Using accept() Strategies to Improve Server Performance

by

David Pariag

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

David Pariag

I further authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

David Pariag

Abstract

This thesis evaluates techniques for improving the performance of three architecturally different web servers. We study strategies for effectively accepting incoming connections under conditions of high load. The experimental evaluation shows that the method used to accept new connection requests can significantly impact server performance. By modifying each server's accept strategy, we improve the performance of the kernel-mode TUX server, the multi-threaded Knot server and the event-driven μserver. Under two different workloads, we improve the throughput of these servers by as much as 10% – 39% for TUX, 0% – 32% for Knot, and 39% – 71% for the μserver. This thesis provides and in-depth look at these results, including analysis of throughput, queue drops, response times, and other server statistics. Interestingly, the performance improvements realized by the user-mode μserver allow it to obtain performance that rivals that of an unmodified TUX server.

Acknowledgements

I would like to thank my supervisor, Tim Brecht, for his guidance and support throughout my Masters. Tim has taught me much about computer systems, and research. He has made my time at the University more enjoyable, and he has been a pleasure to work with. I am also indebted to Louay Gammo for his assistance and support. I would also like to thank Peter Buhr and Martin Karsten for their valuable feedback, which has improved the quality of this thesis.[1]

I am grateful for the generous funding provided by the Natural Sciences and Engineering Research Council, the Government of Ontario, the University of Waterloo, and Tim Brecht. I would like to thank my wife for her understanding nature throughout the program. Lastly, I thank my family for their encouragement and affirmation.

[1]Portions of this thesis have been developed from a paper originally published in the USENIX 2004 Annual Technical Conference: General Track, June 2004.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

1.1 Thesis Statement

This thesis seeks to explore techniques for improving web server performance while simultaneously protecting servers from overload at saturation. To this end, we examine server strategies for accepting new connections under high loads.

1.2 Motivation

The Internet plays an increasingly important role in modern life. Today, people turn to the Internet for services like news updates, weather forecasts, current stock prices, and even driving directions. Many corporations conduct a significant amount of their business over the Internet. In fact, some businesses (like amazon.com) conduct all their retail business over the World Wide Web. The Internet has grown in scale and importance, and all signs indicate that this growth will continue into the near future.

Web servers, which handle Hypertext Transfer Protocol (HTTP) requests, are a key part of the Internet infrastructure. In fact, a recent characterization of Internet traffic shows that HTTP requests significantly outnumber all other kinds of requests combined [32]. As a result, web server performance has been extensively investigated by the research community. Some of this research (as well as anecdotal evidence) has shown that web servers often do not scale well. Denial of service attacks, news events, and even flash crowds resulting from the so-called "slashdot effect" can lead to high volumes of traffic. Such conditions can cause server performance to degrade sharply, and may eventually lead to a total loss of service. In this sense, many servers are not well-conditioned to overload. Our goal is to discover strategies that can be used to simultaneously improve the peak throughput, and overload behavior of a web server.

1.3 Contributions

This thesis examines different strategies for accepting new connections under high load conditions. We study the accept strategies used by three architecturally different web servers: the kernel-mode TUX server [30] [18], the event-driven, user-mode μserver [5] [12], and the multi-threaded, user-mode Knot server [41] [40].

We examine the connection accepting strategy used by each server, and propose modifications that permit us to tune each server's accept strategy. We implement our modifications and evaluate them experimentally using workloads that generate true overload conditions. Our experiments demonstrate that accept strategies can significantly impact server throughput, and must be considered when comparing different server architectures.

The experiments presented in this work show that:

- Under high loads, a server must accept new connections at a sufficiently high rate.
- In addition to rapidly accepting new connections, the server must actively service existing connections. Our results show that the server must *balance* the accepting of new connections with the processing of existing connections.
- We demonstrate that accept strategies can be used to significantly improve the performance of three architecturally different web servers.
- Contrary to previous findings, we demonstrate that a user-mode server is able to serve an in-memory static SPECweb99-like workload at a rate that compares very favourably with the kernel-mode TUX server.

1.4 Outline

Chapter 2 presents related work in the fields of operating systems research and server architecture. This chapter also discusses the steps that must occur before a client can establish a connection to a server. Finally, Chapter 2 introduces the notion of an "accept strategy", and discusses its relevance to server performance. In Chapter 3, we describe the architecture of each of the web servers under study. We also introduce our workloads and explain our experimental methodology. Chapter 4 presents our experimental results and analysis. Our experiments expose each server to two different workloads. We then analyze each server's throughput and response time as load increases. Chapter 5 presents a direct comparison of the μserver and TUX. Previous research and conventional wisdom have suggested that kernel-mode servers hold a wide performance advantage over their user-mode counterparts. Our work shows that this gap may not be as large as previously reported. Chapter 6 summarizes our work and presents directions for future work.

Chapter 2

Background and Related Work

Current approaches to implementing high-performance Internet servers require special techniques for dealing with high levels of concurrency. This point is illustrated by first considering the logical steps taken by a web server to handle a single client request, as shown in Figure 2.1. Of course, modern web servers must process hundreds or even thousands of connections simultaneously. This leads to high levels of concurrency within the server.

1. Wait for and accept an incoming network connection.
2. Read the incoming request from the network.
3. Parse the request.
4. For static requests, check the cache and possibly open and read the file.
5. For dynamic requests, compute the result.
6. Send the reply to the requesting client.
7. Close the network connection.

Figure 2.1: *Logical steps required to process a client request.*

Note that almost all Internet servers and services follow similar steps. For simplicity, the example in Figure 2.1 does not handle persistent connections. Several of these steps can block because of network or disk I/O, or because the web server must interact with another process. Consequently, a high performance server must be able to concurrently process partially completed connections by quickly identifying those connections that are ready to be serviced (i.e., those for which the application would not have to block). This means that high-performance servers must be able to efficiently multiplex several thousand simultaneous connections [3] and to dispatch network I/O events at high rates.

Research into improving web server performance tends to focus on improving operating system support for web servers, or on improving the server's architecture and design. We now describe related work in these areas.

2.1 Operating System Improvements

Significant research [2] [1] [3] [23] [28] [29] [6] has been conducted into improving web server performance by improving operating system mechanisms and interfaces for obtaining information about the state of socket and file descriptors. Some of these studies have developed improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data copied between the application and the kernel. Other studies have reduced the number of events delivered by the kernel, for example, the signal-per-fd scheme proposed by Chandra and Mosberger [6]. All of these studies have been motivated by the high overhead incurred by `select`, `poll`, and similar system calls under high loads. The `select` call has the following signature:

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

An fd_set is a bitmap that represents the set of available descriptors in the system. Each descriptor is represented by one bit in the bitmap. The application turns a bit "on" to declare interest in that descriptor for a particular event. The `select` system call examines three fd_sets on behalf of the application. The readfds set is examined for descriptors that are readable (without blocking). Similarly, the writefds set is examined for descriptors that are writable without blocking. Lastly, the fds represented in exceptfds will be watched for exceptions. Taken together, these three fd_sets represent the set of descriptors and events on those descriptors that the application is interested in. This is often referred to as the application's interest set.

Over the years, `select` has been heavily studied. This research has revealed (and addressed) many of `select`'s shortcomings. For example, `select` has been shown to scale poorly in WAN environments [2] where the server must handle large numbers of high-latency connections. In addition, `select` is notorious for modifying the application's interest set during each call. As a result, the application is forced to re-declare parts of its interest set before every `select` call.

Another drawback of `select` is that it performs work that depends on the size of the interest set, rather than the number of events returned. This leads to poor performance when the interest set is much larger than the active set. Although the `poll` system call does not destroy the application's interest set, it presents many of the same drawbacks as `select` including doing work that is proportional to the number of descriptors rather than the number of events returned. Having summarized the disadvantages of `select` and `poll` we now present a more in-depth examination of the research that has sought to improve these event notification mechanisms.

Early work by Banga and Mogul [2] found that despite performing well under low-latency laboratory conditions, popular event-driven servers performed poorly under real-world conditions. They demonstrated that the discrepancy is due the inability of the `select` system call to scale to the large number of simultaneous connections that are found in WAN environments. In an attempt to remedy this situation, Banga and Mogul implemented more efficient versions of the `select` system call. They also noted that the UNIX algorithm for allocating new file descriptors scaled poorly because it used a linear scan to find the lowest-numbered free descriptor. Banga and Mogul replaced the linear scan with a logarithmic-time search using a two-level tree of bitmaps. They found that the improved `select` implementation together with the faster file descriptor allocation algorithm dramatically improved server scalability and throughput.

Subsequent work by Banga et al. [3] sought to further improve on `select`'s performance by (among other things) separating the declaration of interest in events from the retrieval of events on that interest set. Event mechanisms like `select` and `poll` have traditionally combined these tasks into a single system call. However, this amalgamation requires the server to re-declare its interest set every time it wishes to retrieve events, since the kernel does not remember the interest sets from previous calls. This results in unnecessary data copying between the application and the kernel. Removing the need to repeatedly declare the interest set significantly reduces the amount of data that is passed between the application and the operating system.

To this end, Banga et al. implemented two new system calls. The `declare_interest` system call allows the application to specify which events are of interest on a particular file descriptor. Subsequently, the `get_next_event` call allows the server to retrieve events of interest for a given file descriptor. They found that this new API significantly outperformed both classical `select`, and their previous improved version [3].

The `/dev/poll` mechanism [37] was adapted from Sun Solaris to Linux by Provos et al. [28]. This mechanism improved on `poll`'s performance by introducing a new interface that separated the declaration of interest in events from the retrieval of events. Their `/dev/poll` mechanism further reduced data copying (relative to `poll`) by using a shared memory region to return events to the application.

Jonathon Lemon introduced the kqueue event mechanism [17] which addressed many of the deficiencies of `select` and `poll` for FreeBSD systems. In addition to separating the declaration of interest from the retrieval of events, `kqueue` allows an application to retrieve events from a variety of sources including file and socket descriptors, signals, AIO completions, filesystem changes, and changes in process state.

Lemon's experimental evaluation of `kqueue` used a commercial web caching proxy server and the thttpd [27] web server to compare `kqueue`, `select`, and `poll`. With respect to `select`, Lemon found that the `kqueue` system call consumed a constant amount of CPU time, regardless of the number of idle connections in the interest set. In comparison, `select`'s use of CPU time grew quite sharply as the number of idle connections increased. Lemon also found that the thttpd web server provided significantly lower response times when the `poll` event mechanism was replaced by the `kqueue` mechanism.

Linux's epoll event mechanism also separates the declaration of interest in events from their retrieval. The `epoll_create` system call instructs the kernel to create an event data structure that can be used to track events on a number of descriptors. Thereafter, the `epoll_ctl` call is used to modify interest sets, while the `epoll_wait` call is used to retrieve events. Unlike `select` and `poll`, the epoll mechanisms do not perform work that is proportional to the size of the interest set. As a result, they provide performance that is largely independent of the size of the interest set. Gammo et al. [11] evaluated the `select`, `poll`, and epoll mechanisms under representative workloads. They demonstrated that (unlike `select` and `poll`) epoll scales to large interest sets in which idle connections outnumber active connections. However, they also noted that `select` and `poll` perform comparably to epoll in the absence of idle connections.

User-mode web servers, which run as applications outside the operating system, usually invoke several system calls to process each HTTP request. Each system call entails overhead in the form of a kernel crossing (context switch). In an attempt to reduce the number of system calls (and kernel crossings) required by user-mode web servers, Rosu and Rosu [31] have implemented user-level connection tracking. This technique allows the server to track the state of its connections using a user-level API (instead of the `select` system call). The API exports the `uselect` function, which is a wrapper for the `select` system call. User-level connection tracking uses a shared memory region to exchange events between the application and the kernel. The application is allowed to access this region directly, without executing system calls. The shared memory region is also used to propogate information that is not available from `select`. For example, user-level connection tracking allows the server to determine how many bytes of data are available for reading in a socket buffer. Such information allows the server to schedule its I/O more intelligently.

Rosu and Rosu evaluated their mechanism using the Squid web proxy and a forward proxy workload biased towards cache hits and small files. They find that user-level connection tracking significantly reduces CPU overheads relative to the original Squid (with `select`). Although this mechanism was evaluated with a web proxy, it is clearly relevant to many kinds of Internet servers.

This thesis investigates accept strategies that improve the performance of architecturally different servers. Our work complements research into improved event notification mechanisms. The μserver can be configured to use several different event notification mechanisms, including `select`, `poll`, and epoll. By default, Knot uses `poll` for its event notification, although it can be configured to use epoll. Because of its kernel-mode status, TUX does not use traditional event notification mechanisms. Instead, it retrieves events by directly inspecting operating system data structures. Our work shows that a good accept strategy can reduce the overhead incurred by the server to retrieve events. More details on this will be provided in Chapter 4.

Of course, not all relevant operating systems research has focused on improving event notification mechanisms. Important work by Pai et al. [26] has focused on reducing data copying costs by providing a unified buffering and caching system (called IO-Lite) that can be used by the operating system and its applications. IO-Lite uses immutable buffers encapsulated in a buffer aggregate ADT to provide efficient data sharing between applications and/or the operating system. Pai et al. demonstrate that eliminating redundant data copying can improve the performance of a web server by 40% - 80% under realistic workloads. This thesis demonstrates techniques that can improve the performance of both highly optimized, data-aware web servers, and their less efficient counterparts. The in-kernel TUX web server provides zero-copy reads and writes through integration with the operating system. The μserver can be configured to use Linux's zero-copy `sendfile` facility for writing data, although data reads always involve copying. The Knot server cannot currently use `sendfile`, so it incurs copying overhead on both reads and writes. We now turn our attention to an examination of the different architectures that have been proposed for implementing high performance Internet servers.

2.2 Server Application Architecture

High-performance servers must efficiently transition from connections that will block to those that are ready to be serviced. One approach is to use a single process event-driven (SPED) [25] architecture. The SPED model places each socket into non-blocking mode and only issues system calls on those sockets that will not block. An event notification mechanism such as `select`, `poll`, or Linux's epoll is used to determine when a system call can be made without blocking. In multi-processor environments, multiple copies of a SPED web server can be used to obtain excellent performance [43] [33].

The multi-threaded (MT) [25] model, also known as the thread per connection model, offers an alternative approach in which each connection is associated with a thread. In this approach, connections are multiplexed by context-switching from a thread that is about to block to a thread that can continue to execute without blocking. The multi-process (MP) model is analogous to the MT model, except it uses processes rather than threads and relies on the operating system for context switching from a blocked process to one that can execute without blocking.

In the staged event driven architecture (SEDA) [42], applications consist of a network of event-driven stages connected by explicit queues. Each SEDA stage uses a thread pool to process events entering that stage. The size of the thread pool is governed by an application-specific resource controller. While SEDA does not use a separate thread for each request entering the system, concurrency still requires the use of (possibly large) thread pools.

The Flash server implements the asymmetric multi-process event driven (AMPED) [25] architecture, which combines the event-driven approach of SPED with helper processes dedicated to performing blocking disk I/O. The use of helper processes allows the main SPED process to continue execution even when one or more of the helpers is blocked. The AMPED approach attempts to work around the performance penalties incurred by SPED servers due to lack of support for asynchronous (or non-blocking) disk operations in many operating systems [25].

In light of the considerable demands placed on the operating system by web servers, some have argued that the web server should be implemented in the kernel as an operating system service. The TUX server (also knows as the Red Hat Content Accelerator) follows this paradigm. It is implemented as a Linux kernel module. TUX's in-kernel implementation provides many advantages including direct access to kernel data structures (*e.g.*, a listening socket's accept queue) without the need to make system calls like `select`. The absence of such overheads gives TUX a clear advantage when compared to user-mode servers. Indeed, recent work[15] suggests that kernel-mode servers can outperform their user-mode counterparts by a factor of three or more on static, in-memory workloads. Our findings in this thesis are considerably different as will be discussed in Chapter 5. In addition, other recent work [33] has shown that user-mode servers can outperform kernel-mode servers on workloads that contain a large proportion of dynamic requests.

Although, we recognize that each architecture has advantages and disadvantages, our research is architecture-independent. By carefully tuning accept strategies we are able to improve the performance of the SPED μserver, the multi-threaded Knot server, and the TUX server whose in-kernel architecture is similar to the AMPED model [19].

This section has discussed architectures for building high performance Internet servers. A successful server architecture must provide an efficient framework for a server to receive and process I/O events concurrently. How should the architecture manage such concurrency? The MT, MP and SEDA architectures make heavy use of threads/processes for concurrency and performance. In contrast, the SPED architecture eschews the use of threads in favour of a pure event-driven approach. Lastly, the AMPED architecture adopts a hybrid approach that makes use of a small number of processes. This range of approaches highlights an important issue in server design: What are the relative merits of using multi-threaded versus event-driven models to manage server concurrency? We now present an overview of the debate surrounding multi-threaded versus event-driven methods for managing concurrency.

2.2.1 Threads versus Events

The debate surrounding threads versus events has raged for decades. In 1978, Needham and Lauer [16] divided operating system designs into two categories. Message-oriented systems are characterized by a small, static number of message-passing processes. Procedure-oriented systems are characterized by a large, variable number of small processes along with a process synchronization mechanism. For the purposes of operating system design, Needham and Lauer argued that neither model was inherently preferable, and that they were duals.

Since then, the debate has evolved along two lines: ease-of-programming and performance. In the programmability debate, proponents of events [24] [9] have argued that threaded systems are hard to program, harder to debug, and prone to race conditions and synchronization problems. Thread proponents [40] [41] have countered that event-based programming leads to complicated control-flow and an unnatural programming style. This thesis does not contribute to the programmability debate as we believe it is a matter of personal preference. Instead, we focus on performance issues.

In the arena of high performance web servers, threads have garnered a poor reputation. Overhead due to thread scheduling, context-switching, and contention for shared locks often combine to degrade performance in threaded applications. In fact, architects of early systems found it necessary to restrict the number of concurrently running threads [13] [3].

Recent work by von Behren et al. [40] argues that many of the observed weaknesses of threads are due to poorly implemented threading libraries, and are not inherent to the threaded model. As evidence, the authors present experiments that compare the Knot and Haboob [42] web servers. Knot is a multi-threaded server, written in C, that uses the lightweight, co-operatively scheduled threads provided by the Capriccio threading library. Haboob is implemented in Java, and is based on the SEDA architecture. Each of Haboob's stages contains a thread pool and application logic responsible for part of the processing required to handle an HTTP request.

Based on an experimental comparison of Knot and Haboob, von Behren et al. conclude that multi-threaded servers can match or exceed the performance of event-driven servers. However, they also observe that Haboob context switches more than 30,000 times per second under their workloads [40]. This is partly due to the fact that a context switch is required whenever events pass from one SEDA stage to another. As a result, Haboob suffers from many of the drawbacks that plague MT servers.

This thesis does not contribute directly to the threads versus events debate. Instead, it presents strategies that can be used to improve the performance of either type of server. The μserver is event-driven, while Knot is multi-threaded. TUX is event-driven, but can use multiple threads to improve performance when multiple CPUs are available.

We now shift our focus from server architectures and concurrency issues to strategies for accepting new connections efficiently. Section 2.3 introduces the basic steps involved in establishing a connection between a client and a server. Subsequently, Section 2.4 introduces the notion of an accept strategy, and presents relevant related work.

2.3 Establishing a TCP Connection

An HTTP transaction starts with the client establishing a TCP connection to the server. Such a connection is only established if the TCP three-way handshake [36] completes successfully. The TCP three-way handshake is illustrated in Figure 2.2.

The server prepares to admit new connections by creating a socket, binding it to a

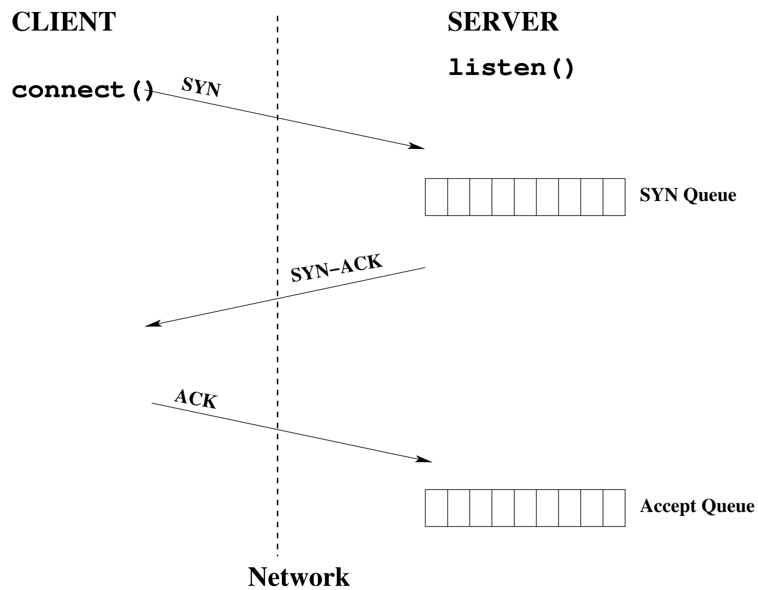

Figure 2.2: *The TCP three-way handshake*

server address, and executing the `listen` system call. We refer to this socket as the listening socket. The client creates its own socket, and initiates communication by issuing the `connect` system call. This system call causes the client operating system to send a SYN packet to the server. This packet informs the server that the client wishes to establish a connection. The server operating system places a new connection request in the listening socket's SYN queue, and acknowledges the SYN packet by replying with a SYN-ACK packet. The handshake completes when the server receives an ACK from the client in response to its SYN-ACK.

Upon receipt of the ACK packet, the server's operating system creates a new socket and adds it to the listening socket's accept queue (sometimes referred to as the listen queue). To exchange data with the client, the server must successfully complete the `accept` system call on its listening socket. Each time the server invokes the `accept` system call a socket is removed from the front of the accept queue, and an associated file descriptor is returned to the server application.

As mentioned, the server's listening socket has an associated SYN queue and an accept queue. The SYN queue is used to buffer incoming SYN packets. The accept queue holds pending connections that are waiting to be accepted. In current Linux kernels (2.4.20-8 and 2.6.0), the SYN queue is configured to hold a maximum of 1,024 SYN packets. The length of the accept queue is theoretically determined by the application when it specifies a value for the `backlog` parameter to the `listen` system call. In practice however, the Linux kernel silently limits the `backlog` parameter to a maximum of 128 connections. This behaviour has been verified by examining several Linux kernel versions (including 2.4.20-8 and 2.6.0-test7). In our work, we have intentionally left this behaviour unchanged because of the large number of installations that currently operate with this limit. We thought it was best to first try to understand how to best operate within this limit.

