Exploring the Performance of Select-based Internet Servers

Tim Brecht, Michal Ostrowski[1]
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-314
November 28th , 2001*

E-mail: brecht@hpl.hp.com, mostrows@us.ibm.com

Internet servers, Web servers, performance, select, event dispatching

Several previous studies have examined techniques for improving Internet server performance by investigating and improving operating system support for event-dispatching mechanisms. These studies have been mainly motivated by the commonly held belief that the overhead incurred in obtaining network I/O events using the select system call was too high to be used in environments with high request loads and large numbers of concurrent connections. However, recent work by Chandra and Mosberger [6] shows that a properly implemented select-based server can outperform servers that utilize new and improved event-dispatching mechanisms that are significantly more scalable than select.

In this work we conduct a detailed investigation into precisely why the Chandra and Mosberger server performs well. We use our findings to further improve the performance of select-based servers and provide insights that we believe are key to understanding and improving the performance of Internet server applications.

We find that an essential ingredient required to obtaining good performance in heavily loaded Internet servers is the ability to control the flow of work into and through the server. By employing controls that enable the server to both accept new connections at a high rate and make forward progress on existing connections, we are able to increase peak server throughput by 14 - 21% when compared with the server used by Chandra and Mosberger. Additionally, the differences in performance obtained using our new approach increase substantially as the workload intensifies.

* Internal Accession Date Only Approved for External Publication
[1] IBM TJ Watson Research Center, Yorktown Heights, NY

1 Introduction

Internet-based (or network-centric) applications have experienced incredible growth in recent years and all indications are that such applications will continue to grow in number and in importance. How such applications should be structured and how they should interact with operating systems is the subject of much activity in the research community, where it is commonly believed that existing interfaces are ill-suited to supporting such applications [4],[18], [14].

Current approaches to building Internet server software suffer from the problem that if the demand from client applications exceeds the server's ability to handle the demand, the performance of the server and hence the performance seen by the clients degrades dramatically. That is, existing servers are not well-conditioned to load.

In fact, in such systems the throughput of the server approaches zero as the number of simultaneous requests to that server continues to grow. This is reflected in unpredictable and extremely long wait times, or a complete lack of response for many of the users of such systems. It is precisely during these periods of high demand when being able to service customers may be most important to those who are relying on the server. Examples of such periods occur during sharp changes in the stock market, breaking news events, and the Christmas shopping season.

Unfortunately, it is not practical or cost effective to provision a system in order to handle peak demands because the peak demand on web servers can be several to hundreds of times higher than the average demand [1] [17].

The goal of this work is to examine the impact of various design considerations on a simple `select`-based web-server to determine how such an application can be better conditioned to load.

2 Background and Related Work

Current approaches to implementing high-performance web 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. Note that nearly all Internet-based servers and services follow similar steps.

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.*

Several of these steps can block because they require interaction with a remote host, the network, a database or some other subsystem, or potentially a disk. Consequently, in order to provide high levels of performance the server must be able to simultaneously service partially completed connections and to quickly and easily multiplex those connections that are ready and able to be serviced (i.e., those for which the application would not have to block and wait). This may result in the need to be able to handle several thousands or tens of thousands of simultaneous connections [4].

Initial attempts to implement web servers handled concurrency issues by creating a separate thread of control for each new connection and relying on the operating system to automatically block and unblock threads appropriately. Unfortunately, threads consume significant amounts of resources and server architects found that it was necessary to restrict the number of executing threads [8] [4].

More recent approaches to high-performance server design treat each connection as a finite state machine (FSM) with transitions between states being triggered by the event being processed. Several connections are managed simultaneously by multiplexing between different connections (FSMs) with the server working on the connection on which forward progress can be made without invoking an operating system call that would block to wait for a result. This is accomplished by tracking the file and socket descriptors of interest and periodically querying the operating system for information about the state of these descriptors (using a system call like `select` or `poll`). The results of these calls indicate to the application which operations can be performed on which descriptors without causing the application to block. Obtaining this information is the key component in providing the ability to multiplex between outstanding connections.

