Appears in the Proceedings of 14th IBM Center for Advanced Studies Conference (CASCON), October, 2004

Evaluating the Performance of User-Space and Kernel-Space Web Servers

Amol Shukla[†], Lily Li[‡], Anand Subramanian[†], Paul A.S. Ward[‡], Tim Brecht[†]
University of Waterloo
{ashukla,l7li,anand,pasward,brecht}@uwaterloo.ca

Abstract

There has been much debate over the past few years about the practice of moving traditional user-space applications, such as web servers, into the kernel for better performance. Recently, the user-space μserver web server has shown promising performance for delivering static content. In this paper we first describe how we augmented the μserver to enable it to serve dynamic content. We then evaluate the performance of the μserver and the kernel-space TUX web server, using the SPECweb99 workload generator under a variety of static and dynamic workloads. We demonstrate that the gap in the performance of the two servers becomes less significant as the proportion of dynamic-content requests increases. In fact, for workloads with a majority of dynamic requests, the μserver outperforms TUX. We conclude that a well-designed user-space web server can compete with an in-kernel server on performance, while retaining the reliability and security benefits that come from operating in user space. The results presented in this paper will help system developers and administrators in choosing between the in-kernel and the user-space approach for web servers.

1 Introduction

The demand for web-based applications has grown rapidly over the past few years, which has motivated the development of faster and more efficient web servers. High-performance web servers need to multiplex between thousands of simultaneous connections without degrading the performance.

Many modern web applications require the generation of *dynamic content*, where the response to a user request is the result of a server-side computation. Dynamic content creation allows web pages to be personalized based on user preferences, thereby enabling a more interactive experience for the user than that possible with static-only content. On-the-fly response generation also provides a web interface to the information stored in databases. Dynamic requests are often compute-bound [1] and involve more processing than static-content requests. The on-demand creation of content can significantly impact the scalability and the performance provided by a web server [1, 2, 3].

Various techniques have been investigated to improve the performance of web servers. These include novel server architectures [4, 5] as well as modifications to operating system interfaces and mechanisms [6, 7, 8, 9, 10, 11]. An emerging trend has seen web servers being moved into the kernel for better performance. TUX [12] (now known as Content Accelerator) is a kernel-space web server from Red Hat. By running in kernel-space, TUX avoids the overheads of context switching and event notification, and implements several optimizations to reduce redundant data reads and copies. TUX supports the delivery of both static and dynamic content.

Initial research has indicated that kernel-based web servers provide a significant performance advantage over their user-space counterparts in the delivery of *static content*, and the in-kernel TUX web server has been shown to be nearly twice as fast as the best user-space servers (IIS on Windows

[†]School of Computer Science
[‡]Department of Electrical and Computer Engineering

2000 and Zeus on Linux) [13]. In this paper we revisit the kernel-space versus user-space debate for web servers. In particular, we expand it to include dynamic content workloads. Recent research has shown that a significant proportion of workloads in real-world web sites consists of dynamic requests [1, 14], with Arlitt et al. [1] reporting that over 95% of the requests in an e-commerce web site are for dynamic content.

Recently, the user-space μserver [15] web server has shown promising results that rival the performance of TUX on certain static-content workloads [16]. Prior to our work, the μserver did not have the ability to handle dynamic requests. We first demonstrate how to augment the μserver to enable it to serve dynamic content. We then compare the performance of the μserver and TUX (the fastest reported in-kernel web server in Linux) under a variety of static and dynamic workloads. We demonstrate that the gap in the performance of the two servers becomes less significant as the proportion of dynamic-content requests increases. Further, we show that for workloads with a majority of dynamic requests, the μserver can outperform TUX. Based on the results of our experiments, we suggest that a well-designed user-space web server can compete with an in-kernel server on performance, while retaining the reliability and security benefits that come from operating in user space.

The rest of this paper is organized as follows: Section 2 provides additional background information and summarizes some of the related work. In Section 3 we present an overview of various approaches for supporting the delivery of dynamic content in web servers, and describe how we augmented the μserver to handle dynamic requests. Section 4 includes a detailed description of our experimental environment, methodology and workloads. In Section 5 we present the results of our experiments and analyze them. In Section 6 we summarize our findings, discuss their implications and outline some ideas for future work.

2 Background and Related Work

Modern web servers require special techniques to handle a large number of simultaneous connections. In order to understand the performance characteristics of web servers, it is important to consider the steps performed by a web server in handling a single client request. Figure 1, due to Brecht et al. [16], illustrates this process.

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 based on the data provided in the request and the information stored at the server.
6. Send the reply to the requesting client.
7. If required, close the network connection.

Figure 1: Steps performed by a web server to process a client request.