If either the SYN queue or the accept queue becomes full, the server kernel is forced to drop the current packet (SYN or ACK). This is commonly referred to as a queue drop. Queue drops can occur for a variety of reasons. The main reasons (but not the only reasons) are:

1. The SYN Queue is 100% full when the SYN arrives.
2. The Accept Queue is full when the SYN arrives.
3. The SYN Queue is more than 75% full when the SYN arrives.
4. The Accept Queue is full when the SYN-ACK arrives.
5. The server operating system runs out of memory.
6. The server operating system is unable to send a SYN-ACK.

Note that SYN packets may be dropped when the SYN queue is either 75% full or 100% full. If TCP_SYNCOOKIES are enabled, then SYNs are dropped when the SYN queue is completely full. Otherwise, SYNs are dropped when the SYN queue is 3/4 full. The rate at which the server accepts new connections influences both the queue drop rate (measured in queue drops per second) and the server's performance directly. If the server accepts new connections more slowly than they are arriving, the accept queue (and possibly the SYN queue) will eventually become full. When the accept queue is full, all new connection requests are dropped even if there is space in the SYN queue. Such queue drops are problematic for both the client and server. The client is unable to send requests to the server, and is forced to re-attempt the connection. Meanwhile, the server-side kernel has invested resources to complete the TCP three-way handshake, only to discover that the connection must be dropped. For these reasons, the server needs to accept new connections at a high enough rate.

However, the server could devote too much time (and resources) to admitting new connections. In the extreme case, the server would be so focused on admitting new connections that it would neglect the processing of already accepted connections. Of course, this would lead to poor performance.

Every server must choose a policy for accepting new connections. We call this strategy the server's accept strategy. This thesis investigates the impact of different accept strategies on three architecturally different web servers under two different workloads. Our goal is to demonstrate that a server's accept strategy can significantly improve its peak throughput as well as its performance under overload. A secondary goal is to determine which accept strategies work well for a given server and workload. We now present more background information on accept strategies and summarize relevant related work.

2.4 Improving Accept Strategies

In early web server implementations, the strategy for admitting new connections was to accept one new connection at a time (whenever pending connections were available). Recent work by Chandra and Mosberger [6] introduced improvements to Linux's POSIX.4 Real Time signals. The improved mechanism, which coalesces several RT signals into a single signal, ensures that at most one RT signal is delivered per descriptor. This mechanism is appropriately called *signal-per-fd*. Chandra and Mosberger found that the signal-per-fd mechanism simultaneously decreased the complexity of the server implementation, and improved its performance and robustness under high network loads.

However, they also found that a small modification to a `select`-based web-server (with a stock operating system) outperformed their operating system modifications, as well as the modifications proposed by other researchers [28]. The server modification was simple; each time the server learned that one or more new connections was pending, it would accept as many new connections as possible. In essence, the server would repeatedly call `accept` until either the call failed (and the `errno` was set to `EWOULDBLOCK`) or the limit (defined by the server or operating system) on the maximum number of open connections was reached. This heuristic meant that the server periodically drained its entire accept queue. They referred to the resulting server as a *multi-accept* server. Chandra and Mosberger's experiments demonstrate that this aggressive strategy towards accepting new connections improved event dispatch scalability for workloads that request a single one byte file or a single 6 KB file.

The work in this thesis is motivated by the above findings, which demonstrate that even simple server designs exhibit a wide range of variation in performance that is not well understood. We believe that not enough emphasis has been placed on understanding basic Internet server design. Therefore, in this thesis we consider a number of strategies for accepting new connections under high loads.

In particular, we concentrate on finding accept strategies that allow servers to accept and process more connections under conditions of high load. Note that this is quite different from simply reducing the number of queue drops (*i.e.*, failed connections) because queue drops could be minimized by only ever accepting connections and never actually processing any requests. Naturally this alone would not lead to good performance. Instead our strategies focus on enabling us to find a balance between accepting new connections and processing existing connections.

This thesis builds on Chandra and Mosberger's work by exploring a wider spectrum of accept-strategies in three architecturally different web servers under more representative workloads. We devise a simple mechanism to permit us to implement and tune a variety of accept strategies, and to experimentally evaluate the impact of different accept strategies on three server architectures. In each case we are able to show that performance can be impacted both positively and negatively by the accept strategy. In particular, we find that certain accept strategies, provide high peak throughput and well-conditioned overload behaviour.

More recent work [40] [41] has also noted that accept-strategies can significantly impact performance. Our work specifically examines different strategies used under a variety of servers in order to understand how to choose a good accept strategy. The next chapter discusses the servers, workloads, and experimental environment used in our research.

Chapter 3

Servers, Workloads and Methodology

This chapter provides important details about our experimental techniques, as well as the hardware and software used in our experiments. Section 3.1 describes each of the three servers used in our experiments. Section 3.2 describes the hardware used to conduct our experiments, and provides details on our experimental methodology including server and operating system configurations. This chapter concludes in Section 3.3 with a discussion of the two workloads used in our experiments.

3.1 The Web Servers

This section provides background information on each of the servers investigated in this thesis. We describe the architecture of each server, as well as its procedure for accepting new connections. Lastly, we describe any modifications we have made to the base server behaviour.

3.1.1 The μserver

The μserver (pronounced micro-server) [5] [12] is a single process event-driven web server. Its behaviour can be carefully controlled through the use of more than fifty command-line parameters, which allow us to investigate the effects of several different server configurations using a single web-server. In addition, the μserver accumulates several useful statistics, which are printed when it exits. These statistics are used throughout this thesis to explain the μserver's behaviour and performance.

The μserver uses either the `select`, `poll`, or `epoll` system call (chosen through command line options) in concert with non-blocking socket I/O to multiplex among concurrent connections. The server operates by tracking the state of each active connection (states roughly correspond to the steps in Figure 2.1). State information for each connection is explicitly maintained using a simple bitmask. It repeatedly loops over three phases. The first phase (which we call the getevents-phase) determines which of the connections have accrued events of interest. In our experiments this is done using `select`. In the second phase (called the accept-phase), the server checks if `select` reported that the listening socket was readable. A readable listening socket indicates that one or more connections are pending. If there are pending connections, the μserver will accept one or more of the waiting connections using the `accept` system call. If there are no pending connections, then no new connections are admitted in that accept-phase.

The third phase (called the work-phase) iterates over each of the non-listening connections which `select` indicates can be processed without blocking. For connections that are readable, the server uses the `read` system call to retrieve the HTTP GET request. If the bytes returned do not constitute a fully formed HTTP GET request, then the connection will be read again in subsequent work phases. Once a fully formed HTTP GET request is obtained, the server parses the request, and updates the connection state to indicate that the HTTP request has been received. The server also removes the socket from the read interest set, and adds it to the write interest set. This indicates to `select` that the server is no longer interested in reading from the connection, but is interested in writing the HTTP reply to the connection.

The HTTP reply consists of an HTTP header, and (usually) some HTTP content. For static requests, the server must check its document cache. If the requested file is not cached, the server must open the file and read its contents. For dynamic requests, the server must obtain a computed result. When the HTTP header and any associated content is ready, the server must send the reply to the client. In our experiments, which use a static workload, the μserver uses the `write` system call to send the HTTP header, and the `sendfile` system call to send the file contents.

Note that the server will only write data to the connection when `select` indicates that the socket is writable. Thus, the reading of the request and the writing of the reply usually occur in separate work phases. If either `write` or `sendfile` is unable to write its portion of the reply in one call, the remainder of the reply will be written in subsequent work phases. If the request is HTTP 1.0, then the connection is closed once the reply has been written. For HTTP 1.1 requests, the server will continue to read requests until the client closes the connection. The three phases of μserver operation are illustrated in Figure 3.1.

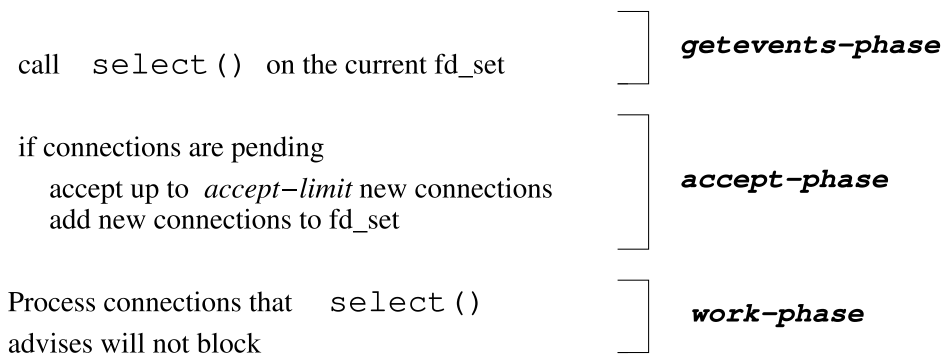

Figure 3.1: *Phases of operation in the μserver*

A key point is that for the μserver options used in our experiments the work-phase does not consider any of the new connections accumulated in the immediately preceding accept-phase. Instead, the μserver only performs those network operations which `select` has indicated can be completed without blocking. In this respect, the μserver differs from the other servers in our comparison, which attempt to completely process existing connections before accepting new ones.

The μserver is based on the multi-accept server written by Chandra and Mosberger [6]. That server implements an accept policy that drains its accept queue when it is notified of a pending connection request. In contrast, the μserver uses a parameter that permits us to accept up to a pre-defined number of the currently pending connections. This defines an upper limit on the number of connections accepted consecutively. For ease of reference, we call this parameter the accept-limit parameter, and refer to it throughout the rest of this thesis (the same name is also used in referring to modifications we make to the other servers we examine). Parameter values range from one to infinity (*Inf*). A value of one forces the server to accept a single connection, while Inf causes the server to accept all currently pending connections.

Early investigations [5] revealed that the accept-limit parameter could significantly impact the μserver's performance. This motivated us to explore the possibility of improving the performance of other servers, as well as quantifying the performance gains under more representative workloads. As a result, we have implemented accept-limit mechanisms in two other well-known web servers. We now describe these servers and mechanisms.

3.1.2 Knot

Knot [40] is a multi-threaded web server which makes use of the Capriccio [41] threading package. Knot is a simple web server. It derives many benefits from the Capriccio threading package, which provides lightweight, cooperatively scheduled, user-level threads. Capriccio also features several different thread schedulers. The resource-aware scheduler attempts to intelligently schedule threads by tracking their CPU, memory, and file-descriptor usage patterns [41]. The graph-batch scheduler uses Capriccio's blocking graph abstraction to locate blocking points in a thread's execution path. This scheduler tries to resume all runnable threads blocked at a particular blocking point before moving on to the next blocking point. Lastly, the round-robin scheduler simply schedules threads in FIFO order. Our preliminary investigations revealed that the round-robin scheduler yielded the best performance under our workloads. As such, Knot was configured to use the round-robin scheduler in all our experiments.

Knot operates in one of two modes [40]. Both modes use a separate user-level thread to process each connection. This model is often called the thread-per-connection model. Note that the number of concurrent connections is governed by the number of active threads in this model. Knot achieves concurrency by running hundreds or even thousands of threads concurrently. The main difference between Knot's modes, which are referred to as Knot-A and Knot-C, is the manner in which the number of threads is controlled.

Knot-C allows the user to fix the number of threads used at runtime (via a command-line parameter). Threads are pre-forked during initialization. Thereafter, each thread executes a loop in which it accepts a single connection and processes it to completion. Knot-A creates a single acceptor thread which loops attempting to accept new connections. For each connection that is accepted, a new worker thread is created to completely process that connection. As such, the number of threads used by Knot-A is not fixed at runtime.

Knot-C is meant to favour the processing of existing connections over the accepting of new connections, while Knot-A is designed to favour the accepting of new connections. By having a fixed number of threads, and using one thread per connection, Knot-C contains a built-in mechanism for limiting the number of concurrent connections in the server. In contrast, Knot-A allows increased concurrency by placing no limit on the number of concurrent threads or connections.

Our preliminary experiments revealed that Knot-C performs significantly better than Knot-A, especially under overload where the number of threads (and open connections) in Knot-A becomes very large. Our comparison agrees with findings by the authors of Knot [40], and as a result we focus our studies on Knot-C.

We modified Knot-C to allow each of its threads to accept multiple connections before processing any of the new connections. This was done by implementing a new function that is a modified version of the `accept` call in the Capriccio library. This new call loops to accept up to accept-limit new connections provided that they can be accepted without blocking. If the call to `accept` would block and at least one connection has been accepted the call returns and the processing of these accepted connections proceeds. Otherwise the thread is put to sleep until a new connection request arrives. After accepting new connections, each thread fully processes all of the accepted connections before admitting any more new connections.

Therefore, in our modified version of Knot each thread oscillates between an accept-phase and a work-phase. There is no getevents-phase in the Knot server. During its work-phase, Knot uses the `read` system call in a loop to read HTTP requests. Similarly, Knot loops calling the `write` system call to write replies. However, the system calls invoked by Knot are not traditional Linux system calls. Instead, the underlying threading package (Capriccio) associates each socket descriptor with a thread, and uses `poll` to identify runnable threads. Capriccio provides system call wrappers, which threads invoke in place of customary system calls. When a thread executes a system call wrapper, Capriccio ensures that the associated socket is in non-blocking mode and attempts to execute the system call. If the call would block, Capriccio adds the socket descriptor to the `poll` interest set and switches to another thread. Later, when `poll` indicates that system call will no longer block, the call is completed, and the original thread is resumed.

As with the μserver, the accept-limit parameter ranges from 1 to infinity. With an accept-limit of 1, our version of Knot behaves identically to an unmodified version of Knot. The rest of this thesis uses the accept-limit parameter to explore the performance of our modified version of Knot-C.

3.1.3 TUX

TUX [30] [18] (which is also referred to as the Red Hat Content Accelerator) is an event-driven, kernel-mode web server for Linux developed by Red Hat. It is compiled as a kernel-loadable module (similar to many Linux device drivers), which can be loaded and unloaded on demand. TUX's kernel-mode status affords it many I/O advantages including true zero-copy disk reads, zero-copy network writes, and zero-copy request parsing. In addition, TUX accesses kernel data structures (e.g., the listening socket's accept queue) directly, which allows it to obtain events of interest with very low overhead compared to user-level mechanisms like `select`. Lastly, TUX avoids the overhead of kernel crossings that user-mode servers must incur when making system calls. This optimization is important in light of the large number of system calls needed to process a single HTTP request.

An examination of the TUX source code provides detailed insight into TUX's structure. TUX's processing revolves around two queues, which are illustrated in Figure 3.2. The first queue is the listening socket's accept queue. The second is the `work_pending` queue which contains items of work (*e.g.*, reads and writes) that are ready to be processed without blocking. TUX's main loop performs an accept-phase followed by a work-phase. In this way, TUX alternately processes events from each of these queues in a tight loop.

TUX does not require a getevents-phase because it has access to the kernel data structures where event information is available. In the accept-phase, the default version of TUX enters a loop in which it accepts all pending connections (thus draining its accept queue). In the work-phase, TUX processes all items in the `work_pending` queue before starting the next accept-phase. Note that new items may be enqueued on either queue even as TUX dequeues and processes items from that queue.

We modified TUX to include an accept-limit parameter, which governs the number of

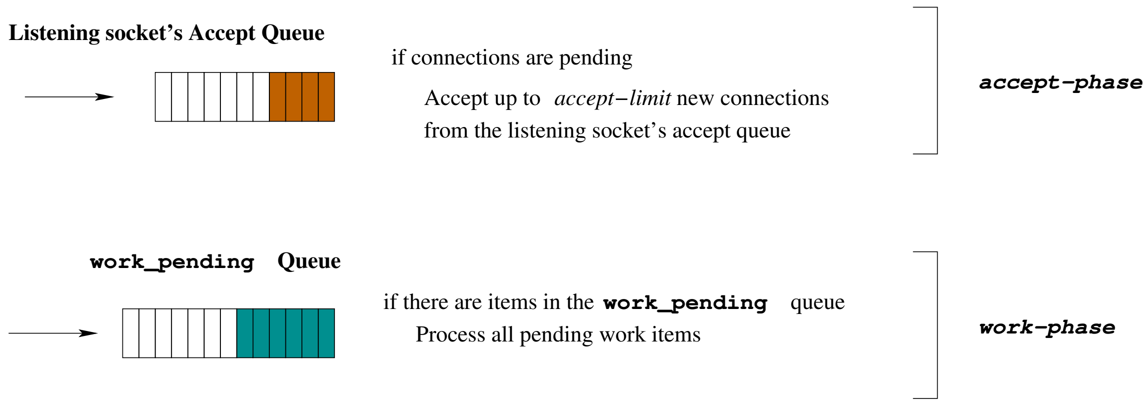

Figure 3.2: *Phases of operation in TUX*

connections that TUX will accept consecutively. Since TUX is a kernel-loadable module, it does not accept traditional command line parameters. Instead, the new parameter was added to the Linux `/proc` filesystem, in the `/proc/sys/net/tux` subdirectory. The `/proc` mechanism is convenient in that it allows the new parameter to be read and written without restarting TUX. The new parameter provides a measure of control over TUX's accept policy, and allows us to compare different accept-limit values with the default policy of accepting all pending connections.

Note that there is an important difference between how the μserver and TUX operate. In the μserver the work-phase processes a fixed number of connections (determined by `select`). In contrast TUX's `work_pending` queue can grow during processing, which prolongs its work phase. As a result we find that the accept-limit parameter impacts these two servers in dramatically different ways. This will be seen and discussed in more detail in Chapter 4.

It is also important to understand that the accept-limit parameter does not control the accept rate directly, but merely influences it. The accept rate is determined by a combination of the frequency with which the server enters the accept-phase and the number of connections accepted while in that phase. The amount of time spent in the other phases determines the frequency with which the accept-phase is entered. This is discussed in greater detail in Chapter 4.

3.2 Experimental Environment

Our experimental environment is made up of two separate client-server clusters. The first cluster (Cluster 1) contains a single server and eight clients. The server contains two Xeon processors running at 2.4 GHz, 1 GB of RAM, a 10,000 RPM SCSI disk, and two Intel e1000 Gigabit Ethernet cards. The clients are identical to the server with the exception of their disks which are EIDE. The server and clients are connected with a 24-port Gigabit switch. Since the server has two network cards, we avoid network bottlenecks by partitioning the clients into different subnets. In particular, the first four clients communicate with the IP address associated with the server's first Ethernet card, while the remaining four communicate using a different IP address linked to the second Ethernet card.

Each client runs Red Hat 9.0 which uses the 2.4.20-8 Linux kernel. The server also uses the 2.4.20-8 kernel, but not the binary that is distributed by Red Hat. Instead, the Red Hat sources were re-compiled after we incorporated our changes to TUX. The resulting kernel was used for all experiments on this machine. The aforementioned kernel is a uni-processor kernel that does not provide SMP support. The reasons for this are twofold. Firstly, the Capriccio threading package does not currently include SMP support. Secondly, we find it instructive to study the simpler single-processor problem, before considering complex SMP interactions.

The second machine cluster (Cluster 2) also consists of a single server and eight clients. The server contains two Xeon processors running at 2.4 GHz, 4 GB of RAM, several high-speed SCSI drives and two Intel e1000 Gigabit Ethernet cards. The clients are dual-processor Pentium III machines running at 550 MHz. Each client has 256 MB of RAM, a SCSI disk, and one Intel e1000 Gigabit Ethernet card. The server runs a Linux 2.4.19 uni-processor kernel, while the clients use the 2.4.7-10 kernel that ships with Red Hat 7.1.

This cluster of machines is networked using a separate 24-port Gigabit switch. Like the first cluster, the clients are divided into two groups of four with each group communicating with a different server NIC. In addition to the Gigabit subnets, all machines are also connected to a separate 100 Mbps network which is used for co-ordinating experiments. Each cluster is completely isolated from other network traffic.

Cluster 1 is used to run all μserver and TUX experiments while Cluster 2 is used to run all Knot experiments. Because our clusters are slightly different, we do not directly compare results taken from different clusters. Ideally, we would use one cluster for all our experiments, but the sheer number of experiments required necessitates the use of two clusters.

3.2.1 Web Server Configuration

In the interest of making fair and scientific comparisons, we carefully configured TUX and the μserver to use the same resource limits. TUX was configured to use a single kernel thread. This enables comparisons with the single process μserver, and was also recommended in the TUX user manual [30]. The TUX accept queue backlog was set to 128 (via the `/proc/sys/net/tux/max_backlog` parameter) which matches the value imposed on the user-mode servers. By default, TUX bypasses the kernel-imposed limit on the length of the accept queue, in favour of a much larger backlog (2,048 pending connections).

Additionally, both TUX and the μserver allow a maximum of 15,000 simultaneous connections. In the μserver case this was done by using an appropriately large FD_SETSIZE. For TUX this was done through `/proc/sys/net/tux/max_connections`. All μserver and TUX experiments were conducted using the same kernel.

The Knot server was configured to use the Knot-C behaviour. That is, it pre-forks and uses a pre-specified number of threads. In our experiments, Knot is configured to use 1,000 threads. We have spent a modest amount of time investigating Knot's parameter space, and have found that Knot performs best on our workloads with 1,000 threads and the round-robin scheduler. Under Knot's thread per connection architecture, the number of server threads is an upper bound on the number of simultaneous connections. However, the accept-limit modification allows Knot to maintain several active connections per thread, thus raising the upper bound.

Logging is disabled on all servers to avoid unnecessary performance perturbations. Lastly, we ensure that all servers can cache the entire file set. This ensures that differences in server performance are not due to differences in caching strategies or cache hit rates.

3.3 Workloads

This section describes the two workloads that we use to evaluate server performance. We emphasize that each workload is able to generate overload conditions. We note that these two workloads are quite different and will stress different aspects of the server and operating system. In particular, we note that average transfer sizes and the demand for new connections varies quite sharply across these two workloads.

3.3.1 SPECweb99-like workload

The SPECweb99 benchmarking suite [35] is a widely accepted tool for evaluating web server performance. It was developed by the Standard Performance Evaluation Corporation (SPEC), based on their analysis of server logs taken from several popular Internet servers, and some smaller web sites[8]. In spite of its careful design and widespread use, the benchmark does suffer from notable shortcomings. One of the most important problems is that the SPECweb99 suite is unable to generate overload conditions.

The problem arises because the SPECweb99 HTTP load generator operates in a *closed-loop*. This means that the load generator will only send a new request once the server has replied to its previous request. Banga et al. [4] show that in this naive load generation scheme, the client's request rate is throttled by the speed of the server. As such, it is impossible for a small number of closed-loop load generators to overload a web server. In fact, Banga et al. demonstrate that several thousand closed-loop generators would be needed to generate overload conditions. Clearly, this requirement is unreasonable for modest research efforts.

We address this problem by using httperf, [21], an *open-loop* load generator that is capable of generating overload conditions. We note that httperf avoids the naive load generation scheme by implementing aggressive connection timeouts. Every time a connection to the server is initiated, a timer is started. If the timer expires before the connection is established and the HTTP transaction completes, the connection is aborted and retried. This strategy ensures that the server is sent a continuous stream of requests that is independent of the server's reply rate. In addition, only a single copy of httperf is needed (per client CPU) to generate sustainable overload.