Significant research 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 from the operating system [3] [12] [2] [4] [13] [14] [6]. These studies have been motivated by the belief that under high loads with a large number of concurrent connections, the overhead incurred by `select` (or similar calls) is prohibitive to implementing high-performance Internet servers. As a result they have mainly developed 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 required by the kernel to perform such operations (e.g., by only delivering one signal per descriptor in the case of `sigwaitinfo`).

Interestingly, recent work by Chandra and Mosberger [6], in addition to introducing operating system modifications designed to improve application performance, demonstrates that a rather simple modification to a `select`-based web-server (with a stock operating system) provides better performance than their attempts and the attempts of others to improving performance by modifying the operating system. They refer to this server as a "multi-accept" server because upon learning of a request for a new incoming connection, rather than accepting a single connection, it attempts to accept as many incoming connections as possible. Calls to `accept` are repeated until a value of `EWOULDBLOCK` is returned, which indicates that there are currently no more outstanding connections that can be established. The results of Chandra and Mosberger's experiments were contrary to conventional wisdom which believed that `select`-based servers perform poorly under high loads.

Our work in this paper is largely motivated by this recent work by Chandra and Mosberger [6]. We believe that their work demonstrates that even simple server designs exhibit a wide range of variation in performance that is not well understood. In this paper we attempt to characterize the behaviour of some of these design options and use these results to gain insight into some of the issues affecting server designs in general.

Our work differs from previous work in that we investigate a variety of techniques for improving the performance of web-servers by concentrating mainly on the software architecture of the server. Specifically, we are interested in determining which aspects of different server application designs contribute to, or help to prevent, server meltdown during periods of high load. The focus of much of this paper is on the interplay between how the server accepts new incoming connections, obtains information about kernel events, and uses that information to process existing connections. We believe that this approach has provided us with a better understanding of techniques that can be deployed within the application to significantly improve performance.

3 Methodology

In this study, we use as a starting point the micro-servers developed by Chandra and Mosberger [6] to examine scalable event-dispatching mechanisms in Linux. We focus on their "multi-accept" server which provides the highest performance of all of the servers and kernel dispatching mechanisms considered in their study. We use the core of this server to create a new, highly parameterized micro-server that is designed to permit us to quickly and easily explore a wide variety of options with respect to implementing various aspects of the web server.

This approach is both necessary and important. In addition to creating a framework within which different options can be explored, it ensures that any differences in performance are actually due to differences in the software architecture of the server and not due to other artifacts of the implementations being compared (e.g., differences in the caching algorithms or numbers of file descriptors being used).

While this approach is attractive, it is not without its drawbacks. Perhaps the main drawback is that it is not feasible to examine and compare all combinations of parameters. Fortunately we have found that we've been able to apply insights gained in some experiments to eliminate the need to explore some combinations of options. Although we have not done an exhaustive study of all combinations of options or performed a completely systematic elimination of different combinations, we have been able to explore combinations of options that result in quite significant improvements in server throughput. Interestingly, we've found some seemingly minor modifications to the server can have a significant influence on the resulting performance of the server.

4 Server Implementation

We have designed and implemented our server to permit the exploration of several issues related to the implementation of high-performance web-servers. Again, our goal is to be able to provide a controlled environment in which we can fairly and accurately compare various design and implementation options. Among some of the issues we have examined (not all are reported on in this paper) are how performance is impacted by: aggressively accepting new connections, the order in which the open socket descriptors are processed, the size of the listen queue used in accepting new connections, and how that choice interacts with the size of the TCP SYN queue. We have also explored how caching impacts the design of high-performance web-servers.

