$ is at least as large as $Stream_i$'s percentile rank, its latency requirement is met. When the latency requirement of a stream is not met during an interval, using its meet rate as a performance metric enables us to determine the scheduler that comes closest to meeting the stream's latency requirement. Note that, to the best of our knowledge, this dissertation is the first publication in the literature to adopt meet rate as a performance metric to evaluate latency guarantees. To determine if $Stream_i$'s latency requirement is met during a period comprised of a number of consecutive time intervals, we compute its *average meet rate*, denoted by $AvgMeetRate_i$, which is the average of $Stream_i$'s interval meet rates over the specified period of time. If $AvgMeetRate_i$ is at least as large as $Stream_i$'s percentile rank, $Stream_i$'s latency requirement is met during the period. Another use $AvgMeetRate_i$ is to determine if ^{2}TL has overly prioritized the latency requirements of a workload over its throughput requirements. When the performance capacity of the storage system is insufficient to simultaneously meet the performance requirements of all of the streams in a workload, ^{2}TL gives preference to the workload's latency requirements (if any) over its throughput requirements (if any). If ^{2}TL meets the workload's latency requirements, it allocates as much of the remaining performance capacity as is possible to the workload's throughput-bound streams. Therefore, if $AvgMeetRate_i$ exceeds $Stream_i$'s percentile rank, ^{2}TL has overly prioritized and overly met $Stream_i$'s latency requirement.

Similar to latency performance, during each one-second time interval we calculate a stream's throughput (in MB/s), denoted by $Throughput_i$ for $Stream_i$, based on the volume (in MB) of $Stream_i$'s requests serviced during the interval. If, during an interval, a stream's throughput is at least as large as its target throughput, then its throughput requirement is met during the interval. Similar to the average meet rate for latency-bound streams, to determine if a stream's throughput requirement is met during a period comprised of a number of consecutive time intervals, we compute its *average throughput*, denoted by $AvgThroughput_i$ for $Stream_i$, which is the average of its interval throughputs over the time period. $Stream_i$'s throughput requirement is met if its average throughput is at least as large as its target throughput. Otherwise, we calculate $Stream_i$'s *throughput deficiency*, denoted by $Deficiency_i$, which is defined in terms of $ThroughputTarget_i$, $Stream_i$'s throughput target, and $AvgThroughput_i$, its average throughput as follows:

$$ThroughputDeficiency_i = \frac{ThroughputTarget_i - AvgThroughput_i}{ThroughputTarget_i} * 100\%. \quad (5.1)$$

When computing a stream's average meet rate and average throughput for an experimental run, its meet rate and throughput in the first interval are excluded. This is because at the beginning of an experimental run SLAC and ^{2}TL do not have sufficient history data to accurately estimate their parameter values, i.e., the request storage latency threshold (for both SLAC and ^{2}TL) and the exclusive scheduling rate for latency-bound requests (for ^{2}TL only).

As mentioned in Chapter 3, ^{2}TL provides throughput guarantees by proportionally allocating storage service to throughput-bound streams based on their throughput requirements. When the performance capacity of the storage system available to the throughput-bound streams is sufficient, proportional service allocation leads to throughput guarantees. Therefore, if there are multiple throughput-bound streams in the workload driving an experiment, we calculate the error in proportional throughput sharing for each stream, i.e., $Error_i$ for $Stream_i$. As shown in Equation 5.2, $Error_i$, $i \in \{1, ..., n\}$, where n is the number of streams, is defined in terms of $AvgThroughput_i$, the average throughput of $Stream_i$,

and $ThroughputTarget_i$, its throughput target as follows:

$$\begin{aligned} Error_i &= (ActualRatio_i - IdealRatio_i) * 100\%, \text{ where} \\ IdealRatio_i &= \frac{ThroughputTarget_i}{\sum_i (ThroughputTarget_i)} \text{and} \\ ActualRatio_i &= \frac{AvgThroughput_i}{\sum_i (AvgThroughput_i)}. \end{aligned} \tag{5.2}$$

Note that in this dissertation the I/O duration of each throughput-bound stream spans the entire experiment. That is, a throughput-bound stream issues requests fast enough to consume its allocated storage service in order to meet its throughput requirement, providing that the storage system's performance capacity is sufficient. This is because, in experiments driven by synthetic workloads, the synthetic workload generator maintains a constant number of pending requests for each throughput-bound stream in the I/O hierarchy, i.e., the Shim, I/O driver, and storage system. This guarantees that each throughput-bound stream always has sufficient pending requests to consume the throughput service requested. In Experiment Set 5, in which a workload comprised of two real I/O traces drives the experiment, the throughput-bound stream issues requests at a throughput consistently above its throughput target.

5.2 Overview of Experiments

Given a storage system with sufficient performance capacity to meet the performance requirements of a set of I/O streams concurrently accessing a storage system, ^{2}TL guarantees that it will meet all of the streams' performance requirements, i.e., ^{2}TL provides performance guarantees. Each of the streams has either a latency requirement or a throughput requirement. ^{2}TL meets latency requirements by dynamically adjusting the request scheduling rate of each latency-bound stream based on its experienced request storage latencies and its request arrival rate. To meet the latency requirement of a stream at a high percentile, upon the arrival of a burst of requests, ^{2}TL proactively increases the stream's request scheduling rate. ^{2}TL meets throughput requirements by allocating service

to throughput-bound streams proportional to their throughput targets. ^{2}TL can provide a workload with throughput guarantees as long as it (1) meets the performance requirements of all of the throughput-bound streams in the workload; (2) does not overly meet the latency requirements of the workload, and (3) proportionally allocates service to the throughput-bound streams.

When the storage system does not have sufficient performance capacity to meet the performance requirements of all of the streams in a workload, ^{2}TL cannot provide performance guarantees. This also can happen when a bursty stream's request arrival rate (in IOPS) is higher than the storage system's request service rate or when it has a large number of pending requests that cause a long queuing delay. Nonetheless, since ^{2}TL gives preference to the scheduling of the requests of the latency-bound streams in a workload, several scenarios can result:

a. If the latency-bound streams in the workload consume all of the storage system's performance capacity, then the throughput-bound streams (if any) receive none. If the performance capacity is sufficient to meet the performance requirements of all of the latency-bound streams, then their latency requirements are met. If not, then either some (the ones with the higher priorities) are met and some are not (the ones with the lower priorities), or none are met. Of course, the throughput requirements of the throughput-bound streams (if any) are not met. This is demonstrated by Experiment 3b.

b. If the latency-bound streams in the workload do not consume all of the performance capacity, then the throughput-bound streams' performance requirements are not met. However, they do receive service and that service is proportionately shared according to their throughput requirements. In this case, the performance requirements of all of the latency-bound streams may be met (as stated in "a." above). This is demonstrated by Experiments 1b and 2a.

c. If the workload is comprised of only throughput-bound streams, then the throughput

requirements of all of the streams will not be met. This is because ^{2}TL allocates service to throughput-bound streams proportional to their throughput requirements. As demonstrated by Experiment 6e, given a workload comprised of only throughput-bound streams and sufficient performance capacity, all of the throughput requirements are met.

As indicated by the three scenarios described above, ^{2}TL focuses on providing latency guarantees. More specifically, it is designed to meet the latency requirements of streams that issue request bursts. ^{2}TL's proactive scheduling component, which gives priority to the scheduling of the requests of the latency-bound streams in a workload, is particularly beneficial when latency-bound streams issue request bursts, which is the case in most of the experiments that we conducted to demonstrate the efficacy of ^{2}TL. When the latency-bound streams in a workload do not issue request bursts (Experiment Set 1) or when all of the streams in a workload are throughput-bound (Experiment 6e), ^{2}TL's proactive scheduling component does not provide an advantage over SLAC's reactive scheduling.

The efficacy of ^{2}TL's proactive scheduling in meeting latency requirements is subject to the storage system's performance capacity in two ways. First, as demonstrated by Experiment Set 5, a stream's latency requirement may not be met if its request arrival rate (in IOPS) is higher than the storage system's request service rate (also, in IOPS). Second, as demonstrated by Experiment 1a, a stream's latency requirement may not be met if it has a large number of pending requests that cause a queuing delay that is longer than its request latency target. However, it is important to note that it is possible for ^{2}TL to provide a latency guarantee to each latency-bound stream in a workload when the streams' total request arrival rate does not exceed the storage system's request service rate, and the number of pending requests of each stream does not cause long queuing delays. Otherwise, ^{2}TL cannot provide a latency guarantee to the streams.

To provide evidence of the five properties of ^{2}TL (P1-P5) listed in the introduction to this chapter and to demonstrate the various scenarios described above, we conducted six sets of experiments, each of which is simulated with the three different I/O schedulers used in this dissertation, i.e., FCFS, SLAC and ^{2}TL:

- Experiment Set 1: Non-bursty Access Characteristics (Section 5.4.1),
- Experiment Set 2: Bursty Access Characteristics (Section 5.4.2),
- Experiment Set 3: Latency-bound Streams with Different Burst Intervals (Section 5.4.3),
- Experiment Set 4: Scalability (Section 5.4.4),
- Experiment Set 5: Real Workload (Section 5.4.5), and
- Experiment Set 6: Homogeneous Performance Requirements (Section 5.4.6).

Section 5.3 briefly describes the synthetic and real workloads used in the experiments, while Section 5.4 presents the details of each set of experiments. For each experiment we identify the properties of ^{2}TL that it demonstrates as well as the scenarios that it presents. Referring to the properties of ^{2}TL as P1, P2, P3, P4, and P5, Table 5.1 relates the experiments to the demonstration of the properties. In addition, for each experiment we provide a detailed description of the workload that drives it, the experimental results, and analysis of the results.

5.3 Workloads

Synthetic workloads drive all of the experiments except for Experiment 5 in which a real (application) workload is used. Synthetic workloads are generated dynamically (during a simulation) by DiskSim's synthetic workload generators. For our experiments, each synthetic workload generator was parameterized to generate one stream of random-access requests, 66% of which are read requests. Thus, if n streams drive an experiment, n synthetic workload generators are used. Using all random-access requests, rather than all sequential-access requests or a mix of both, lowers the storage system's performance capacity. A lower performance capacity allows us to more easily assign stream performance requirements that produce different scenarios, for example, where the performance capacity is sufficient to simultaneously meet the performance requirements of all of the streams in a

workload or where it is sufficient to simultaneously meet the performance requirements of a subset of the streams in a workload. In each experiment driven by a synthetic workload, the simulation time is 60 seconds. Request sizes (in blocks) follow an exponential distribution with 0 as the base and 8 as the mean. As mentioned in Section 4.1.2, we enhanced DiskSim's synthetic workload generators to generate streams that issue request bursts. To generate a stream that does not issue bursts of requests, the synthetic workload generator maintains the number of pending requests in the stream's Shim queue at 50. As described in Section 4.1.4, to generate a stream that issues request bursts, the synthetic workload generator controls the number of the stream's pending requests. (Recall that the bursty request behavior of a stream is defined by three parameters: Burst Interval, Burst Size, and Base Pending Requests.) Section 5.4.5 discusses the real I/O traces used in Experiment 5.

5.4 Experiments

This section discusses the six sets of experiments that we conducted to evaluate the efficacy of ^{2}TL. The commonality among the experiments in a set is the type of workload, i.e., the number of latency- and throughput-bound streams, that drives the experiments. Table 5.2 summarizes the workload that was used in each set of experiments to evaluate the efficacy of ^{2}TL. As indicated in the table associated with each set of experiments, the experiments in a set may differ in terms of stream characteristics, e.g., their performance requirements or burstiness characteristics. As mentioned earlier in this chapter, for each experiment we identify the properties of ^{2}TL that it demonstrates as well as the scenarios that it demonstrates. And, we provide a detailed description of the workload that drives the experiment, including how the performance requirements of each stream were determined, the experimental results, and analysis of the results. Again, each experiment is run first with FCFS, and then with SLAC and with ^{2}TL. We present each stream's performance with FCFS for two reasons. First, it shows that the streams' performance cannot be met trivially without careful scheduling. This ensures that the performance requirements we assigned

to the streams of a workload stress the schedulers' capabilities. Second, as mentioned in Section 4.4, the performance requirement assigned to a stream is based on its performance with FCFS.

5.4.1 Experiment Set 1: Non-bursty Access Characteristics

Table 5.3 provides a high-level description of the three 60-second simulations/experiments, called 1a, 1b, and 1c. As indicated by the table, each of the three experiments is driven by a synthetic workload comprised of one latency-bound stream ($Stream_0$) that does not generate request bursts and two throughput-bound streams ($Stream_1$ and $Stream_2$).

Objectives

This set of experiments has three objectives:

1. Compare the effectiveness of SLAC and ^{2}TL in meeting the performance requirements of a latency-bound stream in a workload: The results of all three experiments, Experiments 1a, 1b, and 1c, demonstrate that when there are no request bursts generated by the latency-bound stream in these workloads, ^{2}TL's proactive scheduling component does not provide any advantage over SLAC's reactive scheduling.

2. Compare the effectiveness of SLAC and ^{2}TL in allocating service to the throughput-bound streams in a workload proportional to their throughput targets: The results of Experiments 1b and 1c demonstrate that SLAC and ^{2}TL are similarly effective in allocating service to the two throughput-bound streams in the workload proportional to their throughput targets. Given sufficient performance capacity, both SLAC and ^{2}TL are able to provide throughput guarantees via proportional service allocation. This is demonstrated by Experiment 1c.

3. Demonstrate ^{2}TL's efficacy in prioritizing a workload's latency requirements over its throughput requirements: The results of Experiments 1a and 1b demonstrate that

when the performance capacity is insufficient to simultaneously meet the performance requirements of all of the streams in a workload, ^{2}TL prioritizes the workload's latency requirements over its throughput requirements. Nonetheless, while prioritizing the latency requirements, ^{2}TL strives to meet the workload's throughput requirements with best effort. The results of Experiment 1b also show that ^{2}TL does not excessively meet the latency requirement of $Stream_0$ and does not overly discount the service allocated to the two throughput-bound streams.

Workloads

As mentioned above, the workload that drove this set of experiments consists of one latency-bound stream, $Stream_0$, and two throughput-bound streams, $Stream_1$ and $Stream_2$. Since the streams do not generate request bursts, the synthetic workload generator maintained a fixed level of 50 pending requests in the I/O hierarchy, i.e., the Shim and the storage system.

As shown in Table 5.3, the performance requirements of the two throughput-bound streams are the same in all three experiments, i.e., 1.02 MB/s and 0.51 MB/s, respectively. In contrast, the latency requirement of $Stream_0$ is different in each experiment, i.e., in Experiment 1a it is $< 100ms, 99\% >$, which represents a high demand; in 1b it is $< 350ms, 99\% >$, representing a medium demand; and in 1c it is $< 500ms, 99\% >$, representing a low demand. These three different latency requirements translate to three different scenarios with respect to the storage system's performance capacity. In Experiment 1a, the storage system has insufficient performance capacity to meet $Stream_0$'s latency requirement and, thus, the throughput requirements of $Stream_1$ and $Stream_2$; in 1b, the system has partially sufficient capacity, i.e., its capacity is sufficient to meet $Stream_0$'s latency requirement but not sufficient to concurrently meet the throughput requirements of $Stream_1$ and $Stream_2$; and in 1c, the performance capacity is sufficient to simultaneously meet the performance requirements of all three streams.

To determine $Stream_0$'s latency requirements for these experiments, we used the results

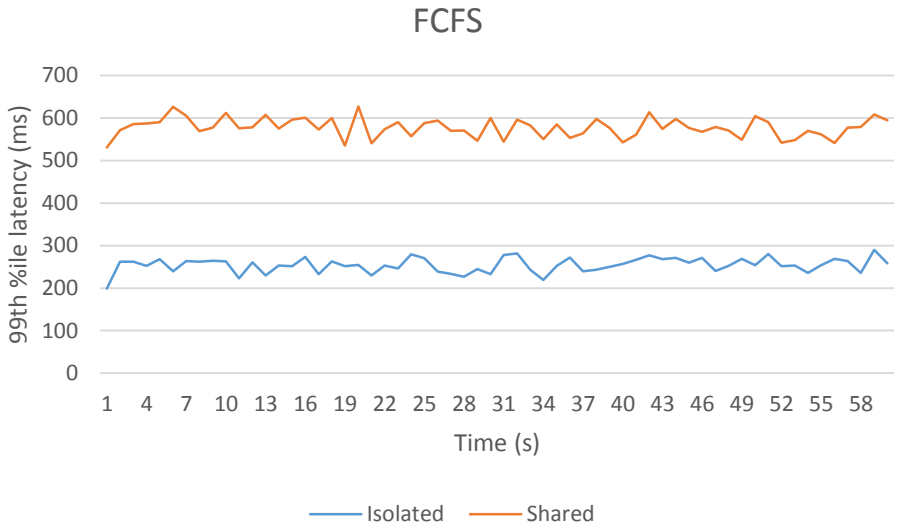


Figure 5.1: Experiment Set 1: $Stream_0$ Latencies with FCFS during Isolated and Shared Access.

obtained by running this experiment with FCFS. We used $Stream_0$'s request latencies with FCFS during (1) isolated access (i.e., when it was not sharing the storage system with the throughput-bound streams), and (2) shared access (i.e., when it was sharing the storage system with the throughput-bound streams). Figure 5.1 shows $Stream_0$'s latencies in these two cases. As shown in the figure, to assign latency requirements that cannot be met trivially, i.e., without careful scheduling of requests (with FCFS), the latency targets for the three experiments must be shorter than $Stream_0$'s latencies with FCFS during shared access, i.e., about 550ms. Since Experiment 1a is meant to quantify scheduler performance when there is insufficient performance capacity to meet $Stream_0$'s latency requirement, we set the latency target to 100ms (high demand), which is impossible for the storage system to achieve even when $Stream_0$ is the only stream that accesses the storage system (isolated access). ($Stream_0$'s latencies with FCFS during isolated access to the storage system are between 200ms and 275ms.) In Experiments 1b and 1c, the performance capacity of the storage system is sufficient to meet $Stream_0$'s latency requirement, either with (1c) or without (1b) meeting the throughput requirements of $Stream_1$ and $Stream_2$. Thus, we

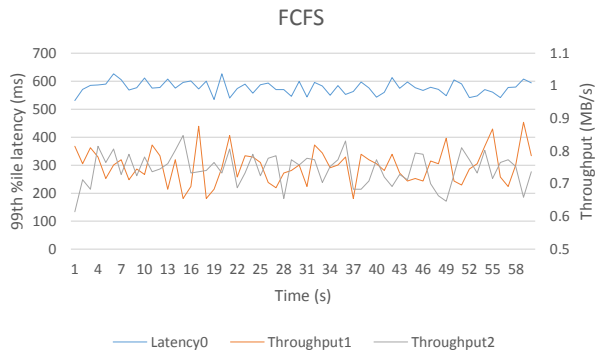
set the latency targets for $Stream_0$ to 500ms (low demand) for Experiment 1c (sufficient performance capacity) and 350ms (medium demand) for Experiment 1b (partially sufficient capacity), respectively. These latency targets are shorter than what $Stream_0$ achieves trivially with FCFS during shared access (i.e., with throughput-bound streams) but longer than what the storage system cannot deliver during isolated access by $Stream_0$.

Experiment 1c presents a scenario where the performance requirements of all the streams in a workload can be met simultaneously. In this case, the sum of the throughput targets of $Stream_1$ and $Stream_2$ must not exceed the performance capacity of the storage system available to the throughput-bound streams after meeting $Stream_0$'s latency requirement. Referring to Figure 5.2a, since $Stream_0$'s latency target (500ms) is very close to its latencies with FCFS (about 550ms), the performance capacity available to the throughput-bound streams should be similar to the total average throughput of $Stream_1$ and $Stream_2$ with FCFS, i.e., 1.53 MB/s. Accordingly, we conservatively estimated that the performance capacity available to $Stream_1$ and $Stream_2$ with SLAC and with ^{2}TL after meeting $Stream_0$'s latency requirement is 1.20 MB/s. Then, we divided this estimate in a 2:1 ratio between $Stream_1$ and $Stream_2$, setting their throughput targets to 0.8 MB/s and 0.4 MB/s, respectively.

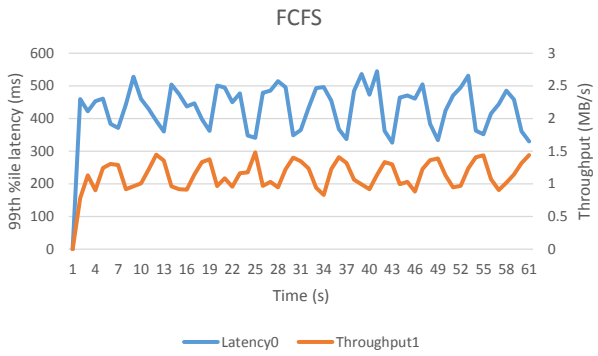
Results

The results of Experiment Set 1 are organized in three parts, those that relate to: (1) the latency requirement of $Stream_0$, (2) the throughput requirements of $Stream_1$ and $Stream_2$, and (3) latency requirement prioritization.