We use httperf in conjunction with SPECweb99 file sets and HTTP traces that have been carefully constructed to mimic the SPECweb99 workload. Our traces, though synthetic, accurately recreate the file classes, access patterns, and number of requests issued per (HTTP 1.1) connection that are used in the static portion of SPECweb99. When examining results obtained with this workload, the reader may wish to remember that each HTTP 1.1 connection is used to request an average of 7.2 files. Thus, this workload creates less demand for new connections than a workload that requests fewer files per connection. As a result, we expect performance under this workload to be less sensitive to changes in accept strategy, especially when compared to workloads that request fewer files per connection. As we will see in Section 3.3.2, the SPECweb99 workload may significantly overestimate the number of requests per connection seen on the World Wide Web. The mean file size under the SPECweb99-like workload is approximately 15 KB.

We examine server performance using a SPECweb99 file set containing 36 files, and occupying just over 5 MB of disk space. Obviously, the entire file set can be easily cached, and the server does little or no disk I/O once the cache has been populated. This avoids any distortions in server performance due to disk latency. Clearly, a 5 MB file set is not representative of the server file sets found in production environments. However, the salient point is that our file set fits entirely in main memory. We claim that our strategy is roughly equivalent (from a performance point of view) to using a much larger file set (*e.g.*, 256 MB) that is also completely cacheable. There may be differences in performance because of the processor's ability to cache a larger portion of the 5 MB file set. However, we do not expect these differences to change the qualitative results. The principal advantage of using a smaller file set is that it is fully cached in only a few seconds. This in turn reduces the length of each experiment. A larger file set would require a longer "warm up" period before the entire file set is cached.

3.3.2 One-packet workload

Our second workload is motivated by real-world events. During the September 11th terrorist attacks, CNN.com was subjected to crippling overload [7]. The staff at CNN.com responded by replacing their main page with a small, text-only page containing the latest headlines. The new page was sized so the server reply would fit in a single TCP/IP packet.

With this in mind, we devised a static workload that simulates the load experienced by CNN.com on September 11th. The workload is simple; all requests are for the same file, which is sized so that the HTTP headers and file contents fill a single packet. Each request is sent over a new HTTP 1.1 connection.

We believe this simple workload tests many aspects of server performance that are neglected by SPECweb99-based workloads. Nahum [22] analyzes the characteristics of the SPECweb99 workload in light of data gathered from several real-world web server logs. His analysis reveals many important shortcomings of the SPECweb99 benchmark. For example, the SPECweb99 benchmark does not use conditional GET requests. With conditional GETS, if the requested file has not been modified since the client's last request, the server returns a header containing `HTTP 304 Not Modifed` and zero bytes of file data. Interestingly, such requests accounted for up to 28% of all requests in some server traces. With the transmission of only an HTTP header, the server response is quite small, and easily fits in a single packet.

Nahum also reports significantly greater use (51% – 95%) of HTTP 1.0 than the 30% used by SPECweb99. He also reports that SPECweb99 significantly overestimates average transfer sizes. SPECweb99's median transfer size of 5,120 bytes is an order of magnitude larger than the transfer sizes captured in the sample logs in his study. In fact, the median transfer size in one of Nahum's popular logs was a mere 230 bytes! The combinations of these observations indicates that the demand for new connections at web servers is likely to be much higher than the demand generated by a SPECweb99-like workload.

Further evidence for this conclusion is provided in recent work by Jamjoom et al. [14]. They report that in an attempt to minimize user response time, many popular browsers (on Linux and Windows 2000) tend to issue multiple requests for embedded objects in parallel. This is in contrast to using a single sequential persistent connection to request multiple objects from the same server. They report that although there were on average 21 unique embedded objects per page visited, the average requests per connection issued by the different browsers examined is between 1.2 and 2.7. This is considerably lower than the average of 7.2 requests per connection used by SPECweb99, and reinforces the conclusion that SPECweb99 workloads underestimate the demand for new connections seen at many web servers.

While a SPECweb99-like workload is still useful for measuring web server performance, it is deficient in many respects and should not be used as the sole measure of server performance. Our one-packet workload highlights a number of phenomena reported in recent literature (small transfer sizes, a small number of requests per connection). More importantly, as implemented by CNN.com, this is perhaps the best way to serve the most clients under conditions of extreme overload. For the purposes of our study, it is useful because it places high demands on the server to accept new connections.

The next chapter presents and analyzes our experimental results. For each combination of server and workload we present graphs showing throughput, server latency, and server queue drops. In many cases, additional graphs and tables are included to aid analysis.

Chapter 4

Experimental Results

This chapter presents the results of experiments that subject each of the three servers to two different workloads. For each experiment, we present graphs showing throughput, response times, and queue drops. In our graphs, each data point is the result of a two minute experiment. Trial and error revealed that two minutes provided sufficient time for each server to achieve steady state execution. Longer durations did not alter the measured results, and only served to prolong experimental runs.

A two minute delay was used between consecutive experiments. This allowed all TCP sockets to clear the TIME_WAIT state before commencing the next experiment. Prior to running experiments, all non-essential Linux services (*e.g.*, sendmail, dhcpd, cron etc.) are shutdown. This eliminated interference from daemons and periodic processes (*e.g.*, cron jobs) which might confound results.

Prior to determining which accept-limit values to include in each graph a number of alternatives were run and examined. The final values presented in each graph were chosen in order to highlight the most interesting accept policies. The following sections present and analyze our experimental results. The results obtained under the SPECweb99-like workload are presented first, followed by the results obtained under the one-packet workload.

4.1 In-Memory SPECweb99-like workload

4.1.1 μserver Performance

Figure 4.1 examines the performance of the μserver as the accept-limit parameter is varied. Recall that the accept-limit parameter controls the number of connections that are accepted consecutively. The line labeled Accept-1 traces the performance of the μserver under the Accept-1 policy, where the μserver tries to accept a single connection in each accept-phase. Similarly, the Accept-10 policy causes the server to accept at most 10 connections in each accept-phase. Lastly, the Accept-Inf policy causes the server to drain its accept queue, meaning that it accepts all pending connections.

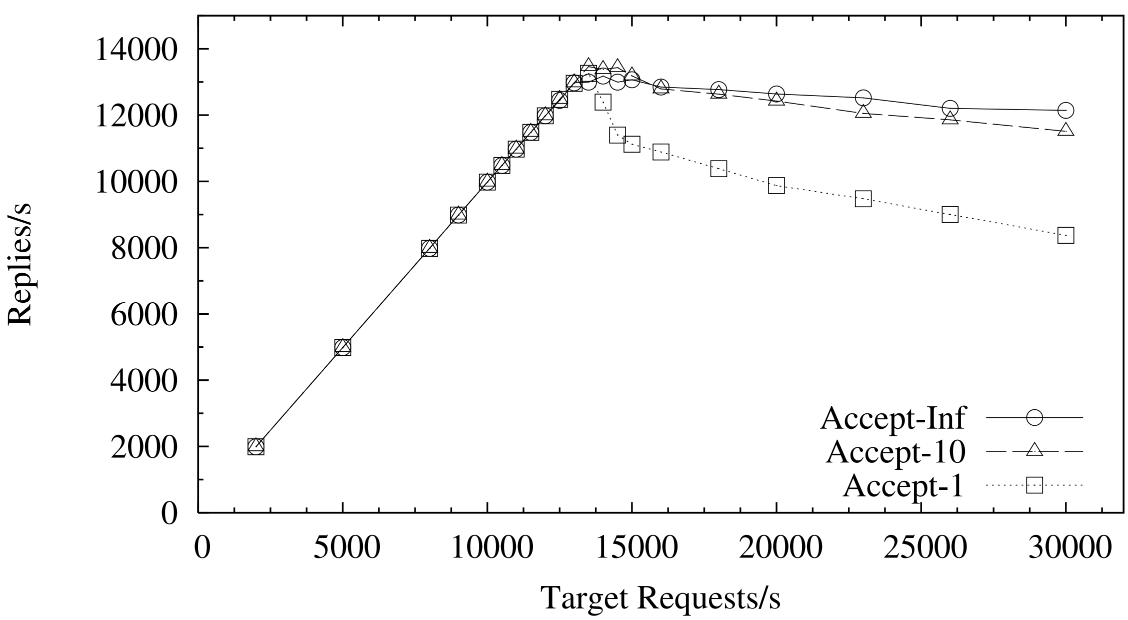

Figure 4.1: *μserver throughput under the SPECweb99-like workload*

Interestingly, all three policies obtain similar peak throughput. However, the Accept-10 and Accept-Inf policies fare best under overload conditions, with the Accept-Inf policy posting a slight performance advantage. As Figure 4.1 shows, the Accept-1 policy degrades under overload. At a target load of 23,000 requests/sec the Accept-Inf policy outperforms the Accept-1 policy by 32%. At the extreme target load of 30,000 requests/sec, the gap grows to 39%. This graph illustrates that a larger accept-limit can significantly improve throughput in the μserver under overload conditions.

Statistics collected by the μserver provide insight that confirms the benefits of the high accept-limit value after overload. At a target load of 30,000 requests/sec, the Accept-Inf server accepts an average of 1,571 new connections per second In comparison, the Accept-1 server averages only 1,127 new connections per second (39% fewer). This difference is especially significant when we consider that each SPECweb99 connection is used to send an average of 7.2 requests.

The difference in performance is partly explained by examining queue drop rates at the server. The queue drop rates are obtained by running `netstat` on the server before and after each experiment. The number of failed TCP connection attempts and listen queue overflows are summed and recorded before and after each experiment. Subtracting these values and dividing by the experiment's duration provides a rate, which we report in our queue drop graphs.

Figure 4.2 shows that in most cases the more aggressive accept policy leads to lower queue drop rates (QDrops/s). The lower drop rates indicate that the μserver is admitting new connections faster under the more aggressive accept policies. This allows the server to efficiently amortize the overhead of the `select` system call.

Figure 4.3 shows the number of `select` calls made per second by the μserver as the target load is varied. Under light loads (*e.g.*, 5,000 - 8,000 requests/sec) all three accept policies call `select` at a rapid rate. This behaviour is expected, because under light loads the server spends a lot of its time polling for new work. This means the server calls `select` often, hoping to receive notification of pending work. As the server approaches peak throughput, the number of open connections increases, and the server spends less time polling and more time servicing open connections. As a result, the `select` rate falls dramatically. At request rates of 15,000 requests/sec or higher the `select` rates for all three accept policies have stabilized. However, there is a marked difference in the rate at which the Accept-1 policy calls `select`.

As Figure 4.3 shows, the Accept-1 policy calls `select` at a much higher rate than either of the two other policies. At 26,000 requests/sec, the Accept-1 policy averages 1,248 `select` calls per second. In contrast, the Accept-10 policy makes 177 calls/sec, and the Accept-1

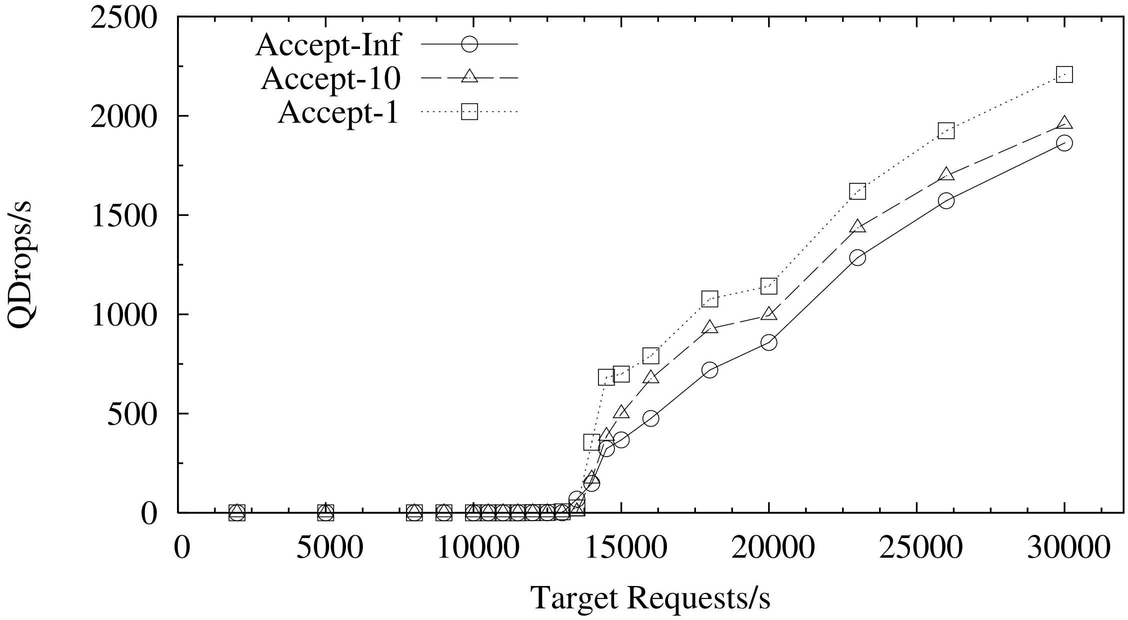

Figure 4.2: *µserver queue drops under the SPECweb99-like workload*

policy makes a mere 35 calls/sec. The additional `select` calls do not improve performance for the Accept-1 policy, which suffers 28% lower throughput than the Accept-Inf policy. In fact, it will soon be clear that the extra `select` calls contribute to the lowered throughput.

Figure 4.4 shows the average number of fds returned by `select` as the target load is varied. At low request rates, `select` returns notification for very few file descriptors, regardless of the accept-policy.

This is expected since the server is underutilized and is actively polling for work. As request rates approach 15,000 requests/sec, the server approaches its peak throughput, and `select` returns more fds for the Accept-Inf and Accept-10 policies. The Accept-Inf policy in particular shows dramatic growth in the number of fds returned by `select`. At 26,000 requests/sec, `select` returns an average of 885.8 fds under the Accept-Inf policy compared to 21.6 and 171.3 for the Accept-1 and Accept-10 policies, respectively. This data is summarized in Table 4.1, which presents `select` statistics for the µserver. All the data in Table 4.1 is based on a target load of 26,000 reqs/sec under the SPECweb99-like workload.

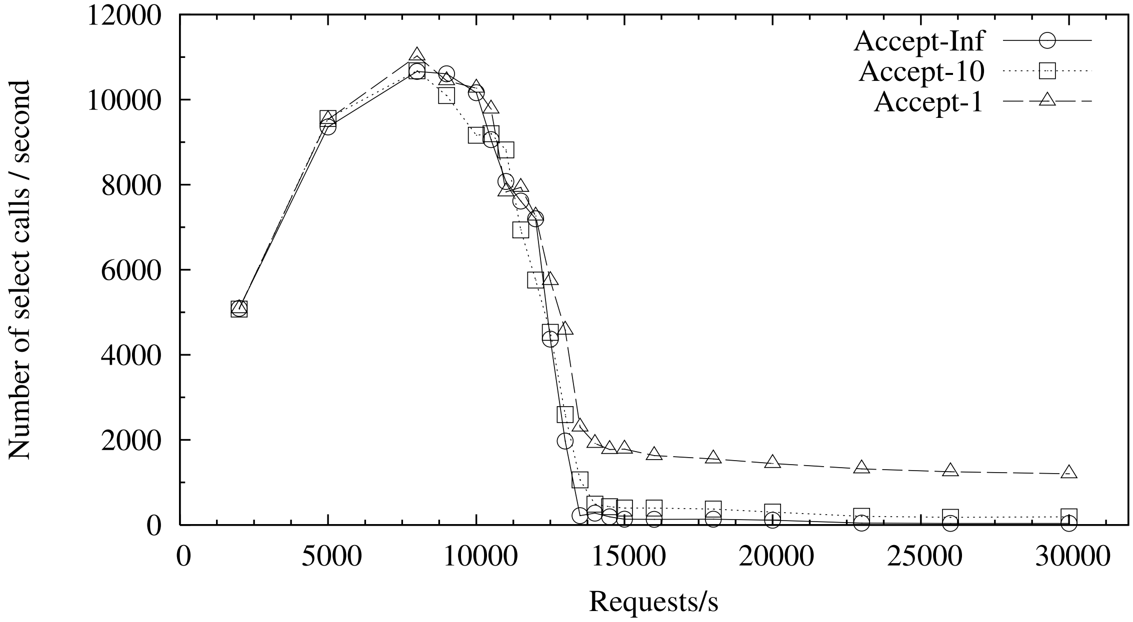

Figure 4.3: *Number of* `select` *calls/sec made by the µserver under the SPECweb99-like workload*

	select-rate	**Avg fds from select**	**fd-rate**	**Throughput**
Accept-Inf	35	885.8	31,003	12,249
Accept-10	177	171.3	30,320	11,470
Accept-1	1,248	21.6	26,957	8,888

Table 4.1: `select` *statistics for µserver @ 26,000 reqs/sec under the SPECweb99-like workload*

The select-rate column shows the number of `select` calls made per second under each accept policy. The Avg fds from select column lists the average number of file descriptors returned by a `select` call under each policy. The fd-rate column is simply the product of the select-rate and Avg fds from select columns, and gives the rate (in fds per second) at which fds are returned from `select` to the application. The rightmost column lists the server throughput (in replies/sec) for each accept policy.

An analysis of the data for the Accept-1 and Accept-Inf policies provides valuable in-

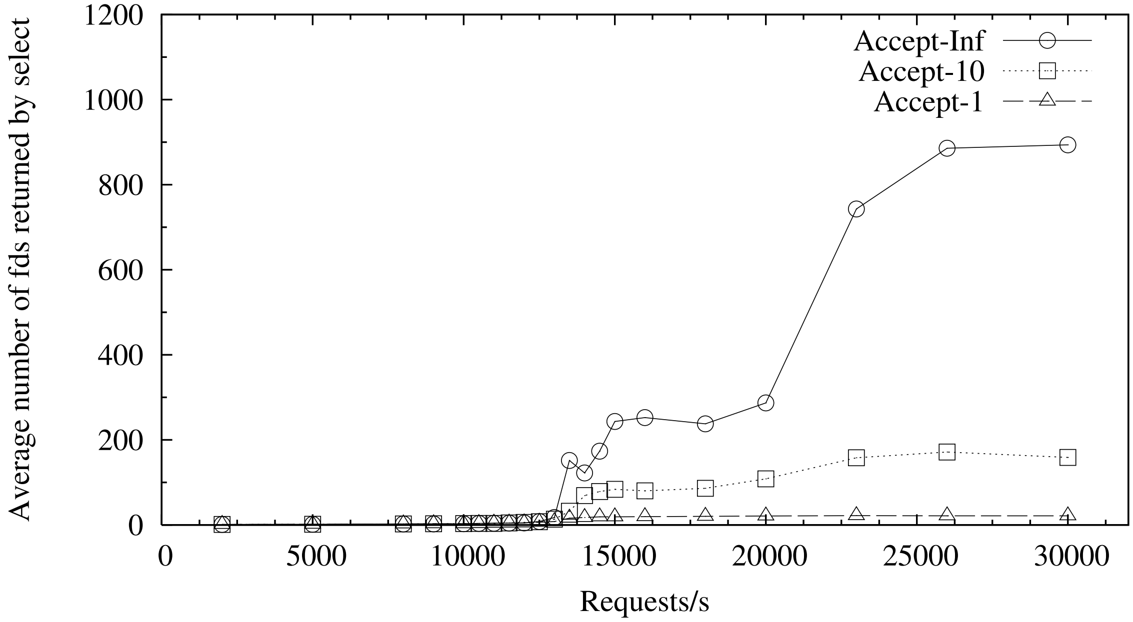

Figure 4.4: *Average number of fds returned by* `select` *in the μserver under the SPECweb99-like workload*

sight. The Accept-1 policy makes approximately 35 times more `select` calls, but receives 41 times fewer fds per call than the Accept-Inf policy. In the end, despite making more calls to `select` the Accept-1 policy receives fd-notification at a lower rate than the Accept-Inf policy. Clearly, this represents a poor amortization of `select` overhead. The higher select-rate represents unnecessary overhead that degrades performance. Because the Accept-Inf policy is able to more efficiently amortize its `select` overhead, it achieves significantly higher throughput. The key to efficient amortization lies in the rate at which new connections are accepted under the Accept-Inf policy. Table 4.2 presents μserver statistics related to the `accept` system call. These statistics were gathered while the server was subjected to a target load of 26,000 reqs/sec.

The leftmost column shows the number of accept-phases the server completed during the experiment. The second column, C_{avg}, shows the average number of successful `accept` calls that were completed in each accept-phase. A successful `accept` call is one that returns a new file descriptor, as opposed to returning `EWOULDBLOCK`. C_{avg} is computed

	N_{phases}	C_{avg}	Total Accepts	Accept Rate	Throughput
Accept-Inf	5,202	47.30	246,063	1,670	12,249
Accept-10	26,275	8.81	231,395	1,570	11,470
Accept-1	184,323	0.97	178,061	1,208	8,888

Table 4.2: *Accept-phase statistics for µserver @ 26,000 reqs/sec under the SPECweb99-like workload*

by dividing the total number of successful `accept` calls by the number of accept-phases. Recall than in the µserver an accept-phase may admit zero new connections. This explains why C_{avg} is less than one for the Accept-1 policy. Unsuccessful `accept` calls (*i.e.*, those that return EWOULDBLOCK) are not counted in C_{avg}. The third column lists the total number of `accept` calls that were successfully completed during the experiment. The fourth column gives the average number of successful `accept` calls completed per second during the experiment. It is computed by dividing the total number of successful `accept` calls by the experimental duration. The last column displays the policy's throughput under a load of 26,000 reqs/sec.

The data in Table 4.2 further illuminates the differences between each accept-policy. This data shows that the Accept-Inf policy is relatively bursty compared to the other two policies. In particular, the Accept-Inf policy engages in a relatively small number (5,202) of accept-phases, but accepts a comparatively large number (47.3 on average) of new connections in each phase. This contrasts sharply with the Accept-1 policy which goes through many more (184,323) accept-phases, but accepts many fewer (0.97 on average) new connections in each phase. Overall, the Accept-Inf policy accepts new connections 38.2% faster than the Accept-1 policy at this request rate. The Accept-10 policy lies between the two extreme policies. The difference in accept rates explains differences in how each policy amortizes its `select` overhead.

The Accept-Inf policy accepts large batches of new connections. As a result, it maintains a large number of open connections, and submits large, dense fd_sets to `select`. Consequently, `select` returns a large number of fd-notifications with each call. This leads to an efficient amortization of `select` overhead. In comparison, the Accept-1 policy averages only 0.97 new connections per accept-phase. Despite a large number of accept-phases, this policy returns 38.2% fewer new connections. As a result, the server maintains a smaller number of open connections and submits sparser fd_sets to `select`. These sparse fd_sets, coupled with a large number of `select` calls, translates into unnecessary `select` overhead which hurts performance.