Many of the steps in the figure can block due to network or disk I/O. A high-performance web server has to handle thousands of such requests simultaneously without degrading the performance. Different server architectures [4, 5] have been proposed to handle this problem and are summarized by Pai et al. [4]. An event-driven approach is often implemented in high-performance network servers to multiplex a large number of requests over a few server processes. Using an event notification mechanism, such as the `select` system call, the server identifies connections on which forward progress can be made without blocking, and processes only those connections. By using just a few processes, event-driven servers avoid excessive context-switches required by the thread- or process- per-connection approach taken by multi-threaded or multi-process servers like Apache [4].

Other researchers have suggested modifications in operating system interfaces and mechanisms for efficient notification and delivery of network events to user-space servers [6, 7, 8, 17], reducing the amount of data copied between the kernel and the user-space [10], reducing the number of kernel-boundary crossings [9], and a combination of the above [11].

In light of the considerable demands placed on the operating system by web servers, some researchers have sought to improve the performance of user-space web servers by migrating them into the kernel. By running in the kernel, the cost of event notification can be minimized through fewer kernel crossings, and the redundant copying and buffering of data in the user-space application can be avoided. The web server can also implement optimizations to take advantage of direct access to kernel-level data structures. In particular, a closer integration with the TCP/IP stack, the network interface, and the file-system cache is possible.

Following an in-kernel implementation of the Network File System (NFS) server in Linux, two kernel-space web servers were proposed. kHTTPd [18] runs in the kernel as a module and handles only static-content requests. TUX [12] is a kernel-based high-performance web server with the ability to serve both static and dynamic data. It implements several optimizations such as zero-copy parsing, zero-copy disk reads, zero-copy network writes, and caching complete responses in the kernel's networking layer to accelerate static content delivery. By providing support for modules that generate dynamic content and mass virtual hosting, TUX is pitched as a high-performance replacement for traditional user-space web servers.

Joubert et al. [13] demonstrated that the best performing user-space web servers, IIS on Windows 2000 and Zeus on Linux, are about two times slower than TUX on the SPECweb96 benchmark. They also reported that TUX is more than three times faster than the popular Apache web server and provides better performance than kHTTPd. However, their paper only deals with static-content requests and non-persistent (HTTP 1.0) connections. In this paper, we update their kernel versus user-space comparison on Linux by evaluating the performance of the in-kernel TUX web server with the event-driven, user-space μserver using SPECweb99 workloads, which include both static and dynamic content requests, and HTTP 1.1 persistent connections. The μserver has recently shown promising results that compare favourably against the performance of TUX on completely cached static workloads [16]. By implementing support for dynamic requests in the μserver we broaden the comparison of user-space and in-kernel web servers to include dynamic as well as out-of-cache (disk-bound) static workloads.

While the delivery of static content is a simple process, the generation of dynamic content requires server-side computation, which can affect the scalability and the performance of a web server [2, 3]. Our hypothesis was that since the generation of dynamic content is CPU-bound [1], processing these requests in the kernel might not prove to be as advantageous as the case with static-content delivery. Recent workload-characterization studies by Arlitt et al. [1] and Wang et al. [14] report that a significant proportion of the requests handled by modern e-commerce web sites are for dynamic content. To our knowledge, no studies examining the relative benefits of the user-space versus kernel-space approach for web servers using workloads containing dynamic-content requests have been published, and our work attempts to fill this void.

3 Implementing Dynamic Content Support in the μserver

Prior to this paper, the μserver did not support the delivery of dynamic content. In this section, we describe how we modified it to support dynamic requests for evaluation against TUX on the SPECweb99 benchmark. First we provide a brief overview of existing dynamic content generation mechanisms.

The Common Gateway Interface (CGI) is a standard for running external programs which generate dynamic content from web servers. Whenever a server receives a dynamic request, it creates a new process to handle the request and uses the CGI specification to interact with that process. After the dynamic content has been generated, the external process passes the results to the web server and terminates. Launching a new process is an expensive operation in terms of system resources. The CGI approach has been identified as having poor scalability due to the process-creation overhead for every request [19, 20].

FastCGI or Servlets offer an improvement over CGI by using a pool of persistent processes (or threads) to handle dynamic requests. The processes that generate dynamic content are persistent in that they are not terminated after every request, but are reused for later requests. The web server does not launch a new process for every dynamic request but utilizes an existing process from the pool to generate content. A FastCGI or Servlet-like mechanism reduces the process-creation overhead and allows resources such as database connections to be shared between different dynamic-content handlers. A related dynamic content generation mechanism called *templating* is used in PHP and Java Server Pages, and involves embedding a scripting or a programming language in HTML to generate request-specific responses. However, both the CGI and FastCGI-like approaches involve the overhead of Inter-Process Communication between the web server and the external processes while handling each dynamic request.

Various server extensions have been proposed to minimize the inter-process communication overheads. Web servers such as IIS provide APIs which allow dynamic-content generation operations to run as part of the main server process. The extension API approach involves the development of modules that use published server interfaces to handle particular types of dynamic requests. Most high-performance web servers, including TUX, have APIs to allow modules to use the services of the web server to generate dynamic content.