Figure 2 shows the basic structure of the server. This figure is also used to provide a context in which we describe the various options and how they modify the server's behaviour. The pseudo-code contains some annotations regarding command line options. For example, if the `[-s]` option is used we call `new_conns` which tries to more aggressively accept new connections by adding one or more calls to `accept` in the main server loop, prior to each call to `select`.

```
while(1) {
    if ([-s] server_loop_accepts) {
        // [-m #] controls multi-accept
        new_conns();
    }

    rdfds = readfds;
    wrfds = writefds;
    // find out which fds won't block
    n = select(...rdfds, wrfds...);

    // order based on [-o order]
    while (fd = iterate_over_fds()) {
        if (ISSET(fd, rdfds)) {
            if (fd = accept_fd) {
                // [-m #] controls multi-accept
                new_conns();
            } else {
                if ([-L 1] {
                    // loop until failure
                    while socket_readable(fd);
                } else {
                    // read from socket process request
                    socket_readable(fd);
                }
            }
        }

        if (ISSET(fd, wrfds)) {
            // write response to client socket
            socket_writable(fd);
        }

    } // while iterate
} // while(1)
```

Figure 2: *The basic structure of the main server loop.*

4.1 Server Parameters

We now provide a more complete list of options available for controlling how the server operates and explain how they are used to change the server's behaviour.

- `[-m #]` modifies the behaviour of `new_conns`. When this options is used `new_conns` calls `accept` repeatedly until it either returns `EWOULDBLOCK` or until # consecutive calls have resulted in connections. Without this option `new_conns` makes a single call to `accept` to attempt to accept a new connection. The [-m 0] option is used to mimic the multi-accept behaviour used in the Chandra and Mosberger [6] study (0 is used to represent no limit).
- `[-o order]` controls the order of file descriptor (fd) processing, where order is one of the following:
  `up`: checks fds from 0 to the maximum fd.
  `down`: checks fds from the maximum fd to 0.
  `fifo`: checks fds in the order from first accepted to last.
  `lifo`: checks fds in the order from last accepted to first.
  `writes-up`: checks fds in the order from 0 to the maximum fd but does the write check before the read check.
  `writes-down`: checks fds in the order from the maximum fd to 0 but does the write check before the read check.
  The default is `[-o down]` since this was what was done in the Chandra and Mosberger multi-accept server.
- `[-w]` specifies that the server is to call `socket_writable` to write the reply to the client's socket as soon as the request is parsed (i.e., from within `socket_readable`). The idea here and with the `[-r]` option is to attempt to speed the processing of the current connection by making a direct call to try to immediately make forward progress rather than waiting for the next call to select. The potential drawback is that the call might simply return `EWOULDBLOCK` if the operating system is unable to immediately process the system call (i.e., `write` in this case or `read` in the case of `[-r]`).
- `[-r]` specifies that the server is to call `socket_readable` to read the request as soon as the connection is accepted (i.e., from within `new_conns`). See `[-w]` for the motivation for this option.
- `[-s]` is used to force an extra `new_conns` call (potentially doing a multi-accept) prior to doing a `select` in the main server loop. The idea behind this option is to be more aggressive about accepting new connections.
- `[-g]` tells the server to try to get new connections after closing an existing connection.
- `[-C]` is used to turn caching on. Note that the real purpose of this option is to be able to eliminate the effects of file system accesses and to focus on the remaining system calls.
- `[-L 1]` specifies that server should loop when calling `socket_readable` until the call fails. This is included in order to completely recreate the behaviour of the Chandra and Mosberger [6] multi-accept server. We are uncertain why this was done but it might be to more quickly determine when the socket is closed. Typically the first read returns the request and the second returns 0 to indicate that the end of file has been reached (i.e., that the connection has been closed by the client). Note that our design considers alternative modifiers for `[-L]` but we currently consider only `[-L 1]`.

- `[-l #]` is used to set the server's listen queue length. The default is 128. This option did require us to modify our version of the Linux kernel to permit this call to work correctly for values larger than 128.
- `[-c #]` sets the maximum number of connections permitted. This option needs to take into account the maximum number of file descriptors available and is used to avoid running out of available file descriptors (i.e., no new connections can be accepted unless there are file descriptors available). Note that when caching is not used this may need to be less than one half of the maximum number of available file descriptors (since potentially each socket could require an open file). The default is `[-c 15000]`.

