Comparing and Evaluating epoll, select, and poll Event Mechanisms

Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag
University of Waterloo
{lgammo,brecht,ashukla,db2pariag}@cs.uwaterloo.ca

Abstract

This paper uses a high-performance, event-driven, HTTP server (the μserver) to compare the performance of the select, poll, and epoll event mechanisms. We subject the μserver to a variety of workloads that allow us to expose the relative strengths and weaknesses of each event mechanism.

Interestingly, initial results show that the select and poll event mechanisms perform comparably to the epoll event mechanism in the absence of idle connections. Profiling data shows a significant amount of time spent executing a large number of `epoll_ctl` system calls. As a result, we examine a variety of techniques for reducing `epoll_ctl` overhead including edge-triggered notification, and introducing a new system call (`epoll_ctlv`) that aggregates several `epoll_ctl` calls into a single call. Our experiments indicate that although these techniques are successful at reducing `epoll_ctl` overhead, they only improve performance slightly.

1 Introduction

The Internet is expanding in size, number of users, and in volume of content, thus it is imperative to be able to support these changes with faster and more efficient HTTP servers.

A common problem in HTTP server scalability is how to ensure that the server handles a large number of connections simultaneously without degrading the performance. An event-driven approach is often implemented in high-performance network servers [14] to multiplex a large number of concurrent connections over a few server processes. In event-driven servers it is important that the server focuses on connections that can be serviced without blocking its main process. An event dispatch mechanism such as `select` is used to determine the connections on which forward progress can be made without invoking a blocking system call. Many different event dispatch mechanisms have been used and studied in the context of network applications. These mechanisms range from `select`, `poll`, `/dev/poll`, RT signals, and epoll [2, 3, 15, 6, 18, 10, 12, 4].

The epoll event mechanism [18, 10, 12] is designed to scale to larger numbers of connections than `select` and `poll`. One of the problems with `select` and `poll` is that in a single call they must both inform the kernel of all of the events of interest and obtain new events. This can result in large overheads, particularly in environments with large numbers of connections and relatively few new events

occurring. In a fashion similar to that described by Banga et al. [3] epoll separates mechanisms for obtaining events (`epoll_wait`) from those used to declare and control interest in events (`epoll_ctl`).

Further reductions in the number of generated events can be obtained by using edge-triggered epoll semantics. In this mode events are only provided when there is a change in the state of the socket descriptor of interest. For compatibility with the semantics offered by `select` and `poll`, epoll also provides level-triggered event mechanisms.

To compare the performance of epoll with `select` and `poll`, we use the μserver [4, 7] web server. The μserver facilitates comparative analysis of different event dispatch mechanisms within the same code base through command-line parameters. Recently, a highly tuned version of the single process event driven μserver using `select` has shown promising results that rival the performance of the in-kernel TUX web server [4].

Interestingly, in this paper, we found that for some of the workloads considered `select` and `poll` perform as well as or slightly better than epoll. One such result is shown in Figure 1. This motivated further investigation with the goal of obtaining a better understanding of epoll's behaviour. In this paper, we describe our experience in trying to determine how to best use epoll, and examine techniques designed to improve its performance.

The rest of the paper is organized as follows: In Section 2 we summarize some existing work that led to the development of epoll as a scalable replacement for `select`. In Section 3 we describe the techniques we have tried to improve epoll's performance. In Section 4 we describe our experimental methodology, including the workloads used in the evaluation. In Section 5 we describe and analyze the results of our experiments. In Section 6 we summarize our findings and outline some ideas for future work.

2 Background and Related Work

Event-notification mechanisms have a long history in operating systems research and development, and have been a central issue in many performance studies. These studies have sought to improve mechanisms and interfaces for obtaining information about the state of socket and file descriptors from the operating system [2, 1, 3, 13, 15, 6, 18, 10, 12]. Some of these studies have developed improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data copied between the application and kernel. Other studies have reduced the number of events delivered by the kernel, for example, the signal-per-fd scheme proposed by Chandra et al. [6]. Much of the aforementioned work is tracked and discussed on the web site, "The C10K Problem" [8].