Modules responsible for dynamic-content generation can be loaded statically or dynamically (on-the-fly) by the web server. Dynamic loading can be conceptually viewed as follows (adapted from Millaway and Conrad [20]): the first time a user module is requested in a URI or when the server starts up, it loads the corresponding shared object (using a function similar to `dlopen`), and obtains an entry point (typically a function pointer) into the module (using a function similar to `dlsym`). The module communicates with the web server using published APIs. The server can pass on the HTTP requests to the appropriate module by maintaining a mapping between the function pointer provided and the module name. TUX provides an interface for the generation of dynamic content by trusted modules, which can be dynamically loaded. Lever et al. [21] describe in detail how support for user modules is implemented in TUX.

Since the server extension API approach places maximum emphasis on high performance, we have developed support for static modules in the μserver. In our implementation, modules responsible for dynamic-content generation have to be compiled and statically linked with the μserver. Dynamic requests are delegated by the μserver to an appropriate module, using a procedure similar to the one described above, which then generates and delivers the content. In the future, we intend to implement support for dynamic loading of modules with a cleaner API in μserver. Note that the results of our experiments are not affected by the static or dynamic loading of modules.

For TUX, we use a freely available user-space module for handling the dynamic requests required for the SPECweb99 benchmark [22]. Our module for handling SPECweb99 requests in the μserver borrows heavily from this TUX module, with the TUX APIs replaced by those specific to the μserver. By using a common code base for the modules in TUX and the μserver, we expect to eliminate any differences in performance due to different SPECweb99 request-handler implementations, thereby providing a truer comparison between in-kernel and user-space web servers for the benchmark's dynamic requests.

4 Experimental Environment

All experiments are conducted in an environment consisting of 8 client machines and a server connected via two full-duplex Gigabit Ethernet links. All client machines have 550 MHz Pentium III processors, 256 MB of memory and run the 2.4.7-10 Linux kernel. The server is a 400 MHz Pentium II machine with 512 MB of memory and a 7200 RPM IDE disk, and runs Red Hat Linux 8 with the 2.4.22 kernel. Since the server has two network cards, we avoid network bottlenecks by partitioning the clients into two subnets. The first four clients communicate with the server's first Ethernet cards, while the remaining four use a different IP address linked to the second Ethernet card. This client-server cluster is completely isolated from other network traffic. In order to ensure that we are able to generate sufficient load to drive the server without letting the clients become a bottleneck, we use the more powerful machines as our clients. For all our experiments the server kernel is run in uniprocessor mode.

4.1 SPECweb99 Workload Generator

We evaluate the performance of the user-space μserver against the in-kernel TUX web server using the SPECweb99 workload generator.

The SPECweb99 benchmark [23] has become the de-facto tool for web server performance evaluation in the industry [24]. Nahum [24] examines how well SPECweb99 captures real-world web server workload characteristics such as request methods, request inter-arrival times, and URI popularity. Detailed specifications of the benchmark can be found at the SPECweb99 web site [23, 25]. The SPECweb99 workload generator uses multiple client systems to distribute HTTP connections made to the web server. The number of simultaneous connections is fixed for the duration of a SPECweb99 iteration. Each connection is used to make HTTP requests to the server according to the predefined workload.

The SPECweb99 clients can create workloads containing both static-content and dynamic-content requests. The model for dynamic content in SPECweb99 is based on two prevalent features of commercial web servers: advertising and user registration [23]. A portion of SPECweb99 dynamic GET requests simulate ad rotation in a commercial web site, where the ads appearing on a web page are generated and rotated on-the-fly based on the preferences of the user tracked through cookies. The dynamic POST requests in SPECweb99 model user registration at an ISP site, where client information is passed to the web server as a POST request, and the server is responsible for storing the data in a text file.

SPECweb99 specifies four classes of dynamic requests: standard dynamic GET, dynamic GET with custom ad rotation through cookies, dynamic POST, and CGI dynamic GET [23]. Since the scalability problems of CGI are well-documented [19, 20], we do not use CGI dynamic GET requests in any of our experiments. The workload mix in SPECweb99 can be configured to vary the proportion of the type of requests (static or dynamic), the HTTP method (GET or POST), and version (1.0 or 1.1) used. While we experimented with a variety of workload mixes involving both static and dynamic workloads, we report only some of the most representative results in this paper.

4.2 Methodology

Each experiment involves a distinct SPECweb99 workload mix. A workload mix is repeated for both of the web servers. Note that we do not use the 'maximum number of conforming simultaneous connections' performance metric specified by the SPECweb99 benchmark because it describes the behaviour of the web server at a *single* data point. Instead, we use the throughput reported for successful HTTP requests as the number of simultaneous connections is increased as our performance metric. Consequently, our results do not comply with the reporting guidelines of the SPECweb99 benchmark. However, we believe that our performance metric provides a better understanding of web server behaviour under varying loads, giving us a range of data points so that we can compare web server performance under light, medium and heavy load.

