USENIX Association

Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

accept()able Strategies for Improving Web Server Performance

Tim Brecht,[*] *David Pariag, Louay Gammo*
School of Computer Science
University of Waterloo
{brecht,db2pariag,lgammo}@cs.uwaterloo.ca

Abstract

This paper 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. Our 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 19% – 36% for TUX, 0% – 32% for Knot, and 39% – 71% for the μserver. Interestingly, the performance improvements realized by the user-mode μserver allow it to obtain performance that rivals an unmodified TUX server.

1 Introduction

Internet-based applications have experienced incredible growth in recent years and all indications are that such applications will continue to grow in number and importance. Operating system support for such applications is the subject of much activity in the research community, where it is commonly believed that existing implementations and interfaces are ill-suited to network-centric applications [4] [30] [23].

In many systems, once client demand exceeds the server's capacity the throughput of the server degrades sharply, and may even approach zero. This is reflected in long (and unpredictable) client wait times, or even a complete lack of response for some clients. Ironically, it is precisely during these periods of high demand that quality of service matters most. Breaking news, changes in the stock market, and even the Christmas shopping season can generate flash crowds or even prolonged periods of overload. Unfortunately, over-provisioning of server capacity is neither cost effective nor practical since peak demands can be several hundred times higher than average demands [1].

Because modern Internet servers multiplex among large numbers of simultaneous connections, much research has investigated modifying operating system mechanisms and interfaces to efficiently obtain and process network I/O events [3] [4] [22] [23] [7]. Other research [20], has analyzed the strengths and weaknesses of different server architectures. These include multi-threaded (MT), multi-process (MP), single process event-driven (SPED) and even a hybrid design called asymmetric multi-process event-driven (AMPED) architecture. More recent work [31] [9] [29] [28] has re-ignited the debate regarding whether to multiplex connections using threads or events in high-performance Internet servers. In addition, an interesting debate has emerged concerning the relative merits of kernel-mode versus user-mode servers, with some research [15] indicating that kernel-mode servers enjoy significant performance advantages over their user-mode counterparts.

In this paper, we examine different strategies for accepting new connections under high load conditions. We consider three architecturally different web servers: the kernel-mode TUX server [24] [16], the event-driven, user-mode μserver [6] [11], and the multi-threaded, user-mode Knot server [29] [28].

We examine the connection-accepting strategy used by each server, and propose modifications that permit us to tune each server's 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.

Our experiments show that:

- Under high loads a server must ensure that it is able to accept new connections at a sufficiently high rate.
- In addition to accepting new connections at a high rate, the server must spend enough time servicing existing connections. That is, a balance must be maintained be-

[*] Some of the research for this paper was conducted while this author was employed by Hewlett Packard Labs.

tween accepting new connections and working on existing connections.

- Each server that we examine can significantly improve its throughput by improving the aforementioned balance.
- Contrary to previous findings, we demonstrate that a user-level 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.

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 1.

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 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 1 does not handle persistent or pipelined connections (although all servers used in our experiments handle persistent connections).

Many 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 the server must be able to efficiently multiplex several thousand simultaneous connections [4] 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 briefly describe related work in these areas.

2.1 Operating System Improvements

Significant research [3] [2] [4] [19] [22] [23] [7] has been conducted into improving web server performance by improving both operating system mechanisms and interfaces for obtaining information about the state of socket and file descriptors. These studies have been motivated by the overhead incurred by `select`, `poll`, and similar system calls under high loads. As a result, much research has focused on developing improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data that needs to be copied between user space and kernel space or by reducing the amount of work that must be done in the kernel (e.g., by only delivering one signal per descriptor in the case of `sigwaitinfo`). Other work [21] has focused on reducing data copying costs by providing a unified buffering and caching system.

In contrast to previous research on improving the operating system, this paper presents strategies for accepting new connections which improve server performance under existing operating systems, and which are relevant to both user-mode and kernel-mode servers.

2.2 Server Application Architecture

One approach to multiplexing a large number of connections is to use a SPED architecture, which uses a single process in conjunction with non-blocking socket I/O and an event notification mechanism such as `select` to deliver high throughput, especially on in-memory workloads [20]. The event notification mechanism is used to determine when a network-related system call can be made without blocking. This allows the server to focus on those connections that can be serviced without blocking its single process.

Of course, a single process cannot leverage the processing power of multiple processors. However, in multiprocessor environments multiple copies of a SPED server can be used to obtain excellent performance [32].

The multi-process (MP) and multi-threaded (MT) models [20] offer an alternative approach to multiplexing simultaneous connections by utilizing a thread (or process) per TCP connection. In this approach, connections are multiplexed by context-switching from a thread that can no longer process its connection because it will block, to another thread that can process its connection without blocking. Unfortunately threads and processes can consume large amounts of resources and architects of early systems found it necessary to restrict the number of executing threads [13] [4].

The Flash server implements a hybrid of the SPED and MP models called AMPED (asymmetric multi-process event-driven) architecture [20]. This architecture builds on the SPED model by using several helper processes to perform disk accesses on behalf of the main event-driven process. This approach performed very well on a variety of workloads and outperformed the MP and MT models.

More recent work has revived the debate concerning event-driven versus multi-threaded architectures. Some papers [31] [9] [32] conclude that event-driven architectures afford higher-performance. Others [28] [29] argue that highly efficient implementations of threading libraries allow high performance while providing a simpler programming model.

Our work in this paper uses servers that are implemented using both event-driven and multi-threaded architectures. We demonstrate that improved accept strategies can increase throughput in either type of server.