Early work by Banga and Mogul [2] found that despite performing well under laboratory conditions, popular event-driven servers performed poorly under real-world conditions. They demonstrated that the discrepancy is due the inability of the `select` system call to scale to the large number of simultaneous connections that are found in WAN environments.

Subsequent work by Banga et al. [3] sought to improve on `select`'s performance by (among other things) separating the declaration of interest in events from the retrieval of events on that interest set. Event mechanisms like select and poll have traditionally combined these tasks into a single system call. However, this amalgamation requires the server to re-declare its interest set every time it wishes to retrieve events, since the kernel does not remember the

interest sets from previous calls. This results in unnecessary data copying between the application and the kernel.

The `/dev/poll` mechanism was adapted from Sun Solaris to Linux by Provos et al. [15], and improved on poll's performance by introducing a new interface that separated the declaration of interest in events from retrieval. Their `/dev/poll` mechanism further reduced data copying by using a shared memory region to return events to the application.

The kqueue event mechanism [9] addressed many of the deficiencies of `select` and `poll` for FreeBSD systems. In addition to separating the declaration of interest from retrieval, `kqueue` allows an application to retrieve events from a variety of sources including file/socket descriptors, signals, AIO completions, file system changes, and changes in process state.

The epoll event mechanism [18, 10, 12] investigated in this paper also separates the declaration of interest in events from their retrieval. The `epoll_create` system call instructs the kernel to create an event data structure that can be used to track events on a number of descriptors. Thereafter, the `epoll_ctl` call is used to modify interest sets, while the `epoll_wait` call is used to retrieve events.

Another drawback of `select` and `poll` is that they perform work that depends on the size of the interest set, rather than the number of events returned. This leads to poor performance when the interest set is much larger than the active set. The epoll mechanisms avoid this pitfall and provide performance that is largely independent of the size of the interest set.

3 Improving epoll Performance

Figure 1 in Section 5 shows the throughput obtained when using the μserver with the select, poll, and level-triggered epoll (epoll-ET) mechanisms. In this graph the x-axis shows increasing request rates and the y-axis shows the reply rate as measured by the clients that are inducing the load. This graph shows results for the one-byte workload. These results demonstrate that the μserver with level-triggered epoll does not perform as well as `select` under conditions that stress the event mechanisms. This led us to more closely examine these results. Using `gprof`, we observed that `epoll_ctl` was responsible for a large percentage of the run-time. As can been seen in Table 1 in Section 5 over 16% of the time is spent in `epoll_ctl`. The gprof output also indicates (not shown in the table) that `epoll_ctl` was being called a large number of times because it is called for every state change for each socket descriptor. We examine several approaches designed to reduce the number of `epoll_ctl` calls. These are outlined in the following paragraphs.

The first method uses epoll in an edge-triggered fashion, which requires the μserver to keep track of the current state of the socket descriptor. This is required because with the edge-triggered semantics, events are only received for transitions on the socket descriptor state. For example, once the server reads data from a socket, it needs to keep track of whether or not that socket is still readable, or if it will get another event from `epoll_wait` indicating that the socket is readable. Similar state information is maintained by the server regarding the whether or not the socket can be written without blocking. This method is referred to in our graphs and the rest of the paper `epoll-ET`.

The second method, which we refer to as

epoll2, simply calls `epoll_ctl` twice per socket descriptor. The first to register with the kernel that the server is interested in read and write events on the socket. The second call occurs when the socket is closed. It is used to tell epoll that we are no longer interested in events on that socket. All events are handled in a level-triggered fashion. Although this approach will reduce the number of `epoll_ctl` calls, it does have potential disadvantages.