Each experiment consists of a series of data points, where each data point corresponds to a SPECweb99 iteration where the number of simultaneous connections attempted is fixed. Note that the number of simultaneous connections attempted is not necessarily equal to the number of simultaneous connections established. Each iteration consists of one minute of warm-up time and two minutes of actual measurement time. The warm-up time between successive data points eliminates the cost of dynamically loading modules and allows various system resources such as the state of socket descriptors to be cleared from previous tests. Prior to running experiments, all non-essential services such as sendmail, dhcpd, and cron were shutdown to prevent these daemons from confounding the experimental results. The results are depicted using graphs where the horizontal axis represents the number of simultaneous connections requested and the vertical axis shows the server throughput.

The size of the file set is determined by the number of simultaneous connections generated by the SPECweb99 workload generator and is computed using the formula described in [25]. The file set size varies from 1.7 GB for 500 simultaneous connections to 3.3 GB for 1000 simultaneous connections to 4.8 GB for 1500 simultaneous connections, and the file set cannot be completely cached in memory.

4.3 The Web Servers

We use the in-kernel TUX (kernel module version 2, user-module version 2.2.7) and μserver 0.4.5 in our experiments.

In the interest of making fair and scientific comparisons, we carefully configured TUX and μserver to use the same resource limits. The maximum number of concurrent connections is set to 15,000 for TUX as well as the μserver. Logging is disabled in both the servers. The dynamic content modules in both the μserver and TUX use the `sendfile` system call, which enables quick data transfer between file and socket descriptors. The Linux kernel versions that we examined (2.4.22, 2.6.1) silently limit the accept queue backlog set by applications (through the `listen` system call) to 128. By default, TUX bypasses this kernel-imposed limit in favour of a much larger value, viz., 2048. We changed the accept queue backlog in TUX to 128 to match the value imposed by the kernel on the user-space μserver. While we could have recompiled the kernel to allow the μserver to use an accept queue backlog of 2048, we opted not to use this approach because a large number of systems currently operate with the accept queue limit of 128.

For the μserver we use the *Accept-Inf* option [16] while accepting new connections, which causes the server to consecutively accept all currently pending connections, and the `select` system call as the event notification mechanism. Brecht et al. have demonstrated that using these two options the μserver can provide high performance [16]. The μserver relies on the file system cache for caching static files, but the size of the available memory (512 MB) is small compared to the size of the file set used in each data point in our experiments. TUX uses a pinned memory cache for accelerating the delivery of static content, along with a number of other high-performance optimizations such as zero-copy parsing, zero-copy disk reads and zero-copy network writes.

There is one major limitation to the current handling of requests in the μserver. Some system calls can block due to the lack of support for non-blocking or asynchronous disk reads in Linux. Some system calls responsible for writing dynamically generated content to the network can also block in our implementation of dynamic content support in the μserver. The equivalent disk read and network writes in TUX are non-blocking because TUX uses an architecture similar to that of Flash [4] to asynchronously manage those operations [21]. We address this performance problem in the μserver by using multiple processes. Note that each μserver process behaves in event-driven fashion and multiplexes a number of connections using the `select` system call. All the μserver processes listen for connections on the same port, share all the machine resources, and depend on the operating system for the distribution of work. The advantage of using multiple processes kicks in only when one of them is suspended due to a disk read or network write, and the operating system can schedule another process that is ready to run. The TUX user manual recommends that the number of worker threads used in the server should not be greater than the number of CPUs available [12]. Since we use a single CPU for the experiments, we set the number of worker threads in TUX to one. By running some experiments with more than one TUX worker thread, we verified that a single worker thread results in the best performance for TUX for our static, dynamic and mixed workloads. Note that in addition to the worker thread, TUX uses a pool of kernel threads to obtain new work and asynchronously handle disk I/O tasks [21].

5 Results

Our first experiment involves a static-only workload (with no dynamic requests), which can be seen as a yardstick against which we evaluate our later workloads that contain a varying proportion of dynamic requests. Note that previous studies comparing in-kernel and user-space web servers have concentrated on static workloads that can be completely cached in memory. In comparison, the static workload used in our experiment is disk-bound. Figure 2 illustrates the relative performance of the μserver with a single process (labelled ‘userver-1-process’), the μserver running 16 event-driven processes (labelled ‘userver-16-processes’) and TUX.