Table 4.3 shows the percentage of time the μserver spends doing key system calls. This data was gathered by recompiling the μserver to include `gprof` profiling, and re-running the experiment at 26,000 reqs/sec.

Function	**% Time Accept-1 policy**	**% Time Accept-Inf policy**	**Difference**
`select`	39.96	5.49	-34.47
`accept`	2.41	1.32	-1.09
`read`	5.92	15.02	9.10
`setsockopt`	5.66	10.75	5.09
`write`	3.77	6.66	2.89
`sendfile`	21.72	43.88	22.16
`close`	2.17	3.48	1.31

Table 4.3: *Percentage of time spent in μserver functions under the SPECweb99-like workload*

The first (leftmost) column lists the system calls. The second column lists the percentage of time spent executing each system call under the Accept-1 policy. The third column does the same for the Accept-Inf policy. The rightmost column simply subtracts the second column from the third, thus highlighting the differences between the two policies. The system calls are listed in the order they are executed in the main server loop.

The reader may notice that the server executes both the `write` and `sendfile` system calls, and that a significant amount of time is spent in the `setsockopt` system call. The `sendfile` call is used because it allows zero-copy writes. However, `sendfile` cannot be used to write HTTP headers. As such, the server uses the `setsockopt` call with the TCP_CORK flag to force the kernel to buffer any data written to the socket. The server then executes the `write` system call to write the HTTP header, followed by the `sendfile` system call to write the file data. Lastly, the server uses `setsockopt` with the TCP_UNCORK flag to direct the kernel to flush all the buffered data. Note that `setsockopt` is also used to set the TCP_NODELAY option, which disables Nagle's algorithm for aggregating small packets.

The data shows that the Accept-Inf policy spends 34% less time executing `select` calls, and considerably more time reading and writing data. Thus, the server effectively spends less time getting events of interest, and more time servicing connections. In fact, the server effectively doubles the amount of time spent in `sendfile` calls, while nearly tripling the time spent in `read` calls. The observed reduction in event notification overhead reinforces our analysis regarding the efficient amortization of `select` overhead.

We now turn our attention to the impact of accept policy on client response times. Response time, also known as server latency, is a widely used measure of server performance that is directly related to user satisfaction. Figure 4.5 shows the μserver's average response time under each accept policy as the load on the server increases. Although it increases throughput, the aggressive Accept-Inf policy presents a tradeoff in the form of increased response times. This response time data is obtained by measuring (at the client side) the time needed to complete each HTTP 1.1 request. The resulting measurements are summed, averaged and tabulated at the end of each experiment.

Although it is somewhat difficult to discern from Figure 4.5, all three accept policies offer comparably low latencies at request rates below 13,000 req/sec. At this point, there is a dramatic increase in response time (regardless of the accept policy). As load increases, the Accept-Inf policy produces noticeably higher response times than the Accept-10 and especially the Accept-1 policy. Interestingly, the spectrum of accept policies allows a system administrator to choose a policy whose throughput versus response time tradeoff best meets his/her needs. The Accept-1 policy provides low response times and low throughput. More aggressive policies increase throughput at the expense of response time.

The tradeoff between throughput and response time is worth careful consideration. Response time often correlates with user satisfaction. However, it cannot be used as the sole measure of server performance. For instance, a server could trivially obtain stellar response times by serving only a few requests (and ignoring all others). This would lead to low average response times and dismal throughput. As such, it is important to consider throughput and average response times together. Increased throughput and lowered response times are ideal. Increases in throughput that are accompanied by small increases

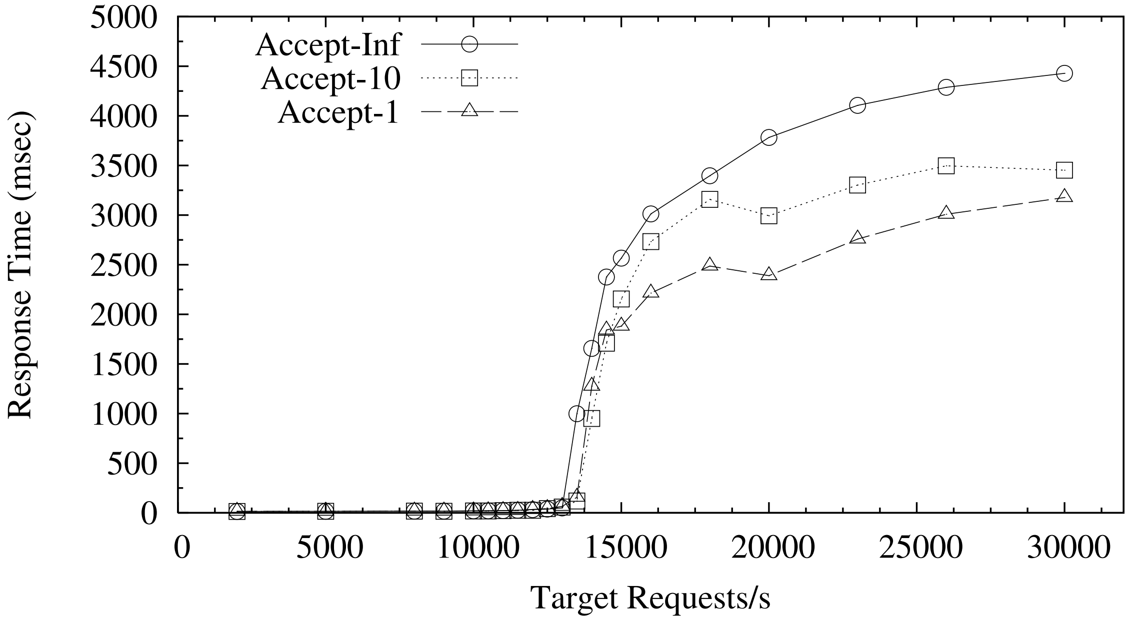

Figure 4.5: *Response times for the µserver under the SPECweb99-like workload*

in response times are often desirable. However, large increases in response time may be unacceptable.

It is worth noting that even the most aggressive of the accept policies shown here still provides reasonable average response times. In summary, the Accept-Inf policy outperforms the Accept-1 policy by as much as 39% on the in-memory SPECweb99-like workload. Analysis of a variety of server statistics shows that:

- The Accept-Inf policy accepts new connections at a significantly higher rate than the Accept-1 policy.
- The higher accept rate leads to a larger number of concurrently open connections in the server.
- With a large number of concurrent connections, the server submits dense fd_sets to `select`.
- As a result, more readable or writable fds are returned per `select` call, and fewer

`select` calls are made.

- More file descriptors are processed per `select` call (*i.e.*,the work phases are longer). This represents a superior amortization of `select` overhead, and leads to increased throughput.

- Longer work phases lead to more burstiness and higher response times.

4.1.2 TUX Performance

In Figure 4.6 we show that TUX's performance can be noticeably improved by choosing the right accept strategy. The Accept-Inf policy forces TUX to drain its accept queue by accepting all pending connections. This is the accept-policy that an unmodified TUX server uses. The Accept-50 policy allows TUX to consecutively accept up to 50 connections, while the Accept-1 policy limits TUX to accepting a single connection in each accept-phase.

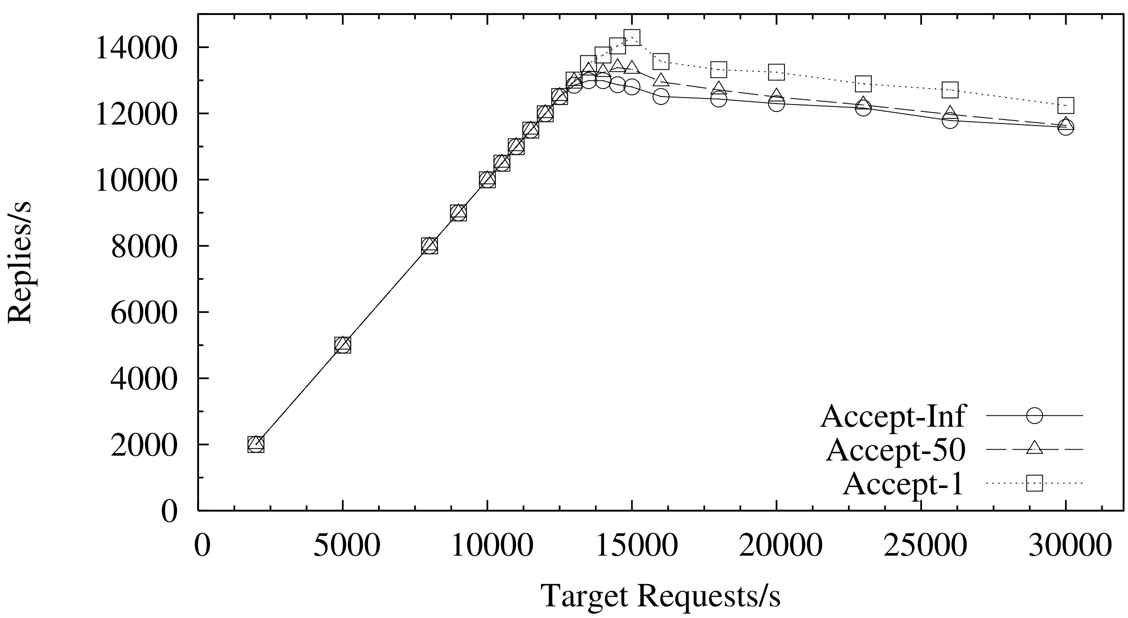

Figure 4.6: *TUX throughput under the SPECweb99-like workload*

Figure 4.6 shows that the Accept-1 policy results in a 10% increase in peak throughput compared to the Accept-Inf policy. Surprisingly, our server-side instrumentation shows that an Accept-1 policy causes TUX to accept connections faster than the higher accept-limit values. While this behaviour may seem unintuitive, it is important to remember that TUX's accept rate is not directly governed by the accept-limit parameter. Rather, the accept-limit controls the maximum number of connections that are accepted consecutively. The server's accept rate is determined by the number of consecutive accepts as well as the number of times that TUX enters its accept-phase. Equation 4.1 formalizes this simple mathematical relationship.

$$AcceptRate = \frac{N_{phases} C_{avg}}{t_{elapsed}} \tag{4.1}$$

In this equation, $t_{elapsed}$ denotes the elapsed time for a given experiment, N_{phases} represents the number of accept-phases the server completes during the experiment, and C_{avg} denotes the average number of new connections accepted per accept-phase. Using these terms, the accept-limit parameter represents an upper bound on the value of C_{avg} as shown in equation 4.2

$$C_{avg} \leq \textit{accept-limit} \tag{4.2}$$

In TUX, lowering the accept-limit has two effects. Firstly, C_{avg} decreases to comply with equation (4.2). Secondly, N_{phases} increases dramatically. In our experiments, the increase in N_{phases} outweighs the decrease in C_{avg} and leads to a net increase in the observed accept rate. This was confirmed by carefully instrumenting and experimenting with a separate version of TUX. The resulting run-time statistics are presented in Table 4.4. The leftmost column shows the number of accept-phases that TUX completed during the experiment. The second column shows the average number of connections that were accepted in each accept-phase. The third column lists the total number of connections that were accepted during the experiment. The rightmost column gives the average number of connections that were accepted per second during the experiment. It is computed by dividing the total number of accepts by the experimental duration.

This data shows that the Accept-1 policy increases the rate at which TUX accepts new connections by approximately 17.6% when compared to the default Accept-Inf policy (1,474 versus 1,734 accepts/sec). The increase arises because C_{avg} decreases by a factor of 503 (from 503 to 1), while the number of accept-phases increases by a factor of 543 (from

	N_{phases}	C_{avg}	Total Accepts	Accept Rate	Throughput
Accept-Inf	430	503.82	216,641	1,474	11,788
Accept-1	233,639	0.99	233,639	1,734	12,710

Table 4.4: *Accept-phase statistics for TUX @ 26,000 reqs/sec under the SPECweb99-like workload*

430 to 233,639). According to equation 4.1, this translates into a net increase in accept rate.

An examination of TUX's `work_pending` queue is helpful in understanding this behaviour. Table 4.5 presents some statistics related to TUX's processing of its `work_pending` queue. The left column lists the average number of connections accepted by each policy. The right column lists the average number of items processed from the `work_pending` queue in each work-phase. The difference is startling. The Accept-1 policy processes an average of 46 items in each work-phase, while the Accept-Inf policy averages 11,083 items per work-phase. These figures show that the Accept-Inf policy has long accept-phases followed by even longer work-phases (since each accepted connection generates several work items). During these long work-phases, no new connections are accepted. In fact, many connections may be rejected because the relevant operating system queues are full. When the server enters its next accept-phase, many of the waiting connections may be stale. In contrast, the Accept-1 policy engages in short accept-phases followed by relatively short work-phases. This policy avoids bursty processing and is able to re-enter the next accept-phase in short order. Overall, this translates into a higher accept rate.

	C_{avg}	Pending Work items
Accept-Inf	503.82	11,083
Accept-1	0.99	46

Table 4.5: *Pending work statistics for TUX @ 26,000 reqs/sec under the SPECweb99-like workload*

Interestingly, the accept-limit parameter affects TUX and the μserver in very different ways, despite the fact that both are event-driven servers with accept-phases and work-phases. Because of this similarity, Equation (4.1) applies equally to both servers. In the

μserver, lowering the accept-limit parameter lowers C_{avg}, and increases N_{phases}. This, is similar to what was observed for TUX. However, for the μserver the increase in N_{phases} is unable to compensate for the decrease in C_{avg}. As a result, the μserver accept-rate falls when its accept-limit is lowered.

This analysis shows that although both servers experience an increase in N_{phases} and a decrease in C_{avg}, the magnitude of the changes are quite different for each server. The difference in magnitude arises because of the getevents-phase that exists in the μserver but not in TUX. In the μserver each accept-phase is preceded by a getevents-phase (essentially a call to `select`). Increasing the number of accept-phases also increases the number of getevents-phases. This adds an overhead prior to each accept-phase, and limits the μserver's ability to perform more accept-phases. In comparison, TUX incurs no overhead for extra accept-phases.

In addition to improving TUX's throughput, the Accept-1 policy also reduces request latency. There are several reasons for this. First, by accepting fewer connections, TUX is able shorten its `work_pending` queue. As a result, work items endure shorter wait times and are processed faster. Second, by accepting new connections faster, the Accept-1 policy reduces the time it takes the server to establish its connection with the client. In fact, reduced connection-times are a major contributor to lower latencies. Figure 4.7 graphs TUX's latency under each accept policy as the load increases.

Although it is somewhat difficult to discern from Figure 4.7, all three accept policies offer comparably low latencies at request rates below 13,000 req/sec. At this point, all three policies experience a dramatic increase in client response times. The Accept-1 policy provides slightly lower latency than the other two policies, especially at higher request rates. As such, the Accept-1 policy provides moderate improvements to both throughput and response time when compared to the default Accept-Inf policy.

Figure 4.8 shows the queue-drop rates for all three policies. The graph shows that all three policies have comparable queue drop rates. This is not surprising for two reasons. Firstly, Figure 4.6 shows that the differences in performance among these three policies are not huge. Secondly, the SPECweb99-like workload sends several requests per connection, which means that fewer connections are established and fewer queue drops will occur.

In summary, the Accept-1 policy increases TUX's throughput by as much as 12%, and

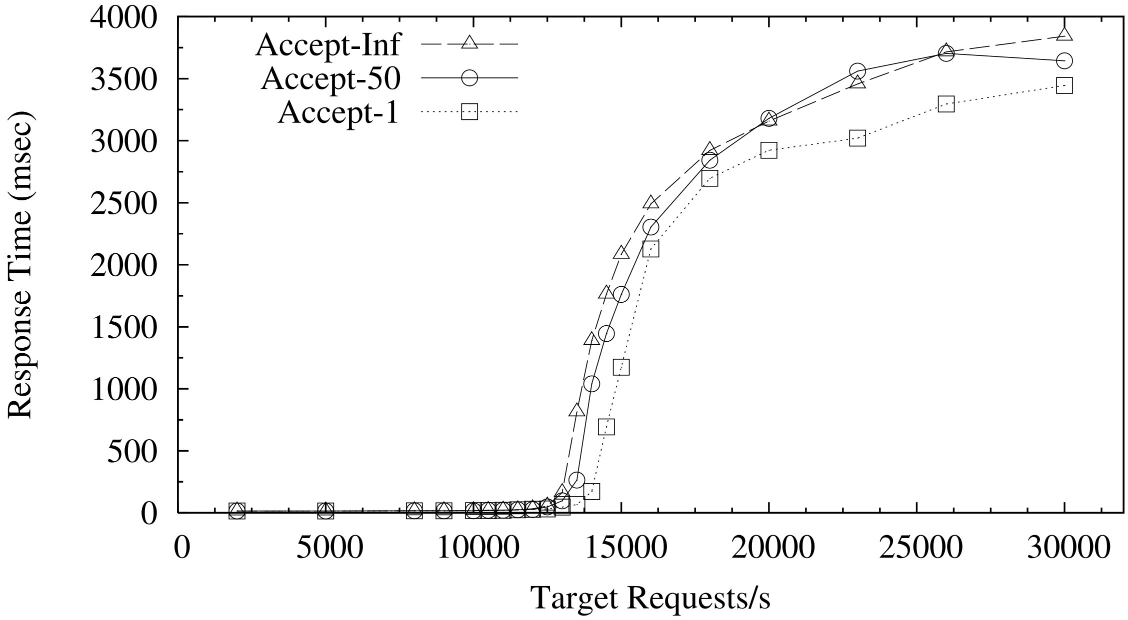

Figure 4.7: *Response times for TUX under the SPECweb99-like workload*

reduces client response times by 15% to 23%. Our analysis shows that:

- The default Accept-Inf policy causes TUX to accept large batches of connections.
- Long accept phases are invariably followed by long work phases which process thousands of work items.
- During these long work phases, operating system queues fill and connections are dropped. Long work phases also inhibit the server's ability to re-enter the accept-phase.
- The Accept-1 policy remedies these problems by shortening the accept-phase and the work-phase. As a result, connections are accepted and processed in smaller batches.
- The smaller batches increase TUX's accept rate, which leads to higher throughput than the default Accept-Inf policy.
- The reduced batch size reduces TUX's burstiness, and provides lower latencies.

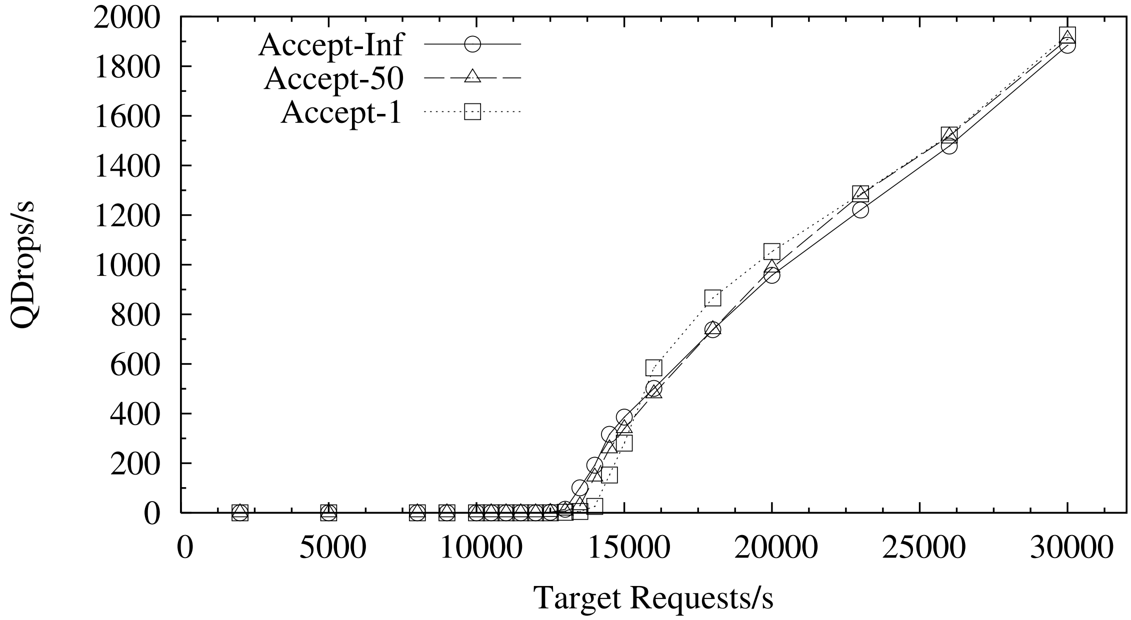

Figure 4.8: *TUX queue drops under the SPECweb99-like workload*

4.1.3 Knot Performance

For the Knot server, we experimented with a variety of different accept strategies. The results are summarized in Figures 4.9 and 4.10. Figure 4.9 illustrates the throughput obtained using three different accept policies. With the accept-limit parameter set to 1, our modified version of Knot behaves identically to an unmodified copy of Knot.

As a sanity check, we confirmed that the original version and the modified server using the Accept-1 policy produce results that are indistinguishable. To reduce clutter, the results for the original version of Knot are omitted.

Higher accept-limits (50 and 100) represent our attempts to increase Knot's throughput by increasing its accept rate. Our server-side measurements confirm that we are able increase Knot's accept rate. For example, statistics collected by Knot indicate that at a load of 20,000 requests/sec, the Accept-100 strategy accepts new connections 2.4 times faster (on average) than the Accept-1 (default) strategy Further evidence is provided in Figure 4.10 which shows that the Accept-50 and Accept-100 servers enjoy significantly lower queue drop rates than the Accept-1 policy.