2.3 Kernel-mode Servers

In light of the considerable demands placed on the operating system by web servers, some people [24] [12] have argued that the web server should be implemented in the kernel as an operating system service. Recent work [15] has found that there is a significant gap in performance between kernel-mode and user-mode servers when serving memory-based workloads. Our findings in this paper challenge these results. In fact, on a static, in-memory, SPECweb99-like workload the μserver's performance compares very favourably with that of the kernel-mode TUX server.

2.4 Accept Strategies

In early web server implementations, the strategy for accepting new connections was to accept one connection each time the server obtained notification of pending connection requests. Recent work by Chandra and Mosberger [7] discovered that a simple modification to a `select`-based web-server (with a stock operating system) outperformed operating system modifications they and other researchers [22] had performed in order to improve event dispatch scalability. They referred to this server as a *multi-accept* server. Upon learning of a pending connection, this server attempts to accept as many incoming connections as possible by repeatedly calling `accept` until the call fails (and the `errno` is set to `EWOULDBLOCK`) or the limit on the maximum number of open connections is reached. This multi-accept behaviour means that the server periodically attempts to drain the entire accept queue. Their 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.

In this paper, we explore more representative workloads and demonstrate that their multi-accept approach can overemphasize the accepting of new connections while neglecting the processing of existing connections. The resulting imbalance leads to poor performance.

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. We demonstrate that a carefully tuned accept policy can significantly improve performance across all three server architectures.

More recent work [28] [29] has also noted that the strategy used to accept new connections can significantly impact performance. Our work specifically examines different strategies applied to a variety of servers in order to understand how to choose a good accept strategy.

3 Improving Accept Strategies

In order for a client to send a request to the server it must first establish a TCP connection to the server. This is done by using the TCP three-way handshake [26]. Once the three-way handshake succeeds the kernel adds a socket to the *accept queue* (sometimes referred to as the listen queue) [5]. 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.

In Linux, 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.

If the server accepts new connections too slowly, then either the accept queue or the SYN queue will quickly fill up. If either queue fills, all new connection requests will be dropped. Such queue drops are problematic for both 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 process packets and complete the TCP three-way handshake, only to discover that the connection must be dropped. For these reasons, queue drops should be avoided whenever possible.

Our work in this paper concentrates on improving accept strategies to enable servers to accept and process more connections. Note that this is quite different from simply reducing the number of queue drops (i.e., failed connection attempts) 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 finding a balance between accepting new connections and processing existing connections.

4 The Web Servers

This section describes the architecture of each of the servers investigated as well as the procedure each uses for accepting new connections. We also describe any modifications we have made to the base server behaviour.

4.1 The μserver

The micro-server (μserver) [6] is a single process event-driven web server. Its behaviour can be carefully controlled through the use of a large number of command-line parameters, which allow us to investigate the effects of several different server configurations using a single web-server. 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 process multiple connections concurrently.

The μserver operates by tracking the state of each active connection (states roughly correspond to the steps in Figure 1). 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`. The second phase (called the *accept-phase*) is entered if `select` reports that connections are pending on the listening socket. The third phase (called the *work-phase*) iterates over each of the non-listening connections that have events of interest that can be processed without blocking. Based on the event-type and the state of the connection, the server calls the appropriate function to perform the required work. 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. That is, it only works on connections when `select` informs it that work can proceed on that connection without blocking.

The μserver is based on the multi-accept server written by Chandra and Mosberger [7]. 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 paper. The same name is also used to refer to similar modifications to Knot and TUX. Parameter values range from one to infinity (*Inf*). An accept-limit of one forces the server to accept a single connection in each accept-phase, while Inf causes the server to accept all currently pending connections.

Our early investigations [6] 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 their accept mechanisms.

4.2 Knot

Knot [28] is a multi-threaded web server which makes use of the Capriccio [29] 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 features a number of different thread schedulers, including a resource-aware scheduler which adapts its scheduling policies according to the application's resource usage. Knot operates in one of two modes [28] which are referred to as Knot-C and Knot-A.

Knot-C uses a thread-per-connection model, in which the number of threads is fixed 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.

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. These results agree with findings reported by the authors of Knot [28], and as a result we focus our tuning efforts 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 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 the accepted connections proceeds. Otherwise the thread is put to sleep until a new connection request arrives. After accepting new connections, each thread fully processes the accepted connections before admitting any new connections. Therefore, in our modified version of Knot each thread oscillates between an accept-phase and a work-phase. As in the μserver, the accept-limit parameter ranges from 1 to infinity. The rest of this paper uses the accept-limit parameter to explore the performance of our modified version of Knot-C. Note that when the accept-limit is set to 1 our modified version of Knot operates in the same fashion as the original.

4.3 TUX

TUX [24] [16] 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 relatively low overhead when 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 is important in light of the large number of system calls needed to process a single HTTP request.

A look at the TUX source code provides detailed insight into TUX's structure. TUX's processing revolves around two queues. 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 oscillates between an accept-phase and a work-phase. It does not require a getevents-phase because it has access to the kernel data structures where event information is available. In the accept-phase 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 events can be added to each queue while TUX removes and processes them.

We modified TUX to include an accept-limit parameter, which governs the number of connections that TUX will accept consecutively before leaving the accept-phase. 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` file system, 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. This parameter gives us 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 Section 6.

It is also important to understand that the accept-limit parameter does not control the accept rate, 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 connections accepted while in that phase. The amount of time spent in the work and getevent phases determines the frequency with which the accept-phase is entered.

5 Experimental Methodology

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.) were 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 interesting accept policies and differences. The following sections describe our experimental environment and the parameters used to configure each server.