One disadvantage of the epoll2 method is that because many of the sockets will continue to be readable or writable `epoll_wait` will return sooner, possibly with events that are currently not of interest to the server. For example, if the server is waiting for a read event on a socket it won't be interested in the fact that the socket is writable until later. Another disadvantage is that these calls return sooner, with fewer events being returned per call, resulting in a larger number of call. Lastly, because many of the events will not of interest to the server the server is required to spend a bit of time to determine if it is or isn't interested in each event and in discarding events that aren't of interest.

The third method uses a new system call named `epoll_ctlv`. This system call is designed to reduce the overhead of multiple `epoll_ctl` system calls by aggregating several calls to `epoll_ctl` into one call to `epoll_ctlv`. This is achieved by passing an array of epoll events structures to `epoll_ctlv`, which then calls `epoll_ctl` for each element of the array. Events are generated in level-triggered fashion. This method is referred to in the figures and the remainder of the paper as epoll-ctlv.

We use `epoll_ctlv` to add socket descriptors to the interest set, and for modifying the interest sets for existing socket descriptors. However, removal of socket descriptors from the interest set is done by explicitly calling `epoll_ctl` just before the descriptor is closed. We do not aggregate deletion operations because by the time `epoll_ctlv` is invoked, the μserver has closed the descriptor and the `epoll_ctl` invoked on that descriptor will fail.

The μserver does not attempt to batch the closing of descriptors because it can run out of available file descriptors. Hence, the epoll-ctlv method uses both the `epoll_ctlv` and the `epoll_ctl` system calls. Alternatively, we could rely on the `close` system call to remove the socket descriptor from the interest set (and we did try this). However, this increases the time spent by the μserver in `close`, and does not result in changes in performance. We verified this empirically and decided to explicitly call `epoll_ctl` to perform the deletion of descriptors from the epoll interest set.

4 Experimental Environment

The experimental environment consists of a single server and eight clients. The server contains dual 2.4 GHz Xeon processors, 1 GB of RAM, a 10,000 rpm SCSI disk, and two one Gigabit Ethernet cards. The clients are identical to the server with the exception of their disks which are EIDE. The server and clients are connected with a 24-port Gigabit switch. To avoid network bottlenecks, 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. The server machine runs a slightly modified version of the 2.6.5 Linux kernel in uniprocessor mode.

4.1 Workloads

This section describes the workloads that we used to evaluate performance of the μserver with the different event notification mechanisms. In all experiments, we generate HTTP loads using *httperf* [11], an open-loop workload generator that uses connection timeouts to generate loads that can exceed the capacity of the server.

Our first workload is based on the widely used SPECweb99 benchmarking suite [17]. We use httperf in conjunction with a SPECweb99 file set and synthetic HTTP traces. Our traces have been carefully generated to recreate the file classes, access patterns, and number of requests issued per (HTTP 1.1) connection that are used in the static portion of SPECweb99. The file set and server caches are sized so that the entire file set fits in the server's cache. This ensures that differences in cache hit rates do not affect performance.

Our second workload is called the one-byte workload. In this workload, the clients repeatedly request the same one byte file from the server's cache. We believe that this workload stresses the event dispatch mechanism by minimizing the amount of work that needs to be done by the server in completing a particular request. By reducing the effect of system calls such as `read` and `write` this workload isolates the differences due to the event dispatch mechanisms.

To study the scalability of the event dispatch mechanisms as the number of socket descriptors (connections) is increased, we use *idleconn*, a program that comes as part of the httperf suite. This program maintains a steady number of idle connections to the server (in addition to the active connections maintained by httperf). If any of these connections are closed idleconn immediately re-establishes them. We first examine the behaviour of the event dispatch mechanisms without any idle connections to study scenarios where all of the connections present in a server are active. We then pre-load the server with a number of idle connections and then run experiments. The idle connections are used to increase the number of simultaneous connections in order to simulate a WAN environment. In this paper we present experiments using 10,000 idle connections, our findings with other numbers of idle connections were similar and they are not presented here.