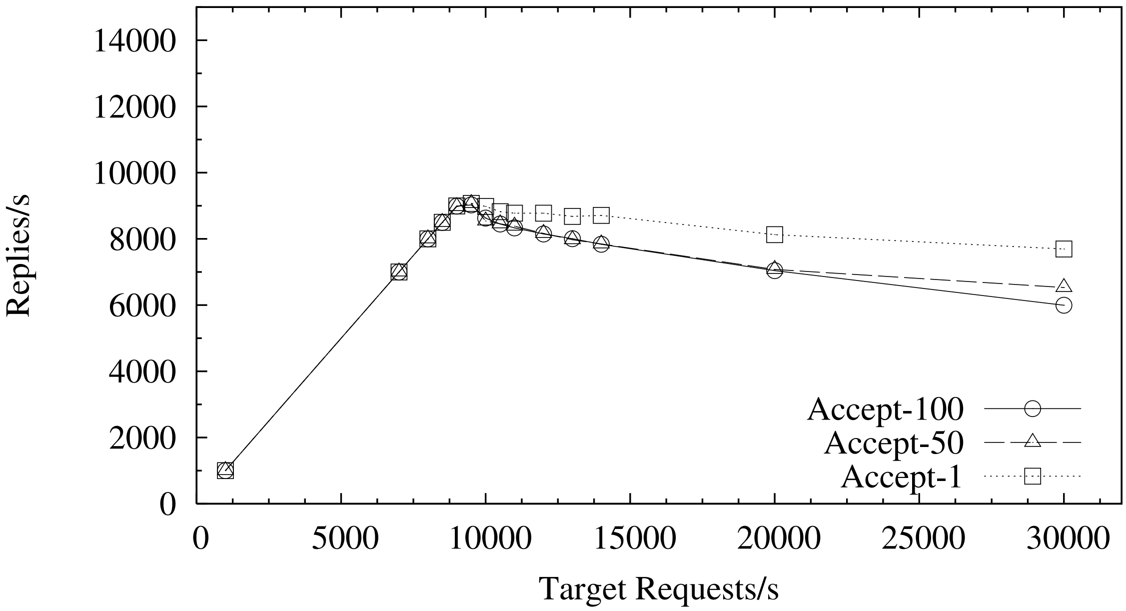

Figure 4.9: *Knot throughput under the SPECweb99-like workload*

Unfortunately, the higher accept rates (and lowered queue drop rates) do not improve Knot's throughput. To the contrary, performance suffers. To make matters worse, Figure 4.11 shows that average response times are considerably higher (and growing sharply) under the more aggressive accept policies. It is difficult to gain insights into Knot's behaviour because Knot uses multiple user-level and kernel-level threads to perform I/O operations. As a result, it cannot be profiled with tools like `gprof`. We have attempted to profile Knot with the `oprofile` system profiler. However, at the time these experiments were run, `oprofile` only provided flat profiles (as opposed to the more useful call-graph profiles). The flat profiles only accounted for time spent executing in the body of a function. They did not accumalate time spent in child functions into the parent's running total. As a result, the flat profiles did not provide any insight into Knot's behaviour. Lastly, we added a number of counters and timers to Knot in order to help us understand its behaviour. Although these modifications provided some insight, they did not provide any strong evidence that would explain Knot's performance.

With the recent arrival of the Linux 2.6 kernel, `oprofile` now provides more useful

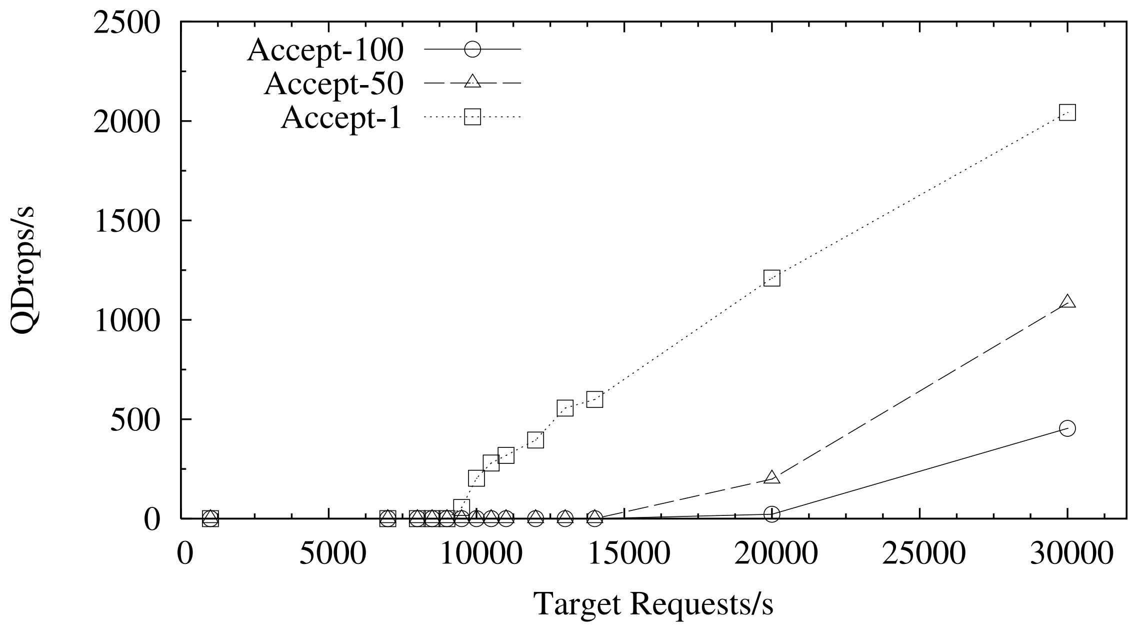

Figure 4.10: *Knot queue drops under the SPECweb99-like workload*

call-graph profiles. We intend to study Knot's performance using `oprofile` and the 2.6 kernel in the near future. We also plan to further modify Knot so that it gathers and reports more statistics about its own behaviour. We hope that these measures will allow us to better understand and explain Knot's performance.

One of the run-time statistics gathered by Knot shows that with an accept-limit of 50 or higher, the number of concurrent connections in the server grows quite sharply compared to the Accept-1 policy. We suspect that performance degrades with a large number of connections because of overheads in the Capriccio threading library. As a result, we find that under this workload, accepting new connections more aggressively does not improve Knot's performance. These findings agree with previously published results [40] in which overly aggressive accepting also hurt Knot's performance.

This section has examined the performance of the μserver, TUX, and Knot under the in-memory SPECweb99-like workload. The results presented here have demonstrated that a more aggressive accept strategy can improve the throughput of the μserver and TUX both at peak and under overload conditions. For the μserver, the increase in throughput

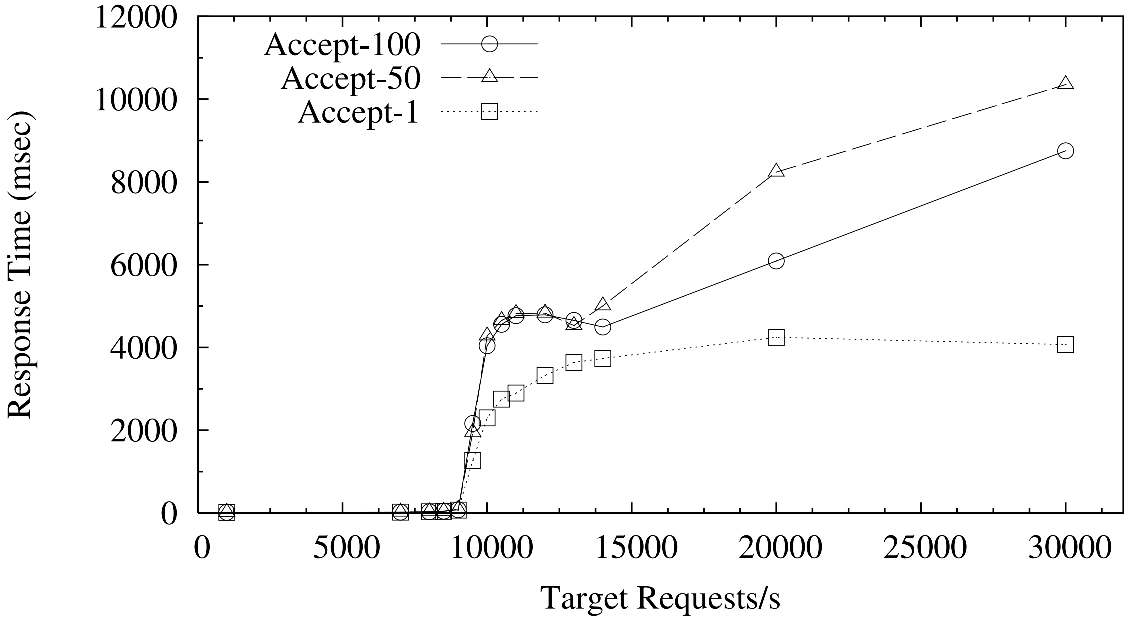

Figure 4.11: *Response times for Knot under the SPECweb99-like workload*

came at the expense of a small increase in average response time. For TUX, the improvement in throughput was accompanied by a decrease in average response time. Increasing Knot's accept rate led to decreased throughput under the SPECweb99-like workload. This demonstrates that a more aggressive accept policy does not always improve performance. Instead, a server must balance the accepting of new connections with the processing of existing connections. The next section examines the performance of each server under the one-packet workload, which emphasizes many phenomena that are neglected by the SPECweb99-like workload.

4.2 One-packet workload

This section examines the performance of each server under the one-packet workload. This workload features smaller transfer sizes and fewer requests per connection than the SPECweb99-like workload. These factors translate into a higher demand for new connections at the server than was seen with the SPECweb99-like workload.

4.2.1 µserver Performance

Figure 4.12 graphs the µserver's throughput under the one-packet workload using three different accept policies. This graph demonstrates that a well-chosen accept strategy increases the µserver's peak throughput from 19,500 replies/sec using the naive Accept-1 strategy to 22,000 replies/sec using the Accept-10 strategy. This is an improvement of 13%. More importantly, the Accept-Inf strategy improves performance versus the naive strategy by as much as 65% at 21,000 requests/sec and 71% at 30,000 requests/sec. Interestingly, the Accept-10 strategy achieves a slightly higher peak than the Accept-Inf strategy, although it experiences larger decreases in throughput as the load increases. This suggests that better performance might be obtained by dynamically adjusting the accept strategy. This is something we plan to investigate in future research.

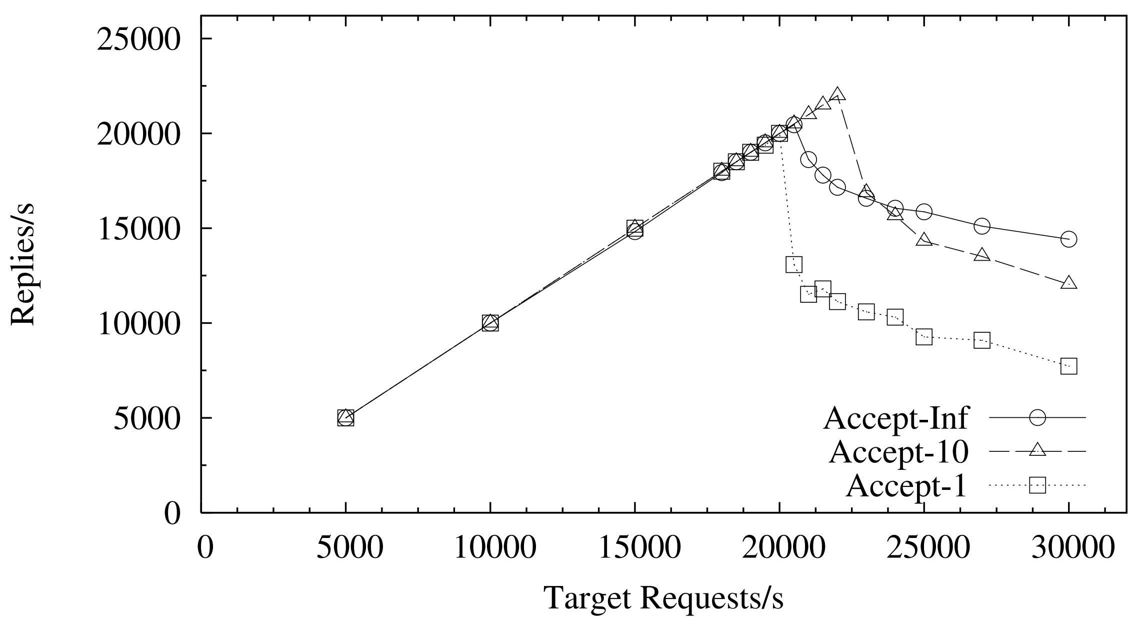

Figure 4.12: *µserver throughput under the one-packet workload*

This graph also exhibits a number of interesting features that were not observed under the SPECweb99-like workloads. For example, there is a marked difference in peak throughput across the accept policies. The Accept-1 policy achieves significantly lower peak throughput than either the Accept-Inf (7.1% lower) or the Accept-10 (11.4% lower) policies.

This constrasts sharply with the results from the in-memory SPECweb99-like workload in which all three policies achieved similar peak throughputs. This difference arises from a fundamental difference between the two workloads. Specifically, the SPECweb99-like workload uses each connection to send an average of 7.24 HTTP 1.1 requests. This average arises from the request distributions specified in the SPECweb99 documentation. In comparison, the one-packet workload sends a single request over each connection. As a result, the one-packet workload generates a much higher demand for new connections than the SPECweb99-like workload.

Table 4.6 highlights the differences in the demand for new connections across each workload. The data in Table 4.6 is aggregated from statistics produced by httperf on each client machine. In this table, the request rate column gives the target request rate in requests/sec, while the rightmost column gives the rate at which new connection requests are sent to the server. The data clearly shows that the one-packet workload generates almost eight times more demand for new connections.

	Request Rate	**Connection Reqs/sec**
One-packet workload	20,000	20,000
SPECweb99-like workload	20,000	2,542

Table 4.6: *Workload-specific demand for new connections at 20,000 reqs/sec*

Another interesting feature of Figure 4.12 is that all three accept policies exhibit significant degradation in throughput once saturation occurs. At 30,000 reqs/sec, throughput under the Accept-1 policy falls to 8,741 replies/sec which is a mere 45% of peak throughput. The Accept-10 and Accept-Inf policies fare somewhat better, degrading to 58% and 71% of their respective peak throughputs at 30,000 reqs/sec. In comparison, degradation under the in-memory SPECweb99 workload is relatively mild, especially for the Accept-10 (which falls to 90% of peak) and Accept-Inf (which falls to 86% of peak) policies.

Figure 4.13 shows the number of queue drops incurred under each accept policy. Two features of this graph are interesting. Firstly, we observe that the Accept-1 policy suffers a relatively early onset of queue drops at 20,000 reqs/sec.

The Accept-10 and Accept-Inf policies avoid significant numbers of queue drops until 23,000 reqs/sec and 21,500 reqs/sec respectively. For each policy, the onset of queue drops

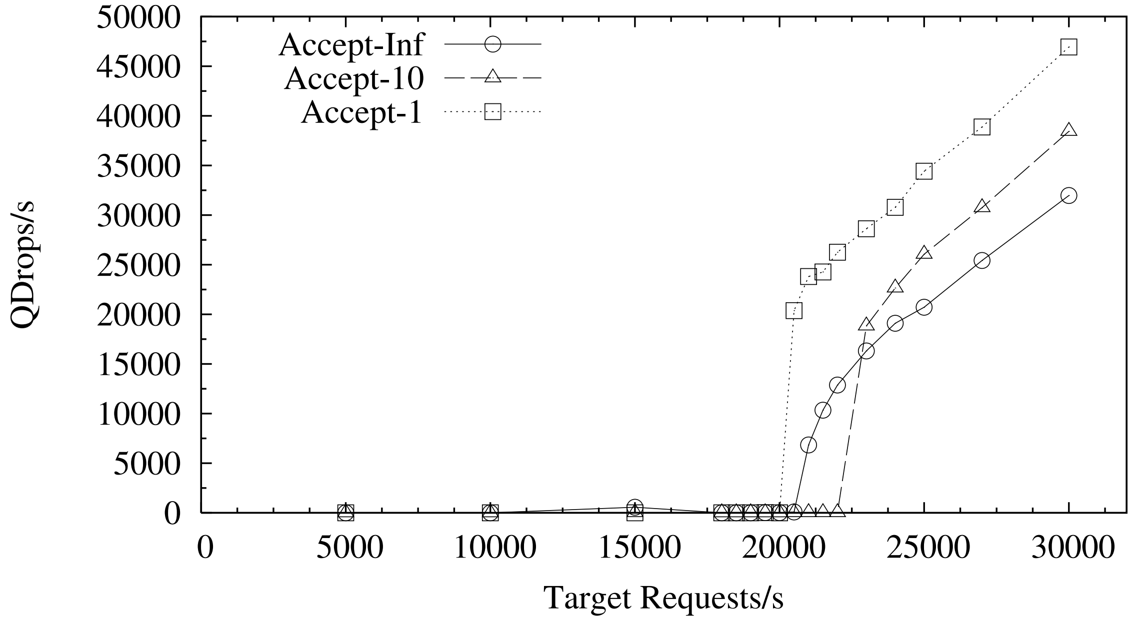

Figure 4.13: *µserver queue drops/sec under one-packet workload*

corresponds to server saturation. At this point, new connection requests are arriving faster than they can be accepted, and the relevant operating system queues are already full. The lower queue drop rates incurred by the Accept-Inf and Accept-10 policies demonstrate that they do accept more aggressively than the Accept-1 policy.

The analysis in Section 4.1.1 showed that improved amortization of `select` overhead was a key to improving performance. A similar analysis holds under the one-packet workload, although many details are different. Figure 4.14 shows the number of `select` system calls made per second by the µserver as the target load increases. Under loads of less than 20,000 reqs/sec, all three policies call `select` frequently. This is not surprising, since the server is not saturated (under any of the policies), and spends a lot of its time polling for new work. This means the server calls `select` often, hoping to receive notification on fds that are readable or writable. It is noteworthy that even before saturation the Accept-1 policy calls `select` at a much higher rate than the other two policies, because `select` is called for each connection that is accepted. This does not affect performance since the server is not yet saturated.

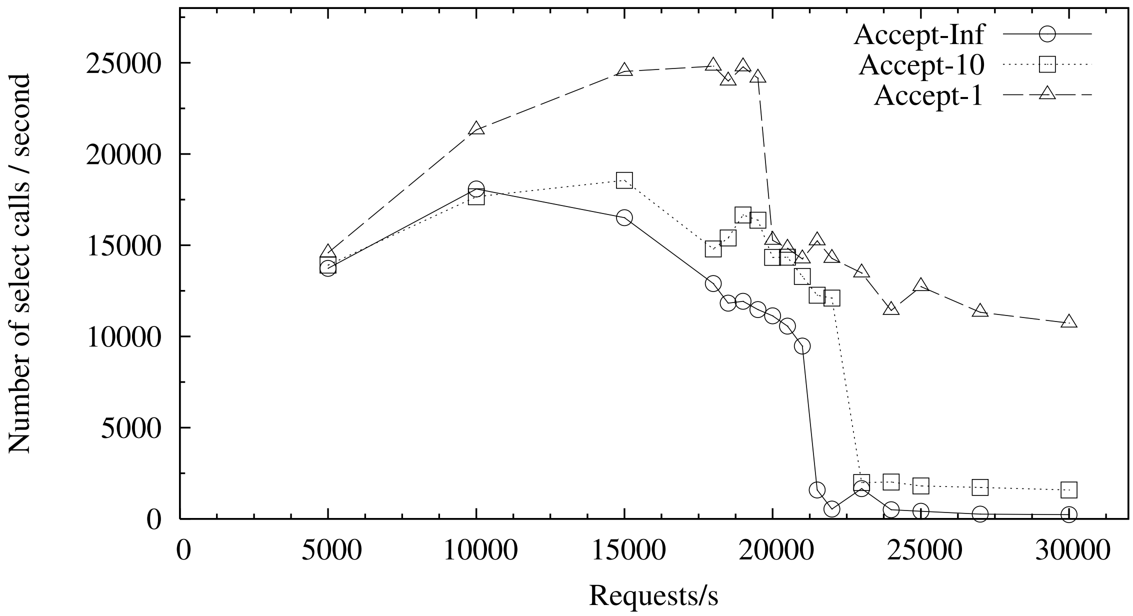

Figure 4.14: *Number of* `select` *calls made per second by the µserver under the one-packet workload*

After saturation, the select-rate falls dramatically for each policy. However, the Accept-Inf policy emerges as the clear winner in reducing `select` calls. The key to success in the Accept-Inf policy is to get more useful work done with each `select` call.

Figure 4.15 makes it clear that the Accept-Inf policy makes efficient use of `select` at all request rates. This graph shows the average number of fds returned by `select` as load increases. Prior to saturation, `select` calls return few fds regardless of the accept policy employed. This is reasonable, since the server has relatively few connections to service. As loads increase (and overload conditions are encountered) the Accept-Inf policy makes increasingly better use of each `select` call. At 24,000 reqs/sec, each call returns 107 fds. This grows to 192 fds/call at 27,000 reqs/sec, and 205 fds/call at 30,000 reqs/sec. In contrast, both the Accept-10 and Accept-1 policies plateau quickly. The former plateaus at approximately 26 fds/call, while the latter stays constant at 4 fds/call.

The picture that is emerging shows that the Accept-Inf policy makes relatively few `select` calls, and extracts more useful work from each call. This means that the server

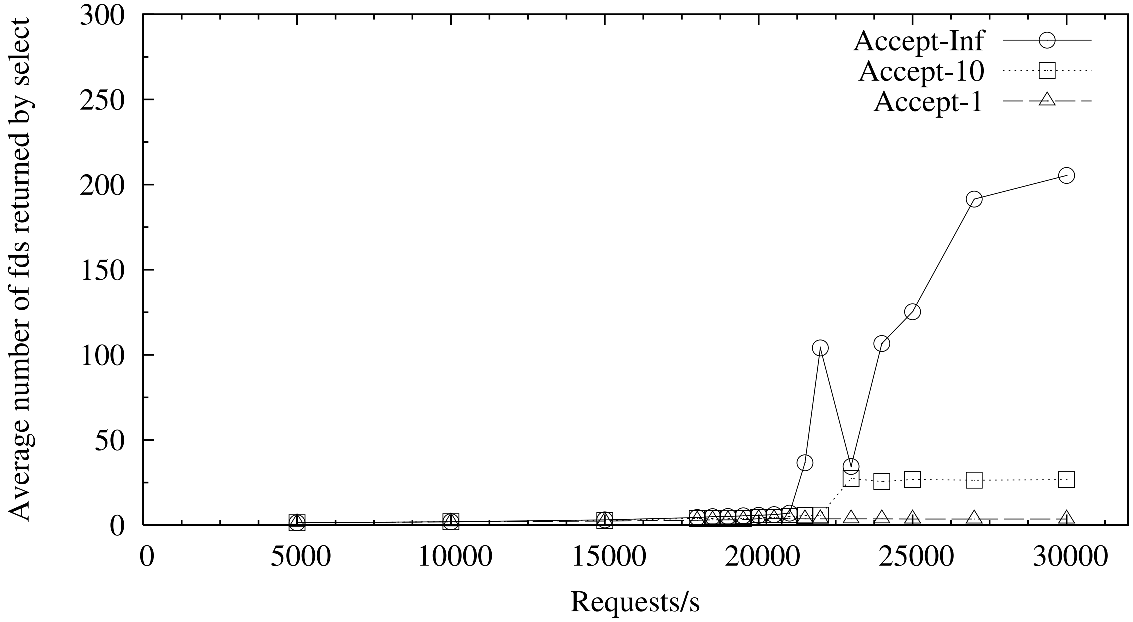

Figure 4.15: *Average number of fds returned by* `select` *in the μserver under the one packet workload*

incurs less overhead from `select` calls. Table 4.7 emphasizes this by presenting key statistics concerning each policy's use of `select`. The select-rate column shows the number of `select` calls made per second. The Avg fds per select call column lists the average number of file descriptors returned by a `select` call. The fd-rate column is simply the product of the select-rate and Avg fds from select columns, and gives the rate (in fds per second) at which fds are returned from `select` to the application. The rightmost column lists the server throughput for each accept policy. All the data in Table 4.7 is based on a target load of 27,000 reqs/sec.

	select-rate	**Avg fds per select call**	**fd-rate**	**Throughput**
Accept-Inf	262	191.5	50,173	15,918
Accept-10	1,724	26.4	45,513	13,845
Accept-1	11,372	3.6	40,939	9,442

Table 4.7: `select` *statistics for μserver @ 27,000 reqs/sec under the one-packet workload*

The table shows that the Accept-Inf policy makes approximately 43 times fewer `select` calls than the Accept-1 policy, and 6.5 times fewer than the Accept-10 policy. In spite of this, the Accept-Inf policy receive event notifications at a higher rate than the other two policies. The fd-rate column shows that the Accept-Inf policy returns 22.5% more fds/sec than the Accept-1 policy. As a result, the server obtains event notifications faster and is able to efficiently amortize `select` overhead by completing more work (reads and writes) per `select` call. This is precisely the pattern exhibited under the SPECweb99-like workload. In both cases, the Accept-Inf policy is superior because it allows the server to efficiently amortize the overhead of getting events from the operating system.