5.1 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 Gbps 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 Gbps switch. Since the server has two cards, we avoid network bottlenecks by partitioning the clients into different subnets. In particular, the first four clients communicate with the server's first ethernet card, while the remaining four use 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 Gbps Ethernet cards. The clients are dual-processor Pentium III machines running at 550 MHz. Each client has 256 MB of RAM, an EIDE disk, and one Intel e1000 Gbps 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 Gbps 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 Gbps network, all machines are connected by 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. Instead, each graph presents data gathered from a single cluster. Ideally, we would use one cluster for all our experiments, but the number of experiments required necessitated the use of two clusters.

5.2 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 [24]. 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). This adjustment also eases comparison and understanding of accept-limit-Inf strategies.

Additionally, both TUX and the μserver use limits of 15,000 simultaneous connections. In the μserver case this is 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 case we used 1,000 threads. Although we have not extensively tuned Knot we have noticed that as long as the number of threads was not excessively small or large, the performance of Knot-C was not greatly impacted by the number of threads. Note that in this architecture the number of threads used also limits the maximum number of simultaneous connections. When the accept-limit modification is added to Knot it permits several connections per thread to be open, thus increasing this limit.

Finally, logging is disabled on all servers and we ensure that all servers can cache the entire file set. This ensures that differences in server performance are not due to caching strategies.

6 Workloads and Experimental Results

This section describes the two workloads used in our experiments, and discusses the results obtained with each of the three servers. Our results show that the accept strategy significantly impacts server performance for each server.

6.1 SPECweb99-like Workload

The SPECweb99 benchmarking suite [25] is a widely accepted tool for evaluating web server performance. However, the suite is not without its flaws. The SPECweb99 load generators are unable to generate loads that exceed the capacity of the server. The problem is that the SPECweb99 load generator will only send a new request once the server has replied to its previous request. Banga et al. [5] show that under this approach the clients' request rates are throttled by the server. As such, the clients are unable to overload the server.

We address this problem by using *httperf*, an http load generator that is capable of generating overload [17]. httperf avoids the naive load generation scheme by implementing connection timeouts. Every time a connection to the server is initiated, a timer is started. If the connection timer expires before the connection is established and the HTTP transaction completes, the connection is aborted and retried. This allows the clients to generate loads that exceed the server's capacity. We use httperf in conjunction with a SPECweb99 file set and a session log file that we have constructed to mimic the SPECweb99 workload. Although our traces are synthetic, they are carefully generated to accurately recreate the file classes, access patterns, and the number of requests issued per persistent HTTP 1.1 connection used in the static portion of SPECweb99 [25].

In all experiments, the SPECweb99 file set and server caches are sized so that the entire file set fits in main memory. This is done to eliminate differences between servers due to differences in caching implementations. While an in-memory workload is not entirely representative, it does permit us to compare our results with those of Joubert et al. [15], who used an in-memory SPECweb96 workload to compare the performance of kernel-mode and user-mode servers.

Figure 2 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. This graph shows that a larger accept-limit can significantly improve performance in the μserver, especially under overload. In fact, at the extreme target load of 30,000 requests/sec, the accept-limit-Inf policy outperforms the accept-limit-1 policy by 39%.

Statistics collected by the μserver provide insights that confirm the benefits of the high accept-limit value. At a target load of 30,000 requests/sec, the accept-limit-Inf server accepts an average of 1,571 new connections per second. In comparison, the accept-limit-1 server averages only 1,127 new connections per second (28% fewer). This difference is especially significant when we consider that each SPECweb99 connection is used to send an average of 7.2 requests. Figure 3 shows that in all cases the higher accept-rates

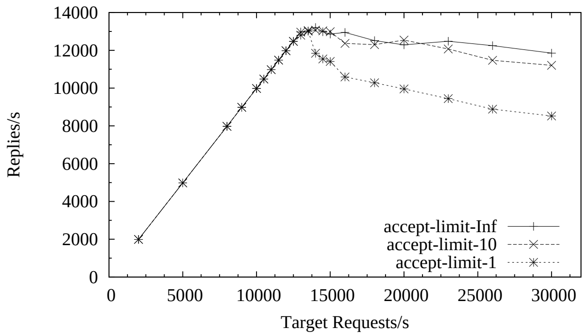

Figure 2: *μserver performance under SPECweb99-like workload*

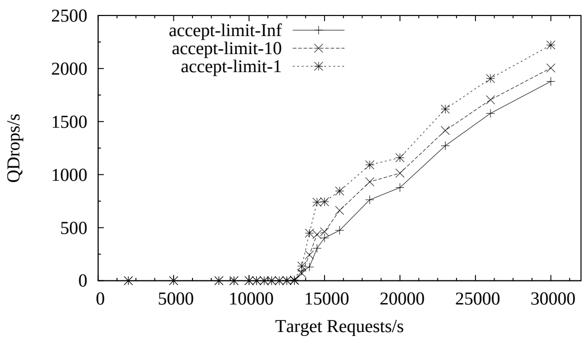

Figure 3: *μserver queue drops/sec under SPECweb99-like workload*

result in lower queue drop rates (QDrops/s). The lower drop rates mean that less time is wasted in the processing of packets that will be discarded, and more time can be devoted to processing client requests. As seen in Figure 2, this translates into a healthy improvement in throughput.

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* is 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.

For the Knot server, we experimented with a variety of different accept strategies. The results are summarized in Figures 4 and 5. Figure 4 illustrates the throughput obtained using 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-limit-1 policy produce indistinguishable results. To reduce clutter, we omit results for the original version of Knot.