4.2 Server Configuration

For all of our experiments, the μserver is run with the same set of configuration parameters except for the event dispatch mechanism. The μserver is configured to use `sendfile` to take advantage of zero-copy socket I/O while writing replies. We use TCP_CORK in conjunction with `sendfile`. The same server options are used for all experiments even though the use of TCP_CORK and `sendfile` may not provide benefits for the one-byte workload.

4.3 Experimental Methodology

We measure the throughput of the μserver using different event dispatch mechanisms. In our graphs, each data point is the result of a two minute experiment. Trial and error revealed that two minutes is sufficient for the server to achieve a stable state of operation. A two minute delay is used between consecutive experiments, which allows the TIME_WAIT state on all sockets to be cleared before the subsequent run. All non-essential services are terminated prior to running any experiment.

5 Experimental Results

In this section we first compare the throughput achieved when using level-triggered epoll with that observed when using `select` and `poll` under both the one-byte and SPECweb99-like workloads with no idle connections. We then examine the effectiveness of the different methods described for reducing the number of `epoll_ctl` calls under these same workloads. This is followed by a comparison of the performance of the event dispatch mechanisms when the server is pre-loaded with 10,000 idle connections. Finally, we describe the results of experiments in which we tune the accept strategy used in conjunction with epoll-LT and epoll-ctlv to further improve their performance.

We initially ran the one byte and the SPECweb99-like workloads to compare the performance of the select, poll and level-triggered epoll mechanisms.

As shown in Figure 1 and Figure 2, for both of these workloads select and poll perform as well as epoll-LT. It is important to note that because there are no idle connections for these experiments the number of socket descriptors tracked by each mechanism is not very high. As expected, the gap between epoll-LT and select is more pronounced for the one byte workload because it places more stress on the event dispatch mechanism.

We tried to improve the performance of the server by exploring different techniques for using epoll as described in Section 3. The effect of these techniques on the one-byte workload is shown in Figure 3. The graphs in this figure show that for this workload the techniques used to reduce the number of `epoll_ctl` calls do not provide significant benefits when compared with their level-triggered counterpart (epoll-LT). Additionally, the performance of select

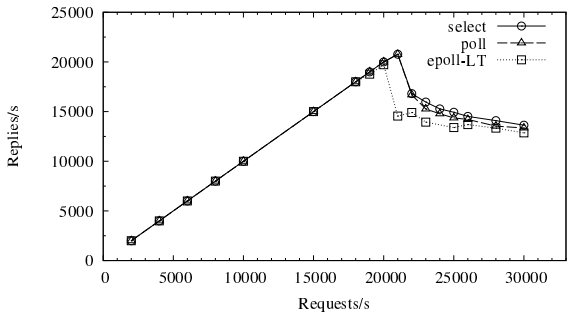

Figure 1: *µserver performance on one byte workload using select, poll, and epoll-LT*

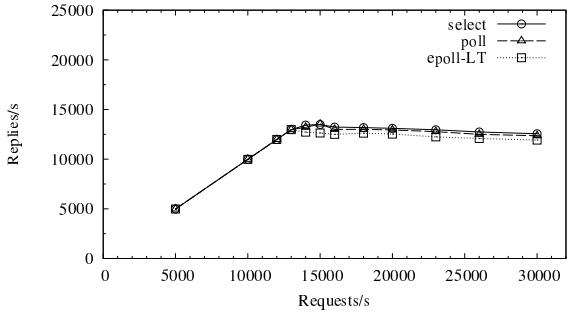

Figure 2: *µserver performance on SPECweb99-like workload using select, poll, and epoll-LT*

and poll is equal to or slightly better than each of the epoll techniques. Note that we omit the line for poll from Figures 3 and 4 because it is nearly identical to the select line.

We further analyze the results from Figure 3 by profiling the µserver using gprof at the request rate of 22,000 requests per second. Table 1 shows the percentage of time spent in system calls (rows) under the various event dispatch methods (columns). The output for system calls and µserver functions which do not contribute significantly to the total run-time is left out of the table for clarity.

