Improving the performance of static and dynamic requests at a busy web site

Bianca Schroeder

CMU-CS-05-167

August 30, 2005

School of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

Thesis Committee:

Mor Harchol-Balter, Chair

Anastassia Ailamaki

Todd C. Mowry

Erich M. Nahum (T. J. Watson Research Center, IBM Research)

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

This research was sponsored by the National Science Foundation under grant nos. CCR-0133077 and CCR-9457766, by a generous grant from the Pittsburgh Digital Greenhouse and by an IBM graduate student fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Abstract

Running a high-volume Web site is a challenging task. Web traffic is bursty with peak request rates rising far above average rates and likely phenomena such as flash crowds and hot spots. Yet Web users are very demanding: they expect Web sites to be quickly accessible from around the world 24 hours a day, 7 days a week. Recent studies show that even a short period of slowdown or service interruption can have severe effects: it not only sends customers to the "just a click-away" competitor; it also reflects negatively on the corporate image as a whole.

The broad goal of this thesis is to improve the user-perceived performance of both static and dynamic requests at a busy Web site. By static requests we mean requests of the form "GET me a file". By dynamic requests we mean those which require the Web server to create the requested information on the fly, by accessing a database backend. The approach taken in this thesis does not require buying more hardware or any other costly system upgrades. The main idea is to schedule the existing resources better among requests so as to either improve overall mean performance or improve the performance of a small subset of high-priority requests.

The first part of the thesis presents a very simple solution for improving overall mean response times for static Web workloads by favoring those requests that are quick, or have small remaining processing requirements in accordance with the SRPT scheduling policy. Our policy is particularly effective during transient overload. The second part of this thesis focuses on the more complex dynamic Web requests. We propose and implement various approaches for providing differentiated levels of service for database-driven dynamic requests. We propose and evaluate algorithms for both scheduling of database internal resources and scheduling outside the DBMS through an external front-end. In the third part of the thesis we study the effect of the experimental model on performance

evaluation. In particular, we investigate the impact of choosing a closed versus an open system model, a question that has received little attention in the past.

.

Acknowledgements

First, I would like to thank my advisor Mor for her enthusiasm and unshakable optimism which kept me going even during those times when progress seemed slow. I also owe many thanks to my committee members Anastassia Ailamaki, Todd Mowry and Erich Nahum for their feedback and comments that helped improve this document. Nikhil Bansal, Mukesh Agrawal, and Christos Gkantsidis deserve thanks not only for their help with the initial SRPT experiments, but also for providing support during my initial frustrations with systems research. Mengzhi Wang and Minglong Shao saved me a lot of time by sharing their expertise on IBM DB2 and Shore. John Wilkes provided excellent feedback on many parts of this work, in particular his comments on SRPT scheduling for overload led to a much improved paper. Arun Iyengar and Erich Nahum gave me the opportunity to spend two summers at IBM, which not only led to some of the work in this thesis, but were also great fun. A big thank-you goes to the members of the SQALL group for listening to many practice talks and always providing great feedback. I would also like to thank Jochen Koenemann for his persistent encouragement and support during my undergrad years without which I would not have pursued computer science. Brigitte Pientka, Orna Raz, and Laurie Hiyakumoto were a great female support group at CMU, providing much needed coffee breaks and distraction from work. Ursula Egenhofer and Zeynep Marbach have been great friends who regularly reminded me that there's life outside Wean Hall. Thanks to Stratos for patiently putting up with my frayed nerves during the final stages of this work and the increasing number of cats in our household. Finally, I would like to thank my parents, who have always supported me in all my endeavors, even if my decisions didn't always make sense to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Internet and the Web

Exponential growth rates have been commonplace in computer science since its early days. Semiconductor performance, for example, is known to increase exponentially with a doubling rate of 18 months, as is disk storage capacity and RAM capacity. These trends have first been observed by Moore in 1965, just four years after the first planar integrated circuit was discovered, and still hold today. Nevertheless, even in this environment where exponential growth is a ubiquitous phenomenon, the growth and scale of the Internet and the Web has been unique and unparalleled, leading even seasoned researchers and market analysts to predict its early downfall.

The Internet started in 1969 as a single, closed network with four nodes whose purpose it was to allow scientists in different geographic locations to share centralized compute resources. Thirty-five years later, in 2004, the Internet has become an integral part in the life of millions of people worldwide who use it for purposes as diverse as communication, entertainment, and commerce. Much of the Internet growth and popularity among the general public can be contributed to the World Wide Web, which was born in 1993 with the release of the Mosaic Web browser. While at that time there were about 150 Web sites in existence, this number grew to 3000 in 1994 and to 25000 in 1995. For the second half of 1993, the Web had a doubling period of under 3 months, and even today the doubling period is still nearly 5 months

The large scale of the Internet has created unique business opportunities by putting an estimated user base of more than 800 million people within reach as potential customers. At the same time the unprecedented scale introduces huge new challenges. Popular Web sites today need to be able to process millions of requests per second, while at the same time ensuring fast and predictable service times, security, privacy and reliability, among others. Even today, more than a decade after the emergence of the Web, we are far from having solved these challenges. The oftentimes sluggish download times have earned the WWW the nick-name world-wide-wait, and the unpredictability of Web traffic often leaves even large corporate Web sites overwhelmed and unresponsive.

Below we discuss in more detail the opportunities and the challenges the Internet and the Web have brought about, before we then describe the goals of this thesis.

1.2 Opportunities and challenges

The huge scale of the Web offers new world of opportunities. One of the biggest opportunities for e-businesses is the ability to easily reach a huge number of potential customers. Taking as an example a book store, while a traditional store relies on customers that live in the same or a nearby city, an online book store can be accessed by any of the more then 800 million Internet users. Moreover, even for the smallest home-based business it is affordable to create a Web site, and make their products available to people all over the world. Popular Web sites manage to attract millions of hits per second.

In addition to providing access to a huge potential customer base, it also provides the means for running a large scale, global enterprise.

First, the Internet greatly simplifies the logistics compared to those involved in a traditional offline business. An e-business can send updates, and announcements for new products or special offers to a large customer base virtually for free by e-mail. Many products, such as software, digital music or movies, can be made available for download on a Web site, instead of packaging and shipping them to the individual client. Similarly, software upgrades and documentation can be published online to be instantly available for customers.

Second, new Internet-based forms of communication, such as video-conferencing or e-mail, in combination with easy and fast data-exchange through the Internet enable efficient collaboration among employees at different geographic locations. Resources such as compute clusters or a centralized database system can be efficiently shared by employees everywhere in the world.

Finally, management of global enterprises is simplified, since the Internet allows efficient access to data at any branch or warehouse, including those that are geographically remote. Moreover, in combination with the right tools the geographically distributed data from different branches or warehouses can be automatically integrated providing a manager with a complete picture of the company's state as a whole.

While the large scale of the Web opens up a vast array of new opportunities for businesses, it also comes with huge challenges, including for example the unpredictability of request patterns and the high level of competition among e-businesses.

One major challenge is to sustain the rapid growth in traffic, and number of connected hosts and users, while at the same time providing *fast and reliable service*. Online users are demanding and unforgiving. While for example customers in a traditional business environment take it for granted to wait in line for service, online customers demand instantaneous, predictable service on a 24/7 basis. User studies indicate that for a Web user to be satisfied, delays should not exceed 5 seconds and that delays that are longer than 10 seconds are considered intolerable.

Meeting the users' high expectations is crucial in running a successful e-business, since competition is fierce. The same features of the Web, that enable new business opportunities, such as a the large scale with a huge user base and support for global businesses, also open the door for way more competition. While the local bookstore competes only with a handful of other bookstores in the same city, an online store finds itself in competition of global scale with thousands of other stores.

The strong competition among e-businesses is aggravated by the fact that it is easy and effortless for a disappointed client to switch to a competitor: a single mouse-click is all that it takes. Moreover, providing dissatisfactory service not only results in losing the current transaction the client was requesting, but also future transactions. After one disappointing experience customers often don't come back for a second time, not only to avoid delays, but also because previously experienced high delays reflect badly on the e-business as a whole. In the customer's mind the seeming disability of the Web site to properly manage their resources to provide smooth service, rises doubt as to whether other aspects of the Web site, such as security and privacy, are managed well.

Ensuring the consistently low delays needed to keep customers satisfied is difficult. The WWW, dubbed by pundits "The World Wide Wait" frequently becomes a victim of its own popularity, when congested network links or heavy load at a Web site cause large slowdowns for users. Sharp and overwhelming surges in the traffic volume can affect individual Web sites or even the Web as a whole leading to very slow service or in the worst case complete unavailability of service. Unexpected surges in the traffic have been a problem for the Web since its early days. For example, during the announcement of the O.J. Simpson trial outcome in 1995, the Web experienced huge brownouts throughout the nation as online debates started. Six years later, during the terrorist attacks on September 11, most news sites became temporarily inaccessible. Over the past years the Web sites of several major online retailers became unavailable the day after thanksgiving due to high traffic volume.

Avoiding slowdowns and servive unavailability, e.g. by accordingly provisioning resources at a Web site, is complicated by the burstiness and unpredictability of Internet traffic. Sudden spikes in the load can either be caused by an event or an announcement that suddenly draws a large crowd (often referred to as flash crowd) or by a malicious act such as a denial of service (DOS) attack. While some events that are likely to cause unusually high traffic can be predicted ahead of time (such as sales events or promotions at an e-commerce site), others like news events or DOS attacks usually arise unexpectedly. One difficulty with sharp load surges is that, even in cases where a surge in traffic is predictable,it is difficult to provision the system accordingly in advance, since it is hard to accurately predict the magnitude of the increase.

In addition to challenges involving latency, there are also many other challenges, such as ensuring privacy and security. Before the emergence of the Internet computer systems were typically accessible to only a small group of "trusted" users. In contrast, the Internet is accessed by millions of anonymous users, potentially including ones with malicious or criminal intentions. Examples for when security became an issue include the Code Red worm who in 2001 forced the Pentagon to temporarily shut down public access to all of its Web sites, or the Mydoom worm which in 2004 crippled many Web sites, including large commercial ones. These challenges are beyond the scope of this thesis and are beautifully described in [191].

1.3 Reducing the World-Wide-Wait

Among the many challenges related to the Web, in this thesis we focus on improving the user-perceived performance of requests at a busy Web site. Since one of the most important factors in user satisfaction is the speed of the service they receive, the main metric we consider is *mean response time*. We define the response time for a Web request as the period of time between the moment users make a request and the time they receive the response in its entirety.

At a high level, executing a Web request involves the following steps, each factoring into the response time a user experiences.

1. After a client types a URI into their browser or clicks on a hyper-link, the browser uses DNS to resolve the URI into the server's IP address.

2. The browser opens a connection to the server and compiles and sends the HTTP request to the server. Sending the browser's request to the server involves several underlying protocols, in addition to HTTP. For example, the connection to the server is based on the TCP/IP transport protocol suite. The routing of the actual network packets relies on both intra-domain routing protocols (such as BGP), and inter-domain routing protocols (such as IS/IS or OSPF).

3. After the server receives and parses the request, it decides what actions need to be taken to produce the proper response. The proper actions depend on the *type* of the request: The response to a *static* request of the form "Get me a file" can be directly retrieved from the disk or a backend storage utility. The response to a *dynamic* request needs to be generated on the fly. This can involve access to an application server, which might run e.g. cgi-script or java servlet, or or it can involve access to a database backend.

4. After the server has generated the proper response, it sends the response through the network back to the client.

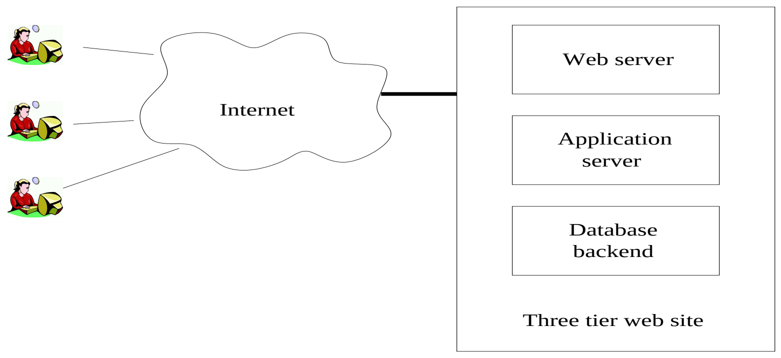


Figure 1.1: *Three tier Web architecture.*

Approaches for improving Web response times have targeted all of the above steps, resulting in solutions that apply either to the client side, the server side, or the network.

Methods for reducing delays at the browser include for example the caching of documents requested by the user, so that later requests for the same documents can be served from the cache rather than going to the origin server. In addition to just caching previously seen documents, another idea is to have browsers prefetch new documents that the user will likely request in the future. Also, since the time spent for DNS name resolution can be significant, browsers usually cache the IP addresses for the URIs they have seen and resolved previously. Caching of Web objects is also commonly applied at different points on the path between the client and the server in the form of proxy caches. Proxy caches act as intermediaries between a client and a server and usually cache documents from a larger user population, thereby increasing cache hit rates.

Efforts on improving Web performance that target the network mostly involve improvements in routing protocols. Those improvements usually affect any Internet traffic, not just Web traffic. A large volume of research on routing protocols deals with ways of improving the stability of the protocols. Instabilities can arise when a link or a router fails, e.g. due to overload, or when the routing tables of a router are misconfigured. One approach for Web sites to achieve better routing of their requests that does not rely on changes to network protocols, is to employ multi-homing: the site maintains several uplinks to different providers and dynamically picks the best one to use at any given time.

Many different techniques have been proposed for improving Web performance at the server side.

In order to efficiently service a large number of requests at the same time, changes to the design of Web server software and server operating systems have been suggested. One of the suggestions to make Web servers more scalable is to follow an event-driven approach, rather than a process-per-request approach. The Zeus Web server, for example, is based on an event-driven architecture. Changes to the server's operating system focus mostly on the network stack, e.g. by making the `select` or the `accept` system calls more scalable. When it comes to reducing server delays for dynamic requests, e.g. requests that access a database backend, a common approach is to use cached data from previous runs of the same query. Using cached versions of dynamic content involves a tradeoff between short service time and the freshness of the served data.

Some solutions require support both at the client and the server end. For example, many optimizations that have been suggested target the interaction between the HTTP and the TCP protocol and require changes to the HTTP protocol. These changes need to be supported by both the client and the server in order to become effective. One example for such an optimization is the introduction of persistent connections in HTTP 1.1. In the original version of the HTTP protocol a client was required to set up a new connection for each request. Persistent connections allow the client and the server to keep a connection open even after a request is completed, so that it can be reused for later requests.

Other optimizations that require support both at the client and the server end are the introduction of new, more efficient data formats, such as PNG (Portable Network Graphics). PNG images render more quickly on the screen and are often smaller than images in other formats, but a server can employ them only if the client browser knows how to interpret the PNG format.

1.4 Thesis overview

The broad goal of this thesis is to improve the user-perceived performance of both static and dynamic requests at a busy Web site. Our focus is on improvements that can be implemented entirely at the server side, without support of the client's browser or the network. Our solutions are very different from the ones described in the previous section in that they are based on the idea of scheduling the resources at a Web server better among requests so as to either improve overall performance, or the performance of particularly important, high-priority requests.

This thesis has three parts with different goals. The first part of the thesis presents a very simple solution for improving overall mean response times for static Web workloads. The main idea is to give preference to those requests that are quick, or have small remaining processing requirements in accordance with the SRPT scheduling policy. The second part of this thesis is motivated by the fact that requests vary in how important they are and focuses on providing differentiated levels of service for dynamic requests that rely on access to a database management system (DBMS). We propose and evaluate algorithms for both scheduling of database internal resources and scheduling outside the DBMS through an external front-end. In the third part of the thesis we study the effect of the experimental model on performance evaluation. In particular, we investigate the impact of choosing a closed versus an open system model, a question that has received little attention in the past, despite the large volume of research on workload analysis.

1.4.1 Part I: Improving mean performance of static Web requests

The first part of the thesis presents a very simple solution to improving overall mean performance at a Web server. The solution does not require buying more hardware or any other costly system upgrades and can easily be integrated into any Web server software. The main idea is to give preference to those requests that are quick, or have small remaining processing requirements in accordance with the SRPT scheduling policy. We apply this idea to *static requests*. By static requests we mean requests of the form "GET me a file"

The theoretical advantages of SRPT with respect to mean response time have been understood for a long time in the queuing-theory community. While in the single queue context SRPT's advantages may seem obvious, SRPT's applicability to real Web servers is not. First, it is not clear where or how SRPT should be implemented in a Web server. Second, it is not clear whether SRPT is still effective in the context of network protocols such as TCP; under WAN conditions including propagation delay and loss; and under client/server behavior patterns – all of which are outside of the server's control. Third, even if SRPT does reduce mean response time, it is unclear what the penalty to requests for large files will be under SRPT, especially in the case of transient overload conditions. Addressing the above three questions is the focus of this part of the thesis.

We implement SRPT in an Apache Web server running on Linux by scheduling the bandwidth on the server's uplink. This is done by modifying the order that socket buffers are drained within the kernel. We then compare the performance of the SRPT server in a thorough performance study, including many different parameters that affect Web response times, to the performance of a standard server, which we refer to as a FAIR server.

We find that the delay at a busy Web server can be greatly reduced by applying SRPT-based scheduling of the server's bandwidth. We show further that the reduction in server delay often results in a reduction in the client-perceived response time. In a LAN setting, our SRPT-based scheduling algorithm reduces mean response time significantly over the standard FAIR scheduling algorithm. In a WAN setting the improvement is still significant for high uplink loads.

We also find that SRPT significantly improves both server stability and client experience during *overload conditions*, that is times where the request arrival rate exceeds the server's capacity. Under persistent overload, the number of connections at the FAIR server grows quickly compared with the buildup of connections under SRPT, and consequently the FAIR server is also quick to reach the point where incoming SYNs are dropped. As a result, the client experience in terms of mean response time and the time until the first byte is received, is greatly improved under the SRPT server compared to the FAIR server. With respect to transient overload, we find that SRPT improves mean response times by factors of $1.5 - 8$ over the traditional FAIR scheduling, across ten different transient overload workloads. This is significant since mean response times under FAIR can get quite high, and peak response time are many-fold higher than mean response times.

Performance improvements are measured under a vast range of environmental factors including a range of WAN delays and loss rates, use of persistent connections, user behaviors, packet sizes, and Web server configurations. Results are consistent across different traces. Importantly, we find that requests for large files are *not* penalized by SRPT scheduling under transient overload. In fact, for the largest 1% of requests, the response time under SRPT is very close to that under FAIR. Furthermore these gains are achieved under no loss in byte throughput or request throughput.

The organization of Part I of the thesis is as follows. Chapter 2 reviews common approaches for improving Web performance and introduces the main idea of SRPT scheduling. Chapter 3 describes how we implement kernel-level SRPT scheduling for static Web workloads in a standard Web server. Chapter 4 evaluates the performance of our SRPT server implementation in both LAN and WAN testbeds for the case where the system load stays within the server's capacity. Chapter 5 shows how the same basic idea of SRPT scheduling can be used to improve performance during transient periods of overload. Chapter 6 concludes Part I of the thesis by summarizing the results and describing the impact of our work.

1.4.2 Part II: Providing differentiated quality of service for database workloads

Requests at a Web site vary in how time-critical they are and in how important they are to the site. It is therefore important for a Web site to be able to provide different levels of service for different classes of requests. This is particularly the case for *database-driven* dynamic Web requests, which tend to have long response times. This part of the thesis is therefore concerned with providing differentiated quality of service for OLTP (Online Transaction Processing) database workloads.

Our work proposes and evaluates two different approaches to providing differentiated services for database workloads. In the first approach, which we call *internal scheduling*, we integrate priority scheduling into the database engine, which allows us full control over all the databases resources. Since a DBMS (database management system) utilizes many different resources, including hardware resources like CPUs or disks, and software resources like the locks used for concurrency control, the important questions are which resource should be scheduling and what is the best scheduling policy for each resource.

To answer these questions we begin with a detailed resource utilization breakdown for OLTP workloads executing on a range of commercial and open-source database platforms. The workloads are created based on two standard OLTP benchmarks, TPC-C and TPC-W. We identify the bottleneck resource by dividing the lifetime of a transaction into three components: CPU, I/O, and lock wait times. We find that across a wide range of configurations, the bottleneck for TPC-C running on DBMS using 2-phase-locking (2PL) is *lock waiting*. By contrast, the bottleneck for TPC-C running on DBMS with Multiversion-Concurrency-Control (MVCC) is *I/O synchronization* for low loads, although locking can dominate at extremely high concurrency levels. For TPC-W workloads, *CPU* is always the bottleneck.

Based on the results from the bottleneck analysis we focus on lock and CPU scheduling to directly or indirectly schedule the bottleneck resource. We evaluate the effectiveness of simple prioritization, priority inheritance, and preemptive abort scheduling, and the results are broken down by workload and concurrency control mechanism. We find that for the CPU-bound workload TPC-W CPU scheduling is extremely effective in prioritizing transactions, while lock scheduling shows no effect. For TPC-C on MVCC DBMS, CPU scheduling is most effective, due to its ability to indirectly schedule the I/O bottleneck. For TPC-C on 2PL systems, non-preemptive lock scheduling with priority inheritance yields significant improvements in high priority response times. Preemptive lock scheduling is even more effective in reducing the response times of high priority transactions, but this improvement comes at an excessive penalty for low-priority transactions.

In the second approach, which we call *external scheduling*, we implement scheduling outside the DBMS, where all scheduling is done as a front end to the DBMS by temporarily delaying certain transactions. The advantage of this approach is that it doesn't depend on changes to the DBMS or knowledge of the resource utilization of the workload. Potential drawbacks are (1) a drop in throughput due to the reduced concurrency inside the DBMS; (2) an increase in overall mean response time due to head-of-line blocking; or (3) limited effectiveness in comparison with internal scheduling due to the lack of fine-grained control over DBMS resources. A large part of our work on external scheduling will therefore deal with a general feasibility study of the external scheduling approach.

We begin with an experimental study of the trade-off between maximizing scheduling control, and minimizing negative side effects such as loss in throughput. We find that, in particular for multi-resource systems, a too low MPL can dramatically hurt throughput and therefore a careful choice of the MPL is imperative. However, we also show that for all workloads there exists some feasible MPL that manages to satisfy both the conflicting goals of not hurting throughput or response time while providing sufficient prioritization differentiation between classes. We develop analytical tools for optimizing the MPL and algorithms for scheduling the external queue in order to achieve complex QoS targets.

Part II of the thesis is organized as follows. Chapter 7 surveys the area of QoS for dynamic requests and explains how our work fits in. Chapter 8 studies the problem of providing transaction priorities through scheduling of database internal resources Chapter 9 tackles the same problem of providing transaction priorities as Chapter 5, but strives to do so without touching database internals: scheduling for priorities is done in an external scheduling frontend. Chapter 10 investigates how to achieve more complex QoS goals for database transactions, such as response time goals and percentile goals. Chapter 11 summarizes the work of Part II and concludes by describing extensions and the more general applicability of the work.

1.4.3 Part III: Experimental models: Understanding their impact on performance evaluation and system design

A good understanding of a system's workloads is essential for both designing good workload generators and for designing high-performance systems. Understanding workloads involves many things, including for example (1) good understanding of workload parameters, such as the distribution of service request demands, popularity distributions, locality distributions, and correlations between requests; (2) understanding of which of the workload parameters impact important performance metrics most; and (3) understanding of the system model, i.e. whether the job arrival stream obeys a closed or open system model and the effect the choice of system model has.

While a large body of research exists that analyzes the workload parameters of Web workloads and some work has been done on quantifying the performance effects of the individual parameters, very little attention has been paid to the impact of choosing a closed versus an open system model. In this part of the thesis we use implementation and simulation experiments together with theoretic support in order to identify the major differences between open and closed system models. We compare how system load, service time variability, and the scheduling discipline affect response times under an open versus a closed system model. We also study a partly-open model that provides a more realistic representation of application behavior, and discuss the effect of parameter settings under that model.

Based on our study we derive eight simple principles that function to explain the differences in behavior of closed, open, and partly-open systems and validate these principles via trace-based simulation and real-world implementation. The more intuitive of these principles point out that response times under closed systems are typically lower than in the corresponding open system, and that as MPL increases, closed systems approach open ones. Less obviously, our principles point out that: (a) the magnitude of the difference in response times between closed and open systems can be very large, even under moderate load; (b) the convergence of closed to open as MPL grows is very slow, especially when service demand variability (C^2) is high; and (c) scheduling is far more beneficial in open systems than in closed ones. We also compare the partly-open model with the open and closed models. We illustrate the strong effect of the number of visits and C^2 on the behavior of the partly-open model, and the surprisingly weak effect of think time.

Part III of the thesis is organized as follows. Chapter 12 formally defines the open, closed, and partly-open system model and surveys how existing experimental research uses the different models. Chapter 13 provides an analytical comparison of how the system model affects response times under different system parameters and scheduling policies. Chapter 14 presents results from case studies based on simulation or implementation of real systems and their workloads. Chapter 15 summarizes our findings and the impact on experimental performance evaluation of systems.

Part I

Improving mean performance of static web requests

Chapter 2

Introduction to Part I

Web clients are very demanding. While clients in the real (non-online) world routinely wait in line for a teller at a bank or the cashier at a super-market, and accept limited opening hours as a matter of course, online users have come to expect instantaneous 24/7 service. Meeting those expectations is critical for a succesful e-business, since it takes dissatisfied customers only seconds to leave a site and take their business to a competitor. At the same time, providing consistently good service at a busy web site is a challenging task. Web traffic is bursty and hard to predict and popular web sites serve millions of requests per second.

This part of the thesis considers how we might reduce the wait a client experiences in the case of *static* web requests, of the form "Get me a file." In contrast to dynamic requests, which require the web server to generate the response on the fly, the response to static requests is pre-generated and just needs to be retrieved from the storage system (if not already in main memory). While web sites are increasingly generating dynamic content, studies from 1997-2001 [116, 136, 79] suggest that static requests still make up a significant fraction of request stream at web sites. Even logs as recent as 2004 from proxy servers suggest that 67-73% of requests are for static content [96]. Static requests are especially important when a web site experiences overload, since popular web sites often convert their dynamic content into static content during overload [125].

The performance metric we are concerned with in this part of the thesis is *response time*, which is defined to be the time from when the client sends out the SYN-packet requesting to open a connection until the client receives the last byte of the file requested. Improving the response time of static requests is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research. Below we first survey common approaches and issues in improving mean response times of static requests, before we then introduce the main idea behind our approach.

2.1 Common approaches to improving web response times

Research has investigated many different ways of improving response time, such as caching and replication, improvement of web protocols, building faster web servers, and optimizing server operating systems. We give a brief survey of this work below.

2.1.1 Replication

The basic idea behind replication is to create replicas of web objects and store them in a server that is closer to the client than the origin server. This will reduce user-perceived response times, as well as the load in the network and at the origin server. The simplest form of replication is to copy the entire contents of an origin server to a mirror site [38]. Clients either choose which mirror site to use, or alternatively, their requests may be automatically redirected to a mirror, e.g. through the use of DNS.

The most common form of replication is through caching. Caching can be employed at different places in the internet. At the client side, the web browser stores the pages requested during a web session, so that later requests to the same document can be served from the browser's cache rather than the origin server. On the network path between the client and the server, web proxies often function as caches. A caching web proxy acts as an intermediary between the client and the server. It stores web responses received in reply to clients' requests, and tries to serve new requests directly from its cache, rather than forwarding them to the origin server. Web caching is a large research area by itself and covering it in detail is beyond the scope of this thesis. In short, two of the main challenges in web caching are (i) the choice of the right cache replacement strategy [21, 13, 163, 51, 181] that determines which object to remove from the cache if the storage capacity is reached; and (2) the choice of a cache coherency protocol [128, 97, 50, 214] that ensures the freshness of the served content.

In addition to just caching previously requested documents, it has been suggested that caches also prefetch web objects that clients will likely request in the future. For a survey of web caching see for example [116] or [207].

A third form of replication is through content distribution networks (CDNs). In CDNs the content served by an origin server is selectively replicated in CDN servers. Unlike the content stored at caches, the content at a CDN server is completely controlled by the origin server (similar to a selective mirror site), hence obviating the need for consistency and freshness protocols at the CDN server. Content distribution solutions are provided by many companies, the most prominent arguably being Akamai Technologies, with tens of thousands of CDN servers placed all over the Internet. CDNs differ in how they decide when to replicate a web object and which CDN server to place it on, and how to route clients requests to the appropriate CDN server. A common approach is to use a combination of DNS-based redirection and proxy-caching techniques. A detailed survey of different solutions can be found in [187].

2.1.2 Improving Web protocols

Other work on reducing web response times focuses on optimizing the protocols involved in web transfers, particularly HTTP. HTTP was originally proposed in 1990 by Tim Berners-Lee and by 1997 made up more than 75% of the Internet backbone traffic. The protocol version at use at the time was HTTP 1.0 and it didn't take long until several weaknesses of the protocol became evident. In a collaborative effort both Web researchers and developers worked on solutions to these problems, resulting in a new version of the protocol, HTTP 1.1 [117].

With respect to reducing response times, the most important changes in HTTP 1.1 are arguably those targeting the interaction with TCP. One weakness of HTTP 1.0 is that it opens a new TCP connection for every request a client sends to a web site. This approach is not only wasteful due the overhead of setting up a new connection for every request; it also makes it hard for a connection to ever leave TCP slowstart, since individual web requests are typically short. HTTP 1.1 overcomes these problems by introducing *persistent* connections, i.e. the same TCP connection can be used for several consecutive downloads. Secondly, HTTP 1.1 introduces *pipelining* which allows a web client to simultanesously send several requests over one connection, instead of waiting for the response to one request before sending a new request. Other changes from HTTP 1.0 to HTTP 1.1 include the introduction of range requests, which allow a user to request only parts of a web resource. Range requests provide shorter latencies for web clients interested in only part of a resource, or web clients that want to resume a previously interrupted download. The work by Krishnamurthy et al. [117] gives an excellent summary of other key differences between HTTP 1.0 and HTTP 1.1.

In addition to the above protocol optimization researchers have also proposed other optimizations for HTTP that have not yet been incorporated in the current version of the protocol. Examples include *delta encoding* and policies for managing persistent HTTP connections.

Delta encoding exploits the fact that many requests are for objects that have been retrieved and cached earlier, but that have undergone small modifications by the origin server in the mean time. The idea behind delta encoding is to transfer a description of the modifications rather than the entire object to the client. Several research studies have demonstrated the potential for delta encoding to improve Web performance [150], and a standardization effort is now underway [147].

Policies for managing TCP connections are needed to decide when to keep a persistent HTTP connection open and when to close it. A TCP connection which is kept open and reused for the next HTTP request reduces overhead and response times. On the other hand, keeping a connection open consumes sockets and memory for socket-buffers. The simplest policy for managing this tradeoff (and the one most commonly implemented in practice) is to use a fixed timeout after which a connection is closed. Krishnamurthy et al. [117] propose more sophisticated policies which exploit embedded information in the HTTP request messages, e.g., senders' identities and requested URLs.

2.1.3 Web server architecture

An important component of web performance is the architectural design of the web server software. The biggest challenge in designing an efficient web server lies in efficiently supporting thousands of connections concurrently. Different web server architectures therefory vary mainly in how they implement concurrency. Below we briefly summarize the standard web architectures. A more complete treatment of the topic can be found in [102].

The first web servers are all process driven, i.e. they maintain a separate operating system process for each request. The process driven approach simplifies code development, since all responsibility for handling concurrency is relayed to the operating system. However, a major drawback of processes are that they are heavy-weight, and creating and deleting them and context-switching between them is expensive.

An alternative approach is to use threads instead of processes. Threads are more light-weight since they allow sharing of the same address space. While sharing the address space reduces the context switching cost, it has the disadvantage that a poorly programmed thread can crash the entire server. To optimize the tradeoff between the process and the thread driven design, some web servers, including for example the latest version of the popular Apache web server [19], follow a hybrid approach that utilizes both processes and threads.

A completely different form of providing concurrency is through an event driven architecture, where the entire web server is implemented in a single process. The single server process uses asynchronous I/O in combination with a notification mechanism such as the `select()` system call to manage the I/O events of many concurrent connections. The biggest advantage of the event-driven approach is that it minimizes overhead, since there's no context switching and memory overhead. As with the purely thread-based approach though, one bug in the code can crash the entire server. Moreover, implementation of an event-driven web server poses a bigger burden on the programmer, who is now solely responsible for managing the concurrency rather than relying on operating system support. Examples of web servers following an event-driven architecture include Flash [160] and Zeus [130].

In all the above approaches the web server software is implemented in user space. Some web servers such as Tux [100] are entirely implemented in kernel space thereby avoiding overheads associated with systems calls and copying of data between user and kernel space. The downside of the in-kernel implementation is that it is extremely sensitive to programming errors, since a crash of the web server code translates to a crash of the kernel and therefore the whole machine.

2.1.4 Optimizing operating systems

Much of the work on improving operating systems for internet servers focuses on improving the scalability of system calls. A study by Mogul et al. [29] in 1998 shows that both, the `select()` system call used to support non-blocking I/O, and the kernel routine that allocates a new file descriptor, do not scale well with the number of open connections at a web server. In their work they design and implement a more scalable version of those system calls. They later propose an entirely new event delivery mechanism for UNIX to overcome inherent limitations in the scalability of the `select()` system call [28]. More recent work by Brecht et al. [46] focuses on the scalability of the `accept()` system call, which is used for accepting incoming connections.

Another approach for increasing the efficiency of the networking subsystem is by avoiding interrupts and reducing the number of context switches associated with network processing. This is the path taken for example in the work on Soft timers [23]. Soft timers are a new operating system facility that provide efficient scheduling of software events at a granularity down to tens of microseconds. This allows transport protocols like TCP to efficiently perform rate-based clocking of packet transmissions and also facilitates network polling to eliminate network interrupts without sacrificing delay.

Other work concentrates on optimizing the I/O subsystem. As an example, the work by Pai et al. [161] strives to improve server performance by eliminating copying and multiple buffering of I/O data. They achieve this goal through a unified I/O buffering and caching system that allows applications, interprocess communication, the filesystem, the file cache, and the network subsystem to safely share a single physical copy of the data. Work by Iyer et al. [104] increases the efficiency which with disks process requests by increasing the chances that the disk sees multiple outstanding requests from the same server process or thread. The idea is to sometimes introduce a short, controlled delay period during which the disk scheduler waits for additional requests to arrive from the process that issued the last request.

2.1.5 Overprovisioning and server farms

In practice, a common approach for improving web performance is to simply overprovision a web site's resources. While an expensive approach, overprovisioning reduces the load at a server, thereby reducing response times and variability in response times. Overprovisioning is commonly achieved by configuring multiple servers in a server farm, or through the use of multiprocessor machines [67, 74].

2.1.6 Content adaptation

Content adaptation for improving user-experienced web performance relies on the observation that the client population accessing a web site is very diverse. For example, some clients might access the web site through a slow modem connection, while others might use a high-speed DSL connection. As a result some web objects that appear to download instanteneously for some clients, seem to have excessively long download times for other clients.

Krishnamurthy et al. [118, 119] develop techniques for a server to adapt to the individual needs of its clients. In their work, the server first characterizes a client as poor or rich. It then uses this information to alter content, alter how content is delivered, alter policy and caching decisions, or decide when to redirect the client to a mirror site. Chandra et al. [55, 56] investigates in detail techniques for altering content according to client and network characteristics, in particular the transcoding of Web images. Content adaptation plays an important role in managing system overload as we will explain in Section 2.3.

2.2 The problem of overload

A major challenge in providing low response times lies in the burstiness and unpredictability of web traffic. The peak loads experienced by a web site are often orders of magnitudes larger than the average loads. Examples for events causing sudden surges in traffic include special promotions or sales events offered by an e-commerce site, or an abrupt increase of a sites popularity, e.g. after being featured on national television or in a major newspaper. While some of these events can be predicted, it is often difficult to predict exactly the magnitude of the associated increase in traffic volume. As a result even well-provisioned web sites can experience *transient periods of overload* during which the volume of requests outstrips the system's capacity. Famous examples include the overload of the Firestone web site after a tire recall, the outage of several large e-tailers during the holiday shopping season in 2000, and overload of the official Florida election site, which became overwhelmed after the presidential election of 2000.

During overload the response times at a web site rapidly grow to a point where the site seems completely unresponsive to a client. In the worst case, overload can even lead to a crash of the entire site. The resulting loss in revenue at an e-commerce site can be huge. Not only is business lost during the actual overload period, when clients find the site unresponsive. The company might also loose future business, because customers loose trust in the company and take their business elsewhere. The importance of avoiding overload has been widely recognized and has resulted in a large body of research. We summarize the most common approaches below.

2.3 Common approaches to overload

Overload control typically consists of two steps; detecting when a system reaches overload and reacting by taking measures to reduce the load. The approaches suggested for load control vary depending on where they are implemented. Load control can be integrated into the operating system of the web server, it can be implemented at the application level, or, in the case of database-driven dynamic requests, it can be moved to the database back-end. Below we first describe different approaches for detecting overload and then survey possible reactions.

2.3.1 Detecting Overload

Approaches for overload detection at the *operating system level* fall into two categories. Approaches in the first category focus on the network stack and typically monitor the occupancy of the SYN queue or the TCP listen queue [204]. While the occupancy of these queues provides only a very limited view of the overall state of the server, the back-pressure caused by over-utilization of resources in one of the higher system layers will eventually lead to a backlog in those kernel queues.

Another approach for overload detection at the operating system level is based on measuring the utilization of server resources. One of the motivations for considering server resource utilization is that some QoS algorithms can be theoretically proven to guarantee a certain level of service, assuming that the server utilization is below some threshold [12].

When using information on the utilization level of the system in load control decisions, it is important to distinguish between high utilization caused by a short, transient burst of new traffic, versus high utilization resulting from a persistent increase in traffic that calls for load control. Cherkasova et al. [62] therefore propose to use a predictive admission control strategy that is based on the weighted moving average of the utilization level observed in previous measurement periods, rather than the instantaneous utilization level.

Methods for detecting overload at the *application level* are either based on monitoring the occupancy of application internal queues, or on developing an estimate of the work involved in processing the currently active requests.

The prior approach is taken by [41] and [40]. Both employ mechanisms for rapidly draining the TCP listen queue and managing the outstanding requests in an internal system of queues. The lengths of the internal queues can then indicate when to apply load control. Moving load control from the kernel queues to an application level queue helps to avoid TCP timeouts experienced by requests dropped in the kernel queues. Moving the load control to the application level also allows for the use of application-level information in the load control process.

Chen et al. [59] and Elnikety et al. [78] follow the latter approach, i.e. they approximate the current system load based on estimates of the work imposed by each request in progress. Chen et al. experiment with the CGI scripts included in the WebStone benchmark. They measure the CPU usage for each CGI script and use it as an estimate for the work associated with serving the corresponding dynamic request. Elnikety et al. consider java servlets that communicate with a database back-end and find that estimates of per-servlet service time converge relatively quickly. Hence the per-servlet estimates can indicate the load a given dynamic request introduces to the system. Both studies, then approximate the system load at any given time by summing up the per-request service time estimates for the requests in progress. Load control is triggered if the estimated load in the system is close to the system capacity, which is determined in off-line experiments.

Research on integrating load control into the *database server* has mostly focused on the avoidance of lock thrashing caused by data contention. A database-integrated approach offers the possibility of utilizing more detailed information on the transactions in progress. Rather than simply basing decisions on the number of transactions in progress, the load control can utilize knowledge of the state of the individual transactions (running vs blocked waiting for a lock) and the progress the transactions have made.

Choosing the right approach for implementing overload control involves several trade-offs. Integration into the operating system allows overload control by immediately dropping new requests before occupying any system resources. On the other hand, the advantage of application-level load control is that application-level information, e.g. the expected work to process a given request, can be taken into account. For complex applications like database systems the application level and the operating system have only limited information of the state of the system, potentially allowing only for coarse-grained load detection and control. However, the more fine-grained approach of integrating QoS mechanisms into the database system comes at the price of modifying a complex piece of software.

2.3.2 Reacting to Overload

After detecting an overload situation, measures must be taken to reduce the server load. One common approach is to reduce the number of requests at the server by employing admission control, i.e. selectively rejecting incoming requests. An alternative approach is to reduce the work required for each request, rather than reducing the number of requests, by serving lower quality content, that requires less resources. The premise for using content adaptation rather than admission control is that clients prefer receiving lower quality content over being denied service completely.

Dealing with overload through content adaptation

Content adaptation to control overload has first been suggested by Bhatti et al [11]. Some of the proposed mechanisms apply mostly to static content, e.g. the suggestion to replace large, high resolution images by small, low resolution images to reduce the required bandwidth. Others of their approaches can also help reduce high load that is due to dynamic content. For example, the authors propose reducing the number of local links in each page, e.g. by limiting the web site's content tree to a specific depth. A smaller number of links in a page affects user behavior in a way that tends to decrease the load on the server.

The work in [57] shows how to apply the concept of service degradation to dynamic content that is generated by accessing a back-end information systems, such as a database server. They propose to generate a less resource intensive, lower quality version of this type of content, by trading in the freshness of the served data by using cached or replicated versions of the original content.

Chen et al. implement the notion of service degradation by complementing the high-end database back-end server at a web site with a set of low-end servers that maintain replicated versions of the original data with varying update rates. If the load at the high-end server gets to high, traffic can be offloaded to the low-end servers, at the price of serving more outdated data.

Li et al. [126] propose a similar approach for database content that is replicated in data centers across the wide area network. They characterize the dependency between request response times and the frequency of invalidating cached content (and hence the freshness of the cached content) and exploit this relationship by dynamically adjusting the caching policy based on the observed response times.

A famous example for the application of service degradation in practice includes the way CNN handled the traffic surge at its site on Sept 11, 2001 [125]. CNN's homepage, which usually features a complex page design and extensive use of dynamic content, was reduced to static content with one page containing 1247 bytes of text, the logo, and one image, designed to fit into one network packet.

Dealing with overload through admission control

The simplest form of admission control would be to reject all incoming requests, either in the network stack of the operating system or at the application level, until the load drops below some threshold. There are at least two problems caused by the indiscriminate dropping of requests in this naive approach.

1. The decision to drop a request does not take into account whether the request is from a new user

or part of a session that has been lasting for a while. As a result long sessions are much more likely to experience rejected requests at some point during their lifetime than short sessions. However, it is often the long sessions at an e-commerce server that finally lead to a purchase.

2. The decision to drop a request does not take the resource requirements of the request into account. To effectively shed load through admission control one ideally wants to drop incoming requests with high resource requirements. On the other hand, despite high load, the server might want to accept a request, if this request has very small resource requirements, or requires only little service at the bottleneck resource.

The work of Cherkasova et al. [63] and Chen et al. [58] addresses the first problem by taking session characteristics into account when making admission control decisions. Simply put, if the load at the server exceeds a specified threshold, only requests that are part of an active session are accepted, while requests starting new sessions are rejected. In practice, this approach can be implemented by using cookies to distinguish whether an incoming request starts a new session or is part of a session in progress.

The work in [78] and [59] bases admission control decisions on estimates of the service requirements of incoming requests and estimates of the server capacity (determined as described above). The system load at any given time is computed as the sum of the service requirements of all requests in progress. Elnikety et al. admit a new request to the system only if adding its service requirement to the current load does not increase the load beyond the server capacity. Requests that would increase the server load beyond its capacity are stored in a backup queue. Only once the backup queue fills up, are requests dropped.

Orthogonal to the above approaches, Welsh et al. [209] propose a whole new design paradigm for architecting internet services with better load control, that they call SEDA (Staged-event-driven-architecture). In SEDA internet services are decomposed into a set of event-driven stages connected with request queues. Each stage can control the rate at which to admit requests from its incoming request queue and decide which requests to drop in the case of excessive load. This approach allows fine-grained admission control. Moreover, combating overload can be focused on those requests that actually lead to a bottleneck (since requests that never enter an overloaded stage are not affected by the admission control).

Work for load control in database backends is mostly concerned with avoiding thrashing due to data contention. Database internal methods reduce data contention not only by employing admission control, but also by canceling transactions that are already in progress. The conditions for triggering admission control and the canceling of transactions depends on the state of the transactions in progress. One option is to trigger admission control and transaction cancellation once the ratio of the number of locks held by all transactions to the number of locks held by active transactions exceeds a critical threshold [146]. Another possibility is to apply admission control and cancel an existing transaction, if more than half of all transactions are blocked waiting for a lock, after already having acquired a large fraction of the locks they require for their execution [53]. In both cases, the transaction to be canceled is one that is blocked waiting for a lock and at the same time is blocking other transactions.

2.4 Our approach for improving web performance and overload behavior

Our idea is very different from the above approaches and is based on connection scheduling. Traditionally, requests at a web server are time-shared: the web server proportions its resources fairly among those requests ready to receive service. We call this scheduling policy **FAIR** scheduling. We propose, instead, *unfair scheduling*, in which priority is given to requests for short files, or those requests with short remaining file size, in accordance with the well-known scheduling algorithm preemptive Shortest-Remaining-Processing-Time-first (**SRPT**). It is well-known from queueing theory that SRPT scheduling minimizes queueing time [182]. Allowing short requests to preempt long requests is desirable because forcing long requests to wait behind short requests results in much lower mean response time than the situation where short requests must wait behind long requests. Our expectation is that using SRPT scheduling of requests at the server will reduce the queueing time at the server, and therefore the total response time.

An additional motivation for applying SRPT scheduling at a web server are its benefits during periods of overload. Most of the performance problems a web server experiences during periods of overload are due to the rapidly growing number of requests at the server. In a FAIR server the fair shares of service that each request receives are eventually getting so small, that no request makes any significant progress. Intuitively, SRPT can be viewed as a greedy strategy to minimize the number of requests at the server by always working on the request that is closest to completion. In fact, queueing theory proves that SRPT scheduling minimizes the number of outstanding jobs at a server [182].

Despite the obvious advantages of SRPT scheduling with respect to mean response time, applications have shied away from using this policy for two reasons: First SRPT requires knowing the time to service the request. We find that in the case of static requests the size of the requested file is a good indicator of the service time. Second, there is the fear that SRPT "starves" requests for large files [35], [190] (p. 410), [185] (p. 162). For example, Bender et. al. [35] reject the idea of using SRPT scheduling in web servers because they prove that SRPT will cause large files to have an arbitrarily high *max slowdown*. However, that paper assumes a worst-case adversarial arrival sequence of web requests. A primary goal of this thesis is to investigate whether this fear is valid in the case of web servers serving typical web workloads. We will give special consideration to the case of transient periods of overload where starvation becomes a particularly big concern.

In Chapter 3 we explain how we implement SRPT in a web server serving static requests. In Chapter 4 we evaluate our new SRPT server in a LAN and a WAN environment during normal operation, i.e. when the system load is less than one. Chapter 5 investigates how SRPT can help during transient periods of overload. In Chapter 6 we describe the general impact of our work and conclude.

Chapter 3

Implementing SRPT in a web server

While SRPT scheduling is a well-defined concept in queueing theory, it is not immediately clear what SRPT means in the context of a web server. Unlike the theoretical M/G/1 queueing model, a web server is a complex system with many resources, including the CPU, memory, disks, and network bandwidth.

Previous approaches at implementing size-based scheduling for web servers mostly focus on scheduling at the application level, thereby avoiding the complexity of scheduling individual resources. The authors in [71] design a specialized Web server which allows them to control the order in which `read()` and `write()` calls are made, but does not allow any control over the low-level scheduling which occurs inside the kernel, below the application layer (e.g., control over the order in which socket buffers are drained). Via the experimental Web server, the authors are able to improve mean response time by a factor of up to 4, for some ranges of uplink load, but the improvement comes at a price: *a drop in throughput by a factor of almost 2.* The explanation, which the authors offer repeatedly, is that scheduling at the application level does not provide fine enough control over the order in which packets enter the network. In order to obtain enough control over scheduling, the authors are forced to limit the throughput of requests.

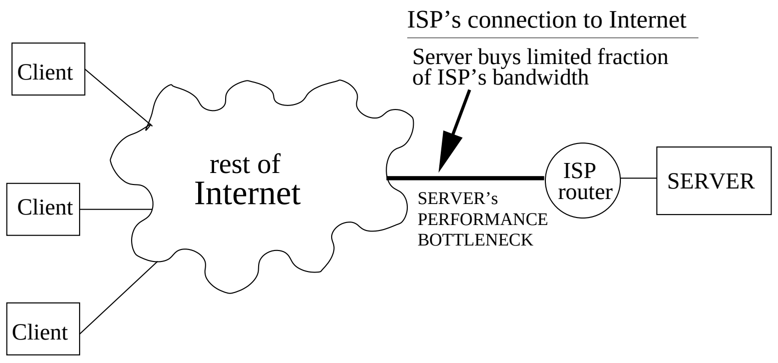
Almeida et. al. [16] modify the Apache web server to include a scheduler process which determines the order in which requests are fed to the web server. Again this modification is at the application level and therefore does not have any control over what the OS does when servicing the requests.

However, the authors add some kernel-level support by simply setting the operating system priority of the process which handles a request in accordance with the priority of the request. Observe that setting the priority of a process only allows very coarse-grained control over the scheduling of the process, as pointed out in the paper. The approaches in this paper are good starting points, but the results show that more fine-grained implementation work is needed. For example, in their experiments, the high-priority requests only benefit by 20% and the low priority requests suffer by up to 200%.

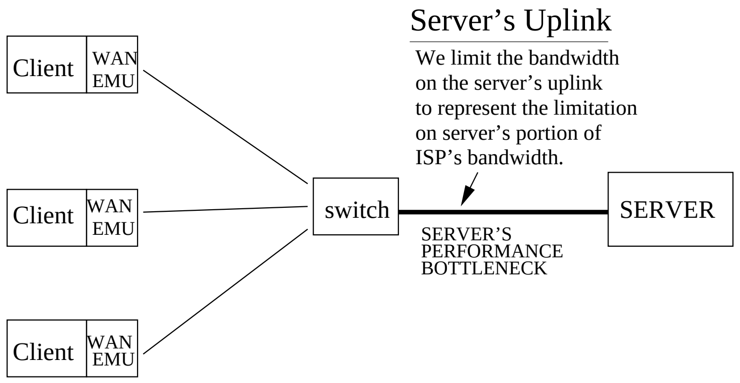
The goal of this chapter is to implement fine-grained connection scheduling at the kernel level, with absolute control over packets entering the network. Such an implementation will yield greater performance improvements than previous work and will not come at the cost of any decrease in throughput.

To implement effective connection scheduling at the kernel level, we need to apply scheduling directly at the *bottleneck* resource of the web server, that is the resource which experiences high load first. The three contenders are: the CPU; the disk to memory bandwidth; and the server's limited fraction of its ISP's bandwidth. On a site consisting primarily of *static content*, a common performance bottleneck is the limited bandwidth which the server has bought from its ISP [142, 66, 134]. Even a fairly modest server can completely saturate a T3 connection or 100Mbps Fast Ethernet connection. Also, buying more bandwidth from the ISP is typically relatively more costly than upgrading other system components like memory or CPU.

In our work we model the limited bandwidth that the server has purchased from its ISP by placing a limitation on the server's uplink, as shown in Figure 3.1. In all our experiments (using both a 10Mbps and 100 Mbps uplink, and 256 MB of RAM, and running various trace-based workloads) the bandwidth on the server's uplink is always the bottleneck resource. The system load is therefore defined in terms of the load on the server's uplink, which we refer to as the **uplink load**. For example, if the web server has a 100 Mbps uplink and the average amount of data requested by the clients is 80 Mbps, then the uplink load is 0.8. Although we assume in our work that the bottleneck resource is the limited bandwidth that the server has purchased from its ISP, the main ideas can also be adapted for alternative bottleneck resources.