We provide this list of options to identify the range of parameters we have explored, even though experimental results for several of the options are not presented here. We found that the differences in throughput for some of the options was only significant when used in conjunction with other combinations of options that resulted in relatively poor performance (e.g., `[-o]`). Additionally, we found that although system calls are exercised differently with the `[-C]` option, the results obtained were not qualitatively different from those obtained without the option. As a result, we felt they would not add to the discussion and we have not included the results here.

5 Environment

All experiments are conducted using a dedicated 100 Mbps Fast Ethernet switch that connects client hosts to the server. The server executes on a 400 MHz Pentium-III based dual-processor HP NetServer LPr system running Linux 2.4.0-test7 in uniprocessor mode. The client load is generated using `httperf` [10] and ten B180 PA-RISC machines running HP-UX 11.0.

For each data point in the graphs shown in this paper we start a new copy of the server. This is done because the server collects several statistics of importance that we use in analyzing its behaviour and it permits us to collect `gprof` [7] statistics from each run. In a number of preliminary experiments comparing the results obtained with and without `gprof` we found that using `gprof` did not significantly alter the results of our experiments. Although `gprof` is unable to apply accurate accounting techniques during interrupts (it simply adds the time spent handling the interrupt to the function currently being executed) we found the output of `gprof` quite instructive when viewed at a fairly course grain.

Each experiment is conducted by having the clients attempt to provide the desired load for a duration of 2 minutes using a time-out period of 3 seconds for each connection. The 2 minutes is sufficiently long to stress any of the system and application resources that must be limited, but short enough to permit us to conduct a reasonable number of experiments. Any request that the server is unable to respond to is recorded by the client as an error. These errors may occur either because the server is unable to accept the connection or because it isn't able to provide a response before the client time-out period is exceeded. The client time-out of 3 seconds is also small enough to prevent retries if a connection can't be established. Unless otherwise noted, all client requests are for a one-byte file. This is done in order to place as much stress as possible on the web-server and the underlying operating system. Additionally, because of the high response rates obtained with these servers the network would become a bottleneck with files much larger than about 2 KB (at 4250 replies per second a 2 KB would consume about 70 Mbps).

We have modified the default maximum number of open files permitted to 32768. This is done using `/proc/sys/fs/file-max`. To accommodate the increased number of possible open files we have also increased the size of an `fd_set` by modifying the definition of __FD_SETSIZE to 32000. Additionally, we use a default TCP SYN queue size of 1024 using `/proc/sys/net/ipv4/tcp_max_syn_backlog`.

6 Experiments

In this section we conduct a number of experiments to evaluate the performance obtained using our web-server with several different options. This permits us to explore a variety of web-server implementation options and to compare their performance.

As mentioned previously, there is an extremely large number of combinations of parameters that could be explored. We have run substantially more experiments than described in this paper and rather than presenting all of those results we have tried to focus on some of the more interesting aspects of our findings.

Table 1 lists the servers discussed in the upcoming sections and provides a quick reference for options used in each case and how the combination of options changes the behaviour of the server.

6.1 Basic Configuration Alternatives

The first experiment we conduct is to verify that when using the appropriate set of parameters our server will execute identically to the multi-accept server used by Chandra and Mosberger [6] (who were kind enough to supply us with the source code for the micro-servers used in their study). The options required for this are `[-m 0 -r -w -g -L 1]`. We use this combination of options as a starting point for further exploration because Chandra and Mosberger [6] have shown that this server configuration outperforms several alternative options including those that use kernel modifications to support more scalable event-dispatching.

Figure 3 plots the combined request rate of all of the clients on the x-axis and the server's response rate (throughput) on the y-axis. From this graph we see that the throughput of the original multi-accept server and our server executing with the `[-m 0 -r -w -g -L 1]` options is nearly identical.

We also use Figure 3 to examine how some fairly minor modifications to the server's behaviour impacts the server's throughput. This graph also contains results that show how the throughput of our base-line multi-accept equivalent server is influenced if we remove the sets of options `[-L 1]`, `[-g]`, and `[-L1 -g]` to produce servers we will refer to as `[-m 0 -r -w -g]`, `[-m 0 -r -w -L 1]`, and `[-m 0 -r -w]`, respectively.