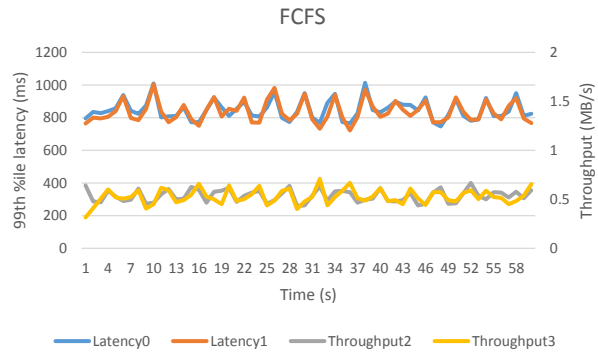
Latency Requirement: Table 5.4 presents $Stream_0$'s latency performance in the three experiments of Experiment Set 1 with the three schedulers. FCFS failed to meet $Stream_0$'s latency requirement in all three experiments with an average meet rate of 0%, which is below $Stream_0$'s percentile rank of 99%. This shows that $Stream_0$'s latency requirement cannot be met without careful scheduling. As was expected, in Experiment 1a (insufficient performance capacity) with all three schedulers, $Stream_0$'s high-demand latency require-



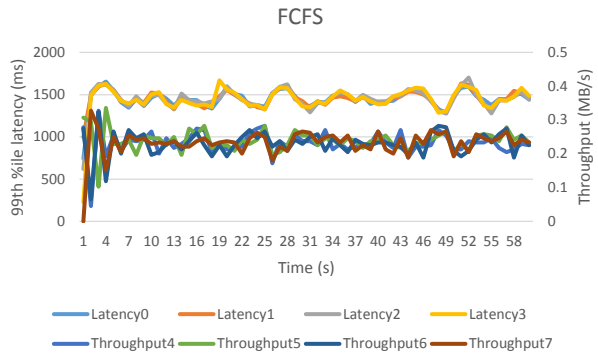
(a) Experiment Set 1: Non-bursty Access Characteristics.



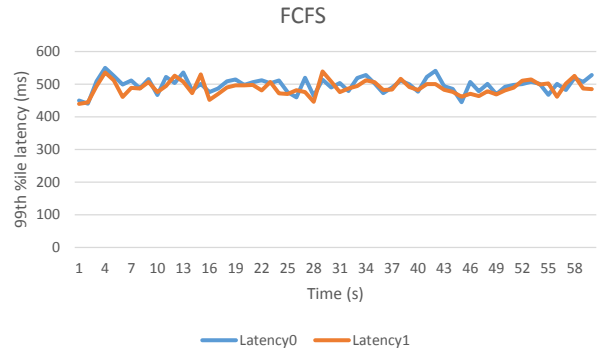
(b) Experiment Set 2: Bursty Access Characteristics.



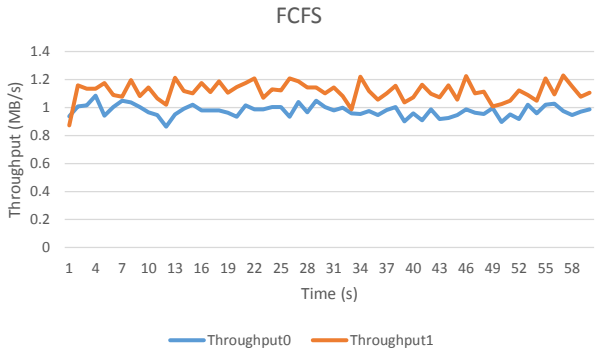
(c) Experiment Set 3: Latency-bound Streams with Different Burst Intervals.



(d) Experiment Set 4: Scalability.

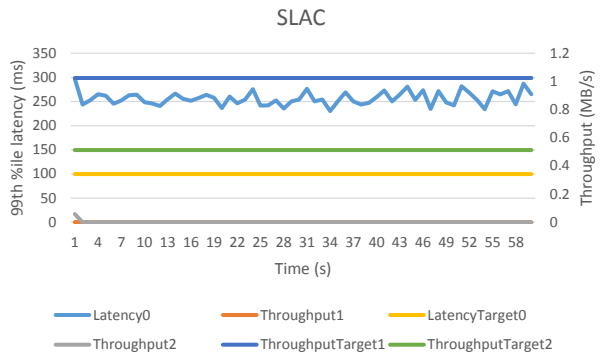


(e) Experiment 6b: Homogeneous Performance Requirements - Latency.

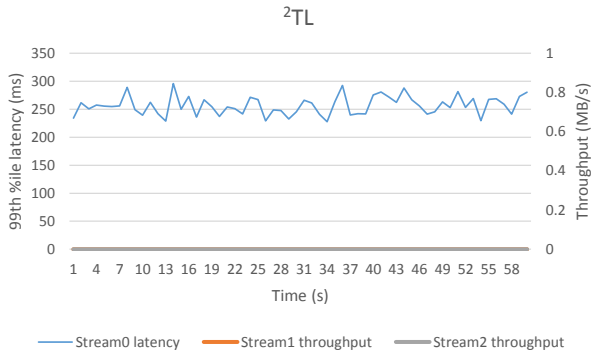


(f) Experiment 6b: Homogeneous Performance Requirements - Throughput.

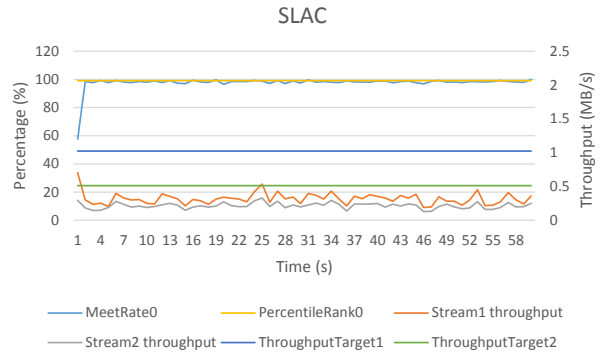
Figure 5.2: Stream Performance with FCFS (Experiment Set 5 is shown separately), where $Latency_i$=$Stream_i$ and $Throughput_j$=$Stream_j$.



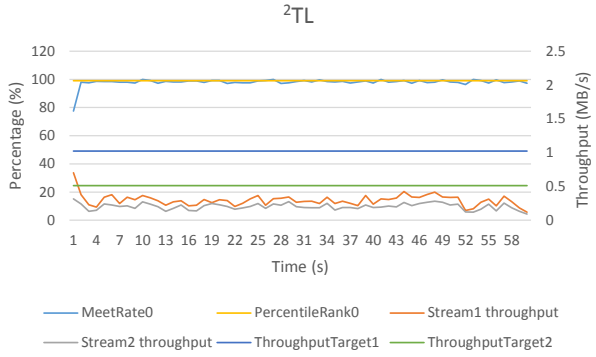
(a) Experiment 1a: SLAC. Performance Requirements of $Stream_0$, $Stream_1$, and $Stream_2$: $< 100ms, 99\% >$, 1.02 and 0.51 MB/s.



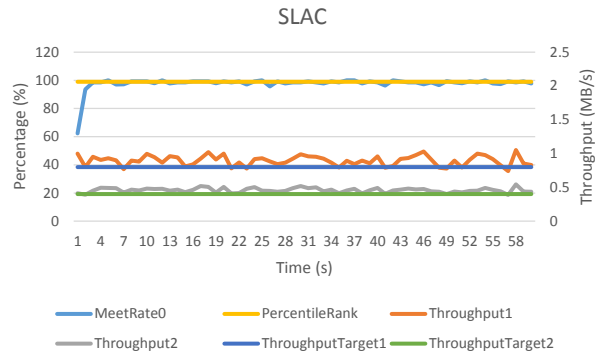
(b) Experiment 1a: ^{2}TL.



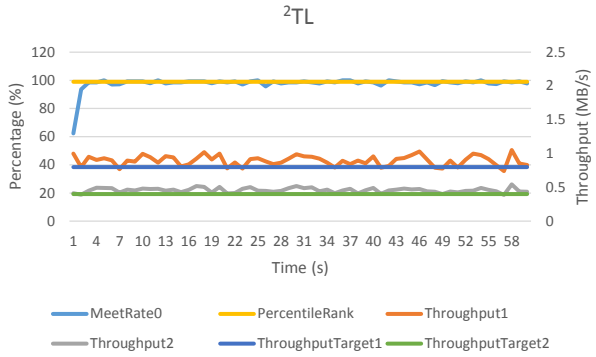
(c) Experiment 1b: SLAC. Performance Requirements of $Stream_0$, $Stream_1$, and $Stream_2$: $< 350ms, 99\% >$, 1.02 and 0.51 MB/s.



(d) Experiment 1b: ^{2}TL.



(e) Experiment 1c: SLAC. Performance Requirements of $Stream_0$, $Stream_1$, and $Stream_2$: $< 500ms, 99\% >$, 0.80 and 0.40 MB/s.



(f) Experiment 1c: ^{2}TL.

Figure 5.3: Experiment Set 1: Stream Performance with SLAC and ^{2}TL.

ment is not met and its average meet rate is 0%. In Experiment 1b (partially sufficient performance capacity), $Stream_0$'s average meet rate with SLAC and ^{2}TL is 98% and 99%, respectively. Thus, $Stream_0$'s latency requirement is met by ^{2}TL but not by SLAC, although SLAC misses it by only 1%. In Experiment 1c (sufficient performance capacity), $Stream_0$'s average meet rate with both SLAC and ^{2}TL is 99%, i.e., both schedulers are able to meet $Stream_0$'s latency requirement. In summary, this set of experiments show that *given a latency-bound stream without bursts and two throughput-bound streams, the performance of SLAC and ^{2}TL are similar in terms of meeting $Stream_0$'s latency requirement. Given sufficient performance capacity, both SLAC and ^{2}TL meet $Stream_0$'s latency requirement. Given partially sufficient or insufficient performance capacity, ^{2}TL's proactive scheduling component does not demonstrate significant advantage over SLAC's reactive scheduling when the latency-bound stream does not generate bursts of requests.*

Throughput Requirements: Table 5.5 presents the throughput performance of $Stream_1$ and $Stream_2$ in the experiments of Experiment Set 1 with FCFS, SLAC and ^{2}TL. In all of the three experiments with FCFS, $Stream_0$'s latency requirement was not met, while $Stream_1$'s throughput requirement (1.02 MB/s in 1a and 1b, 0.80 MB/s in 1c) was not met and $Stream_2$'s throughput requirement (0.51 MB/s in 1a and 1b, 0.40 MB/s in 1c) was met. In all of the three experiments, storage service was equally allocated to $Stream_1$ and $Stream_2$ (0.75 MB/s each) rather than being proportionally allocated based on their throughput targets; the error in proportional sharing was 17%.

In terms of SLAC and ^{2}TL, in Experiment 1a (insufficient performance capacity), since both schedulers prioritize latency requirements over throughput requirements, the latency-bound $Stream_0$ consumed all the performance capacity, and $Stream_1$ and $Stream_2$ received no service allocation. Thus, the workload's throughput requirements were not met.

In Experiments 1b (partially sufficient performance capacity) and 1c (sufficient performance capacity), after meeting (1b for ^{2}TL and 1c for both) or almost meeting (1b for SLAC) the latency requirement of $Stream_0$, the remaining available performance capacity was allocated to $Stream_1$ and $Stream_2$ proportional to their throughput targets with neg-

ligible errors (not larger than 3%). In Experiment 1b, the performance capacity available to the throughput-bound streams was not sufficient to simultaneously meet the throughput requirements of both $Stream_1$ and $Stream_2$, thus, neither throughput requirement (1.02 and 0.51 MB/s, respectively) was met. In contrast, in Experiment 1c, where the performance requirements of the three streams could be met by the performance capacity of the storage system, both SLAC and ^{2}TL met the performance requirements of $Stream_1$ and $Stream_2$ (0.80 and 0.40 MB/s, respectively), along with the latency requirement of $Stream_0$ ($< 400ms, 99\% >$). *Based on the results of Experiments 1b and 1c, we conclude that the effectiveness of SLAC and ^{2}TL is similar in allocating storage service to throughput-bound streams proportional to their throughput targets. Given sufficient performance capacity available to the throughput-bound streams, both schedulers are able to simultaneously meet the throughput-bound streams' performance requirements through proportional sharing.*

Latency Requirement Prioritization: With FCFS, $Stream_0$'s latency requirements were not met in the three experiments. This is because FCFS is unable to discount service allocated to the throughput-bound streams, $Stream_1$ and $Stream_2$, in order to meet the latency requirement of $Stream_0$. In Experiment 1a (insufficient performance capacity), ^{2}TL gave priority to $Stream_0$'s latency requirement over the throughput requirements of $Stream_1$ and $Stream_2$. Therefore, $Stream_0$ consumed all of the storage service. In Experiment 1b (partially sufficient performance capacity), ^{2}TL gave priority to $Stream_0$'s latency requirement, and it allocated sufficient storage service to $Stream_0$, while allocating the remaining storage service to $Stream_1$ and $Stream_2$. Accordingly, $Stream_0$'s average meet rate was 99%, which is the same as its percentile rank, i.e., ^{2}TL met $Stream_0$'s latency requirement. In addition, ^{2}TL accomplished this without excessive allocation of service to $Stream_0$ and provided $Stream_1$ and $Stream_2$ with as much service as was possible. In summary, based on the results of these experiments, *when the performance capacity is insufficient to simultaneously meet the performance requirements of all of the streams in a workload, ^{2}TL is able to appropriately prioritize the workload's latency requirements, while striving to meet its throughput requirements with best effort.*

5.4.2 Experiment Set 2: Bursty Access Characteristics

Table 5.6 provides a high-level description of the six 60-second simulations/experiments, in this set, called 2a, 2b, 2c, 2d, 2e, and 2f. As indicated, all six experiments are driven by a synthetic workload comprised of one latency-bound stream ($Stream_0$) that generates request bursts and one throughput-bound stream ($Stream_1$).

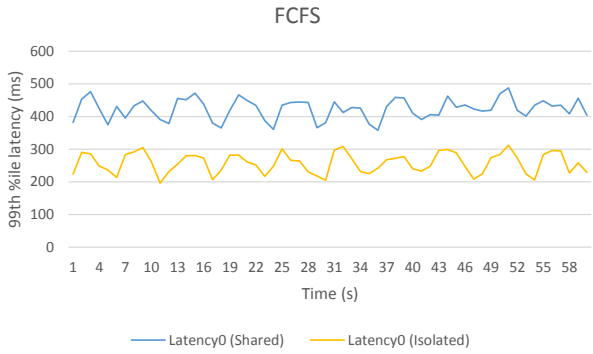
Objectives

This set of experiments has three objectives:

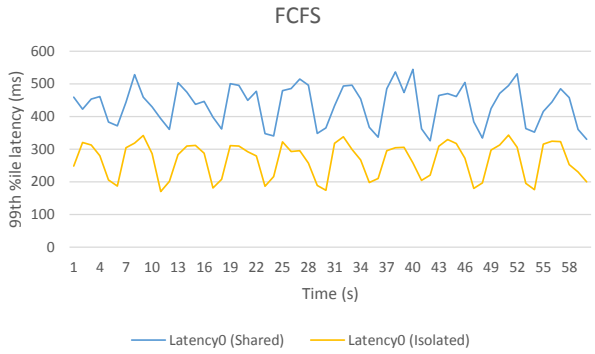
1. Demonstrate ^{2}TL's efficacy in meeting the performance requirement of the latency-bound stream in a workload that issues request bursts: The results of this set of experiments, which employ a range of burst parameters (see Table 5.6) demonstrate that FCFS and SLAC cannot meet the latency requirement of $Stream_0$. This is because (1) the latency requirement cannot be met without careful scheduling, i.e., with FCFS; and (2) SLAC's reactive scheduling fails to meet the latency requirement when bursts arrive. In addition, these experiments demonstrate that, because of its proactive scheduling component, ^{2}TL is able to meet $Stream_0$'s latency requirement regardless of its bursty behavior.

2. Demonstrate that ^{2}TL is able to prioritize a workload's latency requirements over its throughput requirements when necessary, even when the latency-bound stream in the workload issues request bursts: ^{2}TL dynamically decides when to prioritize $Stream_0$'s latency requirement over $Stream_1$'s throughput requirement. When $Stream_0$ is issuing a request burst, if the storage system's performance capacity is insufficient to simultaneously meet $Stream_0$'s latency requirement and $Stream_1$'s throughput requirement, ^{2}TL discounts service allocated to $Stream_1$ and allocates extra service to $Stream_0$. Otherwise, ^{2}TL does not prioritize $Stream_0$'s latency requirement.

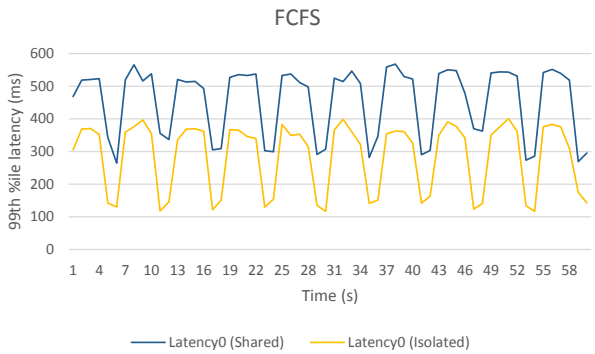
3. Demonstrate that, given sufficient performance capacity, ^{2}TL is able to simultaneously meet both the latency and throughput requirements of the streams in a work-



(a) Burst Size= 20, Base Pending Requests= 40



(b) Burst Size= 40, Base Pending Requests= 30



(c) Burst Size= 80, Base Pending Requests= 20

Figure 5.4: Experiment 2b, 2c, 2d: $Stream_0$ Latencies with FCFS during Isolated and Shared Access (Burst interval= 6s)

load: ^{2}TL prioritizes the latency requirement of $Stream_0$ over the throughput requirement of $Stream_1$. After meeting $Stream_0$'s latency requirement, it allocates the remaining storage service to $Stream_1$. When there is sufficient performance capacity ^{2}TL meets the latency and throughput requirements of both streams.

Workload

As mentioned above, the workload of all five experiments in this set is comprised of one latency-bound stream, $Stream_0$, and one throughput-bound stream, $Stream_1$. To demonstrate the benefit of ^{2}TL's proactive scheduling component over a range of burst parameters,

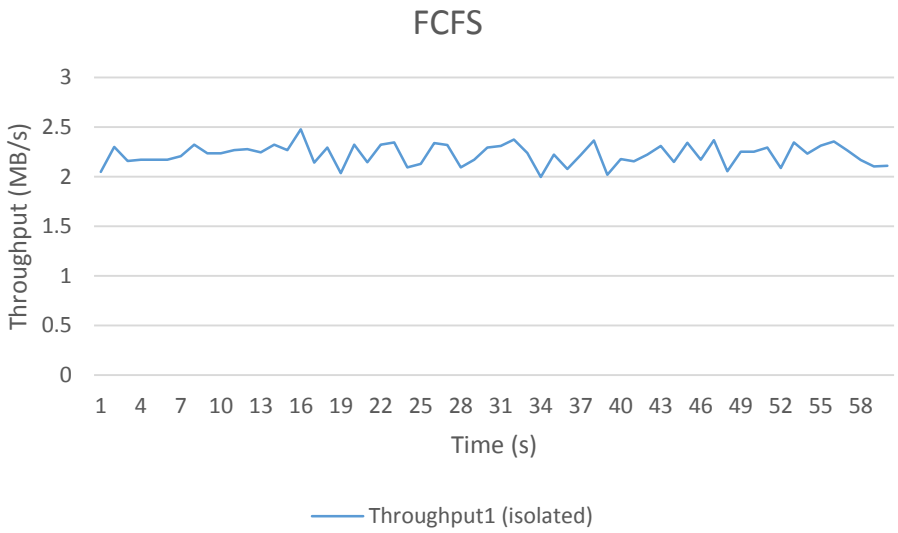


Figure 5.5: Experiment Set 2: $Stream_1$ Throughput with FCFS during Isolated Access. ($Stream_1$ does not issue bursts and has 50 pending requires in the I/O hierarchy in all of the simulations in Experiment Set 2.)

the five experiments use different sets of burst parameters (Burst Interval, Burst Size, and Base Pending Requests) to describe $Stream_0$'s bursty behavior in each experiment. (These parameters were introduced in Section 4.1.4. Burst Interval and Burst Size translate to the burstiness and burst intensity of the stream.) As shown in Table 5.6, when $Stream_0$'s number of Base Pending Requests is at the "lower" level, it is at 10 (Experiment 2d), 40 (2b), or 30 (2a, 2c, 2e, and 2f). Since the throughput-bound $Stream_1$ does not issue request bursts, it maintains a constant number of pending requests in the I/O hierarchy (the Shim and storage system combined), i.e., 50.