ISP's connection to Internet
Server buys limited fraction of ISP's bandwidth
Client
Client
Client
rest of Internet
SERVER's PERFORMANCE BOTTLENECK
ISP router
SERVER

(a)


Server's Uplink
We limit the bandwidth on the server's uplink to represent the limitation on server's portion of ISP's bandwidth.
Client
WAN EMU
Client
WAN EMU
Client
WAN EMU
switch
SERVER
SERVER'S PERFORMANCE BOTTLENECK

(b)

Figure 3.1: *(a) Server's bottleneck is the limited fraction of bandwidth that it has purchased from its ISP. (b) How our implementation setup models this bottleneck by limiting the server's uplink bandwidth.*

The focus in the remainder of Part I of this thesis is on how to schedule the server's uplink bandwidth, and the performance effects of this scheduling. To schedule the server's uplink bandwidth, we need to apply the SRPT algorithm at the level of the network. Our approach is to control the order in which the server's socket buffers are drained. Recall that for each (non-persistent) request a connection is established between the client and the web server. Corresponding to each connection, there is a socket buffer on the web server end into which the web server writes the contents of the requested file. Traditionally, the different socket buffers are drained in Round-Robin Order, with equal turns being given to each eligible connection, where an eligible connection is one that has packets waiting to be sent and for which TCP congestion control allows packets to be sent. Thus each eligible connection receives a fair share of the bandwidth of the server's uplink. We instead propose to give priority to those sockets corresponding to connections requesting small files or where the *remaining data* required by the request is small. Throughout, we use the Linux OS.

Our goal is to compare FAIR scheduling with SRPT scheduling. These are defined as follows:

FAIR scheduling This uses standard Linux (fair-share draining of socket buffers) with an unmodified web server.

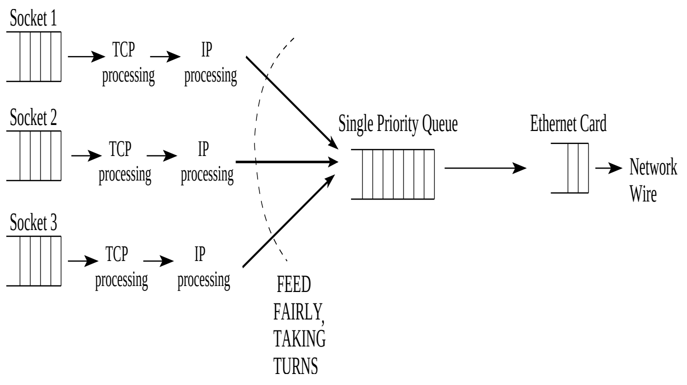
SRPT scheduling This uses modified Linux (SRPT-based draining of socket buffers) with the web server modified only to update socket priorities.

We experiment with two different web servers: the common Apache server [19], and the Flash web server [160], which is known for speed. Our clients make requests according to a web trace, which specifies both the time the request is made and the size of the file requested. Experiments are also repeated using requests generated by a web workload generator[32].

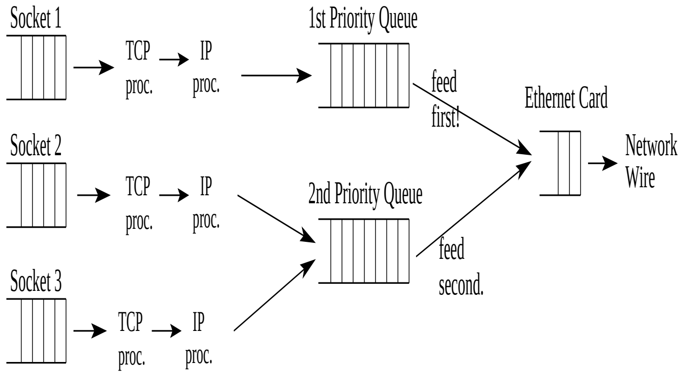
In Section 3.1 we explain how socket draining works in standard Linux, and we describe how to achieve priority queueing in Linux (versions 2.2 and above). Section 3.2 describes the implementation end at the web server and also deals with the algorithmic issues such as how to choose good *priority classes* and the setting and updating of priorities. Furthermore we consider the problem that for small file requests, a large portion of the time to service the request is spent *before* the size of the requested file is even known, and we find a solution for this problem.

3.1 Achieving priority queueing in Linux

Figure 3.2(left) shows data flow in standard Linux.



(a) Standard Linux - FAIR



(b) Modified Linux - SRPT

Figure 3.2: *(Left) Data flow in standard Linux. The important thing to observe is that there is a* single *priority queue into which all ready connections drain fairly. (Right) Linux with priority queueing. There are several priority queues, and queue i is serviced only if all of queues 0 through $i-1$ are empty.*

There is a socket buffer corresponding to each connection. Data streaming into each socket buffer is encapsulated into packets which obtain TCP headers and IP headers. Throughout this processing, the packet streams corresponding to each connection are kept separate. Finally, there is a *single*[1] "priority queue" (*transmit queue*), into which *all* streams feed. All eligible streams (eligible via TCP

[1]The queue actually consists of 3 priority queues, a.k.a. bands. By default, however, all packets are queued to the same band.

congestion control) take *equal* turns draining into the priority queue. Although the Linux kernel does not explicitly enforce fairness, we find that under conditions where clients are otherwise equal, TCP governs the flows so that they share fairly on short time scales. This single "priority queue," can get as long as 100 packets. Packets leaving this queue drain into a short Ethernet card queue and out to the network.

To implement SRPT we need more priority levels. To do this, we first build the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queueing, and the Prio Pseudoscheduler. Then we use the `tc`[17] user space tool to switch the Ethernet card queue from the default 3-band queue to the 16-band prio queue. Further information about the support for differentiated services and various queueing policies in Linux can be found in [90, 167, 17, 18].

Figure 3.2(right) shows the flow of data in Linux after the above modification: The processing is the same until the packets reach the priority queue. Instead of a single priority queue (transmit queue), there are 16 priority queues. These are called bands and they range in number from 0 to 15, where band 15 has lowest priority and band 0 has highest priority. All the connections of priority i feed fairly into the ith priority queue. The priority queues then feed in a prioritized fashion into the Ethernet Card queue. Priority queue i is only allowed to flow if priority queues 0 through $i-1$ are all empty.

A note on experimenting with the above implementation of priority queueing: Consider an experiment where each connection is assigned to one of two priorities. We have found that when the number of simultaneous connections is very low, the bandwidth is not actually split such that the first priority connections get 100% of the bandwidth and the second priority connections get 0% of the bandwidth. The reason is that with very few connections, the first priority connections are unable to fully utilize the link, and thus the second priority connections get a turn to run. However, when the number of simultaneous connections is higher (e.g., above 10), this is not a problem, and the first priority connections get 100% of the bandwidth. In all our experiments, we have hundreds of simultaneous connections and the above implementation of priority queueing works perfectly.

3.2 Modifications to web server and algorithmic issues in approximating SRPT

The modified Linux kernel provides a mechanism for prioritized queueing. In our implementation, the Apache web server uses this mechanisms to implement the SRPT-based scheduling policy. Specifically, after determining the size of the requested file, Apache sets the priority of the corresponding socket by calling `setsockopt`. As Apache sends the file, the remaining size of the requested file decreases. When the remaining size falls below the threshold for the current priority class, Apache updates the socket priority with another call to `setsockopt`.

3.2.1 Implementation Design Choices

Our implementation places the responsibility for prioritizing connections on the web server code. There are two potential problems with this approach. These are the overhead of the system calls to modify priorities, and the need to modify server code.

The issue of system call overhead is mitigated by the limited number of `setsockopt` calls which must be made. Typically only one call is made per connection. Even in the worst case, we make only as many `setsockopt` calls as there are priority classes (6 in our experiments) per connection.

A clean way to handle the changing of priorities totally within the kernel would be to enhance the `sendfile` system call to set priorities based on the remaining file size. We do not pursue this approach here as neither our version of Apache (1.3.14) nor Flash uses `sendfile`.

3.2.2 Size cutoffs

SRPT assumes infinite precision in ranking the remaining processing requirements of requests. In practice, we are limited to only 16 priority bands (16).

Based on experimentation, we have come up with some *rules-of-thumb* for partitioning the requests into priority classes which apply to the heavy-tailed web workloads. The reader not familiar with heavy-tailed workloads will benefit by first reading Section 4.4.

Denoting the cutoffs by $x_1 < x_2 < \ldots < x_n$:

- The lowest size cutoff x_1 should be such that about 50% of requests have size smaller than x_1. These requests comprise so little total load in a heavy-tailed workload that there's no point in separating them.
- The highest cutoff x_n needs to be low enough that the largest (approx.) .5% – 1% of the requests have size $> x_n$. This is necessary to prevent the largest requests from starving.
- The middle cutoffs are far less important. A logarithmic spacing works well.

In all experiments, we use only 6 priority classes to approximate SRPT. Using more improved performance only slightly.

3.2.3 Priority to SYNACKs

At this point one subtle problem remains: For requests for small files, a large portion of time to service the request is spent during the connection setup phase, *before* the size of the requested file is even known. The packets sent during the connection startup might therefore end up waiting in long queues, making connection startup very costly. For requests for small files, a long startup time is especially detrimental to response time. It is therefore important that the SYNACK be isolated from other traffic. Linux sends SYNACKs, to priority band 0. It is important when assigning priority bands to requests that we:

1. Never assign any sockets to priority band 0.
2. Make all priority band assignments to bands of *lower* priority than band 0, so that SYNACKs always have highest priority.

Observe that giving highest priority to the SYNACKs doesn't negatively impact the performance of requests since the SYNACKs themselves make up only a negligible fraction of the total uplink load.

Giving priority to SYNACKs is important in SRPT because without it the benefit that SRPT gives to small file requests is not noticeable. Later in Section 4.4.1 we consider whether the FAIR policy might also benefit by giving priority to SYNACKs, but find the improvement to FAIR to be less significant.

Assigning highest priority to SYNACKs has the negative effect of increasing the system's vulnerability to SYN-flooding attacks and severe overload. One possible solution to this problem is to take advantage of the fact that SYNACKs have their own priority band under SRPT and to monitor the rate of arrivals into this priority band. If the rate suddenly increases, indicating a potential SYN-flooding attack, we could drop the priority of SYNACKs, thus returning to a system closer to FAIR.

3.2.4 The final algorithm

Our SRPT-like algorithm is thus as follows:

1. When a request arrives, it is given a socket with priority 0 (highest priority). This allows SYNACKs to travel quickly as explained in Section 3.2.3.

2. After the size of the file requested is determined (by looking at the URL of the file), the priority of the corresponding socket is reset based on the size of the requested file, as shown in the table below.

Priority	Size (Kbytes)
0 (highest)	-
1	$\leq$ 1K
2	1K - 2K
3	2K - 5K
4	5K-20K
5	20K - 50K
6 (lowest)	$>$ 50K

3. As the remaining size of the requested file diminishes, the priority of the socket is dynamically updated to reflect the remaining size.

In the next chapter we evaluate our new SRPT server in a LAN and a WAN environment during normal operation, i.e. when the system load is less than one.

Chapter 4

Scheduling to improve mean performance for system load < 1

In this chapter we experimentally evaluate the performance of the FAIR and the SRPT server described in the previous chapter. We measure performance in terms of the following metrics:

- *Mean response time.* The response time of a request is the time from when the client submits the request until the client receives the last byte of the requested file.
- *Mean slowdown.* The slowdown metric attempts to capture the idea that clients are willing to tolerate long response times for requests for large files and yet expect short response times for small file requests. The slowdown of a request is therefore its response time divided by the time it would require if it were the sole request in the system. Slowdown is also commonly known as *normalized response time* or *stretch factor* and has been widely used [35, 174, 76, 91].
- *Mean response time as a function of the size of the file requested.* This metric indicates whether requests for large files are being treated *unfairly* under SRPT as compared with FAIR-share scheduling.

Experiments are executed first in a LAN, so as to isolate the reduction in queueing time at the server. Response time in a LAN is dominated by queueing delay at the server and TCP effects. Experiments are next repeated in a WAN environment. The WAN allows us to incorporate the effects of propagation delay, network loss, and congestion in understanding more fully the client experience. WAN experiments are executed both using a WAN emulator and by using geographically-dispersed client machines.

Synopsis of results obtained for a LAN:

- SRPT-based scheduling decreases mean response time in a LAN by a factor of 3 – 8 for uplink loads greater than 0.5.
- SRPT-based scheduling helps requests for small files tremendously, while negligibly penalizing requests for large files. Under an uplink load of 0.8, 80% of the requests improve by a factor of 10 under SRPT-based scheduling. Only the request for the largest file suffers an increase in mean response time under SRPT-based scheduling (by a factor of only 1.2).
- There is no negative effect on network throughput or CPU utilization from using SRPT as compared with FAIR.

Synopsis of results obtained for a WAN:

- While propagation delay and loss diminish the improvement of SRPT over FAIR, loss has a much greater effect.
- For an RTT of 100ms, under an uplink load of 0.9, SRPT's improvement over FAIR is still a factor of 2.
- Network loss diminishes the improvement of SRPT over FAIR further. Under high network loss (10%), SRPT's improvement over FAIR is only 25% under an uplink load of 0.9.
- Unfairness to requests for large files remains negligible or non-existent under WAN conditions.

The outline of this chapter is as follows: In Section 4.1 we describe the experimental setup, including the machine configuration, and how the workload is generated. Section 4.2 and Section 4.3 describe experimental results obtained in the LAN environment and the WAN environment respectively. Section 4.4 provides an in depth look at *why* SRPT scheduling improves over FAIR scheduling. Section 4.5 summarizes our results.

4.1 Experimental setup

4.1.1 Machine configuration

Our experimental setup involves six machines connected by a 10/100 Ethernet switch. Each machine has an Intel Pentium III 700 MHz processor and 256 MB RAM, and runs Linux 2.2.16. One of the machines is designated as the server and runs Apache 1.3.14. The other five machines act as web clients and generate requests as described below. Below we show results for both the case where the server uplink bandwidth is 10 Mbps and the case where the server uplink bandwidth is 100 Mbps. For the case of the 10 Mbps bandwidth, at any moment in time there may be a couple hundred simultaneous connections at the server. For the case of 100 Mbps bandwidth the number of simultaneous connections is in the thousands.

In order to experiment with WAN effects we setup a WAN emulation at each client machine. The two most frequently used tools for WAN emulation are probably NistNet [156] and Dummynet [173]. NistNet is a separate package available for Linux which can drop, delay or bandwidth-limit *incoming* packets. Dummynet applies delays and drops to both incoming and outgoing packets, hence allowing the user to create symmetric losses and delays. Since Dummynet is currently available for FreeBSD only we implement Dummynet functionality in form of a separate module for the Linux kernel. More precisely, we changed the `ip_rcv()` and the `ip_output()` function in the TCP-IP stack to intercept in- and out-going packets to create losses and delays.

In order to delay packets, we use the `timeout()` facility to schedule transmission of delayed packets. We recompile the kernel with `HZ=1000` to get a finer-grained millisecond timer resolution.

In order to drop packets we use an independent, uniform random loss model (as in Dummynet) which can be configured to a specified probability.

In addition to WAN experiments based on the LAN testbed with added WAN emulation, we will also experiment with physically geographically-dispersed client machines.

4.1.2 The workload

To properly evaluate the performance of a server we need to understand how clients generate requests which drive the web server. The process by which clients generate requests is typically modeled either as an *open system* or as a *closed system*, as shown in Figure 4.1.

In an *open system* each user is assumed to visit the web site just once. The user requests a file from the web site, waits to receive the file, and then leaves. A request completion *does not* trigger a new request. A new request is only triggered by a new user arrival.

In a *closed system* model, it is assumed that there is some fixed number of users. These users sit at the same web site forever. Each user repeats these 2 steps, indefinitely: (i) request a file, (ii) receive the file. In a closed system, a new request is only triggered by the completion of a previous request.

When using a trace to generate requests under an open system model, the requests are generated at the times indicated by the trace, where interarrival times have been scaled to create the appropriate test load. When using a trace to generate requests under a closed system model, the arrival times of requests in the trace are ignored.

Neither the open system model nor the closed system model is entirely realistic. Throughout Part I of this thesis we use the open system model. We also present results, however, for a different model which we call the *partly-open model*, which captures properties of both the open and closed models. Under the partly-open model, each user is assumed to visit a web site, make k requests for files at the web site, and then leave the web site. The k requests are made consecutively, with each request completion triggering the next request. We find that the results of the partly-open model are largely similar to those for an open model, see Figure 4.5.

In all the figures below, unless otherwise stated, we assume an open system model.

We use a trace-based workload consisting of 1-day from the 1998 World Soccer Cup, obtained from the Internet Traffic Archive [101]. The trace contains 4.5 million HTTP requests, virtually all of which are *static*. In our experiments, we use the trace to specify *the time* the client makes the request and the *size in bytes* of the file requested.

The entire 1 day trace contains requests for approximately 5000 different files. Given the mean

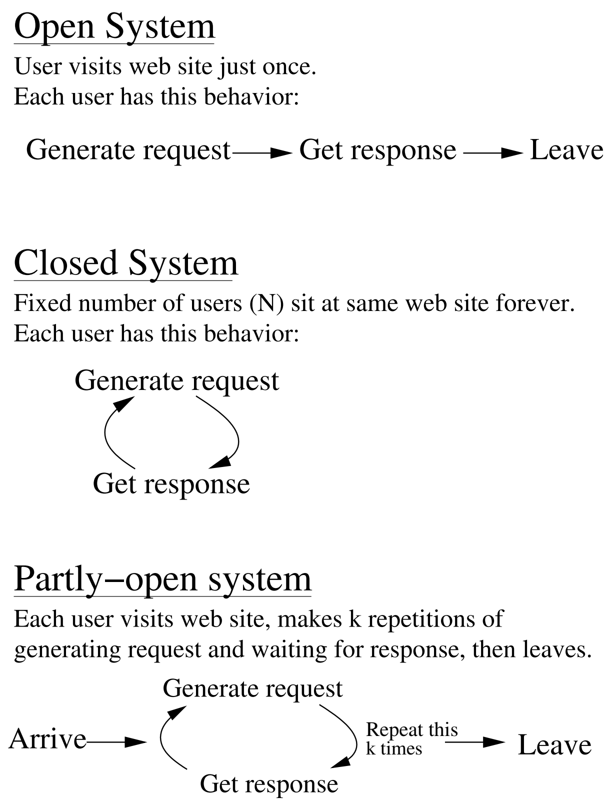

Open System
User visits web site just once.
Each user has this behavior:
Generate request
Get response
Leave
Closed System
Fixed number of users (N) sit at same web site forever.
Each user has this behavior:
Generate request
Get response
Partly−open system
Each user visits web site, makes k repetitions of
generating request and waiting for response, then leaves.
Arrive
Generate request
Get response
Repeat this k times
Leave

Figure 4.1: *Three models for how the requests to a web server are generated. In all cases, every individual request averages into the mean response time.*

file size of 5K, it is clear why all files fit within main memory and why the disk is not a bottleneck. Each experiment was run using a busy hour of the trace (10:00 a.m. to 11:00 a.m.). This hour consisted of about 1 million requests.

Some statistics about our trace workload follow: The mean file size requested is 5K bytes. The minimum size file requested is a 41 byte file. The maximum size file requested is a 2.02 MB file. The distribution of the file sizes requested has been analyzed in earlier work [20] and found to be *heavy-tailed*: while the body of the distribution can be reasonablywell modeled by a log-normal distribution, the tail is best fit by a Pareto distribution with an α-parameter less than 1.5. We find that the largest $< 3\%$ of the requests make up $> 50\%$ of the total load (in terms of total number of bytes), exhibiting a strong heavy-tailed property. 50% of files have size less than 1K bytes, and 90% of files have size less than 9.3K bytes. Figure 4.2 shows the full complementary cumulative distribution of requested file sizes.

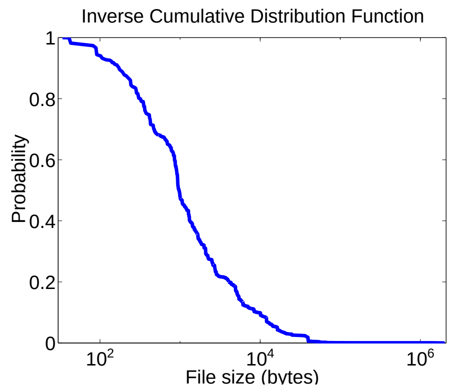


Figure 4.2: *Inverse Cumulative Distribution Function, $\bar{F}(x)$, for the trace-based workload.* $\bar{F}(x) = \Pr\{$*size of the file requested* $> x\}$

We also repeated all experiments using a *web workload generator*, `Surge` [32] to generate the requests at the client machines. The `Surge` workload is created to be statistically representative of the file sizes at a web site, the sizes of files requested from a web site, the popularity of files requested, and more. We modified `Surge` simply to make it an open system. We have included in the associated technical report [93] the same set of results for the `Surge` workload. The `Surge` workload had a higher mean size of file requested (7K, rather than 5K), however in all other respects was statistically very similar to our trace-based workload. Not surprisingly, the factor improvement of SRPT over FAIR is very similar under the `Surge` and trace-based workloads. To be precise, all the response times for both FAIR and for SRPT are 50% higher under the `Surge` workload, and therefore the factor improvement is the same.

4.1.3 Generating requests at client machines

In our experiments, we use `sclient` [26] for creating connections at the client machines. The original version of `sclient` makes requests for a certain file in periodic intervals. We modify `sclient` to read in traces and make the requests according to the arrival times and file names given in the trace. As in `sclient`, we assume a new connection for each request, i.e. no persistent connections. The effect of persistent connections will be evaluated in detail in Chapter 5.

To create a particular uplink load, say 0.8, we simply scale the interarrival times in the trace's request sequence until the average number of bits requested per second is 8Mb/sec. We validate the uplink load both analytically and via measurement.

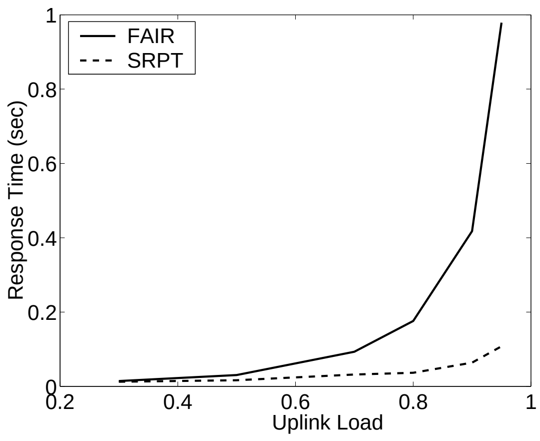
4.2 Experimental results – LAN

"Is it possible to reduce the expected response time of every request at a web server, simply by changing the order in which we schedule the requests?"

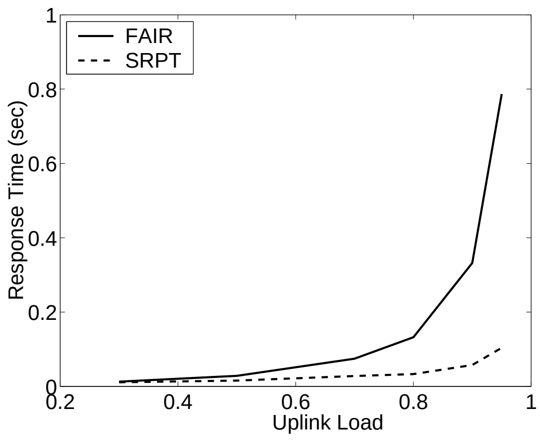
Before presenting the results of our experiments, we make some important comments.

- In all of our experiments the server's uplink bandwidth was the bottleneck resource. CPU utilization during our experiments remained below 5% for all the 10 Mbps experiments and below 80% for the 100 Mbps experiments, even for uplink load 0.95.
- The measured throughput and bandwidth utilization under the experiments with SRPT scheduling is *identical* to that under the same experiments with FAIR scheduling. The same exact set of requests complete under SRPT scheduling and under FAIR scheduling.
- There is no additional CPU overhead involved in SRPT scheduling as compared with FAIR scheduling. Recall that the overhead due to updating priorities of sockets is insignificant, given the small number of priority classes that we use.

Figure 4.3 shows the mean response time under SRPT scheduling as compared with the traditional FAIR scheduling as a function of uplink load. Figure 4.3(a) assumes that requests are generated according to an open model and Figure 4.3(b) assumes a partly-open system model, where each user generates $k = 5$ requests. Results are very similar in (a) and (b). For lower uplink loads the mean response times are similar under FAIR and SRPT. However for uplink loads > 0.5, the mean response time is a factor of $3 - 8$ lower under SRPT scheduling.



(a) Open system model



(b) Partly-open system model

Figure 4.3: *Mean response time under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with uplink bandwidth 10 Mbps. (a) Assumes open system model (b) Assumes partly-open system model with* $k = 5$ *request-iteration cycles per user.*

The performance results are even more dramatic for mean slowdown. Figure 4.4 shows the mean slowdown under SRPT scheduling as compared with the traditional FAIR scheduling as a function of load. For lower loads the slowdowns are the same under the two scheduling policies. For uplink load 0.5, the mean slowdown improves by a factor of 4 under SRPT over FAIR. Under an uplink load of 0.9, mean slowdown improves by a factor of 16.

Looking at the partly-open system model more closely we observe that mean response times are almost identical, regardless of the value of k. Figure 4.5 shows the performance of FAIR under a range of k values: $k = 1$, $k = 5$, and $k = 50$. It turns out that SRPT is even less sensitive to the choice of k. [1]

[1]Having experimented with many k values, we find the following subtle trend as we increase k: When we initially increase k, we find that response times drop a bit. The reason is that by synchronizing the times at which requests

Throughout we show results for the open system model, however we have verified that all these results are almost identical under the partly-open system model with $k = 5$.

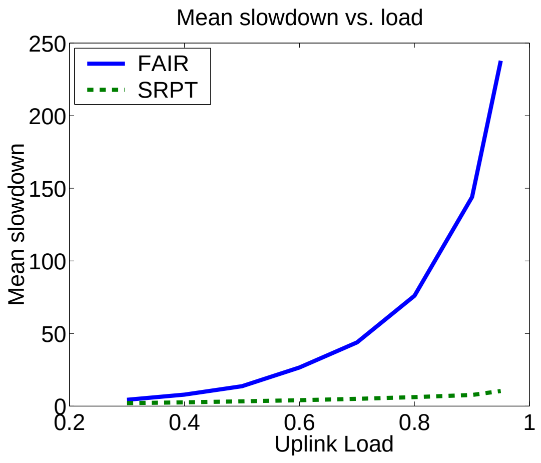


Figure 4.4: *Mean slowdown under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with uplink bandwidth 10 Mbps. This slowdown plot corresponds to the experiment in Figure 4.3(a) and looks identical for the experiment shown in Figure 4.3(b).*

We conclude this section by once again considering the improvement of SRPT over FAIR, but this time in the case of a 100 Mbps uplink. Results are shown in Figure 4.6 under the Flash web server. We see that SRPT performs 5 times better than FAIR for an uplink load of 0.8, (i.e., 80 Mbps requested through a 100Mbps uplink). This is comparable to the factor improvement achieved in the case of the 10Mbps uplink under the Apache server, Figure 4.3(a).

The significant improvements of SRPT over FAIR observed in this section are easily explained. The time-sharing behavior of FAIR causes small requests to be delayed in part by requests for large files, whereas SRPT allows requests for small files to jump ahead of requests for large files. Since most requests are for small files, most requests see an order of magnitude improvement under SRPT. Another way to think of this is that SRPT is an opportunistic algorithm which schedules requests so as to minimize the number of outstanding requests in the system (it always works on those requests with the least remaining work to be done). By minimizing the number of outstanding requests in

are generated, so that they are generated only when a previous request completes, we do a better job of evening the burstiness in the number of connections at the server. As k increases further, however, the partly-open system starts to look like a closed system with zero think time. This has the effect of creating a near-one uplink load at all times, which causes response times to go up.

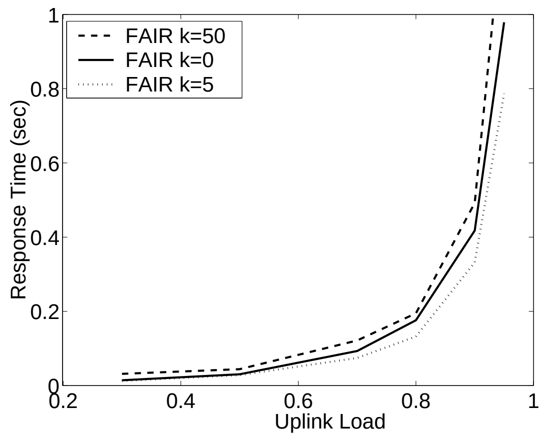


Figure 4.5: *Performance of FAIR shown for a partly-open system model, using trace-based workload in a LAN with uplink bandwidth 10 Mbps, where* $k = 1$*,* $k = 5$*, and* $k = 50$*.*

the system, Little's Law [127] tells us that SRPT also minimizes the mean response time.

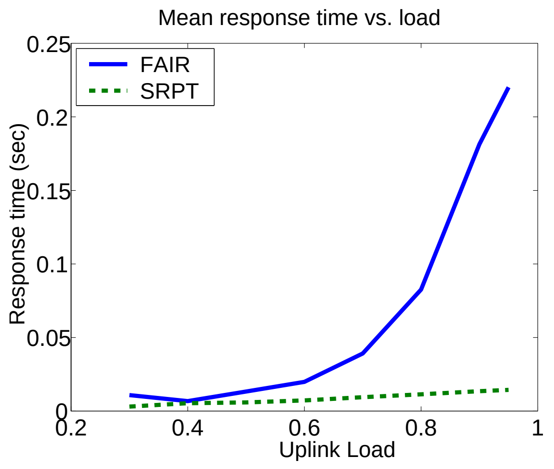


Figure 4.6: *Mean response time under SRPT versus FAIR as a function of uplink load, under trace-based workload, in LAN environment with server uplink bandwidth 100Mb/sec.*

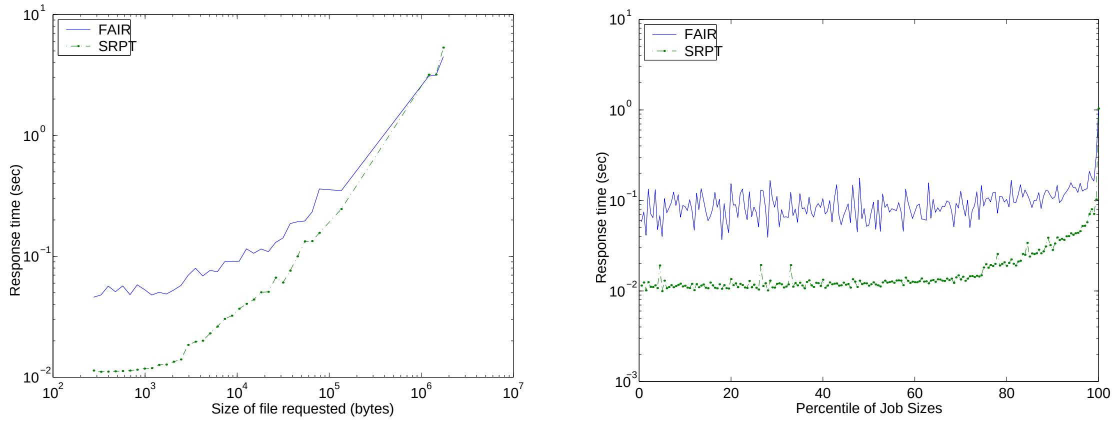
4.2.1 Performance of requests for large files under SRPT in LAN

The important question is whether the significant improvements in mean response time come at the price of significant unfairness to large requests. We answer this question for both the open system model and the partly-open system model. We first look at the case of 10 Mbps uplink and then at the case of 100 Mbps uplink.

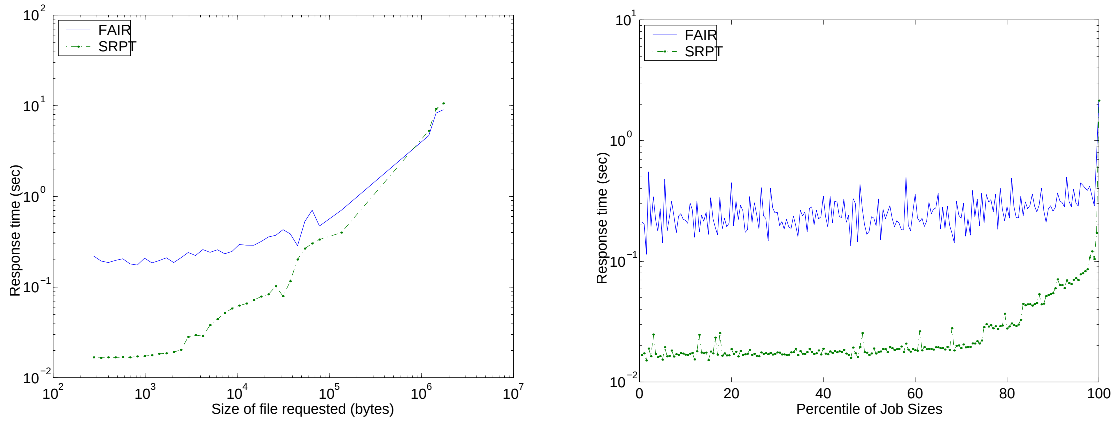
Figure 4.7 shows the mean response time as a function of the size of the requested file, in the case where the uplink load is 0.6, 0.8, and 0.9 and the bandwidth on the server's uplink is 10 Mbps. In the left column of Figure 4.7, the sizes of the files requested have been grouped into 60 bins, and the mean response time for each bin is shown in the graph. The 60 bins are determined so that each bin spans an interval $[x, 1.2x]$. Note that the last bin actually contains only requests for the very biggest file. Observe that requests for small files perform far better under SRPT scheduling as compared with FAIR scheduling, while requests for large files, those > 1 MB, perform only negligibly worse under SRPT as compared with FAIR scheduling. For example, under uplink load of 0.8 (see Figure 4.7(b)) SRPT scheduling improves the mean response times of requests for small files by a factor of close to 10, while the mean response time for the very largest size request only goes up by a factor of 1.2.

Note that the above plots give equal emphasis to small and large files. As requests for small files are much more frequent, these plots are not a good measure of the improvement offered by SRPT. To fairly assess the improvement, the right column of Figure 4.7, presents the mean response time as a function of the percentile of the requested file size distribution, in increments of half of one percent (i.e. 200 percentile buckets). From this graph, it is clear that at least 99.5% of the requests benefit under SRPT scheduling. In fact, requests for the smallest 80% of files benefit by a factor of 10, and all requests outside of the top 1% benefit by a factor of > 5. For lower uplink loads, the difference in mean response time between SRPT and FAIR scheduling decreases, and the unfairness to requests for large files becomes practically nonexistent. For higher uplink loads, the difference in mean response time between SRPT and FAIR scheduling becomes greater, and the unfairness to requests for large files also increases. Even for the highest uplink load tested though (.95), there are only 500 requests (out of the 1 million requests) which complete later under SRPT as compared with FAIR. These requests are so large however, that the effect on their slowdown is negligible.

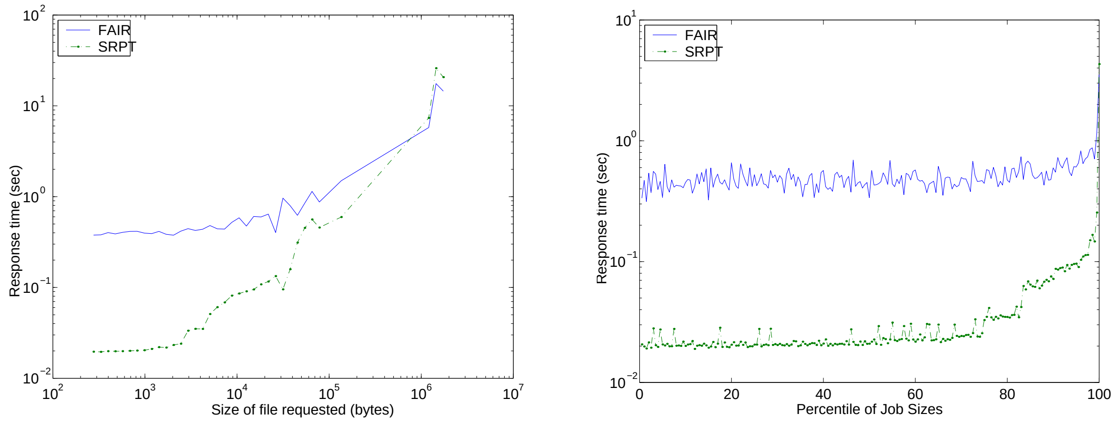
Results for the partly-open system model are similar to those in Figure 4.7, with slightly more penalty to the requests for large files, but still hardly noticeable penalty. For the case of $k = 5$, with uplink load $\rho = 0.8$, the mean response time for the largest 1% of requested files is still lower under SRPT (1.09 seconds under SRPT as compared with 1.12 seconds under FAIR). The request for the



(a) uplink load = .6



(b) uplink load = .8



(c) uplink load = .9

Figure 4.7: *Mean response time as a function of the size of the requested file under trace-based workload, shown for a range of uplink loads (corresponds to Figure 4.3(a)). The left column shows the mean response time as a function of the size of the file requested. The right column shows the mean response time as a function of the percentile of the requested file size distribution.*

very largest file has a mean response time of 9.5 seconds under SRPT versus 8.0 seconds under FAIR.

For the 100 Mb/sec experiments *all requests*, large and small, preferred SRPT scheduling in expectation under all uplink loads tested.

4.2.2 SRPT with only two priorities

Our SRPT algorithm is only a rough approximation of true SRPT since we use only 6 priority classes. An interesting question is how much benefit one could get with only 2 priority classes. That is, each request would simply being of high priority or low priority.

To explore the performance of SRPT with only two priority classes, we define high-priority requests as those corresponding to the smallest 50% of files and low-priority requests as those corresponding to the largest 50% of files. The cutoff file size falls at 1K. We find that this simple algorithm results in a factor of 2.5 improvement in mean response time and a factor of 5 improvement in mean slowdown over FAIR. We also find that *all requests*, of either priority, have lower expected response times under SRPT than under FAIR using this simple algorithm.

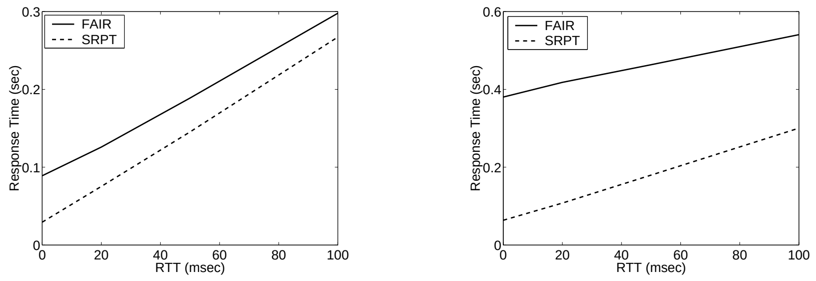
4.3 Experimental results – WAN

To understand the effect of network congestion, loss, and propagation delay in comparing SRPT and FAIR, we also conduct WAN experiments. We perform two types of WAN experiments: (i) experiments using our LAN setup together with the WAN emulator (Section 4.3.1) and (ii) experiments using physically geographically-dispersed client machines (Section 4.3.2). Throughout this section we use an uplink bandwidth of 10 Mbps.

4.3.1 WAN emulator experiments

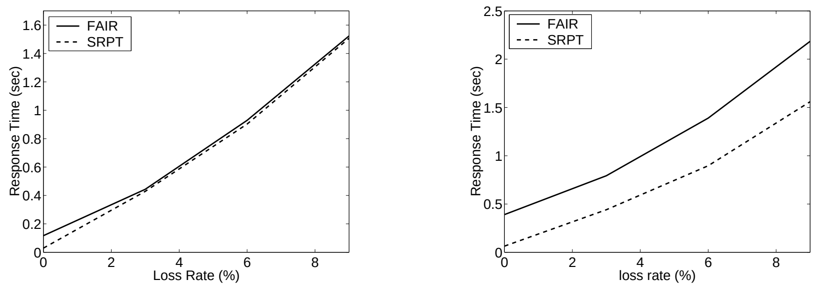
The experimental setup for the experiments in this section is identical to that used for the LAN experiments except that the WAN emulator functionality is now included in each client machine.

Figure 4.8 shows the effect of increasing the round-trip propagation delay (RTT) from 0 ms to 100 ms for FAIR and SRPT in the case of uplink load 0.7 and uplink load 0.9. Adding WAN delays increases response times by a constant additive factor on the order of a few RTTs for both FAIR



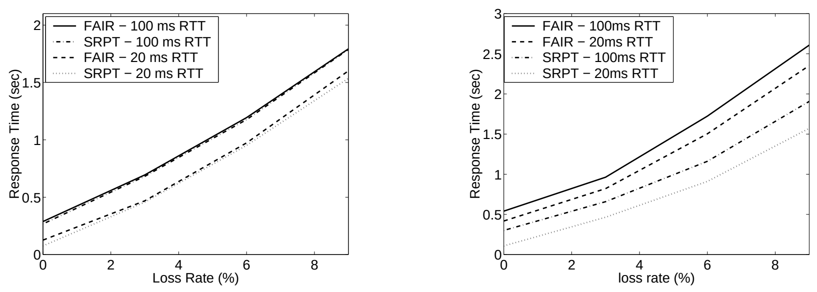
(a) uplink load = 0.7 (b) uplink load = 0.9.

Figure 4.8: *Effect on SRPT and FAIR of increasing RTT from 0 ms to 100 ms.*



(a) uplink load = 0.7 (b) uplink load = 0.9.

Figure 4.9: *Effect on SRPT and FAIR of increasing loss from 0% to 10%.*



(a) uplink load = 0.7 (b) uplink load = 0.9.

Figure 4.10: *Effect on SRPT and FAIR of increasing loss and delay.*

and SRPT. The effect is that the relative improvement of SRPT over FAIR drops. Under uplink load $\rho = 0.9$, SRPT's improvement over FAIR drops from a factor of 4 when the RTT is 0 ms to a factor of 2 when the RTT is 100 ms. Under uplink load $\rho = 0.7$, the factor improvement of SRPT over FAIR drops from a factor of 2 to only 15%.

With respect to unfairness, we find that any unfairness to requests for large files decreases as the RTT is increased. The reason is obvious – any existing unfairness to requests for large files is mitigated by the additive increase in delay imposed on both FAIR and SRPT.

Figure 4.9 assumes that the RTT is 0 ms and shows the effect of increasing the network loss from 0% to 10% under both FAIR and SRPT. Increasing loss has a more pronounced effect than increasing the RTT. We observe that the response times don't grow linearly in the loss rate. This is to be expected since TCPs throughput is inversely proportional to the square root of the loss. Under uplink load $\rho = 0.9$, SRPT's improvement over FAIR drops from a factor of 4 when loss is 0% to a factor of 25% when loss is 10%. Under uplink load $\rho = 0.7$, loss beyond 2% virtually eliminates any improvement of SRPT over FAIR.

With respect to unfairness, we find that loss slightly increases the unfairness to the request for the largest file under SRPT. The request for the largest file performs 1.1 times worse under 3% loss, but 1.5 times worse under loss rates up to 10%. Nevertheless, even in a highly lossy environment, the mean response time of requests for files in the top 1%-tile is still higher under FAIR as compared to SRPT.

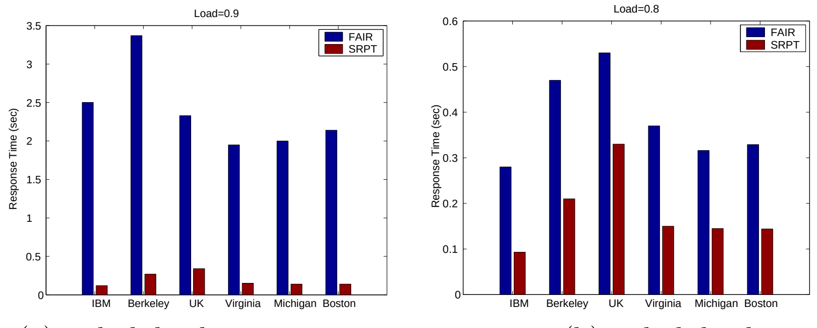
Finally Figure 4.10 combines loss and delay. Since the effect of loss dwarfs the effect of propagation delay, the results are similar to those in Figure 4.9 with loss only.

4.3.2 Geographically-dispersed WAN experiments

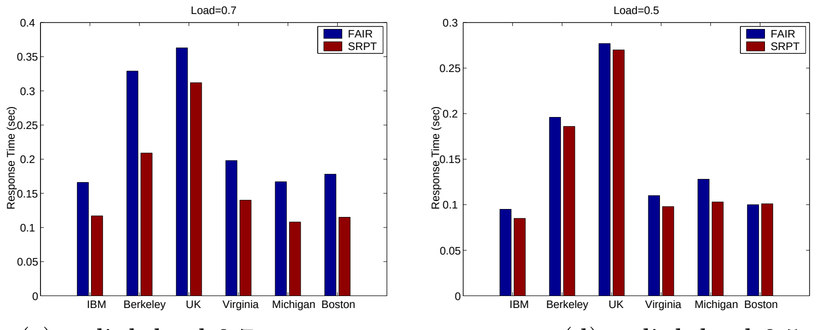
We now repeat the WAN experiments using physically geographically-dispersed client machines. The experimental setup is again the same as that used for the LAN except that this time the client machines are located at varying distances from the server. The table below shows the location of each client machine, indicated by its RTT from the server machine.[2]

Location	Avg. RTT
IBM, New York	20ms
Univ. Berkeley	55ms
UK	90-100ms
Univ. Virginia	25ms
Univ. Michigan	20ms
Boston Univ.	22ms

Unfortunately, we were only able to get accounts for Internet2 machines (schools and some research labs). The limitation in exploring only an Internet2 network is that loss and congestion may be unrealistically low.



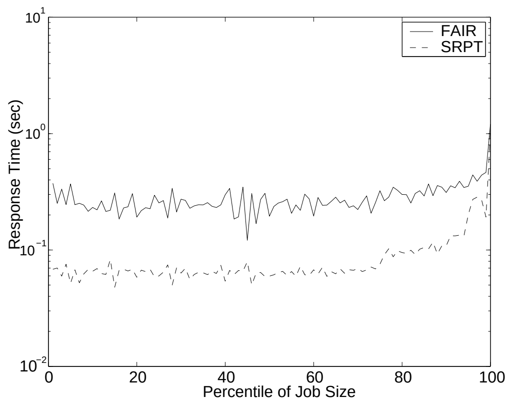
(a) uplink load 0.9 (b) uplink load 0.8



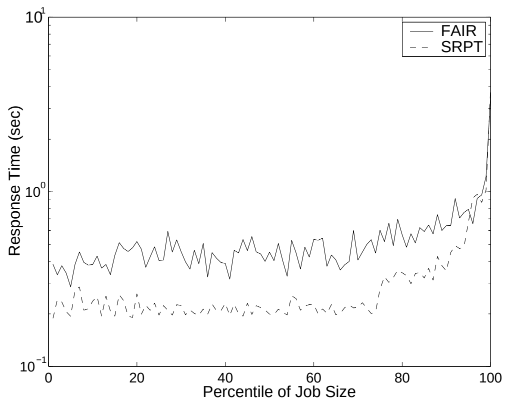
(c) uplink load 0.7 (d) uplink load 0.5

Figure 4.11: *Mean response time under SRPT versus FAIR in a WAN under uplink load (a) 0.9, (b) 0.8, (c) 0.7, and (d) 0.5.*

[2]The measured bandwidth available at these client sites ranged from 1 Mbps, at Boston University, to 8 Mbps, at IBM. Experiments were instrumented such that the bandwidth at the client site would not be a bottleneck.



(a) IBM clients



(b) UK clients

Figure 4.12: *Response time as a percentile of the size of the requested file under SRPT scheduling versus traditional FAIR scheduling at uplink load 0.8, measured for (a) the IBM host and (b) the UK host.*

Figure 4.11 shows the mean response time as a function of uplink load for each of the six hosts. The improvement in mean response time of SRPT over FAIR is a factor of 8–20 for high uplink load (0.9) and only about 1.1 for lower uplink load (0.5).

Figures 4.12(a) and 4.12(b) show the mean response time of a request as function of the percentile the size of the requested file, at an uplink load of 0.8, for the hosts at IBM and UK respectively. It turns out that *all* requests have higher mean response time under FAIR, as compared with SRPT. For the largest file, the mean response time is almost the same under SRPT and FAIR. The reason for the lack of unfairness is the same as that pointed out in the WAN emulation experiments for the case of significant RTT, but near-zero loss.

We next compare the numbers in Figure 4.11 with those obtained using the WAN emulation. For the case of uplink load 0.5, 0.7, and 0.8, the values of response time in Figure 4.11 are comparable with those obtained using the WAN emulator with propagation delay, but near-zero loss (compare with Figure 4.8).

Observe that the response times under uplink load 0.9 in Figure 4.11 are much higher than those for the WAN emulator for the case of FAIR but not for SRPT. The reason is that the WAN environment creates some variance in the uplink load. Thus an average uplink load of 0.9 translates to fluctuations ranging from 0.75 to 1.05, which means that there are moments of *transient overload*.[3] Transient overload affects FAIR far worse than SRPT because the buildup in number of requests at the server during overload is so much greater under FAIR than under SRPT. Transient overload even occasionally results in a full SYN queue under FAIR in our experiments. This means that incoming SYNs may be dropped, resulting in a timeout and retransmit. In the LAN environment where uplink load can be better controlled, we never experience SYN drops in our experiments (although SYN drops might occur in alternative setups where the CPU is the bottleneck resource).

The trends shown in Figures 4.11 and 4.12 are in agreement with the WAN emulator experiments. To summarize: (i) The improvement of SRPT over FAIR is higher at higher uplink loads; (ii) The improvement of SRPT over FAIR is diminished for far away clients; (iii) The unfairness to requests for large files under SRPT becomes non-existent as propagation delay is increased.

4.4 Why does SRPT work?

In this section we look in more detail at where SRPT's performance gains come from and we explain why there is no starvation of requests for large files.

4.4.1 Where do mean gains come from?

The high-level argument has been given before: SRPT is an opportunistic algorithm which schedules requests so as to minimize the number of outstanding requests in the system (it always works on those requests with the least remaining work to be done). By minimizing the number of outstanding requests in the system, Little's Law tells us that SRPT also minimizes the mean response time: Little's Law [127] states that the mean number of requests in the system equals the product of the average arrival rate and the mean response time. In fact our measurements show that when the load is 0.7 the number of open connections is 3 times higher under FAIR than under SRPT. At load 0.9, this number jumps to 5 times higher. This corresponds to the improvement in mean response time of SRPT over FAIR.

[3]When we say that there is a transient load of 1.05, we mean that during some 1-second intervals there may be 10.5 Mbits of data *requested* where the uplink bandwidth is only 10 Mbps.

Mathematically, the improvement of SRPT over FAIR scheduling with respect to mean response time has been derived for an M/G/1 queue in [30].

At an implementation level, while our implementation of SRPT, described in Section 3.1 is not an exact implementation of the SRPT algorithm, it still has the desirable properties of the SRPT algorithm: requests for small files (or those with small remaining time) are separated from requests for large files and have priority over requests for large files. Note that our implementation does not interact illegally with the TCP protocol in any way: scheduling is only applied to those connections which are ready to send via TCP's congestion control algorithm.