The servers show similar behaviour up to 300 simultaneous connections. Many real-world web sites experience loads lighter than this and the negligible difference in the performance provided by the servers would tilt the scales in favour of the μserver, which offers the security and reliability benefits of running in the user-space. However, large commercial and informational web sites often have to handle a significantly higher number of concurrent connections. As the load increases, TUX provides considerably higher throughput than the μserver. The gap in the peak throughput (at 700 simultaneous connections) of the 16-process μserver and TUX is 40%. Past 700 simultaneous connections, the throughput of both servers deteriorates, and the gap between the two becomes as large as 61% at 1100 simultaneous connections. Although this difference is significant, it is a lot smaller than previously-reported results, where the

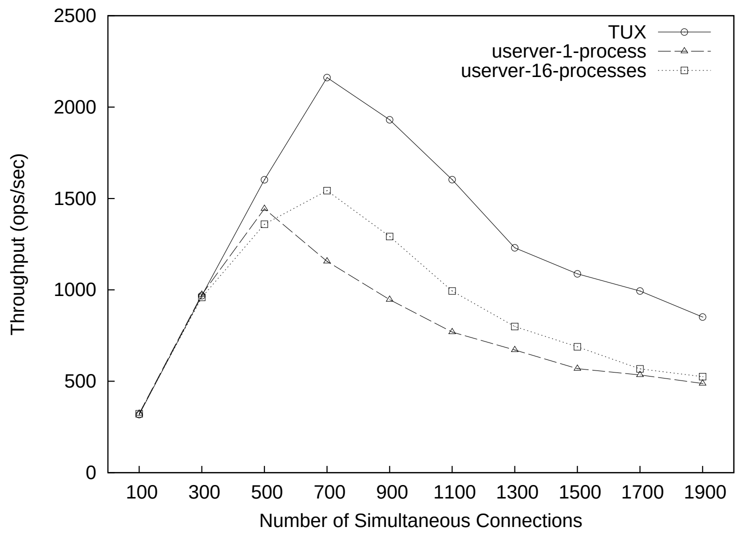

Figure 2: 100% Static Workload Performance: μserver vs. TUX

kernel-space TUX web server was shown to be almost twice as fast as the best user-space web server [13].

Using 16 processes does improve the performance of μserver, helping it achieve higher peak throughput than the single-process μserver. With a single process, the μserver is brought to a standstill if a disk-read operation blocks, but with 16 processes even if some of them are blocked, the operating system can schedule a process that is ready to run allowing the server to continue processing requests. It should be noted that using too many processes can actually inhibit the performance due to excessive context switches and high scheduling overhead. For the static workload using more than 16 processes resulted in a drop in performance. These results have been left out to reduce clutter in the graphs.

We now move to the evaluation of TUX and the μserver under a workload consisting exclusively of dynamic-content requests. For this experiment we use the default SPECweb99 mix for dynamic requests (without dynamic CGI GET requests, as explained earlier), which comprises of 42% dynamic GET, 42% dynamic GET with cookies and 16% dynamic POST requests. The results of this experiment appear in Figure 3. Note that using more than 8 processes in the μserver for this workload resulted in degradation in performance due to excessive content-switching.

We can see that the in-kernel TUX web server is not as effective in the delivery of dynamic content and is outperformed by the μserver with 8 processes. The gap in their peak throughput (at 500 simultaneous connections for TUX and 700 simultaneous connections for the μserver) is 13%. The μserver with 8 processes outperforms TUX by as much as 42% at the extreme points in the gap at 1300 simultaneous connections.

For the dynamic requests workload, using multiple processes impacts the performance of the μserver significantly, the peak throughput of the μserver with 8 processes is 29% higher than that of the single-process μserver. We used the `vmstat` and `sar` utilities to gather information about system load activity at 500 simultaneous connections. The statistics indicate that the μserver using a single process is blocked (reading large files from disk or writing large responses to the network) for at least 10% of the duration of the experiment. On the other hand, with the 8-process μserver, we seldom reach a point where all eight processes are blocked, so some requests can always be serviced. If all the server processes are blocked, CPU cycles are wasted. Indeed for the single-process μserver, the CPU is idle 40% of the time. For dynamic requests which are compute-bound, this translates to poor performance.

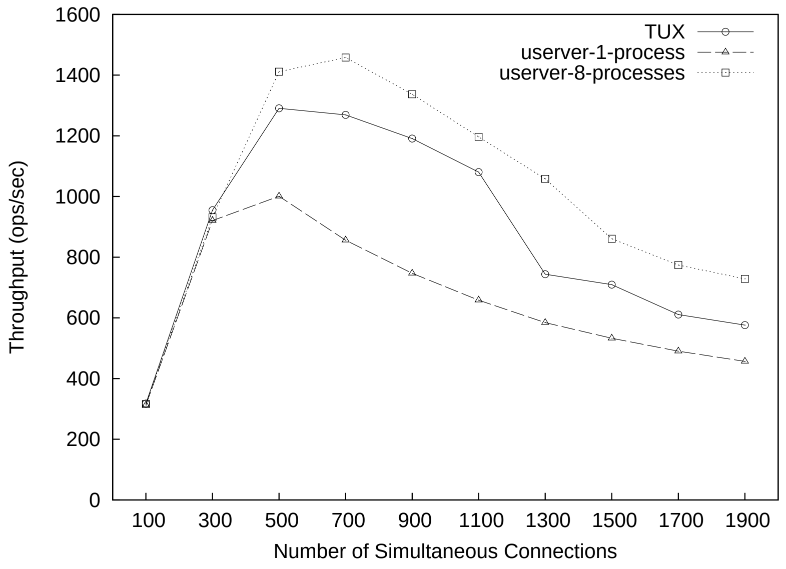

Figure 3: 100% Dynamic Workload Performance: μserver vs. TUX

At this point we reiterate that using more than one worker thread in TUX resulted in a degradation in its performance on the workloads that we examined. Using `vmstat` and `sar` we verified that the reduction in TUX's throughput on workloads containing dynamic requests is *not* because it is blocked more often than the multi-process μserver. In trying to understand why TUX performs poorly compared to the μserver on the dynamic content intensive workload we have identified two possible causes.