The results show that removing the `[-L 1]` option has mixed results (labelled `[-m 0 -r -w -g]`). Recall that this means that the server will call `socket_readable` (and hence `read`) only once, instead of attempting to call `read` until it fails. Without this option, while the request rate is less than 7000 requests per second, throughput is about the same or a bit worse than the base-line multi-accept version. However, for higher request rates there may be a benefit to performing the loop.

Perhaps more interesting are the results obtained by removing the `[-g]` option and by removing the `[-L1 -g]` options. The resulting throughput of these two versions of the server appears to be identical; our discussion centers around the version without the `[-g]` option, referred to as the `[-m 0 -r -w]` server.

Server Options	Server Behaviour
`[-m 0 -r -w -g -L 1]`	The multi-accept server. These options provide the behaviour of the server used by Chandra and Mosberger [6]. When the listening socket is ready (readable) `accept` is called repeatedly until it returns `EWOULDBLOCK` (`[-m 0]`). Attempt to make as much forward progress on the current file descriptor as possible; `socket_readable` and `socket_writable` are called from within `new_conns` and `socket_readable`, respectively (`[-r -w]`. Attempt to get accept new connections whenever an existing connection is closed (`[-g]`) and loop on calls to `socket_readable` (`[-L 1]`).
`[-m 0 -r -w -g]`	The multi-accept server without looping on calls to `socket_readable`.
`[-m 0 -r -w -L 1]`	The multi-accept server without trying to get new connections whenever a connection is closed.
`[-m 0 -r -w]`	The multi-accept server without trying to get new connections whenever a connection is closed and without looping on calls to `socket_readable`.
`[-m 0 -r -w -s]`	The multi-accept server without trying to get new connections whenever a connection is closed and without looping on calls to `socket_readable` but with extra calls to `new_conns` in the server loop to try to accept new connections more aggressively.
`[-m 0]`	A purely `select`-driven server with repeated attempts to accept new connections when the listening socket is ready (readable).
`[-m 0 -s]`	A purely `select`-driven server with repeated attempts to accept new connections when the listening socket is ready (readable) and with extra calls to `new_conns` in the server loop to try to accept new connections more aggressively.
`[-m 25]`	A purely `select`-driven server with repeated attempts to accept new connections when the listening socket is ready (readable) but with a limit of 25 consecutive connections accepted each time the listening socket is ready.
`[-m 75]`	A purely `select`-driven server with repeated attempts to accept new connections when the listening socket is ready (readable) but with a limit of 75 consecutive connections accepted each time the listening socket is ready.

Table 1: *List of servers tested, options used to create them, and how the options affect their behaviour.*

With the `[-m 0 -r -w]` server there is no longer a check for new connections when connections are closed. Note that when `[-g]` is enabled such checks are only performed if there are file descriptors available. The removal of the `[-g]` option results in a slight increase (6.7%) in peak throughput when compared with the multi-accept version. This comparative increase in throughput is maintained until the request rate reaches about 5500 requests per second. Then from 5500 to 6000 requests per second throughput is slightly degraded (by about 6.5% at 5750 requests per second). However, as the request rate increases, significant benefits are obtained and the throughput is better than the multi-accept version by more than a factor of 3 for request rates between 7000 and 10000 requests per second.