The above discussion shows that one reason that SRPT improves over FAIR with respect to mean response times is because it allows small file requests to avoid time-sharing with large file requests. We now explore two other potential reasons for the improvement of SRPT over FAIR and eliminate both.

One potential reason for the improvement of SRPT over FAIR might be that FAIR causes the SYN queue to overflow (because of the rapid buildup in number of connections) while SRPT does not. Recall that if the web server's SYN queue fills up, new connection requests will experience expensive timeouts (on the order of 3 seconds). Our measurements show that the SYN queue is in fact significantly fuller under FAIR than under SRPT for high uplink loads, as expected, since SRPT minimizes the number of outstanding requests. However, in all of our experiments except one WAN experiment, the SYN queue *never* fills up under FAIR or SRPT.

Yet another potential reason for SRPT's performance gains over FAIR is that by having multiple priority queues SRPT is essentially getting to use more buffering, as compared with the single transmit queue of FAIR (see Figure 3.2). It is possible that there could be an advantage to having more buffering inside the kernel, since under high uplink loads we have observed some packet loss (5%) within the kernel at the transmit queue under FAIR, but not under SRPT. To see whether SRPT is obtaining an unfair advantage, we experimented with increasing the length limit for the transmit queue under FAIR from 100 to 500, and then to 700, entirely eliminating the losses. This helped just a little — reducing mean response time from about 400ms to 350ms under FAIR. Still, performance was nowhere near that of SRPT.

4.4.2 Why are requests for large files not hurt?

It has been suspected by many that SRPT is a very unfair scheduling policy for requests for large files. The above results have shown that this suspicion is false for web workloads. It is easy to see why SRPT should provide huge performance benefits for the requests for small files, which get priority over all other requests. In this section we describe briefly why the requests for large files also benefit under SRPT, *in the case of workloads with a heavy-tailed property.*

In general a heavy-tailed distribution is one for which

$$\Pr\{X > x\} \sim x^{-\alpha},$$

where $0 < \alpha < 2$. A set of file sizes following a heavy-tailed distribution has some distinctive properties:

1. Infinite variance (and if $\alpha \leq 1$, infinite mean). (In practice, variance is not really infinite, but simply very high, since there is a finite maximum requested file size).

2. The property that a tiny fraction (usually $< 1\%$) of the very longest requests comprise over half of the total uplink load. We refer to this important property as the **heavy-tailed property**.

The lower the parameter α, the more variable the distribution, and the more pronounced is the heavy-tailed property, *i.e.* the smaller the fraction of requests for large files that comprise half the uplink load.

The sizes of requested files have been shown to often follow a heavy-tailed distribution [70, 72]. Our traces have strong heavy-tailed properties. (In our trace the largest $< 3\%$ of the requests make up $> 50\%$ of the total uplink load.)

Consider a workload where the sizes of the files requested exhibit the *heavy-tailed property.* Now consider a request for a file in the 99%-tile of the requested file size distribution. This request will actually do much better under SRPT scheduling than under FAIR scheduling. The reason is that, under SRPT, this request only competes against 50% of the uplink load (the remaining 50% of the uplink load is made up of requests for the top 1%-tile of files) whereas it competes against 100% of the uplink load under FAIR scheduling. The same argument could be made for a requested file in the 99.5%-tile of the file size distribution.

However, it is not obvious what happens to a request in the 100%-tile of the requested file size distribution (i.e. the largest possible file). It turns out that, provided the uplink load is not too close to 1, the request in the 100%-tile will quickly see an idle period, during which it can run. As soon as the request gets a chance to run, it will quickly become a request in the 99.5%-tile, at which time it will clearly prefer SRPT. For a mathematical formalization of the above argument, in the case of an M/G/1 queue, we refer the reader to [30].

Despite our understanding of the above theoretical result, we were nevertheless still surprised to find that results in practice matched those in theory – i.e., there was little if any unfairness to large requests. It is understandable that in practice there should be more unfairness to requests for large files since requests for large files pay some additional penalty for moving between priority queues.

4.5 Summary

In this chapter we have demonstrated that the delay at a busy web server can be greatly reduced by SRPT-based scheduling of the bandwidth that the server has purchased from its ISP. We show further that the reduction in server delay often results in a reduction in the client-perceived response time.

In a LAN setting, our SRPT-based scheduling algorithm reduces mean response time significantly over the standard FAIR scheduling algorithm. In a WAN setting the improvement is still significant for very high uplink loads, but is far less significant at moderate uplink loads.

Surprisingly, this improvement comes at no cost to requests for large files, which are hardly penalized, or not at all penalized. Furthermore these gains are achieved under no loss in byte throughput or request throughput.

Chapter 5

Scheduling to improve overload performance

"What happens to requests for large files if the SRPT server experiences transient periods of overload?"

Most well-managed web servers perform well most of the time. Occasionally, however, every popular web server experiences transient *overload*. Overload is defined as the point when the demand on at least one of the web server's resources exceeds the capacity of that resource. While a well designed web server should not be persistently overloaded, transient periods of overload are often inevitable, since the traffic increase at the server that leads to the transient period of overload is difficult to predict. As an example, the amount of traffic received by a web site might rise because of an unexpected increase of the site's popularity, e.g. after being featured on national television or in a major newspaper. Another situation that could lead to transient periods of overload is under-provisioning for sales-boosting holidays. Although an online retailer knows to expect more web site hits during those days, it is difficult to predict exactly the associated increase in traffic volume.

An overloaded web server typically displays signs of its affliction within a few seconds. Work enters the web server at a greater rate than the web server can complete it. This causes the number of connections at the web server to build up. Very quickly, the web server reaches the limit on the number of connections that it can handle. From the client perspective, the client's request for a connection will either never be accepted or will get through only after several trials. Even when the client's request for a connection does get accepted, the time to service that request may be very long because the request has to *timeshare* with all the other requests at the server.

The solution most commonly suggested to avoid overload is admission control [61, 103, 204, 203, 209, 78]: the web server or a front-end monitors the load and the capacity of the server and, if necessary, rejects incoming requests, in order to ensure satisfactory service to those requests being accepted to the server. The decision of when and which request to reject is based on sophisticated algorithms, e.g. leveraging application-level knowledge in the admission decision, applying techniques from control theory, or by combining admission control with QoS. Nevertheless, in the end it comes down to denying some customers service.

In this chapter we suggest SRPT scheduling as a solution to server overload, that provides stable server performance, without dropping requests. Our intuition is based on the observation that the dramatic drop in performance during overload in a standard FAIR server is due to the rapidly growing number of requests at the server. The fair shares of service that each request receives in the FAIR server are getting smaller and smaller during overload, until no request makes any significant progress any more. On the other hand, SRPT can be viewed as a greedy strategy to minimize the number of requests at the server by always working on the request that is closest to completion.

While we have already seen in the previous chapter that SRPT can greatly improve the mean performance of static requests under normal load conditions, overload is a regime where SRPT has never been evaluated (not analytically nor via simulation). While it might be intuitively clear that SRPT will help overall performance during overload by minimizing the number of outstanding requests at the server, it is not clear at all how the performance of large requests will be affected.

In the remainder of this chapter we make two contributions:

First, we provide a detailed performance study of a web server under overload, showing just how bad overload can be. We experiment with both *persistent overload* (the request rate exceeds the server capacity during the entire experiment) and *transient overload* (alternate periods of overload and low load where the overall mean load is below 1). Load will be defined formally in the next section. We find that within a very short period, even low amounts of overload cause the server to experience instability, dropping requests from its SYN queue, while the client experiences very high response times (response time is defined as the time from when the client submits the request until receiving the last byte of the request). We evaluate a full range of complex environmental conditions, summarized in Table 5.2, including: the effect of WAN delay and loss, the effect of user aborts, the effect of persistent connections, the effect of SYN cookies, the effect of the RTO TCP timer, the effect of the packet length, and the effect of the SYN queue and ACK queue length limits.

Our motivation for this performance study is that, while there are many studies of web server performance in general (no overload), there exist relatively few studies on servers running under overload. Studies evaluating the effect of external factors on web performance typically concentrate on the effect that network protocols/conditions have on web server performance. We list just a few below: In [22] and [45] the authors find that the TCP RTO value has a large impact on server performance under FreeBSD. This agrees with our study. In [153] the authors study the effect of WAN conditions, and find that losses and delays can affect response times. They use a different workload from ours (Surge workload) but have similar findings. The benefits of persistent connections are evaluated by [148] and [33] in a LAN environment. There are also several papers which study real web servers in action, rather than a controlled lab setting, e.g., [149] and [183].

As a second contribution, this chapter proposes and evaluates SRPT-like scheduling as a means to combat overload. We show that contrary to intuition, SRPT scheduling at the web server under transient overload does not unduly harm requests for large files as compared with traditional FAIR scheduling used in web servers. Furthermore SRPT scheduling significantly improves mean response times overall by up to an order of magnitude. Lastly, even the mean time until the first byte is improved by close to an order of magnitude under SRPT as compared with FAIR. Our implementation results are corroborated via theoretical approximations which match the trends we observe.

The outline of this chapter is as follows: In Section 5.1 we describe the experimental setup, including the machine configuration, the workload, and how overload is generated. Section 5.2 studies exactly what happens in a traditional (FAIR) web server under persistent overload and contrasts that with the performance of the modified (SRPT) web server. Section 5.3 compares the performance of the FAIR server and the SRPT server under transient overload. Section 5.4 analyzes where exactly SRPT's performance gains come from and Section 5.5 summarizes the results.

5.1 Experimental setup

5.1.1 Machine configuration

We use the same experimental testbed involving six machines as in the previous chapter. The switch and the network cards of the six machines are forced into 10Mbps mode to make it easier to create overload at the bottleneck device.

On the server machine we increased the size of the SYN and the ACK-queue to 512 as is common practice for high performance web servers We also increased the upper limit on the number of Apache processes from 150 to 350.[1]

Furthermore, we have instrumented the kernel of the server to provide detailed information on the internal state of the server. This includes the length of the SYN and ACK-queues, the number of packets dropped inside the kernel, and number of incoming SYNs that are dropped. We also log the number of active sockets at the server, which includes all TCP connections that have resources in the form of buffers allocated to them, except for those in the ACK-queue. Essentially, this means sockets being serviced by an Apache process, and sockets in the FIN-WAIT state.

Below we provide a brief tutorial of the processing of requests and sources of delays within a Linux-based web server.

A connection begins with a client sending a SYN. When the server receives the SYN it allocates an entry in the SYN-queue and sends back a SYN-ACK. After the client ACKs the server's SYN-ACK, the server moves the connection record to its ACK-queue, also known as the Listen queue. The connection waits in the ACK-queue until an Apache process becomes idle and processes it. Each Apache process can handle at most one request at a time. When an Apache process finishes handling a connection, the connection sits in the FIN-WAIT states until an ACK and FIN are received from the client.

[1]Our FAIR server performed best with the above values: SYNQ = ACKQ = 512, and #Apache Processes = 350. The SRPT server is not affected at all by these limits, since SYNQ occupancy is always very low under SRPT.

There are standard limits on the length of the SYN-queue (128), the length of the ACK-queue (128), and the number of Apache processes (150). These limits are often increased for high-performance web servers.

The limits above impose many sources of delays. If the server receives a SYN while the SYN-queue is full, it discards the SYN forcing the client to wait for a timeout and then retransmit the SYN. Similarly, if the server receives the ACK for a SYN-ACK while the ACK-queue is full, the ACK is dropped and must be retransmitted. The timeouts are long (typically 3 seconds) since at this time the client doesn't have an estimate for the retransmission timer (RTO) for its connection to the server. Lastly, if no Apache process is available to handle a connection, the connection must wait in the ACK-queue.

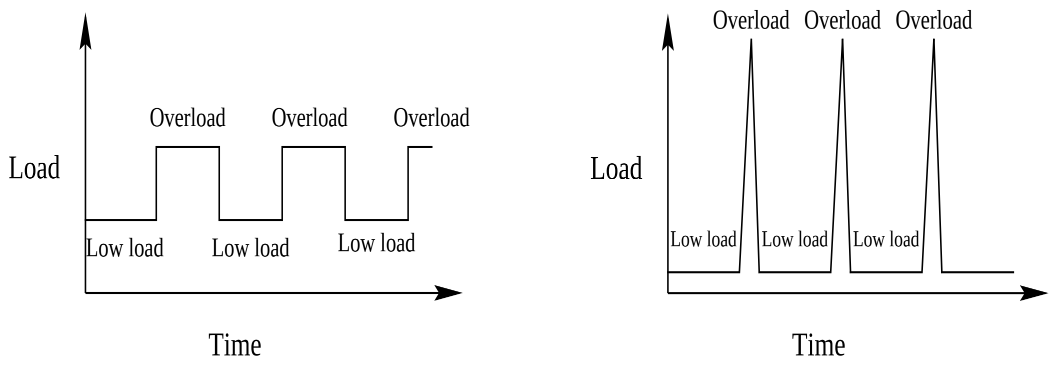
5.1.2 The workload

The workload in our experiments is based on the trace described in the previous chapter (Section 4.1.2). Results for a second trace are included in the appendix. Based on the traces, we create two types of overload, *persistent overload* and *transient overload*.

Persistent overload is used to describe a situation where the server is run under a fixed load $\rho > 1$ during the whole experiment. The motivation behind experiments with persistent overload is mainly to gain insight into what happens under overload. The overloaded state is unlikely to persist for too long in practice, due to system upgrades. Nevertheless, due to the burstiness of web traffic, even in the case of regular upgrades a popular web server is still likely to experience *transient* periods of overload.

We consider two different types of *transient overload*: In the first type, called *alternating overload*, the load alternates between overload and low load, where the length of the overload period is equal to the length of the low load period (see Figure 5.1(left)). In the second, called *intermittent overload*, the load is almost always low, but there are occasional "spikes" of overload, evenly spaced (see Figure 5.1(right)). *In all cases the overall mean system load is less than 1.*

As before, we define *load* to be the ratio of the bandwidth requested and the maximum bandwidth available on the uplink. To obtain a particular load we scale the interarrival times in the trace as



(a) Alternating Overload (b) Intermittent Overload

Figure 5.1: *Two different types of transient overload, alternating (left) and intermittent (right). Experimentally, the load will never look as constant as above, since arrival times and request sizes come from a trace.*

follows: We first measure the system load (i.e. the bandwidth utilization) in an experiment using the original (unscaled) interarrival times from the trace. In order to achieve a system load that is x times higher than the original load, we divide all interarrival times by x. Scaling the interarrival times (while keeping the sequence of requested web objects constant) models a general increase in traffic at the web server, e.g. due to sudden popularity or due to holiday shopping. Note that this type of overload is different from the case where a sudden rise in the popularity of one single object at a site causes overload. In this thesis we are addressing only overload conditions that are caused by an increase in the arrival rate at the server, while the distribution of the documents requested remains the same (or similar) to that before overload.

We run all experiments in this chapter for several different alternating and intermittent workloads, which are defined in Table 5.1. All our results are based on 30 minute experiments although we show only shorter fragments in the figures for better readability.

5.1.3 Why generating overload is difficult

Experimenting with persistent overload is inherently more difficult than running experiments where the load is high but remains below 1. The main issue in experimenting with overload is that running the server under overload is very taxing on *client* machines. While both the client machines and the server must allocate resources (such as TCP control blocks or file descriptors) for all accepted

Workload	Type	Duration low load (seconds)	Duration overload (seconds)	Avg. low load	Avg. overload	Avg. load
W1	Alternating	25	25	$\rho = 0.2$	$\rho = 1.2$	0.7
W2	Alternating	10	10	$\rho = 0.2$	$\rho = 1.2$	0.7
W3	Alternating	50	50	$\rho = 0.2$	$\rho = 1.6$	0.9
W4	Alternating	25	25	$\rho = 0.4$	$\rho = 1.4$	0.9
W5	Alternating	10	10	$\rho = 0.4$	$\rho = 1.4$	0.9
W6	Alternating	40	40	$\rho = 0.1$	$\rho = 1.7$	0.9
W7	Intermittent	63	6	$\rho = 0.4$	$\rho = 5$	0.8
W8	Intermittent	63	10	$\rho = 0.45$	$\rho = 3$	0.8
W9	Intermittent	20	3	$\rho = 0.735$	$\rho = 2$	0.9
W10	Intermittent	13.3	2	$\rho = 0.735$	$\rho = 2$	0.9

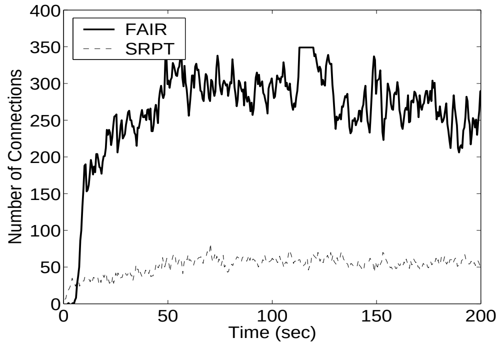
Table 5.1: *Definition of trace-based workloads*

requests, the client machines must additionally allocate resources for all connections that the server has repeatedly refused (due to a full SYN-queue).

As explained in Section 4.1.2 requests to a web server may be generated either using an "open" system, a "closed" system, or some hybrid combination of the two. To obtain overload, an open or partly-open system is necessary [26], since in a closed system a new request will be made only if another request finishes (see Figure 4.1(middle)). By contrast, an open system allows one to create any amount of overload by simply generating a rate of requests where the sum of the sizes of the files requested exceeds the server uplink bandwidth.

Since none of the existing web workload generators support all the features we want to experiment with (persistent connections, user abort and reload, etc.) we choose to implement our own trace-based web workload generator based on the `libwww` library [206]. `Libwww` is a client side web API based on *select()*. One obstacle in using `libwww` in building a web workload generator is that it does not support multiple parallel connections *to the same host*. After passing a URL to `libwww`, it first checks whether there is already a socket open to this destination. If so it either multiplexes the request on this socket (if pipelining is enabled) or otherwise puts the request into a *pending queue* at this socket. We modify `libwww` in the following way to perform our experiments with persistent connections: Whenever our application passes a URL to our modified `libwww`, it first checks whether there is a socket open to this destination that is (a) idle and (b) that has not reached the limit on the maximum number of times that we want to reuse a connection. If it doesn't find such a socket it establishes a new connection. We validate our workload generator by running experiments involving only the features supported by `Sclient`, and we find that `Sclient` and our new workload generator yield the same results. We also verify that the workload generator never becomes the bottleneck in the experiments, by checking that all requests are actually made at the desired times.

5.2 Experimental Results – Persistent overload



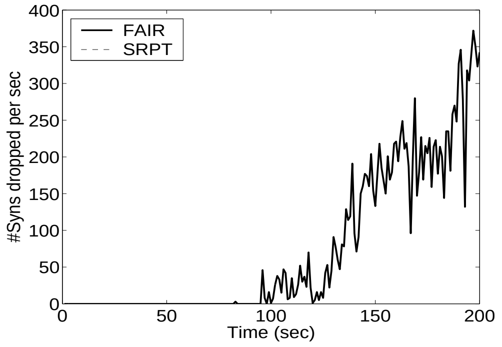
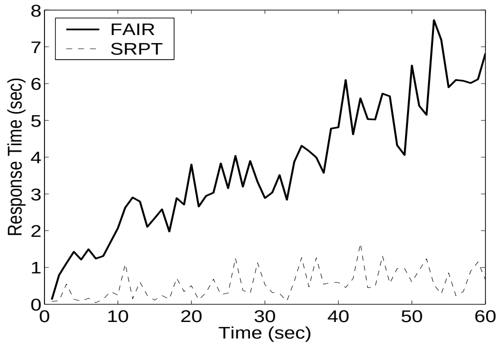


Figure 5.2: *Results for a persistent overload of 1.2 shown from the perspective of the server. (Left graph): Buildup in connections at the server; (Right graph): Number of incoming SYN packets dropped by the server.*



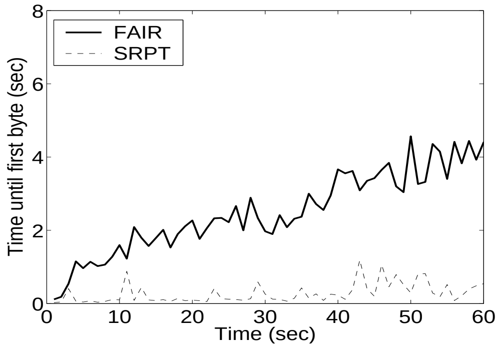


Figure 5.3: *Results for a persistent overload of 1.2 shown from the perspective of the client. (Left graph): Mean response time; (Right graph): Time until first byte is received.*

In this section we study exactly what happens in a standard (FAIR) web server during persistent overload and contrast that with the performance of our modified (SRPT) server. We run the web server under persistent overload of 1.2, i.e., the average amount of data requested by the clients per second exceeds the bandwidth on the uplink by a factor of 1.2. We analyze our observations from two different angles, the server's view and the client's view.

We start with the server's view. One indication for the health of a server is the buildup in the number of connections at the server, shown in Figure 5.2(left). In FAIR, the number of connections grows rapidly, until after around 50 seconds the number of connections reaches 350 – the maximum number of Apache processes. At this time all the Apache processes are busy, and consequently the SYN and the ACK queues fill up. After around 70 seconds the first incoming SYNs are dropped and the rate of SYN drops increases steadily and rapidly from that point on, as shown in (Figure 5.2(right)). By contrast, in the SRPT server the number of connections grows at a much slower rate. The reason is that the SRPT server queues up only requests for large files. Observe, that for an overload of 1.2 and an uplink capacity of 10 Mbps, each second the server is unable to complete 2 Mbit worth of requests on average. It turns out that largest requests compromising load between 1 and 1.2 have a mean size of about 1 Mbyte. Thus the SRPT server accumulates only one quarter of one request per second. After 200 seconds, it thus makes sense that the number of accumulated requests under SRPT is only 50, as shown in Figure 5.2(left). In fact our experiments show that the SRPT server does not start dropping requests until approximately the half hour mark.

Another server-oriented metric that we consider is *byte throughput.* We find that the byte throughput is the same under FAIR and SRPT . Despite the differences in the way FAIR and SRPT schedule requests, they both manage to keep the link fully utilized.

Next we describe the clients' experience in terms of *mean response time* and the *mean time until the first byte of a request is received.* In computing these metrics for persistent overload, we consider only those requests that finished before the final request arrived (subsequently, load will drop). Figure 5.3(left) shows that for the FAIR server response times grow rapidly over time. After only 40 seconds (long before the SYN-queue fills up) the mean response time is 5 sec, already intolerable. By contrast, under SRPT the response times are significantly lower and hardly grow over time. Figure 5.3(right) shows that the mean time until the first byte is received follows a trend very similar to that of the mean response time. We will therefore in the remainder of this chapter use only the response time metric.

To understand the difference in response times between SRPT and FAIR under persistent overload we need to examine the effect of those requests that don't complete on the requests that do complete (and factor into the mean response time). Under FAIR, many of the requests that don't end up completing still steal a fair share of bandwidth from the other requests. Hence, they cause the response times of even requests for short files to increase. By contrast, under SRPT, all of the requests for the largest files (those that increase load from 1.0 to 1.2) do not receive any bandwidth under persistent overload: requests for small files are completely isolated from those for large files under SRPT. Thus, response times of the completing requests do not increase over time, even after tens of minutes.

To summarize the above observations, we see that after less than 100 seconds of very modest overload the FAIR server starts to drop incoming requests and the response times reach values that are not tolerable by users. The SRPT server significantly extends the time until SYNs are dropped and improves the client experience notably. We emphasize that the above experiments assumed very modest overload (as in the high load portion of workload W1 of Table 5.1). Some of the other workloads in Table 5.1 have more severe high loads. For example under a persistent high load of 1.4, we find that SYNs are dropped after only 18 seconds of persistent overload under FAIR, whereas SRPT avoids SYN drops for nearly half an hour of persistent overload. Response time growth is also much more dramatic under a persistent overload of 1.4.

5.3 Experimental Results – Transient overload

In this section we evaluate the performance of the standard (FAIR) web server and our modified (SRPT) server for *transient* overload. To level the playing field between SRPT and FAIR, we purposely choose to start by evaluating in detail the performance for the transient workload W1 from Table 5.1 (see Section 5.3.1). Workload W1 has the property that the duration and intensity of overload are modest (high load is only 1.2 for a duration of only 25 seconds), so that the SYN queue does not fill up under FAIR. Thus FAIR does not suffer the expensive timeouts under workload W1 due to a SYN drop, which appear under some of the other workloads. Our study considers all factors described in Table 5.2. In Section 5.3.2, we consider the other transient workloads in Table 5.1.

Setup	Factor	Specific case shown in the left and middle columns of Figure 5.5 and Figure 5.6.	Range of values studied. Shown in rightmost column of Figure 5.5 and Figure 5.6.
(A)	Baseline Case	RTT=0, Loss=0%, No Persistent Conn., RTO=3 sec, packet length=1500, No SYN cookies, SYNQ=ACKQ=512, #ApacheProcesses=350	
(B)	WAN Delays	baseline + 100 ms RTT	RTT=0-150 ms
(C)	Loss	baseline + 5% loss	Loss=0-15%
(D)	WAN delay & loss	baseline + 100 ms RTT+ 5% loss	RTT=0-150 ms, Loss =0-15%
(E)	Persistent Connections	baseline + 5 req. per conn.	0-10 requests/conn.
(F)	Initial RTO value	baseline + RTO 0.5 sec	RTO = 0.5sec-3sec
(G)	SYN Cookies	baseline (SYN Cookies OFF)	SYN cookies=ON/OFF
(H)	User Abort/Reload	baseline + user aborts: User aborts after 10 sec and retries up to 3 times	Abort after 3-15 sec with up to 2, 4, 6, or 8 retries
(I)	Packet length	baseline + 536 bytes packet length	Packet length = 536-1500 bytes
(J)	Realistic Scenario	RTT=100 ms, Loss=5%, 5 req. per conn., RTO=3 sec, pkt. len.=1500, No SYN cookies, SYNQ=ACKQ=512, #ApacheProcs=350, User aborts after 7 sec and retries up to 3 times	

Table 5.2: *Columns 1 and 2 list the various factors. Column 3 specifies one value for each factor. This value corresponds to Figure 5.5(left, middle) and to Figure 5.6(left, middle). Column 4 provides a range of values for each factor. The range is evaluated in Figure 5.5(right) and in Figure 5.6(right).*

5.3.1 Results for workload W1

(A) The simple baseline case

In this section we study the simple baseline case described in Table 5.2, row (A). We first consider how the health of the server is affected during the low load and overload periods. We observe, as in the case of persistent overload, that the number of connections grows at a much faster rate under FAIR than under SRPT. While under SRPT the number of connections never exceeds 50, it frequently exceeds 200 under FAIR. However, neither server reaches the maximum SYN-queue capacity (since the overload period is short) and therefore no SYNs are dropped.

Figure 5.4(a) and (b) shows the response times over time of all jobs (averaged over 1 sec intervals) under FAIR and SRPT. These plots show just 200 sec out of a 30-minute experiment. The overload periods are shaded light gray while the low load periods are shaded dark gray. Observe that under FAIR the mean response times go up to more than 3 seconds during the overload period[2], while under SRPT they hardly ever exceed 0.5 sec.

Figure 5.4(c) shows the complementary cumulative distribution of the response times. Note that there is an order of magnitude separation between the curve for FAIR and SRPT. The mean response time taken over the entire length of the experiment is 1.1 sec under FAIR as compared to only 138 ms under SRPT. Furthermore, the variability in response times measured by the squared coefficient of variation is 6.39 under FAIR compared to 1.08 under SRPT. Another interesting observation is that the FAIR curve has bumps at regular intervals. These bumps are due to TCP's exponential backoff in the case of packet loss during connection setup. Given that we have a virtually loss-free LAN setup and the SYN-queue never fills up, one might wonder where these packets are dropped. Our measurements inside the kernel show that the timeouts are due to reply packets of the server being dropped inside the server's kernel. We will study the impact of this effect in greater detail in Section 5.4.

The big improvements in mean response time are not too surprising given the opportunistic

[2]Note that this is not quite as bad as for persistent overload, because a job arriving into the overload period in transient overload at worst has to wait until the low load period to receive service, so its expected response time is lower than under persistent overload.

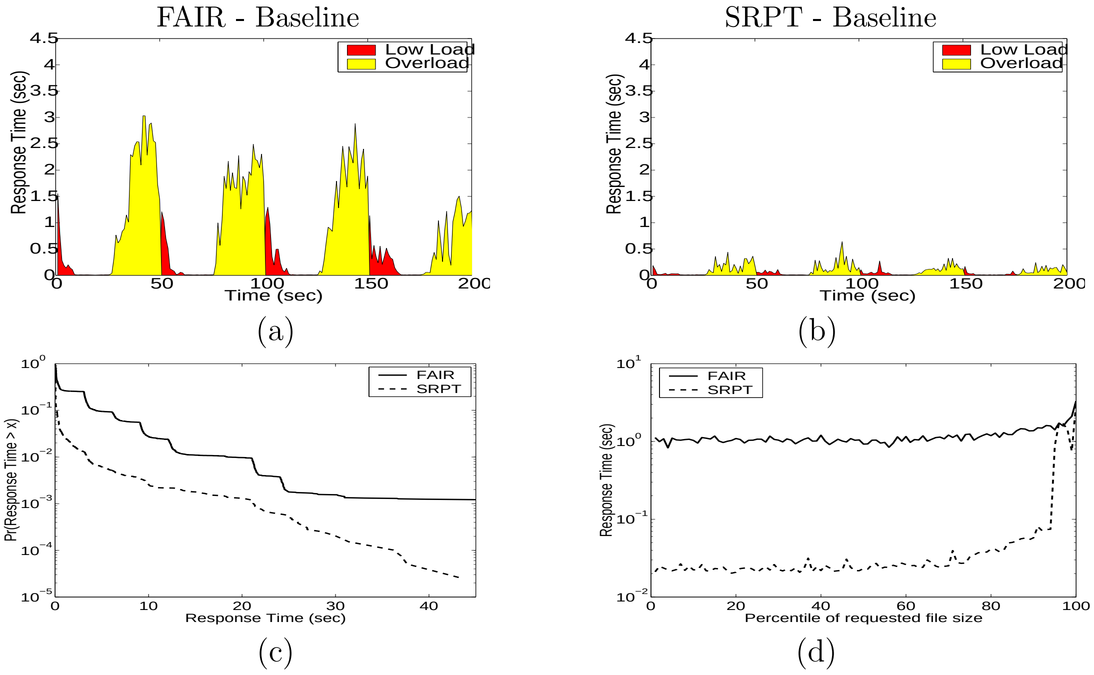


Figure 5.4: *Detailed performance under alternating workload W1 under the baseline case (setup (A)) of Table 5.2: (a) Mean response time under FAIR; (b) Mean response time under SRPT; (c) The complementary cumulative distribution of response times under FAIR and SRPT; (d) Response times as a function of the request size, showing requests for large files do not suffer.*

nature of SRPT: schedule to minimize the number of connections. A more interesting question is what price large requests have to pay to achieve good mean performance.

This question is answered in Figure 5.4(d) which shows the mean response times as a function of the request size. We see that surprisingly *even the big requests hardly do worse under SRPT*. The very biggest request has a mean response time of 19.4 sec under SRPT compared to 17.8 sec under FAIR. If we look at the biggest 1 percent of all requests we find that their average response time is 2.8 sec under SRPT compared to 2.9 sec under FAIR, so these requests perform better on average under SRPT. We will explain this counter-intuitive result in Section 5.4.

(B) WAN delays

The two most frequently used tools for WAN emulation are probably NistNet [156] and Dummynet [173]. NistNet is a separate package available for Linux that can drop, delay or bandwidth-limit

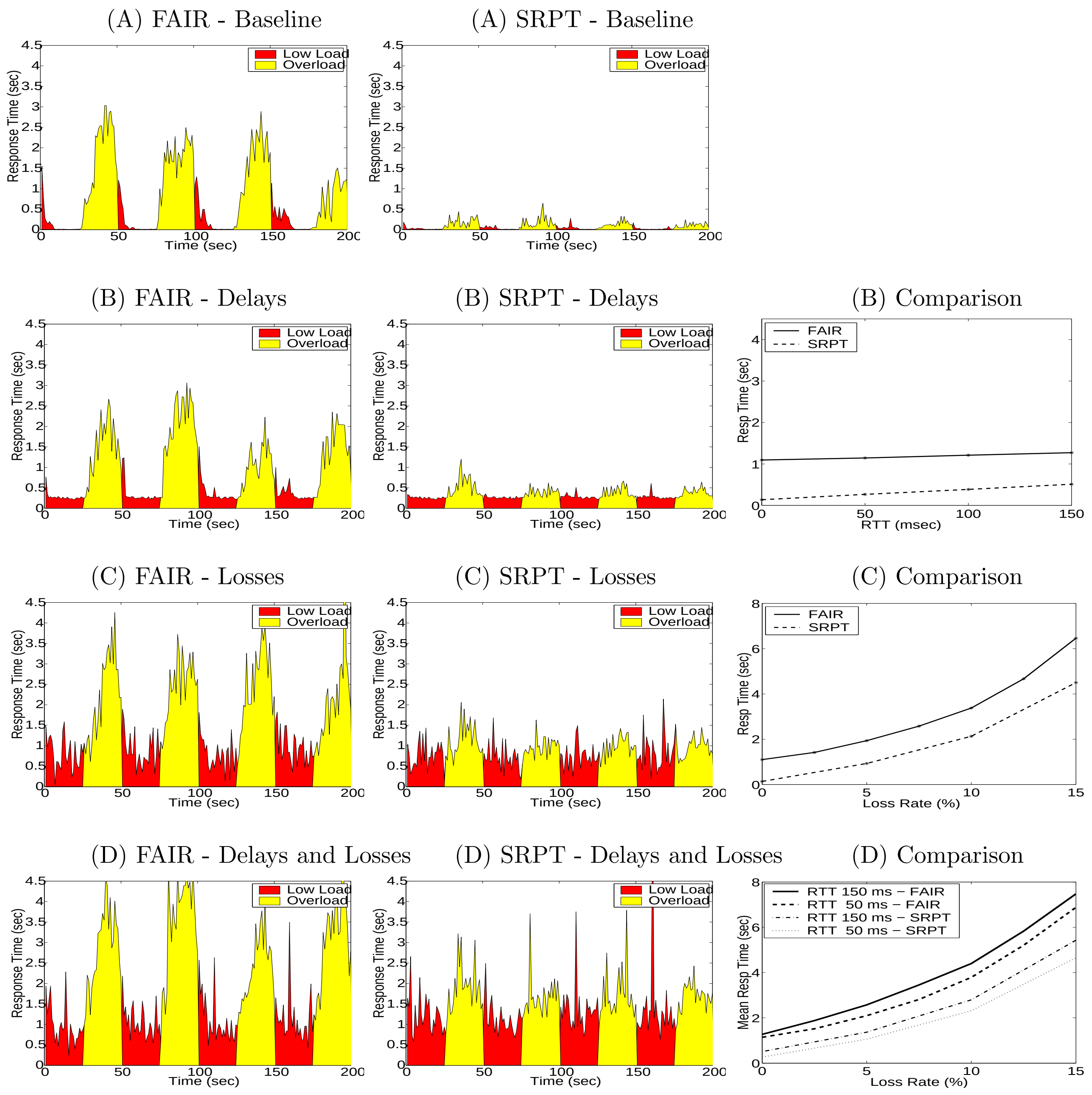


Figure 5.5: *Each row above compares SRPT and FAIR under workload W1 for one of the first four setups from Table 5.2. The left and middle columns show the response times over time for the specific values given in Table 5.2, column 3. The right column evaluates the range of values given in column 4 of Table 5.2.*

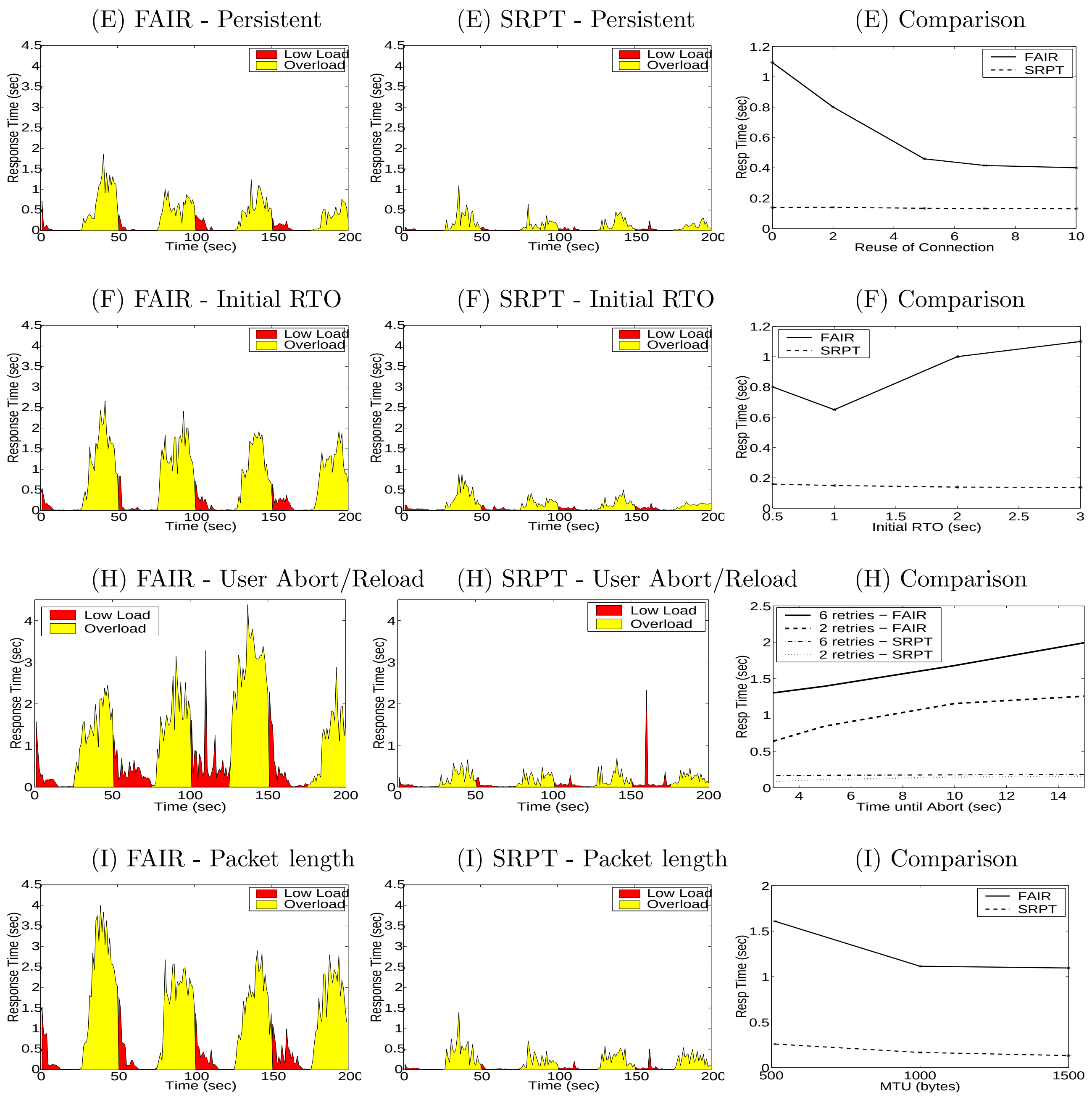


Figure 5.6: *Each row above compares SRPT and FAIR under workload W1 for one of setups (E), (F), (H), and (I). from Table 5.2. The left and middle columns show the response times over time for the specific values given in Table 5.2, column 3. The right column evaluates the range of values given in column 4 of Table 5.2.*

incoming packets. Dummynet applies delays and drops to both incoming *and* outgoing packets, hence allowing the user to create symmetric losses and delays. Since Dummynet is currently available for FreeBSD only, we implement Dummynet functionality in the form of a separate module for the Linux kernel. More precisely, we change the `ip_rcv()` and the `ip_output()` function in the Linux TCP-IP stack to intercept in- and out-going packets to create losses and delays.

In order to delay packets, we use the `add_timer()` facility to schedule the transmission of delayed packets. We recompile the kernel with `HZ=1000` to get a finer-grained millisecond timer resolution. In order to drop packets, we use an independent, uniform random loss model (as in Dummynet) which can be configured to a specified probability.

We experiment with delays between 0 and 150 msec and drop probabilities between 0 and 15%. This range of values was chosen to cover values used in related work [153] and values reported in actual live Internet measurements [171]. For example, for the month of October 2004 the Internet Traffic Report [171] reported maximum round-trip-times of 140 msec and maximum loss rates of 3.5%.

Figure 5.5(B)(left, middle) shows the effect of adding WAN delay (setup (B) in Table 5.2). This assumes a baseline setup, with an RTT (round-trip-time) delay of 100 msec. While FAIR's mean response time is hardly affected, since it is large compared to the additional delay, the mean response time of SRPT more than doubles.

Figure 5.5(B)(right) shows the mean response times for a range of RTTs from 0 to 150 msec. Observe that adding WAN delays increases response times by a constant additive factor on the order of a few RTTs. SRPT improves upon FAIR by at least a factor of 2.5 for all RTTs considered.

In the above experiments we assumed that all clients experience the *same* WAN delays. We also experimented with a heterogeneous environment, where each of the five client machines emulates a different WAN delay ranging from 0 ms to 150 ms, where 150 ms represents our notion of the maximum end-to-end network delay. We find that the mean response time for a client with a given WAN delay equals that in an experiment where all clients emulate the same delay.

While one might think that in a heterogeneous environment, high delay clients might interfere with low delay clients, depriving them of the full benefits of SRPT scheduling, this is not the case for two reasons: First, under overload the time a connection stays alive at the server is mostly dominated by server delays, rather than by WAN delays; second, and most importantly, a slow connection will not "block" a fast connection: data for different connections is kept completely separate until after TCP/IP processing when it enters one of the priority queues shared by all connections (recall Figure 3.2). The data packets of a slow connection will therefore make it to the priority queues only after receiving the corresponding TCP ACKs (i.e. when the data is ready to be sent), and can therefore not slow down the fast connections (which will receive their ACKs at a fast rate). Thus the difference between SRPT and FAIR is minimized when all clients have WAN delay of 150 ms, and even here, the improvement factor of SRPT over FAIR is 2.5.

(C) Network losses

Figure 5.5(C)(left, middle) shows the mean response times over time for the setup in Table 5.2 row (C): a loss rate of 5%. In this case, for both FAIR and SRPT, the response times increase notably compared to the baseline case. FAIR's overall response time increases by almost a factor of 2 from 1.1 seconds to 1.9 seconds. SRPT's response time increases from less than 140 ms in the baseline case to around 930 ms.

Figure 5.5(C)(right) shows the mean response times for loss rates ranging from 0 to 15%. Note that the response times don't grow linearly with the loss rate. This is expected since TCPs throughput is inversely proportional to the square root of the loss [159].

Introducing frequent losses can increase FAIR's response times to more than 6 seconds and SRPT's response time to more than 4 seconds (as in the case of 15% loss). Even in this case SRPT still improves upon FAIR by about a factor of 1.5.

(D) Combination of Delays and Losses

Finally, we look at the combination of losses and delays. Figure 5.5(D)(left, middle) shows the results for the setup in Table 5.2 row (D): an RTT of 100 ms with a loss rate of 5%. Here SRPT improves upon FAIR by a factor of 2. Figure 5.5(D) (right) shows the response times for various combinations of loss rates and delays. We observe that the negative effect of a given RTT is accentuated under high loss rates. The reason is that higher RTTs make loss recovery more expensive since timeouts depend on the (estimated) RTT.

(E) Persistent Connections

Next we explore how the response times change if multiple requests are permitted to use a single *serial*, *persistent* connection ([148]) for several requests. Figure 5.6(E)(left, middle) shows the results for the setup in Table 5.2, row (E): where every connection is reused 5 times. Figure 5.6(E)(right) shows the response time as a function of the number of requests per connection, ranging from 0 to 10. We see that using persistent connections greatly improves the response times of FAIR. For example, reusing a connection five times reduces the response time by a factor of 2. SRPT on the other hand is hardly affected by using persistent connections. To see why FAIR benefits more than SRPT observe that reusing an existing connection avoids the connection setup overhead; this overhead is bigger under FAIR, mainly because it suffers from drops inside the kernel.[3] SRPT doesn't see this improvement since it experiences hardly any packet loss in the kernel.

Nevertheless, we observe that SRPT still improves upon FAIR by a factor of 3, even if up to 10 requests can use the same connection.

(F) Initial TCP RTO value

We observed previously that packets that are lost in the connection setup phase incur very long delays, which we attributed to the conservative initial RTO value of 3 seconds. We now ask how much of the total delay a client experiences in the FAIR server is due to the high initial RTO. To answer this question, we change the RTO value in Linux's TCP implementation. Figure 5.6(F)(left, middle) shows the results for the setup in Table 5.2, row (F): an RTO of 500ms. Figure 5.6(F)(right) explores the range from 500 ms up to the standard 3 seconds. Lowering the initial RTO can reduce FAIR's mean response time from originally 1.1 seconds (for the standard RTO of 3 seconds) to 0.65 seconds

[3]These drops occur when attempting to feed packets into an already full transmit queue (see Figure 3.2).

(for an initial RTO of 1 second). Reducing the initial RTO below 1 second introduces too much overhead due to spurious retransmissions, and therefore doesn't improve performance any further. SRPT's response times don't improve for lower initial RTOs since SRPT has little loss in the kernel. Nevertheless, for all initial RTO values considered, SRPT always improves upon FAIR by a factor of at least 4.

Having observed that the mean response times of a (standard) FAIR server can be significantly reduced by reducing TCP's initial RTO, we are not suggesting to change this value in current TCP implementations, since there are reasons for why it is set conservatively [162].

(G) SYN cookies

Recall that one of the problems under overload is the dropping of incoming packets due to a full SYN-queue. This leads us to the idea of using SYN cookies [36] in overload experiments. SYN cookies were originally developed to avoid denial of service attacks, however they have the added side-effect of eliminating the SYN-queue. (When using SYN cookies the server makes the SYN-ACK contents purely a function of the SYN contents. This way the SYN contents can be recomputed upon receipts of the next ACK, thereby avoiding the need for maintaining a SYN-queue.) Our hope is that by getting rid of the SYN-queue we will also eliminate the problems involving a full SYN-queue.

It turns out that the use of SYN cookies hardly affects the response times under workload W1 since in workload W1 the SYN-queue never fills up. In other transient workloads which exhibit SYN drops due to a full SYN-queue, the response times do improve, but only slightly by 2–5%. The reason is that now instead of the incoming SYN being dropped, it is the ACK for the SYN-ACK that is dropped due to a full ACK-queue.

Since SRPT does not lose incoming SYNs, its performance is not affected by using SYN cookies.

(H) User abort/reload

So far we have assumed that a user patiently waits until all the bytes for a request have been received. In practice, users abort requests after a while and hit the reload button of their browser.

We model this behavior by repeatedly aborting a connection if it hasn't finished after a certain number of seconds and then opening a new connection.

Figure 5.6(H)(left,middle) shows the response times in the case where a user waits for at most 10 seconds before hitting reload, and retrying this procedure up to 6 times before giving up. Figure 5.6(H)(right) shows mean response times if connections are aborted after 3 to 15 seconds and for either 2 or 6 retries.

We observe that taking user behavior into account can, depending on the parameters, sometimes increase and sometimes decrease response times. If users are impatient and abort connections quickly and retry only very few times, the response times can decrease by up to 10%. This is because there is now a (low) upper bound on the maximum response times and also the work at the server is reduced since clients might give up before the server has even invested any resources in the request. However, this reduced mean response time does not necessarily mean greater average user satisfaction, since some number of requests never complete. For example, if users abort a connection after 3 seconds and retry at most 2 times, more than 5 percent of the requests under FAIR never complete for workload W1.

If users are willing to wait for longer periods of time and retry more often, response times significantly increase. The reason is that response times are long both for those users that go through many retries, but also for other users, since frequent aborts and reloads increase the work at the server. For example, if users retry up to 6 times and abort a connection only after 15 seconds the response times under FAIR almost double compared to the baseline case that doesn't take user behavior into account. On the other hand the number of incomplete requests is very small in this case – under 0.02%.

For all choices of parameters for user behavior, the response times under SRPT improve upon those under FAIR by at least a factor of 8. Also the number of incomplete requests is always smaller under SRPT, by as much as a factor of 7. The reason is that SRPT is smarter than FAIR. Because SRPT favors small requests, the small requests (the majority) have no reason to abort. Only the few big requests are harmed under SRPT. Nevertheless, even requests for the longest file suffer no more incompletes under SRPT than under FAIR.

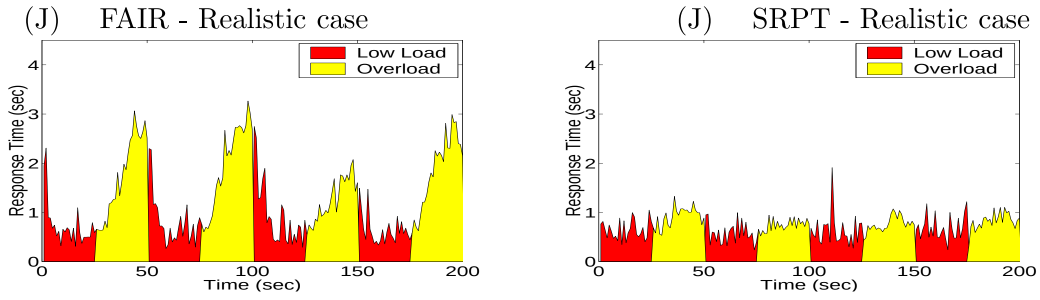


Figure 5.7: *Comparison of FAIR (left) and SRPT (right) for the realistic setup (J) for workload W1.*

(I) Packet Length

Next we explore the effect of the maximum packet length (MSS). Two different packet lengths are commonly observed in the Internet [81]: 1500 bytes, since this is the MTU (maximum transmission unit) for Ethernet, and 536 bytes, which is used by some TCP implementations that don't perform MTU discovery [121].

Figure 5.6 (I)(left,middle) shows the results for setup (I) in Table 5.2, where the packet length is changed from the 1500 bytes in the baseline case to 536 bytes. As expected, the mean response time increases, since for a smaller packet length more RTTs are necessary to complete the same transfer. FAIR's response time increases by almost 50% to 1.6 seconds and SRPT's response time doubles to 260 msec.

Figure 5.6 (I)(right) shows the mean response times for different packet lengths ranging from 500 bytes to 1500 bytes. For all packet lengths considered, SRPT improves upon FAIR by at least a factor of 6.

(J) A realistic scenario

So far, we have looked at several factors affecting web performance in isolation. Figure 5.7 shows the results for an experiment that combines all the factors: We assume an RTT of 100 ms, a loss rate of 5%, no SYN-cookies, and the use of persistent connections (5 requests per connection). We leave the RTO at the standard 3 seconds and the MTU at 1500 bytes. We assume users abort after

7 seconds and retry up to 3 times.

We observe that the mean response time of both SRPT and FAIR increases notably compared to the baseline case. It is now 1.48 seconds under FAIR and 764 msec under SRPT – a factor 2 improvement of SRPT over FAIR. Observe that both these numbers are still better than those in experiment (D), where we combined only losses and delays and saw a response time of 2.5 seconds and 1.2 seconds, respectively. The reason is that the use of persistent connections alleviates the negative effects of losses and delays during connection setup.