The performance requirements of the two streams that drive the six experiments in this set are based on the guideline presented in Section 4.4. Because ^{2}TL prioritizes latency requirements, $Stream_0$'s latency requirement is deduced before $Stream_1$'s throughput requirement. To evaluate ^{2}TL's efficacy, the latency requirement of $Stream_0$ needs to be short enough that it cannot be met with FCFS, but it must be longer than $Stream_0$'s latencies during isolated access. Figure 5.4 presents $Stream_0$'s latencies with FCFS during

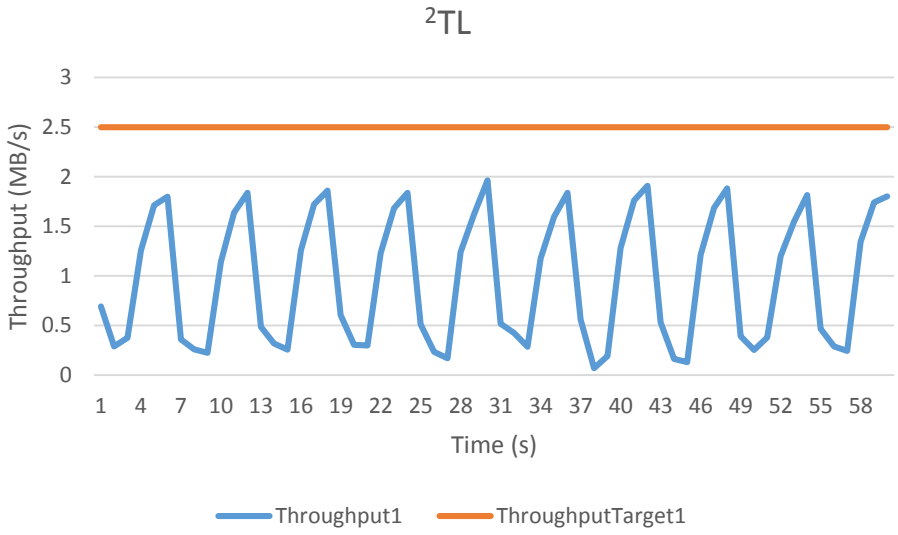


Figure 5.6: Experiment 2d: $Stream_1$ Throughput with ^{2}TL. Performance Requirements of $Stream_0$ and $Stream_1$: $< 400ms, 99\% >$, and 2.5 MB/s.

isolated and shared access. Among the three values of Burst Size used in this set of experiments, i.e., 20, 40, and 80, $Stream_0$ has the longest latencies (approximately 400ms) during isolated access when Burst Size is 80. Therefore, we set $Stream_0$'s latency requirement to $< 400ms, 99\% >$ in the five experiments. This is the shortest latency requirement that is potentially achievable by the storage system with the three burst sizes. As shown in Figure 5.4, with the three different values of Burst Size, $Stream_0$'s request latencies during shared access with FCFS are mostly longer than 400ms. $Stream_0$'s average meet rates during shared access with the three values of Burst Size, i.e., 20, 40, and 80, are 95%, 82%, and 47%, respectively, which are all below $Stream_0$'s percentile rank (99%). This shows that $Stream_0$'s latency requirement cannot be met with FCFS, i.e., without careful request scheduling.

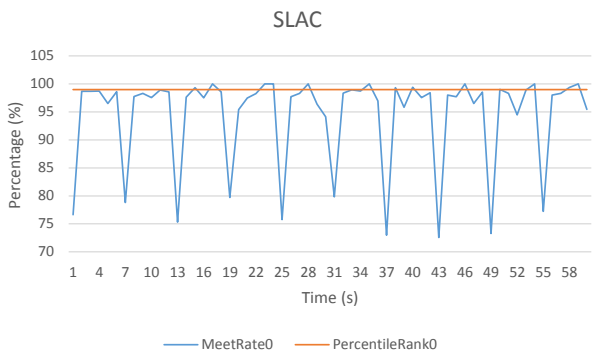
In order to assign $Stream_1$'s throughput requirement, we first need to quantify the amount of performance capacity available to $Stream_1$ after meeting $Stream_0$'s latency requirement. Once we obtain the performance capacity, we can assign $Stream_1$'s throughput requirement based on the scenarios of the experiments (e.g. insufficient performance capac-

ity to meet throughput requirements). Figure 5.5 shows that $Stream_1$'s average throughput with FCFS during isolated access to the storage system does not exceed 2.5 MB/s, which is the highest $Stream_1$ throughput achievable on the storage system. To obtain the performance capacity available to $Stream_1$ after meeting $Stream_0$'s latency requirement, we ran the six experiments described in Table 5.6 with ^{2}TL with $< 400ms, 99\% >$ and 2.5 MB/s as the performance requirements of $Stream_0$'s and $Stream_1$, respectively. Figure 5.6 shows that, in the burst parameters in Experiment 2d, after meeting $Stream_0$'s latency requirement, the performance capacity available to $Stream_1$ is approximately 1.00 MB/s. Note that the performance capacity available to $Stream_1$ in the other five experiments is also approximately 1.00 MB/s. Thus, in order to demonstrate that ^{2}TL is able to prioritize latency guarantees when the performance capacity is insufficient in Experiments 2a to 2e, we set $Stream_1$'s throughput requirement in the five experiments to 1.28 MB/s (2.5 blocks/ms), which exceeds the performance capacity available to $Stream_1$, after $Stream_0$'s latency requirement has been met (if it can be). In contrast, in order to demonstrate that, given sufficient performance capacity, ^{2}TL is able to simultaneously meet the latency requirement of the bursty stream and the performance requirement of the throughput-bound stream in Experiment 2f, we set $Stream_1$'s throughput requirement 0.90 MB/s (1.76 blocks/ms), which is below the performance capacity available to $Stream_1$, after $Stream_0$'s latency requirement has been met (if it can be).

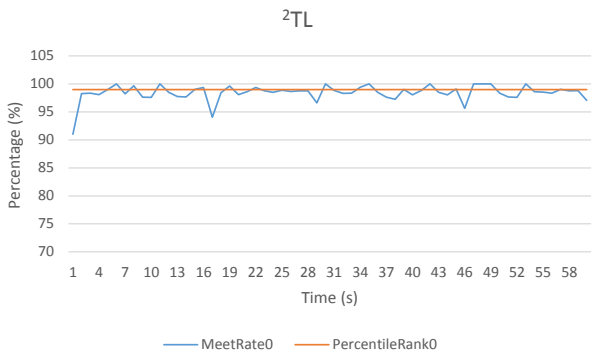
Results

The results of Experiment Set 2 are organized in three parts, those that relate to: (1) the latency requirement of $Stream_0$, (2) latency requirement prioritization, and (3) performance guarantees with sufficient performance capacity.

Latency Guarantee: Table 5.7 presents the average meet rate of $Stream_0$ in the Experiments 2a to 2e. As shown, $Stream_0$'s latency requirement was never met with FCFS; its meet rate ranges from 46% to 95%, all of which are below $Stream_0$'s percentile rank of 99%. This shows that $Stream_0$'s latency requirement cannot be met without careful



(a) Experiment 2d: Meet Rates with SLAC.



(b) Experiment 2d: Meet Rates with ^{2}TL.

Figure 5.7: Experiment 2d: $Stream_0$ Meet Rates with SLAC and ^{2}TL. Percentile Rank of $Stream_0$: 99%.

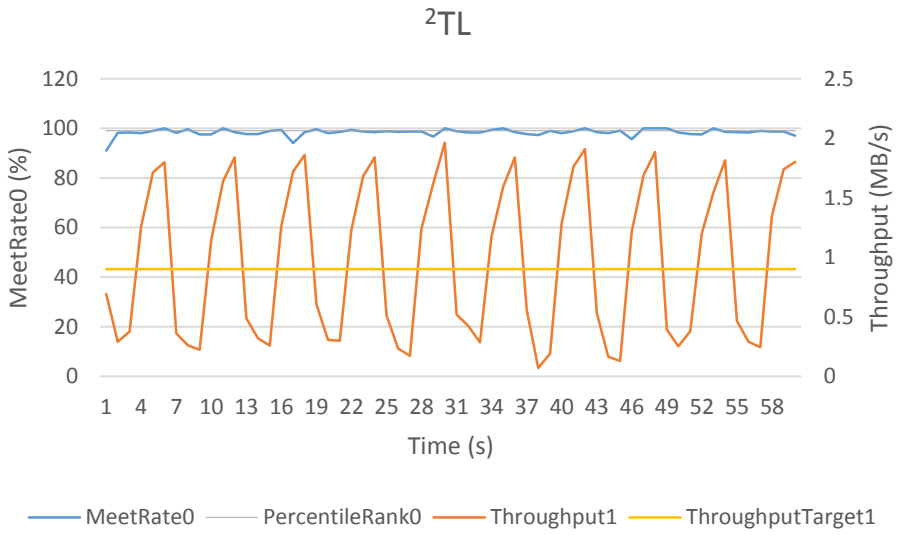


Figure 5.8: Experiment 2f: Stream Performance with FCFS. $Stream_0$: Burst Interval= 6s, Burst Size= 80, and Base Pending Request= 10. Performance Requirements of $Stream_0$, and $Stream_1$: $< 400ms, 99\% >$, and 0.90 MB/s

scheduling.

In each of Experiments 2a to 2e with SLAC, $Stream_0$'s average meet rate is smaller than its percentile rank (99%), with 98% being the closest. In contrast, in each experiment with ^{2}TL, $Stream_0$'s average meet rate is equal to its latency requirement's percentile rank. Therefore, ^{2}TL is able to meet $Stream_0$'s latency requirement, while SLAC cannot. These experiments demonstrate that the benefit of ^{2}TL's proactive scheduling component in meeting latency requirements is more pronounced as the Burst Size increases (larger bursts) and as the Burst Interval decreases (more frequent bursts). For example, for the largest (smallest) Burst Size, i.e., 80 (20), $Stream_0$ has a meet rate of only 46% (95%) with FCFS, while with SLAC it is 95% (98%), and with ^{2}TL it is 99%. Figure 5.7 compares $Stream_0$'s meet rates with SLAC and ^{2}TL in Experiment 2d, where Burst Interval was 6 seconds, Burst Size was 80, and Base Pending Requests was 20. As shown in the figure, with SLAC, when a request burst arrives, the meet rates drop sharply. In contrast, with ^{2}TL the meet rates are perturbed only slightly. With SLAC and ^{2}TL, up to 26.44% and 5.05% of $Stream_0$ requests miss the latency target, respectively. ^{2}TL is more effective in meeting $Stream_0$'s latency requirement because ^{2}TL proactively increases service allocation to $Stream_0$ when a request burst arrives in order to avoid latency requirement violations. In summary, because of ^{2}TL's proactive scheduling component, *given a workload consisting of a latency-bound stream that issues request bursts and a throughput-bound stream,* ^{2}*TL is more effective than SLAC in meeting the performance requirement of the latency-bound stream.*

Latency Guarantee Prioritization: Table 5.8 presents the average throughput achieved by $Stream_1$ in Experiments 2a to 2e. $Stream_1$'s throughput requirement (1.28 MB/s) was not met in any of the experiments with any of the schedulers. This is because the performance capacity is insufficient to simultaneously meet the performance requirements of both streams in the workload. Both SLAC and ^{2}TL prioritize $Stream_0$ to first meet its latency requirement, and then allocate the remainder of the performance capacity to $Stream_1$. In all of the five experiments, $Stream_1$'s maximum throughput deficiency with

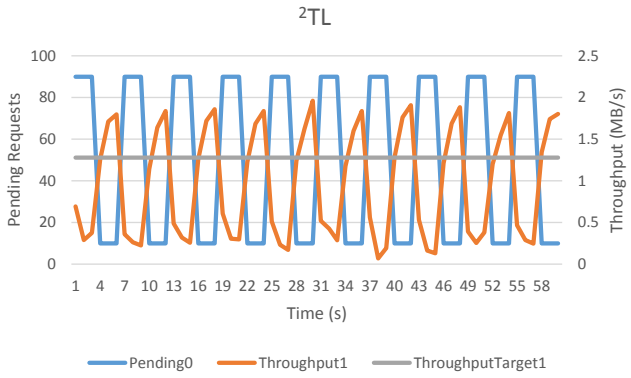


Figure 5.9: Experiment 2d: ^{2}TL Dynamically Gave Priority to Latency-bound Streams, i.e., It gave priority to $Stream_0$ over $Stream_1$. Throughput Requirements of $Stream_1$: 1.28 MB/s

SLAC and ^{2}TL is 20% and 24%, respectively. $Stream_1$'s throughput performance in all the five experiments is lower with ^{2}TL than it is with FCFS or SLAC because ^{2}TL is more prompt in prioritizing the scheduling of the requests of $Stream_0$ to avoid latency requirement violations. In addition, $Stream_0$'s average meet rates in the experiments with ^{2}TL do not exceed its percentile rank of 99%. This implies that ^{2}TL is able to appropriately prioritize the scheduling of the requests of $Stream_0$ and allocate as much storage service as is possible to $Stream_1$. Figure 5.9 presents $Stream_0$'s number of pending requests (blue) and $Stream_1$'s throughput in Experiment 2d. This figure shows that ^{2}TL intentionally discounted service allocated to $Stream_1$, while prioritizing service to $Stream_0$ only when $Stream_0$'s pending request level was high. Otherwise, ^{2}TL allocated sufficient service to $Stream_1$ to meet its throughput requirement (grey). This demonstrates that ^{2}TL is able to dynamically determine when to prioritize a latency-bound stream in a workload. In summary, *given a workload consisting of a latency-bound stream that issues request bursts and a throughput-bound stream, ^{2}TL is able to appropriately prioritize service to the latency-bound stream, while striving to meet the performance requirement of the throughput-bound stream with best effort.*

Performance Guarantees with Sufficient Performance Capacity: In Experiments 2a to 2e, the performance requirements were set to exceed the storage system's performance capacity. The purpose of doing this was to demonstrate that ^{2}TL is able to prioritize a workload's latency requirements over its throughput requirements when the storage system does not have sufficient performance capacity to simultaneously meet both. Here we present the results of Experiment 2f, where the performance capacity is sufficient to simultaneously meet the performance requirements of the latency-bound stream and the throughput-bound stream in a workload. The purpose of this experiment is to demonstrate that, given sufficient performance capacity, ^{2}TL is able to simultaneously meet the latency requirement of the bursty stream and the performance requirement of the throughput-bound stream.

In this experiment, $Stream_0$ has the same latency requirement, $< 400ms, 99\% >$, as in the five other experiments. However, we set $Stream_1$'s throughput requirement to 0.90 MB/s so that there is sufficient performance capacity to meet the performance requirements of both streams in the workload. As mentioned previously, the performance capacity available to $Stream_1$ after meeting $Stream_0$'s latency requirement is approximately 1 MB/s, which is sufficient to meet $Stream_1$'s throughput requirement. As in Experiment 2d, for $Stream_0$ Burst Interval is 6 seconds, Burst Size is 80, and Base Pending Requests is 10. Figure 5.8 presents the streams' performance. Given sufficient performance capacity, both streams' performance requirements were simultaneously met by ^{2}TL. $Stream_0$'s average meet rate with ^{2}TL was 99%, matching its percentile rank (99%), while $Stream_1$'s average throughput was 0.97 MB/s, which is above its throughput target (0.90 MB/s). In summary, *given a workload consisting of a latency-bound stream that issues request bursts and a throughput-bound stream, and a storage system with sufficient performance capacity, ^{2}TL is able to simultaneously meet the performance requirements of both streams. In addition, in this case, ^{2}TL does not overly meet $Stream_0$'s latency requirement; its average meet rate is 99%, which is the same as its percentile rank. ^{2}TL is able to meet $Stream_0$'s latency requirement, while allocating as much storage service as possible to $Stream_1$ to meet its throughput requirement.*

Table 5.1: Five Properties of ^{2}TL Demonstrated by Experiments.

Experiment	**^{2}TL Properties**				
	P1	**P2**	**P3**	**P4**	**P5**
1a		X			
1b		X		X	
1c	X			X	
2a		X			X
2b		X			X
2c		X			X
2d		X			X
2e		X			X
2f	X				
3a		X		X	
3b		X	X		
4a		X		X	
5a		X			
6b			X		
6e	X			X	

Table 5.2: High-Level Description of Experiments.

Experiment Set	**Workload**			
	Type	**No. of Streams**	**Description**	
			No. of Latency-bound Streams	**No. of Throughput-bound Streams**
1	synthetic	3	1 w./o bursts	2
2	synthetic	2	1 w./ bursts	1
3	synthetic	4	2 w./ bursts with different burst intervals	2
4	synthetic	8	4 w./ bursts with different burst parameters	4
5	real	2	1 w./ bursts	1
6	synthetic	2	2: 1 w./ burst, 1 w./o burst (6b)	0 (6b)
			0 (6e)	2 (6e)

Table 5.3: Experiment Set 1: Three 60-second Simulations driven by a Synthetic Workload of 3 Streams : 1 Latency-bound without Bursts and 2 Throughput-bound.

Experiment	**Performance Capacity**	**Performance Requirements**		
		Latency-bound	**Throughput-bound**	
		Stream$_0$	**Stream$_1$**	**Stream$_2$**
1a	Insufficient to meet the latency requirement	$< 100ms, 99\% >$	1.02 MB/s (2 blocks/ms)	0.51 MB/s (1 block/ms)
1b	Sufficient to meet the latency requirement but not all performance requirements	$< 350ms, 99\% >$		
1c	Sufficient to meet all performance requirements	$< 500ms, 99\% >$	0.80 MB/s (1.56 blocks/ms)	0.40 MB/s (0.78 blocks/ms)

Table 5.4: Experiment Set 1: $Stream_0$'s Average Meet Rates.

Experiment	**Scheduler**	**AvgMeetRate$_0$**	**Requirement (99% meet rate) Met?**
1a	FCFS	0%	No
	SLAC	0%	No
	^{2}TL	0%	No
1b	FCFS	0%	No
	SLAC	98%	No
	^{2}TL	99%	Yes
1c	FCFS	0%	No
	SLAC	99%	Yes
	^{2}TL	99%	Yes

Table 5.5: Experiment Set 1: Throughput Performance of $Stream_1$ and $Stream_2$.

Experiment	**Scheduler**	**AvgThroughput$_i$ (MB/s)**		**ActualRatio$_i$**		**Error$_i$ (%)**	
		Stream$_1$	**Stream$_2$**	**Stream$_1$**	**Stream$_2$**	**Stream$_1$**	**Stream$_2$**
1a	FCFS	0.75	0.75	0.5	0.5	-17%	17%
	SLAC	0	0	N/A	N/A	N/A	N/A
	^{2}TL	0	0	N/A	N/A	N/A	N/A
1b	FCFS	0.75	0.75	0.5	0.5	-17%	17%
	SLAC	0.56	0.31	0.64	0.36	-3%	3%
	^{2}TL	0.52	0.29	0.64	0.36	-3%	3%
1c	FCFS	0.75	0.75	0.5	0.5	-17%	17%
	SLAC	0.91	0.42	0.64	0.36	-3%	3%
	^{2}TL	0.89	0.46	0.65	0.35	-2%	2%

Table 5.6: Experiment Set 2: Six 60-second Simulations driven by a Synthetic Workload of 2 Streams: 1 Latency-bound with Bursts and 1 Throughput-bound.

Exper.	**Performance Capacity**	**Performance Requirements**		**Burst Parameters**		
		Latency-bound	**Throughput-bound**	**Burst Interval (s)**	**Burst Size**	**Base Pending Requests**
		Stream$_0$	**Stream$_1$**			
2a	Sufficient to meet only latency requirement	$< 400ms, 99\% >$	1.28 MB/s	2	40	30
2b				6	20	40
2c				6	40	30
2d				6	80	10
2e				10	40	30
2f	Sufficient to meet both of the latency and throughput requirements		0.90 MB/s	6	80	10

Table 5.7: Experiment 2a to 2e: $Stream_0$'s Average Meet Rates. Numbers in Red Indicate that $Stream_0$ Latency Requirement Not Met.

		Burst Size								
		20			**40**			**80**		
		FCFS	**SLAC**	**^{2}TL**	**FCFS**	**SLAC**	**^{2}TL**	**FCFS**	**SLAC**	**^{2}TL**
Burst Interval	**2s**				82%	95%	99%			
	6s	95%	98%	99%	82%	97%	99%	46%	95%	99%
	10s				84%	98%	99%			

Table 5.8: Experiments 2a to 2e: $Stream_1$'s Average Throughput (MB/s). Throughput Deficiencies of $Stream_1$ are Highlighted in Red.

	Burst Size									
		20			**40**			**80**		
		FCFS	**SLAC**	**^{2}TL**	**FCFS**	**SLAC**	**^{2}TL**	**FCFS**	**SLAC**	**^{2}TL**
Burst Interval	**2s**				1.11	1.05	0.98			
					13%	18%	23%			
	6s	1.13	1.06	1.04	1.15	1.04	1.02	1.21	1.02	0.97
		12%	17%	19%	10%	19%	20%	5%	20%	24%
	10s				1.15	1.04	1.02			
					10%	19%	20%			

5.4.3 Experiment Set 3: Prioritization of Latency-Bound Streams

Table 5.9 provides a high-level description of the two 60-second simulations/experiments, in this set, called 3a and 3b. As indicated in the table, the two experiments were driven by a synthetic workload comprised of two latency-bound streams that generated request bursts ($Stream_0$ and $Stream_1$, where the former had the higher priority) and two throughput-bound streams ($Stream_2$ and $Stream_3$).