Higher accept-limits (10, 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 in Knot reported that at a load of 20,000 requests/sec, the accept-limit-100 policy accepts new connections 240% faster (on average) than the accept-limit-1 (default) server. Further evidence is provided in Figure 5 which shows that the accept-limit-50 and accept-limit-100 servers enjoy significantly lower queue drop rates than their less aggressive counterparts.

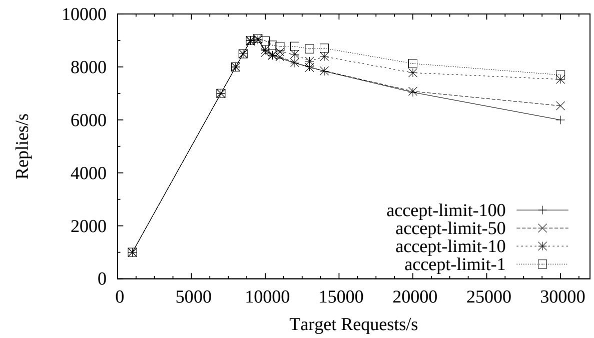

Figure 4: *Knot performance under SPECweb99-like workload*

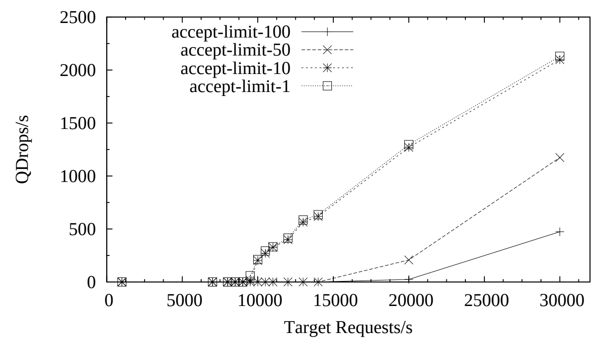

Figure 5: *Knot queue drops/sec under SPECweb99-like workload*

Unfortunately, the higher accept rates (and lowered queue drop rates) do not improve performance. On the contrary, performance suffers. Statistics reported by Knot show that with an accept-limit of 50 or higher, the number of concurrent connections in the server grows quite sharply. We believe 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, more aggressively accepting new connections does not improve Knot's performance. These findings agree with previously published results [28] in which overly aggressive accepting also hurt Knot's performance.

In Figure 6 we show that the accept-limit parameter can be used to improve TUX's performance. The accept-limit-Inf policy corresponds to TUX's default accept behaviour (draining the accept queue). The accept-limit-50 policy allows TUX to consecutively accept up to 50 connections, while the accept-limit-1 policy limits TUX to accepting a single con-

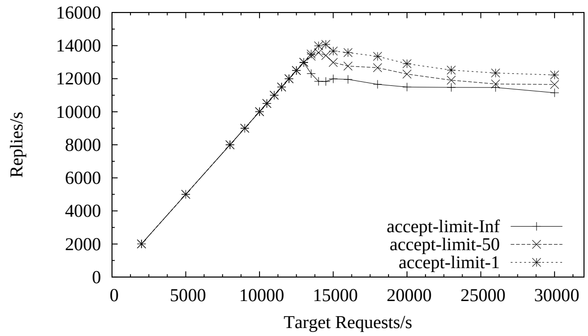

Figure 6: *TUX performance under SPECweb99-like workload*

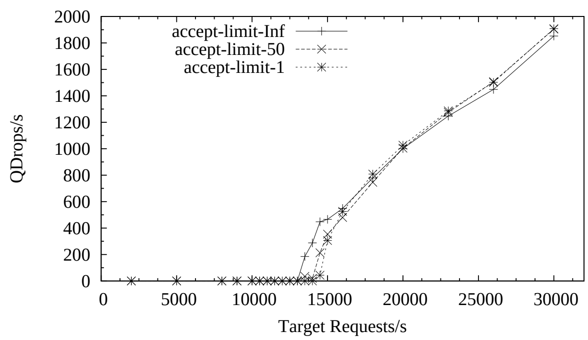

Figure 7: *TUX queue drops/sec under SPECweb99-like workload*

nection in each accept-phase. Figure 6 shows that the accept-limit-1 policy results in a 12% increase in peak throughput, and a 19% increase in throughput at 14,500 reqs/sec. Surprisingly, our server-side instrumentation shows that an accept-limit-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 1 formalizes this simple mathematical relationship. 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. In our experiments, $t_{elapsed}$ is essentially a constant.

$$AcceptRate = \frac{N_{phases} C_{avg}}{t_{elapsed}} \quad (1)$$

In TUX, lowering the accept-limit has two effects. Firstly, C_{avg} decreases since each accept-phase is shortened. 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. We found that for low accept-limits, TUX accepted fewer connections in each accept-phase, but entered its accept-phase more frequently (because the low accept-limit shortened its work-phase). On balance, lower accept-limits lead to a higher accept rate.

Interestingly, the accept-limit parameter has a very different effect on TUX and the μserver in spite of the fact that both are event-driven servers with accept-phases and work-phases. Because of this similarity, Equation (1) applies equally to both servers. In the μserver, lowering the accept-limit parameter also lowers C_{avg}, and increases N_{phases}. However, in this case the increase in N_{phases} is unable to compensate for the decrease in C_{avg}. As a result, the μserver's accept-rate falls when its accept-limit is lowered.