One reason for the reduction in the performance gap in the two servers could be that dynamic requests are processor-intensive [1, 2, 3], hence, the network and memory optimizations in TUX do not play as large a role in influencing the overall performance compared to static content delivery.

A second factor in the drop in performance in TUX might be its strategy for accepting connections. TUX accepts new connections more aggressively, instead of making forward progress on already accepted connections [16]. We used the `netstat` utility to gather data about TCP queue drops in both the servers, and found that the number of queue drops in TUX were an order of magnitude lower than those in the μserver. The fewer queue drops in TUX suggests that it is very aggressive in accepting new connections. In contrast, a higher queue-drop value implies that the μserver favours the completion of pending work before accepting new connections. In investigating this further we observed that the TUX module that generates dynamic content consumes less than 25% of the CPU time on an average. The overall CPU usage by TUX, including that by the dynamic content generation module, is close to 90%. We suspect that as the number of simultaneous connections increases, TUX spends too much time in trying to process new connections, instead of allocating sufficient CPU time to its dynamic content generation module to complete the work on existing connections. As noted by Brecht et al. [16], there seems to be room for improvement in the connection-scheduling mechanism implemented in TUX.

So far we have evaluated TUX and the μserver on workloads comprising exclusively static or exclusively dynamic requests. However, the workload mix seen in most real-world web servers is usually a combination of dynamic and static requests. The proportion of dynamic to static requests differs based on the type of web site. In the final set of experiments we use a mixed workload where the ratio of dynamic to static content is varied. In doing so we attempt to highlight the transition in the gap in the performance of user-space and kernel-space web servers as the proportion of dy-