As with the SPECweb99 workload, the key to using `select` efficiently is to submit large, dense fd_sets to `select`. In return, `select` returns notification on several active file descriptors, and increases the amount of real work the server can do with each `select` call. Using `select` as described requires the server to maintain a large number of concurrent connections. This in turn requires the server to accept new connections aggressively. This is why aggressive accept policies (like Accept-Inf) are able to significantly reduce their `select` overhead. Table 4.8 presents statistics gathered from the μserver that quantify how aggressively each policy accepts new connections. The column headings in this table are identical to those in Table 4.2.

	N_{phases}	C_{avg}	Total Accepts	Accept Rate	Throughput
Accept-Inf	32,249	70.43	2,271,144	18,462	15,918
Accept-10	212,087	9.53	2,021,944	16,436	13,845
Accept-1	1,398,961	0.99	1,398,152	11,365	9,442

Table 4.8: *Accept-phase statistics for μserver @ 27,000 reqs/sec under the one-packet workload*

The reader may notice that each policy has an accept rate that is higher than its throughput. This means that the server accepts connections that it never completes. The μserver's statistics show that the μserver is unable to read any data from these connections. This indicates that the connection has been closed by httperf because the connection timer has expired. If the μserver is unable to read any data from a connection it simply closes that connection. This accounts for the discrepancy between the rate at which connections are accepted (the accept rate) and the rate at which they are completed (the throughput).

Table 4.8 shows that the Accept-1 policy performs an enormous number of accept-phases, but accepts at most one connection in each phase. In contrast, the Accept-Inf policy performs significantly fewer accept-phases, but averages more than 70 new connections in each phase. The statistics in Table 4.8 show that this strategy is quite successful; it allows the Accept-Inf policy to accept 62% more connections than the Accept-1 policy. As a result, the server manages more concurrent connections during execution. This in turn allows the server to submit larger fd_sets to `select`. The end result is that the server obtains a large number of events from each `select` call, which reduces the number of `select` calls that need to be made. As a result the overhead of the `select` call is reduced, which allows the server to spend more time servicing its open connections.

Table 4.9 shows the percentage of time the μserver spends doing key system calls. This data was gathered by recompiling the μserver to include `gprof` profiling, and re-running the experiment at 27,000 reqs/sec. The column headings in this table are identical to those in Table 4.3.

Function	**% Time Accept-1 policy**	**% Time Accept-Inf policy**	**Difference**
`select`	32.43	15.84	-16.59
`accept`	6.81	9.78	2.97
`read`	6.17	19.34	13.17
`writev`	0.02	0.04	0.02
`close`	7.36	16.14	8.78
`setsockopt`	2.11	2.98	0.87

Table 4.9: *Percentage of time spent in various μserver functions under the one-packet workload*

The reader may notice that the `sendfile` and `write` system calls have been replaced by the `writev` system call. This is because under the one-packet workload the overhead of using `setsockopt` with TCP_CORK and TCP_UNCORK is not cost effective. We have found that the μserver obtains better performance under the one-packet workload if it uses `writev` to write both the HTTP header and the file data. We are aware that `writev` does not provide zero-copy writes. However, for the small reply sizes required under the one-packet workload the copying overhead is negligible. The data in Table 4.3 confirms that `writev` consumes a very small amount of the servers's execution time. Note that `setsockopt` is still used to set the TCP_NODELAY option on all newly accepted sockets. This option disables Nagle's algorithm for aggregating small packets.

The gprof data shows that the Accept-Inf policy allows the server to simultaneously reduce the amount of time spent executing `select` calls, and increase the amount of time spent accepting and processing connections. The server reduces its `select` overhead by nearly 17%. These savings allow the server to spend more time executing `accept` (9.78% versus 6.81%), `read` (19.34% versus 13.17%), and `close` (16.14% versus 7.36%) system calls. The server also increases the amount of time spend doing `writev` calls, although these account for a small portion of execution time (because of the single-packet reply). Overall, the changes in where the server spends its time translate into a healthy increase in measured throughput.

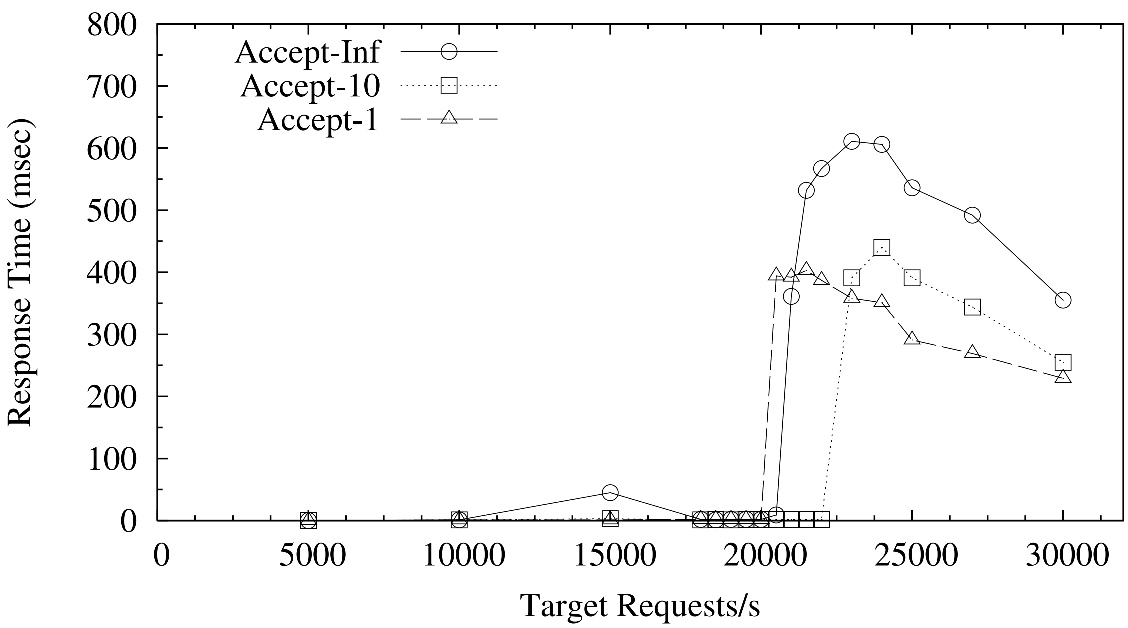

Figure 4.16: *Response times for µserver under the one-packet workload*

We now study the impact of the accept policy on server latency. Figure 4.16 shows the response times for each accept policy under the one packet workload. Interestingly, the

Accept-10 policy simultaneously provides higher peak throughput and lower latency than the Accept-Inf policy. This is ideal since there is no tradeoff between peak throughput and latency for the Accept-10 policy. The graph also shows that the Accept-Inf policy has the highest latency of the three policies. However, all three policies enjoy relatively low latencies, and the small increase in latency under the Accept-Inf policy is a reasonable price to pay for the improvement in throughput under overload. We now turn our attention to the performance of TUX under the one-packet workload.

4.2.2 TUX Performance

Figure 4.17 graphs TUX's reply rate against the target request rate. Recall that the Accept-Inf strategy corresponds to the original TUX accept strategy. In this case the improved Accept-1 strategy results in a peak reply rate of 25,001 replies/sec compared with the original, whose peak is at 20,457 replies/sec. This is an improvement of 22%. Additionally there is an improvement of 39% at 25,000 reqs/sec.

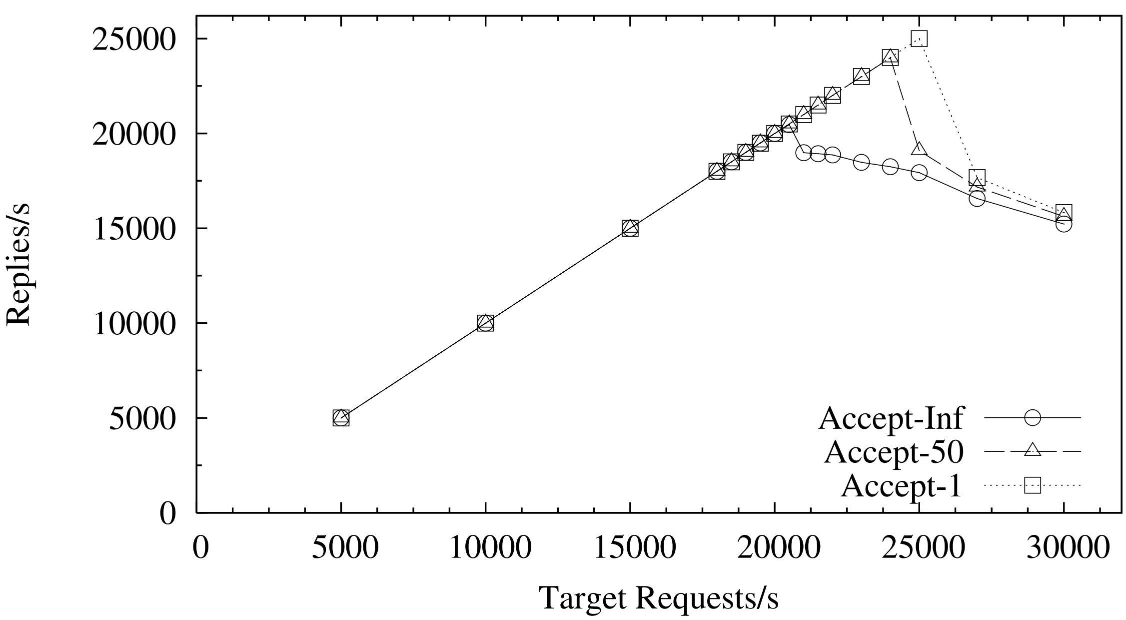

Figure 4.17: *TUX throughput under the one-packet workload*

As with the SPECweb99-like workload, limiting the number of consecutive accepts increases TUX's accept rate. This can be seen by examining the accept statistics presented in Table 4.10. The column headings in this table are identical to those in Table 4.2. These statistics show that (on average) the Accept-1 policy accepts fewer connections in each accept phase, but performs many more accept phases. On balance, this leads to a higher net accept rate. In fact, the Accept-1 policy accepts connections 25% faster than the Accept-Inf policy at a request rate of 23,000 reqs/sec. Further evidence of the higher accept rate is provided by the queue drop rates in Figure 4.18. This graph shows that the Accept-1 strategy delays the onset of queue drops, especially when compared to the Accept-Inf policy. It also maintains the lowest queue drop levels of all three policies, especially from 20,000 reqs/sec to 27,000 reqs/sec.

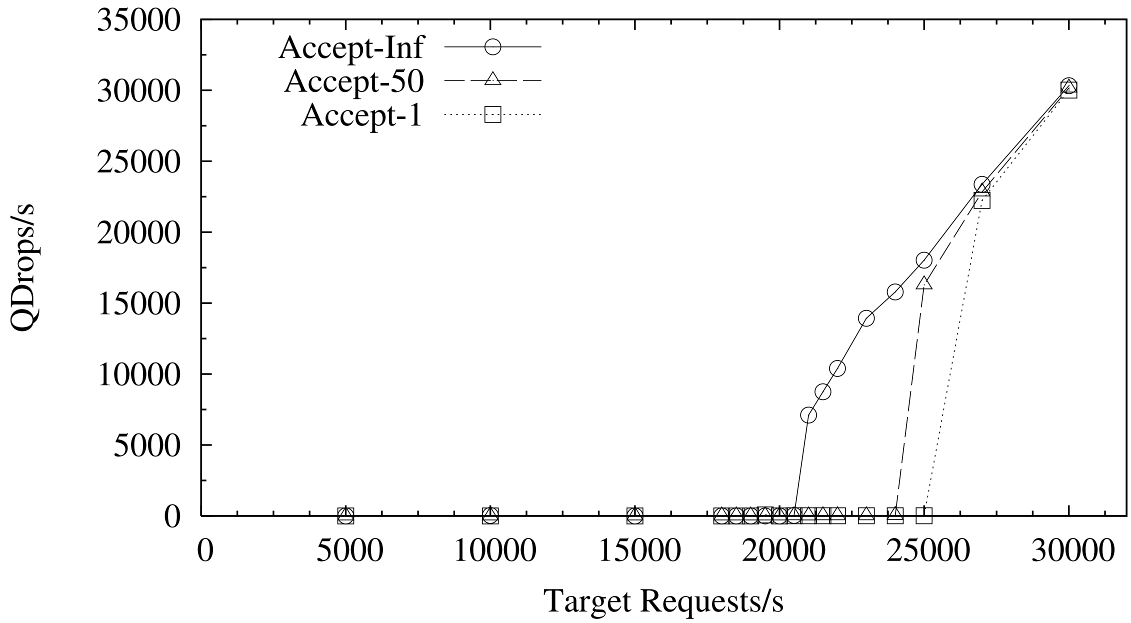

Figure 4.18: *TUX queue drops under the one-packet workload*

The Accept-1 policy improves more than just throughput. Figure 4.19 shows that Accept-1 policy also provides lower response times than its less aggressive counterparts. The difference is most pronounced from 20,000 reqs/sec to 27,000 reqs/sec. This range is also where the greatest gains in throughput are realized. The double-benefit provided by

	N_{phases}	C_{avg}	**Total Accepts**	**Accept Rate**	**Reply Rate**
Accept-Inf	237,551	9.54	2,266,760	18,430	18,470
Accept-1	2,759,968	0.99	2,759,968	22,980	22,997

Table 4.10: *Accept-phase statistics for TUX @ 23,000 reqs/sec under the one-packet workload*

the Accept-1 policy is quite different from the μserver case, in which there was a tradeoff between increased throughput and reduced latency.

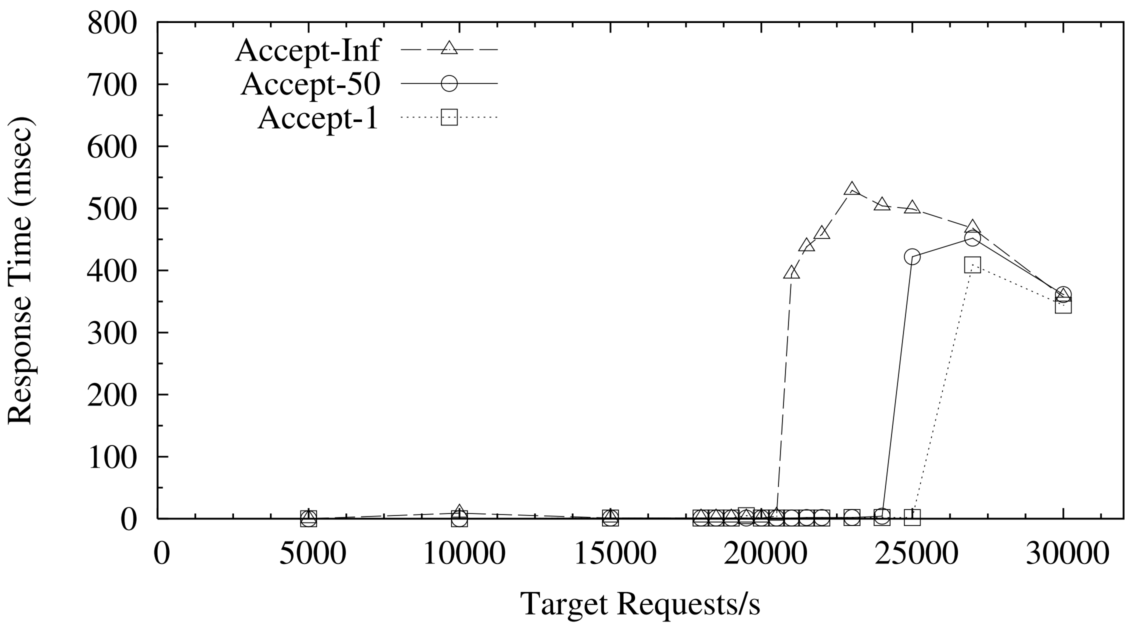

Figure 4.19: *Response times for TUX under the one-packet workload*

The reduction in TUX's latency arises because the Accept-1 policy accepts connections in small batches (short accept phases), and processes them quickly (short work phases). As a result, connections receive service quickly, without accruing large amounts of wait time. In contrast, the μserver under the Accept-Inf policy accepts connections in large batches (long accept phases), and processes them in large batches (long work phases). This strategy allows the server to efficiently amortize its `select` overhead. However, the connections that are processed later in each work phase accrue long wait times, which leads to higher average response times.

Although we have realized significant gains over TUX's default performance, we believe further improvements are possible. However, our simple method of controlling how TUX accepts new connections does not allow us to accept less than one connection in each accept phase. Ultimately, we believe that the best way to control the accept strategy used in TUX, and to control the scheduling of work in general, is to track the number of entries contained in the `accept_queue` and the number of entries in `work_pending` queue. This information can be used to make a more informed decision regarding whether to enter an accept-phase or a work-phase. We also believe that limits should be placed on the amount of time spent in each phase, possibly by limiting the number of consecutive events that are processed from each queue. We believe that this approach might be used to further increase the rate at which the server accepts new connections. The difficulty lies in ensuring that the server strikes a balance between accepting new connections and processing existing connections.

4.2.3 Knot Performance

Knot benefits from the tuning of its accept policy under the one-packet workload. Figure 4.20 shows an interesting spectrum of accept policies. We observe that the Accept-50 strategy noticeably improves throughput when compared with the original accept strategy. Firstly, peak throughput increases by 17% from 12,000 to 14,000 replies/sec. Secondly, throughput increases by 32% at 14,000 reqs/sec, and by 24% at 30,000 reqs/sec.

Interestingly, increasing the accept-limit too much (for example to 100) can result in poor performance. In comparing the Accept-100 strategy with the Accept-1 strategy (the default), we observe that the former obtains a slightly higher peak. However, with the Accept-100 policy throughput degrades sharply once the saturation point is exceeded.

Figure 4.21 shows how the queue drop rates are impacted by the changes in the accept strategy. Here we see that the Accept-100 version is able to tolerate slightly higher loads than the original before suffering from significant queue drops. The Accept-50 version is slightly better, and in both cases peak throughput improves. At request rates of 15,000 and higher the Accept-50 and Accept-100 strategies do a slightly better job of preventing queue drops than the server using an accept-limit of 1. Interestingly, queue drop rates for the accept-limit 50 and 100 options are quite comparable over this range,yet, there is a large

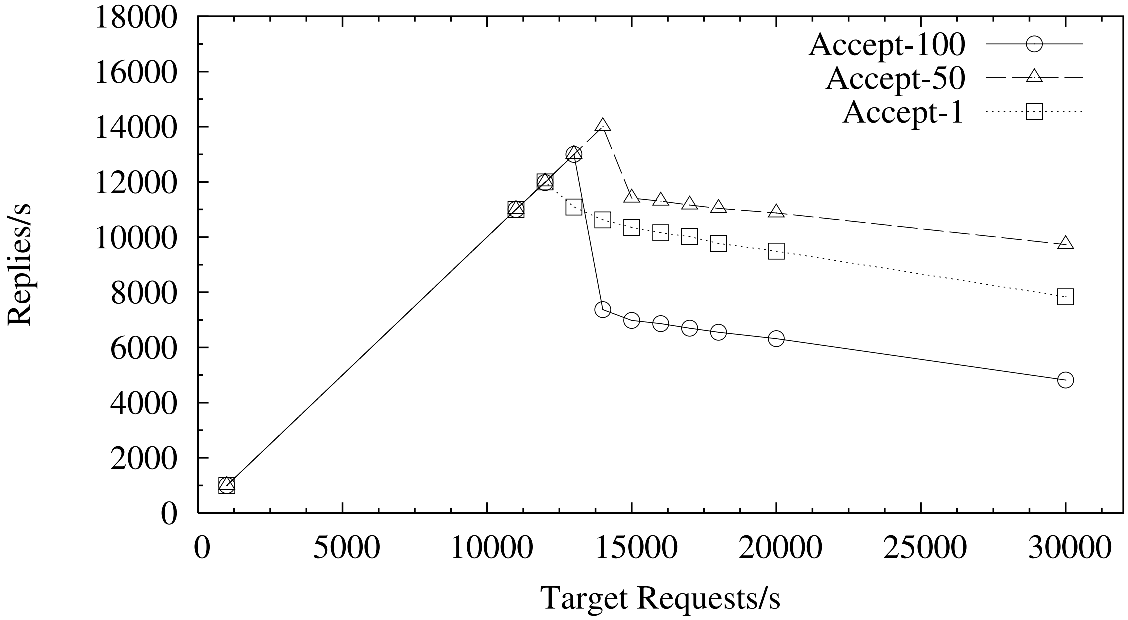

Figure 4.20: *Knot throughput under the one-packet workload*

difference in performance. The statistics printed by the Knot server show that at 15,000 requests/sec the Accept-50 policy operates with approximately 25,000 active connections, while the Accept-100 policy is operating with between 44,000 to 48,000 active connections. Unfortunately, our efforts to understand how Knot's behaviour changes as the number of active connections grows were not successful. However, these experiments highlight that a balanced accept policy provides the best performance.