Table 5.9: Experiment Set 3: Two 60-second Simulations driven by a Synthetic Workload of 4 Streams: 2 Latency-bound with Bursts and 2 Throughput-bound.

Experiment	**Performance Capacity**	**Performance Requirements**			
		Latency-bound		**Throughput-bound**	
		$Stream_0$ Priority 1 (highest)	**$Stream_1$ Priority 2**	**$Stream_2$**	**$Stream_3$**
3a	Sufficient to meet only the latency requirements	$< 750ms, 99\% >$		1.02 MB/s (2 blk/ms)	0.51 MB/s (1 blk/ms)
3b	Sufficient to meet only 1 latency requirement	$< 550ms, 99\% >$			
	Burst Interval (s)	2	4	N/A	
	Burst Size	50			
	Base Pending Requests	25			

Objectives

This set of experiments has three objectives.

1. Demonstrate ^{2}TL's efficacy in meeting the performance requirements of the latency-bound streams in a workload, both of which issue request bursts - show this when there are two latency-bound streams in a workload: The results of this set of experiments demonstrate that FCFS and SLAC cannot meet the performance requirements of the latency-bound streams, $Stream_0$ and $Stream_1$. This is because: (1) the latency requirements cannot be met without careful scheduling, i.e., with FCFS; and (2) SLAC's reactive scheduling fails to meet the latency requirements when bursts arrive. In contrast, these experiments provide evidence that ^{2}TL is able to meet the latency requirements of $Stream_0$ and $Stream_1$. This is because of its proactive scheduling of the requests of the latency-bound streams in a workload.

2. Compare the effectiveness of SLAC and ^{2}TL in prioritizing the scheduling of the requests of latency-bound streams over those of throughput-bound streams: The results of Experiments 3a and 3b demonstrate that, when the storage system does not have sufficient performance capacity to simultaneously meet the performance requirements of all of the streams in a workload, ^{2}TL prioritizes latency requirements over throughput requirements. Because of ^{2}TL's proactive scheduling of the requests of latency-bound streams, it is able do this more promptly than SLAC does. In addition, ^{2}TL does not overly prioritize latency requirements and strives to meet throughput requirements with best effort.

3. Compare the effectiveness of SLAC and ^{2}TL in prioritizing the servicing of latency-bound streams with different priorities: The results of Experiment 3b demonstrate that, when the storage system does not have sufficient performance capacity to simultaneously meet the performance requirements of the two latency-bound streams in a workload, ^{2}TL is more effective in prioritizing the scheduling of the requests

of the latency-bound stream with the higher priority. In addition, when prioritizing these requests, ^{2}TL does not overly discount service allocated to the lower-priority latency-bound stream, but rather strives to meet its latency requirement with best effort.

4. Compare the effectiveness of SLAC and ^{2}TL in allocating service to throughput-bound streams proportional to their throughput targets: The results of both experiments demonstrate that SLAC and ^{2}TL are similarly effective in allocating service to the two throughput-bound streams proportional to their throughput targets.

Workload

As mentioned above, the workload of the two experiments in the set is comprised of two latency-bound streams, $Stream_0$ and $Stream_1$, and two throughput-bound streams, $Stream_2$ and $Stream_3$. To further demonstrate the benefit of ^{2}TL's proactive scheduling of the requests of latency-bound streams, the latency-bound streams issue request bursts with two different size burst intervals. As shown in Table 5.9, in both experiments $Stream_0$ and $Stream_1$ have the same number of base pending requests (25) and the same burst size (50), while $Stream_0$'s burst interval is 2 seconds and $Stream_1$'s is 4 seconds. Since the throughput-bound streams, $Stream_2$ and $Stream_3$, do not issue request bursts, DiskSim maintains a constant number of pending requests in the I/O hierarchy, which includes the Shim and the storage system, for each, i.e., 50.

The performance requirements of the four streams that drove these two experiments were determined using the guidelines presented in Section 4.4. Because ^{2}TL prioritizes latency requirements over throughput requirements, the latency requirements of $Stream_0$ and $Stream_1$ were deduced before the throughput requirements of $Stream_2$ and $Stream_3$. In Experiment 3a, to evaluate ^{2}TL's efficacy in providing latency guarantees, the latency requirements of $Stream_0$ and $Stream_1$ need to be short enough that they cannot be met by FCFS, but they must be longer than the request latencies the streams experienced during

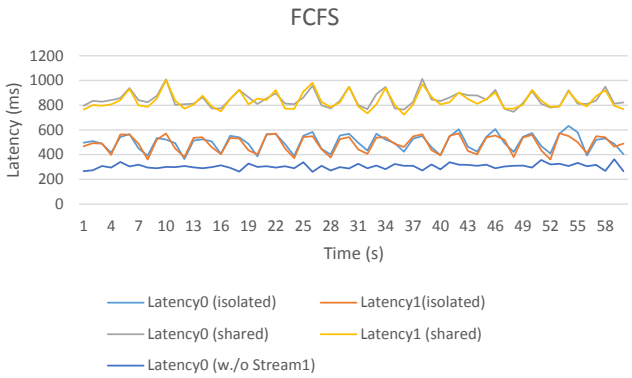


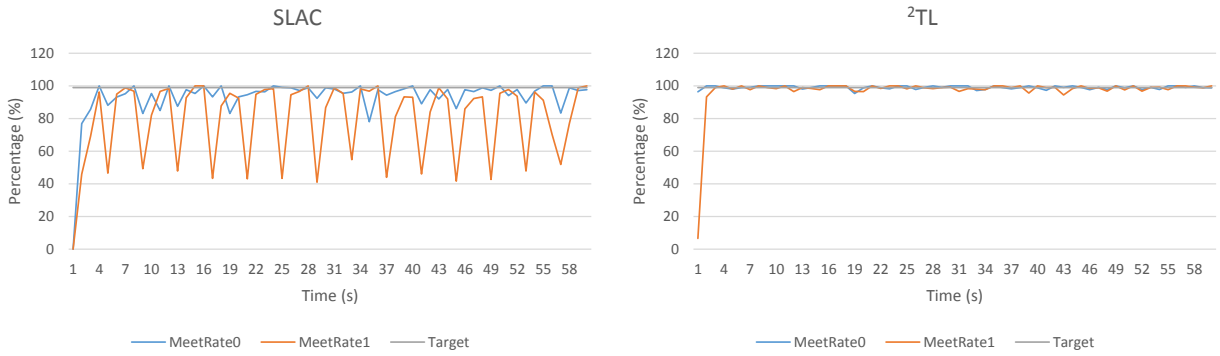
Figure 5.10: Experiment Set 3: Latency-Bound Stream Latencies with FCFS.

isolated access (i.e., $Stream_2$ and $Stream_3$ only) with FCFS. Figure 5.10 presents the latencies experienced by $Stream_0$ and $Stream_1$ in Experiments 3a and 3b with FCFS during both isolated and shared (i.e., 4 streams) access to the storage system. Given this data, we set the latency requirement of each to $< 750ms, 99\% >$, in Experiment 3a. The latency target, 750ms, is shorter than the latencies of either stream during shared access (i.e., with throughput-bound streams, $Stream_2$ and $Stream_3$) with FCFS, but longer than the latencies of either during isolated access. The average meet rates of $Stream_0$ and $Stream_1$ during shared access with FCFS in Experiment 3a were 60% and 65%, respectively, which are below the streams' percentile rank of 99%. This shows that the latency requirements of $Stream_0$ and $Stream_1$ cannot be met with FCFS, i.e., without careful request scheduling.

Experiment 3a is meant to present a scenario where the performance capacity is sufficient to meet the workload's latency requirements but not its throughput requirements. Therefore, to determine the throughput requirements of $Stream_2$ and $Stream_3$, we first estimate the upper bound of the performance capacity available to these two throughput-bound streams after ^{2}TL meets the workload's latency requirements and then divide this among the two streams. During shared access (by the four streams) with FCFS, the total average throughput of $Stream_2$ and $Stream_3$ is about 1.07 MB/s (2.09 blocks/ms). Since with SLAC and ^{2}TL less service is allocated to the throughput-bound streams, in order to provide latency guarantees, the performance capacity available to $Stream_2$ and $Stream_3$

after meeting the latency requirements must be less than 1.07 MB/s (2.09 blocks/ms). Now, based on the ratio of the streams' throughput targets (2-to-1), we divide this performance capacity proportionally between the two streams, obtaining their throughput targets. Since it is not feasible to divide 2.09 blocks/ms between two streams, we set the total throughput target to 3.51 MB/s (3 blocks/ms). Since we want to evaluate the schedulers' efficacies in allocating service to the throughput-bound streams proportional to their throughput targets, we impose a 2-to-1 ratio to the throughput targets and, thus, the throughput requirements of $Stream_2$ and $Stream_3$ are 1.02 MB/s (2 blocks/ms) and 0.51 MB/s (1 block/ms), respectively.

Experiment 3b is meant to present a scenario where the performance capacity is insufficient to simultaneously meet the latency requirements of both $Stream_0$ and $Stream_1$. If we assume that these two streams have the same latency requirement, the latency target must be short enough that the storage system cannot simultaneously meet this requirement for both streams. However, the latency target cannot be so short that the storage system cannot meet even one of the two latency requirements. Figure 5.10 shows that, during isolated access (i.e., without throughput-bound streams) with FCFS, the latencies of $Stream_0$ and $Stream_1$ are not shorter than 360ms. It implies that the storage system does not have sufficient performance capacity to simultaneously meet the latency requirements of $Stream_0$ and $Stream_1$ if their latency targets are shorter than 360ms. Therefore, to create the scenario in this experiment, we set both the latency requirements of $Stream_0$ and $Stream_1$ to $< 350ms, 99\% >$ to ensure that they cannot be met simultaneously. But, as shown on Figure 5.10, the latency target (350ms) is longer than the latency of $Stream_0$ when it has isolated access (i.e., without another stream) to the storage system. Therefore, the storage system has sufficient performance capacity to meet the latency requirement of $Stream_0$. Since the performance capacity is supposed to be insufficient to simultaneously meet both latency requirements, and both SLAC and ^{2}TL prioritize a workload's latency requirements over its throughput requirements, no service will be allocated to the throughput-bound streams. Therefore, the throughput requirements do not matter and we



(a) SLAC. (b) ^{2}TL.

Figure 5.11: Experiment 3a: Average Meet Rates of $Stream_0$ and $Stream_1$. Performance Requirements of $Stream_0$ and $Stream_1$: $< 750ms, 99\% >$

choose the same throughput requirements used in Experiment 3a.

Results

The results of Experiment Set 3 are organized in four parts, those that relate to: (1) latency guarantees, (2) latency requirement prioritization, (3) individual stream prioritization, and (4) proportional sharing.

Table 5.10: Experiment 3a: Average Meet Rates of $Stream_0$ and $Stream_1$ with FCFS, SLAC and ^{2}TL.

Scheduler	**Stream$_0$ (prioritized)**	**Stream$_1$**
FCFS	60%	65%
SLAC	95%	81%
^{2}TL	99%	99%
Difference (^{2}TL benefit)	4%	18%

Latency Guarantees: Table 5.10 presents the average meet rates achieved by $Stream_0$ and $Stream_1$ in Experiment 3a, i.e., 60% and 65%, respectively, which are below their percentile ranks (both 99%). Thus, their latency requirements (both $< 750ms, 99\% >$) were not met with FCFS, showing that they cannot be met without careful scheduling. In contrast, with SLAC, the average meet rates of $Stream_0$ and $Stream_1$ are 95% and 81%, respectively, which also are below the streams' percentile ranks (both 99%). However, with ^{2}TL, the average meet rates of $Stream_0$ and $Stream_1$ both are equal to their percentile ranks and, thus, ^{2}TL is able to meet their latency requirements, while SLAC cannot. Accordingly, this experiment demonstrates the benefit of ^{2}TL's proactive scheduling of the requests of latency-bound streams when there are two latency-bound streams in a workload both issuing request bursts with different burst intervals. Figure 5.11 compares the latency-bound streams' meet rates with SLAC and ^{2}TL in Experiment 3a. As shown in the figure, with SLAC, when a request burst arrives, the meet rates drop sharply. In contrast, with ^{2}TL the meet rates are perturbed only slightly. For the higher-priority $Stream_0$, with SLAC and ^{2}TL, up to 22% and 4% of requests miss the latency target, respectively. For the lower-priority $Stream_1$, the difference is even bigger, i.e., with SLAC and ^{2}TL up to 68% and 6% of requests miss the latency target, respectively. This shows that ^{2}TL is more effective in meeting latency requirements. This is because ^{2}TL proactively increases the service allocation to each latency-bound stream when it issues a request burst in order to avoid latency requirement violations. Thus, we conclude that, *given a workload consisting of two throughput-bound streams and two latency-bound streams issuing request bursts with different burst intervals, ^{2}TL is more effective than SLAC in meeting the performance requirements of the latency-bound streams. This is because of ^{2}TL's proactive scheduling of the requests of latency-bound streams.*

Latency Requirement Prioritization: Table 5.11 presents the average throughput achieved by $Stream_2$ and $Stream_3$ in Experiment 3a. As shown by the data in the table, the throughput requirements of $Stream_2$ and $Stream_3$ (1.02 and 0.51 MB/s, respectively) were not simultaneously met with any scheduler. This is because in this experiment the performance

Table 5.11: Experiment 3a: Throughput Performance.

Scheduler	**AvgThroughput$_i$ (MB/s)**		**Deficiency$_i$ (%)**		**ActualRatio$_i$**		**Error$_i$**	
	Stream		**Stream**		**Stream**		**Stream**	
	2	**3**	**2**	**3**	**2**	**3**	**2**	**3**
FCFS	0.53	0.53	48%	N/A	0.50	0.50	-17%	17%
SLAC	0.67	0.34	34%	33%	0.66	0.34	-1%	1%
^{2}TL	0.63	0.34	38%	33%	0.65	0.35	-2%	2%

capacity is insufficient to simultaneously meet the performance requirements of both the latency-bound and throughput-bound streams in the workload. Both SLAC and ^{2}TL prioritize the scheduling of the requests of $Stream_0$ and $Stream_1$ to first meet their latency requirements and then allocate the remainder of the performance capacity to $Stream_2$ and $Stream_3$. $Stream_2$'s throughput deficiency with SLAC and ^{2}TL is 34% and 38%, respectively; and $Stream_3$'s throughput deficiency is 33% with both schedulers. Less performance capacity is available to the throughput-bound streams because ^{2}TL is more prompt in prioritizing the scheduling of the requests of $Stream_0$ and $Stream_1$ to avoid latency requirement violations. In addition, the average meet rates of $Stream_0$ and $Stream_1$ in this experiment do not exceed its percentile rank (99%). This implies that ^{2}TL is able to appropriately prioritize the scheduling of the requests of $Stream_0$ and $Stream_1$ and allocate as much storage service to $Stream_2$ and $Stream_3$ as is possible. Accordingly, we conclude that *given a workload consisting of two throughput-bound streams and two latency-bound streams issuing request bursts with different burst intervals, ^{2}TL is able to appropriately prioritize the workload's latency requirements, while striving to meet its throughput requirements with the best effort.*

Individual Stream Prioritization: In Experiment 3b, the performance capacity is insufficient to simultaneously meet the latency requirements of $Stream_0$ and $Stream_1$. Table 5.12

Table 5.12: Experiment 3b: Average Meet Rates of $Stream_0$ and $Stream_1$ with FCFS, SLAC and ^{2}TL.

Scheduler	**Stream$_0$ (prioritized)**	**Stream$_1$**
FCFS	1%	1%
SLAC	93%	28%
^{2}TL	99%	26%
Difference (^{2}TL benefit)	6%	-2%

Table 5.13: Experiment 3b: Throughput Performance.

	AvgThroughput$_i$ (MB/s)	
	Stream$_2$	**Stream$_3$**
SLAC	0.07	0.05
^{2}TL	0.08	0.05

presents the average meet rates of these streams in the experiment. As shown in the table, with FCFS the streams' latency requirements (both $< 550ms, 99\% >$) were not met and the scheduling of the requests of the higher-priority $Stream_0$ was not prioritized over those of the lower-priority $Stream_1$. With FCFS, the average meet rates of $Stream_0$ and $Stream_1$ are 1% and 1%, respectively, which are below their percentile ranks (both 99%). This shows that prioritization of latency requirements is not supported by FCFS. With SLAC, the average meet rates of $Stream_0$ and $Stream_1$ are 93% and 28%, respectively, and thus neither latency requirement is met. In contrast, with ^{2}TL, the average meet rates of $Stream_0$ and $Stream_1$ are 99% and 26%, respectively; $Stream_0$'s latency requirement is met but not $Stream_1$'s. Note that $Stream_1$'s average meet rate is lower with ^{2}TL than it is with SLAC. This is because ^{2}TL, unlike SLAC, discounts service allocated to $Stream_1$ in order to give priority to $Stream_0$. Because of its proactive scheduling of requests of

latency-bound streams, when $Stream_0$ issues request bursts, ^{2}TL prioritizes the scheduling of $Stream_0$'s requests more promptly than does SLAC to meet its latency requirement. In addition, $Stream_0$'s average meet rate does not exceed its percentile rank (99%). This implies that ^{2}TL is able to appropriately prioritize the scheduling of $Stream_0$ requests, while scheduling as many requests of $Stream_1$ as is possible.

Table 5.13 presents the throughput delivered to $Stream_2$ and $Stream_3$ with SLAC and ^{2}TL in Experiment 3b. As shown, both streams received negligible storage service and, therefore, the average throughput of each is almost 0. This is because the storage system's performance capacity is insufficient to simultaneously meet both latency requirements and both schedulers prioritize latency requirements. Recall, however, that only ^{2}TL meets $Stream_0$'s latency requirement – SLAC does not. This is because ^{2}TL is more effective in prioritizing the scheduling of the requests of $Stream_0$. Similarly, in Experiment 3a, with SLAC, although $Stream_0$'s latency requirement is not met, $Stream_1$ still receives service. This is because, when $Stream_0$ issues request bursts, SLAC is unable to promptly prioritize the scheduling of $Stream_0$'s requests. Accordingly, we conclude that *given a workload consisting of two throughput-bound streams and two latency-bound streams that issue request bursts with different burst intervals, when the storage system is unable to simultaneously meet both latency requirements, ^{2}TL is able to appropriately prioritize the scheduling of the requests of the higher-priority latency-bound stream over the other, while striving to meet the latency requirement of the lower-priority latency-bound stream with the best effort.*

Proportional Sharing: Table 5.11 presents the average throughput delivered to $Stream_2$ and $Stream_3$ in Experiment 3a with FCFS, SLAC and ^{2}TL. Since in Experiment 3b the average throughputs of $Stream_2$ and $Stream_3$ are negligible, they are not used in the discussion of proportional sharing. As shown in the table, with FCFS storage service was equally allocated to $Stream_2$ and $Stream_3$, i.e., each achieved 0.53 MB/s, rather than being proportionally allocated based on their throughput targets, i.e., 1.02 MB/s and 0.51 MB/s, respectively. Thus, errors in proportional sharing were -17% for $Stream_2$ and

17% for $Stream_3$. This shows that storage service cannot be proportionally allocated to throughput-bound streams without careful scheduling.

In contrast, with SLAC and ^{2}TL, the errors in proportional sharing in terms of $Stream_2$ and $Stream_3$ are 1% and 2%, respectively, which are negligible. Both SLAC and ^{2}TL proportionally allocate storage service to the throughput-bound streams based on their throughput targets. However, because of the insufficient performance capacity, proportional service allocation does not lead to meeting their throughput requirements. With SLAC and ^{2}TL, the average throughput of $Stream_2$ and $Stream_3$ is below their throughput targets. With SLAC, $Stream_2$ achieves 0.67 MB/s (vs. the target of 1.02 MB/s), while $Stream_3$ achieves 0.34 MB/s (vs. the target of 0.51 MB/s). With ^{2}TL, $Stream_2$ achieves 0.64 MB/s and $Stream_3$ achieves 0.34 MB/s. Accordingly, we conclude that *$SLAC$ and ^{2}TL are similarly effective in proportionally allocating storage service to throughput-bound streams based on their throughput targets.*

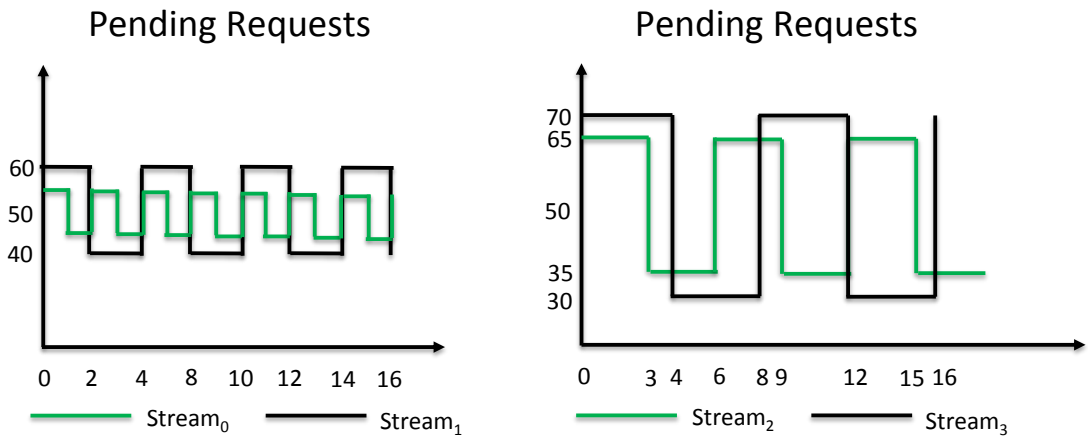


Figure 5.12: Experiment 4a: Number of Pending Requests for Latency-Bound Streams.