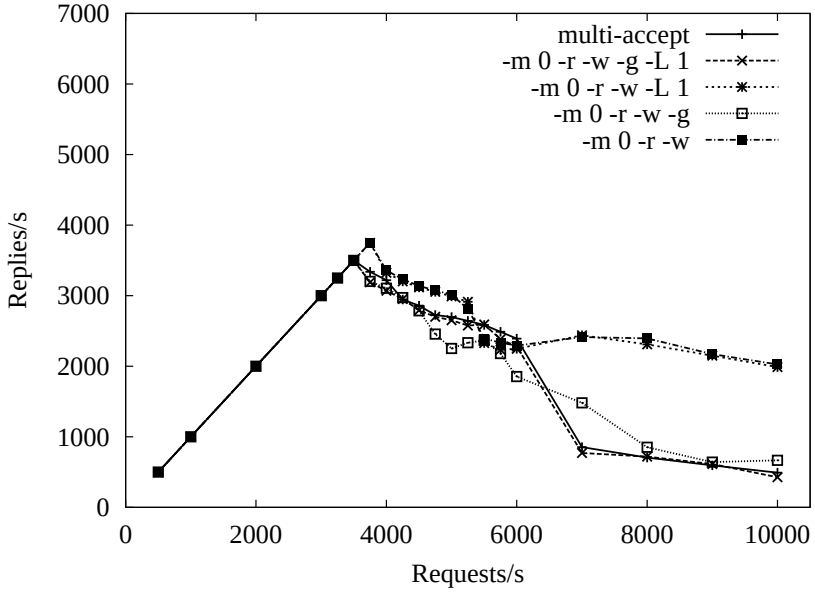

Figure 3: *Comparing the multi-accept server with slightly different options.*

To more closely examine some of the differences between how this version of the server and the baseline multi-accept version operate we look at the results obtained by various system calls when the request rate is 7000 requests per second. In the multi-accept version 20385 of the calls to `accept` (or 6.1%) return `EWOULDBLOCK` compared with 4152 calls (or 1.3%) in the `[-m 0 -r -w -L 1]` version.

More importantly, because there is an upper bound on the number of open file descriptors the server must carefully manage the use of this limited resource. As a result, it must stop accepting incoming requests for connections when that limit has been reached. In our `select`-based server implementation this is easily accomplished by simply excluding the file descriptor that is being used to accept incoming connections (the listening socket) from the `fd_set` being used to determine which descriptors can be read without blocking. Once a request has been completed and the associated socket is closed new connections can be accepted again by simply including the listening socket in the appropriate `fd_set`.

The multi-accept version of the server enters into a realm of execution where it is required to turn on and off the accepting of new connections, as it repeatedly bumps up against the file descriptor limit. This does not happen to the `[-m 0 -r -w]` server because it more actively makes forward progress on and closes existing connections. Consequently, the multi-accept server is unable to keep its pipeline full of connections to work on and throughput drops off significantly.

When the file descriptor limit is reached this situation might be improved by not permitting new connections to be accepted until a specified number of file descriptors becomes available. In fact, our server implements this option as `[-t #]`, where # specifies the threshold for the number of available descriptors required. However, we believe that alternative approaches that are presented in the remainder of the paper are preferred to this approach.

When we examined the `gprof` output for the multi-accept server for request rates between 7000 and 10000 requests per second we observed that it spends less than 0.5% of its execution time in `select` and about 13% of its execution time in `accept`. On the other hand, over the same request rates, the `[-m 0 -r -w]` server spends about 8% of its execution time in `accept` and from 5% (at 7000 requests per second) to 10% (at 10000 requests per second) in `select`.

Compared with the `[-m 0 -r -w]` server the multi-accept server spends proportionally more of its time attempting to accept connections (a greater percentage of which fail). The end result is that there are an insufficient number of connections to multiplex among and this is reflected in the relatively little time the multi-accept server spends in `select`.

Figure 4 compares the throughput obtained using the base-line multi-accept server, the combination of options that results in the best throughput in our previous experiment (`[-m 0 -w -r]` in Figure 3), and the results from some of the best combinations of options that we've explored. We also include a server that uses the options `[-m 0 -r -w -s]` because it will be used and discussed in more detail in the next section.