The largest 1% of requests (counting only those that complete) have a response time of 2.23 seconds under FAIR and 2.28 seconds under SRPT. Even the response time of the very biggest request is higher under FAIR: 13.2 seconds under FAIR and only 12.8 seconds under SRPT. The total number of incomplete requests (those aborted the full three times) is the same under FAIR and SRPT – about 0.2%.

Observe that it makes sense that unfairness to large requests is more pronounced in the baseline case because server delay dominates response time under the baseline case, in contrast to the realistic case where external factors can dominate.

Under workload W1 there were no SYN drops, neither for the baseline nor the realistic setup.

5.3.2 Other transient workloads

Figure 5.8 gives the mean response times for all ten workloads from Table 5.1 for the baseline case (Table 5.2 row (A)). While Figure 5.8 depicts *mean* response times over the entire duration of the experiment, it is important to notice that *peak* response times are actually much higher. For example, for workload W1, when considering the response times *averaged over 1 second intervals* as in Figure 5.4(a) and (b), we witnessed response times three times higher than that shown in Figure 5.8 for workload W1. This underscores the need for SRPT scheduling.

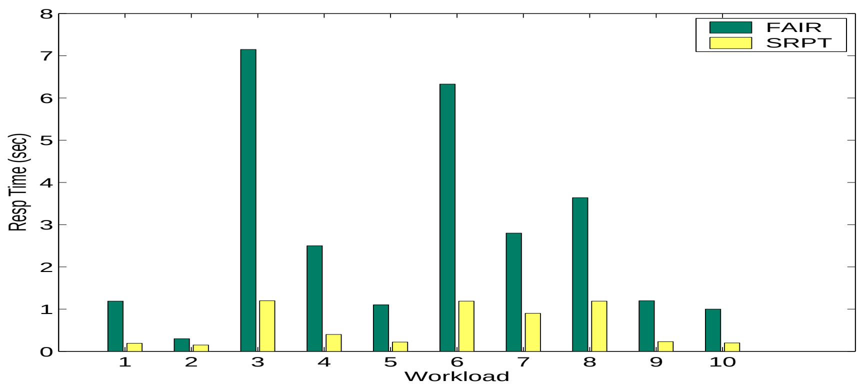
The reason that we choose to show performance for the baseline case rather than the realistic case is that this way the effects of the different workloads are not blurred by external factors. Also, the starvation of large requests is by definition greater for the baseline case than the realistic case, as explained above. In the discussion below, however, we will include the performance numbers for both the baseline and the realistic case.

Mean response time

For the *baseline case*, the mean response time of SRPT improves upon that of FAIR by a factor of 2 – 8 across the ten different workloads. More specifically, FAIR ranges from 300ms – 7.2 seconds, while SRPT ranges from 150ms – 1.2 seconds.

Under the *realistic case* (not shown), SRPT improves over FAIR by a factor of 1.4 – 4 across workloads W1, W2, W4, W5, W7, W8, W9, and W10. Note that we exclude workloads W3 and W6 throughout the discussion of the realistic scenario. This is because these two workloads have such a long overload period, that under the realistic scenario, their behavior mimics persistent overload. To see this, observe that, under the realistic scenario, which involves user aborts, there is an increased load due to multiple aborts and retries, which are especially prevalent in W3 and W6 because of their long overload period. This increased load causes the time-average load in these workloads to rise from 0.9 to over 1.0. Hence the system is running in persistent overload.

For both the baseline and realistic case, the mean performance under each workload is affected by (1) the mean system load and (2) the length of the overload and low load periods. Point (2) should be clear from looking at Figure 5.8, where workloads W3 and W6, which have the longest periods of overload, both stand out. Point (2) is also justified by considering workloads W1 and W2 which only differ in the length of their overload periods, resulting in a factor 5 difference in their mean response time, or workloads W4 and W5 which also differ only in the length of their overload period, again resulting in a factor 2 difference in their mean response times. The length of the overload period and the overall load also affect the number of SYN drops, and consequently the response times. While about half the workloads have zero SYN drops under FAIR and SRPT, under both the baseline and realistic setups, workload W3 which has 50 seconds of overload, results in 50% SYN drops under FAIR (zero SYN drops under SRPT), and consequently very high mean response times. SYN drops do not occur under SRPT under any of our workloads.



Mean Response Time - Baseline

Figure 5.8: *Comparison of mean response times under FAIR and SRPT in the baseline case for the workloads in Table 5.1. Peak response times are far worse than mean response times.*

Performance of large requests

Again, an obvious question to ask is whether this improvement in the mean response time comes at the price of longer response times for the requests for large files. In particular, for the workloads with higher mean system load and longer overload periods it might seem likely that SRPT leads to a higher penalty for the long requests.

We find that these concerns are unfounded. The mean response times of only the biggest 1% of all requests is never more than 10% higher under SRPT than FAIR for any workload in the *baseline case*, and is often lower under SRPT, and this penalty is further substantially diminished under the *realistic scenario*.

When considering the performance of large requests in the case of the *realistic setup*, it is important to also look at the number of incomplete requests (incomplete requests are only a consequence of user aborts). We observe that the lack of unfairness under SRPT in the realistic setup is not a consequence of a large number of incomplete large requests. The overall fraction of incomplete requests is only 0.2% for both FAIR and SRPT when load is 0.7 and ranges from 10 – 15% for both FAIR and SRPT when load is 0.9 (again excluding workloads W3 and W6). Looking only at the largest 1% of requests, the fraction of incomplete requests is much more variable, ranging from 3 – 25% under SRPT and 3 – 30% under FAIR, but is typically smaller under SRPT than under FAIR.

Finally, we observe that for both the baseline and the realistic setup, increasing the length of the overload period or the mean system load does *not* result in more starvation. This also agrees with the theoretical M/GI/1 results in [31].

5.4 Why does SRPT work?

In this section we will look in more detail at where SRPT's performance gains come from and we explain why there is no starvation of long jobs.

5.4.1 Where do mean gains come from?

Recall that we identified in Section 5.2 and Section 5.3 three reasons for the poor performance of a standard (FAIR) server under overload:

1) High queueing delays at the server due to high number of connections sharing the bandwidth, see Figure 5.2 and Figure 5.3.

2) Drops of SYNs because of full SYN queue,

3) Loss of packets inside the kernel.

SRPT alleviates all these problems. It reduces queueing delays by scheduling connections more efficiently. It has a lower number of dropped SYNs since it takes longer for the SYN-queue to fill. Finally and less obviously, an SRPT server also sees less loss inside the kernel. The reason is that subdividing the transmit queue into separate shorter queues allows SRPT to isolate the short requests, which are most requests. These short requests go to a queue which is drained more quickly, and thus experience no loss in their transmit queue.

The question we address in this section is how much of SRPT's performance gain can be attributed to solving each of the above three problems.

We begin by looking at how much of the performance improvements under SRPT stem from alleviating the SYN drop problem. In workload W1, used throughout the chapter, no incoming SYNs were dropped. Hence SRPT's improvement was not due to alleviating SYN drops. Workload W4 did exhibit SYN drops (1% in the baseline case and 20% in the realistic case). We eliminate the

SYN drop advantage by increasing the length of the SYN-queue to the point where SYN drops are eliminated. This only improves FAIR by under 5% for the baseline case and 30% for the realistic case – not enough to alone account for SRPT's big improvement over FAIR.

The remaining question is how much of SRPT's benefits are due to reducing problem 1 (queueing delays) versus problem 3 (packet drops inside the kernel). Observe that problem 3 is mainly a problem because of the high initial RTO. We can mitigate the effect of problem 3 by dropping the initial RTO to, say, 500 ms. The result is shown in Figure 5.6(F). Observe that even when problem 3 is removed, the improvement of SRPT over FAIR is still a factor of 7[4].

We therefore conclude that problem 1 is the main reason why SRPT improves upon FAIR. By timesharing among many requests, the FAIR scheduling policy ends up slowing down all the requests.

5.4.2 Why are long requests not hurt?

In this section we give some intuition for why, in the case of web workloads, SRPT does not unfairly penalize large requests.

To understand what happens under transient overload, first consider the overload period. Under FAIR, all jobs suffer during the overload period. Under SRPT all jobs of size $< x$ complete and all other jobs receive no service, where x is defined such that the load comprised of jobs of size $< x$ equals 1. While it is true that jobs of size $> x$ receive no service under SRPT during the overload period, they also receive negligibly-little service under FAIR during the overload period because the number of connections with which they must share under FAIR increases so quickly (see Figure 5.2). Next consider the low load period. At start of low load, there are many more jobs present under FAIR; only large jobs are present under SRPT. These large jobs have received zero service under SRPT and negligibly-little service under FAIR until now. Thus the large jobs actually finish at about the same time under SRPT and FAIR.

The above effects are accentuated under heavy-tailed request sizes for two reasons: (1) Under

[4]Problem 3 may seem to be a design flaw in Linux, that could be solved by either adding a feedback mechanism when writing to the transmit queue or by increasing the length of the transmit queue. However, this will increase the queueing delays that a packet might experience. Our experiments show that increasing the transmit queue up to a point slightly decreases response time, but beyond that actually hurts response time.

heavy-tailed workloads, a very small fraction of requests make up half the load, and therefore, the fraction of large jobs ($> x$) receiving no service during overload under SRPT is very small. (2) The little service that the large jobs receive during overload under FAIR is even less significant because the large jobs are so large that, proportionately, the service appears small.

5.4.3 Theoretical validation

As a final step in understanding the performance of FAIR vs. SRPT, we consider an M/GI/1 queue with alternating periods of overload and low load and derive an approximation on the expected response time as a function of request size for this model under FAIR and SRPT. The derivation is too involved to include herein, but the interested reader should look at [31] for full details. We choose to evaluate our results when the service requirement distribution, G, is a Bounded-Pareto distribution with α-parameter of 1.1, as has been shown to be representative of web workloads [32].

Although the M/GI/1 queue is at best a rough approximation to our implementation setup, we nevertheless find the same trends in analysis as we have witnessed in this chapter. In particular we find that the buildup in the number of requests is much greater under FAIR than under SRPT, and consequently response times are also far higher under FAIR than under SRPT. Likewise we find that the server "recuperates" more slowly when load is dropped under FAIR than under SRPT. Also similarly to our experiments, we find that in analysis the very largest requests see approximately the same mean response time under SRPT as compared with FAIR.

5.5 Summary

The work presented in this chapter breaks new ground in two different respects: it is the first to apply connection scheduling to improve performance of web servers under overload; and it is the first to provide an evaluation of the effect of external factors (Table 5.2) on the performance of an overloaded web server.

We implement SRPT in an Apache web server running on Linux by scheduling the bandwidth on the server's uplink. This is done by modifying the order that socket buffers are drained within the kernel. We find that SRPT significantly improves both server stability and client experience under persistent as well as under transient overload conditions. Under persistent overload, the number of connections at the FAIR server grows quickly compared with the buildup of connections under SRPT, and consequently the FAIR server is also quick to reach the point where incoming SYNs are dropped. As a result, the client experience in terms of mean response time and the time until the first byte is received, is greatly improved under the SRPT server compared to the FAIR server. With respect to transient overload, we find that SRPT improves mean response times by factors of 1.5 – 8 over the traditional FAIR scheduling, across ten different transient overload workloads. This is significant since mean response times under FAIR can get quite high, and peak response time are many-fold higher than mean response times.

Performance improvements are measured under a vast range of environmental factors including a range of RTT's, loss rates, RTO's, persistent connections, user behaviors, packet sizes, and web server configurations. Results are consistent across different traces. Importantly, we find that requests for large files are *not* penalized by SRPT scheduling under transient overload. In fact, for the largest 1% of requests, the response time under SRPT is very close to that under FAIR.

5.6 Acknowledgements

Thanks to Srini Seshan for help with porting dummynet to Linux; to Christos Gkantsidis for providing detailed information on `libwww`; to Mukesh Agrawal for helping with initial overload experiments; and to Jennifer Rexford, John Wilkes, and Erich Nahum for valuable feedback on this work.

Appendix: Another Trace

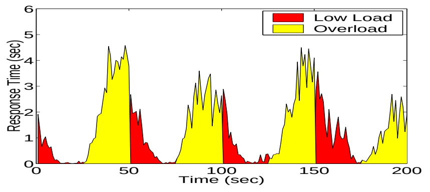
In this appendix we consider one more trace and run all experiments on the new trace. We find that the results are very similar to those shown in the body of the paper, both with respect to comparative mean response times for the different scenarios (A) through (J), and with respect to unfairness issues.

The log used here was collected from a NASA web server and is also available through the Internet Traffic Archive [101]. We use several hours of one busy day of this log consisting of around 100000 mostly static requests. The minimum file size is 50 bytes, the maximum file size is 1.93 Mbytes. The largest 2.5% of all requests make up 50% of the total load, exhibiting a strong heavy-tailed property. The primary statistical difference between the NASA log and the soccer World Cup log (used in body of the paper) is the mean request size: the NASA log shows a mean file size of 19 Kbytes while for the World Cup log it was only around 5 Kbytes.

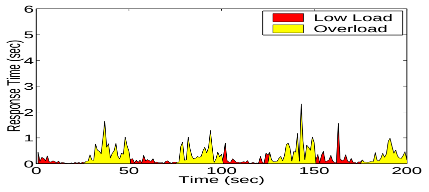
Results for the NASA log are shown in Figure 5.9, 5.10 and 5.11. They are extremely similar to the corresponding Figures 5.5, 5.6, 5.7 and 5.8 for the World Cup trace.

The only difference is that response times are higher under the NASA log as compared with the World Cup log for both FAIR and SRPT. The relative performance gains of SRPT and FAIR are similar. The increase in response times under the NASA log may be attributed to the higher mean file size.

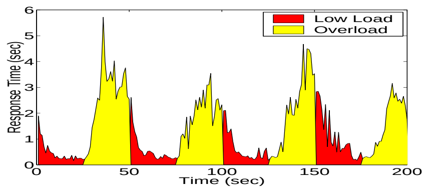
(A) FAIR - Baseline



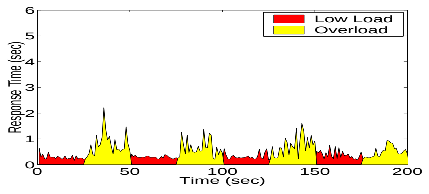
(A) SRPT - Baseline



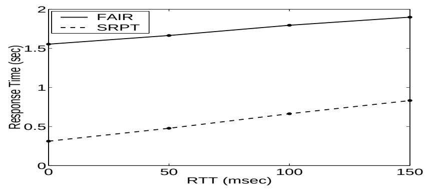
(B) FAIR - Delays



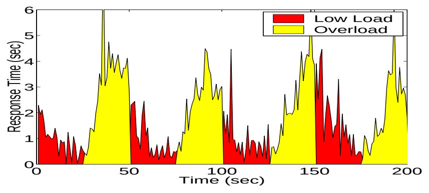
(B) SRPT - Delays



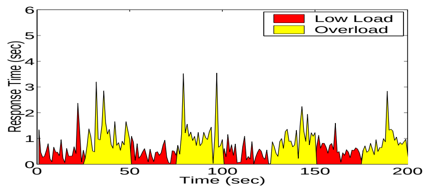
(B) Comparison



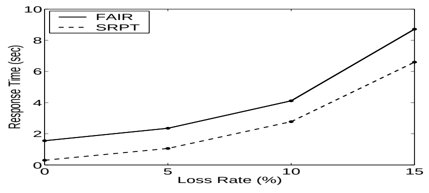
(C) FAIR - Losses



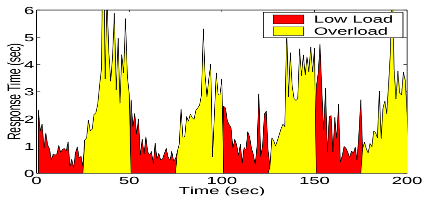
(C) SRPT - Losses



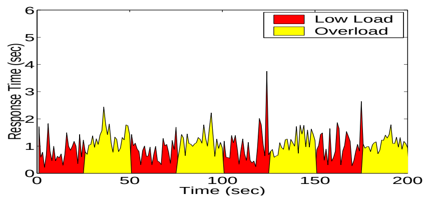
(C) Comparison



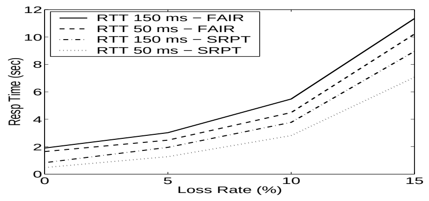
(D) FAIR - Delays and Losses



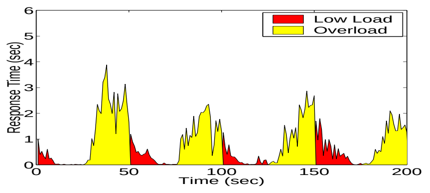
(D) SRPT - Delays and Losses



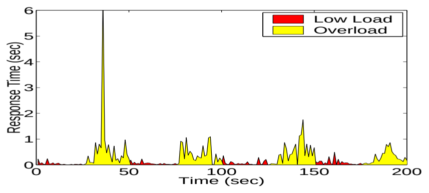
(D) Comparison



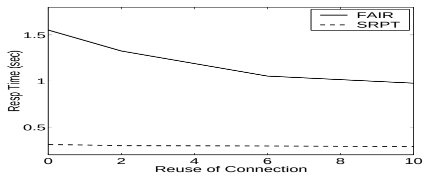
(E) FAIR - Persistent



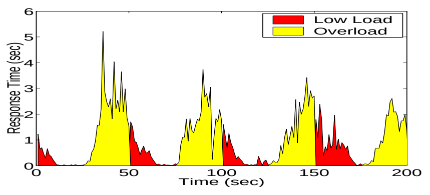
(E) SRPT - Persistent



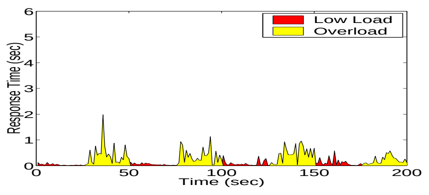
(E) Comparison



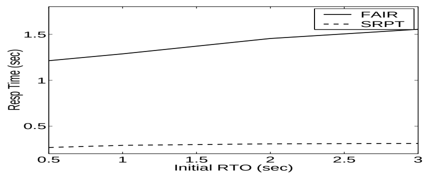
(F) FAIR - Initial RTO



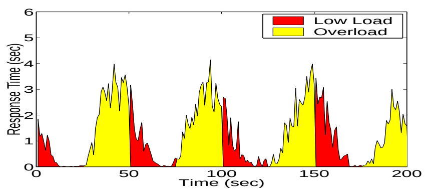
(F) SRPT - Initial RTO



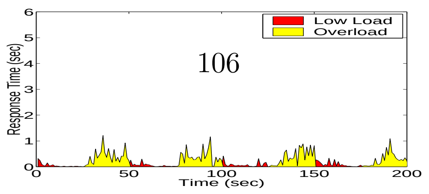
(F) Comparison



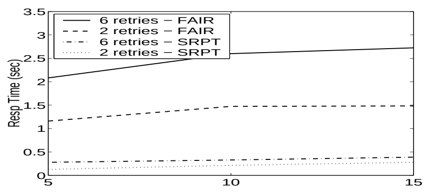
(H) FAIR - User Abort/Reload

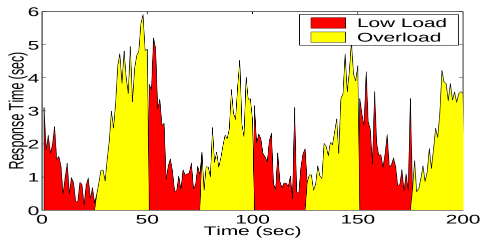


(H) SRPT - User Abort/Reload

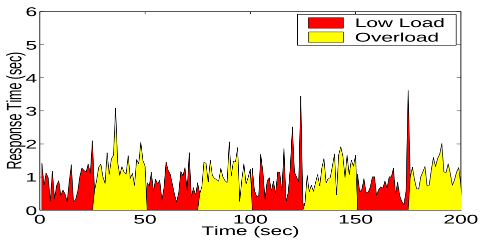


(H) Comparison





(J) FAIR - Realistic case



(J) SRPT - Realistic case

Figure 5.10: *Comparison of FAIR (left) and SRPT (right) for the realistic setup for workload W1, under NASA trace log.*

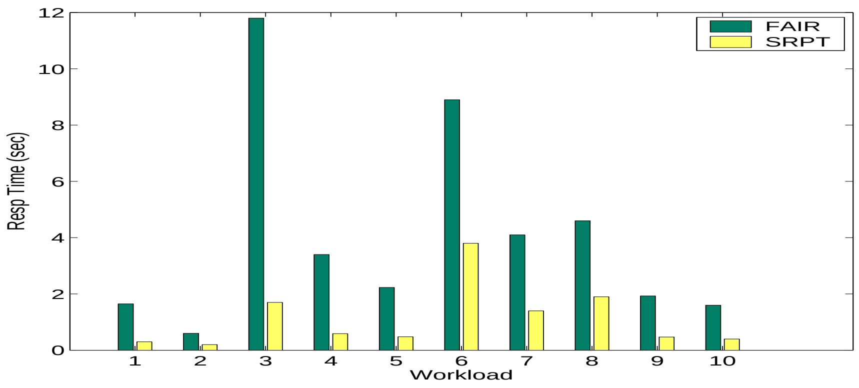


Figure 5.11: *Mean response time comparison of FAIR and SRPT in the baseline case for the workloads in Table 5.1, under the NASA trace log.*

Chapter 6

Conclusion and impact

The purpose of this chapter is to give a proof-of-concept that size-based scheduling is an effective method for reducing the average delay at a server, without starving or unduly penalizing any requests.

The theoretical advantages of SRPT with respect to mean response time have been understood for a long time in the queuing-theory community. While in the single queue context SRPT's advantages may seem obvious, SRPT's applicability to real web servers is not. First, it is not clear where or how SRPT should be implemented in a web server. Second, it is not clear whether SRPT is still effective in the context of network protocols such as TCP; under WAN conditions including propagation delay and loss; and under client/server behavior patterns – all of which are outside of the server's control. Third, even if SRPT does reduce mean response time, it is unclear what the penalty to requests for large files will be under SRPT, especially in the case of transient overload conditions.

The particular application we used in our work was a web server serving static content and the scheduling policy was an adaptation of the SRPT algorithm. However, the general idea of size-based scheduling to reduce response times is in no way limited to this particular setting. We believe that the basic principles we have demonstrated in our work will extend to many other scenarios. The following are the main requirements for directly applying the SRPT-like scheduling we presented in this chapter.

- Processing requirements can be reasonably well estimated (even a distinction between only two classes of short and long can be sufficient as seen in Section 4.2.2).

- The workload exhibits sufficient variability in the size distribution. This requirement is necessary to minimize unfairness to long requests/jobs under SRPT.

Even in systems where the above conditions are not met, there are many generalization possible to make the basic idea of size-based scheduling work. In the remainder of this section we will discuss optimizations and extensions for the general use of our scheduling framework.

6.1 Size-based scheduling and unknown sizes

While we show in our work that for static requests the size of the requested file is a good approximation for the size or service requirement of a request, for many other applications such estimates are not readily available.

In some cases it might be possible to predict service requirement of a request, based on history information of previous requests with similar characteristics. For example, we will see in Chapter 10 that the processing requirements of database accesses can often be well predicted based on the type of the transaction (e.g. "View shopping cart" or "Make payment" in e-commerce systems).

Alternatively, in environments without any a-priori knowledge one could deduce the run time of a dynamic request as it runs. The request is initially assigned high priority, but its priority will decrease as it runs. An example for this type of policy is the Least-attained-service (LAS) policy, which always works on the request that has received the least service. Rai et al. [168] find that for heavy-tailed job size distributions LAS favors short jobs with a negligible penalty to the few largest jobs, and that LAS achieves an overall mean response time close to that of SRPT.

Finally, while we find that for static requests file size is a good first approximation of the service requirements of a request, several optimizations are possible by taking more information into account. For example, the authors in [169] extend the work presented in this chapter by proposing and implementing a scheduling algorithm, which takes, in addition to the size of the request, the distance of the client from the server into account. They show that this new policy can improve the performance of large-sized files by 2.5 to 10%. Similarly, the work by Murta et al [152] extends SRPT to take WAN conditions in addition to file sizes into account. Their policy, called FCF (Fastest Connection First), gives priority to HTTP requests based on the size of the requested file and the estimated throughput of the user connection. They find that FCF gives the best mean response time, although SRPT provides the shortest server delay. Lu et al. [132] apply size-based scheduling in combination with a domain-based service time estimator that predicts the service time of a new request based on history information of previous requests. The history information includes the file size, service time and IP address of previous requests. The prediction is made based on the assumption that the service time of a new request will be similar to previous requests with a similar file size that come from the same neighborhood (domain) in the network topology.

6.2 Size-based scheduling and fairness in other applications

Our work shows that for static web workloads biasing against long requests improves overall mean response time, without significantly penalizing long requests. This result relies on the heavy-tailed distribution of requested file sizes at web sites. Other work that has been carried out in parallel provides a theoretical validation of this result. Bansal et al. [30] show that for an M/G/1 queue with heavy-tailed job size distributions the degree of unfairness under SRPT is very small.

While the workloads of many different applications have been found to exhibit heavy tails, it is still an important question how SRPT performs for general applications, potentially with non-heavy-tailed workload distributions. In practice this question arises for example when several servers are available, and requests are routed to servers based on their size, e.g. one server is reserved only for large requests. Even if the original size distribution is heavy-tailed, the distributions seen by the individual servers can have much lower variability. Towards this end Bansal et al. [30] provide theoretical bounds on the performance of requests under SRPT compared to PS for *general distributions*. They show for example that the expected response times of any job (including the very largest one) under SRPT is never more than 3 times that under PS, when the load is 0.8, and never more than 5.5 times that under PS, when the load is 0.9.

Below we discuss approaches for making size-based scheduling more attractive to applications where performance of large requests is very critical and theoretical bounds like the ones above are not sufficient.

First note that our work covers an entire family of algorithms, rather than one particular algorithm. The kernel-level implementation we have chosen is limited to a fixed number of priority classes, where we pick the size cutoffs between the different priority classes so as to approximate SRPT as closely as possible. Instead one could also use more conservative cutoffs and/or fewer priority classes to achieve performance closer to a FAIR system, including better performance for very large requests.

The idea of using policies that are hybrids between SRPT and FAIR scheduling is investigated in detail by Gong and Williamson [89]. They propose and study two particular hybrid policies and show in simulations that they provide a smooth trade-off between the responsiveness of SRPT and the fairness of PS. In addition to their study of hybrid scheduling policies, Gong and Williamson also investigate the fairness properties of SRPT scheduling in more detail [88]. They identify two different types of unfairness: endogenous unfairness, that a job may suffer because of its own size, and exogenous unfairness, that a job suffers as a consequence of the other jobs it sees when it arrives. They find that jobs experience a smaller degree of exogenous unfairness under SRPT scheduling compared to the standard FAIR (processor sharing) scheduling, which means response times under SRPT are more predictable. This makes SRPT a good choice for environments where predictability of response times is important.

Finally, Friedman and Henderson develop a new size-based policy that provides provable guarantees on the performance of large requests, independent of the file size distribution. The policy is called Fair-Sojourn-Protocol (FSP) and guarantees every single request (including those for large files) a response time not larger than the same request would experience under FAIR scheduling (processor sharing).

6.3 Extension to other types of requests and server systems

While in this chapter we demonstrated the use of size-based scheduling for web servers serving static content, the same idea can be applied to many other types of requests and server systems.

One possible avenue for future research is to expand our technology to *dynamic* web requests that involve for example running cgi-scripts or accessing a database back-end. An important problem in applying size-based scheduling to those workloads is that of determining the processing requirement of dynamic requests. As we alluded to in Section 6.1, the processing requirements of dynamic requests that are based on database accesses can often be well predicted based on the type of the transaction (e.g. “View shopping cart” or “Make payment” in e-commerce systems).

Our work could also be extended to other bottleneck resources. Our current setup considers the bottleneck resource at the server to be the server’s limited bandwidth purchased from its ISP, and thus we do SRPT-based scheduling of that resource. In a different application (e.g. processing of cgi-scripts) where some other resource was the bottleneck (e.g., CPU), it might be desirable to implement SRPT-based scheduling of that resource.

Moreover, our SRPT solution could also be applied to web server farms, rather than a single server. Again the bottleneck resource would be the limited bandwidth that the web site has purchased from its ISP. SRPT-based scheduling could then be applied to the router at the joint uplink to the ISP or at the individual servers.

Our belief that size-based scheduling has much broader applicability has been reinforced by several recently published studies that extend our work to other applications.

The work by Yang et al. [212] extends size-based scheduling to general network transfers, beyond HTTP requests only. They integrate a bandwidth allocation criteria that depends on the residual work of on-going transfers into TPC-Reno. They find that their approach not only improves user experience, but also achieves better overall network goodput.

Rai et al. [168] investigate size-based scheduling for network flows at the router on a bottleneck link in the network. Since a router misses the application-level knowledge that a web server has to estimate the length of a given flow, they propose the Least-attained-service (LAS) policy which doesn’t require a priori knowledge of the service demand.

The work by Lu et al. [133] proposes the use of size-based scheduling policies for peer-to-peer networks and network backup systems.

6.4 Size-based scheduling and overload

Much of systems research has focused on experiments where the load is stable and stays within the system capacity. Experimenting with fixed load levels is easier since experimental results exhibit little variability and the request stream can easily be generated by a single client machine using a simple workload generator. In this chapter we have seen that the response times of a web server can vary widely depending on whether a given system load is generated through a steady arrival stream, or an arrival stream with varying load. For example, if a system load of 0.9 is created through an arrival stream with steady intensity, the mean response time is less than 1 second. If the same average system load is created through alternating high and low load periods with $\rho_{high} = 1.6$ and $\rho_{low} = 0.2$ the response times for the same system rises to 7 seconds.

While our work deals only with the effects of transient overload on a single web server serving static content, other systems are also prone to the problem of transient overload. For example, transient overload is likely to be intensified in server farms, where bursts in the incoming request stream will exacerbate any imperfections in the load balancing scheme. In systems serving dynamic web content, the application server or the database back-end server are as prone to transient overload as the front-end web server under static load. For example, the work of Zhang et al. [216] shows that during burst times overload propagates through all tiers of the system, all the way back to the storage back-end server.

In this chapter we have demonstrated how size-based scheduling can minimize the effects of transient overload in a web server serving static content. Intuitively, the advantage of size-based scheduling during transient periods of overload is that it minimizes the number of active requests in the system. Our solution relies on knowing or estimating the size or service requirement of a request and on the size distribution being heavy-tailed. While these conditions are met in our setting of a web server serving static web content, they might not be in other systems experiencing overload. For example, if one wants to apply size-based scheduling to schedule network flows at routers, the flow length is not known a-priori by a router. Similarly, in processing cgi-scripts or database transactions involved in dynamic web content execution times are not known a-priori. Moreover, even though many types of applications have been shown to exhibit heavy-tailed behavior, for most server systems it's unrealistic to assume that they will never run a workload with low variability (in which case fairness might be compromised).

Fortunately, most of the approaches we explained earlier in this section for generalizing SRPT to other systems and workloads, also apply to the overload case. For example, when job size estimates are not available, a policy like LAS (Least-attained-service [168]) can be used instead of SRPT to bias against long jobs. In cases where fairness is an extreme concern, employing a hybrid policy as suggested in [89] or scheduling according to the FSP protocol [83] can be an attractive solution.

However, the overload case is unique in several ways. First, even under the best possible scheduling some requests might have to be dropped if the overload period lasts too long. There are different ways to choose which request to drop. One way is to drop the requests with the largest service requirements, as suggested by [59, 202], an approach that is in nature very similar to our SRPT-based scheduling scheme. For commercial web sites, it often makes sense to select the request that will yield the least profit for the company. Toward this end Cherkasova et al. [63] present an approach that uses session-based information to ensure that requests that are part of a long session can be completed. The underlying assumption is that long sessions are likely to finally result in a purchase.

Another aspect of size-based scheduling that is particularly important under overload is that of reducing the variability in response times. Several user studies [73, 217, 44, 39] indicate that reducing unpredictability in response times can be more important to users than reducing average response times themselves, because waiting much longer than expected causes far more user frustration than simply waiting longer on average. Users get over time used to receiving a certain level of service from a particular web site and feel "betrayed" [44, 39] if they don't receive the service they are expecting. Moreover, unpredictable response times can taint the image of the web site as a while, since users tend to translate their experience with one aspect of the web site (such as unpredictable response times), to other aspects of the web site, such as security or business management.

Another benefit of size-based scheduling in the case of overload is that it improves network utility and interactions with the underlying network protocols. Because of the way TCP is designed, already a few packet losses due to full queues at a server or a network router, can lead to a time-out on the order of tens of seconds. Delays on this order are likely to cause a user to abort and restart a new connection attempt, exacerbating the problem of high load. Size-based scheduling masks the effects of overload for the majority of requests (that are for small files) leading to much fewer aborted and restarted connections. For the same reason, applying SRPT-like scheduling at routers in the network can increase network goodput during times of excessive load [212].

Finally, size-based scheduling is a valuable tool for improving service levels during overload, since the effectiveness of many other approaches for improving web performance is limited in the case of overload. For example, during normal load patterns overprovisioning is a powerful tool to reduce load and thereby response times. However, it is difficult to apply this technique for overload, since it would require predicting (1) the time when overload will kick such that system upgrades can be made in time and (2) the level of traffic increase to decide on the right amount of resources to add to the system. Both are difficult problems since system load during overload is often orders of magnitudes larger then during normal operation, meaning that mistakes in the predictions can lead to extremely wasteful use of resources.

Part II

Providing differentiated quality of service for OLTP workloads

Chapter 7

Introduction to Part II

Providing good quality of service (QoS) is crucial in serving dynamic content for several reasons. First, one of the main applications of dynamic content is in e-business web sites. For an e-business web site the competitor is only one click away making it very easy for dissatisfied customers to take their business to a competitor. Second, poor quality of service affects the image of the company as a whole. Studies have shown that users equate slow web download times with poor product quality or even fear that the security of their purchases might be compromised.

How users perceive the quality of service they experience when accessing a web site depends on several factors:

- The latency experienced by users

  When users rate the quality of service received at a web site as poor, the most common reason is high latency. Latency, or response time, is defined as the period of time between the moment users make a request and the time they receive the response in its entirety. In order for a response to be perceived as immediate, the latency needs to be on the order of 0.1 seconds. For a user to be satisfied, delays should not exceed 5 seconds. Delays that are longer than 10 seconds are considered intolerable and lead users to assume that an error has occurred in processing their request.

- The predictability of system performance

  Frequent customers at a site are used to a certain level of service. If the quality of service is not delivered as expected users may feel betrayed. Unpredictable service compromises customers' trust in the company.

- The overall system availability
  The worst quality of service is no service. Even short service outages can cost an e-business huge amounts in lost revenues. One common source of service outages is transient overload at the web site due to an unexpected surge of requests. Examples of this phenomenon include the overload of the Firestone web site after a tire recall, the outage of several large e-tailers during the holiday shopping season in 2000, and overload of the official Florida election site, which became overwhelmed after the presidential election of 2000.

- The freshness of data returned to the user
  Freshness of data is a quality of service aspect that is particular to dynamic content. It arises when web sites, in an attempt to serve a dynamic request more efficiently, rely on cached data from an earlier execution of the dynamic request. This approach is for example commonly used when serving dynamic requests that require access to a database back-end. While using cached data reduces work for the server, users might receive outdated data. Consider, for example, an e-commerce site that reports product availability based on cached values of stock levels. The site might realize only after accepting an order that the item is not actually in stock.

The main factor affecting the quality of service is typically the system load a site experiences: the higher the system load the lower the quality of service experienced by the clients. Guaranteeing a certain level of service to all clients therefore requires keeping the system load below some threshold. This can be achieved through the same methods we described in Section 2.3 for overload control, namely admission control or content adaptation. Admission control selectively rejects some of the incoming requests if the load threshold is reached. In content adaptation a server switches to serving lower quality content, if the system load gets too high.

Rather than guaranteeing a certain unified level of service across all requests (potentially at the cost of rejecting some requests), it is often more important for an e-commerce site to offer *differentiated* levels of service. The reason is that requests vary in how important they are and users vary in how tolerant they are to delays. For example, it is in the economic interest of an e-commerce retailer to differentiate between high-volume customers, who are "big spenders", and low-volume customers. Moreover, some customers may have payed for service level agreements (SLAs), which guarantees a certain level service. Finally, studies [44, 39] indicate that customers' patience levels decline in proportion to the time spent at the site. Patience levels are also tied to the type of request users submit. Customers tend to be more tolerant of long search times and less tolerant of long purchase times.

The network community has long studied the efficacy of providing quality of service guarantees and class-based service differentiation. However, much of this work relies on the assumption that network is the typical bottleneck in web transfers. Creating dynamic content at a web site takes orders of magnitudes longer than serving static content [54], and results in a much larger burden on the web server. For web sites that are dominated by dynamic content the bottleneck tends to shift from the network to the server, making it necessary to provide QoS mechanisms at the server.

The goal of this part of the thesis is to develop methods for providing differentiated quality of service for dynamic web requests. We will focus on metrics related to latency and predictability of service, and consider two differenty types of differentiated quality of service.

- Best effort performance differentiation

  Different classes with different priorities, where higher priority classes should receive better service than lower priority classes.

- Absolute guarantees

  Each class has a concrete QoS goal that it needs to meet, e.g. a latency target specified in number of seconds.

In the remainder of this chapter, we will first survey the state of the art in providing best effort performance differentiation and absolute guarantees. We will then introduce the goals of our work.

7.1 Best effort performance differentiation

Methods for providing best effort performance differentiation typically follow one of the following three approaches:

1. providing different quality of content depending on the priority of a request.
2. changing the order in which requests are processed based on request priorities.
3. adapting the rate at which a request is served according to the request priority.

The first approach can be implemented by applying the methods for content degradation described in Section 2.3.2, where the degree of content degradation for a given request is chosen based on the priority of the request. This approach is for example taken in the work by Bhatti et al. [10].

The second approach can be implemented by changing the order in which kernel and application level queues are scheduled (the processing order of requests is changed based on request priorities). For example the work by Voigt et al. [204] prioritizes the kernel SYN and listen queue. This approach is limited to the case where class priorities can be determined based on the client IP address, since at this point no application level information is available. Other work [40, 41] therefore suggests to add an application level queue between the TCP listen queue and the web server and to schedule this queue according to request priorities.

Mechanisms for changing the rate at which requests are processed (approach 3 in the above list) include limiting the number of web server processes available to low priority requests and adjusting the operating system priorities of web server processes [77]. The applicability of both of these mechanisms is limited in that they assume a process-based server architecture.

All the above work deals only with requests that do not access a database backend. When it comes to database-driven requests, a web site depends on the support of the underlying database management system (DBMS). Most commercial DBMS ship with tools that allow the administrator to assign priorities to transactions in order to affect the rate at which transactions is processed. However, the implementation of those tools usually relies on CPU scheduling [172, 99] and is therefore most effective when applied to CPU bound workloads. Database researchers have explored scheduling of transactions in the context of real-time database systems (RTDBMS). RTDBMS rely on their specialized architecture with features such as optimistic concurrency control mechanisms, which are not available in the general purpose database systems used as web server back-ends.

7.2 Achieving absolute guarantees

The most common approach for achieving absolute delay guarantees is to combine admission control with some form of feedback control. At a high level, admission control is applied to either kernel-level or application-level queues, and based on observed per-class response times a control loop is used to adjust the drop rate of low priority requests if high priority classes miss their response time targets. This approach is for example taken in the following papers, [59, 60, 165, 108]. Some of this work assumes that the server provides isolation among service classes, i.e. the server fairly distributes system resources among classes.

The above work focuses only on requests that are not database driven. There is relatively little work in the area of database systems for supporting per class QoS targets. Most work that is concerned with providing performance guarantees is in the large domain of real-time database systems (RTDBMS). The goal in RTDBMS is not improvement of mean execution times for high priority classes of transactions, but rather meeting (usually hard) deadlines associated with each transaction. In achieving this goal RTDBMS often rely on their specialized architecture with features such as optimistic concurrency control mechanisms, which are not available in the general purpose database systems used as web server back-ends.

The existing work in the area of general purpose databases systems for providing multi-class response time goals relies on buffer management strategies [47, 48, 186], however as of now these approaches have not been evaluated for web-driven database workloads.

7.3 Our goal

Access to a DBMS backend for dynamic query processing and data generation is a crucial part in serving dynamic content, since this is where the business logic of an e-commerce web site is implemented. Unfortunately, serving requests which involve database activity can be very slow — orders of magnitude slower than delivering static content. This slowness is exacerbated under heavy load and overload.

The costliness of database accesses create a strong motivation to provide differentiated quality of service to ensure that at least "important" clients recieve good service at any time. The importance of providing differentiated quality of service in e-commerce environments is well-understood and many possible solutions have been developed, as described in Section 7.1 and Section 7.2. However, most existing solutions either focus on static content served by a web server front or dynamic content served by an application server. Very little work exists for supporting differentiated quality of service at DBMS backend servers. The goal of this part of the thesis is to fill this gap.

We will begin in Chapter 8 by investigating how prioritization for transactional workloads can be integrated into general purpose DBMS. Since an effective prioritization scheme will need to target the bottleneck resource, we first perform a detailed bottleneck analysis of transactional workloads on commercial and noncommercial DBMS (IBM DB2, PostgreSQL, Shore). Second, we implement and evaluate the performance of several preemptive and non-preemptive DBMS prioritization policies in PostgreSQL and Shore.

In Chapter 9 we ask whether comparable prioritization differentiation can be achieved via *external* DBMS scheduling, i.e., without touching any DBMS internals. All scheduling is done as a front end to the DBMS by temporarily delaying certain transactions (but not dropping transactions, as is done in admission control). The advantage of external scheduling is that it is extremely portable (usable with any DBMS), as well as being easy to implement, and independent of the bottleneck resource. The drawbacks to external scheduling are (i) a potential drop in throughput (as the multiprogramming level is lowered), (ii) a potential increase in overall mean response time, and (iii) it's not obvious how effective external prioritization can be in comparison with internal prioritization (as external scheduling provides no control over internal DBMS resources). We provide a detailed sensitivity study of the potential drawbacks of external scheduling across many workloads, bottlenecks, and DBMS configurations. We find that external scheduling with the right is able to produce priority differentiation results similar to those using internal scheduling, with only very limited drop in throughput and negligible increase in mean response time.

In Chapter 10 we extend our work on external scheduling to more complex QoS goals including (i) Guaranteeing mean response time targets for different classes of transactions; (ii) Percentile guarantees, where at most $x\%$ of response times are above y; (iii) Lowering overall variability in response times across all classes.

Chapter 8

Providing priority mechanisms for OLTP workloads inside the DBMS

"Which resources inside the DBMS need to be scheduled in order to provide transaction prioritization and what are the right scheduling policies? "

The need for prioritized service in e-commerce sites is well-understood, as not all customers are of equal importance, and thus some should receive higher priority service over others. As the DBMS becomes a central part of more and more sites, and given the relatively large fraction of time spent in the DBMS, it becomes increasingly important to look at prioritization specifically for DBMS transactions.

The scenario motivating our work is a three-tiered e-commerce web site with a backend DBMS. It is a common phenomenon at e-commerce web sites that a small fraction of the shoppers at the site spend a large amount of money, whereas the remaining shoppers spend a small amount of money. It makes sense from an economic perspective to prioritize service to the "big spenders," providing them with lower mean delays. As another example, online financial services might want to offer preferred status to some customers, providing them with high priority service as part of a service level agreement, or as a bonus.

The goal of this chapter is to provide prioritization and differentiated performance classes within a traditional (general-purpose) relational database system running OLTP and transactional web workloads, including read/write transactions. We provide a detailed resource utilization breakdown for OLTP workloads executing on a range of database platforms including IBM DB2[122], Shore[157], and PostgreSQL[164]. IBM DB2 and PostgreSQL are both widely used (commercial and noncommercial) database systems. Shore is an open source research prototype using traditional two-phase locking (2PL), the concurrency control used in DB2. PostgreSQL (like Oracle), on the other hand, uses multiversion concurrency control (MVCC) [37]. We also implements several transaction prioritization policies within Shore and PostgreSQL. The prioritization policies studied include non-preemptive priorities, non-preemptive priorities with priority inheritance, and preemptive abort scheduling. Given the focus on web and complex transactional applications, we use the benchmark OLTP workloads TPC-C and TPC-W.

The primary contributions of this research are twofold:

1. Identification of bottleneck resource(s) across DBMS, workloads and concurrency levels.
2. Demonstration that simple priority scheduling inside the DBMS significantly improves high-priority transaction execution times without penalizing low-priority transactions.

With respect to bottleneck identification, we show that the bottleneck resource for TPC-C on IBM DB2 and Shore, both of which use 2PL, is lock waiting. By contrast, for the same TPC-C workload, PostgreSQL, which uses MVCC, exhibits an I/O synchronization bottleneck. For TPC-W on DB2 and PostgreSQL, we find that the bottleneck is always the CPU.

On the issue of scheduling policies, we find that scheduling of bottleneck resources results in improving high-priority transaction execution times considerably. For systems with lock bottlenecks (TPC-C on DB2 and Shore), CPU scheduling is ineffective, but lock scheduling can improve high-priority performance by a factor of 5.3. For systems with CPU bottleneck (TPC-W), lock scheduling is ineffective, while CPU scheduling improves high-priority performance by a factor of 4.5. For PostgreSQL, which has an I/O synchronization bottleneck, CPU scheduling with priority inheritance yields a factor of 6 improvement of high-priority transactions. Provided that the fraction of high-priority transactions is small, the penalty to the low-priority transactions is negligible as long as preemption is not used.

8.1 Existing approaches

There is a wide range of well-known database research, including that of Abbott, Garcia-Molina, Stankovic, and others, studying different transaction scheduling policies and evaluating the effectiveness of each. Most existing implementation work is in the domain of real-time database systems (RTDBMS), where the goal is not improvement of mean execution times for classes of transactions, but rather meeting deadlines associated with each transaction. These RTDBMS are sufficiently different from the general-purpose DBMS studied in this chapter to warrant investigation as to whether results for RTDBMS apply to general-purpose DBMS as well. In addition to the existing implementation work in RTDBMS, there has also been work on simulation and analytical modeling of prioritization in DBMS and RTDBMS. Unfortunately, the simulation and analytical approaches have difficulty in capturing the complex interactions of CPU, I/O, and other resources in the database system.

In Section 8.1.1 we summarize the most relevant existing research on transaction prioritization within RTDBMS. In Section 8.1.2 we summarize the existing and ongoing work on prioritization in general-purpose DBMS.

8.1.1 Real-Time Databases

Real-time database systems (RTDBMS) have taken center stage in the field of database transaction scheduling for the past decade. These systems are useful for numerous important applications with intrinsic timing constraints, such as multimedia (*e.g.*, video-streaming), and industrial control systems. Traditional DBMS with transaction priorities differ from RTDBMS. In RTDBMS, each transaction is associated with time-dependent constraints (usually deadlines), which must be honored to maintain transactional semantics. The goal of minimizing the number of missed constraints (deadlines), requires maintaining time-cognizant protocols and various specialized data structures [196], unlike general-purpose DBMS. Scheduling issues such as priority inversion may have different costs for RTDBMS as compared to traditional DBMS: *i.e.*, a single priority inversion may cause a missed deadline while hardly affecting overall mean execution time. Lastly, RTDBMS workloads can differ substantially from traditional DBMS workloads.

Abbott and Garcia-Molina [6, 5, 7, 8, 9] extensively study scheduling RTDBMS in simulation, preemptively and non-preemptively scheduling the critical resources (CPU, locks and I/O) to meet real-time deadlines. On the question of which resource needs to be scheduled, Abbot and Garcia-Molina conclude that CPU scheduling is most important, as transactions only acquire resources when they have the CPU [9]. Additionally, they find scheduling of concurrency control resources also improves performance.

With respect to scheduling policies, both Abbott and Garcia-Molina [9] and Huang et. al. [98] examine priority inheritance and preemptive prioritization in RTDBMS that use 2PL, to address the priority-inversion problem. Abbott and Garcia-Molina find that priority inheritance is important when ensuring that deadlines are met, in particular when the database is small. In contrast, Huang et. al. find that standard priority inheritance is not very effective in RTDBMS.

Kang et. al. [107] differentiate between classes of real-time transactions, providing different classes with QoS guarantees on the rate of missed deadlines and data freshness. They focus on main memory databases.

Our results will differ from those above as follows: (i) CPU is not always the most important resource to schedule. For DBMS using 2PL and TPC-C workloads we see that scheduling locks is far more effective than CPU scheduling. (ii) Priority inheritance is not always necessary, and is ineffective for some workloads and DBMS.

We attribute these differences in results to the many differences between real-time and traditional DBMS and their workloads.

8.1.2 Priority Classes

Existing work to establish priority classes for mean performance (rather than meeting specific deadlines), can be divided into techniques which schedule transactions (i) outside the DBMS and (ii) inside the DBMS. External scheduling is typically implemented using admission control to prevent transactions from entering the DBMS. Internal scheduling, by contrast, prioritizes transactions as they execute within the database.

Recent work at IBM implements priority classes in admission control [78]. The approach makes admission control decisions based not only on the number of transactions in the DBMS, but also on transaction priorities, by limiting the number of low-priority transactions that are able to interfere with high-priority transactions. Such admission control reduces lock contention and also limits inefficiencies introduced when the system is under overload, such as virtual memory paging and thrashing. Consequently, high-priority transactions under overload can benefit significantly.

There is much room for further research in transaction scheduling internal to the DBMS. The most pertinent work, by Carey et. al. [52] is a simulation study of our same fundamental problem: evaluating priority scheduling policies within DBMS to improve high-priority transaction performance. They assume a read-only workload, but recommend mixed read/write workloads should also be examined in the future. In contrast, our work assumes mixed read/write workloads and our work uses fully implemented DBMS rather than a simulator.

Brown et. al. [47] address multiclass workloads with per-class response time goals. Again, this is a pure simulation study without experimental validation on a DBMS prototype. Moreover, it focuses on a single resource, memory, while in our work we analyze the resource breakdown for different DBMS and workloads and consider the different bottleneck resources.

Prioritization within traditional DBMS has not been a focus for academic research. As a testament to the importance of the problem, however, both IBM DB2 and Oracle provide prioritization tools (IBM DB2gov and QueryPatroller [122, 4] and Oracle DRM [172]), all of which focus on CPU scheduling. We have experimented extensively with IBM DB2gov, and find it does not provide nearly as large of a prioritization benefit for the lock-bound workloads discussed in this chapter. This chapter addresses a wider range of scheduling policies for both CPU and lock resources.

8.2 Experimental Setup

This section describes experimental setup details including the workloads, hardware, and software used.

8.2.1 Workloads

As representative workloads for OLTP and transactional web applications, we experiment with the TPC-C [69] and TPC-W [198] (TPC-W Shopping Mix) benchmarks.

The TPC-C workload implementation for DB2 and PostgreSQL is written and graciously donated by IBM. The TPC-C Shore implementation was written at CMU. TPC-C is modified to allow each client to access a different warehouse and district for each transaction, which produces more uniform access to the database. The TPC-W workload comes from the PHARM [49] project with minor improvements, such as an improved connection pooling algorithm.