Figure 4.22 shows how each accept policy affects the response times seen by the client. This graph shows that the throughput gains of the Accept-50 policy are accompanined by a sizeable increase in response times. For request rates of 17,000 reqs/sec (and above), the Accept-50 policy increases response times by a factor of more than three over the default Accept-1 policy. By 30,000 reqs/sec, this difference has grown to a factor of four. The Accept-100 policy provides even poorer response times, but without any gains in throughput. The Accept-1 policy offers the lowest latencies, although its throughput is significantly lower than that of the Accept-50 policy. The Accept-50 policy improves throughput over the Accept-1 policy, but at the expense of request latency. The Accept-100

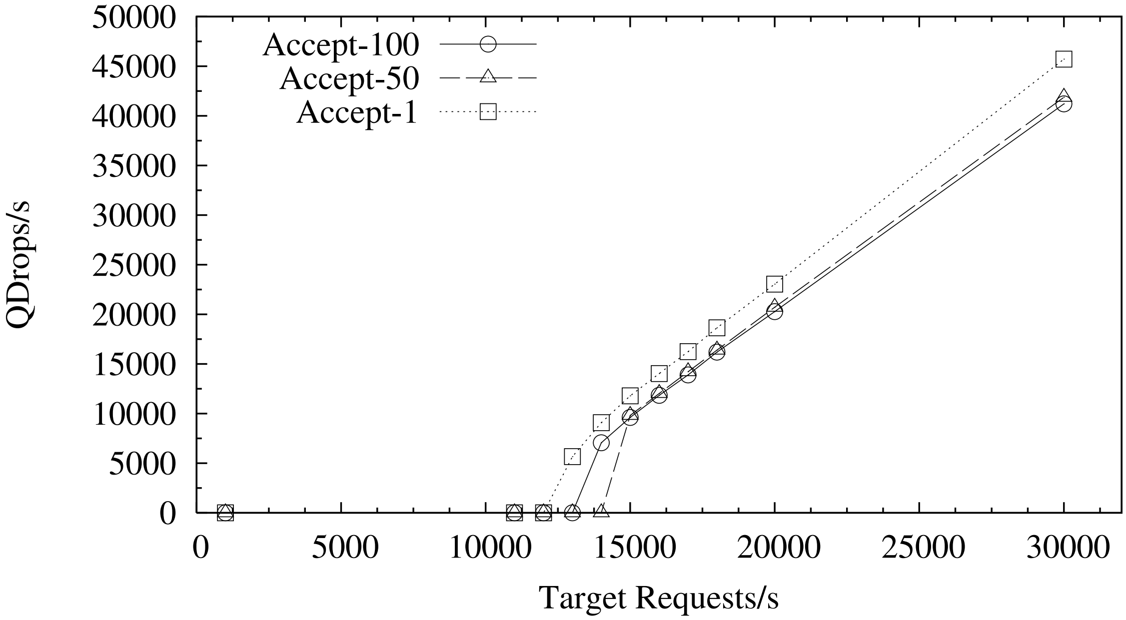

Figure 4.21: *Knot queue drops under the one-packet workload*

policy is inferior as it offers neither low latency nor high throughput.

4.3 Summary

This chapter has examined the impact of accept strategies on server throughput and latency. We have demonstrated that accept strategies can significantly improve peak throughput as well as throughput under overload. In particular, we have realized improved performance for the μserver and TUX under the SPECweb99-like workload, and for all three servers under the one-packet workload. Our experiments have also demonstrated that accept strategies influence throughput to a greater degree under the one-packet workload (as opposed to the SPECweb99-like workload). This arises because the one-packet workload uses each connection for a single short transfer, while the SPECweb99-like workload averages more than seven transfers per connection.

For the μserver, improvements in throughput were realized at the expense of a small increase in server latency. However, for TUX, improvements in throughput were accompa-

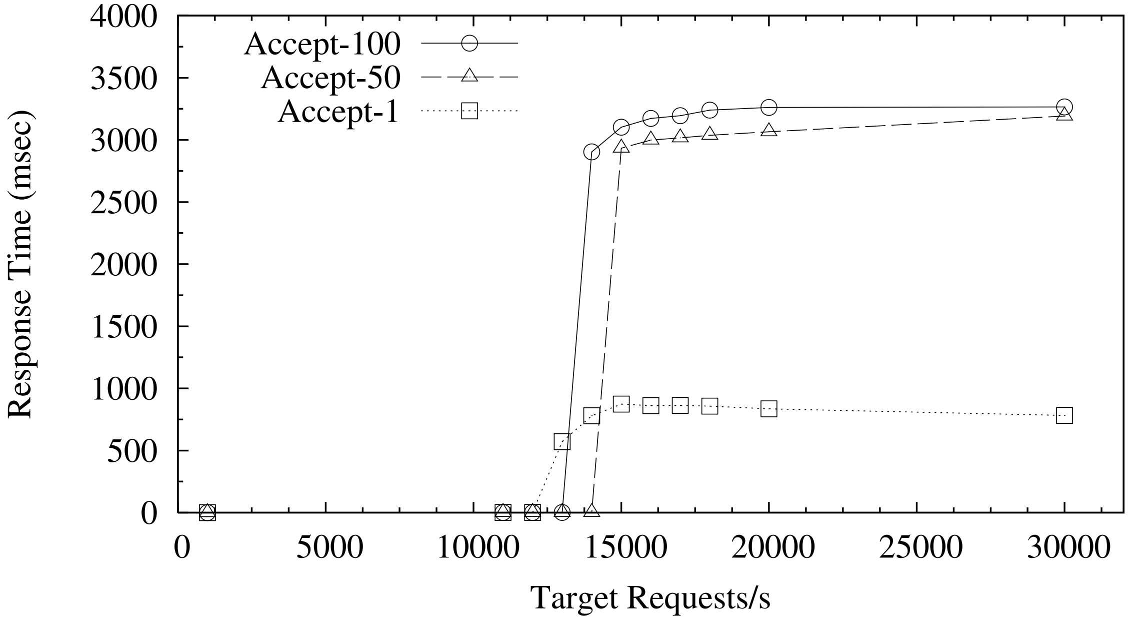

Figure 4.22: *Response times for Knot under the one-packet workload*

nied by a decrease in server latency. This is an especially pleasing result. Improving the performance of Knot under the SPECweb99-like workload proved to be difficult. Despite significant efforts, we failed to understand Knot's behaviour under this workload. However, we were able to improve Knot's performance under the one-packet workload. The increases in throughput were accompanied by increases in server latency.

The next chapter presents a direct comparison of the μserver and TUX under both workloads. We analyze the performance of each server using detailed queue drop counts. We also demonstrate that the user-mode μserver can rival the performance of the kernel-mode TUX server, especially under the SPECweb99-like workload. This result defies conventional wisdom, and differs significantly from the results of an earlier comparison of user-mode and kernel-mode servers.

Chapter 5

Comparing the μserver and TUX

Previous work by Joubert et al. [15] compared the performance of modern user-mode and kernel-mode web servers on Linux and Windows 2000. Their investigation focused on how data movement, event notification, and communication code path affected the performance of both user-mode and kernel-mode servers. They measured the performance of several user-mode and kernel-mode servers using the SPECweb96 [34] and Webstone [20] benchmarks.

They concluded that kernel-mode servers greatly outperformed even highly-optimized user-mode servers on their static, in-memory workloads. This chapter presents our own comparison of user-mode and kernel-mode servers. User-mode servers are represented by the event-driven μserver while kernel-mode servers are represented by TUX. Like Joubert et al., we subject the servers to static, in-memory workload. Our workload is based on the static portion of the SPECweb99 benchmark, and is slightly different from the SPECweb96 workload used by Joubert et al. For example, both workloads are based on four classes of files with nine files in each class. File sizes within each class are identical. However, the SPECweb99-like workload uses a Zipf distribution to decide file access patterns, while SPECweb96 uses a Poisson distribution to determine file access patterns. In addition, SPECweb96 uses only HTTP 1.0 requests while our SPECweb99-like workload uses HTTP 1.1 requests. These differences are not significant enough to prevent a direct comparison of data gathered from each workload. This chapter presents such a comparison, and our findings are remarkably different.

Figures 5.1 and 5.2 compare the performance of the TUX server with the performance of the μserver on the SPECweb99 and one-packet workloads, respectively. These graphs show that the original version of TUX (TUX Accept-Inf) outperforms a poorly tuned version of the user-mode μserver (μserver Accept-1) by as much as 28% under the SPECweb99-like workload, and 84% under the one-packet workload (both at 30,000 requests/sec).

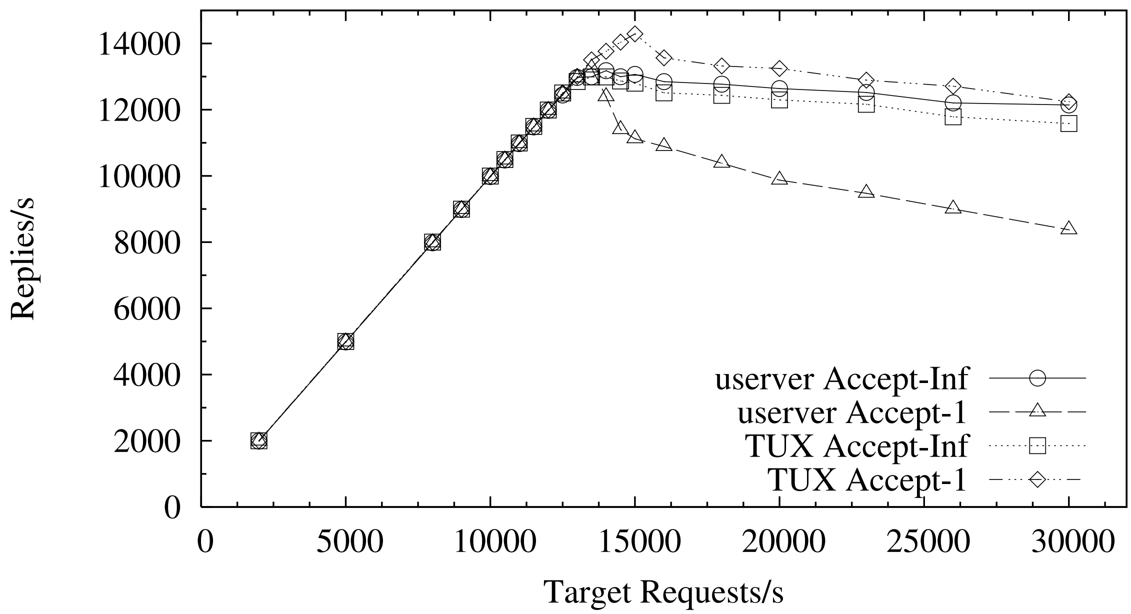

Figure 5.1: *μserver versus TUX throughput under the SPECweb-like workload*

However, the μserver's performance deficit can be ameliorated by adjusting its accept policy. In fact, the more aggressive Accept-Inf policy allows the μserver to actually outperform an unmodified TUX under the SPECweb99-like workload. Under the one-packet workload, the μserver's throughput under the Accept-Inf policy is reasonably close to TUX's throughput under the (default) Accept-Inf policy. Of course, our work has shown that a well chosen accept policy can also improve TUX's performance. Figures 5.1 and 5.2 show that adopting the Accept-1 policy for TUX allows it to regain its performance advantage over the μserver. However, the performance difference is not as large as was reported by Joubert et al. In fact, when both servers use a superior accept policy, the μserver's throughput under the SPECweb99-like workload is always within 10% of TUX's

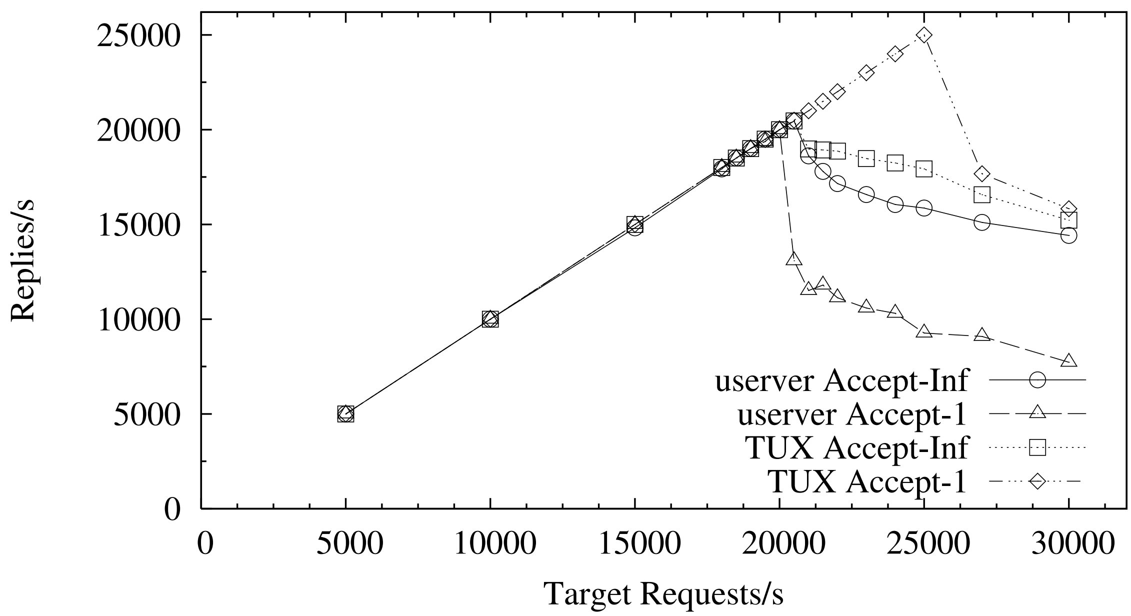

Figure 5.2: *μserver versus TUX throughput under the one-packet workload*

throughput. These results are very promising because they indicate that a well-tuned user-mode server can closely approach the performance of kernel-mode servers. In addition, the performance of the two servers (with tuned accept policies) tends to converge as load increases. We now analyze detailed queue drop statistics that were collected during our experiments. This analysis will help the reader to understand how each accept policy affects server behaviour and performance.

5.1 Understanding Queue Drops

Figures 5.1 and 5.2 contrast a superior accept policy with an inferior one for each server. The performance differences observed in those figures can be partly explained by examining the queue drop data for each policy. We start by examining Figure 5.3 which compares the queue drop rates for the μserver and TUX under the SPECweb99-like workload.

This graph shows that queue drop rates are quite comparable for both servers under this workloads. In fact, the data is tightly clustered at all request rates. However the

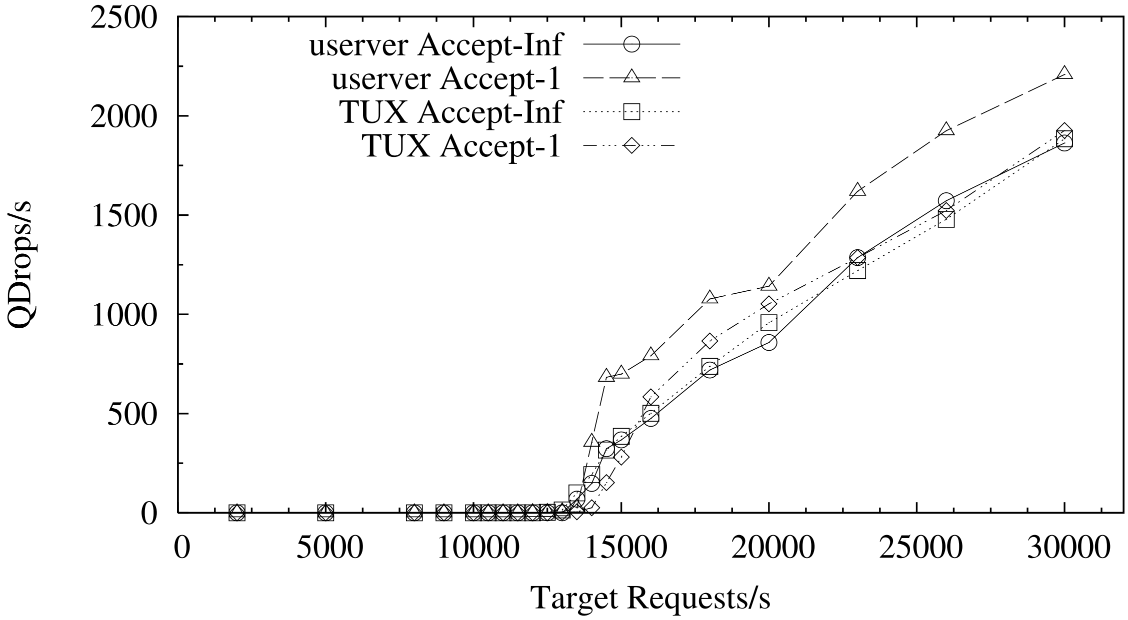

Figure 5.3: *Queue drop rates for µserver and TUX under the SPECweb99-like workload*

graph does show that accept policy does affect queue drop behaviour. In particular, the graph shows that the Accept-Inf policy lowers the µserver's queue drop rate relative to the Accept-1 policy. For TUX, it is the Accept-1 policy that provides lower queue drop rates. However, in both cases the more aggressive accept policy successfully lowers the queue drop rates experienced by the server. Figure 5.4 shows that this qualitative pattern also holds for the one-packet workload, however the differences between the policies are much more pronounced. In fact, there is a stronger correlation between lower queue drop rates and higher throughput under the one-packet workload than under the SPECweb99-like workload. This arises because the one-packet workload sends only one HTTP request per connection, and generates a much higher demand for new connections than the SPECweb99-like workload.

The data in Figure 5.4 is not tightly clustered, and there are significant differences between the servers and between the accept policy for each server. As a result, the remainder of this analysis will focus on the queue drop data obtained from the one-packet workload. In the next section, we explain the conditions that cause a packet to be dropped, and use these conditions to introduce a simple taxonomy for queue drops.

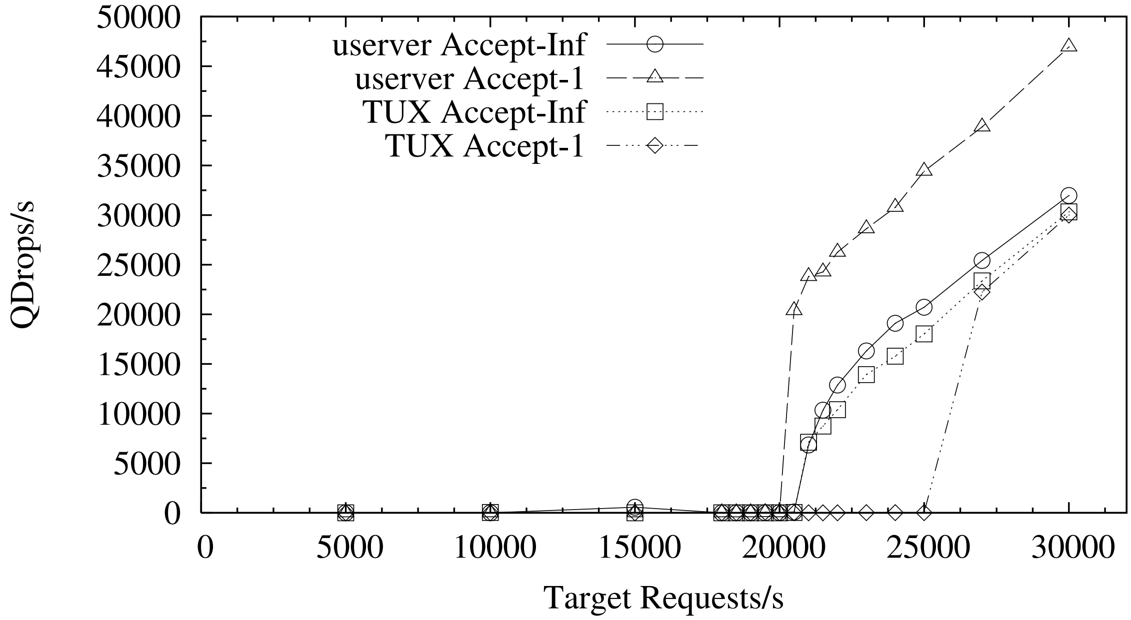

Figure 5.4: *Queue drop rates for µserver and TUX under the one-packet workload*

5.1.1 Categorizing Queue Drops

Under Linux (kernel version 2.4.20-8) a client's connection request may be denied by the server for a variety reasons. The main reasons are listed in Chapter 2. However, the overwhelming majority of packet drops fall into one of the following categories:

- **SynQFull** : The SYN queue is full when the SYN packet arrives.
- **AcceptQFull** : The accept queue is full when the SYN packet arrives.
- **DropRequest** : The SYN queue is 3/4 full when the SYN packet arrives.
- **ListenOverflow** : The accept queue is full when the SYN-ACK packet arrives.

SynQFull drops only occur if TCP_SYNCOOKIES are enabled. Otherwise, SYNs are dropped when the SYN queue is 3/4 full (DropRequests). In our experimental environment, TCP_SYNCOOKIES is not enabled. In order to provide a more complete view of how queue drops impact performance, we added several counters to the Linux kernel's TCP stack. These counters allow us to count the number of packets that are dropped for each of the reasons outlined above. For example, we count the number of packets that are dropped because the Accept queue is full when the SYN packet arrives. Like most kernel counters, our counters are zeroed when the machine first boots, and they accumulate until the next reboot. For easy collection, the counters are integrated with the Linux `/proc` filesystem (at `/proc/net/snmp`). As a result, the counter values can be collected with the well-known `netstat` utility.

These counters allow us to categorize queue drops into one of the four categories outlined above. The queue drop data reported in this section was obtained by re-running certain experiments under our modified kernel. The throughputs obtained under the modified kernel are comparable to those obtained under the standard kernel, which indicates that our kernel modifications do not impact performance in a significant way. Tables 5.1 and 5.2 show detailed queue drop counts for TUX and the μserver when subjected to the one packet workload and a request rate of 27,000 requests per second. At this request rate, both the μserver and TUX are saturated under the one-packet workload.

	AcceptQFull	**DropRequest**	**ListenOverflow**	**Total**
Accept-Inf	2,015,457	681,558	356,740	3,053,755
Accept-1	1,207,341	2,872,130	549,146	4,628,617

Table 5.1: *Breakdown of μserver Queue Drops @ 27,000 reqs/sec under the one-packet workload*

	AcceptQFull	**DropRequest**	**ListenOverflow**	**Total**
Accept-Inf	1,867,979	524,095	412,237	2,804,311
Accept-1	144,472	704,737	519,474	1,368,683

Table 5.2: *Breakdown of TUX Queue Drops @ 27,000 reqs/sec under the one-packet workload*

Table 5.1 shows results for the μserver. It shows that the Accept-Inf policy reduces DropRequests by more than a factor of four. This is especially significant since DropRequests are the largest contributor to total queue drops under the Accept-1 policy. Overall, the Accept-Inf policy lowers the total number of queue drops by 34% over the Accept-1 policy. This reduction in wasted work translates directly into a measurable performance gain. The decrease in the number of dropped SYN packets demonstrates that the Accept-Inf policy successfully reduces the length of the SYN backlog. However, this reduction is partially offset by a higher AcceptQFull count. This can be attributed to the bursty nature of the Accept-Inf policy.

By totally draining the accept queue, the Accept-Inf policy produces large accept-phases where several hundred connections may be accepted consecutively. These long accept phases are eventually followed by long work-phases which are needed to process the new connections. For example, at 27,000 requests/sec the average work-phase under the Accept-Inf policy processes 192.8 connections. In comparison, the average work-phase under the Accept-1 policy processes just 3.6 connections. During these long work-phases, no new connections are accepted and both the SYN queue and the accept queue accumulate entries. However, it is the much shorter accept queue (128 entries versus 1,024 for the SYN queue) that fills first, leading to higher AcceptQFull counts. As noted previously, we did not increase the size of the accept queue because of the large number of installations that currently restrict the size of this queue to a maximum of 128. We plan to examine the relationship between the size of the SYN queue and the accept queue in future work.