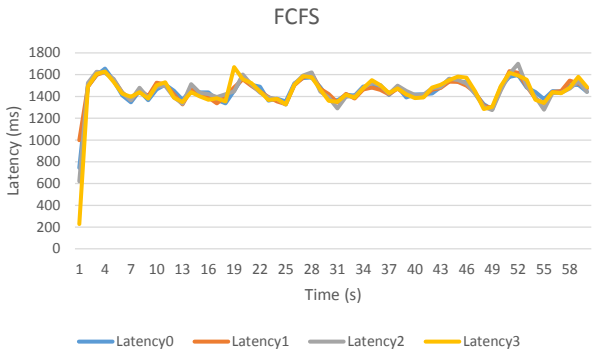
5.4.4 Experiment Set 4: Scalability

There is only one experiment in this set, Experiment 4a. Table 5.14 provides a high-level description of the 60-second simulation/experiment. As indicated by the table, the experiment is driven by a synthetic workload comprised of four latency-bound streams ($Stream_0$, $Stream_1$, $Stream_2$, and $Stream_3$, with a descending order of priorities) that generate request bursts and four throughput-bound streams ($Stream_4$, $Stream_5$, $Stream_6$, and $Stream_7$). The purpose of this experiment is to evaluate the scalability of ^{2}TL. To stress the scheduling algorithm, the streams have diverse performance requirements and burst characteristics, in terms of Burst Interval, Burst Size, and Base Pending Requests.

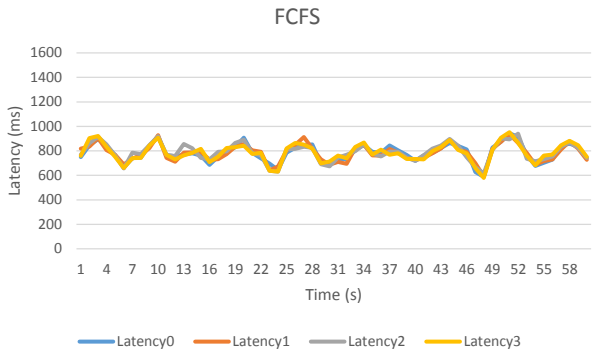
Objectives

This experiment has three objectives.

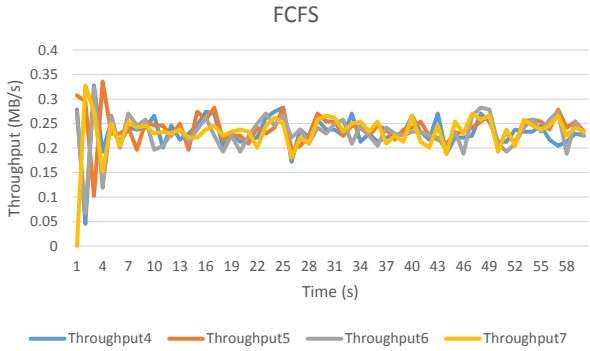
1. Demonstrate ^{2}TL's efficacy in meeting the latency requirements of four latency-bound streams in a workload, each of which issues request bursts and has different burst characteristics: The results of this experiment demonstrate that FCFS and SLAC cannot meet the performance requirements of any of the latency-bound streams, $Stream_0$ through $Stream_3$. This is because (1) the latency requirements cannot be met triv-



(a) Shared Access (i.e., 8 Streams) - Latency.



(b) Isolated Access (i.e., Latency-bound Streams) - Latency.



(c) Shared Access (i.e., 8 Streams) - Throughput.

Figure 5.13: Experiment 4a: Stream Performance with FCFS. Performance Requirements: $< 1000ms, 99\% >$ ($Stream_0$ and $Stream_1$), $< 1250ms, 99\% >$ ($Stream_2$ and $Stream_3$), 0.51 MB/s ($Stream_4$), 1.02 MB/s ($Stream_5$), 1.54 MB/s ($Stream_6$), and 2.05 MB/s ($Stream_7$).

Table 5.14: Experiment 4a: One 60-second Simulation driven by a Synthetic Workload of 8 Streams: 4 Latency-bound with Bursts and 4 Throughput-bound (S_i=$Stream_i$ and P_i=$Priority_i$).

Performance Capacity	**Performance Requirements**							
	Latency-bound				**Throughput-bound (MB/s (blocks/ms))**			
Sufficient to meet only latency requirements	S_0 P_1 (highest)	S_1 P_2	S_2 P_3	S_3 P_4	S_4	S_5	S_6	S_7
	$< 1000ms, 99\% >$		$< 1250ms, 99\% >$		0.51 (1)	1.02 (2)	1.54 (3)	2.05 (4)
Burst Interval (s)	2	4	6	8	N/A			
Burst Size	10	20	30	40				
Base Pending Requests	45	40	35	30				

ially, i.e., with FCFS; and (2) SLAC's reactive scheduling is unable to meet the latency requirements when request bursts arrive. In addition, this experiment demonstrates that, because of its proactive scheduling component, ^{2}TL is able to meet all of the latency requirements of the workload even though the four latency-bound streams have different burst characteristics.

2. Compare the effectiveness of SLAC and ^{2}TL in prioritizing the performance requirements of the latency-bound streams in the workload over those of the throughput-bound streams in the workload: The results demonstrate that, when the storage

system's performance capacity is insufficient to simultaneously meet the performance requirements of all of the streams in the workload, ^{2}TL, employing its proactive scheduling component, gives priority to the workload's four latency requirements over its four throughput requirements more promptly than does SLAC. In addition, ^{2}TL does not overly prioritize latency requirements and strives to meet the throughput requirements with best effort.

3. Compare the effectiveness of SLAC and ^{2}TL in allocating service to the four throughput-bound streams in the workload proportional to their throughput targets: The results of the experiment demonstrate that SLAC and ^{2}TL are similarly effective in allocating service to the four throughput-bound streams in the workload proportional to their throughput targets.

Workload

As mentioned above, the workload that drives Experiment 4a consists of four latency-bound streams, $Stream_0$ through $Stream_3$, and four throughput-bound streams, $Stream_4$ through $Stream_7$. To compose a complicated workload with performance requirements that are hard to meet through careful scheduling, the latency-bound streams in the workload have completely different burst characteristics. Table 5.14 presents the latency-bound streams' burst parameters. Burst Interval is set to 2, 4, 6, and 8 seconds for $Stream_0$ through $Stream_3$, respectively, and Burst Size is set to 10, 20, 30, and 40, respectively. $Stream_0$ issues bursts most frequently, yet its Burst Size (i.e., its burst intensity) is the smallest. Base Pending Requests is set to 45, 40, 35, and 30 for $Stream_0$ through $Stream_3$, resulting in the average of each stream's "higher" and "lower" levels of pending requests being 50. Figure 5.12 presents each stream's number of pending requests over time. Since the throughput-bound streams, $Stream_4$ through $Stream_7$, do not issue request bursts, they maintain a constant number of pending requests, i.e., 50.

The performance requirements of the eight streams in the workload that drives Exper-

iment 4a are based on the guidelines presented in Section 4.4. Because ^{2}TL gives priority to latency requirements, the performance requirements of $Stream_0$ through $Stream_3$ are deduced before those of the throughput-bound streams, $Stream_4$ through $Stream_7$. To evaluate ^{2}TL's efficacy in prioritizing a workload's latency requirements over its throughput requirements, the performance requirements are assigned so that after ^{2}TL meets all of the latency requirements of the workload, the performance capacity of the storage system is insufficient to simultaneously meet all of the workload's throughput requirements. To evaluate ^{2}TL efficacy in providing latency guarantees, the latency requirements of $Stream_0$ through $Stream_3$ need to be short enough that they cannot be met with FCFS (i.e., without careful scheduling), but they must be longer than their latencies during isolated access (i.e., without throughput-bound streams) to the storage system. Otherwise, the storage system would not have enough performance capacity to be able to simultaneously meet all of the latency requirements of the workload. As a result, we would not be able to evaluate the efficacy of the schedulers in fulfilling latency requirements. Figures 5.13a and 5.13b present the request latencies of the four latency-bound streams with FCFS during shared and isolated access to the storage system. During isolated access, the latencies do not exceed 1,000ms, while during shared access, the latencies are not shorter than 1,250ms. To evaluate the effectiveness of the schedulers in fulfilling the performance requirements of latency-bound streams in a workload with different latency requirements, we select two latency targets in the range between 1,000ms to 1,250ms to assign to the four streams. We assign $< 1,000ms, 99\% >$ as the latency requirement of $Stream_0$ and $Stream_1$, and $< 1,250ms, 99\% >$ as the latency requirement of $Stream_2$ and $Stream_3$. The average meet rates of $Stream_0$ through $Stream_3$ during shared access (with all eight streams in the workload) with FCFS are 0%, 0%, 8% and 9%, respectively, which are well below the streams' percentile ranks, each of which is 99%. This shows that the latency requirements of the workload cannot be met with FCFS, i.e., without careful request scheduling.

In order to assign the throughput requirements of the workload, we first quantify the performance capacity available to the throughput-bound streams after meeting the workload's

latency requirements. Then, we assign the throughput requirements based on the scenario required for the experiment, in this case, a storage system with performance capacity that is sufficient to meet all of the workload's latency requirements but none of its throughput requirements. Figure 5.13c presents the throughput achieved by each throughput-bound stream during shared access with FCFS. The sum of the average throughputs of the four streams is 0.92 MB/s (1.84 blocks/ms). To meet the latency requirements of the workload, a scheduler must allocate more service to the latency-bound streams than FCFS does and, therefore, less service to the throughput-bound streams. As a result, if the sum of throughput requirements is at least 0.92 MB/s (1.84 blocks/ms), the storage system will not be able to simultaneously meet all of the throughput requirements. Therefore, we set the throughput targets of $Stream_4$ through $Stream_7$ to be 0.51 MB/s (1 block/ms), 1.02 MB/s (2 blocks/ms), 1.53 MB/s (3 blocks/ms), and 2.04 MB/s (4 block/ms). The sum of these targets is 5.12 MB/s (10 blocks/ms), which exceeds the upper bound of the performance capacity available to the throughput-bound streams, i.e., 0.92 MB/s (1.84 blocks/ms). Thus, the 1:2:3:4 ratio allows us to evaluate the effectiveness of the schedulers in proportionally allocating service to the throughput-bound streams.

Results

The results of Experiment 4a are organized in three parts, those that relate to: (1) latency guarantees, (2) latency requirement prioritization, and (3) proportional service allocation. Note that the results during the first 2,000ms are presented in the figures but are not used in the calculation of the performance measurements. This is because, given the latency targets of $Stream_2$ and $Stream_3$ (1,250ms), ^{2}TL does not start scheduling their requests until the latter part of the second 1,000ms interval.

Latency Guarantees: Table 5.15 presents the average meet rates of the latency-bound streams in the workload that drove Experiment 4a. As shown in the table, their latency requirements were never met with FCFS; the average meet rates of $Stream_0$ through $Stream_3$ are 0%, 0%, 8%, and 9%, respectively, which are far below the streams' percentile ranks

Table 5.15: Experiment 4a: Latency-bound Streams' Average Meet Rates with FCFS, SLAC and ^{2}TL.

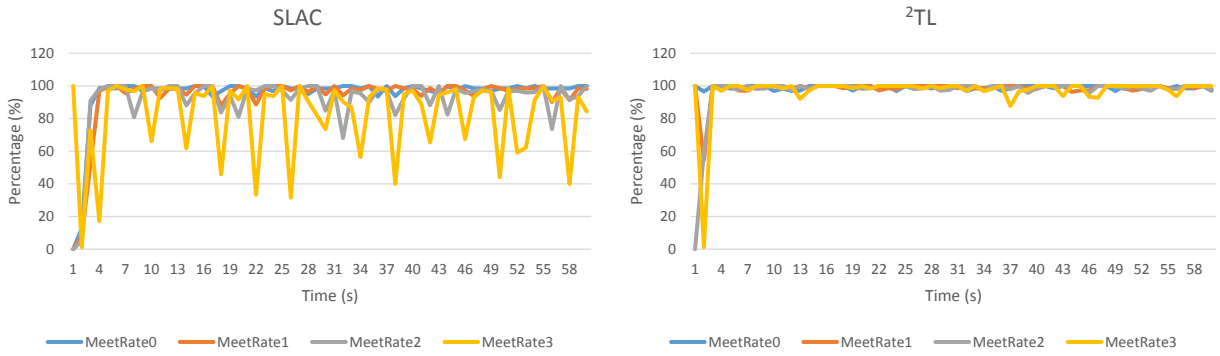
Scheduler	**Stream$_0$**	**Stream$_1$**	**Stream$_2$**	**Stream$_3$**
FCFS	0%	0%	8%	9%
SLAC	97%	95%	93%	82%
^{2}TL	99%	99%	99%	99%
Difference (^{2}TL benefit)	2%	4%	6%	17%

Table 5.16: Experiment 4a: Latency-bound Streams' Minimum Meet Rates with SLAC and ^{2}TL.

Scheduler	**Stream$_0$**	**Stream$_1$**	**Stream$_2$**	**Stream$_3$**
SLAC	88%	52%	68%	17%
^{2}TL	97%	96%	96%	88%
Difference (^{2}TL benefit)	11%	44%	28%	71%

(each of which is 99%). This shows that the latency requirements of the workload cannot be met without careful scheduling.

With SLAC, the average meet rates of $Stream_0$ through $Stream_3$ are 97%, 95%, 93%, and 82%, respectively, which are also below the streams' percentile ranks. In contrast, with ^{2}TL, the average meet rates of $Stream_0$ through $Stream_3$ are equal to streams' percentile rank. Therefore, ^{2}TL met the latency requirements, while SLAC did not. These experiments demonstrate the benefit of ^{2}TL's proactive scheduling component in a complicated workload that consists of four latency-bound streams that issue request bursts and have different burst characteristics, and four throughput-bound streams. Figure 5.14 compares the latency-bound streams' meet rates with SLAC and with ^{2}TL in Experiment 4a. As



(a) SLAC.

(b) ^{2}TL.

Figure 5.14: Experiment 4a: Latency Performance with SLAC and ^{2}TL. Performance Requirements: $< 1000ms, 99\% >$ ($Stream_0$ and $Stream_1$) and $< 1250ms, 99\% >$ ($Stream_2$ and $Stream_3$)

shown in Figure 5.14a, with SLAC, when a request burst arrived, the meet rates decreased sharply. In contrast, as shown in Figure 5.14b, with ^{2}TL, the meet rates remained fairly constant. Table 5.16 presents the minimum meet rates of the latency-bound streams. With SLAC, the minimum meet rates of $Stream_0$ through $Stream_3$ are 88%, 52%, 68%, and 17%, respectively. This translates to up to 12%, 48%, 32%, and 83% of the four streams' requests missing their latency targets during a one-second interval. In contrast, with ^{2}TL, the minimum meet rates of $Stream_0$ through $Stream_3$ are 97%, 96%, 96%, and 88%, respectively. This translates to up to only 3%, 4%, 4%, and 12% of the four streams' requests missing their latency targets during a one-second interval. The results show that many more requests met their latency targets with ^{2}TL than with SLAC. Note that, with ^{2}TL, up to 12% of $Stream_3$ requests missed their latency target during a one-second interval, which is significantly more than for requests of $Stream_0$ (3%), $Stream_1$ (4%), and $Stream_2$ (4%). This may be the result of ^{2}TL delaying the start of its exclusive scheduling of the requests of latency-bound streams (see Chapter 3) because of an over-estimation of the latency-bound streams' scheduling rates. Since $Stream_3$ has the lowest priority of the four latency-bound streams, its requests would be affected the most by this delay. When compared with SLAC,

^{2}TL is more effective in meeting the latency requirements of the workload because in order to avoid latency requirements violations, it proactively increases the service allocated to a latency-bound stream when it issues a request burst. In summary, because of ^{2}TL's proactive scheduling component, *given a workload consisting of four latency-bound streams that have different burst characteristics and four throughput-bound streams, ^{2}TL is more effective than SLAC in meeting the performance requirements of the latency-bound streams.*

Table 5.17: Experiment 4a: Throughput-bound Streams' Performance (MB/s).

Scheduler	**Stream$_4$**	**Stream$_5$**	**Stream$_6$**	**Stream$_7$**
FCFS	0.23	0.24	0.23	0.24
SLAC	0.13	0.16	0.17	0.18
(**^{2}TL benefit**)	0.13	0.15	0.16	0.17

Latency Requirement Prioritization: Table 5.17 presents the average throughput achieved by the throughput-bound streams in the workload that drove Experiment 4a. None of the throughput requirements of $Stream_4$, $Stream_5$, $Stream_6$, and $Stream_7$ (0.51, 1.02, 1.53, and 2.04 MB/s, respectively) was met with any scheduler. This is because the performance capacity of the storage system is sufficient to meet only the latency-requirements of the workload. Both SLAC and ^{2}TL gave priority to the latency-bound streams, endeavoring to first meet their latency requirements before allocating the remainder of the storage service to the throughput-bound streams.

The average throughput of three of the four throughput-bound streams is slightly higher with SLAC; $Stream_4$ has the same average throughput with SLAC and ^{2}TL. This is because, when request bursts arrived, ^{2}TL's proactive scheduling component promptly discounted the service allocated to the throughput-bound streams in order to provide the latency-bound streams with sufficient service to meet their performance requirements. In addition, as shown in Table 5.15, the latency-bound streams' average meet rates achieved in the experiment do not exceed their percentile ranks (99% for each). This demonstrates that

^{2}TL was able to appropriately prioritize latency-bound streams and allocated as much storage service as possible to throughput-bound streams in the workload. In summary, *given a workload consisting of four latency-bound streams with different burst characteristics and four throughput-bound streams, ^{2}TL is able to appropriately prioritize the scheduling of the requests of the latency-bound streams, while striving to meet the workload's throughput requirements with best effort.*

Table 5.18: Experiment 4a: Proportional Service Allocation to Throughput-bound Streams.

	$ActualRatio_i$				$Error_i$			
Scheduler	$Stream_4$	$Stream_5$	$Stream_6$	$Stream_7$	$Stream_4$	$Stream_5$	$Stream_6$	$Stream_7$
FCFS	0.25	0.25	0.25	0.25	15%	5%	-5%	-15%
SLAC	0.20	0.25	0.27	0.28	10%	5%	-3%	-12%
^{2}TL	0.21	0.25	0.27	0.27	11%	5%	-3%	-13%

Proportional Service Allocation: Table 5.18 presents the throughput-bound streams' ratios of throughput service and the errors in proportional service allocation delivered by FCFS, SLAC and ^{2}TL in Experiment 4a. With FCFS, each of the throughput-bound streams, i.e., $Stream_4$, $Stream_5$, $Stream_6$, and $Stream_7$, received a quarter of the storage service available to them. Storage service was equally allocated to the throughput-bound streams rather than being proportionally allocated based on the ratio of their throughput targets (1:2:3:4); the associated errors in proportional sharing are bounded by 15%. This illustrates that FCFS does not proportionally allocate storage service to throughput-bound streams, and that this cannot be done without careful scheduling.

In contrast, with SLAC and ^{2}TL, the errors in proportional sharing of the storage service among the four streams are bounded by 12% and 13%, respectively. These errors are larger than those experienced in Experiment 3a (up to only 2%). This is because the workload in this experiment (four latency-bound streams and four throughput-bound streams) is larger than the workload in Experiment 3a (two latency-bound streams and two throughput-

bound streams), and the ratios of the throughput targets in this experiment (1:2:3:4) is much wider than that of Experiment 3a (1:2). Therefore, we view ^{2}TL's performance to be acceptable in this case – both SLAC and ^{2}TL were able to proportionally allocate storage service to the throughput-bound streams based on their throughput targets. However, because of insufficient performance capacity, proportional service allocation did not lead to fulfillment of their throughput requirements. With SLAC and ^{2}TL, the average throughput of each of $Stream_4$, $Stream_5$, $Stream_6$, and $Stream_7$ is below its throughput target (0.51, 1.02, 1.53, and 2.04 MB/s, respectively). In summary, *SLAC and ^{2}TL are similar effective in proportionally allocating storage service to throughput-bound streams based on their throughput targets.*

5.4.5 Experiment Set 5: Real Workload

There is only one experiment in this set, Experiment 5a. Table 5.19 provides a high-level description of this 10-minute simulation/experiment. As indicated in the table, Experiment 4a is driven by a real workload comprised of one latency-bound stream ($Stream_0$) that generates request bursts and one throughput-bound stream ($Stream_1$). In this experiment, Strategy 2 for estimating the Exclusive Scheduling Rate of Latency-bound Requests, which is based on a moving-percentile model, was used. This is because Strategy 1 did not lead to satisfactory results.