8.2.2 Hardware and DBMS

All of the TPC-C experiments for DB2 and Shore are performed on a 2.2-GHz Pentium 4 with 1GB RAM, one 120GB IDE drive, and a 73GB SCSI drive. The TPC-C PostgreSQL experiments are conducted on a comparable machine with two 1-GHz processors and 2GB of RAM, allowing us to handle the larger memory requirements of PostgreSQL. The results for PostgreSQL on the dual-processor (two 1-GHz) machine are similar to those when performed on the single-processor 2.2-GHz machine used by DB2 and Shore. The TPC-W experiments are all conducted with the database running on the 2.2-GHz machine; the web server and Java servlet engine run on a Pentium III, 736Hz processor with 512MB of main memory; and the client applications run on two other machines. The operating system on all machines is Linux 2.4.

The DBMS we experiment with are IBM DB2 [122] version 7.1, PostgreSQL [164] version 7.3, and Shore [157] interim release 2. Several modifications are made to Shore, to improve its support for `SIX` locking modes, and to fix minor bugs experienced in transaction rollbacks.

8.3 The Bottleneck Resource

Central to this work is the idea that understanding a workload's resource utilization is essential for effective prioritization. In order to improve high-priority transaction execution times, the *bottleneck resource*, where transactions spend the bulk of their execution time, must be scheduled, either directly or indirectly. Given the complexity of modern database systems, predicting the bottleneck resource is non-trivial.

In this section, we derive resource utilization breakdowns and determine the bottlenecks for TPC-C on Shore, DB2, and PostgreSQL and for TPC-W on DB2 and PostgreSQL. First, we describe the model used for breaking down transaction resource utilization. Next, we examine how these resource breakdowns change under varying concurrency levels and database sizes.

8.3.1 DBMS Resources: CPU, I/O, Locks

Since the goal of this chapter is to improve individual transaction execution times, and not overall throughput, it is important to break down execution times from the point of view of a transaction. We focus on three core DBMS resources: CPU, I/O, and locks, chosen since they are under control of the database, and are believed to be important in performance [9].

We define the total execution time of a transaction, T_{Trans}, as the time from when the transaction is first submitted to when it completes. We break T_{Trans} into three components, $T_{Trans} = T_{CPU} + T_{IO} + T_{Lock}$, corresponding to CPU, I/O, and locks, respectively. These components consist of just the synchronous time in which the transaction is completely dedicated to either waiting for or consuming the corresponding resource. T_{CPU} consists of the time spent running on the processor and the time spent in the running state, waiting for the processor. T_{IO} consists of the time spent issuing and waiting for synchronous I/O to complete (although the cost of issuing an I/O operation is negligible). T_{Lock} is the time that a transaction spends waiting for database locks. Of course, time spent holding locks is accounted according to whether the transaction holding the lock is waiting for or consuming CPU or I/O or waiting for another lock.

Database locks are broken into "heavyweight" and "lightweight" locks. Heavyweight locks are used for logical database objects, to ensure the database ACID properties. Lightweight locks include spinlocks and mutexes used to protect data structures in the database engine (such as lock queues). We find that lightweight locking is not a significant component of transaction execution times in either Shore or IBM DB2. PostgreSQL, however, has significant lightweight lock waiting, due to an idiosyncrasy of the PostgreSQL implementation. We find almost all lightweight locking in PostgreSQL functions to serialize the I/O bufferpool and Write-Ahead-Logging activity (via the `WALInsert`, `WALWrite`, and `BufMgr` lightweight locks). As a result, we attribute all the lightweight lock waiting time for the above-listed locks to I/O. We use the term "locks" throughout the remainder of this chapter to refer exclusively to heavyweight locks.

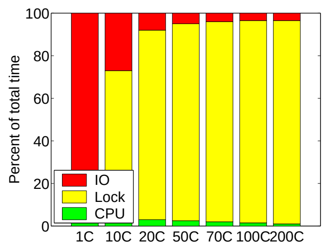
We use two different methods to obtain the desired resource breakdowns, depending on the DBMS used. For DB2, since its source code is unavailable, we rely on its built-in resource measurement facilities: snapshot and event monitoring [122]. For PostgreSQL and Shore, we implement custom measurement functionality by instrumenting the DBMS itself. We compute the total CPU, I/O and lock wait time over all transactions and then determine the fraction each component makes up of the sum of all execution times.

For DB2 and PostgreSQL, which use a process-based architecture, we verify the breakdowns at the operating system via the `vmstat` command, recording the fraction of time DBMS processes spend in the CPU run queue (`TASK_RUNNING`), blocked on I/O (`TASK_INTERRUPTIBLE`), or waiting for locks (`TASK_UNINTERRUPTABLE`). We also use a patch to the Linux kernel to accurately measure CPU wait times (not measured in Linux by default).

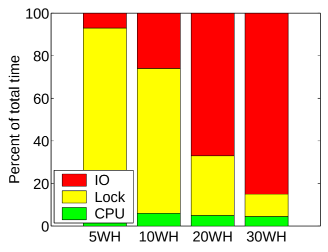
8.3.2 Breakdown Results

TPC-C. Figure 8.1 shows the resource breakdowns measured for TPC-C running on IBM DB2, PostgreSQL, and Shore. The graphs depict the average portions (indicated as percentages) of transaction execution time due to CPU, I/O, and lock resource usage. Although breakdowns are normalized to 100%, there is a small measurement error of less than 10% for DB2. We attribute the error to the high-granularity of DB2's I/O and CPU measurements.

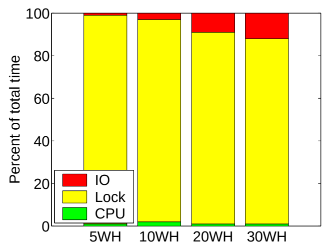
For each DBMS, Figure 8.1 presents three sets of results illustrating the most significant trends. In the first column, the database size is held constant at 10 warehouses (WH), and the number of clients connected to the database (concurrency) is varied. In the second column, the number of clients is held constant at 10, and the size of the database is varied by increasing the number of warehouses. In the third column, we vary the number of clients and warehouses together, always holding the number of clients at 10 times the number of warehouses, as specified by TPC-C, demonstrating



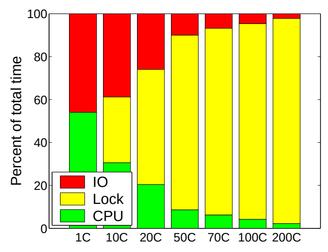
(a) **DB2**: Varying Clients, 10 Warehouses



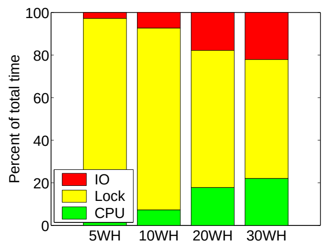
(b) **DB2**: 10 Clients, Varying Warehouses



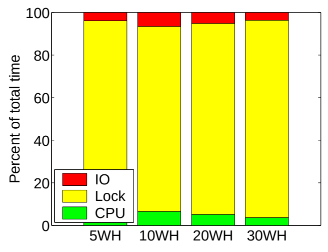
(c) **DB2**: Standard Scaling (10 clients per WH)



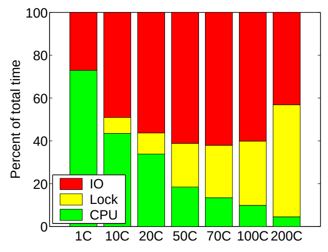
(d) **Shore**: Varying Clients, 10 Warehouses



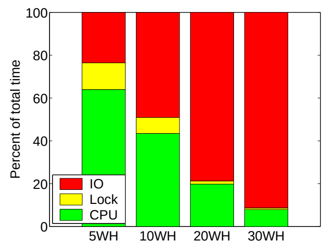
(e) **Shore**: 10 Clients, Varying Warehouses



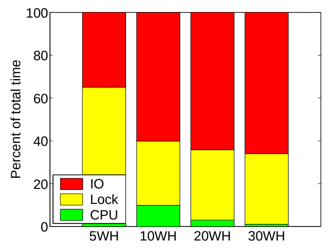
(f) **Shore**: Standard Scaling (10 clients per WH)



(g) **PostgreSQL**: Varying Clients, 10 Warehouses

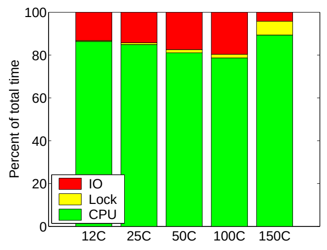


(h) **PostgreSQL**: 10 Clients, Varying Warehouses

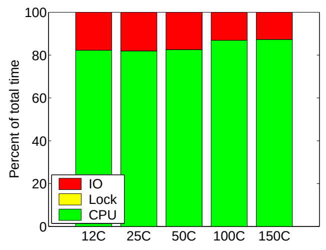


(i) **PostgreSQL**: Standard Scaling (10 clients per WH)

Figure 8.1: Resource breakdowns for TPC-C transactions under varying databases and configurations. The first row shows DB2; the second row shows Shore; and the third row shows PostgreSQL. The first column (Figures 8.1(a), 8.1(d), 8.1(g)) shows the impact of varying concurrency level by varying the number of clients. The second column (Figures 8.1(b), 8.1(e), 8.1(h)) shows the impact of varying the database size (number of warehouses) while holding the number of clients fixed. The third column (Figures 8.1(c), 8.1(f), 8.1(i)) shows the impact of varying both the number of clients and the database size according to the TPC-C specification (10 clients for each warehouse).



(a) **IBM DB2**



(b) **PostgreSQL**

Figure 8.2: Resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL.

breakdowns for standard *"realistic"* configurations. Throughout, the think times are fixed at zero.

The database sizes for TPC-C range from 500MB to 3GB, as the number of warehouses grows from 5 to 30 (100MB per WH). The bufferpool size is approximately 800MB for each DBMS, chosen to minimize transaction execution times.

The main result shown in Figure 8.1 is that locks are the bottleneck resource for both Shore and DB2 (rows 1 and 2), while I/O tends to be the bottleneck resource for PostgreSQL (row 3). We now discuss these in more detail.

We start with some obvious trends. First observe that as concurrency is increased while fixing the database size (column 1), lock contention increases. Also, as the database size grows, while the concurrency level is held constant (column 2), the I/O component grows, and the lock component decreases. When the database and concurrency level are scaled according to TPC-C specifications, the relative resource breakdowns remain fairly stable.

The resource breakdowns for Shore and DB2 (rows 1 and 2) are quite similar, and almost always depict lock bottlenecks. This may be surprising, since concurrency control was a very active area of research in the 1970's and 80's, and thus one might think that locking problems were all resolved at that time. Given our hardware limitations, we can only experiment with up to 30 WH. It is plausible that the bottleneck may shift to I/O as the database size increases. Alternatively, additional RAM and disks may hide the growing I/O for larger databases, leaving locks as the bottleneck resource.

The resource breakdowns for PostgreSQL (row 3) differ greatly from those for Shore and DB2:

PostgreSQL almost always exhibits an I/O bottleneck. As indicated earlier, PostgreSQL I/O time includes the time for both the actual I/O operation and the lightweight lock I/O synchronization. Almost all (80–95%) of the I/O time is due to I/O synchronization in the standard case (Figure 8.1(i)). While this suggests that I/O scheduling will be necessary for PostgreSQL prioritization, in Section 8.4, CPU scheduling will be used to indirectly schedule I/O.

Although not the bottleneck, locks are sometimes a non-trivial component for PostgreSQL. In particular, locks reach 50% when concurrency is increased while fixing the database size (Figure 8.1(g)), and reach 30% when standard TPC-C scaling is used (Figure 8.1(i)).

The fact that PostgreSQL's resource breakdowns differ from those for Shore and DB2 is due to differences in concurrency control in these systems: Shore and DB2 employ 2PL, while PostgreSQL uses MVCC. With MVCC, PostgreSQL transactions only have to wait for write-on-write conflicts. The result is fewer lock waits in PostgreSQL than in Shore and DB2, shifting its bottleneck to I/O.

Each breakdown presented in Figure 8.1 is an average computed over all transactions in an experimental run, and as such, may not be representative of any particular, or even most transactions. The breakdowns can be sharply skewed by a small fraction of exceptional transactions with extremely long execution times. Thus, the breakdowns are primarily an indicator of the relative importance of the resources when minimizing average transaction execution times.

TPC-W. Figure 8.2 shows resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL as a function of the number of clients connected to the database. The size of the database is held constant (150MB), and is representative of a database used by 10 clients according to the TPC-W specification. Increasing the number of clients to 150 models extremely high data contention. PostgreSQL sees almost no locking and DB2 sees very little, as TPC-W intrinsically has very little data contention. I/O costs are also low since the database is so small relative to main memory. Thus, CPU is the bottleneck resource for TPC-W [1].

[1]We find that under extreme configurations, lock waiting can be significant for TPC-W as well. Since these configurations depart so much from the TPC-W specifications, we do not consider them here.

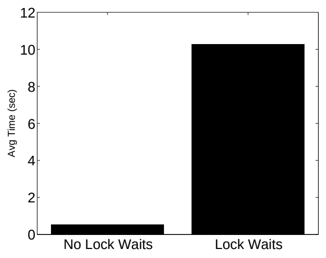


Figure 8.3: Average execution time for TPC-C Shore transactions that never wait for locks compared to those that do, with no prioritization. Think time is 1 second.

8.4 Scheduling the Bottleneck

As seen in Section 8.3, the bottleneck resource for TPC-C on Shore and DB2 is locks, suggesting that lock prioritization will be effective. Figure 8.3 motivates this point, showing that transactions that do not wait for locks are almost 20 times faster than those that do.

The bottleneck resource for PostgreSQL is usually I/O. While I/O scheduling is outside the scope of this work, it is well-known that CPU scheduling may indirectly schedule other resources [9], such as I/O or locks. This is due to the fact that transactions need CPU resources to issue resource requests. Consequently, we investigate whether CPU scheduling is effective for PostgreSQL.

Throughout, we examine *both* lock and CPU scheduling for *both* TPC-C and TPC-W. We have reservations, however, about the TPC-W workload for two reasons: its transactions are (i) extremely simplistic, and (ii) need very little concurrency control. The TPC-C workload, with more complex transaction interactions, is in fact more representative of real-world applications. Note that we do not evaluate any of the scheduling policies on IBM DB2, since it does not support such policies and the source code is unavailable for experimentation.

We begin by defining the specific scheduling policies that we will explore.

8.4.1 Prioritization Workload

Throughout this section, we use a representative 10 warehouse database for TPC-C (1GB) and a 10 client database for TPC-W (150MB). Priorities are assigned to each TPC-C and TPC-W transaction according to a Bernoulli trial with probability 10% of being a high-priority transaction. Observe that the maximally achievable differentiation between high and low priority transactions depends on the frequency of high versus low priority transactions. In the case we are considering (10% of transactions being high priority and 90% being low priority) the maximum possible improvement high priority transactions can hope for is a factor of 10 improvement in their average response time compared to the no priority case. If the percentage of high priority transactions is large, the potential for prioritization is low, e.g. in the case of 90% high priority transactions even a perfect algorithm can not improve average high priority response times by more than 10%. We therefore focus in our work on the case where the majority (90%) of transactions is of low priority.

TPC-C and TPC-W are *closed loop systems*, where a fixed number of clients alternatingly wait and execute transactions against the database. The time spent waiting is known as *think time* and models interactive clients interpreting results. The concurrency level can be adjusted by either fixing the number of clients and varying think time, or fixing the think time and varying the number of clients. We find both methods yield similar results. Throughout our experiments we will fix think time at zero and vary the number of clients. The only exception will be for TPC-C experiments, where we will instead vary the think time and fix the number of clients at 300. We choose 300, because that allows us to use think time to vary the number of running clients both above and below the TPC-C-specified 100 clients. The reason that we vary think time for TPC-C prioritization is that the TPC-C clients can consume significant system resources, and thus using a constant number of clients helps reduce variability due to this overhead.

8.4.2 Definition of the Policies

Our scheduling policies are divided into lock scheduling and CPU scheduling policies:

Lock scheduling policies. We first consider non-preemptive lock scheduling policies, where lock holders are never forced to release their locks abnormally due to preemption. Subsequently, we consider preemptive policies, in which high-priority transactions can preempt low-priority lock holders to acquire their locks. Preemption involves aborting, rolling back, and resubmitting the transaction, adding more work for the DBMS.

The simplest non-preemptive policy, `NP-LQ`, just reorders transactions waiting in the lock queue, and grants locks to high-priority transactions before those of low-priority. This policy has a problem:

high-priority transactions moved to the front of the queue must wait for low-priority transactions already holding the lock to complete (known as “excess time” in queueing theory). The case where a high-priority transaction waits for a low-priority transaction is commonly known as priority inversion. Two techniques are commonly used to address the problem, *priority inheritance* [184] and *preemption*.

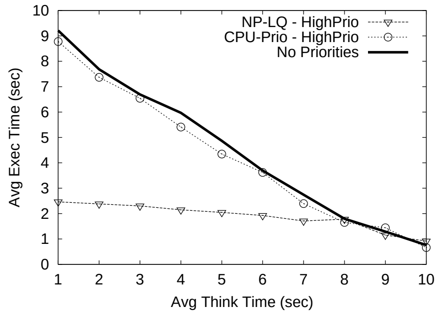
`NP-LQ-Inherit` is a non-preemptive policy that uses priority inheritance to reduce excess times. The policy is identical to `NP-LQ`, but the priority of each transaction is raised to the highest priority of any transaction that waits for it. Thus, high-priority transactions never wait for transactions with priority lower than their own. The intended result is that high-priority excess times are reduced, improving high-priority execution times.

`P-LQ` aims to reduce high-priority excess times by preempting transactions currently holding locks needed by high-priority transactions. The policy is identical to `NP-LQ`, but when a high-priority transaction needs a lock held by a low-priority transaction, the low-priority transaction is aborted (known as *preemptive abort*). In practice, two factors reduce the effectiveness of preemption. First, the preempting high-priority transaction must still wait for (part of) the low-priority transaction rollback to complete before continuing. Second, extra work created by preemption potentially slows down other transactions.

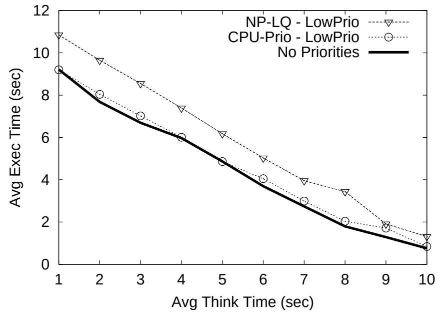
CPU scheduling policies. Each of the DBMS considered relies on approximations of (preemptive) generalized processor sharing (GPS), and as a result we do not distinguish preemptive or non-preemptive scheduling of the CPU device itself. We do, however, consider preemption of transactions due to lock conflicts while using CPU prioritization. We call CPU scheduling policies that preempt lock holders preemptive, and those that do not non-preemptive.

The simplest policy, `CPU-Prio`, is a non-preemptive policy that schedules the CPU using weighted GPS. It simply gives more weight to processes working on high-priority transactions. Specifically, for PostgreSQL, we assign UNIX priority nice level -20 to high-priority processes and $+20$ to low-priority processes. For Shore, high-priority threads get “time critical” priority, while low-priority transactions get “regular” priority.

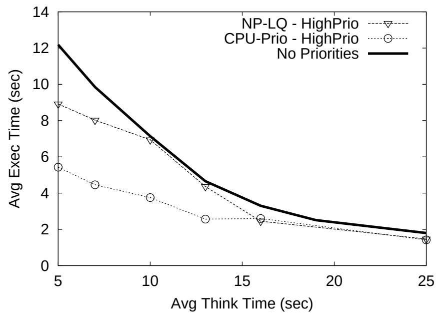
Although the `CPU-Prio` policy prioritizes CPU, the policy may suffer from priority inversions due to locks. A high-priority transaction with high CPU-priority cannot progress if it waits for a lock



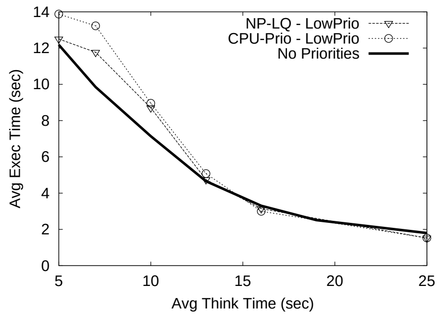
(a) **Shore** High-Priority



(b) **Shore** Low-Priority



(c) **PostgreSQL** High-Priority



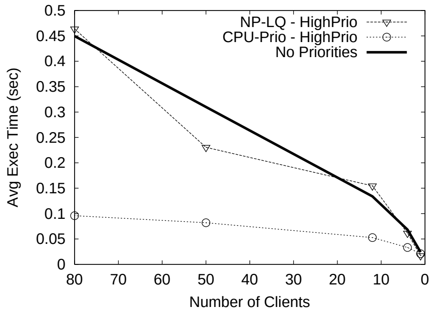
(d) **PostgreSQL** Low-Priority

Figure 8.4: Mean execution times for `NP-LQ` compared to `CPU-Prio` for Shore and PostgreSQL TPC-C with varying contention. Concurrency (load) increases to the left, as think time goes down.

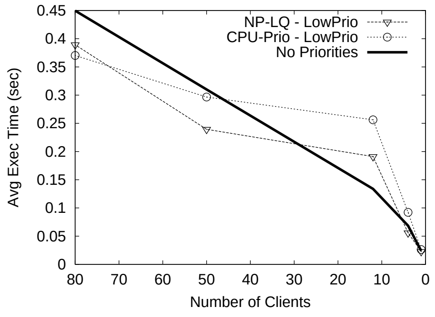
held by a low-priority transaction. `CPU-Prio-Inherit` is a non-preemptive policy that adds priority inheritance to the `CPU-Prio` policy. The priority of low-priority transactions that block high-priority transactions is raised, thus reducing high-priority excess times.

The `P-CPU` policy is a preemptive policy identical to `CPU-Prio` except that low-priority transactions that block high-priority transactions are preempted and rolled back.

Organization of remaining sections. In Section 8.4.3 we present results for simple scheduling policies without preemption or priority inheritance: `NP-LQ` and `CPU-Prio`, defined above. In Section 8.4.4, we examine policies with priority inheritance: `NP-LQ-Inherit` and `CPU-Prio-Inherit`. In Section 8.4.5, we discuss the preemptive policies `P-LQ` and `P-CPU`.



(a) **PostgreSQL** High-Priority



(b) **PostgreSQL** Low-Priority

Figure 8.5: Mean execution times for `NP-LQ` compared to `CPU-Prio` for PostgreSQL TPC-W with varying loads.

8.4.3 Simple Scheduling

The simple scheduling policies with no priority inheritance and no lock preemption, `NP-LQ` and `CPU-Prio`, exhibit striking differences depending on the workload and the DBMS. Figures 8.4 and 8.5 highlight these differences, showing the performance of high- and low-priority transactions using the policies for TPC-C and TPC-W workloads respectively. In all results, the concurrency varies on the X-axis, from high levels of concurrency on the left to low concurrency on the right. Concurrency is controlled either by varying think time (for TPC-C) or, equivalently, by varying the number of clients (for TPC-W).

The best simple scheduling policy for TPC-C depends on the DBMS. For TPC-C running on Shore (see Figure 8.4(a)), `CPU-Prio` does not appreciably improve high-priority transaction execution times. `NP-LQ`, on the other hand, improves high-priority performance by 3.7 times. The penalty to low-priority transactions under both `NP-LQ` and `CPU-Prio` is small (less than 17% for `NP-LQ`) and tracks the “Default” no-priority setting (see Figure 8.4(b)). Lock scheduling is extremely effective for Shore because locks dominate transaction execution times under 2PL.

By contrast, for PostgreSQL, lock scheduling is not as effective as CPU scheduling (see Figure 8.4(c)). Under high loads, `NP-LQ` improves high-priority execution times by a factor of 1.3, whereas `CPU-Prio` improves them by a factor of 2. With both policies, low-priority transactions are not significantly penalized (see Figure 8.4(d)). As the think time increases from 5 to 25 seconds, concurrency decreases from 200 to 20 running (non-thinking) clients on average, and the lock fraction of execution times becomes insignificant. As expected, the result is that lock scheduling (`NP-LQ`) is not very effective.

The effectiveness of `CPU-Prio` for TPC-C on PostgreSQL is surprising, given that I/O (I/O-related lightweight locks) is its bottleneck. Due to CPU prioritization, high-priority transactions are able to request I/O resources before low-priority transactions can. As a result, high-priority transactions wait fewer times (50–90% fewer) for I/O, and when they do wait, they wait behind fewer transactions (30% fewer). The fact that simple CPU prioritization is able to improve performance so significantly suggests that more complicated I/O scheduling is not always necessary.

For TPC-W, locks are never the bottleneck resource (see Figure 8.2), suggesting lock scheduling will be ineffective. As confirmation, Figure 8.5 shows average execution times with `NP-LQ` and `CPU-Prio` for TPC-W as a function of the number of clients. As expected, `NP-LQ` does not significantly improve high-priority transactions. `CPU-Prio`, however, dramatically improves high-priority transaction times by a factor of up to 4.5 under high load (high number of clients) relative to a system with no priorities.

Low-priority transactions, on average, are not significantly penalized by either `NP-LQ` or `CPU-Prio`, for all DBMS and workloads studied. This result is important, and consistent with theoretical results: Performance of a small class of high-priority transactions can be improved without harming the overall low-priority performance.

8.4.4 Priority Inheritance

In this section we evaluate the two policies using priority inheritance: `NP-LQ-Inherit` and `CPU-Prio-Inherit`, which are extensions of the `NP-LQ` and `CPU-Prio` policies, respectively.

Figure 8.6 compares the policies `NP-LQ-Inherit` and `NP-LQ` for TPC-C running on Shore for a range of concurrency levels. We find that adding priority inheritance to simple lock queue reordering (`NP-LQ`) improves performance by 30%. `NP-LQ` improves high-priority transaction execution times by a factor of 3.7 relative to a system without priorities, and `NP-LQ-Inherit` improves execution times by a factor of 5.3.

For TPC-C running on PostgreSQL, adding priority inheritance to `NP-LQ` offers no appreciable gain in performance, however, priority inheritance with CPU scheduling is beneficial. Figure 8.7 shows CPU priority inheritance improves high-priority transactions by a factor of 6, whereas `CPU-Prio` only helps by a factor of 2. The significant improvement in performance is due to the fact that the lock holder(s) are sped up, resulting in significantly smaller wait excesses.

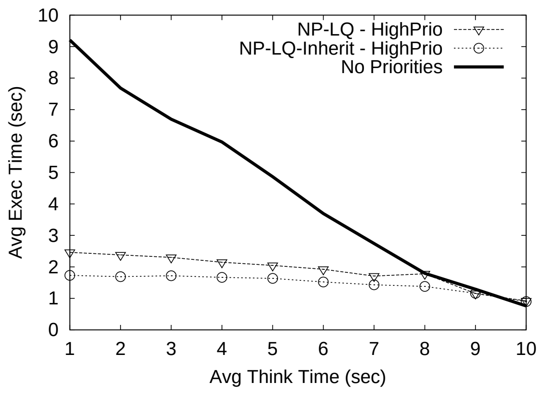


Figure 8.6: `NP-LQ-Inherit` compared to `NP-LQ` for Shore TPC-C.

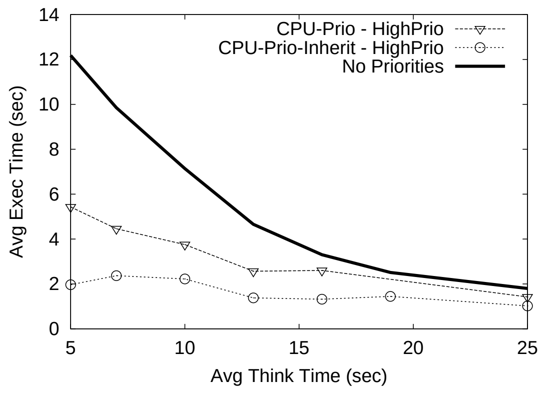


Figure 8.7: `CPU-Prio-Inherit` compared to `CPU-Prio` on PostgreSQL TPC-C.

Recall from Section 8.4.3 that CPU scheduling (`CPU-Prio`) is more effective than `NP-LQ` for TPC-W. Thus Figure 8.8 compares the policies `CPU-Prio-Inherit` to `CPU-Prio` for the TPC-W workload on PostgreSQL. We find that there is no improvement for `CPU-Prio-Inherit` over `CPU-Prio`. This is to be expected given the low data contention found in the TPC-W workload; priority inversions can only occur during data contention. Results for low-priority transactions are not shown, but as in Figure 8.4, low-priority transactions are only negligibly penalized on average.

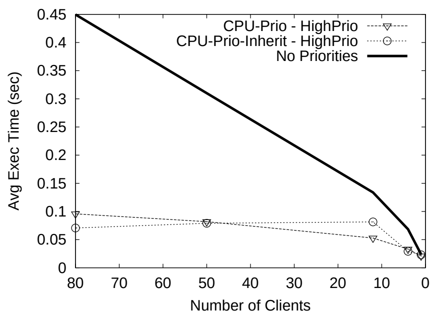


Figure 8.8: `CPU-Prio-Inherit` compared to `CPU-Prio` for TPC-W running on PostgreSQL.

8.4.5 Preemptive Scheduling

Non-preemptive scheduling already provides substantial performance improvements for high-priority TPC-C transactions, using lock scheduling for Shore and CPU scheduling for PostgreSQL. We now focus on whether preemption can provide further benefits. In particular, we evaluate whether `P-LQ` improves on `NP-LQ` for Shore and whether `P-CPU` improves on `CPU-Prio` for PostgreSQL.

With non-preemptive scheduling, high-priority transactions sometimes must wait on lock requests for locks currently held by low-priority transactions (the wait excess). The wait excess time is reduced, but not eliminated, with priority inheritance, which speeds up the low-priority transactions blocking high-priority transactions. Preemptive scheduling (`P-LQ` and `P-CPU`) attempts to eliminate the wait excess for high-priority transactions by preempting low-priority lock holders in the way of high-priority transactions.

We find that preemptive policies provide little benefit over non-preemptive policies. Figures 8.9(a) and 8.9(b) compare the average high- and low-priority execution times for `P-LQ` against `NP-LQ-Inherit` for TPC-C on Shore as a function of think time. High-priority transactions with `P-LQ` improve by a factor of 9.3 whereas `NP-LQ-Inherit` helps only by a factor of 5.3. Low-priority transactions, however, are slowed by a factor of 1.7, which is excessive, making this policy impractical. Figures 8.9(c) and 8.9(d) compare the performance of `P-CPU` to `CPU-Prio-Inherit` for TPC-C on PostgreSQL. Preemption seems to offer no significant benefit or penalty beyond `CPU-Prio-Inherit`.

TPC-W results for `P-LQ` and `P-CPU` are omitted as lock scheduling is ineffective since lock contention is low.

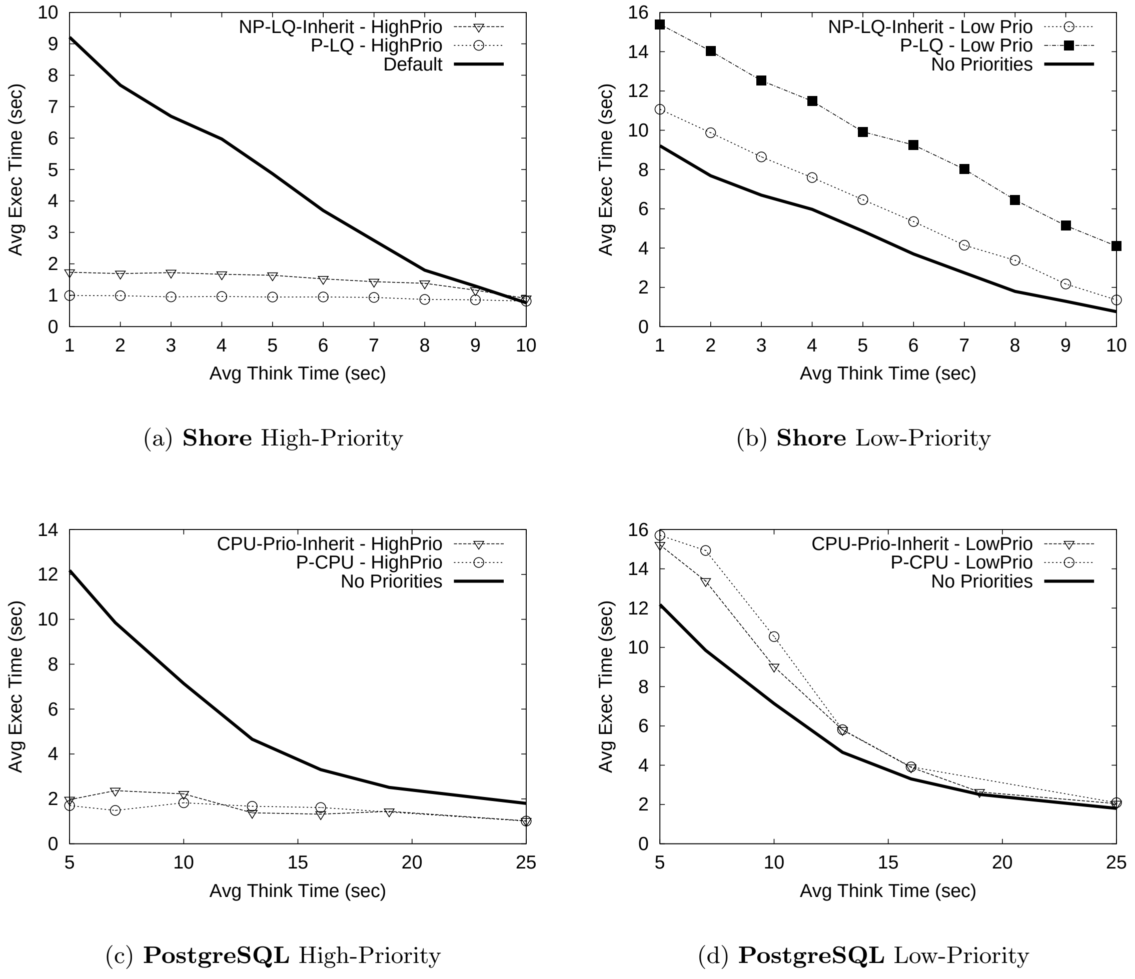


Figure 8.9: Preemptive policies `P-LQ` and `P-CPU` for Shore and PostgreSQL respectively, compared to the best non-preemptive policies for TPC-C.

8.5 Summary

In this chapter we develop and evaluate an implementation of transaction prioritization for differentiated performance classes for TPC-C or TPC-W workloads running on traditional relational DBMS.

We first identify the bottleneck resource at which priority scheduling is most effective. We divide the lifetime of a transaction into three components: CPU, I/O, and lock wait times. The results are clearly differentiated by workload and concurrency control mechanism. Across a wide range of configurations, the bottleneck for TPC-C running on DBMS using 2PL (Shore and DB2) is *lock waiting*. By contrast, the bottleneck for TPC-C running on MVCC DBMS is *I/O synchronization* for low loads, although locking can dominate at extremely high concurrency levels. For TPC-W workloads, *CPU* is always the bottleneck.

This bottleneck analysis provides a roadmap for which resources must be scheduled to improve performance. In this chapter we focus on lock and CPU scheduling to directly or indirectly schedule the bottleneck resource. We evaluate the effectiveness of simple prioritization, priority inheritance, and preemptive abort scheduling, and the results are broken down by workload and concurrency control mechanism.

For TPC-W on all DBMS, we find that lock scheduling is largely ineffective since transactions rarely wait for locks. CPU scheduling, however, is extremely effective. For TPC-W running on PostgreSQL, we find that the simplest scheduling policy, `CPU-Prio`, is best, and improves performance for high-priority transactions by a factor of up to 4.5. Priority inheritance is not necessary since data contention for TPC-W is almost non-existent.

For TPC-C on MVCC DBMS, and in particular PostgreSQL, CPU scheduling is most effective, due to its ability to indirectly schedule the I/O bottleneck. For TPC-C running on PostgreSQL, the simplest CPU scheduling policy (`CPU-Prio`) provides a factor of 2 improvement for high-priority transactions, while adding priority inheritance (`CPU-Prio-Inherit`) provides a factor of 6 improvement while hardly penalizing low-priority transactions. Preemption (`P-CPU`) provides no appreciable benefit over `CPU-Prio-Inherit`.

For TPC-C on 2PL DBMS (Shore), non-preemptive lock scheduling with priority inheritance (`NP-LQ-Inherit`) is most effective. For Shore, high-priority transaction execution times improve 5.3

times, while low-priority transactions are hardly penalized. Priority inheritance does not appreciably help. We find that the preemptive strategy (`P-LQ`) is effective in reducing the response times of high priority transactions, but this improvement comes at an excessive penalty for low-priority transactions. The fact that our simple preemptive strategy, does not mean that preemptive lock scheduling in general is not feasible. It means that more selective methods for deciding when to preempt are necessary. We will discuss the topic of more sophisticated preemptive algorithms in more detail in Chapter 11.

In conclusion, the good news is that priority scheduling using simple policies can yield significant performance improvements for both TPC-C and TPC-W workloads on real general-purpose DBMS. However, these improvements depend on knowing the bottleneck resource and applying scheduling at this resource. In general it can be difficult to know the bottleneck resource, since it depends on several factors, the DBMS, the workload and the hardware configuration. In the next chapter we will therefore investigate a different approach to providing priorities that is independent of the bottleneck resource.

Chapter 9

Providing prioritization for OLTP workloads outside the DBMS

"When is external prioritization for OLTP workloads feasible and effective?"

In the previous chapter we have investigated how to provide transaction prioritization for general purpose DBMS through internal scheduling, that is the prioritization is integrated into the database engine. The obvious advantage of DBMS-internal scheduling is that the scheduler has full control over all the DBMS resources, providing for maximum effectiveness. However, the disadvantages are that the effectiveness relies on knowing the bottleneck resource and limited portability across DBMS since it requires changing DBMS code.

The above disadvantages of internal scheduling motivate us to investigate in this chapter the feasibility and effectiveness of *external prioritization*, i.e., prioritizing the transactions externally to the DBMS. External prioritization has many obvious advantages including portability (across different DBMS) and ease of implementation (only the external queue needs to be scheduled and this is done at the application level).

The idea in external prioritization is demonstrated in Figure 9.1, and simply involves limiting the number of transactions concurrently executing within the DBMS. This limit is referred to as the MPL (multi-programming limit). If the MPL is already met, we queue up all remaining transactions in two external queues, one for high priority transactions, and one for low priority transactions, as shown in the figure. The high priority transaction queue is given priority when slots become available in the DBMS. The capacities of the external queues are unlimited, hence no transactions are ever dropped.

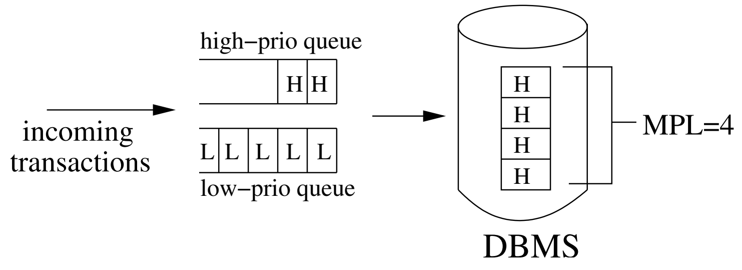


Figure 9.1: *Simplified view of the mechanism used to achieve high versus low priority. A fixed limited number of transactions (MPL=4) are allowed into the DBMS simultaneously. The remaining transactions are held in two unlimited external queues, one for each priority type. Response time is the time from when a transaction arrives until it completes, including time spent queueing externally to the DBMS.*

While the advantages of external scheduling are apparent, it is not clear what the downsides are to external scheduling. Whereas external scheduling is an accepted idea in the networking/web server community, it is not well understood how the complex internals and resource usage in DBMS interact with external scheduling. Some immediate concerns are: (1) By holding back transactions, it is not clear whether the resulting lowered concurrency leads to a drop in throughput, and how much of a drop. Most existing work is concerned in a drop in throughput due to too high a concurrency level, causing lock thrashing and system overload. By contrast, we ask what is the lowest possible MPL value that is necessary to ensure near-optimal throughput levels. (2) There is a potential fear that by holding back transactions, and sequencing them (rather than letting them all share the database resources concurrently), we may create a head-of-line (HOL) blocking situation where some long-running transactions prevent other shorter transactions from entering the DBMS and receiving service. This could potentially result in a too-high penalty for low priority transactions and also in an actual increase in overall mean response time over the non-prioritized case. (3) Lastly, it is not at all obvious that external scheduling will be as effective as internal scheduling in creating priority differentiation, since we don't have any control over the transactions once they're dispatched to the DBMS.

The point of this chapter is to answer the above three questions. We will be experimenting with a wide range of hardware configurations (different numbers of disks, CPUs, and amounts of main memory), different DBMS (Shore, IBM DB2), and many different workloads (CPU-bound, I/O-bound, lock-bound, multiple-resource, etc.). We will see that all of these play a role in determining the appropriate MPL for external prioritization, and in the effectiveness of prioritization.

In Section 9.1 we describe the related prior work on DBMS prioritization. In Section 9.2 we describe the experimental architecture, and explain in detail our experimental workloads, shown in Table 9.1, based on which we create 17 unique experimental "setups," surveyed in Table 9.2. The three main concerns discussed above are then investigated in Sections 9.3, 9.4, and 9.5. Finally we conclude with our findings.

9.1 Existing work on external scheduling for DBMS

Work on external scheduling of database transactions primarily focuses on admission control to combat overload conditions in DBMS. The admission control approach is based on the principle that DBMS performance often declines as the number of executing transactions and the demand on system resources and data contention (locking) increase. Admission control deals with these problems by limiting the number of transactions executing inside the DBMS by dropping transactions or holding transactions "outside of the database" until the load subsides.

Most of the work in admission control targets overload conditions due to data contention. As early as 1985 Katoh et al. [111] proposed a class of schedulers that predict potential lock conflicts a new requests might cause and drop requests that will necessitate any rollback in the future. Heiss et al. [95] propose two algorithms that base their admission control decisions on the system load. [53] present the half-and-half approach to prevent data thrashing. Finally, [146] introduces a data contention performance metric called the critical ratio that the authors base their admission control algorithm on to avoid lock thrashing.

Recent work at IBM[78] proposes admission control in combination with priority classes for overload control in a three-tier web architecture with a database backend. The approach makes admission control decisions based not only on the number of transactions in the DBMS, but also on transaction priorities, by limiting the number of low-priority transactions that are able to interfere with high-priority transactions. Such admission control limits inefficiencies introduced when the system is under overload, such as virtual memory paging and thrashing. Consequently, high-priority transactions under overload can benefit significantly.

Our work differs from the above work both in the goals and in the methods. The work described above tries to find an *upper* limit on the MPL in order to avoid excessive contention, while the goal of our work is to determine a *lower* limit on the MPL that allows close to maximum utilization and at the same time effective prioritization. Moreover, in our work we never drop requests, but only hold back low priority requests.

It is important that while external scheduling for providing transaction priorities has not been investigated in research (except in combination with admission control under overload), most commercial database systems offer a mechanism for creating and managing external queues. One such example is the DB2 query patroller [4]. Our goal is that the algorithms presented in this chapter, along with feasibility analysis, will be used in conjunction with such a mechanism.

9.2 Experimental setup

In answering the questions of this chapter with respect to the feasibility and effectiveness of external prioritization, it is important to understand the effect of system resources and workloads on these questions.

Towards this end, we will be experimenting with a range of hardware configurations (different numbers of disks, CPUs), different DBMS (Shore, IBM DB2), and many different workloads (CPU-bound, I/O-bound, lock-bound, multiple-resource, etc.). We will see that all of these play a role in determining the appropriate MPL for external prioritization, and in the effectiveness of prioritization.

9.2.1 Experimental architectures

The DBMS we experiment with are IBM DB2 [1] version 8.1, and Shore [157]. Shore is a prototype storage manager with state-of-the-art transaction management, 2PL, and Aries-style recovery that we use it because we have the source code, enabling us to implement internal priorities. All of our external scheduling results are also corroborated using PostgreSQL [164] version 7.3, although we do not show these results here for lack of space.

In all experiments the DBMS is running on a 2.4-GHz Pentium 4 running Linux 2.4.23. The buffer pool size and main memory size will depend on the workload (see Table 9.1). The machine is equipped with six 120GB IDE drives, one of which we use for the database log. The number of remaining IDE drives that we use for the data will depend on the particular experiment. The client generator is run on a separate machine with the same specifications as the database server, and is directly connected to the database server through a network switch.

9.2.2 Experimental workloads and setups

As mentioned before, it is very important when discussing throughput and prioritization to consider a wide range of workloads. Unfortunately there are only a limited number of standard OLTP benchmarks which are both well-accepted and publicly available, in particular TPC-C [69] and TPC-W [198]. Fortunately, however, these two benchmarks can be used to create a much wider range of workloads by varying a large number of (i) hardware and (ii) benchmark configuration parameters. The hardware parameters that we vary include: (a) the number of disks (1 – 6) , (b) the number of CPUs (1 or 2), and (c) the main memory (ranging between 512 MB and 3 GB). The benchmark configuration parameters that we vary include: (a) the number of warehouses in TPC-C, (b) the size of the database in TPC-W (this includes both the number of items included in the database store and the number of “emulated browsers” (EBs) which affects the number of customers), and (c) the type of transaction mix used in TPC-W, particularly whether these are primarily “browsing” transactions or primarily “ordering” transactions. We also vary the isolation level to create different levels of lock contention, starting with the default isolation level of 3 (corresponding to RR in DB2 – Repeatable Read), but also experimenting with lower isolation levels (UR – Uncommitted Read), leading to less lock contention. In all workloads, we hold constant the number of clients at 100.

As shown in Table 9.1, Workload $\mathbf{W_{CPU-inventory}}$ is designed to be a CPU-bound workload (CPU utilization between 90% - 100%), based on the TPC-C inventory-based benchmark with 10 warehouses (WH). This workload is ensured to be memory-resident (3GB RAM with 1GB database size, with 1GB bufferpool). To vary this workload we run it for different number of CPUs and different isolation levels (Setups 1, 2, and 17 in Table 9.2).

Workload $\mathbf{W_{CPU-browsing}}$ is a CPU-bound workload (CPU utilization between 95% - 100%), based on the TPC-W browsing-mix with parameters specified in Table 9.1. The workload exhibits low lock contention, since it contains few updates. It is ensured to be memory-resident (3GB RAM with 350MB database size, with 1GB bufferpool). We run this workload with varying numbers of CPUs (Setups 3 and 4 in Table 9.2).

Workload $\mathbf{W_{I/O-browsing}}$ is designed to be a I/O-bound workload, based on the TPC-W browsing-mix. The main memory has been reduced to 512 MB, with a database size of 2 GB and bufferpool of 100MB, so as to ensure high levels of I/O activity. CPU utilization is low since transactions are I/O-bound. Lock contention is low since the workload has few updates. We run the workload with varying numbers of disks (Setup 9 and Setup 10 in Table 9.2).

Workload $\mathbf{W_{I/O-inventory}}$ is designed to be a I/O-bound workload, based on a TPC-C database with 60 WH. This is different from the previous I/O bound workload $W_{I/O-browsing}$ because $W_{I/O-inventory}$ is more update-intensive. The main memory is 512 MB, with a database size of 6 GB and bufferpool of 100MB, so as to ensure high levels of I/O activity. This workload has low CPU utilization (around 10%), since transactions are I/O bound. Unlike the TPC-C configuration in $W_{CPU-inventory}$, this workload has only medium lock contention since the size of the database is larger, but the number of clients is kept at 100. To vary the workload further, we run it with varying numbers of disks (Setup 5-8).

Workload $\mathbf{W_{CPU+I/O-inventory}}$ is designed to be both CPU and I/O intensive, with balanced (equal) demands to both resources, and is based on a TPC-C database with 10 WH. The main memory is 1GB, with a database size of 1 GB and bufferpool of 600MB, so as to ensure high levels of I/O activity, but not so high that it dwarfs CPU demand. Lock contention in this workload is high. We run this workload both with varying numbers of disks and varying numbers of CPUs (Setups 11 and 12).

Finally, Workload $\mathbf{W_{CPU-ordering}}$ is designed to be a CPU-intensive workload (CPU utilization between 95% - 100%) that still exhibits high lock contention, unlike $W_{CPU-browsing}$ that has low

Workload	Benchmark	Configuration	Data-base	Main memory	Bufferpool
$W_{CPU-inventory}$	TPC-C	10 warehouses,	1GB	3GB	1GB
$W_{CPU-browsing}$	TPC-W Browsing	100 EBs, 10K items, 140K customers	300MB	3GB	500 MB
$W_{I/O-browsing}$	TPC-W Browsing	500 EBs, 10K items, 288K customers	2GB	512MB	100 MB
$W_{I/O-inventory}$	TPC-C	60 warehouses,	6GB	512MB	100MB
$W_{CPU+I/O-inventory}$	TPC-C	10 warehouses,	1GB	1GB	1GB
$W_{CPU-ordering}$	TPC-W Ordering	100 EBs, 10K items, 140K customers	300MB	3GB	500MB

Table 9.1: *Description of the workloads used in the experiments. The top table shows the benchmarks and configurations used for each workload.*

lock contention. The $W_{CPU-ordering}$ workload is based on the TPC-W ordering benchmark, which is high in the number of updates. This workload is ensured to be memory-resident (3GB RAM with 350MB database size, with 1GB bufferpool), and CPU intensity is high. We use the default isolation level (RR), and lock contention is high. We run this workload for a varying numbers of CPUs and different isolation levels (Setup 13, 15 and 16).

9.2.3 The algorithm

Our work does not deal with how the transactions obtain their priority class. As stated earlier, we assume that the e-commerce vendor has reasons for choosing some transactions/clients to be higher or lower-priority. Experimentally, we handle this by simply at random assigning 10% of the transaction "high"-priority and the remainder "low"-priority. Clearly priority differentiation can be made stronger by using a smaller percentage of high-priority transactions.

The algorithm that we use for prioritization is relatively simple (see Figure 9.1). For any given MPL, we allow as many transactions into the system as allowed by the MPL, where the high-priority transactions are given first priority, and low-priority transactions are only chosen if there are no more high-priority transactions. The MPL is held fixed during the entire experiment.

	Workload	Number CPUs	Number disks	Isolation level
Experimental Setup 1	$W_{CPU-inventory}$	1	1	RR
Experimental Setup 2	$W_{CPU-inventory}$	2	1	RR
Experimental Setup 3	$W_{CPU-browsing}$	1	1	RR
Experimental Setup 4	$W_{CPU-browsing}$	2	1	RR
Experimental Setup 5	$W_{IO-inventory}$	1	1	RR
Experimental Setup 6	$W_{IO-inventory}$	1	2	RR
Experimental Setup 7	$W_{IO-inventory}$	1	3	RR
Experimental Setup 8	$W_{IO-inventory}$	1	4	RR
Experimental Setup 9	$W_{IO-browsing}$	1	1	RR
Experimental Setup 10	$W_{IO-browsing}$	1	4	RR
Experimental Setup 11	$W_{CPU+IO-inventory}$	1	1	RR
Experimental Setup 12	$W_{CPU+IO-inventory}$	2	4	RR
Experimental Setup 13	$W_{CPU-ordering}$	1	1	RR
Experimental Setup 14	$W_{CPU-ordering}$	1	1	UR
Experimental Setup 15	$W_{CPU-ordering}$	2	1	RR
Experimental Setup 16	$W_{CPU-ordering}$	2	1	UR
Experimental Setup 17	$W_{CPU-inventory}$	1	1	UR

Table 9.2: Translation of workloads needed for Figure 9.13.

Obviously if we set our MPL to 1, the above algorithm produces excellent differentiation between high and low priority transactions. However, this could result in a lower throughput. We are interested in understanding the tradeoff between throughput and prioritization differentiation.