This analysis shows that in spite of the same *relative* change in N_{phases} and C_{avg}, the *magnitude* of each change is quite different in the μserver and TUX. The difference in magnitude arises because of the get-events phase that exists in the μserver but not in TUX. In the μserver each accept-phase is preceded by a get-events phase (essentially a call to `select`). Increasing the number of accept-phases also increases the number of `select` calls. This adds an overhead to each accept-phase, and limits the μserver's ability to perform more accept-phases. In comparison, TUX incurs no extraneous overhead for extra accept-phases.

Figure 7 shows TUX's queue drop rates for each accept policy. In this case the largest differences in drop rates are seen in the 12,000 to 15,000 requests per second range where there are also the largest differences in reply rates.

6.2 One-packet Workload

In the aftermath of the September 11th 2001 terrorist attacks, many on-line news services were flooded with requests. Many services were rendered unavailable, and even large portals were unable to deal with the deluge for several hours. The staff at CNN.com resorted to replacing their main page with a small, text-only page containing the latest headlines [8]. In fact, CNN sized the replacement page so that it fit entirely in a single TCP/IP packet. This clever strategy was one of the many measures employed by CNN.com to deal with record-breaking levels of traffic.

These events reinforce the need for web servers to efficiently handle requests for small files, especially under extreme loads. With this in mind, we have designed a static workload that tests a web server's ability to handle a barrage of short-lived connections. The workload is simple; all requests are for the same file, issuing one HTTP 1.1 request per connection. The file is carefully sized so that the HTTP headers and the file contents fill a single packet. This resembles the type of requests that would have been seen by CNN.com on September 11.

Obviously, this workload differs from the SPECweb99-like workload in several key respects. For instance, it places much less emphasis on network I/O. Also, because a small file is being requested with each new connection it stresses a server's ability to handle much higher demand for new connection requests. We believe that when studying servers under high loads that this is now an interesting workload in its own right. We also believe that it can provide valuable insights that may not be possible using the SPECweb99-like workload. For more discussion related to the workloads used in this paper see Section 7.

Figure 8 shows the reply rate observed by the clients as the load (target requests per second) on the server increases. This graph shows that the accept-limit-Inf and accept-limit-10 options significantly increase throughput when compared with the naive accept-limit-1 strategy. This is because these servers are significantly more aggressive about accepting new connections than the accept-limit-1 approach. Interestingly, the accept-limit-10 strategy achieves a slightly higher peak than the accept-limit-Inf strategy, although it experiences larger decreases in throughput than accept-limit-Inf as the load increases past saturation. This indicates that better performance might be obtained by dynamically adjusting the accept strategy (this is something we plan to investigate in future research).

The differences in performance between the accept-limit-10 and accept-limit-Inf policies can be seen by examining their ability to accept new connections. Figure 9 shows the queue drop rates for the different accept strategies. Here we see that the μserver operating with an accept-limit of 10 is better able to accept new connections. In fact it is able avoid significant numbers of queue drops until 23,000 requests per second. On the other hand the accept-limit-Inf option experiences significant numbers of queue drops at 21,500 requests per second. Both of these points correspond to their respective peak rates.

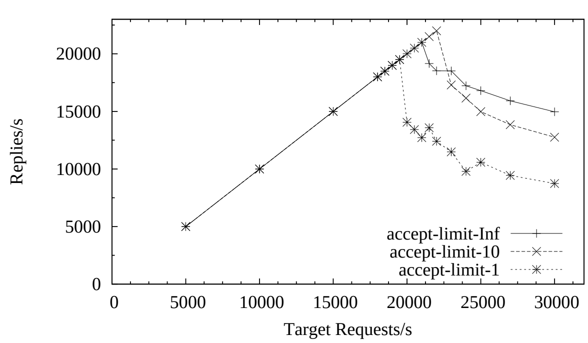

Figure 8: μ*server performance under one-packet workload*

Figure 9 also shows that the accept-limit-1 option does a good job of accepting new connections until a target request rate of 20,000 requests per second. At that point it is unable to keep up with the demand for new connections. The result is that the queue drop rate is 17,832 drops per second, and the reply rate is 14,058 replies per second. Significant expense is incurred in handling failed connection requests. If the server can accept those connections, it can improve performance as long as existing connections are not neglected.

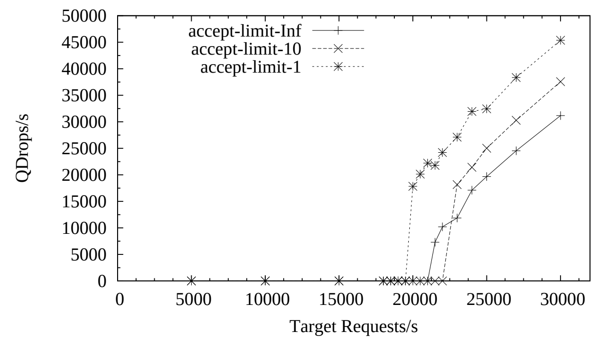

Figure 9: μ*server queue drops/sec under one-packet workload*

Interestingly, the total of these two rates (17,832 + 14,058 = 31,890) exceeds the target request rate of 20,000 requests per second. This is because when a client is attempting to establish a TCP connection using the three-way handshake, if the client does not receive a SYN-ACK packet in response to the SYN packet it sends to the server, it will eventually time-out and retry, which leads to several queue drops per connection.

Using this one-packet workload we are able to increase the μserver's peak throughput from 19,500 replies per second using the naive accept strategy (accept-limit-1) to 22,000 replies per second using the accept-limit-10 strategy. This is an improvement of 13%. More importantly, the accept-limit-Inf strategy improves performance versus the naive strategy by as much as 65% at 21,000 requests per second and 71% at 30,000 requests per second.