If we compare the select and poll columns we see that they have a similar breakdown including spending about 13% of their time indicating to the kernel events of interest and obtaining events. In contrast the epoll-LT, epoll-ctlv, and epoll2 approaches spend about

	select	epoll-LT	epoll-ctlv	epoll2	epoll-ET	poll
read	21.51	20.95	21.41	20.08	22.19	20.97
close	14.90	14.05	14.90	13.02	14.14	14.79
select	13.33	-	-	-	-	-
poll	-	-	-	-	-	13.32
epoll_ctl	-	16.34	5.98	10.27	11.06	-
epoll_wait	-	7.15	6.01	12.56	6.52	-
epoll_ctlv	-	-	9.28	-	-	-
setsockopt	11.17	9.13	9.13	7.57	9.08	10.68
accept	10.08	9.51	9.76	9.05	9.30	10.20
write	5.98	5.06	5.10	4.13	5.31	5.70
fcntl	3.66	3.34	3.37	3.14	3.34	3.61
sendfile	3.43	2.70	2.71	3.00	3.91	3.43

Table 1: gprof profile data for the μserver under the one-byte workload at 22,000 requests/sec

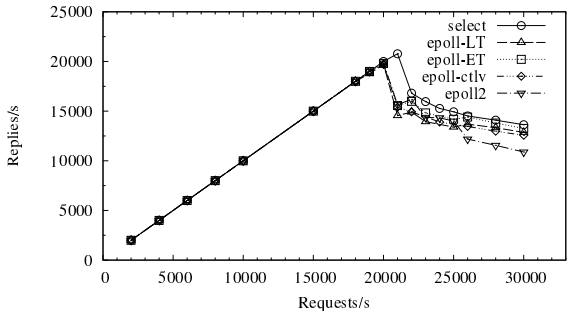

Figure 3: *μserver performance on one byte workload with no idle connections*

21 – 23% of their time on their equivalent functions (`epoll_ctl`, `epoll_ctlv` and `epoll_wait`). Despite these extra overheads the throughputs obtained using the epoll techniques compare favourably with those obtained using `select` and `poll`. We note that when using `select` and `poll` the application requires extra manipulation, copying, and event scanning code that is not required in the epoll case (and does not appear in the gprof data).

The results in Table 1 also show that the overhead due to `epoll_ctl` calls is reduced in epoll-ctlv, epoll2 and epoll-ET, when compared with epoll-LT. However, in each case these improvements are offset by increased costs in other portions of the code. The epoll2 technique spends twice as much time in `epoll_wait` when compared with epoll-LT. With epoll2 the number of calls to `epoll_wait` is significantly higher, the average number of descriptors returned is lower, and only a very small proportion of the calls (less than 1%) return events that need to be acted upon by the server. On the other hand, when compared with epoll-LT the epoll2 server spends about 6% less time on `epoll_ctl` calls so the total amount of time spent dealing with events is comparable with that of epoll-LT. Despite the significant `epoll_wait` overheads epoll2 performance compares favourably with the other methods on this workload.

Using the epoll-ctlv technique, gprof indicates that `epoll_ctlv` and `epoll_ctl` combine for a total of 1,949,404 calls compared with 3,947,769 `epoll_ctl` calls when using epoll-LT. While epoll-ctlv helps to reduce the number of user-kernel boundary crossings, the net result is no better than epoll-LT. The amount of time taken by epoll-ctlv in `epoll_ctlv` and `epoll_ctl` system calls is about the same (around 16%) as that spent by level-triggered epoll in invoking `epoll_ctl`.

When comparing the percentage of time epoll-LT and epoll-ET spend in `epoll_ctl` we see that it has been reduced using epoll-ET from 16% to 11%. Although the `epoll_ctl` time has been reduced it does not result in an appreciable improvement in throughput. We also note that about 2% of the run-time (which is not shown in the table) is also spent in the epoll-ET case checking, and tracking the state of the request (i.e., whether the server should be reading or writing) and the state of the socket (i.e., whether it is readable or writable). We expect that this can be reduced but that it wouldn't noticeably impact performance.