Objectives

This experiment has two objectives:

1. Demonstrate ^{2}TL's efficacy in meeting the latency requirement of a latency-bound stream, when the requests are generated by a real application: The results of this experiment demonstrate that FCFS and SLAC cannot meet the latency requirement of $Stream_0$. This is because (1) the latency requirement cannot be met without careful scheduling, i.e., with FCFS; and (2) SLAC's reactive scheduling fails to meet the latency requirement when bursts arrive.

2. Demonstrate that ^{2}TL is able to prioritize the performance requirement of the latency-bound stream in a real workload over the performance requirement of the throughput-bound stream in the workload when necessary: ^{2}TL dynamically determines when to give priority to $Stream_0$'s latency requirement over $Stream_1$'s throughput requirement. When the storage system's performance capacity is insufficient to simultaneously meet the performance requirements of both streams, ^{2}TL discounts the service allocated to $Stream_1$ and allocates extra service to $Stream_0$. Otherwise, ^{2}TL does not give priority to $Stream_0$'s latency requirement.

Table 5.19: Experiment Set 5: One 10-minute Simulation driven by a Real Workload of 2 Streams: 1 Latency-bound with Bursts and 1 Throughput-bound.

Performance Capacity	**Performance Requirements**	
	Latency-bound	**Throughput-bound**
	Stream$_0$ (OLTP)	**Stream$_1$ (varmail)**
Sufficient to meet only the latency requirement	$< 550ms, 99\% >$	10 MB/s

Workload

As mentioned above and indicated in Table 5.19, the workload of Experiment 4a is comprised of a latency-bound stream, $Stream_0$, and a throughput-bound stream, $Stream_1$. The requests in the two streams are input to DiskSim as I/O traces. The trace of the latency-bound stream captured the requests generated by an Online Transaction Processing (OLTP) application running in a large financial institution [26]. To keep our simulations manageable, we truncated the trace after the first 40 minutes of activity. In order to increase the burstiness of the trace and make the latency requirement harder to meet, we compressed the trace to 10 minutes, thus, quadrupling its request arrival rate. Note that this is an acceptable approach that has been used in the literature [40]. The trace of the throughput-bound stream is the varmail trace [4], which captured the requests generated by four mail servers concurrently accessing a storage system for 10 minutes.

As shown in Table 5.19, the performance requirement of the latency-bound stream, $Stream_0$, is $< 550ms, 99\% >$ and that of the throughput-bound stream, $Stream_1$, is 10 MB/s. Next we explain how we used the guidelines presented in Section 4.4 to determine these performance requirements. Because ^{2}TL prioritizes latency requirements, $Stream_0$'s latency requirement is deduced before $Stream_1$'s throughput requirement. To evaluate ^{2}TL's efficacy in fulfilling latency requirements, $Stream_0$'s latency requirement must be short enough so that it cannot be met with FCFS, but it must be longer than its request latencies during isolated access to the storage system with FCFS. Figures 5.15 and 5.16a

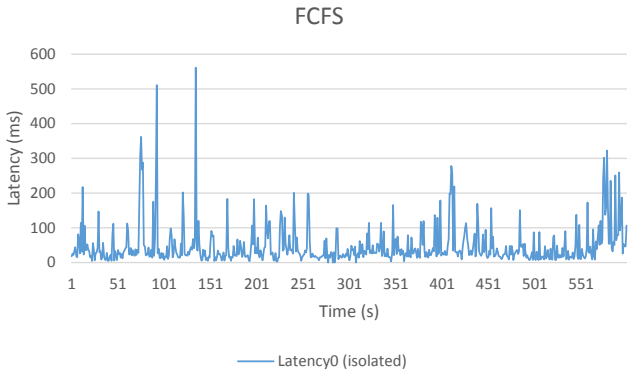
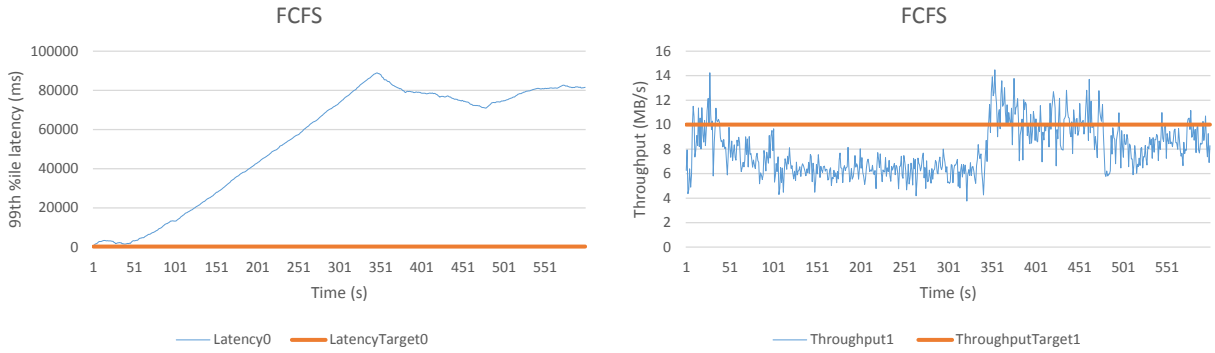


Figure 5.15: Experiment 5a: $Stream_0$'s End-to-end Latency during Isolated Access. $Stream_0$ Latency Requirement: $< 550ms, 99\% >$.



(a) $Stream_0$'s Latency.

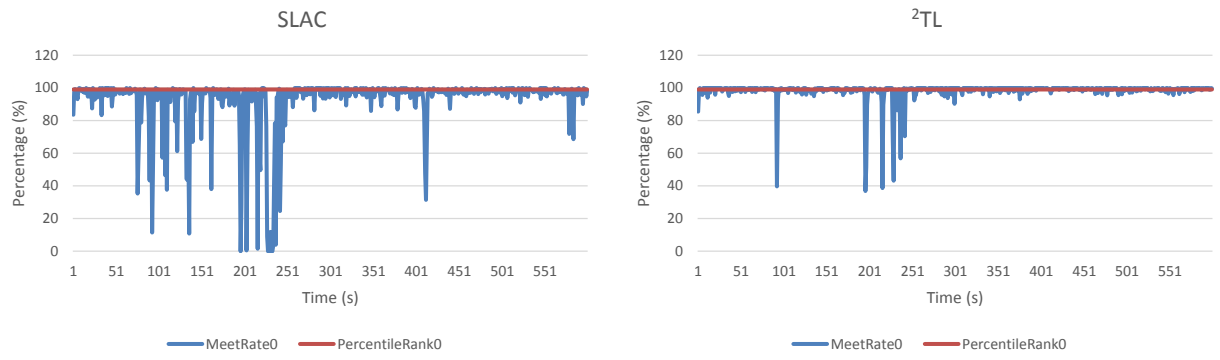
(b) $Stream_1$'s Throughput.

Figure 5.16: Experiment 5a: Stream Performance during Shared Access with FCFS. Performance Requirements of $Stream_0$ and $Stream_1$: $< 550ms, 99\% >$ and 10 MB/s.

present $Stream_0$'s request latencies with FCFS during isolated access and shared access, respectively. As shown in Figure 5.16a, during shared access with FCFS, $Stream_0$'s latencies exceed 550ms almost immediately after the simulation begins. Therefore, a latency target of 550ms, 99% cannot be met with FCFS. As shown in Figure 5.15, the 550ms latency target is nearly always longer than $Stream_0$'s end-to-end latencies in the I/O hierarchy (the Shim and storage system, combined) during isolated access with FCFS. Therefore, the storage system does have sufficient performance capacity to meet $Stream_0$'s latency requirement if we set it to $< 550ms, 99\% >$.

To obtain $Stream_1$'s throughput requirement, we measured $Stream_1$'s throughput during shared access (with $Stream_0$) to the storage system with FCFS. As shown in Figure 5.16b, $Stream_1$'s average throughput in this situation is 7.93 MB/s. $Stream_1$ achieves its best throughput from interval 350 to interval 450, during which its average throughput is about 10.00 MB/s. As shown in Table 5.19, Experiment 5a is supposed to present a scenario when the storage system is able to meet only the latency requirement of the workload. This allows us to evaluate ^{2}TL's efficacy in prioritizing the latency requirements of a workload over its throughput requirements. To create such a scenario, i.e., where $Stream_1$'s throughput requirement cannot be met with SLAC and ^{2}TL, after they meet the latency requirement of $Stream_0$, $Stream_1$'s throughput target must be higher than the average throughput it achieved during shared access with FCFS (7.93 MB/s). Thus, we set $Stream_1$'s throughput requirement to 10.00 MB/s.

In this experiment, ^{2}TL uses Strategy 2, which is based on a moving-percentile model, to estimate the next Exclusive Scheduling Rate of Latency-bound Requests based on recent exclusive scheduling rates. The moving-percentile model takes the 10th percentile of the 10 most recent Exclusive Scheduling Rates of Latency-bound Requests (i.e., history size= 10) to be the next Exclusive Scheduling Rate of Latency-bound Requests.



(a) SLAC.

(b) ^{2}TL.

Figure 5.17: Experiment 5a: $Stream_0$'s Meet Rates with SLAC and ^{2}TL. $Stream_0$ Latency Requirement: $< 550ms, 99\% >$.

Results

The results of Experiment 5a are organized in two parts, those related to: (1) latency guarantees and (2) latency requirement prioritization.

Table 5.20: Experiment 5a: $Stream_0$'s Average Meet Rates with SLAC and ^{2}TL. ($Stream_0$'s Percentile Rank is 99%).

Scheduler	**Intervals**		
	1-600	**1-255**	**256-600**
SLAC	93%	87%	97%
^{2}TL	98%	97%	99%

Latency Guarantees: Table 5.20 presents Stream0's average meet rates with SLAC and ^{2}TL during different sets of intervals of the simulation. As shown, during the entire experiment (Intervals 1-600), $Stream_0$'s latency requirement was not met by either scheduler. $Stream_0$'s meet rates are 93% and 98% with SLAC and ^{2}TL, respectively, which are smaller than its percentile rank (99%). Figure 5.17 presents $Stream_0$'s meet rates with SLAC and ^{2}TL. As shown in the figure, during the first 255 one-second intervals, $Stream_0$'s meet rates

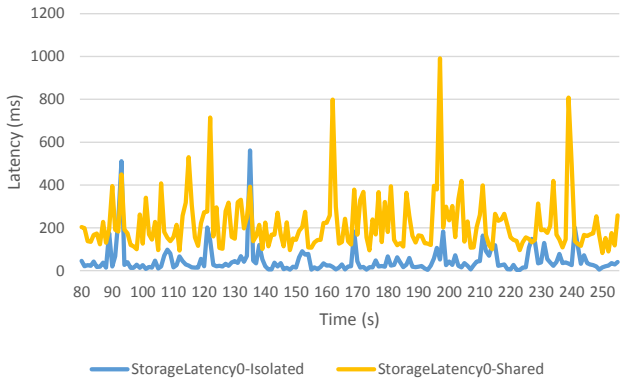


Figure 5.18: Experiment 5a: $Stream_0$'s Request Storage Latency during Shared (with ^{2}TL) and Isolated Access (with FCFS). $Stream_0$ Latency Requirement: $< 550ms, 99\% >$.

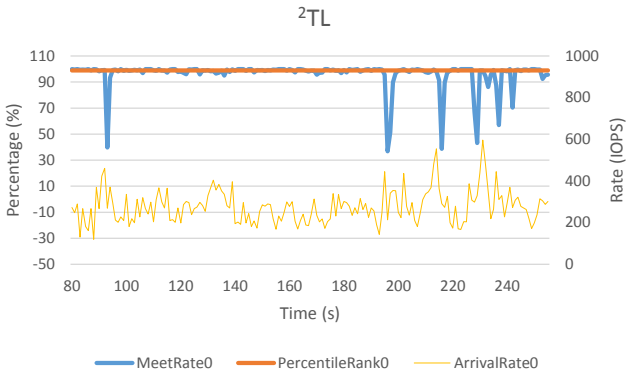


Figure 5.19: Experiment 5a: $Stream_0$'s Request Arrival Rates and Meet Rates with ^{2}TL during Shared Access. $Stream_0$ Latency Requirement: $< 550ms, 99\% >$.

Table 5.21: Experiment 5a: $Stream_1$'s Average Throughput with SLAC and ^{2}TL. ($Stream_1$'s throughput target is 10.00 MB/s.)

Stream$_1$ Average Throughput (MB/s).		
Scheduler	**Intervals**	
	1-600	**250-390**
SLAC	6.48	9.94
^{2}TL	6.39	10.03

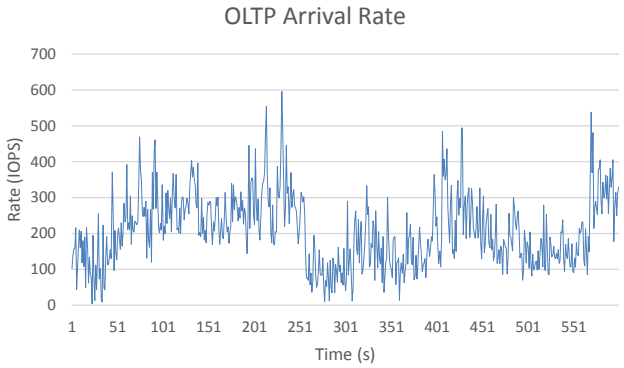


Figure 5.20: Experiment 5a: $Stream_0$'s Request Arrival Rate.

dropped significantly in many intervals - more intervals with SLAC than with ^{2}TL. After the largest sequence of these drops, for the most part $Stream_0$'s meet rates were close to (SLAC) if not at (^{2}TL) its percentile rank.

Because $Stream_0$'s latency performance is worse during the first 255 intervals, we analyzed its performance with ^{2}TL during the two periods separately. As shown in Table 5.20, during the first 255 intervals, $Stream_0$'s latency requirement was not met by either scheduler. $Stream_0$'s average meet rates with SLAC and ^{2}TL for this period are 87% and 97%, respectively. Although $Stream_0$'s latency requirement for this period was not met by ^{2}TL, its average meet rate with ^{2}TL is 10% higher than with SLAC. And, its average meet rate with ^{2}TL is only 2% less than its percentile rank. Also shown in Table 5.20 is $Stream_0$'s

meet rates for intervals 256 to 600; during this period ^{2}TL met $Stream_0$'s latency requirement but SLAC did not. $Stream_0$'s average meet rates during this period with SLAC and ^{2}TL are higher, i.e., 97% and 99%, respectively. For both $Stream_0$'s average meet rates are higher with ^{2}TL than with SLAC. This is because, when $Stream_0$ issued a request burst, ^{2}TL promptly increased service allocation to $Stream_0$ to avoid latency requirement violations. This demonstrates the benefit of ^{2}TL's proactive scheduling component in meeting the latency requirement of this real workload. Nonetheless, ^{2}TL was unable to meet $Stream_0$'s latency requirement during the first 255 intervals for two reasons: (1) $Stream_0$ experienced prolonged request storage latencies during shared access, and (2) $Stream_0$'s request arrival rate (IOPS) exceeded the storage system's request service rate (IOPS). Figure 5.18 compares $Stream_0$'s request storage latencies during isolated access (blue) and shared access (yellow) with ^{2}TL. During shared access, $Stream_0$'s request storage latencies are much higher than during isolated access. In fact, in some intervals, $Stream_0$'s request storage latencies either exceed or are very close to its request latency target (550ms). In this case, even if ^{2}TL scheduled $Stream_0$ requests into the storage system before their service deadlines, many of them would not have met the latency target because of the long request storage latency. Accordingly, during those intervals, although ^{2}TL exclusively scheduled $Stream_0$ requests, $Stream_0$'s latency requirement was not met.

Figure 5.19 presents $Stream_0$'s request arrival rates (yellow) and meet rates (blue) with ^{2}TL. $Stream_0$'s meet rate dropped sharply when it issued request bursts. $Stream_0$ issued some of the bursts at request arrival rates higher than the storage system's request service rate, i.e., its performance capacity. Therefore, the storage system was unable to meet $Stream_0$'s latency requirement during those intervals. Although $Stream_0$ issued some of the bursts at rates lower than the storage system's performance capacity, because of the long request storage latency mentioned above, $Stream_0$'s latency requirement was not met during these intervals as well. When $Stream_0$ issued request bursts, ^{2}TL promptly increased service allocation to $Stream_0$. When necessary, it exclusively scheduled $Stream_0$requests and strived to meet its latency requirement. In this experiment, $Stream_0$'s latency require-

ment during the entire simulation (Intervals 1-600) was not met. This is not because ^{2}TL is incapable of doing so but because of the long request storage latency and the insufficient performance capacity of the storage system. In summary, because of ^{2}TL's proactive scheduling component, *given a real workload consisting of a latency-bound stream and a throughput-bound stream, ^{2}TL is more effective than SLAC in meeting the workload's latency requirement.*

Latency Requirement Prioritization: Table 5.21 presents $Stream_1$'s average throughputs with SLAC and ^{2}TL during Experiment 5a (Intervals 1-600). As shown in the table, $Stream_1$'s throughput requirement (10.00 MB/s) was not met by either scheduler. This is because the performance capacity of the storage system is insufficient to simultaneously meet both the latency and throughput requirements of the workload. Both SLAC and ^{2}TL gave priority to servicing $Stream_0$ to first meet its latency requirements, before allocating the remainder of the performance capacity to $Stream_1$. $Stream_1$'s average throughput is higher with SLAC (6.48 MB/s) than with ^{2}TL (6.39 MB/s) because ^{2}TL, in order to avoid latency requirement violations, was more prompt in prioritizing $Stream_0$'s latency requirement over $Stream_1$'s throughput requirement. In addition, as shown in Table 5.20, during Intervals 256-600 $Stream_0$'s average meet rate does not exceed its percentile rank (99%). This implies that ^{2}TL was able to appropriately prioritize service to $Stream_0$ and allocate as much storage service as was possible to $Stream_1$. As shown in Figure 5.20, $Stream_0$'s request arrival rates were relatively low during Intervals 250-390 and, thus, during this period the performance capacity was sufficient to simultaneously meet both performance requirements of the workload. Accordingly, during this period $Stream_0$'s average meet rate with ^{2}TL was 99%, which equals its percentile rank (99%), and, as shown in Table 5.21, $Stream_1$'s average throughput (10.03 MB/s) is larger than its throughput target. Note that ^{2}TL did not prioritize service to $Stream_0$ during this period. If it had, $Stream_0$'s latency requirement would have been overmet. In summary, *given a real workload consisting of a latency-bound stream and a throughput-bound stream that issues request bursts, ^{2}TL is able to dynamically determine when to give priority to the scheduling*

of requests of the latency-bound streams. And, when it does, [2] *TL appropriately gives priority to the latency requirement, while striving to meet the throughput requirement with best effort.*

5.4.6 Experiment Set 6: Homogeneous Performance Requirements

Tables 5.22 and 5.23 provide high-level descriptions of the five simulations/experiments in this set, called 6a, 6b, 6c, 6d, and 6e. As indicated in the tables, each experiment is driven by a synthetic workload comprised of two streams ($Stream_0$ and $Stream_1$). In Experiments 6a, 6b, and 6c both streams are latency-bound and $Stream_0$ has the higher priority. Also, $Stream_0$ generates request bursts, while $Stream_1$ does not. Note that, in Experiment 6c, Strategy 2 (based on a moving-percentile model) is used to estimate the Exclusive Scheduling Rate of Latency-bound Requests. This is because using Strategy 1 did not lead to satisfactory results. In Experiments 6d and 6e both streams are throughput-bound.

Objectives

This set of experiments has three objectives.

1. Compare the effectiveness of SLAC and ^{2}TL in meeting the performance requirements of all of the latency-bound streams in a workload: The results of Experiment 6c demonstrate that given sufficient performance capacity, both SLAC and ^{2}TL have comparable performance in meeting the performance requirements of all of the latency-bound streams in the workload. In this specific experiment, because there is sufficient performance capacity to meet the performance requirements of all of the latency-bound streams in the workload, ^{2}TL's proactive scheduling component does not provide any advantage over SLAC's reactive component.

2. Compare the effectiveness of SLAC and ^{2}TL in prioritizing the scheduling of the requests of the highest-priority latency-bound stream in a workload: The results of Experiments 6a and 6b demonstrate that, when the storage system does not have sufficient performance capacity to simultaneously meet the performance requirements of the two latency-bound streams in the workload, ^{2}TL, via its proactive scheduling component, is more effective than SLAC in giving priority to the servicing of

the higher-priority latency-bound stream, $Stream_0$, over the lower-priority stream, $Stream_1$, to meet $Stream_0$'s latency requirement.