Figure 10 shows the reply rate versus the target request rate for the TUX server. As with the SPECweb99-like workload, limiting the number of consecutive accepts increases TUX's accept rate. This can be seen by comparing the queue drop rates (QDrops/sec) in Figure 11 for the different TUX configurations examined. In TUX, the accept-limit-1 strategy does the best job of accepting new connections resulting in the lowest queue drop rates of the configurations examined. This translates directly into the highest throughput.

Recall that the accept-limit-Inf strategy corresponds to the original TUX accept strategy. In this case the improved accept-limit-1 strategy results in a peak reply rate of 22,998 replies per second compared with the original, whose peak is at 20,194 replies per second. This is an improvement of 14%. Additionally there is an improvement of 36% at 23,000 requests per second.

We believe further improvements are possible. However, the simple method we used to modify TUX does not permit

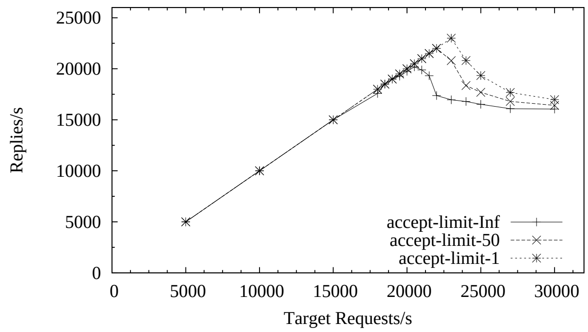

Figure 10: *TUX performance under one-packet workload*

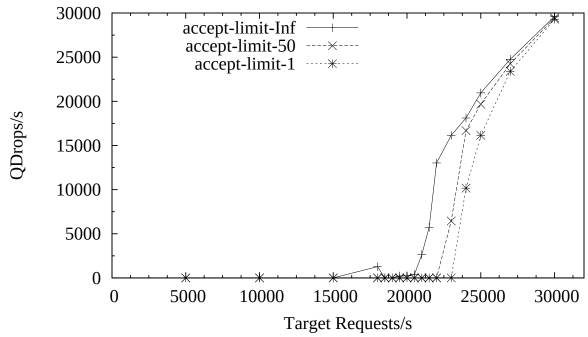

Figure 11: *TUX queue drops/sec under one-packet workload*

us to accept fewer than one connection per 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 in the number of entries in `work_pending` queue. With this information, a more informed decision can be made about 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 events 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.

For this one packet workload, Knot also benefits from tuning its accept policy. Figure 12 shows an interesting spectrum of accept policies. We observe that the accept-limit-50 strategy noticeably improves throughput when compared with the original accept strategy. Firstly, peak throughput is increased by 17% from 12,000 to 14,000 replies per second. Secondly, the throughput is increased by 32% at 14,000 requests per second and 24% at 30,000 requests per second.

Interestingly, increasing the accept-limit value too much (for example to 100) can result in poor performance. In comparing the accept-limit-100 strategy with the accept-limit-1 (default) strategy, we observe that the former obtains a slightly higher peak. However, throughput degrades significantly once the saturation point is exceeded. Figure 13 shows how the connection failure rates are impacted by the changes in the accept strategy. Here we see that the accept-limit-100 version is able to tolerate slightly higher loads than the original before suffering from significant connection failures. The accept-limit-50 version is slightly better, and in both cases peak throughput improves. At request rates of 15,000 and higher the accept-limit-50 and accept-limit-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 difference in performance. The statistics printed by the Knot server show that at 15,000 requests/sec the accept-limit-50 policy operates with approximately 25,000 active connections, while the accept-limit-100 policy is operating with between 44,000 to 48,000 active connections. One possible explanation for the difference in performance is that the overhead incurred by `poll` becomes prohibitive as the number of active connections climbs. These experiments also highlight that a balanced accept policy provides the best performance.

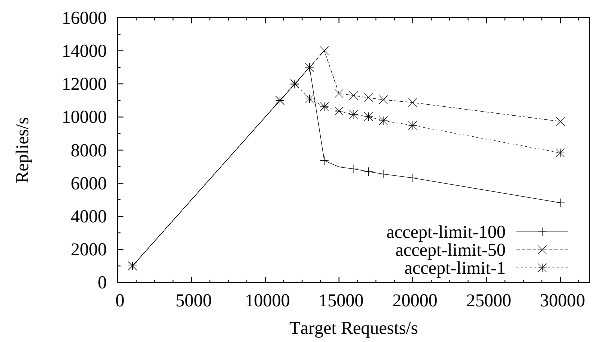

Figure 12: *Knot performance under one-packet workload*

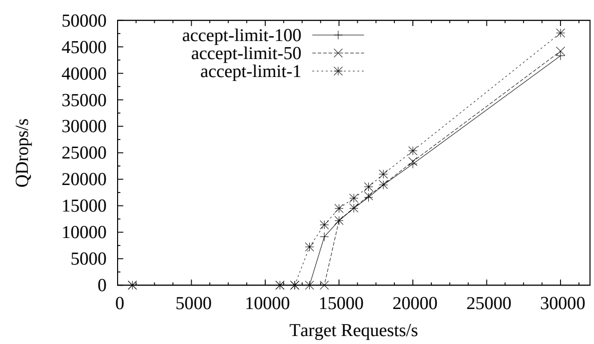

Figure 13: *Knot queue drops/sec under one-packet workload*

6.3 Comparing the μserver and TUX

Figures 14 and 15 compare the performance of the TUX server with the performance of the μserver under the SPECweb99 and one packet workloads respectively. These graphs show that the original version of TUX (accept-limit-Inf) outperforms a poorly tuned (accept-limit-1) version of the user-mode μserver by as much as 28% under the SPECweb99-like workload and 84% under the one-packet workload (both at 30,000 requests/sec). However, the performance gap is greatly reduced by adjusting the μserver's accept policy. As a result we are able to obtain performance that compares quite favourably with the performance of the unmodified TUX server under both workloads.