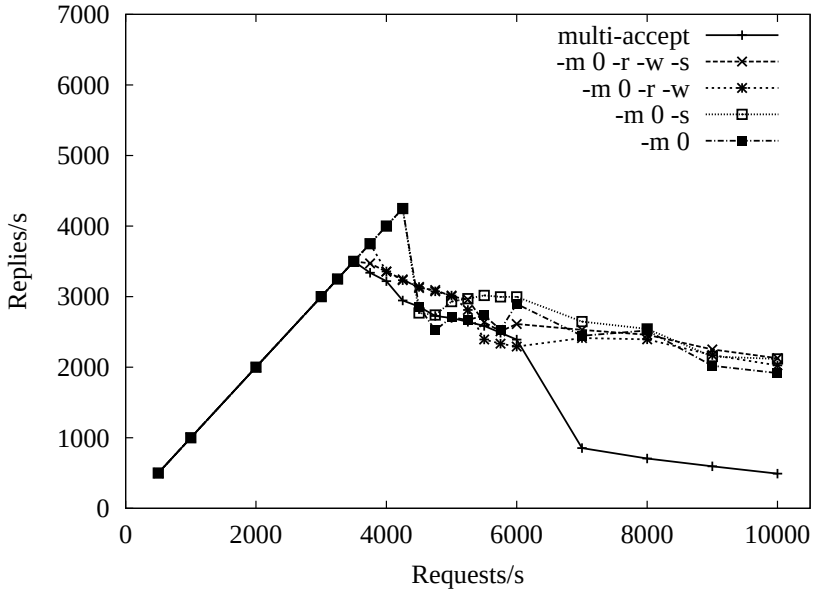

Figure 4: *Comparing the multi-accept server with some of the best observed combinations of options.*

As can be seen in Figure 4 the simple combinations of options `[-m 0]` and `[-m 0 -s]` obtain quite good results. Their peak throughput (or saturation point) is significantly higher than that of the multi-accept server. An increase in throughput of 21%, from 3500 to 4250 replies per second is obtained. Unfortunately, both of these servers suffer from a considerable degradation in throughput (a drop of about 35%) when the load only slightly exceeds the saturation point. This brings the throughput to a point below that obtained using the server with options `[-m 0 -w -r]` but close to or above the throughput obtained using the multi-accept server. It is both interesting and troubling that such a small percentage increase in load results in such a large percentage decrease in throughput.

To see where the `[-m 0 -s]` server is spending its time we examine the output from `gprof` for this server as a function of the load. Figure 5 graphs the `gprof` output for each of the main sources of execution time. The percentages are labeled on the y-axis on the left and the requests per second are labelled on the x-axis. The throughput obtained is also shown using lines with points and the y-axis on the right.

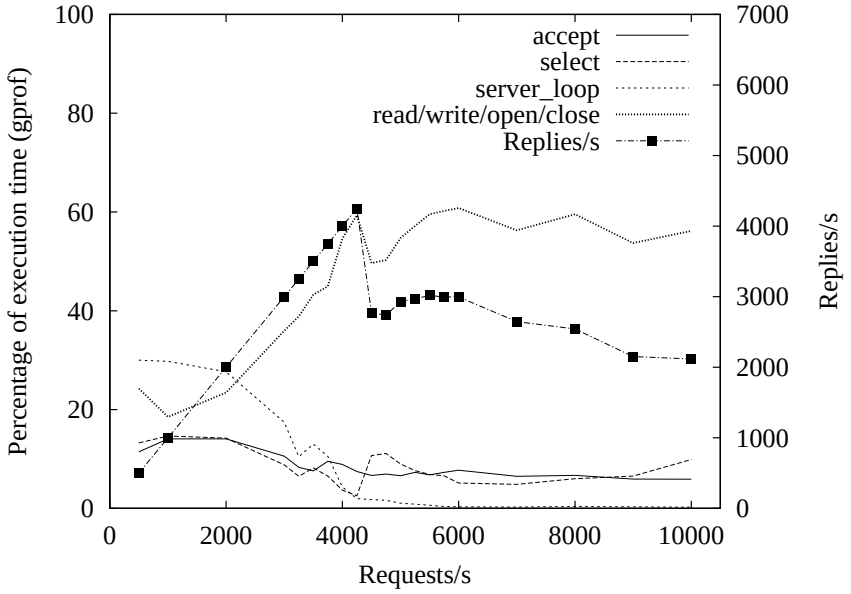

Figure 5: *Examining where the fraction of execution time is spent in the* `[-m 0 -s]` *server as the load increases.*