3. Compare the effectiveness of SLAC and ^{2}TL in allocating storage service to the throughput-bound streams in a workload proportional to their throughput targets: The results of Experiments 6d and 6e demonstrate that SLAC and ^{2}TL are similarly effective in allocating service to the two throughput-bound streams in the workload proportional to their throughput targets. And, when the performance capacity is sufficient to simultaneously meet the streams' throughput requirements (Experiment 6e), proportional service allocation makes it possible to provide throughput guarantees.

Workload

As mentioned above, the workload of each experiment is comprised of two streams, $Stream_0$ and $Stream_1$. In Experiments 6a, 6b, and 6c, both streams are latency-bound. $Stream_0$, the higher-priority stream, issues request bursts, while $Stream_1$ does not. As shown in Table 5.22, $Stream_0$ issues a request burst every 2 seconds with a burst size of 60 and its "lower" level of pending requests is 20. Since $Stream_1$ does not issue request bursts, it maintains a constant number of pending requests in the Shim, i.e., 50. As shown in Table 5.23, in Experiments 6d and 6e, both streams are throughput-bound and do not issue request bursts. For these streams, DiskSim maintains 50 pending requests in the Shim.

The performance requirements of the workloads that drive the five experiments in this set are assigned based on the guidelines presented in Section 4.4. Below we first discuss how we select the performance requirements of the streams that drive Experiments 6a, 6b, and 6c, before discussing how we select them for the streams that drive Experiments 6d and 6e.

To determine $Stream_0$'s latency requirements for Experiment 6a, 6b, and 6c, we used

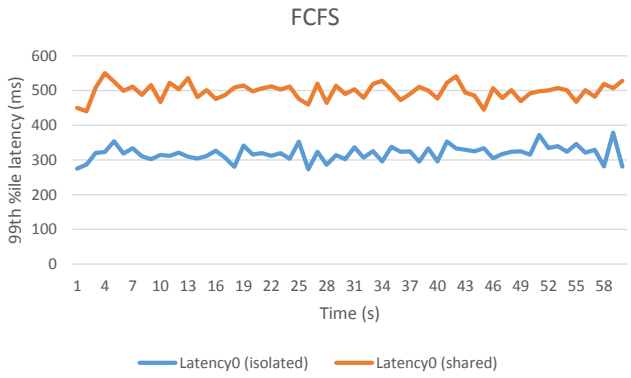


Figure 5.21: Experiment 6a, 6b and 6c: $Stream_0$'s Latencies with FCFS during Isolated and Shared Access. Latency Requirements of $Stream_0$: $< 200ms, 99\% >$ (6a), $< 400ms, 99\% >$ (6b and 6c).

the latencies experienced by $Stream_0$ during two experiments with FCFS: (1) $Stream_0$ only, i.e., during isolated access (without $Stream_1$) to the storage system, and (2) $Stream_0$ and $Stream_1$, i.e., during shared access (with $Stream_1$). Figure 5.21 shows $Stream_0$'s latencies during these two experiments. As shown in the figure, to assign latency requirements that cannot be met trivially, i.e., without careful scheduling of requests (with FCFS), the latency targets for the three experiments must be shorter than $Stream_0$'s latencies with FCFS during shared access, i.e., about 550ms. Since Experiment 6a is meant to quantify scheduler performance when there is insufficient performance capacity to meet $Stream_0$'s latency requirement, we set its latency target to 200ms (high demand), which is impossible for the storage system to achieve even when $Stream_0$ is the only stream that accesses the storage system (isolated access). ($Stream_0$'s latencies with FCFS during isolated access to the storage system are between 273ms and 378ms.) In Experiments 6b and 6c, the performance capacity of the storage system is supposed to be sufficient to meet $Stream_0$'s latency requirement, either with (6c) or without (6b) meeting the latency requirement of $Stream_1$. Thus, for both experiments, we set the latency target for $Stream_0$ to 400ms. This latency target is shorter than what $Stream_0$ achieves with FCFS during shared access

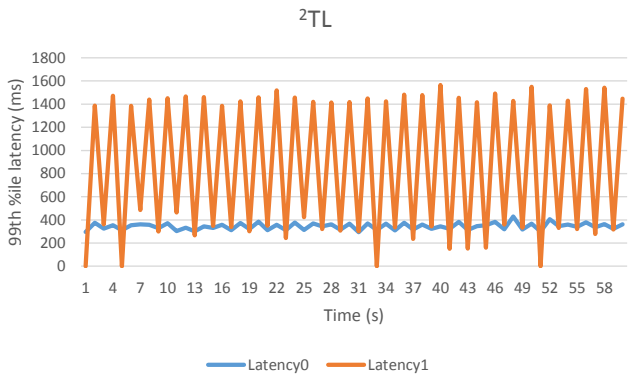


Figure 5.22: Experiment 6c: Stream Latencies with ^{2}TL during Test Simulation. $Stream_0$ Latency Requirement: $< 400ms, 99\% >$.

(i.e., with $Stream_1$) but longer than what the storage system can deliver with FCFS during isolated access by $Stream_0$. Accordingly, this latency requirement can be fulfilled by the performance capacity of the storage system.

Given that for Experiment 6a, we assigned a latency requirement to $Stream_0$ that the performance capacity of the storage system cannot fulfill, we can choose any latency requirement for $Stream_1$. This is because, in this scenario, it will be allocated no service by SLAC or ^{2}TL. Thus, we arbitrarily set the latency requirement of $Stream_1$ to $< 500ms, 99\% >$, which is approximately its average latency during shared access with FCFS, as shown in Figure 5.21.

For Experiment 6b, $Stream_0$ was assigned a latency requirement that could be fulfilled by the performance capacity of the storage system. Since for this experiment the performance capacity is supposed to be sufficient to only fulfill the latency requirement of $Stream_0$, we also set the latency requirement of $Stream_1$ to $< 400ms, 99\% >$. Setting the latency target of both streams to 400ms makes it possible for the performance capacity of the storage system to fulfill $Stream_0$'s latency requirement but not $Stream_1$'s. This is because $Stream_0$ has the higher priority and in the experiment with FCFS driven by both streams, the delivered latencies (with shared access) were between 450ms and 575ms.

Experiment 6c presents a scenario where the latency requirements of all of the streams in the workload can be met. As described in Section 4.4.2, in this case, a test simulation with ^{2}TL is required to finalize the assignment of the latency requirement of $Stream_1$ (with the lowest priority). In the test simulation with ^{2}TL the latency requirement assigned to $Stream_0$ is that which it was assigned for Experiment 6c, i.e., $< 400ms, 99\% >$. Via the test simulation, we obtain the latency performance of $Stream_1$ and use it to assign a latency requirement to $Stream_1$ for Experiment 6c. Since $Stream_0$ has the same latency requirement in Experiment 6b as it would in the test simulation, we consider Experiment 6b to be the test simulation and use the latency performance of $Stream_1$ in this experiment to select a latency requirement for $Stream_1$ in Experiment 6c. As shown in Figure 5.22, the latencies experienced by $Stream_1$ in Experiment 6b are shorter than 1,600ms. Thus, to create a scenario where there is sufficient performance capacity to meet the latency requirement of $Stream_1$ after meeting that of $Stream_0$, we set the latency target of $Stream_1$ to 1,600s. In Experiment 6c, ^{2}TL uses Strategy 2, which is based on a moving-percentile model, to estimate the next Exclusive Scheduling Rate of Latency-bound Requests. As mentioned in Section 4.5, when Strategy 2 is used, the performance of ^{2}TL relies on the appropriate selection of parameters. The parameters used in Experiment 5a (i.e., percentile = 10th and history size = 10) did not lead to satisfactory results in this experiment. Among different sets of parameters that we examined, the following parameters produce the best results: history size = 1,000, percentile of $Stream_0$ = 50th, and percentile of $Stream_1$ = 1st, respectively. The reasons for selecting this set of parameters are presented with the results of this experiment.

As shown in Table 5.23, Experiments 6d and 6e present two different scenarios. For Experiment 6d, the storage system's performance capacity is supposed to be insufficient to simultaneously meet the performance requirements of both throughput-bound streams in the workload. In contrast, in Experiment 6e, the storage system's performance capacity is supposed to be sufficient to simultaneously meet the performance requirements of both throughput-bound streams in the workload. To determine the throughput requirements

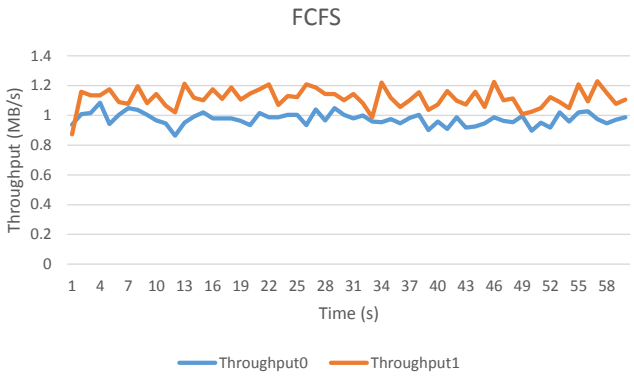
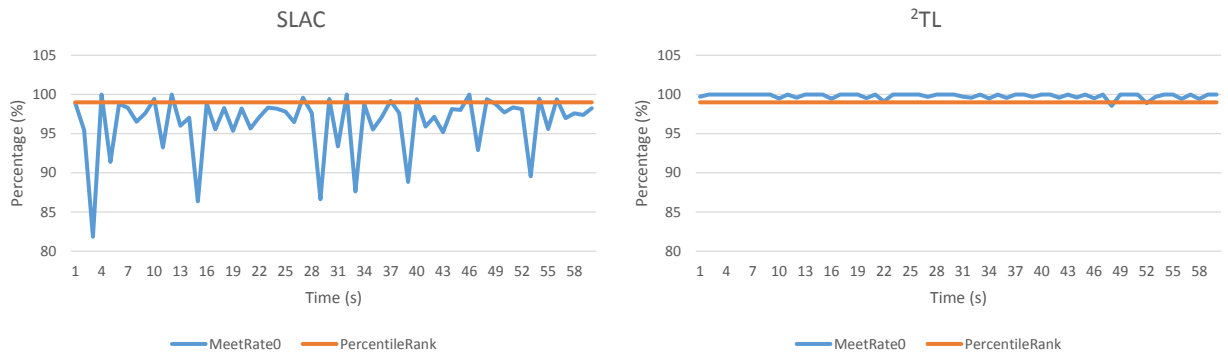


Figure 5.23: Experiments 6d and 6e: Throughput-bound Stream Throughput with FCFS. Throughput Requirements of $Stream_0$ and $Stream_1$: 1.02, and 0.51 MB/s.

of $Stream_0$ and $Stream_1$ for these experiments, we first obtain the performance capacity available to the throughput-bound streams by running the experiment with FCFS. As shown in Figure 5.23, the streams achieved similar throughput. The average throughput achieved by $Stream_0$ and $Stream_1$ is 0.98 and 1.12 MB/s, respectively. This implies that the performance capacity available to the throughput-bound streams is approximately 2.10 MB/s (4.10 blocks/ms). To evaluate the effectiveness of SLAC and ^{2}TL in proportionally allocating throughput service, the throughput targets of $Stream_0$ and $Stream_1$ are selected so that they have the ratio of 2:1. That is, the throughput targets of $Stream_0$ and $Stream_1$ are respectively set to 1.54 MB/s (3 blocks/ms) and 0.77 MB/s (1.5 block/ms) in Experiment 6d, and 1.02 MB/s (2 blocks/ms) and 0.51 MB/s (1 block/ms) in Experiment 6e. In Experiment 6d, the sum of the throughput targets (2.31 MB/s) exceeds the performance capacity available to the throughput-bound streams (2.10 MB/s), which translates to neither throughput requirement being met because of proportional allocation of throughput service by SLAC and ^{2}TL. In contrast, in Experiment 6e, the sum of the throughput targets (1.53 MB/s) does not exceed the performance capacity and, thus, both throughput requirements can be met.



(a) SLAC.

(b) ^{2}TL.

Figure 5.24: Experiment 6b: $Stream_0$'s Meet Rates with SLAC and ^{2}TL. $Stream_0$ Latency Requirement: $< 400ms, 99\% >$.

Results

The results of Experiment Set 6 are organized in three parts, those that relate to: (1) individual stream prioritization, (2) the latency requirements of $Stream_0$ and $Stream_1$, and (3) proportional sharing.

Individual Stream Prioritization: Tables 5.24 and 5.25 summarize the average meet rates of $Stream_0$ in Experiments 6a and 6b. As shown, for both experiments, the latency requirement of $Stream_0$ was not met with FCFS, and the scheduler did not give priority to $Stream_0$'s performance requirement over $Stream_1$'s. In Experiments 6a and 6b, $Stream_0$'s average meet rates are 0% and 51%, respectively, which are below its percentile rank (99%). This shows that FCFS did not give priority to $Stream_0$'s latency requirement over $Stream_1$'s; this cannot be done without careful scheduling. In Experiment 6a (insufficient performance capacity), ^{2}TL gave priority to $Stream_0$'s performance requirement. Therefore, $Stream_0$ consumed all of the storage service. However, because the performance capacity of the storage system is insufficient to meet $Stream_0$'s latency requirement, the average meet rate of $Stream_0$ is only 46%, which is below its percentile rank (99%). Therefore, the latency requirement of $Stream_0$ was not met by ^{2}TL. Similarly,

SLAC gave priority to the latency requirement of $Stream_0$. However, with SLAC, $Stream_1$ received some service and the average meet rate of $Stream_0$ is less than with ^{2}TL. This is because ^{2}TL, due to its proactive scheduling component, is more effective than SLAC in giving priority to $Stream_0$.

Experiment 6b is used to illustrate how ^{2}TL's proactive scheduling component prioritizes $Stream_0$'s latency requirement more effectively than does SLAC. In Experiment 6b, the performance capacity is sufficient to meet only the latency requirement of $Stream_0$. With SLAC, the average meet rates of $Stream_0$ and $Stream_1$ are 96% and 56%, respectively, i.e., neither latency requirement was met with SLAC. In contrast, with ^{2}TL, the average meet rates of $Stream_0$ and $Stream_1$ are 100% and 58%, respectively, i.e., $Stream_0$'s latency requirement was met by ^{2}TL, but $Stream_1$'s was not. Figure 5.24 presents $Stream_0$'s meet rates with SLAC and ^{2}TL. With SLAC, when a request burst arrived, the meet rate dropped below the percentile rank (99%). In contrast, with ^{2}TL the arrivals of request bursts did not perturb $Stream_0$'s meet rates. $Stream_0$'s minimum meet rate with SLAC and ^{2}TL is 81.86% and 98.55%, respectively. This translates to the fact that during a one-second interval up to 18.14% and 1.45% of $Stream_0$ requests missed the latency target with SLAC and ^{2}TL, respectively; that is, many more $Stream_0$ requests met the latency target with ^{2}TL. Because of its proactive scheduling component, ^{2}TL is more effective than SLAC in prioritizing service to $Stream_0$ in order to meet its latency requirement. When $Stream_0$ issued a request burst, ^{2}TL promptly increased the service allocation to $Stream_0$ to avoid violating its latency requirement. This was done by discounting service allocation to $Stream_1$. Note that $Stream_0$'s average meet rate (100%) exceeds its percentile rank (99%). This implies that, in this case, ^{2}TL overly prioritized $Stream_0$ over $Stream_1$. A plausible explanation for this is that ^{2}TL under-estimated the scheduling rate when it exclusively scheduled $Stream_0$'s requests. As future work (Chapter 6), we will conduct a parameter sensitivity study to understand how the selection of parameter values effect the performance of ^{2}TL. The results of this study may provide insights on how to develop a heuristic to guide the selection of parameter values. Although ^{2}TL overly met the latency

requirement of $Stream_0$, it performs better than SLAC in meeting the performance requirements in the workload that drives this experiment. Given the importance of latency guarantees [34, 41, 40, 20], one would rather slightly exceed the latency requirement of $Stream_0$ by 1% with ^{2}TL than miss it by 3% (with SLAC). Furthermore, even though ^{2}TL overly prioritized service to $Stream_0$, in this case, $Stream_1$'s average meet rate with ^{2}TL (58%) is not worse than with SLAC (56%). In summary, based on the results of this experiment, *given a workload consisting of two latency-bound streams, when the storage system is unable to simultaneously meet both latency requirements, ^{2}TL is more effective than SLAC in giving priority to the higher-priority latency-bound stream in order to meet its latency requirement.*

Latency Guarantee: Tables 5.24, 5.25, and 5.26 present the average meet rates of $Stream_0$ and $Stream_1$ in Experiments 6a, 6b, and 6c, respectively. With FCFS, the latency requirement of $Stream_0$ was not met in any of these three experiments. $Stream_0$'s average meet rate is 0%, 51%, and 50%, respectively. The latency requirement of $Stream_1$ was met in Experiments 6a and 6c; its average meet rate is 99% and 100%, respectively, both of which is at least as high as its percentile rank (99%). This is because the latency target of $Stream_1$ was set to be at or above its latencies with FCFS during shared access. In Experiment 6b, the latency requirement of $Stream_1$ was not met; its average meet rate is 61%. This is because, in this experiment, we assigned the same latency target to the two streams, which is below their latencies with FCFS during shared access. Since FCFS does not differentiate storage service to the streams in a workload, the streams achieve similar performance results (i.e., similar average meet rates). The results of FCFS show that, even when the performance capacity is sufficient to meet the performance requirements of some or all of the streams in a workload ($Stream_0$ in Experiment 6b, $Stream_0$ and $Stream_1$ in Experiment 6c), their performance requirements cannot be met without careful scheduling.

As shown in Table 5.25, in Experiment 6b, given sufficient performance capacity to meet the latency requirement of $Stream_0$, ^{2}TL does meet it, while SLAC does not. As mentioned previously, this is because ^{2}TL is more effective than SLAC in prioritizing the performance

requirement of $Stream_0$ over that of $Stream_1$. As shown in Table 5.26, in Experiment 6c both ^{2}TL and SLAC met the latency requirement of $Stream_0$ and did not meet that of $Stream_1$. The average meet rate of $Stream_0$ is 100% with both schedulers, which is larger than the percentile rank (99%) of $Stream_0$; the latency requirement of $Stream_0$ was overly met. However, the latency requirement of $Stream_1$ was not met by either scheduler. $Stream_1$'s average meet rates with both schedulers is 96%, which is below the percentile rank (99%) of $Stream_1$.

As mentioned in Section 4.5, when Strategy 2 is used, if the set of parameters of the moving percentile model, namely the percentile and history size, do not lead to satisfactory results, parameter tuning is necessary. We demonstrate this by comparing the experimental results of Experiment 6c presented above, which uses percentiles = 50th ($Stream_0$) and 1st ($Stream_1$), and history size = 1,000, with the results obtained using the parameters employed in Experiment 5a, i.e., percentile = 10th (for both $Stream_0$ and $Stream_1$) and history size = 10). Using the model parameters of Experiment 5a, Experiment 6c results in the average meet rates of $Stream_0$ and $Stream_1$ being 100% and 85%, respectively. In this case, using Strategy 2 results in the latency performance of $Stream_1$ being even worse than that experienced with the original set of model parameters used in Experiment 6c, i.e., 96%. There are two possible reasons for this. First, with the parameters used in Experiment 5a, ^{2}TL under-estimates the scheduling rates of $Stream_0$, which causes ^{2}TL to schedule $Stream_0$ requests earlier than necessary to meet all of their scheduling deadlines. Since the request scheduling rate in the Shim is limited by the request service rate in the storage system (Section 4.1.2) and $Stream_0$ has a higher priority, the scheduling rate of $Stream_1$ requests may not be high enough to meet the scheduling deadlines of its requests. Second, the methodology we used to obtain $Stream_0$'s Request Storage Latency Threshold in this experiment may have over-estimated the thresholds for both SLAC and ^{2}TL. Given an overly estimated Request Storage Latency Threshold, the scheduling deadlines of requests are set earlier than is necessary, resulting in higher scheduling rates. Since both schedulers prioritize the performance requirement of $Stream_0$ over that of $Stream_1$, when they are

unable to simultaneously meet the scheduling deadlines of both streams, they first meet the scheduling deadlines of the requests of $Stream_0$ before those of $Stream_1$. Therefore, $Stream_0$'s latency requirement was overly met. To validate the first reason, in Experiment 6c we set the percentiles of $Stream_0$ and $Stream_1$ to the 50th and the 1st, respectively. The percentile of $Stream_0$ does not lead to conservative estimations and, thus, avoids under-estimating the scheduling rates of $Stream_0$'s requests. The very low percentile of $Stream_1$ leads to very conservative estimations to ensure that ^{2}TL schedules $Stream_1$'s requests as fast as possible, after meeting the scheduling deadlines of $Stream_0$'s requests. Because of the fluctuation of the scheduling rates of $Stream_1$ requests, we set the history size to 1,000 (compared to 10 in Experiment 5a) to ensure that the percentile of $Stream_1$ indeed leads to conservative estimations. In this case, the average meet rate of $Stream_1$ increases to 96%. Although the latency requirement of $Stream_1$ was still not met (its percentile rank is 99%), its average meet rate increased from 85% to 96%.