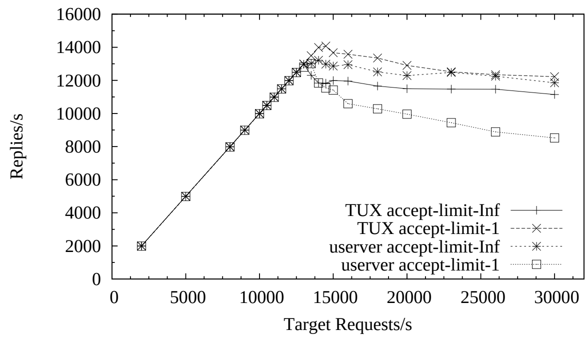

Figure 14: *μserver versus TUX performance under SPECweb-like workload*

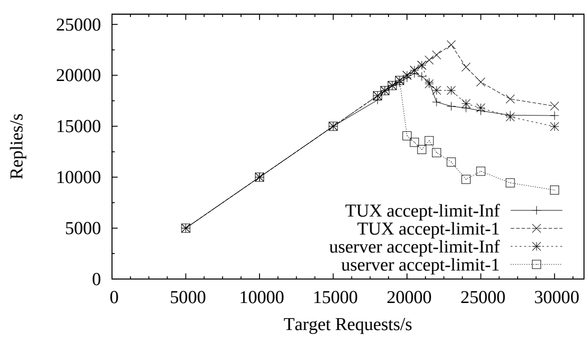

Figure 15: *μserver versus TUX performance under one-packet workload*

Figure 16 contrasts a superior accept policy with an inferior one for each server. The observed performance differences can be partly explained by examining the queue drop counts for each policy. Under Linux (kernel version 2.4.20-8) a client's connection request may be denied by the server for a variety reasons (in the order listed), including:

- **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

To provide a more complete view of how queue drops impact performance, we added several counters to the Linux kernel. These allow us to categorize queue drops according to the cases outlined above. Queue drop data was obtained by re-running selected experiments under our modified kernel. The throughputs obtained under the modified kernel are comparable to those obtained under the standard kernel. Figure 16 shows detailed queue drop counts for TUX and the μserver under the one packet workload at request rates of 24,000, 27,000 and 30,000 requests/sec. In this figure, the SynQFull count is always zero, and has been omitted.

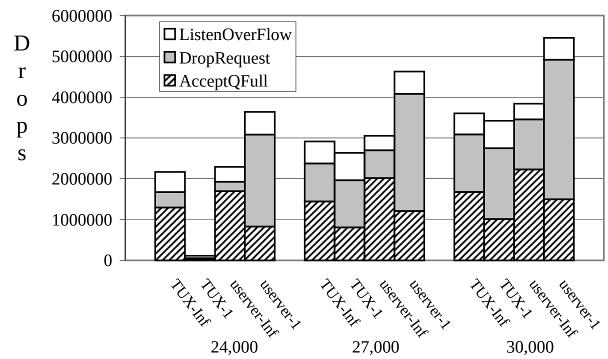

Figure 16: *Different types of queue drops for TUX and the μserver under one packet workload for selected request rates*

The results for the μserver at 24,000 requests per second show that the high-performance accept-limit-Inf policy (userver-Inf) significantly reduces the number of DropRequests and the overall number of queue drops when compared with the accept-limit-1 policy. The decrease in the number of dropped SYN packets demonstrates that the accept-limit-Inf policy successfully reduces the size of the SYN backlog. However, this reduction is partially offset by a higher AcceptQFull count. The latter can be attributed to the bursty nature of the accept-limit-Inf policy.

By totally draining the accept queue, the accept-limit-Inf policy produces large *accept-phases* where several hundred connections may be accepted consecutively. These long accept phases are followed by long *work-phases* which are needed to process connections. For example, at 24,000 requests/sec the average work-phase under the accept-limit-Inf policy processes 184.9 connections. In comparison, the average work-phase under the accept-limit-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 1024 for the SYN queue) that fills first, leading to higher AcceptQFull counts.

The relatively short work-phases of the accept-limit-1 policy also mean 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. The μserver statistics at the request rate of 24,000 indicate that the accept-limit-1 policy makes 12,507 `select` calls per second, compared to only 267 calls per second for the accept-limit-Inf policy. Clearly, the accept-limit-1 policy provides a poor amortization of `select` overhead, which hurts performance. The results at 27,000 and 30,000 requests/sec are qualitatively similar.

The results for TUX reveal that at 24,000 requests per second, the accept-limit-1 policy reduces TUX's queue drops dramatically. This more aggressive accept strategy is successful in keeping both the SYN queue, and the accept queue relatively empty, and the resulting performance is quite good. For TUX, the more aggressive accept-limit-1 policy mostly reduces AcceptQFull counts. TUX with accept-limit-1 obtains higher accept rates not by accepting more connections in bigger batches (as is done in the μserver) but by more frequently accepting one connection. The result is a less bursty accept policy.