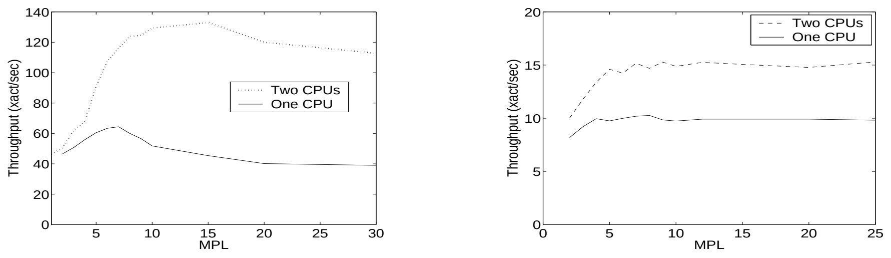
The focus of this chapter is to thoroughly investigate the questions of how throughput is affected by the MPL and how prioritization differentiation can be achieved externally versus internally to the DBMS. By showing that external prioritization is effective, we make it feasible for a vendor to implement an array of more complex algorithms for determining which transactions are allowed into the DBMS, based on various QoS goals, since the vendor has full control of the external queue. In the next chapter, we focus on the question of achieving different QoS goals and explore that question thoroughly.

9.3 Loss in throughput

9.3.1 For CPU bound workloads

Figure 9.2 shows the effect of the MPL on the throughput under two CPU-bound workloads: $W_{CPU-inventory}$ and $W_{CPU-browsing}$. The two lines shown consider the case of 1 CPU versus 2 CPUs. In the single CPU case, under both workloads, the throughput reaches its maximum at the MPL of about 5. In the case of 2 CPUs, the maximum throughput is reached at around $MPL = 10$ in the case of workload $W_{CPU-inventory}$ and at around $MPL = 7$ in the case of workload $W_{CPU-browsing}$. Observe that a higher MPL is needed to reach maximum throughput in the case of 2 CPUs as compared with 1 CPU because more transactions are needed to saturate 2 CPUs. The fact that the $W_{CPU-inventory}$ requires a slightly higher MPL is likely due to the fact that the $W_{CPU-inventory}$ workload has some I/O components due to updates. The additional I/O component means that more transactions are needed to fully utilize the CPU, since some transactions are blocked on I/O to the database log.

All these maximum throughput points are achieved at surprisingly low MPL values, considering the fact that both these workloads are intended to run with 100 clients according to TPC specifications.



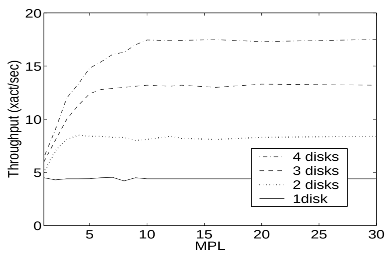
(a) $W_{CPU-inventory}$ (b) $W_{CPU-browsing}$

Figure 9.2: *Effect of MPL on throughput in CPU bound workloads:(a)* $W_{CPU-inventory}$ *(Setups 1 and 2 of Table 9.2) and (b)* $W_{CPU-browsing}$ *(setups 3 and 4 of Table 9.2).*

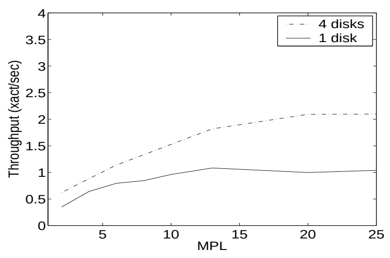
9.3.2 For I/O bound workloads

Figure 9.3 shows the effect of the MPL on the throughput under two I/O-bound workloads: $W_{I/O-inventory}$ and $W_{I/O-browsing}$. The lines shown consider different numbers of disks. The $W_{I/O-inventory}$ workload is a pure I/O-only workload, because of the larger database size. For this workload, the MPL point at which maximum throughput is reached is $MPL = 2$ for the case of 1 disk, $MPL = 5$ for the case of 2 disks, $MPL = 7$ for the case of 3 disks, and $MPL = 10$ for the case of 4 disks. Again, observe that the MPL needed to maximize throughput grows for systems with more disks, since more transactions are required to saturate more resources. Again, these numbers are extremely low considering the fact that the TPC specifications for this workload assumes 600 clients (recall we use 100 clients experimentally).

For $W_{I/O-browsing}$, the MPL at which maximum throughput is reached is higher than for $W_{I/O-inventory}$ (about $MPL = 13$ for one disk and about $MPL = 20$ for four disks). The reason is that the size of this database is smaller than for the $W_{I/O-inventory}$ workload, thus resulting in a larger CPU component than in the purely I/O-based $W_{I/O-inventory}$. As explained in Section 9.3.3 the additional CPU component will add to the MPL needed. Still, it is surprising that an MPL of 20 suffices given that the TPC specifications for this workload assumes 500 clients (recall we use 100 clients experimentally).



(a) $W_{I/O-inventory}$



(b) $W_{I/O-browsing}$

Figure 9.3: *Effect of MPL on throughput in I/O bound workloads: (a) $W_{I/O-inventory}$ (setups 5–8 of Table 9.2) and (b) $W_{I/O(TPC-browsing}$ (setups 9 and 10 of Table 9.2).*

9.3.3 For "balanced" $CPU + IO$ workloads

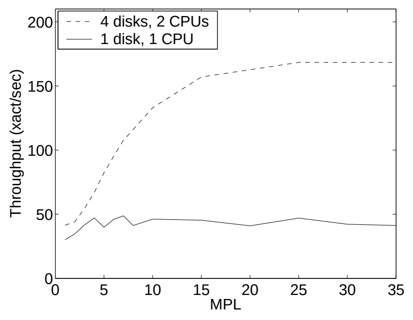
Figure 9.4 considers workload $W_{CPU+I/O-inventory}$ which is balanced (equal) in its requirements of CPU and I/O (both resources are equally utilized). In the case of just 1 disk and 1 CPU, an MPL of 5 suffices to reach maximum throughput. Adding only disks to the hardware configuration changes this value only slightly. The reason is that in a balanced workload, when one adds disks, the CPU bottleneck is still there, making the workload CPU bound, so that adding disks has no further effect. Similarly, adding only CPU to the hardware configuration also changes the required MPL value only slightly, since now the workload becomes solely I/O bound. However if we add 4 disks and 2 CPUs (maintaining the initial balanced proportions of CPU and I/O), we find that the MPL needed to reach maximum throughput increases to around 20. This number is still low in light of the fact that the TPC specified number of clients for this workload is 100.

As shown above, the "balanced" workload represents, in a sense, a "worst-case" workload in terms of throughput loss. This is an important point and we explain it now. Consider a system with multiple hardware resources. If all the transactions require only *one* of these resource types (e.g. only disks are required), then the MPL needed to saturate the system will be in some way proportional with the number of this resource type (i.e. number of disks). However, if the transactions require a mixture of multiple resource types (e.g., CPU and disk), then increasing the number of disks while holding the number of CPUs constant will not increase the throughput capacity of the system. The reason is that the system will become CPU limited, and thus the required MPL to reach maximum throughput will be bounded by the number of CPUs. Hence the only impact will come from increasing *both* resources, (e.g. both the numbers of CPUs and disks), which will cause the throughput capacity to surge since a large number of transactions will be needed to saturate both resources. As the total number of resources goes up, the MPL needed to saturate all these resources increases.

In summary, the MPL required is in some way proportional to the number of resources that are utilized in a system without a limit to the number of transactions. In a balanced workload the number of resources that are utilized will be high, hence the MPL is higher.

9.3.4 For Lock-bound workloads

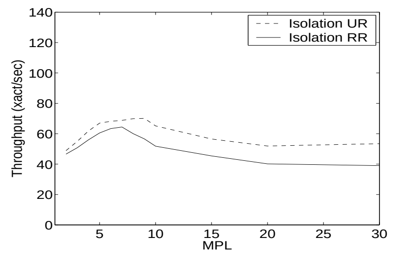
Figure 9.5 illustrates the effect of increasing the locking needed by transactions (increasing the isolation level from UR to RR) on for workloads $W_{CPU-inventory}$ and $W_{CPU-ordering}$. While the MPL needed overall is always under 20, the basic trend is that increasing the amount of locking lowers the MPL. The reason is that when the amount of locking is high, throwing more transactions into the system doesn't increase the rate at which transactions complete, since they are all queueing. Fur-



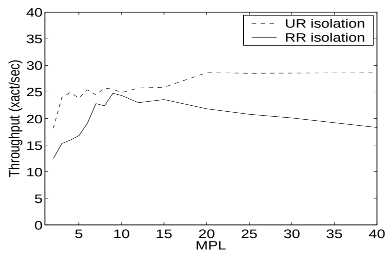
$W_{CPU+I/O-inventory}$

Figure 9.4: *Effect of MPL on throughput in workload exhibiting both high I/O and CPU:* $W_{CPU+I/O-inventory}$ *(setups 11 and 12 of Table 9.2).*

thermore beyond some point, increasing the number of transactions actually lowers the throughput. This is well known and is reported by others, e.g., [111, 95, 146, 53].



(a) $W_{CPU-inventory}$



(b) $W_{CPU-ordering}$

Figure 9.5: *Effect of MPL on throughput in workloads with heavy locking: (a)* $W_{CPU-inventory}$ *(setups 1 and 17 of Table 9.2) and (b)* $W_{CPU-ordering}$ *(setups 15 and 16 of Table 9.2).*

9.3.5 Conclusions and scaling projections

We have seen that in all the above experiments, the MPL value needed to achieve maximum throughput was low, given the overall (TPC specified) number of clients for those workloads. In all the above experiments an MPL of less than 20 sufficed, and often lower, while the number of clients was a hundred or several hundred.

The MPL increased, as expected, with the number of utilized resources in the system. Interesting, however, we found that this increase was a somewhat *linear* function. We also found that there was little difference in the MPL behavior for a CPU-bound versus an I/O-bound workload. Workloads with balanced demands on both CPU and I/O required the highest MPL.

It is important to note that the above experiments are simply prototype experiments to give some sense of the MPL value needed for achieving maximum throughput. In particular, they don't address two issues relevant in practice. First, we assigned priorities at random independent of transaction types. In practice there might likely be a correlation between the type of a transaction and its priority, e.g. a "Buy" transaction might have higher priority than a "Catalog search" transaction. We don't expect such a correlation to affect the choice of a good MPL, though, since the external prioritization doesn't create burstier arrivals of certain transaction types or other adverse effects. Preliminary experiments we carried out to investigate this issue seem to support our assumption. Second, in our experiments we are limited to the hardware available in to us in our testbed. Obviously it would be better to experiment with larger scale systems as well. Since this is infeasible for us, we instead provide theoretical analysis to see whether our observed trends continue as we increase the number of resources.

We start by creating a very simplistic model of the database internal resources. This is shown in Figure 9.6. We model the MPL number of clients allowed in our externally scheduled system by fixing the number of clients in the "closed" system model represented in Figure 9.6. We analyze this "closed" system for different MPL values and see when the maximum throughput is achieved. This in turn yields the desired MPL value needed for our externally scheduled system. The MPL yielded by this analysis is in fact an upper bound on the actual MPL that we would get in experiments for two reasons: First, we purposely create the "worst-case" in our analytical model by assuming that all resources are equally utilized. Second we do not allow for the fact that a client may be able to utilize two resources (e.g., two disks) at once. In our model we assume that the service times of all devices are exponentially distributed with mean 1.

We use our model to produce results similar to our $W_{I/O-inventory}$ workload. For this workload there is almost no CPU, however the number of disks plays an important role. In our experiments, we were able to experiment with up to 4 disks, as shown in Figure 9.3. However in analysis we can go much further. Figure 9.7 shows the results of the analysis with up to 16 disks.

The first observation is that the results of the analysis for 1 to 4 disks look very similar to the actual experimental results from Figure 9.3. Next, we observe that the MPL required to reach near

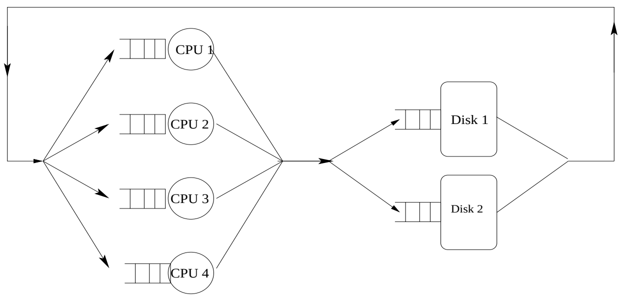


Figure 9.6: *Theoretical model representing the DMBS internals. This model provides us with a theoretical upper bound on the MPL needed to provide maximum throughput.*

maximum throughput grows *linearly* with the number of disks: The minimum MPL that is sufficient to achieve 80% of the maximum throughput is marked with circles, and the minimum MPL that is sufficient to achieve 95% of the maximum throughput is marked with squares. Both the circles and the squares form *straight lines*.

Although it may seem a little scary to observe that the necessary MPL grows linearly with more disks, it is important to notice that systems with many disks also have a proportionately larger population of clients, hence an MPL that seems large may still be small in proportion to the client population.

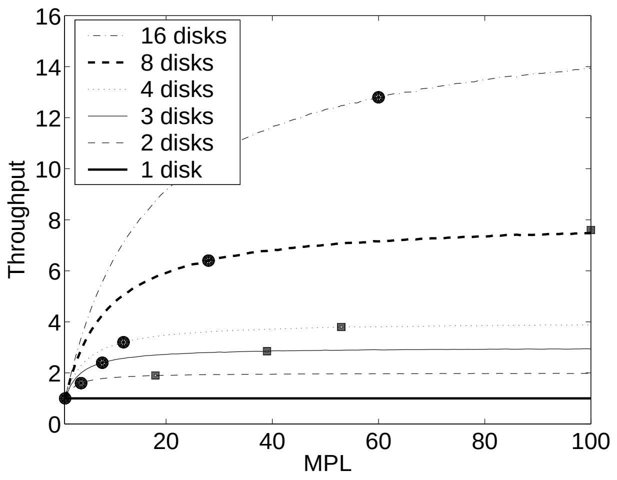


Figure 9.7: *Results of theoretical analysis showing the effect of MPL on throughput as a function of the number of resources. The squares denote the minimum MPL that limits throughput loss to 5%. The circles denote the minimum MPL that limits the throughput loss to 20%. Note that the set of circles form a perfectly straight line, as do the squares.*

Another take-away point is that simple queueing analysis, as we have done, can offer a useful tool for system administrators in roughly predicting the MPL value at which a system should be run as a function of the throughput desired. While we find that the current analysis is a very good predictor of our experimental results for the 4-disk system, it is certainly possible to refine the analytic queueing model further to make it more realistic, or build a simple simulator utilizing existing tools to simulate the real resources and thereby get really accurate predictions of the MPL.

9.4 Increase in response time

The previous section showed that external scheduling with low MPL is feasible in that it doesn't cause throughput to go down (provided the MPL is not too low). Because we are working in a closed system, an immediate consequence of this fact is that the overall mean response time also does not suffer (see Little's Law [127]). However, this point is far less obvious for an *open system*, where response time is not inversely related to throughput. In this section we will investigate the effect of the MPL value on mean response time in great detail, starting with experimental work and then moving to queueing theoretic analytical work.

Experimentally, we modify our experimental setup to an open system with Poisson arrivals. For the open system, for all workloads, we find that the response time is insensitive to the MPL value provided it is at least 4 (for system utilization of 70%) and at least 20 (for a system utilization of 90%).

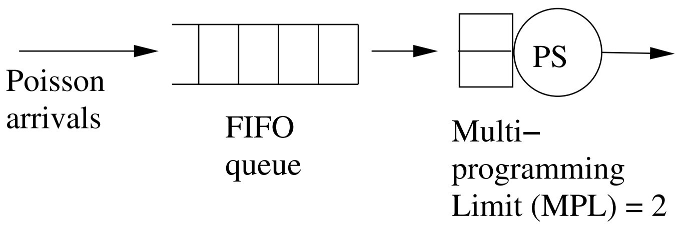


Figure 9.8: *Queueing network model of external scheduling mechanism with MPL = 2.*

Experimental results are limited however in that they do not allow us to vary an important workload parameter affecting response times: this is the variability in transaction demands. To study how the workload variability affects the feasible range of MPL values, we resort to analysis.

The analytical results below show response time both as a function of the MPL value and as a function of C^2, where C^2 is the squared coefficient of variation of the transaction demands.

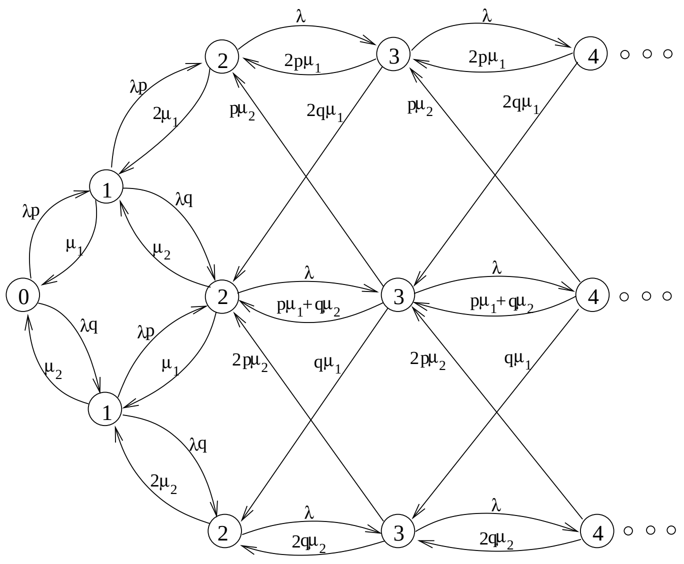


Figure 9.9: *Continuous-time Markov chain corresponding to the flexible multiserver queue representation of the queueing network in Figure 9.8. The two jobs in service may both have service rate μ_1 (top row), or may have rates μ_1 and μ_2 (middle row), or may both have service rates μ_2 (bottom row).*

From a theoretical perspective our external scheduling mechanism with MPL parameter can be viewed as a single unbounded First-in-first-out (FIFO) queue feeding into a Processor-Sharing (PS) server where only MPL jobs may share the PS server. This queueing network for MPL $= 2$ is shown in Figure 9.8. Note that this is not a poor approximation of our system in that, as we will see in the next chapter the DBMS in many ways behaves like a PS system.

To the best of our knowledge, there is no existing simple solution to our queueing network in Figure 9.8. Therefore, we derived the following solution approach: We start by assuming that the job sizes (service requirements) follow a 2-phase hyperexponential (H_2) distribution, with probability parameter p and rates μ_1 and μ_2, allowing us to arbitrarily vary the C^2 parameter. This allows us to equivalently represent the network in Figure 9.8 by a special "flexible multiserver queue" where the number of servers fluctuates between 1 and MPL as needed, and where the *sum* of the service rates at the multiple servers is always maintained constant and equal to that at the single PS server. The continuous-time Markov chain corresponding to the flexible multiserver queue is shown in Figure 9.9

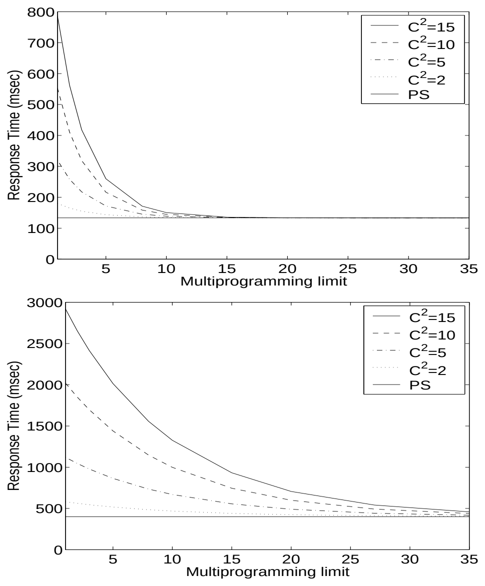


Figure 9.10: *Evaluation of CTMC for different C^2. The system load is 0.7 (top) and 0.9 (bottom).*

for the case of an H_2 service time distribution (with parameters p, μ_1, and μ_2), arrival rate λ and MPL = 2. Note that we define the shorthand $q = 1 - p$. This Markov chain lends itself to Matrix-analytic analysis [124, 155], because of its repeating structure.

Figure 9.10 shows the results of evaluating the Markov chain in Figure 9.9. We find that for low C^2 values of 1 or 2, the mean response time is largely independent of the MPL value and equal to that for the pure PS system (with infinite MPL), assuming the MPL is at least 5. For higher C^2 values of 5–15, we find that the MPL depends on the load and needs to be at least 10 (for load of 0.7) or 30 (for load of 0.9) to ensure low mean response time (similar to PS). In the TPC-C benchmark the C^2 value is 1.5. To get an estimate of the ranges of C^2 in real OLTP workloads, we obtained traces from one of the top-10 online retailers and from one of the top-10 auctioning sites in the US. We find that both are in relatively close agreement with the TPC-C benchmark, exhibiting values for C^2 of around 2. The TPC-W benchmark is an outlier in that its transactions exhibit C^2 values of 15.

9.5 Comparison of external and internal prioritization effectiveness

We have seen in the previous sections that a relatively low MPL value suffices to limit the loss of throughput and also to prevent an increase in overall mean response time. Next we turn to the question of the degree of differentiation between high and low priority response times that external scheduling can provide, and we compare this with what internal scheduling can offer.

9.5.1 Effectiveness of external prioritization

We start by studying the effectiveness of external prioritization.

We first consider the case where the MPL is chosen to limit throughput loss to 5% (compared to the case where no external scheduling is used), see Figure 9.13(top), and then the case where the MPL is chosen to limit throughput loss to 20%, see Figure 9.13(bottom). For each of these two cases, we experiment with all 15 setups shown in Table 9.2. In each experiment we apply the external scheduling algorithm described in Section 9.2.3 and measure the mean response times for high and low priority transaction, in addition to the overall mean response time when no priorities are used.

We find that using external prioritization, in the case of 5% throughput loss (Figure 9.13(top)), high priority transactions perform 4.2 to 21.6 times better than low priority transactions with respect to mean response time. The average improvement of high priority transactions over low priority transactions is a factor of 12.1. The low priority transactions suffer only a little as compared to the case of no prioritization, by a factor ranging from 1.15 to 1.17, with an average suffering of 16 %. The above numbers are visible from the figure (or caption). Not visible from the figure is whether prioritization causes the overall mean response time to rise. It turns out that the overall mean response time is never hurt by more than 6% compared to the orginal system without external scheduling.

We find that using external prioritization, in the case of 20% throughput loss (Figure 9.13(bottom)), high priority transactions perform 7 to 24 times better than low priority transactions with respect to mean response time. The average improvement of high priority transactions over low priority transactions is a factor of 18. The low priority transactions suffer by a factor ranging from 1.35 to 1.39, as compared to the case of no prioritization, with an average suffering of 37%. The above numbers are visible from the figure (or caption). Not visible from the figure is whether prioritization causes the overall mean response time to rise. It turns out that the overall mean response time is never hurt by more than 25% compared to the orginal system without external scheduling. Observe that in the case of 20% throughput loss, the differentiation between high and low priority requests is more pronounced, since the MPL values are lower, but this comes at the cost of lower throughput and higher overall response times.

9.5.2 Implementation of internal scheduling

As seen in Chapter 8 Scheduling the internals of the DBMS is obviously more involved than external scheduling. It is not even clear *which resource* should be prioritized: the CPU, the disk, the lock queues, etc. Once one resolves that first question, there is the follow-up question of *which algorithm* should we use to give priority to high-priority transactions, without extensively penalizing low priority transactions. Both questions are not obvious.

Recall that in Chapter 8 we found that in OLTP workloads run on 2PL (2-phase locking) DBMS, transaction execution times are often dominated by lock waiting times, and hence prioritization of transactions is most effective when applied at the lock queue. We find that other workloads or DBMS lead to transaction execution times being dominated by CPU usage or I/O, and hence prioritization of transactions is most effective when applied at those other resources.

Having seen that it is not obvious which internal resource needs to be scheduled, we now turn to the particular 17 setups shown in Table 9.2. Some of these (e.g., setup 3 and 4) are CPU bound, while others (e.g., 1 and 2) are lock-bound, and still others are I/O bound (e.g. setup 5-10). In our experiments with internal scheduling we consider two particular setups: Setup 1 (Lock-bound) and Setup 3 (CPU-bound).

For setup 1, we implement the *Preempt-on-Wait* (POW) lock prioritization policy in Shore[157]. In POW, high priority transactions move ahead of low-priority transactions in the lock queue, and are allowed to even preempt a low-priority lock holder if that low-priority lock holder is waiting at another lock queue. We give more information on POW in Section 11.1 and the associated publication [139].

For setup 3, CPU prioritization is available in IBM DBM through the DB2gov tool [99]. However, we find that we achieve better priority differentiation by "manually" setting the CPU scheduling priorities used by the Linux operating system. We use the `renice` command in Linux to set the CPU priority of a DB2 process executing a high priority transaction to -20 (the highest available CPU priority) and the CPU priority of a DB2 process executing a low priority transaction to 20 (the lowest available CPU priority).

In the next section we show the results for internal scheduling for these setups.

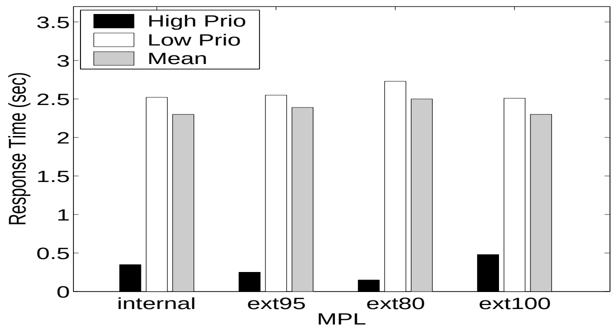


Figure 9.11: Results for setup 1.

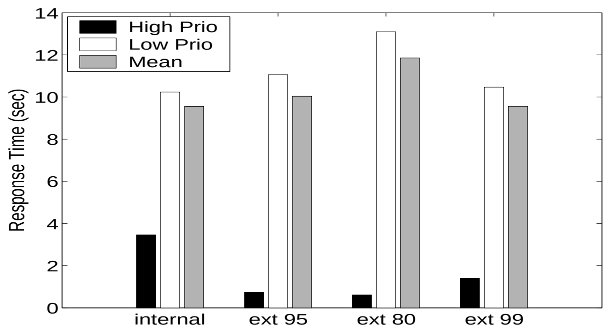


Figure 9.12: Results for setup 3.

9.5.3 Internal prioritization results and comparison with external results

In this section we consider setup 1 and 3 from Table 9.2. For each setup, we compare the performance obtained via internal prioritization with that obtained via external prioritization. We consider 3 versions of external prioritization, the first involving 5% throughput loss, the second involving 20% throughput loss, and the third involving 0% throughput loss. Figure 9.11) shows the results for setup

1, and Figure 9.12 shows the results for setup 3.

For both setups, we find that with respect to differentiating between high and low priority transactions, external scheduling is nearly as effective as internal scheduling (for the case of zero throughput loss), and can even be more effective if one is willing to sacrifice 5% to 20% in throughput. Looking at the suffering of the low priority transactions as compared to the overall mean response time, we find that external scheduling results in no more suffering for the low priority transactions than internal scheduling, if the MPL is chosen such that no throughput is lost. In the case of low MPL with 5-20% sacrifice in throughput, low priority transactions suffer up to 10% more than under internal scheduling.

Because of the inherent complexity of implementing internal scheduling, we were only able to provide numbers for setups 1 and 3 out of the 17 setups in Figure 9.2. We find that for these two setups, external scheduling is a viable approach when compared with internal scheduling. We hypothesize that external scheduling will compare favorably on the remaining setups as well, given the strong results shown for external scheduling in Figure 9.13.

While our results are promising it is important to note that we are not claiming that external scheduling is always as effective as internal scheduling. Although the internal scheduling algorithms that we considered are quite advanced, there may be other internal scheduling algorithms which are superior to our external approach for certain workloads. The point that we make in this work is that external scheduling is an effective approach, when the MPL is adjusted appropriately.

9.6 Summary

This chapter lays the experimental and theoretical groundwork for an exploration of the effectiveness of implementing prioritization externally to the DBMS.

At the heart of our exploration is the questions of *how* exactly should one limit the concurrent number of transactions allowed into the DBMS, i.e., the MPL (multi-programming limit). The obvious tradeoff is that one both wants the MPL to be low enough to create good prioritization differentiation and at the same time high enough so as not to limit throughput or create other undesirable effects like increasing overall mean response time.

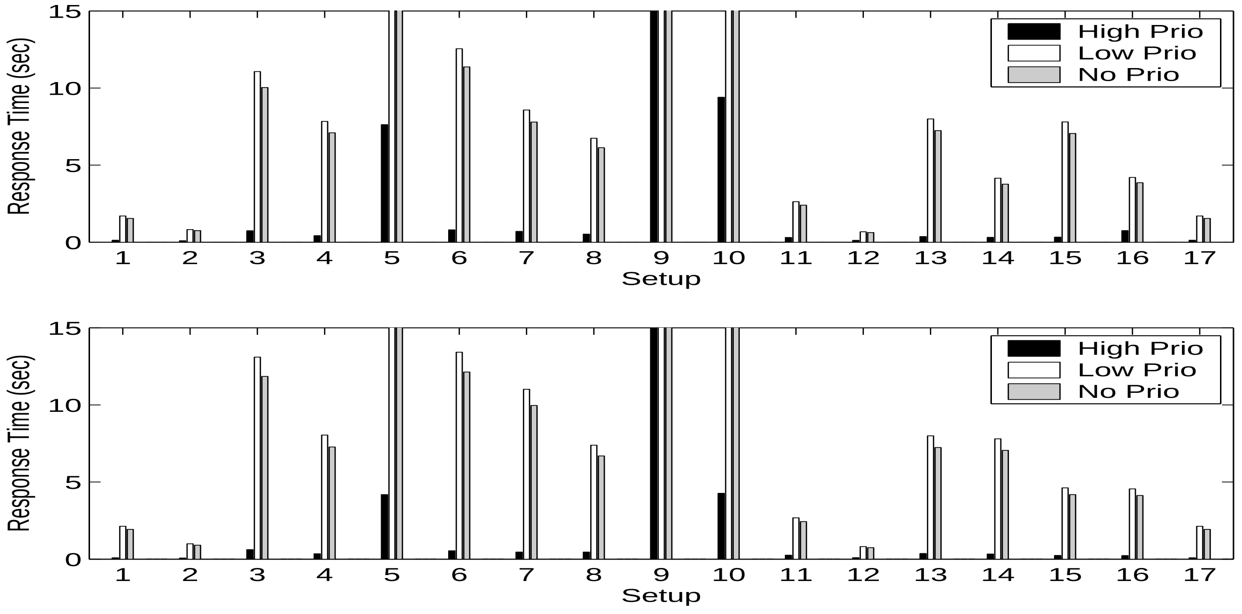


Figure 9.13: *Results of external scheduling algorithm. This figure shows the mean response times for high and low priority requests, as well as the case of no prioritization, for all 17 setups described in Table 9.2. In the* top *graph, the MPLs have been set to sacrifice a maximum of 5% throughput for each experiment. In the* bottom *graph, The MPLs are set to sacrifice a maximum of 20% throughput. Observe that workloads 5, 9, and 10 have been cut off. The values for these workloads in (top) are (7.6 sec, 76.864 sec), (26.2 sec, 111 sec), and (9.4 sec, 50.9 sec), respectively, and in (bottom) are (4.1 sec, 79.3 sec) (15 sec 112 sec) (4.2 sec and 51.9 sec) respectively.*

Our experiments include a vast array of 17 experimental setups (see Table 9.2) spanning a variety of hardware configurations (1–6 disks, 1–2 cpus, main memory from 512 MB - 3 GB), many different workloads (lock-bound, CPU bound, I/O bound, multi-resource), and different DBMS (Shore, IBM DB2).

Experimentally, we find that the optimal choice of the MPL varies across the experimental setup studied. However in all experimental setups there is always some feasible MPL that manages to satisfy both the conflicting goals of not hurting throughput or response time while providing sufficient prioritization differentiation between classes. On average across our 17 experimental setups, we achieve a factor of 12 differentiation in mean response time between high and low priority transactions, while only sacrificing 5% of the total throughput and only increasing overall mean response time by 6%, in the case where 10% of our transactions are high priority. While prioritization is so helpful to high priority transactions (factor of 12 improvement), it only slightly penalizes low-priority transactions, by only 16%. These numbers are flexible in that one can always trade off a greater sacrifice in throughput and low priority performance for a greater gain in prioritization differentiation, and we present such numbers. Interestingly, the values of the MPLs that meet all the above goals are quite low, compared with the typical number of users associated with the above experimental setup workloads. For example in all our experimental setups, we find that the ideal MPL is at or below 20.

Despite these encouraging experimental results, it is still difficult to generalize these findings to larger systems. Our results are limited to only 1–6 disks, 2 cpus, and benchmark-type workloads, rather than real workloads. Therefore, we next explore these same issues using analysis, where we can quickly ascertain the effect of more resources and further analyze the effect of workload characteristics including a huge range of variability in the transaction service demands.

Our first analytical contribution is that even with simplistic analytical queueing models, we are able to validate the experimental results above, obtaining similar numbers for overall throughput and response time as a function of MPL. The second contribution is that the analysis enables us to find surprising and interesting trends with respect to setting the MPL in larger and more variable systems. For example, we show that with respect to throughput loss, the minimum MPL required to limit the throughput loss to some fixed value (e.g. 5% loss) grows linearly in the worst case with the number of utilized resources in the system. This is a positive result in that the population size utilizing the system is also likely to grow proportionately to the number of resources in the system, and hence the optimal MPL will always be a small fraction of the overall number of clients, even for larger systems than we are able to study experimentally. We also find analytically that as the variability of the workload increases, one needs a higher MPL to avoid increasing overall mean response time. However this increase in the MPL seems to be a slow-growing function, allowing a low MPL even when the workload variability is high, e.g., $C^2 = 15$.

In conclusion, we have developed a framework for effectively scheduling transactions outside the DBMS, without touching DBMS internals. We apply the external scheduling approach to the problem of transaction prioritization and compare its effectiveness to internal approaches. We find that differentiation through external scheduling can achieve results close to internal scheduling without a significant loss in throughput or other negative side effects. We will show in the next chapter how the external scheduling framework can be used to achieve more complex QoS goals, such as meeting response time targets, targets on the percentile of the response time, or reducing variability in response times.

Chapter 10

Achieving complex QoS goals for OLTP workloads

"Can we do more than just providing priorities?"

In the previous chapter we developed an external front end scheduler for providing simple priority differentation among classes of transactions. In many cases simple prioritization is not enough. For example, certain clients may demand, as part of their Service Level Agreements (SLA), specific QoS guarantees such as a guaranteed response time, or a guaranteed level of variability. Also, retailers may want to aim for different response time targets for different transaction types, given that recent studies [44, 39] show that the patience level of customers varies depending on the type of transaction (e.g., customers are more tolerant of long search times and less tolerant of long purchase times). Furthermore, previous research [43] indicates that users may judge a relatively fast service still unacceptable unless it is also predictable. This motivates the need for more general QoS goals such as percentile goals, where $x\%$ of response times for a class are guaranteed to be below some value y, or goals which limit variability in response time.

The goal in this chapter is to extend the external scheduler from the previous chapter to a more general scheduling framework with support for complex, class-based QoS targets. The following are the high-level design goals for the system we want to develop:

Diverse per-class QoS target metrics Ideally our system should allow for an arbitrary number of different classes, where the classes can differ in their arrival rates, transaction types, etc. Each class is associated with one or more QoS targets for (per-class) mean response time, percentiles of response time, variability in response time, best effort, or any combination thereof.

Portability and ease of implementation Our system should be portable across DBMS, and easy to implement.

Self-tuning and self-adaptive Our system should ideally have very few parameters, all of which are determined by the system, as a function of the QoS targets, without intervention of the database administrator. The system should also automatically self-adapt to changes in the workloads and QoS targets.

Effective across workloads Database workloads are diverse with respect to their resource utilization characteristics (CPU, I/O, etc.). We aim for a solution which is effective in a large range of workloads.

No sacrifice in throughput & overall mean time Achieving per-class targets should not come at the cost of an increase in the overall (over all classes) mean response time or a drop in overall throughput.

With respect to the above design goals, the prior work is limited. Although there is a tremendous amount of work in the area of real-time DBMS (RTDBMS), this work is concerned with deadlines rather than any target involving mean response time. Commercial DBMS provide a vague sense of priorities; however these are not associated with response time targets. There has been work involving per-class response time guarantees [186, 48, 47, 107]; however these approaches are limited in portability in that they require modifying database internals (e.g. the bufferpool manager) to achieve their goals. Existing mechanisms also often focus on only one resource (e.g. the bufferpool) limiting the effectiveness across different workloads.

Our approach aims at achieving the above design goals through an external front end scheduler. As explained in the previous chapter the scheduler maintains an upper limit on the number of

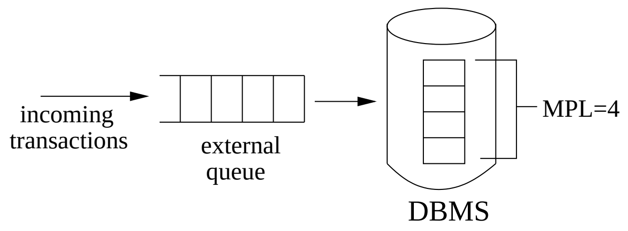


Figure 10.1: *Simplified view of mechanism used to achieve QoS goals. A fixed limited number of transactions (MPL=4) are allowed into the DBMS simultaneously. The remaining transactions are held in an unlimited external queue. Response time is the time from when a transaction arrives until it completes, including queueing time.*

transactions executing simultaneously within the DBMS (called the Multi-Programming Limit, or "MPL"), as shown in Figure 10.1. If a transaction arrives and finds MPL number of transactions already in the DBMS, the arriving transaction is held back in an external queue (no transactions are dropped). Response time for a transaction includes both waiting time in the external queue (queueing time) and time spent within the DBMS (execution time).

The immediately apparent attribute of our approach is that it lends itself to portability and ease of implementation as there is no dependence on DBMS internals. With respect to obtaining diverse QoS goals, the core idea is that by maintaining a very low MPL, we are able to predict the time that a transaction spends within the database (execution time). This in turn gives us an upper bound on the queueing time for a transaction, which can be used by the scheduler in order to ensure that QoS targets are met. The actual algorithms that we use are obviously more complex and rely on queueing analysis in order to meet a more diverse set of QoS goals, and behave in a self-adaptive manner.

Part of the difficulty inherent in achieving QoS goals is that obviously not every arbitrary goal is feasible. We distinguish between two types of infeasible goals. The first one comprises goals that cannot be achieved due to a simple lack of system resources (e.g. suppose every class desires a really low response time guarantee). In our work we start by identifying the class of feasible QoS targets. There are two obvious sources of limitations for what QoS targets are feasible: (i) the per-class mean response time cannot be lower than the mean response time of the transactions in that class when run in isolation; (ii) We don't expect the overall mean response time under class-based prioritization to be lower than for the unprioritized system. That is the weighted average over all per-class mean response time targets is not expected to be lower than the mean response time in the original unprioritized system.

The second type of goals that cannot be achieved has to do with the limitations of our external scheduling approach. We have seen in the previous chapter that the MPL cannot below some threshold, or else the result will be a decrease in overall throughput or increase in overall response time. However, some QoS goals might require setting the MPL lower than that threshold in order to have sufficient control over the transactions.

After determining that the QoS goals are feasible and choosing the appropriate MPL, we move on to our additional design goals of self-tuning and self-adaptivity. Towards this end, we develop algorithms for dynamically tuning the MPL based on the instantaneous response time measurements and the QoS targets.

We demonstrate the effectiveness of our solution in experiments with two different DBMS, IBM DB2 and PostgreSQL. We create a range of workloads, including CPU-bound, I/O-bound, and high vs. low lock contention workloads, based on different configurations of TPC-C[69] and TPC-W[198]. We show that our solutions apply equally well under the full range of workloads studied, although the MPL parameter must be varied depending on the workload. The reason is that the core idea of limiting the MPL reduces contention within the DBMS at the bottleneck resource, independent of what the particular bottleneck resource is.

The chapter is organized as follows: Section 10.1 reviews related work. In Section 10.3 we describe our external scheduling approach in detail and then demonstrate the effectiveness of our approach in achieving a large range of different types of QoS goals. We conclude in Section 10.5.

10.1 Existing approaches

Below we describe prior work on providing response time guarantees for DBMS transactions. Most of the work is in the area of real-time DBMS (RTDBMS), which is concerned with deadlines rather than any target involving mean response time. Commercial DBMS provide a vague sense of priorities; however these are not associated with response time targets. The little work that involves per-class guarantees is primarily simulation-only, is limited to mean response time targets only, and is not portable in that it requires the modification of database internals (e.g. the bufferpool manager).

Work on RTDBMS

In Real-time DBMS, there is a deadline (typically a hard deadline) associated with each transaction. The goals of RTDBMS is to minimize the number of transactions which miss their deadlines. If a hard deadline is missed, the transaction is dropped. Examples of work in this area include: [5, 7, 8, 9, 98].

This work is different from our own in that it does not allow for mean response time targets. Also, in our work, no transactions are dropped. The RTDBMS typically involves using a specialized DBMS, and the mechanism is implemented internally, making it less portable. Kang et al. [107] generalize the above deadline metric for main memory DBMS to allow for per-class guarantees on the rate of missed deadlines, although mean response time targets are not considered.

Commercial DBMS

As a testament to the importance of the problem of providing different service levels most commercial DBMS provide priority mechanisms in some form. For example, both IBM DB2 [122] and Oracle [172] offer CPU scheduling tools for prioritizing transactions. Although different classes are given different priorities with respect to system resources, it is not clear how these priority levels relate to achieving specific response time targets. Towards this end, Kraiss et al. [115] try to map each class to some fixed priority such that scheduling based on priorities will meet the desired response time targets. Such an assignment of priorities to classes does not always exist.

Towards per-class mean response time targets

Carey et al., [52], consider the situation of two classes, where they strive to make the mean response time for the high priority class as low as possible by scheduling internal DBMS resources on a read-only workload. Their work is a simulation study. In our recent work [138] we consider the same problem for a more general workload (TPC-C and TPC-W) under a variety of DBMS via an implementation of (DBMS internal) lock scheduling and CPU scheduling.

More closely related to our current work are the following papers, [47, 48, 186], all of which

Workload	Benchmark	Configuration	Data-base
$W_{IO>CPU}$	TPC-C	10 warehouses, 100 clients	1GB
W_{IO}	TPC-C	40 warehouses, 100 clients	4GB
W_{CPU}	TPC-W Browsing	100 EBs, 10K items, 140K customers	300MB
$W_{CPU>IO}$	TPC-W Shopping	100 EBs, 10K items, 140K customers	300MB

	IO load low	IO load high
CPU load low	N/A	W_{IO}
CPU load high	W_{CPU}	$W_{IO>CPU}$ $W_{CPU>IO}$

Table 10.1: *Description of the workloads used in the experiments. The top table shows the benchmarks and configurations used for each workload. The bottom table shows the resource utilization for each workload. (100 EBs denotes 100 emulated browsers).*

have multiple classes each with a different mean response time goal. Other QoS goals are not considered. Their approach is to schedule internal memory (buffer pool management). The above are all simulation studies.

Some work that is not directly related to our own, but is still relevant is the work of Gillmann et al., [87]. Gillmann et al. do not consider the case of *multiple classes* with per-class response time goals; however, they do look at achieving an overall mean response time target for workflow management systems by configuring the set of replicated servers. Additional tangentially related work deals with using external scheduling and admission control to combat overload conditions in DBMS [111, 95, 146, 53, 78].

10.2 Experimental setup

As representative workloads for transactional web applications, we choose the TPC-C [69] and TPC-W [198] benchmarks. The TPC-C workload in this study is generated using software developed at IBM. The TPC-W workload is generated using the TPC-W Kit from PHARM [49], though minor modifications are made to improve performance, including rewriting the connection pooling algorithm to reduce overhead.

Different configurations of these workloads (number of warehouses, number of clients) result in different levels of resource utilization for the hardware resources: CPU and I/O. We experiment with 4 different configurations of TPC-C and TPC-W as shown in Table 10.1(top). We chose these configurations in order to cover different combinations of resource utilization levels (see Table 10.1(bottom)). In addition, varying the configuration will also result in different levels of lock contention [138]. For example, lock contention is a large component of a transaction's lifetime in workloads W_{I0} and $W_{I0>CPU}$, but not in the other workloads.

The TPC-C and TPC-W benchmarks are defined to be used as closed systems, and we use them this way. We assume a zero "think time" throughout. All results have been repeated with non-zero think times, and with open system configurations and results have been found to be similar. Unless otherwise stated, we show only the results for zero think times, allowing us to focus on the effect of varying the MPL in all the graphs.

The DBMS we experiment with are IBM DB2 [122] version 8.1, and PostgreSQL [164] version 7.3. *Due to lack of space, all results graphs throughout the chapter pertain to the IBM DB2 DBMS. Results for PostgreSQL are very similar and we describe these in words only.* In all experiments the DBMS is running on a 2.4-GHz Pentium 4 with 3GB RAM, running Linux 2.4.23, with a buffer pool size of 2GB. The machine is equipped with two 120GB IDE drives, one of which we use for the database log and the other one for the data. The client generator is run on a separate machine with the same specifications as the database server, and is directly connected to the database server through a network switch.

10.3 Achieving Response Time targets

T^Q	Mean time transactions spend waiting in external queue in system with external scheduling
T^{DBMS}	Mean time transactions spend executing in the DBMS in system with external scheduling
T_i^Q	Mean time transactions in classe i spend waiting in external queue
T_i^{DBMS}	Mean time transactions in class i spend executing in the DBMS in system with external scheduling
T	Overall mean response time, i.e. sum of T_Q and T_{DBMS}
R	Mean response time in original (no external scheduling) system
R_i	Mean response time of class i transactions in original (no external scheduling) system
τ_i	Mean response time target of class i
p_i	Fraction of transactions that are class i
t_{curr}	current time
$T_i^{x\%}$	x-th percentile of T_i
$T_i^{DBMS\,x\%}$	x-th percentile of T_i^{DBMS}
$\tau_i^{x\%}$	Target for x-th percentile of T_i

Table 10.2: Notation

We assume that the system administrator assigns each transaction to one of an arbitrary number of classes, n, depending on its QoS goal. In this section we assume that each of the QoS goals is a specific mean response time target for each class. Specifically, class i transactions have a target mean response time of τ_i.

Throughout this section we will show results for two representative workloads: W_{IO} and W_{CPU}, using a fixed MPL of 20, unless otherwise stated. We choose an MPL of 20 because for our workloads this MPL is high enough that neither throughput nor overall mean response time is sacrificed.

10.3.1 Notation

The notation we use in order to formally explain the external scheduling algorithms is summarized in Table 10.2, and is straightforward. The mean response time of transactions is denoted by T, and can be divided into T^Q and T^{DBMS}, where the former denotes the mean time the transactions spend queueing externally to the DBMS and the latter quantity is the mean time that the transactions spend within the DBMS. That is,

$$T = T^Q + T^{DBMS}$$

We assume that there is a mechanism for measuring these response times. Furthermore, we denote the per-class response times via a subscript i denoting class i, where T_i denotes the mean response time for class i transactions, and

$$T_i = T_i^Q + T_i^{DBMS}$$

We assume that per-class response times can also be measured. Notice that the above notation is different from the τ_i's which denotes the i^{th} class' mean response time target. Lastly, we define R to be the mean response time in the original system, without external scheduling. The remaining notation will be explained as needed.

10.3.2 The basic algorithm

The external scheduling approach is based on setting the MPL so that the time spent within the DBMS is low and predictable. At a high level, the external scheduling algorithm works as follows: Given a set of response time targets $\{\tau_1, \ldots, \tau_n\}$, we start by limiting the MPL such that for each class the expected time within the DBMS is lower than the class' response time target, that is for each class i we ensure that

$$T_i^{DBMS} < \tau_i$$

The remaining question is how to order the transactions within the external queue. Observe that since for each class i we know its mean database execution time T_i^{DBMS} and we know its overall mean target response time τ_i, we can determine how much "slack" we have in scheduling transactions from this class: transactions in class i can afford on average to wait up to but not more than

$$s_i = \tau_i - T_i^{DBMS}$$

time units in the external queue before they should start executing in the DBMS.

Based on the slack of a transaction we compute a timestamp for when the transaction should be dispatched out of the external queue and into the DBMS, which we call the *dispatch target time.*

Formally, if a new transaction of class i arrives at time t_a its dispatch target t_d is

$$t_d = t_a + s_i = t_a + \tau_i - T_i^{DBMS}.$$

Whenever a transaction completes at the DBMS, and we have to pick the next transaction for execution from the external queue, we pick the transactions in increasing order of their dispatch targets (t_d value).

The viability of the above algorithm will be demonstrated experimentally. The above high-level description omits several issues which come up when the algorithm is implemented in practice. First, a database adminstrator needs a method for determining whether the set $\{\tau_1, \ldots, \tau_n\}$ is feasible. Second, we would like our algorithm to adjust to fluctuations in system load without having to drop transactions. Lastly, there are practical questions about how to choose and adjust the MPL. Below we address these practical concerns.

10.3.3 Feasibility of assignment

We describe a simple condition for determining whether a set of per-class mean response time targets is feasible, i.e., whether the set of targets is achievable via some algorithm.

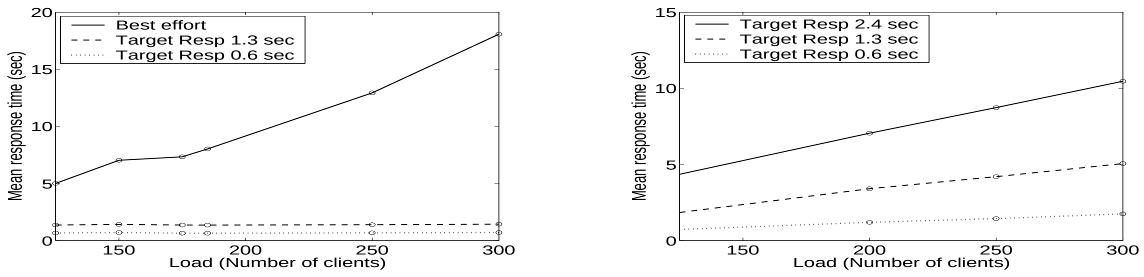


Figure 10.2: *The graphs show the response times for three classes and workload W_{IO} with increasing load (i.e. increasing number of clients).*

We start by defining the overall target mean response time (aggregated over all classes):

$$\tau_{overall} = \sum_{i=1}^{n} p_i \cdot \tau_i$$

Recall R represents the mean response time in the original system (without scheduling).

Obviously a necessary condition for achieving the individual τ_i's (via some ordering of the external queue) is that

$$\tau_{overall} > R$$

We now argue (only intuitively) that this also represents a sufficient condition. The crux of the argument is that the external scheduling (with the limited MPL) does not increase the overall measured mean response time T as compared with the original R. That is, when the MPL is chosen carefully (e.g. as described in the previous chapter),

$$T \approx R$$

Hence the above condition also implies

$$\tau_{overall} > T$$

which is sufficent.

10.3.4 Fluctuations in system load

Thus far, the response time targets have been chosen assuming a fixed system load (arrival process). During the day, however, the system load may fluctuate. Assuming that the system is never in overload, there should be no need to drop transactions, but the load might rise too high for the current set of targets to be feasible, i.e.

$$T > \tau_{overall}$$

We present two approaches for handling this case.