The relatively short work-phases of the Accept-1 policy also means that the server does relatively little work per `select` call. As a result, the server must make many more `select` calls to process the same number of connections. Statistics obtained from the μserver indicate that the Accept-1 policy makes 11,071 `select` calls per second, compared to only 251 calls per second for the Accept-Inf policy (both at 27,000 reqs/sec). Clearly, the Accept-1 policy provides a poor amortization of `select` overhead, which hurts performance.

In the TUX case, Table 5.2 reveals that the Accept-1 policy reduces TUX's queue drops by a dramatic 51%. This decrease is mainly due to an order of magnitude reduction in AcceptQFull drops. Clearly, this more aggressive accept strategy is successful in keeping the accept queue relatively empty under the offered load of 27,000 requests/second. As might be expected, performance is quite good. In contrast, the μserver using the Accept-1 policy is relatively slow at removing entries from the SYN queue and the accept queue, and offers quite poor performance at the same request rate. In particular, the μserver suffers a high number of DropRequests, indicating that the SYN queue is not being adequately drained.

Figure 5.5 summarizes the breakdown of queue drops for request rates of 24,000 reqs/sec, 27,000 reqs/sec and 30,000 reqs/sec. Each bar in the chart shows the breakdown of the total number of queue drops into the three relevant categories.

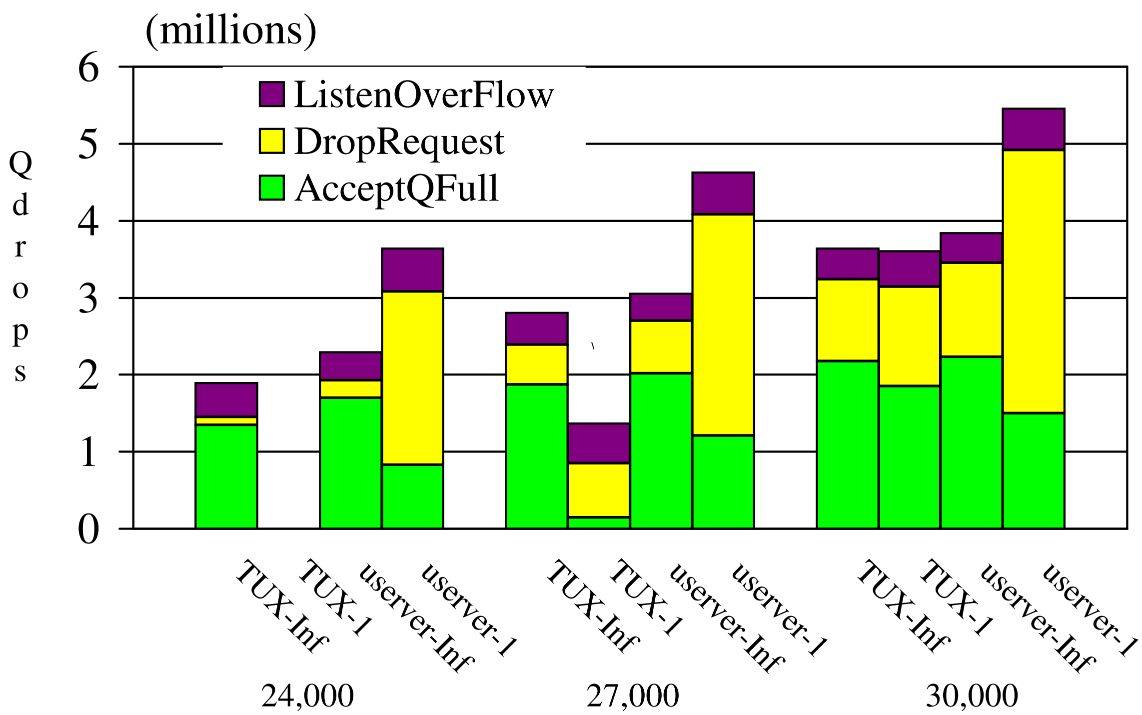

Figure 5.5: *Breakdown of queue drops for TUX and the µserver under one packet workload for selected request rates*

The chart shows that the qualitative pattern observed at 27,000 reqs/sec also holds at other request rates. An aggressive accept policy is able to improve performance by reducing the overall number of queue drops. For the µserver the reduction is primarily due to a marked reduction in the number of DropRequests. This indicates that the Accept-Inf policy is effective in keeping the SYN queue less than 3/4 full. Note that the decrease in DropRequests is partially offset by an increase in AcceptQFull drops. For TUX, the reduction is primarily due to a decrease in the number of AcceptQFull drops, while other categories of queue drop remain largely the same. Clearly, the Accept-1 policy does a better job of keeping the accept queue from overflowing than the Accept-Inf policy

Previous research has investigated techniques for reducing the overheads associated with queue drops. The Lazy Receiver Processing (LRP) architecture [10] provides an improved packet processing model that incorporates early packet demultiplexing, discarding of excess packets, and a lazy processing model. LRP has been shown to provide stable overload behaviour, and increased throughput under heavy loads. Other work by Voigt et al. [39], [38] has investigated kernel-based admission control mechanisms for protecting servers from overload. Their primary mechanism, called TCP SYN policing, limits the acceptance of new TCP SYN packets based on connection attributes. Their chief goal is to limit the amount of new work accepted by the server so as to give priority service to existing connections. TCP SYN policing drops SYNs at an early stage, before a socket has been created for the new connection. In fact, TCP SYN policing could provide an efficient, effective way of controlling a server's accept rate by governing the rate at which new SYNs are admitted to the SYN queue. We believe the aforementioned techniques can complement a well-chosen accept strategy. When used together, they should provide higher throughputs and more stable overload behaviour.

5.2 Comparing Response Times

Figures 5.6 and 5.7 compare the response times of the μserver and TUX under the SPECweb99-like and one-packet workloads respectively.

Figure 5.6 shows that under the SPECweb99-like workload both servers provide very low response times at loads below 14,000 reqs/sec. At loads above 16,000 reqs/sec, it is the μserver under the Accept-1 policy that provides the lowest response times. However, we have seen that this policy provides poor throughput. The Accept-Inf policy improves throughput significantly, at the expense of an increase in average response times. With TUX, the Accept-1 policy is the clear winner as it provides higher throughput and marginally lower response times. Interestingly, TUX exhibits little variation in response time under these two accept policies. The μserver is slightly more sensitive to changes in accept policy. At 30,000 reqs/sec, average response times range from 3.1 seconds (μserver

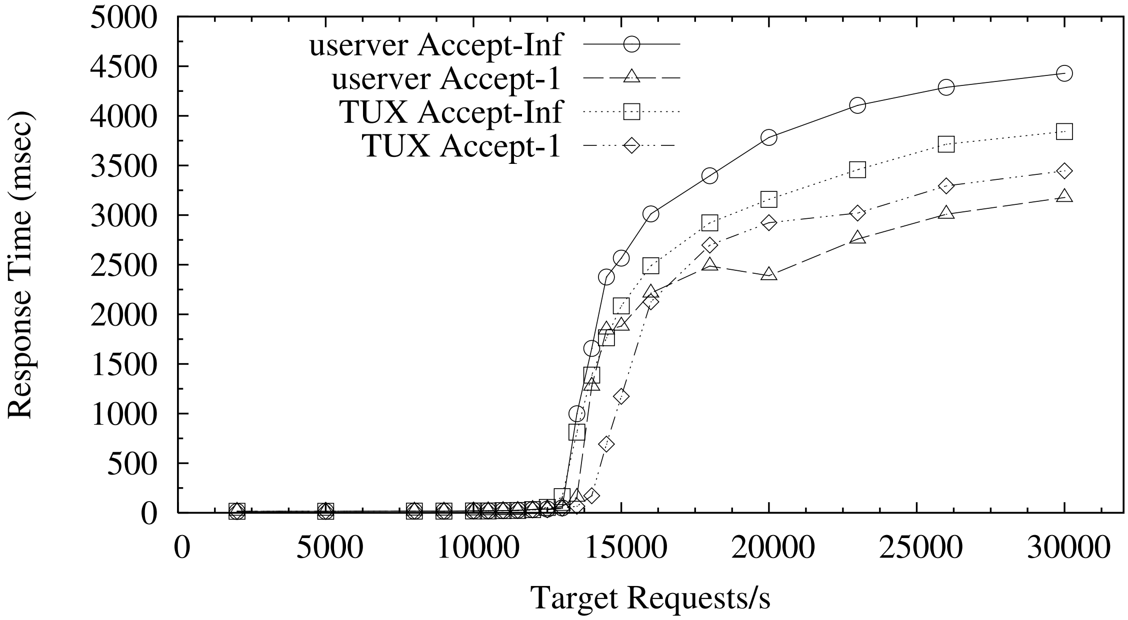

Figure 5.6: *Response times for µserver and TUX under the SPECweb99-like workload*

Accept-1 policy) to 4.6 seconds (µserver Accept-Inf policy). However, both servers provide reasonable response times considering the workload and the environment.

Figure 5.7 compares the response times of both servers under the one-packet workload. This graph presents a few interesting features that were not present in Figure 5.6. Perhaps most interesting is the fact that TUX under the Accept-1 policy is able to maintain very low response times at loads up to 25,000 reqs/sec. In comparison, the µserver under either policy and TUX under the Accept-Inf policy experience sharp spikes in response times at loads approaching 21,000 reqs/sec.

As was the case with the SPECweb99-like workload, the Accept-1 policy provides better response times for the µserver, albeit at the expense of sharply degraded throughput. It is interesting that TUX and the µserver post similar response times under the Accept-Inf policy. Perhaps most important is the fact that both servers provide average response times that are less than 650 milliseconds. Additionally, the response time curves for three of the four policies appear to be converging, and are almost identical by 30,000 reqs/sec.

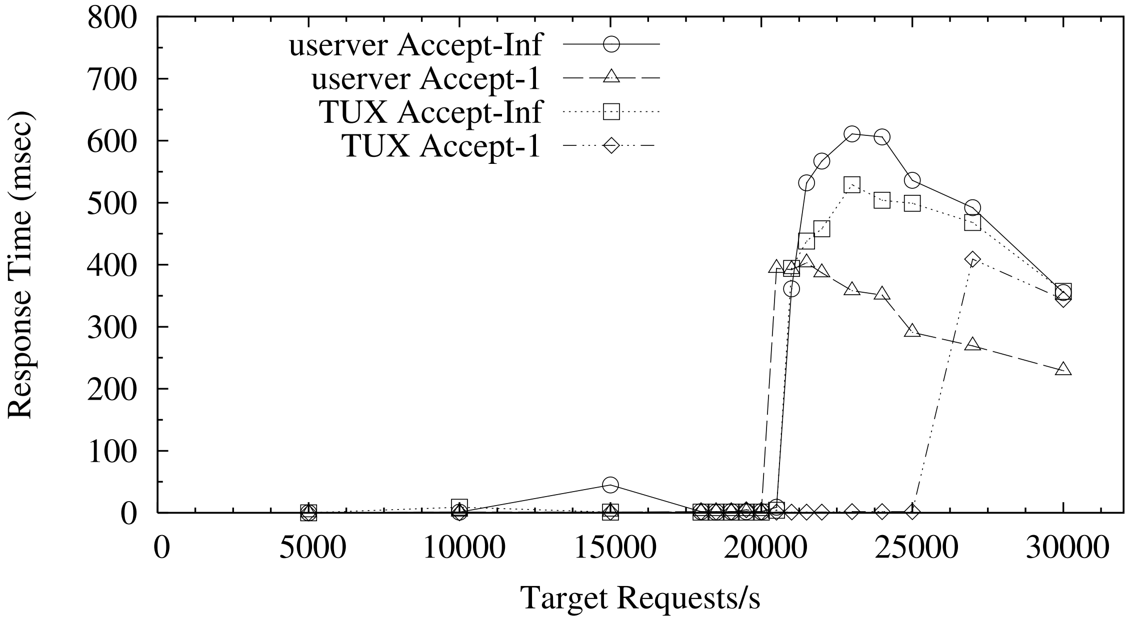

Figure 5.7: *Response times for µserver and TUX under the one-packet workload*

5.3 Differences in Workloads and Environment

Our comparison of user-mode and kernel-mode servers produces considerably different results than the recent work by Joubert et al.. Their research concludes that kernel-mode servers perform two to three times faster than their user-mode counterparts when serving in-memory workloads. Their experiments on Linux demonstrate that TUX (running on a 2.4.0 kernel) achieved 90% higher performance than the fastest user-mode server (Zeus) measured on Linux. While there are undeniable benefits to the kernel-mode architecture (integration with the TCP/IP stack, zero copy network I/O, zero copy disk reads, and eliminating kernel crossings), our comparison of the user-mode µserver with TUX produces considerably different findings. Like Joubert et al., we evaluate the web servers using static, in-memory workloads.

Some of the gains in user-mode performance are due to the zero-copy `sendfile` implementation that is now available on Linux. In separate work we are attempting to quantify the improvements due to zero-copy `sendfile` and the use of the Linux TCP cork and uncork mechanisms. There are also differences in workloads. Specifically, Joubert et al. used an HTTP 1.0 based SPECweb96 workload, while we use an HTTP 1.1 based SPECweb99 workload. Lastly, we note the use of different operating system versions, a different performance metric, different hardware, and possibly different server configurations. In spite of these differences, our work demonstrates that a well chosen accept strategy can greatly improve the performance of the user-mode μserver. Such gains allow the μserver to closely rival the performance of a kernel-mode server under representative workloads.

Chapter 6

Conclusions and Future Work

This thesis examines the impact of connection-accepting strategies on web server performance. We devise and study a simple method for altering the accept strategy of three architecturally different servers: the user-mode, single process, event-driven μserver, the user-mode, multi-threaded, Knot server, and the kernel-mode TUX server. In particular, we modify each of these servers to include an accept-limit that places a strict upper bound on the number of consecutive connections that can be accepted during an accept phase. We use this accept-limit to explore a variety of different accept-strategies for each server.

Our experimental evaluation of different accept strategies exposes these servers to two representative workloads. The SPECweb99-like workload features multiple requests per connection and relatively large transfer sizes. The one-packet workload features small transfer sizes, short-lived connections, and stresses the server's ability to accept new connections. Both workloads involve high connection-rates and genuine overload conditions.

We find that the manner in which each server accepts new connections can significantly affect its peak throughput and overload performance. Our experiments demonstrate that an accept policy that balances the aggressive accepting of new connections with the processing of existing connections can yield noticeable improvements when compared with the base approach used in each server. Under two different workloads, we are able to improve throughput by as much as 10% – 39% for TUX, 0% – 32% for Knot, and 39% – 71% for the μserver. In addition, we demonstrate that a server's accept policy can affect the server's latency. As a result, we point out that researchers in the field of server performance must be aware of the importance of different accept strategies when comparing different types of servers.

Lastly, we present a direct comparison of the user-mode μserver and the kernel-mode TUX server. We show that the gap between user-mode and kernel-mode architectures may not be as large as previously reported. In particular, we find that the throughput of the user-mode μserver rivals that of TUX under the SPECweb99-like workload.

The results presented in this thesis have shown that a server's choice of accept policy can significantly affect its performance. For example, the Accept-Inf policy improved the μserver's throughput under both of our workloads. In contrast, the Accept-1 policy improved TUX's throughput, while the Accept-50 policy improved Knot's throughput under the one-packet workload. It quickly becomes clear that there is no one policy that will work well for all servers under all workloads. As a result, a server must be able to choose the accept policy that best suits its architecture and workload conditions.

A good accept policy will allow the server to minimize the amount of wasted work that it performs. For example, an accept policy performs useless work if it accepts connections that cannot be completed before they time out. Ideally, the accept policy will admit connections at a rate that allows the server maximize its throughput without performing any useless work.

If the server is saturated, then the accept policy should admit connections at the same rate that the server is completing them. This would allow the server to maintain a roughly constant number of open connections. If the accept policy admits new connections faster than they are being completed, then the number of open connections will grow over time. In this situation, the server will be performing unnecessary work as it admits connections that cannot be completed before they time out.

However, if the server accepts new connections slower than they are being completed, the number of open connections will decrease over time. As a result, the server might not complete enough useful work. Alternatively, if the server accepts new connections too slowly, the number of concurrent connections in the server will remain quite small. This was the case with the μserver under the Accept-1 policy. This policy caused the μserver to incur high amounts of `select` overhead. Interestingly, if `select` (or the get-events phase) had low overhead, the Accept-1 policy might be best for the μserver. There would be no need to amortize the overhead of `select`, and the μserver could admit new connections only when there was no useful work to be done. In fact, the μserver would behave quite similarly to TUX. Unfortunately, `select` overhead cannot be ignored, and the Accept-1 policy provides relatively poor throughput.

The Accept-Inf policy allowed the μserver to amortize the overhead of `select` by maintaining a large number of open connections. However, this policy does not place an explicit limit on the number of connections that will be accepted in each accept-phase. In our experiments, this was not a problem, because the server operating system used the default sizes for the SYN queue and the accept queue (1024 and 128 respectively).

However, it is conceivable that the Accept-Inf policy could allow a server to spend a disproportionate amount of time accepting new connections. Such a situation might arise under extreme overload if the operating system provided sufficiently large queues for the buffering of new connection requests.

Under these circumstances, the server may be unable to drain the accept queue because new connections are added as quickly as they are removed. Under these conditions an accept-phase would not terminate unless another system limit is encountered. Another possibility under these conditions is that the Accept-Inf policy would accept new connections much faster than they are being completed. The policy could cause the server to spend too much time accepting, and too little time completing useful work.

The shortcomings of the Accept-1 policy (for user-mode servers) and the Accept-Inf policy suggest that a dynamic policy for determining the accept rate might be best. A simple example would be an accept policy that accepted new connections aggressively when the server is not saturated. After saturation, the policy would accept as many connections as were closed in the previous work phase. This strategy would allow the server to maintain enough open connections to maximize its throughput, while minimizing the amount of wasted work performed.

In the future, we plan to investigate such dynamic strategies in the context of the μserver. The results presented in this thesis show that the μserver offers high performance (comparable to TUX) from a user-mode server that does not require integration with the operating system. We expect that a dynamic accept strategy would amortize the overhead of `select` while eliminating any wasted work due to overly aggressive accepting.

We also hope to examine techniques for making more informed decisions about how to schedule the work that a server performs. We believe that by making more information available to the server we can implement both better and dynamic policies for deciding whether the server should enter a phase of accepting new connections (the accept-phase) or working on existing connections (the work-phase). For example, a simple modification to an existing event notification mechanism (*e.g.*, `select`) would allow the server to determine how many connections are readable, writable, and pending (waiting to be accepted). Such information could be inexpensively obtained, and might assist the server in choosing its next course of action. The server might improve its performance by intelligently scheduling its workload. In addition, this information might allow the server to react dynamically to changing workload conditions.

Lastly, we plan to investigate a wider range of workloads, including large file sets that do not fit in the server's cache and workloads that involve dynamic computation. We expect that these workloads will present new challenges, and new opportunities for exploring the impact of accept strategies on server performance.

Bibliography

[1] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[2] G. Banga and J.C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[3] G. Banga, J.C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[4] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey CA, December 1997.

[5] T. Brecht and M. Ostrowski. Exploring the performance of select-based Internet servers. Technical Report HPL-2001-314, HP Labs, November 2001.

[6] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001.

[7] Computer Science and Telecommunications Board. *The Internet Under Crisis Conditions: Learning from September 11.* The National Academies Press, 2003.

[8] Standard Performance Evaluation Corporation. SPECweb99 frequently asked questions, 2000. Avaliable at http://specbench.org/web99/docs/faq.html.

[9] Frank Dabek, Nickolai Zeldovich, M. Frans Kaashoek, David Mazires, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, September 2002.

[10] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[11] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *6th Annual Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[12] HP Labs. The μserver home page, 2004. Available at http://hpl.hp.com/research/-linux/userver.

[13] J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*. IEEE, November 1997.

[14] Hani Jamjoom and Kang G. Shin. Persistent dropping: An efficient control of traffic aggregates. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[15] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 175–188, 2001.

[16] H.C. Lauer and R.M. Needham. On the duality of operating systems structures. In *Proceedings of the 2nd International Symposium on Operating Systems, IRIA*, October 1978.

[17] Jonathon Lemon. Kqueue – a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.

[18] C. Lever, M. Eriksen, and S. Molloy. An analysis of the TUX web server. Technical report, University of Michigan, CITI Technical Report 00-8, Nov. 2000.

[19] Chuck Lever, Marius Eriksen, and Stephen Molloy. An analysis of the TUX web server. Available at http://citeseer.ist.psu.edu/lever00analysis.html.

[20] Mindcraft Inc. *Webstone - the benchmark for Web servers.* http://-www.mindcraft.com/webstone.

[21] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *The First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[22] Eric Nahum. Deconstructing SPECWeb99. In *Proceedings of the 7th International Workshop on web Content Caching and Distribution*, August 2002.

[23] M. Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000.

[24] J.K. Ousterhout. Why threads are a bad idea (for most purposes), January 1996. Presentation given at the 1996 USENIX Annual Technical Conference.

[25] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[26] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[27] J. Poskanzer. thttpd. Available at http://www.acme.com/software/thttpd.

[28] N. Provos and C. Lever. Scalable network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.

[29] N. Provos, C. Lever, and S. Tweedie. Analyzing the overload behavior of a simple web server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.

[30] Red Hat, Inc. *TUX 2.2 Reference Manual*, 2002.

[31] Marcel-Catalin Rosu and Daniela Rosu. Kernel support for faster web proxies. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[32] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of Internet content delivery systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[33] Amol Shukla, Lily Li, Anand Subramanian, Paul A.S. Ward, and Tim Brecht. Evaluating the performance of user-space and kernel-space web servers. In *Proceedings of the 14th Annual IBM Center for Advanced Studies Conference (CASCON)*, Toronto, Canada, October 2004.

[34] Standard Performance Evaluation Corporation. *SPECWeb96 Benchmark*. http://-www.specbench.org/osg/web96.

[35] Standard Performance Evaluation Corporation. *SPECWeb99 Benchmark*, 1999. http://www.specbench.org/osg/web99.

[36] W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, 1994.

[37] Sun Microsystems. *The /dev/poll interface*. http://docs.sun.com/db/doc/816-0222/-6m6nmlt1h?a=view.

[38] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, June 2001.

[39] Thiemo Voigt and Per Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In *Proceedings of the International Workshop on Protocols For High-Speed Networks*, Berlin, Germany, April 2002.

[40] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea for high-concurrency servers. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[41] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[42] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, Banff, Oct. 2001.

[43] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazieres, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.