We note that previous research [27] [10] has investigated techniques for reducing the overheads associated with queue drops. We believe such techniques can complement a well-chosen accept strategy. Together, they should provide higher throughputs and more stable overload behaviour.

Figure 17 shows the mean response time for the μserver and TUX using the accept-limit-1 and accept-limit-Inf policies as the load increases. These graphs are obtained by measuring the latency of each completed HTTP 1.1 request at the client under the SPECweb99-like workload.

Although it may be difficult to discern, the graph shows that mean response times are essentially zero at low loads. However, once the servers becomes saturated the mean response times increase significantly. The μserver with accept-limit-1 has the lowest mean response times under overload conditions. This is because it does not spend very much time accepting new connections and is able to process the requests that it does accept relatively quickly. In contrast the mean response times for the μserver with accept-limit-Inf are higher because the server is quite bursty, and alternates between long accept-phases and long work-phases.

Under overload conditions TUX with accept-limit-1 obtains both high throughput and relatively low mean response times (in contrast to TUX with accept-limit-Inf). With accept-limit-1, the accept phases are shorter, permitting work phases to be processed sooner. This translates into a high accept-rate and little burstiness, and provides both high throughput and reasonably low response times.

Our comparison of user-mode and kernel-mode servers produces considerably different results than recent work by Joubert et al. [15]. 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 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

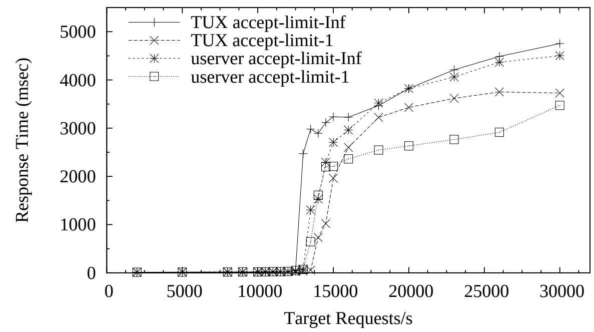

Figure 17: *μserver and TUX latencies under the SPECweb99-like workload*

stack, zero copy disk I/O, eliminating kernel crossings, etc.), our comparison shows that a user-mode server can rival the performance of TUX.

Some of the gains in user-mode performance are due to the zero-copy `sendfile` implementation that is now available on Linux. There are also differences in workloads. Specifically, Joubert et al. used a HTTP 1.0 based SPECweb96 workload, while we use a HTTP 1.1 based SPECweb99 workload. Lastly, we note the use of different operating system versions, a different performance metric, and possibly different server configurations. In spite of these differences, our work demonstrates that a well tuned user-mode server can closely rival the performance of a kernel-mode server under representative workloads.

7 Discussion

Accept strategies can have considerable impact on web server performance. As a result, we believe these strategies should be considered (along with other parameters that affect performance) when comparing different web servers. We point out that every server has an implicit accept strategy. Perhaps without realizing it, every server makes a decision regarding what portion of the available work should be immediately processed. We emphasize that we have not fully explored the parameter space of possible accept strategies. Instead, we have devised a simple method for demonstrating that accept strategies can have considerable impact on performance in three very different servers. In the future, we plan to investigate techniques for dynamically obtaining a balanced accept strategy that will self-tune for different hardware, operating systems, and even server architectures.

This paper presents a detailed comparison of the μserver and TUX. It is tempting to compare the graphs containing the μserver and Knot results in order to compare the performance of the event-driven and multi-threaded servers. However, such a comparison would be unfair since Knot and the μserver were run in slightly different (hardware and software) environments. As a result, we refrain from direct comparisons of these two servers.

The results obtained with the two workloads studied in this paper show that the accept strategy has a bigger impact on throughput under the one packet workload than with the SPECweb99-like workload. This is especially important in light of recent studies that have highlighted deficiencies of the SPECweb99 workload.

Nahum [18] analyzes the characteristics of the SPECweb99 workload in comparison with 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, which account for 28% of all requests in some server traces, and often result in the server transmitting an HTTP header and no file data. Nahum also reports that SPECweb99's 5,120 byte median file size is significantly larger than the 230 bytes observed in one popular log. 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 by Jamjoom et al. [14]. They report that many popular browsers 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. This strategy results in between 1.2 and 2.7 requests per connection which is considerably lower than the average of 7.2 requests per connection used by SPECweb99.

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

8 Conclusions

This paper 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.

Our experimental evaluation of different accept strategies expose these servers to representative workloads involving 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 well-tuned accept policies can yield noticeable improvements compared with the base approach used in each server. Under two different workloads, we are able to improve throughput by as much as 19% – 36% for TUX, 0% – 32% for Knot, and 39% – 71% for the μserver. 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 under the workloads considered the throughput of the user-mode μserver rivals that of TUX.

In future work we plan 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). Additionally this information would permit us to implement more controlled policies by limiting the length of each phase.

9 Acknowledgments

We gratefully acknowledge Hewlett Packard (through the Gelato Federation), the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for portions of this project. This work has benefited substantially from discussions with and feedback from Martin Arlitt, Michal Ostrowski, Brian Lynn, Amol Shukla, Ken Salem, Mohammed Abouzour and our shepherd Vivek Pai.

References

[1] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup web site. *IEEE Network*, 14(3):30–37, May/June 2000.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[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] HP Labs. The userver home page, 2003. Available at http://hpl.hp.com/research/linux/userver.

[12] E. Hu, P. Joubert, R. King, J. LaVoie, and J. Tracey. Adaptive fast path architecture. *IBM Journal of Research and Development*, April 2001.

[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] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[31] 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.

[32] 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.