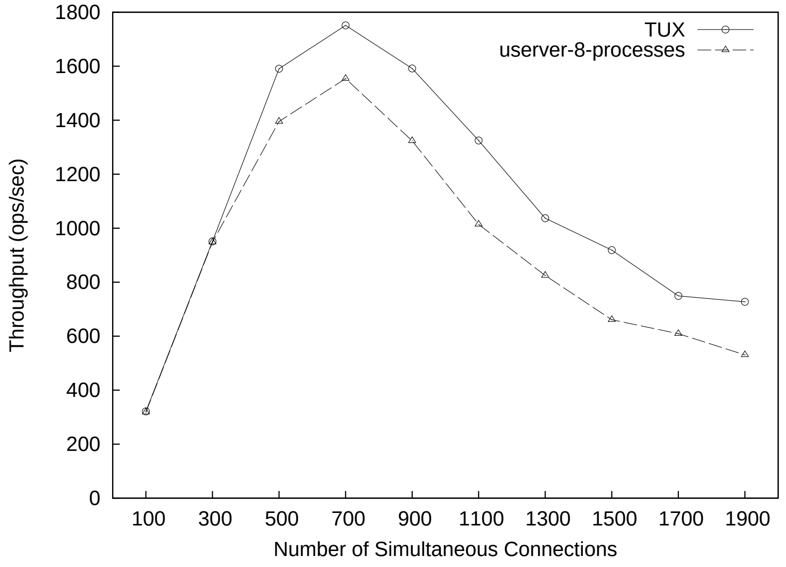

Figure 4: Mixed Workload (30% dynamic-70% static) Performance: μserver vs. TUX

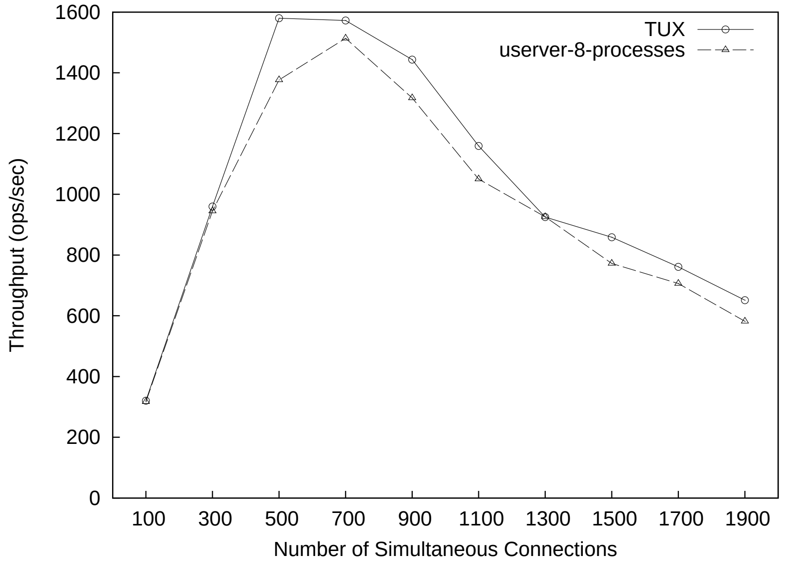

Figure 5: Mixed Workload (50% dynamic-50% static) Performance: μserver vs. TUX

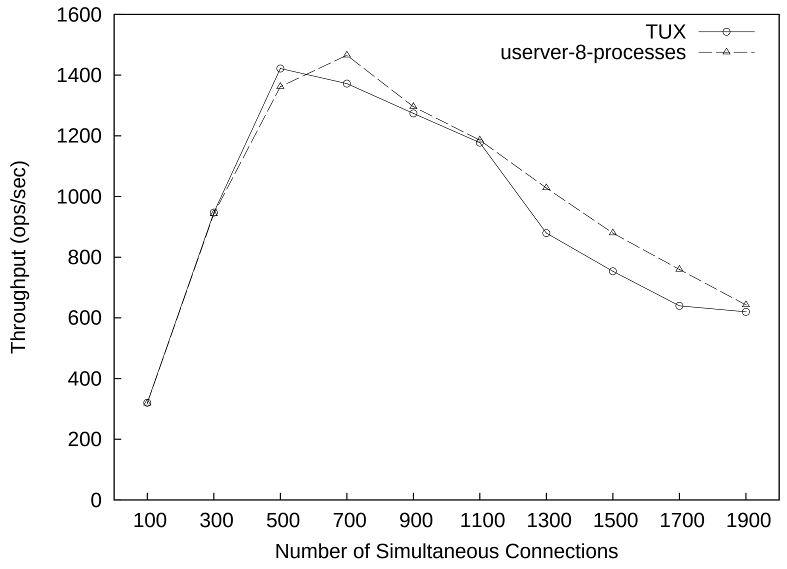

Figure 6: Mixed Workload (80% dynamic-20% static) Performance: μserver vs. TUX

namic requests increases. We use the request mix described earlier for the dynamic-content portion of all our mixed workloads. We only show the results of the μserver with 8 processes, which is the best configuration for the number of processes, in the following figures.

Our first mixed workload consists of 30% dynamic and 70% static requests. This ratio of dynamic to static content is recommended by the SPECweb99 benchmark. The results for this workload appear in Figure 4. TUX provides higher throughput than the μserver for this workload. However, the gap in their performance is much smaller than that in the static-only workload. The difference in their peak throughput is 13%, and this gap becomes as high as 39% at 1500 simultaneous connections.

Figure 5 shows the results of the experiment with a workload containing 50% dynamic and 50% static requests. We can see that the gap in the performance of TUX and the μserver is very small. The difference in their peak throughput is under 4%, and μserver's throughput is always within 15% of that provided by TUX.

Our final mixed workload consists of 80% dynamic and 20% static requests. Figure 6 illustrates the results of this experiment. The μserver outperforms TUX at all but one data point. Its peak throughput is 3% higher than that of TUX. As reported by Arlitt et al., some e-commerce web sites do see workloads where the proportion of dynamic requests is much higher than 80% [1]. For such web sites using a high-performance user-space web server such as the μserver might be a much better option.

The results of our experiments suggest that the performance merits of in-kernel web servers diminish in the presence of dynamic-content requests. The gap in the peak throughput of TUX and the μserver drops from 40% for static-only workloads to under 4% for a workload with evenly split static and dynamic requests, and the μserver outperforms TUX by a small margin as the proportion of dynamic-content requests increases above 80%.

6 Conclusions and Future Work

The contentious issue of kernel-space versus user-space web servers has been previously investigated in the context of completely cached static workloads. We have expanded this discussion to cover dynamic workloads and disk-bound static workloads, using two high-performance web servers:

the user-space μserver and the kernel-space TUX, and the SPECweb99 workload generator.

Our results suggest that the improvements to operating systems interfaces such as the `sendfile` system call and novel implementation techniques such as the multi-accept strategy [16], which amortizes the overhead of event notification, have enabled the user-space web servers to close in the gap with their in-kernel counterparts for static content delivery. However, for predominantly static requests, kernel-space web servers still hold a performance advantage, and further improvements to operating system interfaces and mechanisms are required to improve the performance of user-space web servers.

In-kernel web servers are not as effective in improving the performance on workloads containing dynamic requests. As the proportion of dynamic-content requests increases, the advantages of the kernel-space web servers diminish. Their performance gap with user-space web servers is negligible if most of the requests are for dynamic content.