This shows that when the parameters in Experiment 5a were used, the under-estimated scheduling rate of $Stream_0$ is indeed a reason for the overly met $Stream_0$ latency requirement. (Note that with the percentiles of the streams set at the 10th percentile, the history sizes of 10 and 1,000 produced similar results.) While the results in this experiment are subject to further investigation, the only remaining plausible explanation is that the methodology we used to obtain $Stream_0$'s Request Storage Latency Threshold in this experiment overly estimated the thresholds for both SLAC and ^{2}TL. Nonetheless, since both schedulers overly met the latency requirement of $Stream_0$, the problem is not related to proactive scheduling of latency-bound requests. In summary, *given a workload of two latency-bound streams, where the highest-priority stream issues bursts but the lowest-priority does not, with sufficient performance capacity of the storage system, ^{2}TL delivers performance comparable to that of SLAC in meeting the performance requirements of all of the streams. In this case, ^{2}TL's proactive scheduling component does not provide any advantage over SLAC's reactive scheduling. Otherwise, i.e., if the performance capacity of the storage system is insufficient to simultaneously meet all of the latency requirements in*

the workload, as demonstrated in Experiments 6a and 6b, ^{2}TL is more effective than SLAC in meeting the latency requirement of the highest-priority stream because of its proactive scheduling of latency-bound requests.

Proportional Service Allocation: Tables 5.27 and 5.28 present the throughput performance of $Stream_0$ and $Stream_1$ in Experiments 6d and 6e, respectively, with FCFS, SLAC and ^{2}TL. With FCFS, $Stream_0$'s throughput requirement in Experiments 6d (1.54 MB/s) and 6e (1.02 MB/s) was not met. In contrast, $Stream_1$'s throughput requirement in Experiments 6d (0.77 MB/s) and 6e (0.51 MB/s) was met. In both experiments, with FCFS storage service was almost equally allocated to $Stream_0$ and $Stream_1$ (with actual ratios of 0.47 and 0.53, respectively) rather than being proportionally allocated based on their throughput targets; the error in proportional sharing was 19%. In contrast, with SLAC and ^{2}TL, the storage service was allocated to $Stream_0$ and $Stream_1$ proportional to their throughput targets with small errors (1% and 6% in Experiments 6d and 6e, respectively). In Experiment 6d, because the storage system's performance capacity is insufficient to simultaneously meet the two throughput requirements of the workload, proportional service allocation does not lead to meeting throughput guarantees. In contrast, in Experiment 6e, because the storage system's performance capacity is sufficient to simultaneously meet the two throughput requirements of the workload, proportional service allocation leads to meeting throughput requirements and the ability to provide throughput guarantees. In summary, based on the results of Experiment 6e, the effectiveness of SLAC and ^{2}TL is similar in allocating storage service to a workload comprised of only throughput-bound streams that is proportional to their throughput targets. *Given sufficient performance capacity available to the throughput-bound streams of a workload, both schedulers are able to simultaneously meet the throughput-bound streams' throughput requirements via proportional sharing.*

Table 5.22: Experiment 6a-6c: Three Simulations driven by Synthetic Workloads of 2 Latency-bound Streams.

Experiment	**Duration**	**Performance Capacity**	**Performance Requirements**	
			$Stream_0$ Priority 1 (highest)	**$Stream_1$ Priority 2**
6a	60s	Insufficient to meet $Stream_0$ latency requirement	$< 200ms, 99\% >$	$< 500ms, 99\% >$
6b	60s	Sufficient to meet only one latency requirement	$< 400ms, 99\% >$	
6c	300s	Sufficient to meet all latency requirements	$< 400ms, 99\% >$	$< 1600ms, 99\% >$
		Burst Interval (s)	2	N/A
		Burst Size	60	
		Base Pending Requests	20	

Table 5.23: Experiment 6d-6e: Two Simulations driven by Synthetic Workloads of 2 Throughput-bound Streams.

Experiment	**Duration**	**Performance Capacity**	**Performance Requirements**	
			$Stream_0$	**$Stream_1$**
6d	60s	Insufficient to meet both throughput requirements	1.54 MB/s (3 blocks/ms)	0.77 MB/s (1.5 blocks/ms)
6e	60s	Sufficient to meet both throughput requirements	1.02 MB/s (2 blocks/ms)	0.51 MB/s (1 block/ms)

Table 5.24: Experiment 6a: Average Meet Rates of $Stream_0$ and $Stream_1$ with FCFS, SLAC and ^{2}TL.

Scheduler	**$Stream_0$**	**$Stream_1$**
FCFS	0%	99%
SLAC	43%	9%
^{2}TL	46%	0%

Table 5.25: Experiment 6b: Average Meet Rates of $Stream_0$ and $Stream_1$ with FCFS, SLAC and ^{2}TL.

Scheduler	**$Stream_0$**	**$Stream_1$**
FCFS	51%	61%
SLAC	96%	56%
^{2}TL	100%	58%

Table 5.26: Experiment 6c: Average Meet Rates of $Stream_0$ and $Stream_1$ with FCFS, SLAC and ^{2}TL.

Scheduler	**Stream$_0$**	**Stream$_1$**
FCFS	50%	100%
SLAC	100%	96%
^{2}TL	100%	96%

Table 5.27: Experiment 6d: Stream Throughput Performance with FCFS, SLAC and ^{2}TL.

	AvgThroughput$_i$ (MB/s)		**ActualRatio$_i$**		**Error$_i$**	
Scheduler	**Stream$_0$**	**Stream$_1$**	**Stream$_0$**	**Stream$_1$**	**Stream$_0$**	**Stream$_1$**
FCFS	0.98	1.12	0.47	0.53	-19%	19%
SLAC	1.34	0.69	0.66	0.34	-1%	1%
^{2}TL	1.34	0.67	0.66	0.34	-1%	1%

Table 5.28: Experiment 6e: Stream Throughput Performance with FCFS, SLAC and ^{2}TL.

	AvgThroughput$_i$ (MB/s)		**ActualRatio$_i$**		**Error$_i$**	
Scheduler	**Stream$_0$**	**Stream$_1$**	**Stream$_0$**	**Stream$_1$**	**Stream$_0$**	**Stream$_1$**
FCFS	0.98	1.12	0.47	0.53	-19%	19%
SLAC	1.44	0.56	0.72	0.28	6%	-6%
^{2}TL	1.44	0.56	0.72	0.28	6%	-6%

Chapter 6

Conclusions and Future Work

It is increasingly common for applications with I/O latency requirements and applications with I/O throughput requirements to simultaneously access a shared storage system. Given an I/O workload, which consists of the streams of I/O requests of a set of applications, the storage system, via its I/O scheduler, is expected to simultaneously meet the workload's latency and throughput requirements. In addition, it is expected to provide performance guarantees, i.e., guarantees that it will meet a workload's latency and throughput requirements. (Of course, these guarantees are provided under certain conditions.) It is difficult for most I/O schedulers to meet latency requirements with high percentiles, especially for streams that issue request bursts. This is because of the reactive nature of most schedulers.

This dissertation introduced our I/O scheduler, ^{2}TL, which is comprised of a proactive scheduling component and a proportional allocation component. Together these components allow ^{2}TL to provide both latency guarantees at high percentiles as well as throughput guarantees. During the development of ^{2}TL, another scheduler with a proactive scheduling component, named Courier, was introduced in the literature. Both schedulers continuously monitor the access characteristics of latency-bound streams and proactively adjust scheduling parameters to avoid latency requirement violations. However, ^{2}TL is different from Courier in two significant ways. First, ^{2}TL is designed to meet percentile latency requirements that are needed by today's critical applications. And, second, ^{2}TL takes into consideration disk queuing and its effect on the latency of requests.

^{2}TL is designed to simultaneously provide percentile latency guarantees through proactive scheduling and throughput guarantees (subject to certain conditions) on RAID storage systems with disk queues. In order to achieve this goal, we answered the three research

questions enumerated in Section 1.3. The three questions are enumerated, again, below together with their answers.

1. Can latency guarantees at high percentiles be provided on a storage system with disk queues through proactive scheduling?

   Answer: Yes. ^{2}TL considers request latencies due to disk queuing and proactively adjusts its scheduling parameters to ensure that at least a percentage of requests, specified by the percentile rank, meet the latency target of each latency-bound stream in a workload.

2. What is the dynamic information required for a proactive scheduling method to meet latency requirements at high percentiles?

   Answer: ^{2}TL's proactive scheduling of latency-bound requests is driven by the information of request storage latencies, request arrival rates, and request scheduling rates.

3. What is needed by the proactive scheduling method to meet throughput requirements with best effort, while not overly meeting latency requirements?

   Answer: ^{2}TL needs to be able to correctly estimate scheduling rates of latency-bound streams in a workload. If it under-estimates the scheduling rates, it may overly meet latency requirements and, therefore, unable to meet throughput requirements with best effort. Similarly, if it over-estimates the scheduling rates, it may not be able to meet the latency requirements.

^{2}TL adapts to variations in request storage latencies, proactively increasing the service allocated to the latency-bound streams in order to avoid latency requirement violations due to request bursts. To meet throughput requirements, it allocates storage service to throughput-bound streams proportional to their throughput requirements.

We evaluated the effectiveness of ^{2}TL through simulations on an enhanced version of DiskSim 4.0. Because of the functional limitations of existing schedulers, we are unable to directly compare ^{2}TL with them. Instead, we compared ^{2}TL with SLAC, a scheduler that we designed to functionally resemble ^{2}TL and embody the reactive characteristics of existing schedulers to which we want to compare ^{2}TL with. These properties of SLAC facilitate relatively fair performance comparisons with ^{2}TL that allow us to evaluate the benefits of ^{2}TL's proactive scheduling of latency-bound requests.

Experimental results show that the effectiveness of ^{2}TL in simultaneously providing latency and throughput guarantees is subject to the performance capacity of the storage system in three ways. For latency guarantees, first, the request arrival rate of a stream must not exceed its request service rate. Second, the number of the stream's pending requests cannot be too large that the end-to-end latencies of its requests exceed its latency target. For throughput guarantees, the storage system must service the requests of each throughput-bound stream at a rate at least as large as the stream's throughput requirement. Note that these conditions are not specific to ^{2}TL; they apply to all schedulers that provide performance guarantees. When all three conditions are met, ^{2}TL simultaneously meets the performance requirements of all of the streams in a workload, be they only latency-bound streams with or without request bursts, only throughput-bound streams, or a mix of both. Otherwise, if not all three conditions are met, results show that ^{2}TL is able to appropriately prioritize high-priority streams, be they all latency-bound streams or the latency-bound streams with higher priorities. ^{2}TL is able to prioritize higher-priority streams without overly meeting their latency requirements and allocate the remainder of the storage service to lower-priority streams to meet their performance requirements with best effort.

^{2}TL's proactive scheduling of requests of the latency-bound streams in a given workload provides better performance, as compared to SLAC's reactive scheduling, when there exist one or more latency-bound streams in the workload that issue request bursts. As the burstiness of the latency-bound stream(s) increases, so does the comparative performance of ^{2}TL. In an experiment that was driven by a synthetic workload of a latency-bound stream

that issues request bursts and a throughput-bound stream, ^{2}TL was able to meet the latency requirements of the workload, while SLAC did not; 99% and 95% of the requests met the stream's latency target, respectively. With an experiment using a workload of eight streams, we demonstrated that the advantage ^{2}TL's proactive scheduling over SLAC's reactive scheduling is scalable. Finally, with a real workload, we demonstrated that ^{2}TL is able to effectively meet the latency requirement of a stream generated by a real application. However, ^{2}TL's proactive scheduling of latency-bound requests does not demonstrate any advantage over reactive scheduling when latency-bound streams in a workload do not issue bursts. Also, ^{2}TL and SLAC have similar effectiveness in allocating storage service to throughput-bound streams proportional to their throughput targets. When the performance capacity available to a workload that is comprised of only throughput-bound streams is sufficient, both schedulers simultaneously meet all the throughput requirements of the workload.

The effectiveness of ^{2}TL in meeting latency requirements relies on the accurate estimation of the Exclusive Scheduling Rate of Latency-bound Requests. In Section 4.5, we proposed two strategies that ^{2}TL may use to estimate this scheduling rate: a simple strategy (Strategy 1) and a history-based strategy (Strategy 2). For the experiments driven by synthetic workloads, except Experiment 6c, the choice of prediction strategy does not lead to noticeable changes in the results. However, the history-based prediction strategy was required for both Experiments 5a, which is driven by a real workload, and 6c. Although the history-based prediction strategy may work well for any workload, we hesitate to recommend it in general because its performance depends on the values of the input parameters to the moving-percentile model and, thus, parameter tuning may be necessary.

In the future, we plan to continue this research on the following paths.

1. *Parameter Sensitivity Study*: We will investigate how the choices of ^{2}TL parameters affect its performance. The result of this investigation may provide insights that can be used to develop heuristics for setting the parameters appropriately. As shown on Table 4.4, these parameters are:

(1) the number of recently serviced requests used to deduce the Request Storage Latency Threshold; and

(2) the parameters that are used for estimating the Exclusive Scheduling Rate of Latency-bound Requests. For the simple strategy, there is only one parameter, i.e., the number of past intervals used for calculating the current scheduling rate. For the history-based strategy, the parameters are the number of recent exclusive scheduling rates and the percentile used in the moving-percentile model.

2. *Real Workload*: We will evaluate ^{2}TL with more real workloads generated with a wider set of real applications. For latency-bound applications, we plan to obtain I/O traces generated by user-interacting applications since these applications usually have latency requirements at high percentiles. For throughput-bound applications, we will use scientific applications and data-mining applications since they require high I/O throughput to sustain a high computation rate.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002.

[3] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005.

[4] S. Arunagiri, Y. Kwok, P. J. Teller, R. Portillo, and S. R. Seelam. FAIRIO: An algorithm for differentiated I/O performance. In *Proceedings of 23rd International Symposium on Computer Architecture and High Performance Computing*, pages 88–95, October 2011.

[5] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *Proceedings of 2011 ACM SIGMOD International Conference on Management of Data*, pages 1071–1080, New York, NY, USA, 2011.

[7] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of 1999 IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, 1999.

[8] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Parallel Data Laboratory, Carnegie Mellon University, May 2008.

[9] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of 22nd International Symposium on Reliable Distributed Systems*, pages 109–118, October 2003.

[10] J. Chang. I/O queue depth strategy, October 2010. http://sqlblog.com/blogs/joe_chang/archive/2010/10/18/io-queue-depth-strategy.aspx.

[11] CUBRID. Increasing database performance by query tuning, 2012. http://www.cubrid.org/query_tuning_results.

[12] J. Dean. A practical introduction to disk storage system performance, May 2010. Presented at IBM Power Systems and Storage Symposium, Wiesbaden, Germany.

[13] J. Dean. Achieving rapid response times in large online services, March 2012. Presented at Berkeley AMPLab Cloud Seminar.

[14] J. Dean and L.A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[16] S. Gopisetty, E. Butler, S. Jaquet, M. Korupolu, T.K. Nayak, R. Routray, M. Seaman, A. Singh, C.H. Tan, S. Uttamchandani, and A. Verma. Automated planners for storage provisioning and disaster recovery. *IBM Journal of Research and Development*, 52(4.5):353–365, July 2008.

[17] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *2nd Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, 2009.

[18] A. Gulati, A. Merchant, and P.J. Varman. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *Proceedings of 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 13–24, New York, NY, USA, 2007.

[19] K. Gupta, P. Sarkar, and L. Mbogo. MIRAGE: Storage provisioning in large data centers using balanced component utilizations. *SIGOPS Operating System Review*, 42(1):104–105, January 2008.

[20] J. Hamilton. The cost of latency, 2009. http://goo.gl/Qxys7J.

[21] L. Huang, G. Peng, and T. C. Chiueh. Multi-dimensional storage virtualization. In *Proceedings of 2004 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24, New York, NY, USA, 2004.

[22] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Performance Evaluation Review*, 32:37–48, June 2004.

[23] T. Kaldewey, T. Wong, R. Golding, A. Povzner, S. Brandt, and C. Maltzahn. Virtualizing disk performance. In *Proceedings of 2008 IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 319–330, 2008.

[24] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *2004 IEEE International Workshop on Quality of Service*, pages 67–74, June 2004.

[25] C. Krintz and R. Wolski. Using phase behavior in scientific application to guide Linux operating system customization. In *Proceedings of 19th IEEE International Symposium on Parallel and Distributed Processing*, pages 4–8, April 2005.

[26] M. Liberatore. Storage - UMass trace repository, 2007. http://traces.cs.umass.edu/index.php/Storage/Storage.

[27] Y. Lu, D. H. C. Du, and T. Ruwart. QoS provisioning framework for an OSD-based storage system. In *Proceedings of the 22rd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 28 – 35, April 2005.

[28] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of 2nd USENIX Conference File and Storage Technologies*, pages 131–144, Berkeley, CA, USA, 2003.

[29] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in large disk arrays. In *Proceedings of 8th ACM International Conference on Autonomic Computing*, pages 245–254, New York, NY, USA, 2011.

[30] A. L. Narasimha Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of 1st ACM International Conference on Multimedia*, pages 225–233, New York, NY, USA, 1993.

[31] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3), June 1988.

[32] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T.M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. In *Proceedings of 3rd ACM*

SIGOPS/EuroSys European Conference on Computing Systems, pages 13–25, New York, NY, USA, 2008.

[33] R. Prabhakar, S. S. Vazhkudai, Y. Kim, A. R. Butt, M. Li, and M. Kandemir. Provisioning a multi-tiered data staging area for extreme-scale machines. In *Proceedings of 31st International Conference on Distributed Computing Systems*, pages 1–12, June 2011.

[34] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search, June 2009. Presented at O′Reilly Velocity Web Performance and Operations Conference, San Jose, CA.

[35] S.R. Seelam. *Towards dynamic adaptation of I/O scheduling in commodity operating systems*. Ph.D. dissertation, El Paso, TX, USA, 2006.

[36] A. Silberschatz, P. B. Galvin, and G. Gagne. Mass-storage structure. In *Operating System Concepts*. 8th edition, Hoboken, NJ: Wiley, 2013, ch. 12, pp. 503-554.

[37] D. Skourtis, S. Kato, and S. Brandt. QBox: Guaranteeing I/O performance on black box storage systems. In *Proceedings of 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 73–84, New York, NY, USA, 2012.

[38] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of 6th USENIX Conference File and Storage Technologies*, pages 21:1–21:16, Berkeley, CA, USA, 2008.

[39] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of 5th USENIX Conference File and Storage Technologies*, pages 61–76, San Jose, CA, 2007.

[40] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proceedings of 3rd ACM Symposium on Cloud Computing*, pages 14:1–14:14, New York, NY, USA, 2012.

[41] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz. Sweet storage SLOs with Frosting. In *Proceedings of 4th USENIX Conference on Hot Topics in Cloud Computing*, pages 14–19, Berkeley, CA, USA, 2012.

[42] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of 5th USENIX Conference File and Storage Technologies*, pages 47–60, Berkeley, CA, USA, 2007.

[43] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 241–251, New York, NY, USA, 1994.

[44] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel. Storage performance virtualization via throughput and latency control. In *Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computing and Telecommunication Systems*, pages 135–142, September 2005.

[45] Q. Zhang, D. Feng, and F. Wang. Courier: Multi-dimensional QoS guarantees for the consolidated storage system. *Future Generation Computer Systems*, 37:97–107, July 2013.

[46] X. Zhang, K. Davis, and S. Jiang. QoS support for end users of I/O-intensive applications using shared storage systems. In *Proceeding of 2011 International Conference High Performance Computing, Networking, Storage and Analysis*, pages 18:1–18:12, New York, NY, USA, 2011.

Curriculum Vitae

Yipkei Kwok earned his Bachelor of Science (Honors) degree in Computer Science from Hong Kong Baptist University in 2000. He received his Master of Science degree in Computer Science in 2005 from California State University, East Bay. In 2005 he joined the doctoral program in Computer Science at The University of Texas at El Paso (UTEP).

While pursuing his degree, Yipkei worked as a teaching assistant and research assistant in the Department of Computer Science. He is currently a full-time professor in the Department of Computer Science at LeTourneau University. During his tenure at UTEP, he received a student travel grant from the IEEE International Conference on Cluster Computing in 2011. He also was awarded the UTEP Dodson Research Grant in 2013.

Yipkei presented his research at the International Conference for High Performance Computing, Networking, Storage, and Analysis in 2008, 2009, 2010, and 2011, and at the LCI International Conference on High Performance Clustered Computing in 2009. He published his research in the Proceedings of the International Symposium on Computer Architecture and High Performance Computing, where the paper garnered a Best Paper Award. An extended version of this paper was published in the International Journal of Parallel Programming.

Permanent address: 2100 South Mobberly Avenue
Longview, Texas 75602