Results for the SPECweb99-like workload are shown in Figure 4. Here the graph shows that all techniques produce very similar results with a very slight performance advantage going to epoll-ET after the saturation point is reached.

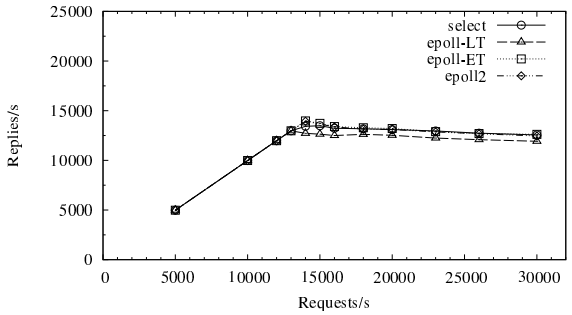

Figure 4: *µserver performance on SPECweb99-like workload with no idle connections*

5.1 Results With Idle Connections

We now compare the performance of the event mechanisms with 10,000 idle connections. The idle connections are intended to simulate the presence of larger numbers of simultaneous connections (as might occur in a WAN environment). Thus, the event dispatch mechanism has to keep track of a large number of descriptors even though only a very small portion of them are active.

By comparing results in Figures 3 and 5 one can see that the performance of select and poll degrade by up to 79% when the 10,000 idle connections are added. The performance of epoll2 with idle connections suffers similarly to select and poll. In this case, epoll2 suffers from the overheads incurred by making a large number of `epoll_wait` calls the vast majority of which return events that are not of current interest to the server. Throughput with level-triggered epoll is slightly reduced with the addition of the idle connections while edge-triggered epoll is not impacted.

The results for the SPECweb99-like workload with 10,000 idle connections are shown in Figure 6. In this case each of the event mechanisms is impacted in a manner similar to that in which they are impacted by idle connections in the one-byte workload case.

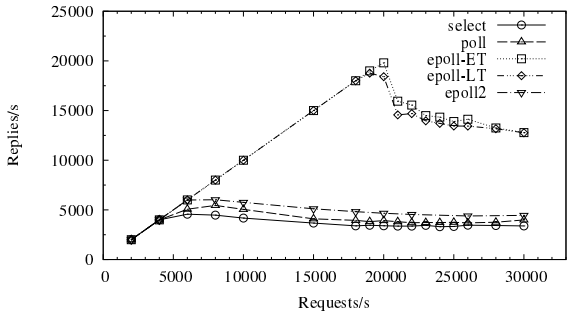

Figure 5: *µserver performance on one byte workload and 10,000 idle connections*

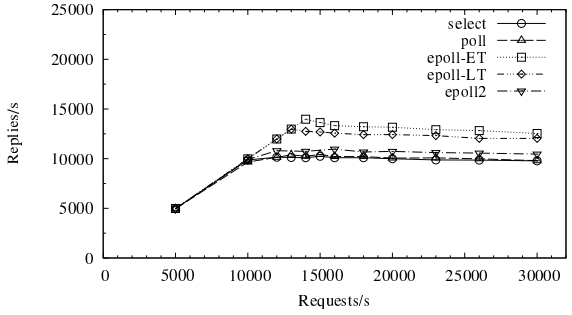

Figure 6: *µserver performance on SPECweb99-like workload and 10,000 idle connections*

5.2 Tuning Accept Strategy for epoll

The μserver's accept strategy has been tuned for use with `select`. The μserver includes a parameter that controls the number of connections that are accepted consecutively. We call this parameter the accept-limit. Parameter values range from one to infinity (Inf). A value of one limits the server to accepting at most one connection when notified of a pending connection request, while Inf causes the server to consecutively accept all currently pending connections.

To this point we have used the accept strategy that was shown to be effective for `select` by Brecht et al. [4] (i.e., accept-limit is Inf). In order to verify whether the same strategy performs well with the epoll-based methods we explored their performance under different accept strategies.