Given that in-kernel servers do not provide the security and reliability benefits of running in user-space, we recommend that system designers and administrators of web sites with a significant portion of dynamic content examine the implications of migrating web servers to the kernel more carefully. During the course of our tests, TUX crashed several times resulting in a kernel panic rendering the server machine unusable. It is typically easier to implement crash-recovery mechanisms for user-space web servers. A crash in the μserver can be fixed by simply restarting it. Configuring TUX to ensure stable behaviour and high performance is a non-trivial task, and debugging problems in TUX requires some familiarity with the kernel.

In the future we intend to compare the μserver with TUX using some of the new operating system modifications which intend to improve the scalability and performance of user-space web servers. While the μserver can match the performance of TUX on dynamic content intensive workloads, using a scalable event notification mechanism such as `epoll` or an I/O mechanism such as asynchronous I/O could reduce its performance gap with TUX on static workloads.

Acknowledgements

We gratefully acknowledge the National Sciences and Engineering Research Council of Canada, Hewlett Packard, and the Ontario Research and Development Challenge Fund for financial support for portions of this project.

About the Authors

- **Amol Shukla** is an M.Math student in the School of Computer Science at the University of Waterloo. His research interests include Network Server Performance, Autonomic Computing, and System Support for Mobile and Distributed Applications.
- **Lily Li** is an M.A.Sc. student in the department of Electrical and Computer Engineering at the University of Waterloo. Her main research interests include Wireless Mesh Networks and Ad Hoc Networks.
- **Anand Subramanian** is an M.Math student in the School of Computer Science at the University of Waterloo. His general areas of research are Operating Systems and Networking Protocols, he is particularly interested in operating system interfaces that improve the scalability and performance of applications including network servers.
- **Paul A.S. Ward** is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Waterloo. His research interests are broadly in dependable distributed systems. Specifically he has worked on problems of distributed-systems observation and control, and more recently on autonomic distributed systems. Prior to pursuing his Ph.D. he worked in both the hardware and software industries, covering the range from electronic parking meter design to developing the fast parallel load utility for the DB2 database system.
- **Tim Brecht** obtained his B.Sc. from the University of Saskatchewan in 1983, M.Math from the University of Waterloo in 1985, and Ph.D. from the University of Toronto in 1994. He is currently an Associate Professor at the University of Waterloo. He has previously held positions as an Associate Professor at York University (in Toronto), a Visiting Sci-

entist at IBM's Center for Advanced Studies, and a Research Scientist with Hewlett Packard Labs. Current research interests include Operating Systems; Internet Systems, Services and Applications; Parallel Computing; and Performance Evaluation.

References

[1] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *IEEE/ACM Transactions on Internet Technology*, 1, June 2001.

[2] A. Iyengar, E. MacNair, and T. Nguyen. An analysis of web server performance. In *Proceedings of GLOBECOM '97*, 1997.

[3] V. Almeida and M. Mendes. Analyzing the impact of dynamic pages on the performance of web servers. In *Computer Measurement Group Conference*, December 1998.

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

[5] 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*, October 2001.

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

[7] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 231–244, 2001.

[8] 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*, June 1999.

[9] Erich Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, Vol. 10, February 2002.

[10] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, Vol. 18:37–66, 2000.

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

[12] Red Hat Inc. *TUX 2.2 Reference Manual*, 2002. Available at http://www.redhat.com/docs/manuals/tux/TUX-2.2-Manual.

[13] Philippe Joubert, Robert B. King, Richard Neves, Mark Russinovich, and John M. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *USENIX Annual Technical Conference, General Track*, pages 175–187, 2001.

[14] Qing Wang, Dwight Makaroff, H. Keith Edwards, and Ryan Thompson. Workload characterization for an E-commerce web site. In *Proceedings of CASCON 2003*, pages 313–327, 2003.

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

[16] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference, General Track*, June 2004.

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

[18] Arjan van de Ven. kHTTPd Linux HTTP accelerator. Available at http://www.fenrus.demon.nl.

[19] Giorgos Gousios and Diomidis Spinellis. A comparison of portable dynamic web content technologies for the Apache web server. In *Proceedings of the 3rd International System Administration and Networking Conference SANE 2002*, pages 103–119, May 2002.

[20] J. Millaway and P. Conrad. C Server Pages: An architecture for dynamic web content generation. In *WWW2003*, May 2003.

[21] 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.

[22] SPECweb99 Dynamic API module source code. Standard Performance Evaluation Corporation. Available at http://www.specbench.org/web99/results/api-src/Dell-20030527-RHCA.tgz.

[23] SPECweb99 Benchmark. Standard Performance Evaluation Corporation. Available at http://www.specbench.org/web99/.

[24] Erich M. Nahum. Deconstructing SPECweb99. In *the 7th International Workshop on Web Content Caching and Distribution (WCW)*, August 2001.

[25] SPECweb99 Design Document. Standard Performance Evaluation Corporation. Available at http://www.specbench.org/web99/docs/whitepaper.html.