The first approach assumes it is more important for some classes to stay within their target than for others. The system adminstrator indicates this by specifying a priority for each class, in addition to specifying per-class response time targets. These priorities only become effective when load conditions make the per-class targets no longer feasible (i.e. $T > \tau_{overall}$). We detect this situation by checking whether there are any "late" transactions in the external queue, i.e., transactions that have already missed their dispatch target. If t_{curr} is the current time and t_d is the transaction's dispatch target time, then late transactions as those for whom

$$t_{curr} > t_d$$

Whenever we have to choose a new transaction for execution in the DBMS from the external queue, we first check whether there are transactions that are late. If there are, we pick the transaction with the highest priority among the late transactions. If there are no late transactions in the queue, we schedule as usual in the order of the dispatch targets.

An alternative approach is to carry the burden of the excess load equally among all the classes. The burden will be proportionately shared between the classes in the following way: For each transaction we compute its *lateness*, ℓ, as

$$\ell = t_{curr} - t_d$$

When scheduling late transactions, we consider the *relative lateness*, ℓ_{rel}, of the transaction normalized by its target response time τ. Relative lateness is defined as

$$\ell_{rel} = \ell/\tau = (t_{curr} - t_d)/\tau$$

Whenever we choose a transaction for execution in the DBMS from the external queue, we first check whether there are transactions that are late. If there are late transactions in the queue, we pick the transaction with the largest relative lateness, ℓ_{rel}, for execution. If there are no late transactions in the queue, we schedule as usual solely based on the dispatch targets.

10.3.5 Tuning and adapting the MPL

There are two important constraints in choosing the MPL for external scheduling. The first constraint, which we've discussed already, is that the MPL be high enough so that there is little or no loss in throughput or increase in mean response time. The second constraint is that the MPL must be low enough to ensure that for each class i

$$T_i^{DBMS} < \tau_i$$

holds, i.e. the mean execution time in the DBMS is less than its response time target. We describe below how to achieve the second constraint.

The main observation is that the expected execution time T is linear in the MPL value. This follows from [127] and agrees with experiments (Figure 10.3). (In the case of W_{CPU}, the line is not as straight as for W_{IO}, since the workload is created using TPC-W which exhibits a very high variability in service times, leading to higher variability in experimental results.) Based on this observation, we

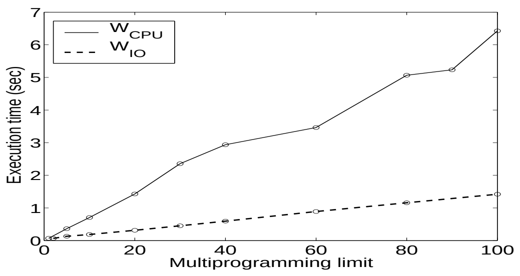


Figure 10.3: *The mean execution time as a function of the MPL under W_{IO} and W_{CPU}.*

can tune the MPL in a simple control loop:

1. Periodically monitor the per-class execution times T_i^{DBMS}.

2. Adjust the MPL if for any class i the mean execution time increases above the per-class target, i.e.

$$T_i^{DBMS} > \tau_i$$

3. Assuming that for some class the execution time is a factor of $f > 1$ times the per-class target goal, i.e.

$$T_i^{DBMS} = f \cdot \tau_i$$

we adjust the MPL as follows:

$$MPL_{new} := MPL_{old} \cdot 1/f$$

The above algorithm involves multiplicative adjustments. In practice, combining this with small constant adjustments, when close to the target, works well. We find that for obtaining a good estimate of the mean execution time in step 1) it suffices to sample a few hundred transactions in the case of W_{IO} and a few thousand transactions in the case of W_{CPU} (due to the inherently higher variability of workload W_{CPU}). In our experiments the above tuning algorithm finds the optimal MPL with at most 10 iterations.

While our approach presents a simple ad-hoc controller for tuning the MPL, it's an interesting question for future work to develop a controller that provides theoretical guarantees regarding stability and speed of convergence. One possible approach would be to follow the guidelines for designing controllable computer systems presented in the recent work by Karamanolis et al. [110].

10.3.6 Results for mean response time targets

Tables 10.3 and 10.4 show results for W_{I0} and W_{CPU} respectively under IBM DB2. Results are shown in the "Measured" column corresponding to the QoS target specified in the previous two columns. Mean and max values are specified for a sequence of 10 experimental runs, each consisting of 25,000 transactions. At the moment, we are only concerned with the first two rows of these tables, which consider per-class response time targets. We have experimented with three different classes with different targets and frequencies. As shown in the tables, we are always able to achieve within 3% of the desired per-class response time targets for W_{IO} (Table 10.3) and within 10% of the desired per-class response time targets for W_{CPU} (Table 10.4). Recall that W_{CPU} corresponds to the TPC-W workload which is more variable. Results are slightly worse for PostgreSQL, but still are within 15% of the targets.

We next present results corresponding to the second row of Table 10.3, where we now allow

Experiment type	Class	Frequency	Priority	QoS goal	Target	Measured		
						Mean	Max	Avg. error
Response times	C1	10%	1	Resp Time	0.7 sec	0.725	0.728	3.5%
	C2	20%	2	Resp Time	1 sec	1.02	1.029	2.0%
	C3	70%	N/A	Best effort	N/A	1.60	1.69	N/A
Response times	C1	40%	1	Resp Time	0.6 sec	0.653	0.657	8.0%
	C2	40%	2	Resp Time	1.3 sec	1.366	1.37	5.0%
	C3	20%	N/A	Best effort	N/A	2.54	2.59	N/A
Percentiles	C1	10%	1	80th %tile	1 sec	0.98	1.026	0%
	C2	10%	2	95th %tile	2 sec	2.03	2.159	1.0%
	C3	80%	N/A	Best effort	N/A	80th %tile: 1.96 95th %tile: 2.51	2.01 2.89	N/A N/A
Percentiles	C1	20%	1	80th %tile	1 sec	0.981	1.06	0%
	C2	20%	2	95th %tile	2 sec	2.002	2.018	.1%
	C3	60%	N/A	Best effort	N/A	80th %tile: 2.08 95th %tile: 2.69	2.18 2.77	N/A N/A
Variability	C1	100%	1	Reduce var	N/A	$C^2 = 0.108$ (before $C^2 = 2.3$)		
Combined	C1	10%	1	Resp Time	0.7 sec	0.73	0.78	4.0%
	C2	10%	2	Resp Time	1 sec	0.98	1.09	2.0%
	C3	10%	3	80th %tile	1 sec	0.978	0.99	2.0%
	C4	10%	4	95th %tile	2 sec	2.04	2.1	2.0%
	C5	60%	N/A	Best effort	N/A		1.7	

Table 10.3: *Summary of results for different QoS targets for* W_{IO}.

Experiment type	Class	Frequency	Priority	QoS goal	Target	Measured		
						Mean	Max	Avrg. error
Response times	C1	10%	1	Resp Time	3 sec	3.37	3.570	12%
	C2	20%	2	Resp Time	6 sec	6.564	6642	9%
	C3	70%	N/A	Best effort	N/A	10.6	10.8	N/A
Response times	C1	25%	1	Resp Time	2.5 sec	2.79	3.15	11%
	C2	25%	2	Resp Time	6.5 sec	6.6	7.4	1.5%
	C3	50%	N/A	Best effort	N/A	12.9	14.08	N/A
Percentiles	C1	10%	1	80th %tile	3 sec	3.068		2.7%
	C2	10%	2	95th %tile	12 sec	5.9	6.2	0%
	C3	80%	N/A	Best effort	NA	80th %tile: 16.1	16.5	N/A
						95th %tile: 19.6	21.4	N/A
Percentiles	C1	20%	1	80th %tile	3 sec	2.7	2.89	0%
	C2	20%	2	95th %tile	9 sec	7.9	8.4	0%
	C3	60%	N/A	Best effort	N/A	80th %tile: 12.01	12.08	N/A
						90th %tile: 19.3	20.0	N/A
Variability	C1	100%	1	Reduce var	N/A	$C^2 = 0.19$ (before $C^2 = 15$)		
Combined	C1	10%	1	Resp Time	3 sec	3.271	3.520	9%
	C2	10%	2	Resp Time	6 sec	6.285	6.678	6%
	C3	10%	3	80th %tile	2.5 sec	2.701	2.945	8%
	C4	60%	N/A	Best effort	N/A	80th %tile 11.8	12.4	N/A
						95th %tile 19.2	20.4	N/A
						Mean 10.5	10.9	N/A

Table 10.4: *Summary of results for different QoS targets for W_{CPU}.*

the load to fluctuate from 100 clients to 300 clients. We implement the two approaches described above for dealing with fluctuating load. In the first approach, the targets for the first two classes are maintained despite the load increase, while only the third (best effort) class suffers. These results for IBM DB2 are shown in Figure 10.2 (left). Observe that the first two classes have nearly constant response times across all loads. In the second approach, we share the burden across all the classes. The results are shown in Figure 10.2 (right), for the case where the third class has target response time 2.4 seconds. In this figure, the response time of all classes increases by the same factor as load increases.

10.4 More complex QoS goals

In this section we consider more complex QoS goals. These include: Reducing overall variance in response times (aggregated over all class transactions)(Section 10.4.1); and achieving targets on the x^{th} percentile of response time for multiple classes (Section 10.4.2).

10.4.1 Reducing variance

In addition to desiring low mean response time, users are equally desirous of low variability in response times [43]. Both W_{CPU} and W_{IO} benchmarks are composed of a fixed set of transaction types. We find that although the variance within each transaction type is not too high, the overall variance in response time across all transaction types is quite high. Specifically, for W_{CPU}, the squared coefficient of variation (C^2) for individual transaction types ranges from $C^2 = 2$ to $C^2 = 5$; however over all transaction types, we measure $C^2 = 15$. W_{IO} is less variable. For individual transaction types we measure values ranging from $C^2 = .15$ to $C^2 = 0.8$, while looking across all transaction types we measure $C^2 = 2.3$. As a reference point, the exponential distribution has $C^2 = 1$.

Figure 10.4(left column) shows the (original) response times for the different transactions under W_{CPU} and W_{IO}, for IBM DB2. Our approach to combatting variability is to decrease the response time of the long transactions (by giving them priority) and in exchange increase the response time of the short transactions, where the goal is to make all transaction response times as close to the overall mean response time as possible. This turns out to be possible because in typical workloads the fraction of transactions with very long response times is quite small as compared with the fraction of transactions with short response times (as in the "80-20 rule"). Figure 10.4(right column) shows the results of equalizing the response times, hence greatly decreasing variance. Under IBM DB2, for W_{CPU} we are able to decrease C^2 from 15 to 0.19. For W_{IO} we are able to decrease C^2 from 2.3 to 0.11. These results are summarized in Tables 10.3 and 10.4. Under PostgreSQL, for W_{CPU} we are able to decrease C^2 from 14 to 0.09. For W_{IO} we are able to decrease C^2 from 1.6 to 0.8.

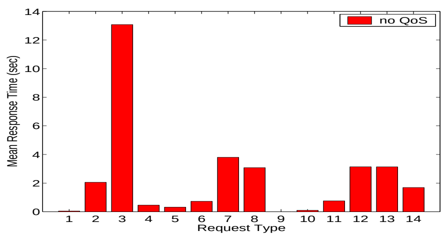
The exact algorithm for reducing variability is easy to implement within our external scheduling framework. We start with the measured overall mean response time of the original system R. We denote the mean response time for the i^{th} transaction type by T_i. Initially some of the T_i's are higher than R and some are lower. To make the system more predictable, we assign type i transactions a target mean response time of $\tau_i = R$. We then apply the standard method for achieving per-class target mean response times.

For this method to work, it is important to note that it is desirable that the variability within each type is low, so that each type is more predictable. For many OLTP servers, e.g. the database backend of a Web site, this is the case: There are a limited number of possible transaction types that the user interface allows for, e.g. ordering, product search, retrieving shopping cart contents, and these transaction types are each limited in scope, resulting in low response time variability within each type.

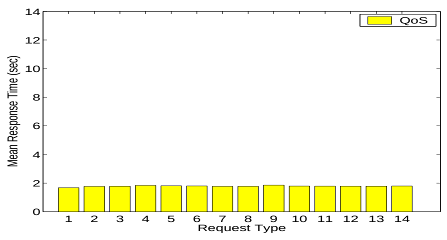
10.4.2 Meeting xth percentile targets

Mean target response times are loose in that they can be heavily influenced by a small percentage of transactions. It is conceivable that some customers might prefer stronger guarantees, namely that 90%, say, of their transactions have response times strictly below some target. In this section we describe how to obtain per-class percentile target guarantees.

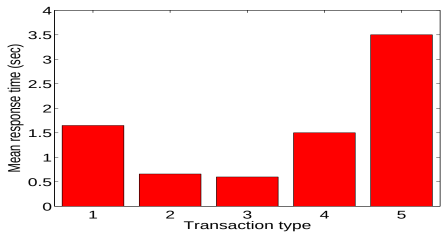
Consider the example of setting a 90^{th} percentile target denoted by $\tau_i^{90\%}$ for the transactions in class i. Our approach for mean response time targets doesn't apply to percentile targets. Thus we need a new approach. Our percentile target approach has two parts: We first determine an MPL value which ensures a 90^{th} percentile goal on just the execution time T_i^{DBMS}. We refer to the 90^{th}



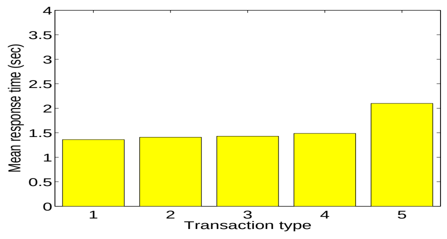
(W_{CPU} before QoS)



(W_{CPU} after QoS)



(W_{IO} before QoS)



(W_{IO} after QoS)

Figure 10.4: *QoS goal reducing variability: Results for W_{CPU}(top) and W_{IO}(bottom).*

percentile of execution times as $T_i^{DBMS\,90\%}$. We next define an algorithm for scheduling the external queue that uses $T_i^{DBMS\,90\%}$ to achieve a 90^{th} percentile goal on the response time for class i.

We solve the first part (achieving a given $T_i^{DBMS\,90\%}$) by "pretending" that our DBMS is a true Processor-Sharing (PS) server. If this is true, then it follows that the $T_i^{DBMS\,90\%}$ scales linearly with the expected number of transactions at the server and the service demand of the transaction (This is based on the queueing-theoretic result that under M/G/1/PS, for a given transaction, its expected response time is proportional to both the number of transactions at the server and its service demand). This enables us to pick an MPL which results in the desired $T_i^{DBMS90\%}$ for all i. The above argument hinges on the assumption that our DBMS behaves similarly to a PS server with respect to the linear scaling of percentile response times as a function of MPL. Our experimental results in Figure 10.5 show that this assumption is in fact valid for our workloads running under IBM DB2. Although not shown, we find that the same result holds for PostgreSQL. We add a disclaimer that the above algorithm works best when all classes have a similar mix of transactions. This was not a necessary condition on all of our previous QoS algorithms, and is needed only here. If that condition is not met, the control loop that adjusts the MPL needs to be more complex.

The second step of our approach is to convert $T_i^{DBMS,90\%}$ into a 90^{th} percentile result for response time. Observe that if the queueing time T_i^Q is bounded by some c, the resulting 90^{th} percentile

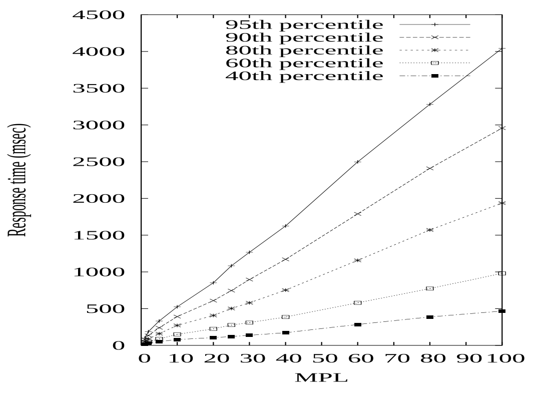


Figure 10.5: *Percentile of the time at the server for different MPLs for W_{IO}.*

response time is bounded by

$$T_i^{90\%} \leq c + T_i^{DBMS,90\%}$$

That means, when scheduling a transaction, in order to ensure a given percentile target $\tau_i^{90\%}$, i.e. ensure that

$$T_i^{90\%} \leq \tau_i^{90\%}$$

the amount of slack we have in scheduling this transaction is

$$\tau_i^{90\%} - T_i^{DBMS,90\%}$$

We can hence translate percentile goals to dispatch targets as follows: assign a transaction with target goal $\tau_i^{90\%}$ the dispatch target of:

$$t_d = t_{curr} + \tau_i^{90\%} - T_i^{DBMS,90\%}$$

As before we schedule transactions from the queue in order of increasing dispatch targets.

Tables 10.3 and 10.4 show results for various experiments with per-class percentile target goals under IBM DB2. As shown, in all experiments, for both workloads, we are able to achieve our percentile response time targets usually within 3%. For our exeriments with PostgreSQL this number becomes 10%.

10.4.3 Combination goals

Finally, it is quite plausible that (i) different customer classes might desire different types of QoS goals, and (ii) a given customer class might simultaneously request multiple target goals (e.g., a goal for the mean and a percentile goal). Both of these "combination" scenarios are easy to achieve in our external scheduling framework since all goals are immediately mapped to dispatch targets and transactions are then pulled from the external queue in order of these targets. A transaction having multiple QoS goals is assigned the most stringent of all of its corresponding dispatch targets.

Some results involving combination goals are shown for IBM DB2 in Tables 10.3 and 10.4, and all targets are achieved with very high accuracy.

10.5 Summary

In this chapter we investigate the question of providing complex QoS goals for OLTP transactions. Our solution is based on the external scheduling framework we started to develop in the previous chapter. Our external scheduling mechanism limits the number of concurrent transactions (MPL) within the DBMS, holding all remaining transactions in an external queue.

We find that for our workloads there is a good range of MPL values which allow us to achieve complex, per-class QoS targets without hurting overall system performance with respect to throughput and overall mean response time. The algorithms needed to achieve the target QoS goals are non-obvious and rely on queueing theory results and analyses. Queueing theory is an integral part of both algorithm design and also feasibility arguments used in the previous chapter. All of our algorithms, including the adaptive ones, are straightforward to implement.

We demonstrate that the above algorithms are effective on our benchmark based workloads, including CPU bound, I/O bound and lock bound workloads, in situtations with multiple classes and multiple targets per class. However, it is desirable to experiment with other real workloads to further validate the algorithms.

Chapter 11

Conclusion and impact

Many time-sensitive applications rely on a DBMS back-end. From the perspective of these applications, the DBMS is an unpredictable, mysterious black box. Transactions are sent into the DBMS and may take either a very short time (msec) or a very long time (tens of secs), depending largely on the *other* transactions concurrently in the DBMS.

This chapter aims to make DBMS response time more predictable by providing a DBA with tools for assigning transactions to different QoS classes. For each class a DBA can specify QoS targets, such as per-class priorities, target response times, percentile targets, variability targets, and combinations of targets.

We have investigated two different approaches for providing QoS for OLTP transactions, *internal* and *external* scheduling. In the internal approach the scheduling is integrated into the DBMS, allowing for maximum control over the DBMS resources. In the external approach scheduling is applied to an external queue, which is created by limiting the number of transactions concurrently executing inside the DBMS (MPL) .

Both approaches have advantages and disadvantages. The internal approach requires knowledge of the resource utilization of the workload, since only scheduling at the bottleneck resource is effective. Moreover, it's implementation relies on modifications to database code. The external approach is resource independent and can be implemented at the application level. However, if not done carefully, it can harm the system's throughput and mean response times, since it requires reducing the concurrency (MPL) inside the DBMS.

In this part of the thesis we study both approaches in detail, using TPC-C and TPC-W workloads and several different general-purpose DBMS. We find that internal priority scheduling using simple policies can yield significant performance improvements for high priority transactions. The best performance for high priority transactions was achieved with *preemptive* scheduling policies, i.e. policies that abort low-priority transactions that are blocking high priority transactions. However, the simple preemptive policies we have studied impose a huge response time penalties for low-priority transactions. This result suggests that more selective preemptive algorithms are an interesting avenue for future research. We discuss such algorithms in Section 11.1 below.

With respect to external scheduling, we begin with an experimental study of the trade-off between maximizing scheduling control, and minimizing negative side effects such as loss in throughput. We find that, in particular for multi-resource systems, a too low MPL can dramatically hurt throughput and therefore a careful choice of the MPL is imperative. However, we also show that for all workloads there exists some feasible MPL that manages to satisfy both the conflicting goals of not hurting throughput or response time while providing sufficient prioritization differentiation between classes. We develop analytical tools for optimizing the MPL and algorithms for scheduling the external queue in order to achieve complex QoS targets. The advantage of external scheduling is that it is done at the application level and therefore very portable. We discuss extensions and the general applicability of our external scheduling framework in Section 11.2.

11.1 More selective preemptive algorithms for internal scheduling

The excessive penalty for low-priority transactions under simple preemptive priority scheduling indicates that a preemptive policy needs to be more selective about which transactions to preempt.

We have experimented with several policies that try to predict a victim transaction's remaining life expectancy and the cost of rolling back the victim to determine whether to preempt or wait. The first idea is to use the number of locks held by the victim to predict its remaining age. If it holds many, it is almost finished, but if it holds few, it is just starting. Second, we use the "wall-clock" age of the victim as a predictor. We find these are both poor predictors of transaction life-expectancy.

In our related work[139] not contained in this thesis we use statistical analysis to study in more detail why simple preemptive schemes fail. We find that the high penalty for low-priority transactions is due to the work wasted on preempted transactions. Moreover, our analysis indicates that the remaining life expectancy of a transactions is large if this transaction is blocked waiting for a lock. Based on this observation we suggest the Preempt-on-Wait (POW) policy, which preempts a low-priority transaction L, which blocks high-priority H, as soon as L must wait for another lock. We find that POW outperforms each of the other policies studied in Chapter 8 on the metric of average high- and low-priority transaction response times.

11.2 More general applicability of external scheduling

Our work on external scheduling focuses on OLTP database workloads and three different types of class-based QoS goals related to response times and predictability. The applications we consider are the database back-end of an online store, represented by the TPC-W benchmark, and an order-entry environment, represented by the TPC-C benchmark. While our experimental work is limited to this particular setting, we believe that there are many possible extensions and interesting directions for future research, which we describe below.

First, while the work in this chapter focuses on three particular types of QoS targets, one of the main contribution of our work is the development of an external scheduling framework that separates the scheduling from the database code. In the internal approach a database developer needs to modify database code to implement a given policy for QoS management. As a results, a database administrator managing a web site is limited to whatever QoS policies the DBMS supports. In particular, there's no flexibility in implementing application specific scheduling policies. By moving the scheduling outside the DBMS, any possible scheduling policy can be implemented at the application level, without any support by the DBMS. We will illustrate the broad applicability of the external scheduling approach in the next chapter, where we will show how it can be used to achieve more complex QoS goals, such as meeting response time targets, targets on the percentile of the response time, or reducing variability in response times.

Second, our external scheduling approach is extremely portable, not just to different DBMS, but also to other types of back-end servers. The queuing theoretic and Markov-chain based analysis in this chapter do not depend on the server being a DBMS and apply to general systems as well. Our work answers questions that have been previously posed in the context of different applications and systems. For example, the very recent work by Chase et al. [106] introduces a method for storage service utility that also relies on choosing an MPL that allows for effective scheduling without compromising overall system efficiency. However, the question of how to choose this MPL is left open in this paper.

Third, in all our experiments the queueing theoretic methods lead to fast convergence times in adapting the system's parameters and accurately met the QoS targets. In settings where this is not the case, the queueing theoretic methods could be augmented with techniques from control theory. Control theory has the advantage of providing convergence guarantees, however it treats each system as a black box ignoring any system specific knowledge that might be available. One way to combine queueing theory and control theory would be to use queueing theory to get a good initial estimate for the value of a parameter, and then use control theory to fine-tune the parameter value. Another option would be to combine both methods within a single framework, that uses in the control loop queueing theory for predictions and control theory for reacting to observed system behavior.

Fourth, our scheduling policies for achieving response time and variability targets rely on good estimates of the time the transactions in a class spend on average inside the DBMS. The same is true for many other QoS policies that we haven't investigated here, such as weighted-fair-queueing algorithms for providing throughput based QoS targets. We found in our experiments that working with a low MPL allows for good estimates accurate QoS results. However, an interesting avenue for future work would be to make scheduling algorithms for QoS more robust toward inaccurate service time estimates. One possible approach would be to modify the scheduling policies to take the current state of the DBMS into account as provided by the DBMS interface (e.g. the transactions currently executing, lock contention) and using information obtained from the DBMS query optimizer. An alternative approach would be to develop algorithms based on adaptations of an SFQ (Start-Time-Fair-Queueing) based algorithm, since SFQ has been shown in the past to be adaptable, e.g. to time-varying server capacities.

Finally, in our work we focus on scheduling policies for three different types of QoS targets that we think are relevant in the context of e-commerce. The big advantage of our work, though, is that makes it very easy for system administrators to implement support for any other QoS target of his or her choice, since it separates the scheduling policy from the database internals. In addition to giving administrators the possibility of implementing their own scheduling outside the DBMS, our work also provides methods, e.g. for and adapting the MPL to ensure feasibility of targets, that will be useful in providing other types of QoS targets as well.

Part III

Experimental models: Understanding their impact on performance evaluation and system design

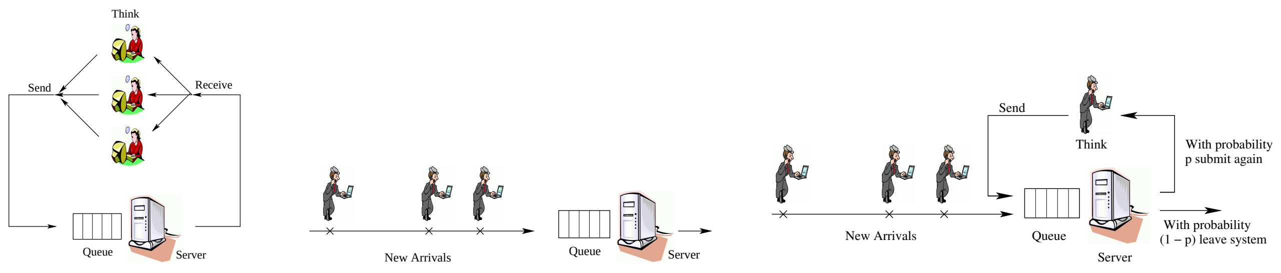
Chapter 12

Introduction to Part III

Every systems researcher is well aware of the importance of setting up one's experiment so that the system being modeled is "accurately represented." Representing a system accurately involves many things, including accurately representing the bottleneck resource behavior and the scheduling of requests at that bottleneck, as well as accurately representing workload parameters, such as the distribution of service request demands, popularity distributions, locality distributions, and correlations between requests.

In the case of web workloads, for example, there exist extensive studies based on logs from web servers or proxy caches that analyze the different characteristics of web traffic. From these studies it is well known that the distribution of the requested file sizes is heavy-tailed, or that the popularity distribution follows Zipf's law. Workload generators are then often engineered to imitate those characteristics.

However, one factor that researchers typically do not pay much attention to is whether the job arrival stream obeys a *closed* or *open* system model. Open and closed models differ in the process by which users generate requests to the system. Figure 12.1(a) depicts a **closed system** configuration. In a closed system model, it is assumed that there is some fixed number of users, who use the system forever. This number of users is typically called the *multiprogramming level* (MPL) and denoted by N. Each of these N users repeats these 2 steps, indefinitely: (a) submit a job, (b) receive the response and then "think" for some amount of time. In a closed system, *a new request is only triggered by*



(a) Closed system (b) Open system (c) Partly-open system

Figure 12.1: *Illustrations of three system configurations.*

the completion of a previous request. At all times there are some number of users, N_{think}, who are thinking, and some number of users N_{system}, who are either running or queued to run in the system, where $N_{think} + N_{system} = N$. The *response time*, T, in a closed system is defined to be the time from when a request is submitted until it is received (this does not include the subsequent think time, which is denoted by Z). In the case where the system is a single server (e.g. a web server), the *server load*, denoted by ρ, is defined as the fraction of time that the server is busy, and is the product of the mean throughput X and the mean service demand $E[\chi]$.

Figure 12.1(b) depicts an **open system** configuration. In an open system model, each user is assumed to submit one job to the system, wait to receive the response, and then leave. The number of users queued or running at the system at any time may range from zero to infinity. The differentiating feature of an open system is that a *request completion does not trigger a new request: a new request is only triggered by a new user arrival.* As before, *response time*, T, is defined as the time from when a request is submitted until it is completed. Also as before, for the case of a single server the *server load* is defined as the fraction of time that the server is busy. Here load, ρ is the product of the mean arrival rate of requests, λ, and the mean service demand $E[\chi]$.

Neither the open system model nor the closed system model is entirely realistic. Consider for example a web site. On the one hand, a user is apt to make more than one request to a web site, and the user will typically wait for the output of the first request before making the next. In these ways a closed system model makes sense. On the other hand, the number of users at the site varies over time; there is no sense of a fixed number of users N. The point is that users visit to the web site, behave as if they are in a closed system for a short while, and then leave the system.

Motivated by the example of a web site, a more realistic alternative to the open and closed system configurations is the **partly-open system** shown in Figure 12.1(c). Under the partly-open model, users arrive according to some outside arrival process as in an open system. However, every time a user completes a request at the system, with probability p the user stays and makes a followup request (possibly after some think time), and with probability $1-p$ the user simply leaves the system. Thus the expected number of visits/requests that a user makes to the system is Geometrically distributed with mean $1/(1-p)$. The *server load* is again defined as the fraction of time that the server is busy. This equals the product of the average outside arrival rate λ, the mean number of visits $E[V]$, and the mean service demand $E[\chi]$. For a given load, when p is small, the partly-open model is more similar to an open model. For large p, the partly-open model resembles a closed model.

Modern workload generators differ in which underlying system model they assume. Even within one type of workload generators, such as web workload generators there is no consensus with respect to the system model. Some web workload generators follow an open system model, while others follow a closed system model. When it comes to analytical work, the focus in classical queueing theory is on open system models, but work for both system models exists. However, there's very little work that compares the two models against each other and analyzes the effect that the choice of the system model has on important metrics such as response times.

Below we first survey the use of system models in current workload generators and then summarize theoretical results on the difference of the two models. We then outline our work presented in the remainder of this part of the thesis.

12.1 System models in existing workload generators

Workload generators are used in many, very different research areas. Even within web-related research, workload generators are used in a variety of applications: (a) scheduling and resource allocation [25, 24, 123, 131]; (b) caching, at the operating system level, as well as at the network level [161, 166, 85]; and (c) evaluation of operating system mechanisms, such as multithreading [14, 137, 170].

Table 12.1 attempts to survey the system models in a large variety of web related workload

Type of benchmark	Name	System model
Model-based web workload generator	Surge [32], WaspClient [153], Geist [109], WebStone [199], WebBench [201], MS Web Capacity Analysis Tool [143]	Closed
	SPECWeb96 [195], WAGON [129]	Open
Playback mechanisms for HTTP request streams	MS Web Application Stress Tool [144], Webjamma [2], Hammerhead [189], Sclient [26], Deluge [188], Siege [84]	Closed
	httperf [151]	Open
Proxy server benchmarks	Wisconsin Proxy Benchmark [15], Web Polygraph [177], Inktomi Climate Lab [86]	Closed
Standard database benchmark for e-commerce workloads	TPC-W [198]	Closed
Standard database benchmark for online transaction processing (OLTP)	TPC-C [69]	Closed
Model-based packet level web traffic generators	IPB (Internet Protocol Benchmark) [135], GenSyn [94] WebTraf [80], trafgen [65]	Closed
	NS traffic generator [215]	Open
Mail sever benchmark	SPECmail2001 [194]	Open
Java Client/Server benchmark	SPECJ2EE [193]	Open
Benchmark for web authentication and authorization	AuthMark [145]	Closed
Benchmark for network file servers	NetBench [200]	Closed
	SFS97_R1 (3.0) [192]	Open
Synthetic streaming media service workload generator	MediSyn [197]	Open

Table 12.1: *A summary table of the system models underlying standard web related workload generators.*

generators used by systems researchers today. The table is by no means complete; however it illustrates the wide range of workload generators and benchmarks available. For many of these workload generators, it was quite difficult to figure out which system model was being assumed — the builders often do not seem to view this as an important factor worth mentioning in the documentation. Thus the “choice” of a system model (closed versus open) is often not really a researcher’s choice, but rather is dictated by the availability of the workload generator. Even when a user makes a conscious choice to use a closed model, it is not always clear how to parameterize the closed system (e.g. set the think time and MPL) and what effect these parameters have.

Most of the workload generators in Table 12.1 use closed system configurations, but occasionally an open systems is assumed. Web workload generators fall in two categories. The first category provides a model capturing the essential characteristics of HTTP requests seen at a server and generates requests based on this model. Such workload generators, e.g. Surge [32], are used by many researchers [154, 123, 14]. The second category of web workload generators just provides the mechanisms for generating requests, and the user has to specify which objects to request through this mechanism. These too are very popular. For example the Sclient [26] web workload generator is used in [27, 160, 25].

Looking outside of web workload generators, there are two other popular benchmark suites in systems research. The first, run by the Transaction Processing Council (TPC), provides database benchmarks, e.g., TPC-C for OLTP workloads and TPC-W for e-commerce workloads. All TPC benchmarks follow the closed system model. The second, provided by the Standard Performance Evaluation Corporation (SPEC), is used in many different systems, e.g. mail servers, file servers, web servers, etc.. In contrast to TPC, all the SPEC benchmarks are open, which the documentation claims is convenient for controlling the correct load mix.

The documentation of the SPEC mail server benchmark [194] gives three different reasons for using an open system. First, under a closed system the server could conveniently throttle the load dynamically - whenever things get critical, its response time would go up, and therefore the load down. Second, it would also be more difficult to control a correct load mix - a server might serve more requests of the type which are convenient for it. Third, response times might become unacceptable, and therefore the reported rate meaningless. The SPEC benchmarks are also quite popular. For example, SPECmail is used in work on reducing IO by improving the queue structure of the IO subsystem [208]. SPECweb is used in [137].

In addition, the other workload generators/benchmarks listed in Table 12.1 tend to be a mixture of open or closed, but never both. These are used in many research papers, for example, the MediSyn benchmark is used in work on resource allocation in streaming media servers in [64].

12.2 System models in analytical research

The world of analytical models is similar to that of workload generators in that there is a great divide between the books/papers that assume open system configurations and those that assume closed system configurations, with a big gap in understanding. Most traditional theoretically-oriented queueing books, e.g., Ross [176], Wolff [211] and others [68, 114] are devoted to open system models and don't mention closed models. By contrast typical systems-oriented performance modeling books, e.g., Menasce et.al. [140] are exclusively devoted to closed system models and their applications. There are very few performance modeling or queueing theory books that give a thorough treatment to both open and closed systems [105, 113], and even these books treat open and closed systems independently: they do not provide a comparison of the systems, nor do they provide intuition as to when one type of system might be more appropriate than the other.

Even within research papers, very little analytical work has looked at the comparisons between open and closed models. There has been a tremendous amount of prior analytical work involving scheduling within *open* system models (see e.g., [68, 113, 114, 211]). In *closed* systems, the analysis is often more difficult, and hence is either restricted to product-form networks, [34, 158, 210] or takes the form of approximations for size-based policies, e.g. [112, 178, 205].

Work explicitly comparing open and closed system models is primarily limited to FCFS queues. For example, Bondi and Whitt [42] study a general network of FCFS queues and give principles about how service variability effects open and closed systems. They conclude that the effect of service variability, though dominant in open systems, is almost inconsequential in closed systems (provided the MPL is not too large). We corroborate this principle in Section 13.1.1, where we also elucidate the more subtle effects of service variability and justify why MPL plays such a key role. Further, we will illustrate the magnitude of this effect in a variety of real-world case studies in Chapter 14.

Two crucial papers of Schatte [179, 180] again study a single FCFS queue in a closed loop with think time. In this model, Schatte proves that, as the MPL grows to infinity (and the think time is adjusted to maintain a fixed system load), the closed system converges monotonically to an open system. This result provides a fundamental understanding of the effect of the MPL parameter; however Schatte does not evaluate the rate of this convergence, which is important when choosing between open and closed system models.

Though these theoretical results provide many useful intuitions about the differences between open and closed systems, theoretical results alone cannot evaluate the effects of factors such as trace driven job service demand distributions, correlations, implementation overheads, and more complex scheduling policies. Hence, simulation and implementation-based studies are needed.

Outline of Part III

Our work aims to provide systems researchers with guiding principles on the impact of choosing a closed, open, or partly-open system model. Our principles extend the queueing-theoretic understanding of this topic which rests on two intuitions. First, the queue length is bounded in a closed system and unbounded in an open system. Second, open systems can be run under overload; whereas closed systems cannot. While these two intuitions are important, they do not provide an understanding of the magnitude of the difference between closed and open in real-world applications, and the effect of parameters such as think time, MPL, variability in service demands. Moreover, theoretical work is primarily limited to FCFS and PS. We have seen in Part I of this thesis that size-based scheduling policies are a useful mechanism for improving mean response time. From existing work it is not clear what the impact of different scheduling policies in the different system models is.

In this part of the thesis we investigate the above questions in detail, and we further discuss partly-open systems, and whether an open or closed model best approximates the partly-open system.

We start our study in Chapter 13 in a controlled setting using simulation experiments to compare the behavior of closed, open, and partly-open systems. We begin by looking at the simplest case, First-Come-First-Served (FCFS) scheduling, and then move to evaluating the effect of size based scheduling policies in open and closed systems. Finally, we transition to the more realistic partly-open system model.

To demonstrate the practical effect of the the principles from Chapter 13, we investigate in Chapter 14 the question of open versus closed systems in real-world case studies: in web servers receiving static HTTP requests; in the backend database in e-commerce applications; in an auctioning web site; and at a supercomputing center.

We conclude in Chapter 15 with a summary and a discussion of how system designers should pick an appropriate system model based on workload characteristics.

Chapter 13

Open vs closed vs partly-open system models

The goal of this chapter is to compare the behavior of closed, open, and partly-open systems in a controlled setting using simulation experiments. We begin by defining the relevant parameters and metrics for both the open and the closed system models and discuss how we set parameters in order to compare open and closed system models.

The relevant parameters for a closed system are the MPL N, the think time Z, and the distribution of the job service demands χ. The relevant parameters for an open system are the arrival rate λ and the distribution of the job service demands χ. Throughout the chapter we choose the service demand distribution to be the same for the open and the closed system. We use hyperexponential service demands, in order to capture the highly variable service distributions in web applications. In the case studies in the next chapter the service demand distribution is either taken from a trace or determined by the benchmark used in the experiments. Throughout, we measure the variability in the service demand distribution using the square coefficient of variation, C^2. The think time in the closed system follows an exponential distribution, and the arrival process in the open system is either a Poisson arrival process or provided by traces. The results for all simulations and experiments are presented in terms of mean response times and the system load ρ. While we do not explicitly report numbers for another important metric, mean throughput, an interested reader can directly infer those numbers by interpreting load as a simple scaling of throughput: In an open system, the mean throughput is simply equal to the mean arrival rate $\lambda = \rho/E[\chi]$. In a closed system the mean throughput is also equal to $\rho/E[\chi]$.

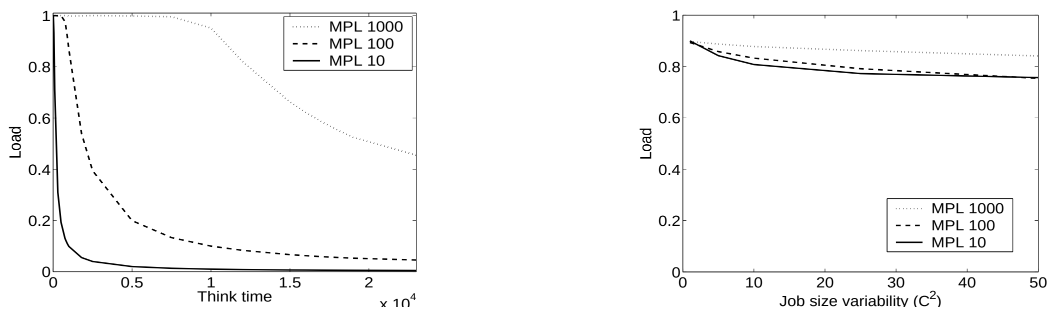
In order to fairly compare the open and closed systems, we will hold the load (utilization) of the two systems equal, and study the effect of open versus closed system models on mean response time[1]. The load/utilization in the open system is $\rho = \lambda E[\chi]$, and fixing the load is therefore equivalent to fixing λ. Fixing the load of a closed system is more complex, since the load is affected by many parameters including the MPL, the think time, the service demand variability, and the scheduling policy. The fact that system load is influenced by many more system parameters in a closed system than in an open system is a surprising difference between the two systems, and we will gradually build an understanding of this difference in Section 13.1. *Throughout, we will achieve a desired system load by adjusting the think time of the closed system (see Figure 13.1(a)).* This is because increasing (decreasing) the think time has the affect of decreasing (increasing) the mean arrival rate to the queue in a closed system, which is parallel to decreasing (increasing) the mean arrival rate in an open system.

In the remainder of this chapter we will first compare open versus closed systems, and then transition to the more realistic partly-open system model.

13.1 Open versus Closed Systems

We will begin our comparison of open and closed system models by assuming that jobs at the bottleneck resource (server) are scheduled according to FCFS, as shown in Figure 13.2. The study of this simple case will help illustrate principles that we will exploit when studying more practical policies. However, even for a simple FCFS server, the comparison between closed and open models is non-trivial and counter-intuitive.

[1]Other comparisons of open and closed systems are possible. Although it obviously makes sense to fix the service distribution across both models, instead of fixing the load one might instead fix the *throughput* of both systems or alternatively fix the *response time* of both systems. Fixing the throughput of both systems is equivalent to fixing the load of both systems; however, fixing the response time of both systems would lead to a different comparison. In this work, we choose to fix the load because we are interested in understanding how response times compare across the two systems, which cannot be understood by holding the response time constant.



(a) Think time vs. load (b) Variability vs. load

Figure 13.1: *We illustrate how the service demand variability, the MPL, and the think time can affect the system load in a closed system. These plots use FCFS scheduling, however results are similar under other scheduling policies. We will discuss the impact of scheduling policies on load in more detail in Section 13.1.2.*

13.1.1 FCFS

In this section we present three principles relating open and closed systems under FCFS scheduling.

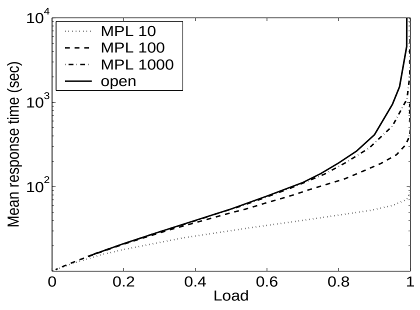
Principle (i): ***Mean response times are significantly lower in closed systems than in open systems for a given load.***

Principle (i) is perhaps the most noticeable performance issue differentiating open and closed systems. As shown in Figure 13.2(a), for fixed high loads, the response time under a closed system is *orders of magnitude* lower than for the open system. While Schatte [179, 180] has proven that, under FCFS, the open system will always serve as an upper bound for the response time of the closed system, the magnitude of the difference has not previously been studied.

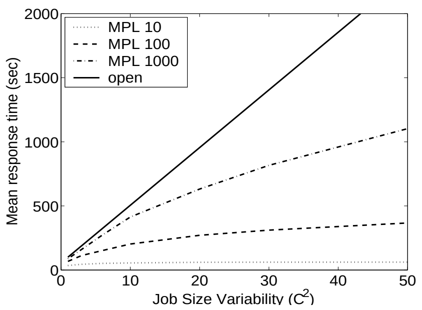
Intuitively, this difference in mean response time between open and closed systems is a consequence of the fixed MPL, N, in closed systems, which limits the queue length seen in closed systems to N even under very high load. By contrast, no such limit exists for an open system.

This principle is important for the vast literature on capacity planning [140], which typically relies on closed models, and hence may underestimate the resources needed for open models.

Open vs. Closed: FCFS



(a) Resp. time vs. load



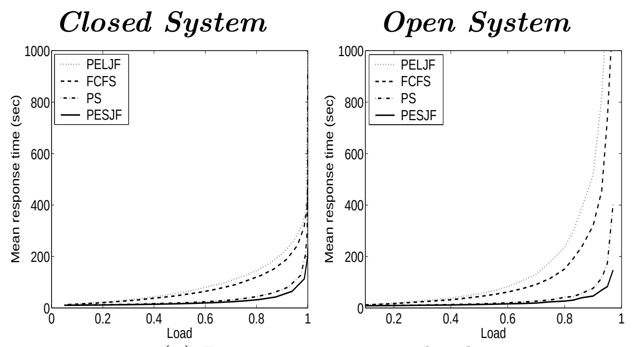
(b) Resp. time vs. variability

Figure 13.2: *Mean response time as a function of (a) load and (b) service demand variability under FCFS scheduling. The solid line represents an open system and the dashed lines represent closed systems with different MPLs. In all cases, $E[\chi] = 10$. In (a) we fix the service demand variability ($C^2 = 8$) and in (b) we fix the load ($\rho = 0.9$). The load is adjusted via the think time in the closed system, and via the arrival rate in the open system. The results show a significant difference between response times in closed and open systems, where the difference increases as the MPL decreases.*

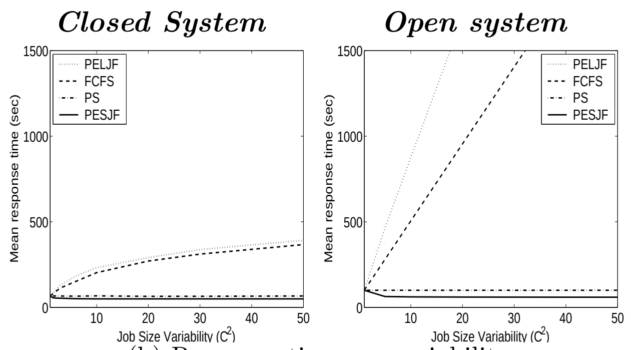
Principle (ii): ***As the MPL grows closed systems become open, but convergence is slow for practical purposes.***

Principle (ii) is illustrated by Figure 13.2. We see that as the MPL, N, increases from 10 to 100 to 1000, the curves for the closed system approach the curves for the open system. Schatte [179, 180] proves formally that as N grows to infinity, a closed FCFS queue converges to an open M/GI/1/FCFS queue. What is interesting however, is how slowly this convergence takes place. When the service demand has high variability (C^2), a closed system with an MPL of 1000 still has much lower response times then the corresponding open system. Even when the job service demands are lightly variable, an MPL of 500 is required for the closed system to achieve response times comparable to the corresponding open system.

We can explain the convergence of a closed system to an open one as N gets very large. In the open M/GI/1 system, the arrival process is Poisson and thus has exponential interarrival times each having a constant rate, which is independent of the completions at the server. In the closed system, the interarrival times are governed by the exponential think times; however, the rate changes with each job completion. When N is small, the rate can change drastically. When N is large though,



(a) Response time vs. load



(b) Response time vs. variability

Figure 13.3: *We illustrate the different effects of scheduling in closed and open systems. In the closed system the MPL is 100, and in both systems the service demand distribution has mean 10 and $C^2 = 8$. Notice that there is a much smaller difference between the best and worst scheduling policies across loads and variabilities in the closed system, than there is in the open system.*

there will likely be many jobs "thinking" at any given point in the closed system. So a completion, which increments $N_{thinking}$ by one, has very little effect on the arrival rate. As N goes to infinity, the effect of a completion on the arrival rate disappears completely and the closed arrival process matches the open arrival process.

This intuitive principle impacts the choice of whether an open or closed system model is appropriate. One might think that for a closed system with a high MPL a reasonable approximation would be to use an open system model; however, it turns out that the closed and open system models may still behave significantly differently if the service demands are highly variable.

Principle (iii): ***While variability has a large effect in open systems, the effect is much smaller in closed systems.***

We now turn to Figure 13.2(b), which compares open and closed systems under a *fixed load* $\rho = 0.9$, as a function of the service demand variability C^2. For an open system we see that C^2 directly affects mean response time. This is to be expected since high C^2, under FCFS service, results in short jobs being stuck behind long jobs, increasing mean response time. In contrast, for the closed system with MPL 10, C^2 has comparatively little effect on mean response time. This is counterintuitive, but can be explained by observing that for lower MPL there are *fewer* short jobs stuck behind long jobs in a closed system, since the number of jobs in the system (N_{system}) is bounded. As MPL is increased, C^2 can have more of an effect, since N_{system} can be higher.

It is important to point out that by holding the load constant in Figure 13.2(b), we are actually performing a conservative comparison of open and closed systems. If we didn't hold the load fixed as we changed C^2, increasing C^2 would result in a slight drop in the load of the closed system as shown in Figure 13.1(b). This slight drop in load, would cause a drop in response times for closed systems, whereas no such effect takes place in open systems.

This principle is most significant for supercomputing applications where C^2 can be very high ($C^2 = 43$) and jobs are scheduled in FCFS order (see Section 14.4).

13.1.2 The impact of scheduling

The value of scheduling in open systems is well understood and cannot be overstated. In open systems, there are order of magnitude differences between the performance of scheduling policies because scheduling can prevent small jobs from queueing behind large jobs. In contrast, scheduling in closed systems is not well understood.

A wide variety of policies are used in computer systems. We focus on four policies that span the range of behaviors:

FCFS (First-Come-First-Served): Jobs are processed in the same order as they arrive.

PS (Processor-Sharing): The server is shared evenly among all jobs in the system.

PESJF (Preemptive-Expected-Shortest-Job-First): The job with the smallest expected duration (size) is given preemptive priority.

PELJF (Preemptive-Expected-Longest-Job-First): The job with the longest expected size is given preemptive priority. PELJF is an example of a policy that performs badly and is included to understand the full range of possible response times.

In this section we present 3 principles contrasting the impact of scheduling in open and closed systems. We demonstrate these principles by evaluating response times in closed and open systems under the above scheduling policies in simulation. The service demands are drawn from a hyperexponential distribution allowing for two classes of jobs – those short in expectation and those long in expectation. In the case where the exact service demand of a job is known a priori, one can apply an exact size based policy. More commonly, however, service demands are only known approximately (in expectation).

Principles (iv,v):

(iv) While open systems benefit significantly from scheduling with respect to response time, closed systems improve much less.

(v) Scheduling only significantly improves response time in closed systems under very specific parameter settings: moderate load (moderate think times) and high MPL.

Figure 13.3 illustrates the fundamentally different behavior of mean response time in the open and closed systems as a function of (a) load and (b) variability. Under the open system, as load increases, the disparity between the response times of the scheduling policies grows, eventually differing by orders of magnitude. In contrast, at both high and low loads in the closed system, the scheduling policies all perform similarly; only at moderate loads is there a significant difference between the policies — and even here the differences are only a factor of 2.5. Another interesting point is that, whereas for FCFS the mean response time of an open system bounded that in the corresponding closed system from above, this does not hold for other policies such as PESJF, where the open system can result in lower response times than the closed system.