Figure 7 examines the performance of level-triggered epoll after the accept-limit has been tuned for the one-byte workload (other values were explored but only the best values are shown). Level-triggered epoll with an accept limit of 10 shows a marked improvement over the previous accept-limit of Inf, and now matches the performance of select on this workload. The accept-limit of 10 also improves peak throughput for the epoll-ctlv model by 7%. This gap widens to 32% at 21,000 requests/sec. In fact the best accept strategy for epoll-ctlv fares slightly better than the best accept strategy for select.

Varying the accept-limit did not improve the performance of the edge-triggered epoll technique under this workload and it is not shown in the graph. However, we believe that the effects of the accept strategy on the various epoll techniques warrants further study as the efficacy of the strategy may be workload dependent.

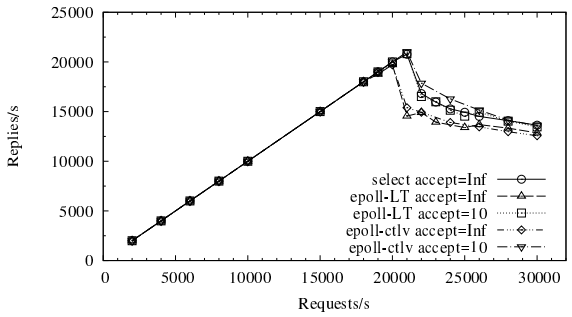

Figure 7: *μserver performance on one byte workload with different accept strategies and no idle connections*

6 Discussion

In this paper we use a high-performance event-driven HTTP server, the μserver, to compare and evaluate the performance of select, poll, and epoll event mechanisms. Interestingly, we observe that under some of the workloads examined the throughput obtained using `select` and `poll` is as good or slightly better than that obtained with epoll. While these workloads may not utilize representative numbers of simultaneous connections they do stress the event mechanisms being tested.

Our results also show that a main source of overhead when using level-triggered epoll is the large number of `epoll_ctl` calls. We explore techniques which significantly reduce the number of `epoll_ctl` calls, including the use of edge-triggered events and a system call, `epoll_ctlv`, which allows the μserver to aggregate large numbers of `epoll_ctl` calls into a single system call. While these techniques are successful in reducing the number of `epoll_ctl` calls they do not appear to provide appreciable improvements in performance.

As expected, the introduction of idle connections results in dramatic performance degradation when using `select` and `poll`, while not noticeably impacting the performance when

using epoll. Although it's not clear that the use of idle connections to simulate larger numbers of connections is representative of real workloads, we find that the edge-triggered epoll mechanisms perform well with the level-triggered epoll mechanisms offering performance that is very close.

In the future we plan to re-evaluate some of the mechanisms explored in this paper under more representative workloads that include more representative wide area network conditions. The problem with the technique of using idle connections is that the idle connections simply inflate the number of connections without doing any useful work. We plan to explore tools similar to Dummynet [16] and NIST Net [5] in order to more accurately simulate traffic delays, packet loss, and other wide area network traffic characteristics, and to re-examine the performance of Internet servers using different event dispatch mechanisms and a wider variety of workloads.

7 Acknowledgments

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

References

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

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

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

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

[5] M. Carson and D. Santay. NIST Net – a Linux-based network emulation tool. *Computer Communication Review*, to appear.

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

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

[8] Dan Kegel. The C10K problem, 2004. Available at `http://www.kegel.-com/c10k.html`.

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

[10] Davide Libenzi. Improving (network) I/O performance. Available at `http://www.xmailserver.-org/linux-patches/-nio-improve.html`.

[11] 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.

[12] Shailabh Nagar, Paul Larson, Hanna Linder, and David Stevens. epoll scalability web page. Available at `http://lse.sourceforge.net/epoll/index.html`.

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

[14] 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.

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

[16] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

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

[18] David Weekly. /dev/epoll - a highspeed Linux kernel patch. Available at `http://epoll.hackerdojo.com`.