We can build intuition for the limited effects of scheduling in closed systems by first considering a closed feedback loop with no think time. In such a system, surprisingly, the scheduling done at the queue is inconsequential – all work conserving scheduling policies perform equivalently. To see why, note that in a closed system Little's Law states that $N = XE[T]$, where N is the constant MPL across policies. We will now explain why X is constant across all work conserving scheduling policies (when think time is 0), and hence it will follow that $E[T]$ is also constant across scheduling policies. X is the long-run average rate of completions. Since a new job is only created when a job completes, over a long period of time, all work conserving scheduling policies will complete the same set of jobs plus or minus the initial set N. As time goes to infinity, the initial set N becomes unimportant; hence X is constant. This argument does not hold for open systems because for open systems Little's Law states that $E[N] = \lambda E[T]$, and, though λ is constant, $E[N]$ is not constant across scheduling policies.

Under closed systems with think time, we now allow a varying number of jobs in the queue, and thus there is some difference between scheduling policies. However, as think time gets high, load becomes small and so scheduling has little effect.

Throughout Figure 13.3, the MPL is held constant at 100. Recall from Principle (ii) in Section 13.1.1 that the effect of MPL is to transition between open and closed systems. Thus, under smaller MPL, the effects of scheduling are even less noticeable in the closed system; however, for larger MPL scheduling can have a larger effect.

A very subtle effect, not yet mentioned, is that in a closed system the scheduling policy actually affects the throughput, and hence the load. "Good" policies, like PESJF, increase throughput, and hence load, slightly (less than 10%). Had we captured this effect (rather than holding the load fixed), the scheduling policies in the closed system would have appeared even closer, resulting in even starker differences between the closed and open systems.

The impact of Principles (iv) and (v) is clear. For closed systems, scheduling provides small improvement across all loads, but can only result in substantial improvement when load (think time) is moderate. In contrast, scheduling always provides substantial improvements for open systems.

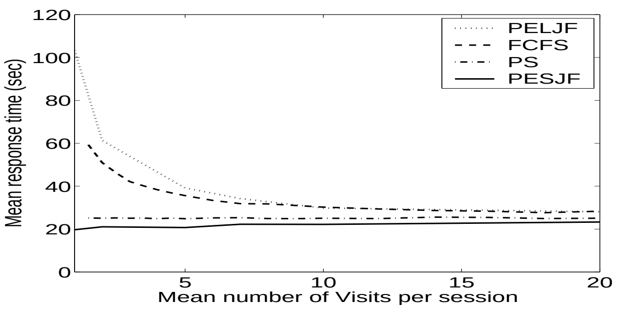
Principle (vi): ***Scheduling can limit the effect of variability in both open and closed systems.***

For both the open and closed systems, better scheduling (PS and PESJF) helps combat the effect of increasing variability. The improvement; however, is less dramatic for closed systems due to Principle (iii) in Section 13.1.1, which tells us that variability has less of an effect on closed systems in general.

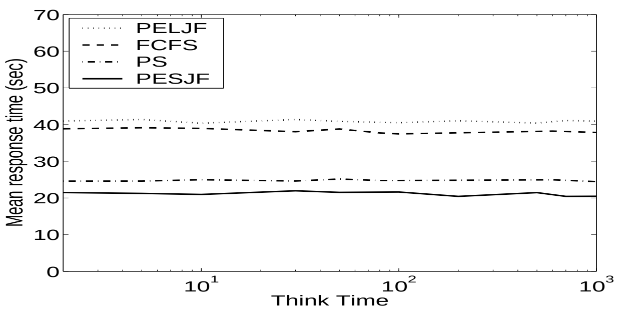
13.2 Partly-open systems

In this section, we discuss a partly-open model that (a) serves as a more realistic system model for many applications; and (b) helps illustrate when a "purely" open or closed system is a good approximation of user behavior.

Partly-open System



(a) Resp. Time vs. # visits



(b) Resp. Time vs. Think time

Figure 13.4: *For the partly-open system, (a) shows mean response time as a function of the expected number of visits per user and (b) shows the mean response time as a function of the think time between visits for a fixed load. In both cases,* $\rho = 0.6$*,* $E[\chi] = 10$*, and the job sizes have moderate variability (*$C^2 = 8$*). In (b), we fix* $p = 0.75$*, which leads to a mean number of visits of 4.*

The partly-open model we discuss aims to mimic user behavior at a web site where, after making a request, the user will stay and make another request with probability p. This model has been mentioned both by prior theoretical [175, 82, 213] and implementation [92] research. Many other variations of partly-open systems have also been proposed in the literature. For instance, Dowdy and Chopra create a hybrid system by specifying the MPL of a closed system using a probability distribution [75]. Another proposed hybrid model places upper and lower bounds on the number of jobs allowed into an open system [120]. A differentiating feature of the partly-open model we discuss is its behavioral nature.

In this section we focus on the effects of the mean number of visits and the think time because the other parameters, e.g. load and job size variability, have similar effects to those observed in Sections 13.1.1 and 13.1.2. We introduce two principles that discuss the impact of scheduling in partly-open systems. Throughout the section we fix the load of the partly-open system by adjusting the arrival rate, λ. Note that, in contrast to the closed model, adjusting the think time of the partly-open model has no impact on the load.

Principle (vii): ***A partly-open system behaves similarly to an open system when the expected number of visits is small (less than*** 10 ***as a rule-of-thumb) and similarly to a closed system when the expected number of visits is large.***

Principle (vii) is illustrated in Figure 13.4(a). When the mean number of visits is 1 we have a significant separation between the response time under the scheduling policies, as in open systems. However, when the mean number of visits is large, we have comparatively little separation between the response times of the scheduling policies; as in closed systems. This plot is just one example of the range of configurations we studied. Across a wide range of parameters, the point where the separation between the performance of scheduling policies becomes small is, as a rule-of-thumb, around 10 visits, though this point can range between 3 and 20 visits as C^2 ranges from 4 to 49 respectively.

This rule-of-thumb serves as a guideline for choosing between a purely open or a purely closed generator (when no partly-open generator is available) based on the average number of requests per user. For example, at the online bookstore and auction site discussed by Menascé and Almeida [141], only 10% of all sessions have more than 10 visits. Thus, using principle (vii), it appears an open system is more appropriate than a closed for this application.

Principle (viii): ***In a partly-open system, think time has only little effect on mean response time.***

Figure 13.4(b) illustrates Principle (viii). We find that the think time in the partly-open system does not affect the mean response time or load of the system under any of these policies. This observation holds across all partly-open systems we have investigated (regardless of the number of visits), including the case-studies described in Chapter 14.

Principle (viii) may seem surprising at first, but for PS and FCFS scheduling it can be shown formally under product form workload assumptions. Intuitively, we can observe that changing the think time in the partly-open system has no effect on the load because the same amount of work must be processed. To change the load, we must adjust either the number of visits or the arrival rate. The only effect of think time is to add small correlations into the arrival stream.

Chapter 14

Real-world case studies

In this chapter we compare the behavior of four different applications under closed, open, and partly open system models. The four applications include (a) a web server delivering static content, (b) the database back-end at an e-commerce web site, (c) the application server at an auctioning web site and (d) scheduling at a supercomputing center. These applications vary in many respects, including the bottleneck resource, the workload properties (e.g. job size variability), and the types of scheduling policies considered. We study applications (a) and (b) through full implementation in a real testbed, while our study of applications (c) and (d) relies on trace-driven simulation.

Sections 14.1 to 14.4 describe our implementation/simulation setup of the four applications. In Section 14.5 we discuss the results of evaluating these applications under closed, open, and partly open models, in relation to the eight principles that we established.

14.1 Implementation: Static web content

Our first case study is an Apache web server running on Linux and serving static content, i.e. requests of the form "Get me a file." Our experimental setup involves six machines connected by a 10/100 Ethernet switch. Each machine has an Intel Pentium III 700 MHz processor and 256 MB RAM, and runs Linux 2.2.16. One of the machines is designated as the server and runs Apache 1.3.14. The others generate web requests based on a web trace.

Standard scheduling of static requests in a web server is best modeled by processor sharing (PS).

However, recent research suggests favoring requests for small files can improve mean response times at web servers [92]. In this section we therefore consider both PS and SRPT (Shortest-Remaining-Processing-Time-First) policies.

Scheduling at the server: We have modified the Linux kernel and the Apache Web server to implement SRPT scheduling at the server. For static HTTP requests, the network (access link out of the server) is typically the bottleneck resource. Thus, our solution schedules the bandwidth on this access link by controlling the order in which the server's socket buffers are drained. Traditionally, the socket buffers are drained in Round-Robin fashion (similar to PS); we instead give priority to sockets corresponding to connections where the remaining data to be transferred is small. Figure 14.2 shows the flow of data in Linux after our modifications. There are multiple priority queues and queue i may only drain if queues 0 to $i-1$ are empty. The implementation is enabled by building the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queuing, and the Prio Pseudoscheduler and by using the `tc`[17] user space tool. We also modify Apache to use `setsockopt` calls to update the priority of the socket as the remaining size of the transfer decreases. For details on our implementation see [92].

Generating the workload: We use a trace-based workload consisting of 1-day from the 1998 World Soccer Cup, obtained from the Internet Traffic Archive [101]. Virtually all requests in this trace are *static*. In our open system experiments we use the trace to specify both the *time* the client makes the request and the *size in bytes* of the file requested. For the closed system experiments, only the file sizes are used. Some statistics about the trace workload follow.

Number of Files	Mean size	Variability (C^2)	Min size	Max size
$4.5 \cdot 10^6$	5KB	96	41 bytes	2MB

14.2 Implementation: Online bookstore

Our second case study considers the database backend server of an online bookstore. We use a PostgreSQL[164] database server running on a 2.4-GHz Pentium 4 with 3GB RAM, running Linux 2.4.23, with a buffer pool of 2GB. The machine is equipped with two 120GB IDE drives, one used for the database log and the other for the data. The workload is generated by four separate client machines having similar specifications to the database server and connected via a network switch.

The workload: We experiment with the TPC-W [198] (TPC-W Browsing Mix) benchmark, which aims to model an online retailer such as Amazon.com. TPC-W consists of 16 different transaction types including the "ShoppingCart" transaction, the "Payment" transaction, and others. Statistics of our configuration are as shown:

Database size	Mean service	Variability (C^2)	Min service	Max service
3GB	101 ms	4	2 ms	5s

Implementing scheduling in the database server: The bottleneck resource in our setup is the CPU, as observed in our earlier work [138]. The default scheduling policy is therefore best described as PS, in accordance with Linux CPU scheduling. In addition we experiment with the SEJF and LEJF policies. The expected length of a transaction is based on its type. In our setup the "Bestseller" transaction, which makes up 10% of all requests, is on average the longest running transaction. In the context of the SEJF and the LEJF policies the "Bestseller" transactions are therefore "expected to be long" and all other transactions are "expected to be short."

To implement the priorities needed for achieving SEJF and LEJF, we modify our PostgreSQL server as follows. We use the `sched_setscheduler()` system call to set the scheduling class of a PostgreSQL process working on a high priority transaction to "SCHED_RR," which marks a process as a Linux real-time process. We leave the scheduling class of a low priority process at the standard "SCHED_OTHER." Real-time processing in Linux always has absolute, preemptive priority over standard processes.

14.3 Simulation: Auctioning web site

Our third case investigates a top-ten online auction site in the U.S. via trace-driven simulation. The trace contains the service demands for 300000 different requests as shown:

Number of Jobs	Mean Service Reqt. (sec)	Variability (C^2)	Min (sec)	Max (sec)
300000	0.09	9.19	0.01	50

Since no data on the request arrival process is available to us, in our simulations of the open and hybrid systems we generate a Poisson arrival process. We consider three policies: FCFS, PS, and SRPT.

14.4 Simulation: Supercomputing center

In this section we model the Cray J90 and Cray C90 machines at the Pittsburgh Supercomputing Center (PSC)[3]. These servers have between 4 and 16 processors and typically execute exactly one job at a time. The jobs are run-to-completion, i.e. no preemption or no timesharing.

In addition to standard FCFS we consider two size-based policies that are of interest in this setting. The Non-preemptive-Shortest-Job-First (SJF) policy gives preference to the shortest job, while the Non-preemptive-Longest-Job-First policy (LJF) favors the longest one.

The two traces used were collected from January through December in 1997 and are summarized below.

System	Number of Jobs	Mean Service (sec)	C^2	Min Service (sec)	Max Service (10^6 sec)
PSC C90	54962	4562.6	43.16	1	2.22
PSC J90	3582	9448.6	10.02	4	0.61

14.5 Results and discussion

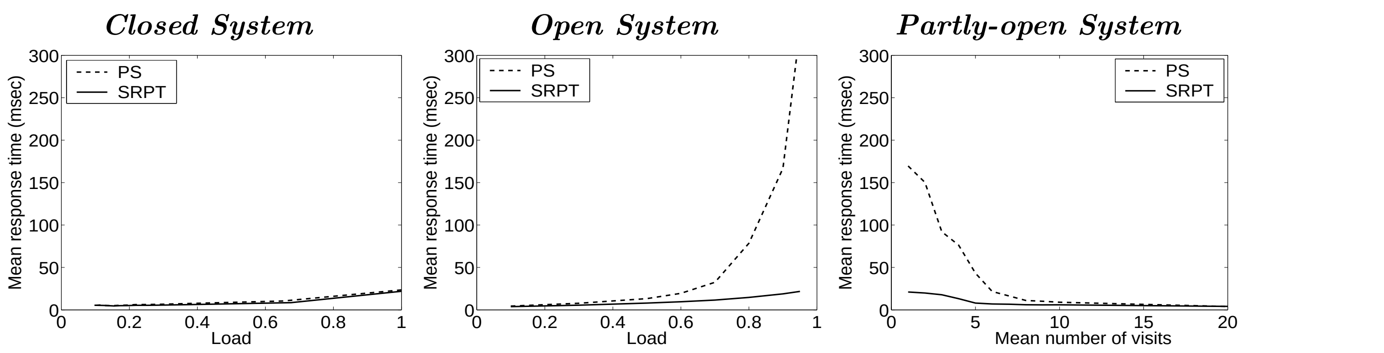
Figure 14.1 shows results from the four case studies described above. Each application is implemented and evaluated under a closed, open, and partly open system model.

Looking at Figure 14.1, we see that the closed system response times are vastly different from the open response times. Response times can be orders of magnitude lower for the closed system than the open system (Principle (i)). Furthermore, scheduling is far less effective for the closed systems than the open ones (Principle (iv)).

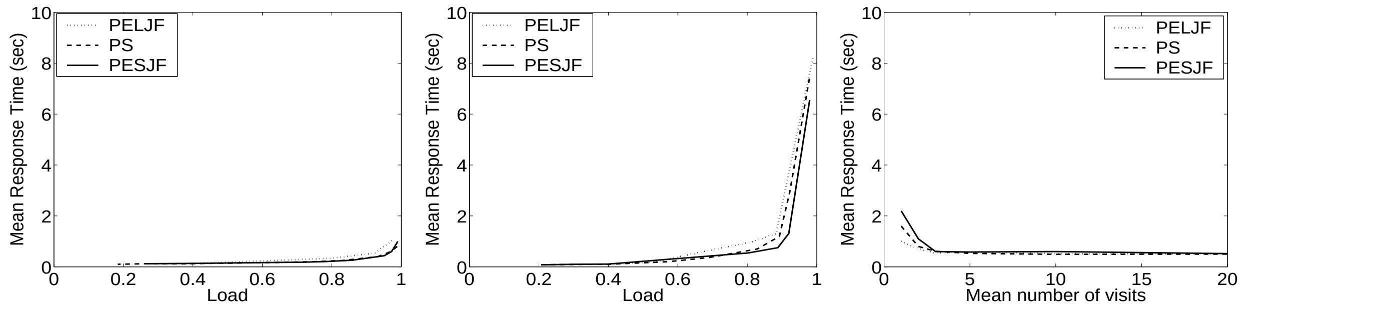
Looking more closely at the closed systems, we see that scheduling is only significant in certain regions of moderate load (Principle (v)) and only when service demands are highly variable, as in the supercomputing workload. Not visible in Figure 14.1 is the impact of the MPL, which has the effect of raising response times in the closed system, until they are near those in the open system (Principle (ii)).

For open systems, the effect of variability is much more pronounced than for closed systems (Principle (iii)). In all but the e-commerce site, we have high variability, resulting in orders of magnitude disparity between the scheduling policies (Principle (vi)), in sharp contrast to the closed system performance.

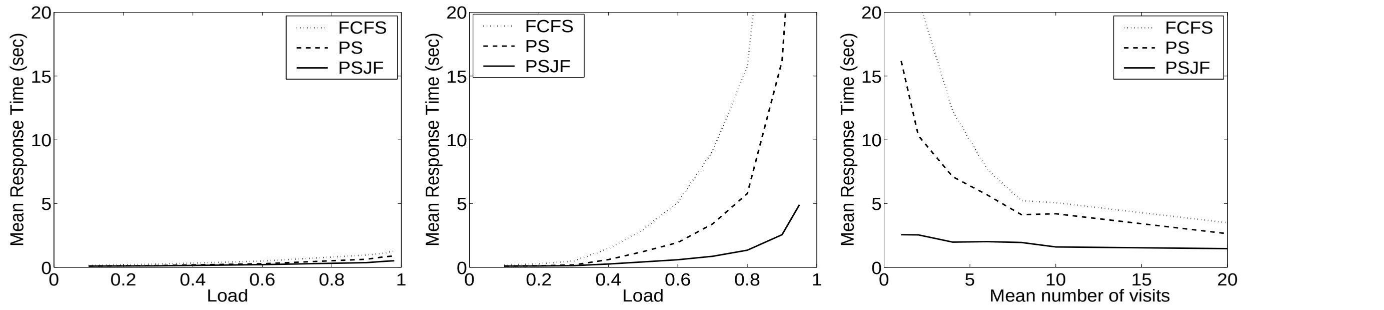
The third column of Figure 14.1 shows the results for the partly-open system. Across all applications, when the mean number of visits is small, the partly-open system behaves very much like the open system; as the mean number of visits grow, the partly-open system behaves more like a closed system (Principle (vii)). The actual convergence rate depends on the variability of the service demands (C^2). In particular, the e-commerce case study (low C^2) converges quickly, while the supercomputing case study (high C^2) converges slowly. Finally, we find, in accordance with principle (viii), that think time in the partly-open system has very little effect on response times (not shown in Figure 14.1).



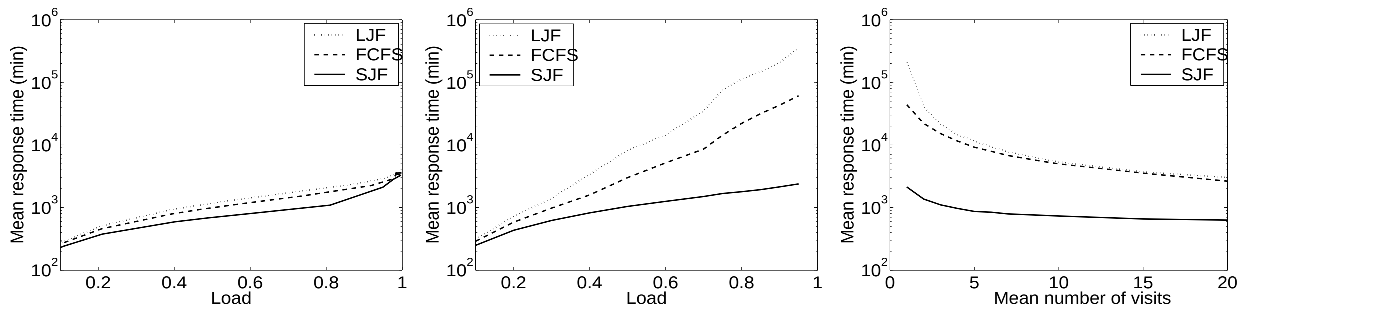
(a) Static web



(b) E-commerce site



(c) Auctioning site



(d) Supercomputing

Figure 14.1: *Results for real-world case studies. There are four rows, one for each of the real-world workloads; and three columns, one for each of the closed, open, and partly open system models. In all experiments with the closed system model the MPL is 50. The partly-open system is run at fixed load* 0.9.

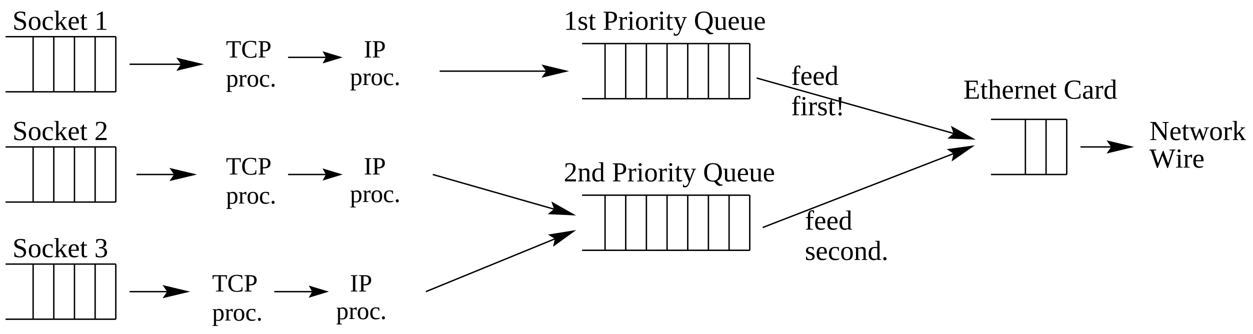


Figure 14.2: *Flow of data in Linux with SRPT-like scheduling (only 2 priority levels shown).*

Chapter 15

Conclusion and impact

The question of how to choose an appropriate system model for a particular application is a difficult one. When experimenting with a workload generator there is a tradeoff between the ease of implementation and the accuracy of the system model. In particular, the partly-open system model seems to more accurately represent web user behavior than either the open or closed models; however common workload generators are all built on either open or closed models. Thus, there is a tradeoff between building a workload generator from scratch and accurately representing the system model.

This part of the thesis provides eight simple principles that function to explain the differences in behavior of closed, open, and partly-open systems and validates these principles via trace-based simulation and real-world implementation. The more intuitive of these principles point out that response times under closed systems are typically lower than in the corresponding open system, and that as MPL increases, closed systems approach open ones. Less obviously, our principles point out that: (a) the magnitude of the difference in response times between closed and open systems can be very large, even under moderate load; (b) the convergence of closed to open as MPL grows is very slow, especially when service demand variability (C^2) is high; and (c) scheduling is far more beneficial in open systems than in closed ones. We also compare the partly-open model with the open and closed models. We illustrate the strong effect of the number of visits and C^2 on the behavior of the partly-open model, and the surprisingly weak effect of think time.

These principles underscore the importance of choosing the appropriate system model. For example, in capacity planning for an open system, choosing a workload generator based on a closed model can greatly underestimate response times and underestimate the benefits of scheduling.

All of this is particularly relevant in the context of web applications, where the arrival process at a web site is best modeled by a partly-open system. Yet, most web workload generators are either strictly open or strictly closed. Our findings provide rough *guidelines for choosing whether an open or closed model is the better approximation* based on characteristics of the workload. A high number of simultaneous users (more than 1000) suggests an open model, but a high number of visits per user (more than 10) suggests a closed model. Both these cutoffs are affected by service demand variability: highly variable demands requires larger cutoffs. Contrary to popular belief, it turns out that think times are irrelevant to the choice of an open or closed model since they only affect the load.

Once you have determined whether the closed or open model is a better approximation, that in turn provides a guideline for the effectiveness of scheduling. Scheduling is most effective in open system models, but can have moderate impact in closed models when both the load is moderate (roughly 0.7-0.85) and C^2 is high.

The principles comparing closed, open, and partly-open systems provided in this work have wide ranging impact for workload generators and capacity planning. We have illustrated through real-world case studies that the differences between response times under open and closed system models is significant. This difference can play a key role in practical situations. For instance, using a workload generator with a closed system model when in reality your system is closer to an open system may result in severely underestimating mean response time. This in turn may affect your capacity planning when you are, for example, trying to determine how fast a server you need to achieve certain response times. Currently in workload generators, emphasis is placed on the correct parameterization (e.g. setting think times); however, we have illustrated that providing flexibility and accuracy in the underlying system model is perhaps more important.

We hope that the principles we have presented can help researchers in this situation by guiding the choice of whether an open or closed model is more appropriate. Two important statistics that can guide this choice are (i) the number of visits to the server a user makes before leaving the system;

and (ii) the maximum number of users in the system at any time. If it is rare that users visit the system more than 5-10 times, an open model is likely more appropriate; whereas if the number of visits is typically more than 10-15 a closed system is likely more appropriate. Further, if we can have upwards of 1000 users in the system (and the service demands are not too highly variable) an open system is likely more appropriate; whereas if we never see (even at high load) more than 500 users, a closed system may be more appropriate.

In addition to guiding the choice of a workload generator, whether an open or closed model is more appropriate can guide system design decisions as well. The principles we have presented can be used to predict the improvements in response time that can be realized using scheduling techniques. Across both closed and open systems scheduling can provide improvement in response times; however under an open system model the benefits of scheduling are much more extreme. In a closed system model, scheduling can provide some gains in all situations, but the most significant gains happen for moderate loads (between 0.7 and 0.9).

The fundamental principles comparing closed, open, and partly-open systems provided in this work have wide-ranging impact in capacity planning, workload generation, workload characterization, queueing theory, and system design.

Capacity planning: Using a workload generator for a closed system model, when in reality your system is closer to an open system, may result in severely underestimating mean response time. This in turn may affect your capacity planning, when you are, for example, trying to determine how fast a CPU you need to achieve certain response times. This is of great concern given that much of the capacity planning literature and tools today are founded on closed system models.

Workload generation: When considering user-centric metrics like response time, we need to redesign workload generators to put less emphasis on parameters such as think time and more emphasis on providing flexibility and accuracy in the underlying system model.

Workload characterization: In characterizing workloads, we need to pay more attention to whether the users behave in a closed or open fashion. The degree of openness affects the impact system parameters have on performance, e.g. in open systems variability is crucial while in closed systems MPL is more important than variability.

Queueing theory: Queueing theory is predominantly based on open system models. We present a novel application of a complex, recently developed technique, dimensionality reduction, that allows the analysis of scheduling policies under partly-open models. Analysis of other scheduling policies in partly-open systems is an important new direction for the field.

System design: We hope that our work will alert system designers to some of the pitfalls of not being attentive to the system model underlying their workload generator. The nine principles provided herein serve as a guide to system designers on how to find scheduling policies that maximize system performance, how to configure closed and partly-open system models, and how to compare results obtained from open and closed system models.

In conclusion, while much emphasis has been placed on accurately representing workload parameters such as service demand distribution, think time, locality, etc, we have illustrated that similar attention needs to be placed on accurately representing the system itself as either closed, open, or partly-open.

References

[1] DB2 product family. `http://www-3.ibm.com/software/data/db2/`.

[2] Webjamma world wide web (www) traffic analysis tools. http://research.cs.vt.edu/chitra/webjamma.html.

[3] The PSC's Cray J90's. http://www.psc.edu/machines/cray/j90/j90.html, 1998.

[4] IBM DB2 query patroller adminsistration guide, ftp://ftp.software.ibm.com/ps/products/db2/info/vr7/pdf/letter/db2dwe70.pdf.

[5] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *Proceedings of SIGMOD*, pages 71–81, 1988.

[6] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of Very Large Database Conference*, pages 1–12, 1988.

[7] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of Very Large Database Conference*, pages 385–396, 1989.

[8] R. K. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *IEEE Real-Time Systems Symposium*, pages 113–125, 1990.

[9] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *Transactions on Database Systems*, 17(3):513–560, 1992.

[10] T. Abdelzaher and N. Bhatti. Web server qos management by adaptive content delivery. In *Proceedings of the 7th Int. Workshop on QoS (IWQoS'99)*, 1999.

[11] Tarek F. Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31(11–16):1563–1577, 1999.

[12] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[13] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *SIGCOMM '96: Con-*

ference proceedings on Applications, technologies, architectures, and protocols for computer communications, pages 293–305, New York, NY, USA, 1996. ACM Press.

[14] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Workshop On Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2002.

[15] J. Almeida and P. Cao. Wisconsin proxy benchmark 1.0. http://www.cs.wisc.edu/cao/wpb1.0.html, 1998.

[16] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[17] W. Almesberger. Linux network traffic control — implementation overview. White paper available at http://diffserv.sourceforge.net/, 1999.

[18] W. Almesberger, J. H. Salim, and A. Kuznetsov. Differentiated services on Linux. White paper available at http://lrcwww.epfl.ch/linux-diffserv/, 1999.

[19] The Apache software foundation. The Apache web server. `http://httpd.apache.org`.

[20] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[21] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997.

[22] M. Aron and P. Druschel. TCP implementation enhancements for improving webserver performance. Technical Report TR99-335, Rice University, 6, 1999.

[23] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, 2000.

[24] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of the Sigmetrics conference on Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[25] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, 2000.

[26] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web*, 2(1-2):69–83, 1999.

[27] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58, 1999.

[28] Gaurav Banga, Jeff Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[29] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[30] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *Proceedings of* ACM SIGMETRICS '01, pages 279 – 290, 2001.

[31] N. Bansal and M. Harchol-Balter. Scheduling solutions for coping with transient overload. Technical Report CMU-CS-01-134, Carnegie Mellon University, May 2001.

[32] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM Sigmetrics*, 1998.

[33] P. Barford and M. E. Crovella. A performance evaluation of hyper text transfer protocols. In *Proceedings of ACM SIGMETRICS '99*, pages 188–179, May 1999.

[34] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J.A.C.M*, 22:248–260, 1975.

[35] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[36] D. J. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html, 1997.

[37] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *TODS*, 8(4):465–483, 1983.

[38] Krishna Bharat and Andrei Broder. Mirror, mirror on the web: a study of host pairs with replicated content. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pages 1579–1590, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[39] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th International World Wide Web Conference*, 2000.

[40] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.

[41] P. Bhoj, S. Ramanathan, and S. Singhal. Web2K: Bringing qos to web servers. Technical Report HPL-2000-61, HP Laboratories, 2000.

[42] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Performance Evaluation*, 6:219–234, 1986.

[43] A. Bouch and M.A. Sasse. It ain't what you charge it's the way that you do it: A user perspective of network QoS and pricing. In *Proceedings of IM'99*, 1999.

[44] Anna Bouch, Allan Kuchinski, and Nina Bhatti. Quality is in the eye of the beholder: meeting users requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2000.

[45] L. Brakmo and L. Peterson. Performance problems in 4.4 BSD TCP. *ACM Computer Communications Review*, 25(5), 1995.

[46] Tim Brecht, David Pariag, and Louay Gammo. accept()able Strategies for Improving Web Server Performance. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, 2004.

[47] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of the Very Large Database Conference*, pages 328–341, 1993.

[48] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited.

In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 353–346, 1996.

[49] Trey Cain, Milo Martin, Tim Heil, Eric Weglarz, and Todd Bezenek. Java TPC-W implementation. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2000.

[50] L. Y. Cao and M. T. Ozsu. Evaluation of strong consistency web caching techniques. *World Wide Web*, 5(2):95–123, 2002.

[51] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, pages 193–206, December 1997.

[52] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of Very Large Database Conference*, pages 397–410, 1989.

[53] Michael J. Carey, Sanjey Krishnamurthy, and Miron Livny. Load control for locking: The 'half-and-half' approach. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1990.

[54] Jim Challenger, Paul Dantzig, Arun Iyengar, Mark Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2), 2004.

[55] Surendar Chandra and Carla Schlatter Ellis. Jpeg compression metric as a quality-aware image transcoding. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[56] Surendar Chandra, Ashish Gehani, Carla S. Ellis, and Amin Vahdat. Transcoding characteristics of web images. In *Proceedings of the SPIE Conference on Multi-Media Computing and Networking (MMCN)*, 2001.

[57] Huamin Chen and Arun Iyengar. A tiered system for serving differentiated content. *World Wide Web*, 6(4), 2003.

[58] Huamin Chen and Prasant Mohapatra. Overload control in qos-aware web servers. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 42(1):119–133, 2003.

[59] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An admission control scheme for predictable server response time for web accesses. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 545–554, 2001.

[60] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An admission control scheme for predictable server response time for web accesses. In *World Wide Web*, pages 545–554, 2001.

[61] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded web server. Available at http://www.hpl.hp.com/techreports/98/HPL-98-119.html., 1998.

[62] Ludmila Cherkasova and Peter Phaal. Predictive admission control strategy for overloaded commercial web server. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 500, 2000.

[63] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[64] Ludmila Cherkasova and Wenting Tang. Multimedia and visualization (mv): Providing resource allocation and performance isolation in a shared streaming-media hosting service. In *Proceedings of the 2004 ACM symposium on Applied computing*, 2004.

[65] Rigoberto Chinchilla, John Hoag, David Koonce, Hans Kruse, Shawn Ostermann, and Yufei Wang. The trafgen traffic generator. In *Characterization of Internet Traffic and User Classification: Foundations for the Next Generation of Network Emulation, Proc. of Int. Conf. on Telecommunication Sys., Mod. and Anal. (ICTSM10)*, 2002.

[66] A. Cockcroft. Watching your Web server. The Unix Insider at http://www.unixinsider.com, April 1996.

[67] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed Web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585–699, 1998.

[68] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling*. Addison-Wesley Publishing Company, 1967.

[69] Transaction Processing Performance Council. TPC benchmark C. Number Revision 5.1.0, December 2002.

[70] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[71] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in Web servers. In *USENIX Symposium on Internet Technologies and Systems*, October 1999.

[72] M. Crovella, Murad S. Taqqu, and A. Bestavros. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails*, pages 3–26. Chapman & Hall, New York, 1998.

[73] B. Dellart. How tolerable is delay? Consumers evaluation of internet web sites after waiting. *Journal of Interactive Marketing*, 13:41–54, 1999.

[74] D. M. Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A scalable and highly available web server. In *COMPCON*, pages 85–92, 1996.

[75] L.W. Dowdy and M.S. Chopra. On the applicability of using multiprogramming level distributions. In *Proceedings of ACM Sigmetrics*, 1985.

[76] A. B. Downey. A parallel workload model and its implications for processor allocation. In *Proceedings of High Performance Distributed Computing*, pages 112–123, August 1997.

[77] Lars Eggert and John S. Heidemann. Application-level differentiated services for web servers. *World Wide Web*, 2(3):133–142, 1999.

[78] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pages 276–286, 2004.

[79] A. Feldmann. Web performance characteristics. IETF plenary Nov.'99. http://www.research.att.com/∼anja/feldmann/papers.html.

[80] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proc. ACM Sigcomm*, 1999.

[81] Cooperative Association for Internet Data Analysis (CAIDA). Packet length distributions. http://www.caida.org/analysis/AIX/plen_hist, 1999.

[82] S. Ben Fredj, T. Bonald, A. Proutiere, G. Regnie, and J. W. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. *SIGCOMM Comput. Commun. Rev.*, 31(4):111–122, 2001.

[83] E. J. Friedman and S. G. Henderson. Fairness and efficiency in web server protocols. In *Proceedings of the 2003 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 2003.

[84] Jeffrey Fulmer. Siege. http://joedog.org/siege.

[85] Ron Lee Gary Tomlinson, Drew Major. High-capacity internet middleware: Internet caching system architectural overview. *ACM SIGMETRICS Performance Evaluation Review*, 27(4), 2000.

[86] Stephane Gigandet, Ashok Sudarsanam, and Anshu Aggarwal. The inktomi climate lab: an integrated environment for analyzing and simulating customer network traffic. In *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement*, pages 183–187. ACM Press, 2001.

[87] M. Gillmann, G. Weikum, and W. Wonner. Workflow management with service quality guarantees. In *Proceedings of the 2002 ACM SIGMOD Conference on Management of Data*, 2002.

[88] Mingwei Gong and Carey Williamson. Quantifying the properties of SRPT scheduling. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2003.

[89] Mingwei Gong and Carey Williamson. Simulation evaluation of hybrid SRPT scheduling policies. In *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 04)*, pages 355–363. IEEE Computer Society Press, 2004.

[90] A. Halikhedkar, A. Uggirala, and D. K. Tammana. Implementation of differentiated services in Linux (Diffspec). Available at http://www.rsl.ukans.edu/ dilip/845/FAGASAP.html.

[91] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[92] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[93] M. HarcholBalter, B. Schroeder, N. Bansal, and M. Agrawal. Implementation of SRPT scheduling in web servers. Technical Report CMU-CS-00-170, 2000.

[94] Poul E. Heegaard. Gensyn - generator of synthetic internet traffic. http://www.item.ntnu.no/ poulh/GenSyn/gensyn.html.

[95] H.U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *Proceedings of the Very Large Database Conference*, pages 47–54, 1991.

[96] IRCache Home. The trace files. http://www.ircache.net/Traces/, 2004.

[97] Barron C. Housel and David B. Lindquist. Webexpress: a system for optimizing web browsing in a wireless environment. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 108–116, New York, NY, USA, 1996. ACM Press.

[98] J. Huang, J.A. Stankovic, K. Ramamritham, and D. F Towsley. On using priority inheritance in real-time databases. In *IEEE Real-Time Systems Symposium*, pages 210–221, 1991.

[99] IBM DB2. Technical support knowledge base; Chapter 28: Using the governor. `http://www-3.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/document.d2w /report?fn=db2v7d0frm3toc.htm`.

[100] Red Hat Inc. Red hat content accelerator manuals. `http://www.redhat.com/docs/manuals/tux/`, 2005.

[101] ITA. The Internet traffic archives. Available at `http://town.hall.org/Archives/pub/ITA/`, 2002.

[102] A. Iyengar, E. Nahum, A. Shaikh, and R. Tewari. Enhancing web performance. In *Proceedings of the 2002 IFIP World Computer Congress (Communication Systems: State of the Art)*, 2002.

[103] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for web servers. In *Workshop on Performance and QoS of Next Generation Networks*, November 2000.

[104] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to

overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, pages 117–130, 2001.

[105] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley & Sons, 1991.

[106] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 37–48, New York, NY, USA, 2004. ACM Press.

[107] K. D. Kang, Sang H. Son, and John A. Stankovic. Service differentiation in real-time main memory databases. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 29 2002.

[108] Vikram Kanodia and Edward W. Knightly. Ensuring latency targets in multiclass web servers. *IEEE Transactions on Parallel Distributed Systems*, 14(1):84–93, 2003.

[109] K. Kant, V. Tewari, and R. Iyer. Geist: Generator of e-commerce and internet server traffic. http://kkant.ccwebhost.com/geist/.

[110] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems (HotOS) '05*, June 2005.

[111] Naoki Katoh, Toshihide Ibaraki, and Tiko Kameda. Cautious transaction schedulers with admission control. *ACM Trans. Database Syst.*, 10(2):205–229, 1985.

[112] J.S. Kaufman. Approximation methods for networks of queues with priorities. *Performance Evaluation*, 4:183–194, 1984.

[113] L. Kleinrock. *Queueing Systems*, volume I. Theory. John Wiley & Sons, 1975.

[114] L. Kleinrock. *Queueing Systems*, volume II. Computer Applications. John Wiley & Sons, 1976.

[115] A. Kraiss, F. Schoen, G. Weikum, and U. Deppisch. With heart towards response time guarantees for message-based e-services. In *VIII. Conference on Extending Database Technology (EDBT 2002)*, pages 732–735, 2002.

[116] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001.

[117] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between http/1.0 and http/1.1. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pages 1737–1751, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[118] Balachander Krishnamurthy and Craig E. Wills. Improving web performance by client characterization driven server adaptation. In *WWW '02: Proceedings of the eleventh international conference on World Wide Web*, pages 305–316, New York, NY, USA, 2002. ACM Press.

[119] Balachander Krishnamurthy, Yin Zhang, Craig E. Wills, and Kashi Vishwanath. Design, implementation, and evaluation of a client characterization driven web server. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 138–147, New York, NY, USA, 2003. ACM Press.

[120] V. Kumar, J.N. Kapur, and O. Hawaleshka. On the interrelationship between semi-open and closed queueing network models for flexible manufacturing system. *J. of. Information and Optimization Sciences*, 8:167–187, 1987.

[121] J. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley Longman, Inc., 2001, pp.319.

[122] IBM Toronto Lab. IBM DB2 universal database administration guide version 5. Document Number S10J-8157-00, 1997.

[123] James Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *Usenix Annual Technical Conference*, 2002.

[124] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. ASA-SIAM, 1999.

[125] William LeFebvre. CNN.com: Facing a world crisis. Invited talk at the USENIX Technical Conference, June 2002.

[126] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selcuk Candan, and Divyakant Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceed-*

ings of the twelfth international conference on World Wide Web, pages 587–598. ACM Press, 2003.

[127] J. Little. A proof of the theorem $L = \lambda W$. *Operations Research*, 9:383 – 387, 1961.

[128] C. Liu and P. Cao. Maintaining strong cache consistency in the world-wide web. In *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[129] Zhen Liu, Nicolas Niclausse, and Cesar Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Performance Evaluation*, 46(2-3):77–100, 2001.

[130] Zeus Technology Ltd. The Zeus web server. `http://www.zeus.com`.

[131] Chenyang Lu, Tarek F. Abdelzaher, John A. Stankovic, , and Sang H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, 2001.

[132] D. Lu, P. Dinda, Y. Qiao, H. Sheng, and F. Bustamante. Applications of SRPT scheduling with inaccurate information. In *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 04)*. IEEE Computer Society Press, 2004.

[133] Dong Lu, Huanyuan Sheng, and Peter A. Dinda. Size-based scheduling policies with inaccurate scheduling information. In *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 04)*. IEEE Computer Society Press, Jan. 2004.

[134] B. Maggs. Personal communication with Vice President of Research, Akamai Technologies, Bruce Maggs, 2001.

[135] Bruce A. Mah, Peter E. Sholander, Luis Martinez, and Lawrence Tolendino. Ipb: An internet protocol benchmark using simulated traffic. In *MASCOTS 1998*, pages 77–84, 1998.

[136] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 USITS*, 1997.

[137] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. Correctly implementing value pre-

diction in microprocessors that support multithreading or multiprocessing. In *34th Annual International Symposium on Microarchitecture (MICRO-34)*, 2001.

[138] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *20th IEEE Conference on Data Engineering (ICDE'2004)*, 2004.

[139] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the 21th IEEE Conference on Data Engineering (ICDE'2005)*, 2005.

[140] D. A. Menasce and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, Methods*. Prentice Hall, PTR, 1998.

[141] D.A. Menasce and V.A. Almeida. *Scaling for E-Business: technologies, models, performance, and capacity planning*. Prentice Hall, 2000.

[142] Microsoft. The arts and science of Web server tuning with Internet information services 5.0. Microsoft TechNet - Insights and Answers for IT Professionals: At http://www.microsoft.com/technet/, 2001.

[143] Microsoft IIS 6.0 Resource Kit Tools. Microsoft web capacity analysis tool (wcat) version 5.2.

[144] Microsoft TechNet. Ms web application stress tool (wast). http://www.microsoft.com/technet/itsolutions/intranet/ downloads/webstres.mspx.

[145] Mindcraft. The authmark benchmark. http://www.mindcraft.com/authmark/.

[146] Axel Moenkeberg and Gerhard Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *Proceedings of the Very Large Database Conference*, pages 432–443, 1992.

[147] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, January 2002.

[148] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM '95*, pages 299–313, October 1995.

[149] J. C. Mogul. Network behavior of a busy Web server and its clients. Technical Report 95/5, Digital Western Research Laboratory, October 1995.

[150] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM'97*, pages 181–194, September 1997.

[151] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance, 1998.

[152] C. D. Murta and T. P. Corlassoli. Fastest connection first: A new scheduling policy for web servers. In *Teletraffic Contributions for the Information Age, Proceedings of the 18th International Teletraffic Congress (ITC-18)*, 2003.

[153] E. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on WWW server performance. In *Proceedings of* ACM SIGMETRICS '01, pages 257–267, 2001.

[154] Erich Nahum, Tsipora Barzilai, and Dilip D. Kandlur. Performance issues in www servers. *IEEE/ACM Transactions on Networking (TON)*, 10(1), 2002.

[155] M. F. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. Johns Hopkins University Press, 1981.

[156] NISTNet. National institute of standards and technology. `http://snad.ncsl.nist.gov/itg/nistnet/`, 2002.

[157] University of Wisconsin. Shore - a high-performance, scalable, persistent object repository. `http://www.cs.wisc.edu/shore/`.

[158] R.O. Onvural and H.G. Perros. Equivalencies between open and closed queueing networks with finite buffers. *Performance Evaluation*, 9:263–269, 1998-1989.

[159] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, 2000.

[160] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of USENIX 1999*, June 1999.

[161] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified I/O buffering and caching system. In *ACM Transactions on Computer Systems*, 1997.

[162] V. Paxson and M. Allman. Computing TCP's retransmission timer. RFC 2988, `http://www.faqs.org/rfcs/rfc2988.html`, November 2000.

[163] Stefan Podlipnig and Laszlo Boszormenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[164] PostgreSQL. `http://www.postgresql.org`.

[165] P. Pradhan, R. Tewari, S. Sahu, C. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. In *Proceedings of the Int. Workshop on Quality of Service (IWQoS 2002)*, 2002.

[166] M. Rabinovich and H. Wang. Dhttp: An efficient and cache-friendly transfer protocol for web traffic. In *Proceedings of IEEE INFOCOM 2001*, 2001.

[167] Saravanan Radhakrishnan. Linux – advanced networking overview version 1. Available at http://qos.ittc.ukans.edu/howto/, 1999.

[168] Idris A. Rai, Guillaume Urvoy-Keller, and Ernst Biersack. Analysis of LAS scheduling for job size distributions with high variance. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2003.

[169] M. Rakwat and A. Kshemkayani. SWIFT: Scheduling in web servers for fast response time. In *Second IEEE International Symp. on Network Comp. and App.*, April 2003.

[170] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proc. of conf. on Arch. support for prog. lang. and operating systems*, 2000.

[171] Internet Traffic Report. http://www.internettrafficreport.com, 2004.

[172] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle Database Resource Manager: Scheduling CPU resources at the application level. 2001.

[173] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), 1997.

[174] J. Roberts and L. Massoulie. Bandwidth sharing and admission control for elastic traffic. In *ITC Specialist Seminar*, 1998.

[175] J. W. Roberts. A survey on statistical bandwidth sharing. *Comput. Networks*, 45(3):319–332, 2004.

[176] S. Ross. *Stochastic Processes.* John Wiley & Sons, 1996.

[177] A. Rousskov and Duane Wessels. High performance benchmarking with web polygraph. *Software - Practice and Experience*, 1:1–10, 2003.

[178] K.O. Salawu. Incremental delay analysis of priority scheduling in closed queueing networks. In *CMG Int. Conf. on the Management and Perf. Eval. of Comp. Sys.*, 1984.

[179] P. Schatte. On conditional busy periods in n/M/GI/1 queues. *Math. Operationsforsh. u. Statist. ser. Optimization*, 14:455–465, 1983.

[180] P. Schatte. The M/GI/1 queue as limit of closed queueing systems. *Math. Operationsforsh. u. Statist. ser. Optimization*, 15:161–165, 1984.

[181] Peter Scheuermann, Junho Shim, and Radek Vingralek. A case for delay-conscious caching of web documents. In *Selected papers from the sixth international conference on World Wide Web*, pages 997–1005, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[182] L. E. Schrage and L. W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14:670–684, 1966.

[183] S. Seshan, Hari Balakrishnan, V.N. Padmanabhan, M. Stemm, and R. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proceedings of* Conference on Computer Communications (IEEE Infocom), pages 252–262, 1998.

[184] L. Sha, R. Rajkumar, and J. Lehozky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[185] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, Sixth Edition.* John Wiley & Sons, 2002.

[186] Markus Sinnwell and Arnd C. Koenig. Managing distributed memory to meet multiclass

workload response time goals. In *Proceedings of the 15th IEEE Conference on Data Engineering (ICDE'99)*, 1997.

[187] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Replication for web hosting systems. *ACM Comput. Surv.*, 36(3):291–334, 2004.

[188] sourceforge.net. Deluge - a web site stress test tool. http://deluge.sourceforge.net/.

[189] sourceforge.net. Hammerhead 2 - web testing tool. http://hammerhead.sourceforge.net/.

[190] W. Stallings. *Operating Systems, Fourth Edition.* Prentice Hall, 2001.

[191] William Stallings. Network security essentials (2nd edition). Prentice Hall, 2002.

[192] Standard Performance Evaluation Corporation (SPEC). SFS97_R1 (3.0) benchmark. `http://www.specbench.org/osg/web99/`.

[193] Standard Performance Evaluation Corporation (SPEC). SPECJ2EE benchmark. `http://www.specbench.org/osg/web99/`.

[194] Standard Performance Evaluation Corporation (SPEC). SPECmail2001 benchmark. `http://www.specbench.org/osg/web99/`.

[195] Standard Performance Evaluation Corporation (SPEC). SPECweb99 benchmark. `http://www.specbench.org/osg/web99/`.

[196] John A. Stankovic, Sang Hyuk Son, and Jorgen Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, 1999.

[197] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. Medisyn: A synthetic streaming media service workload generator. In *Proceedings of 13th NOSSDAV*, 2003.

[198] Transaction Processing Performance Council. TPC benchmark W (web commerce). Number Revision 1.8, February 2002.

[199] G. Trent and M. Sake. WebStone: the first generation in HTTP server benchmarking. Technical report, MTS Silicon Graphics, February 1995.

[200] VeriTest. Netbench 7.0.3. http://www.etestinglabs.com/benchmarks/netbench/.

[201] VeriTest. Webbench 5.0. http://www.etestinglabs.com/benchmarks/webbench/.

[202] Akshat Verma and Sugata Ghosal. On admission control for profit maximization of networked service providers. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 128–137, New York, NY, USA, 2003. ACM Press.

[203] T. Voigt and P. Gunnigberg. Kernel-based control of persistent web server connections. *ACM SIGMETRICS Performance Evaluation Review*, 29(2):20–25, 2001.

[204] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[205] M. Vroblefski, R. Ramesh, and S. Zionts. General open and closed queueing networks with blocking: a unified framework for approximation. *INFORMS J. on Computing*, 12(4):299–316, Fall 2000.

[206] The World Wide Web Consortium (W3C). Libwww - the W3C protocol library. `http://www.w3.org`.

[207] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, 1999.

[208] Qingsong Wei, Xianliang Lu, Liyong Ren, and Xu Zhou. A novel disk queue to reduce disk i/o of messaging system. *ACM SIGOPS Operating Systems Review*, 37(3), 2003.

[209] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 2003 USENIX Symposium on Internet Technologies and Systems*, 2003.

[210] W. Whitt. Open and closed models for networks of queues. *AT&T Bell Laboratories Technical Journal*, 63:1911–1977, 1984.

[211] Ronald W. Wolff. *Stochastic Modeling and the Theory of Queues.* Prentice Hall, 1989.

[212] Shanchieh Jay Yang and Gustavo de Veciana. Enhancing both network and user performance for networks supporting best effort traffic. *IEEE/ACM Trans. Netw.*, 12(2):349–360, 2004.

[213] S.F. Yashkov. Mathematical problems in the theory of shared-processor systems. *J. of Soviet Mathematics*, 58:101–147, 1992.

[214] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture.

In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 163–174, New York, NY, USA, 1999. ACM Press.

[215] M. Yuksel, B. Sikdar, K. S. Vastola, and B. Szymanski. Workload generation for ns simulations of wide area networks and the internet. In *Proc. of Comm.. Networks and Distributed Sys. Modeling and Simulation Conference (CNDS)*, 2000.

[216] Z. L. Zhang. Large deviations and the generalized processor sharing scheduling for a two-queue system. *Queueing Systems: Theory and Applications*, 26:229–264, 1997.

[217] M. Zhou and L. Zhou. How does waiting duration information influcence customers' reactions to waiting for services. *Journal of Applied Social Psychology*, 26:1702–1717, 1996.