By comparing the dip in throughput when the load increases from 4250 to 4500 requests per second with the percentage of time spent in the major portions of the server, we see that there is a significant spike in the amount of time the server spends in `select`. This also corresponds to a dip in the percentage of time spent in `read`, `write`, `open`, and `close`. At 4250 requests per second this server is spending only about 2% of its execution time obtaining event information (i.e., in `select`) and 59% of its time processing requests (i.e., executing `read`, `write`, `open`, and `close`). At 4500 requests per second significantly more time (11%) is spent obtaining event information from `select` and a significantly smaller fraction of the execution time (50% compared with 59%) is now spent processing the requests.

Some of the details of the differences between the behaviour of the `[-m 0 -s]` server prior to and after saturation are shown in Table 2. We see that when the request rate increases from 4250 to 4500 requests per second the maximum number of consecutive connections accepted increases from 73 to 325. This increases the number of active file descriptors or simultaneous connections the server is multiplexing. This can be seen by the increase in the average maximum file descriptor passed into `select`, from 208 to 1074. This increases the work required by and overhead incurred by `select`. At 4250 requests per second the maximum number of ready file descriptors returned from `select` is 162, compared with 783 at 4500 requests per second. As noted previously this accounts for the sharp increase in the percentage of time spent in `select`.

Request/s	4250	4500
Calls to `new_conns`	48715	47445
Successful accepts	510000	358199
Max Consecutive accepts	73	325
Successful selects	38915	41497
Avg max fd into select	208	1074
Max fds from select	162	783
Avg fds from select	40	26

Table 2: *Comparing the details of accept and select calls.*

Although more work is being done by `select`, the average number of events (file descriptors) returned by select actually decreases from 40 at 4250 requests per second to 26 at 4500 requests per second. So even though more time is being spent calling `select` the average number of file descriptors that it is able to return as ready has actually decreased. The combination of more time being spent in `select` and its on average returning fewer ready file descriptors per call accounts for the decrease in the percentage of time the server spends making forward progress on existing connections (i.e., the time spend in `read`, `write`, `open`, and `close`). This is the main cause of the significant drop in the reply rate and demonstrates the need for a mechanism to ensure that the server is able to make forward progress on existing connections.

There is only a relatively minor difference between this `[-m 0 -s]` server and the `[-m 0 -r -w -s]` server. The `[-m 0 -r -w -s]` server by virtue of the addition of the `[-r -w]` options attempts to make as much forward progress on each connection as possible. Therefore, when a connection is accepted the server immediately tries to read from the socket or when the request is read and parsed the server immediately tries to write the result to the client socket. The results of these efforts can be seen in its `gprof` output which is shown in Figure 6. If we compare the `gprof` output of the `[-m 0 -r -w -s]` server at its saturation point (3750 request per second) with that of the `[-m 0 -s]` server at the same request rate, we see that the `[-m 0 -r -w -s]` server is spending 55% of its execution time working on existing connections (i.e., performing `read`, `write`, `open`, and `close` calls) compared with 45% for the `[-m 0 -s]` server. This shows that the `[-m 0 -r -w -s]` server is spending comparatively more of its time processing existing connections. In this case the server is attempting to ensure forward progress on existing connections at the expense of accepting new connections. This results in a lower response rate because it isn't accepting connections as fast as the `[-m 0 -s]` server.

6.2 Limiting New Connections

Neither the `[-m 0 -r -w -s]` server nor the `[-m 0 -s]` server discussed in detail in the previous section is very satisfying. The `[-m 0 -r -w -s]` server suffers from lower peak throughput in favour of a much smaller drop in performance once the saturation point is reached. On the other hand, the `[-m 0 -s]` server suffers from a significant drop in throughput once its saturation point is reached but benefits from a much higher saturation point. In this section we examine a technique for trying to ameliorate the severity of this drop in throughput while still maintaining high peak throughput. In a sense, we attempt to devise a server that is a compromise between these two approaches.

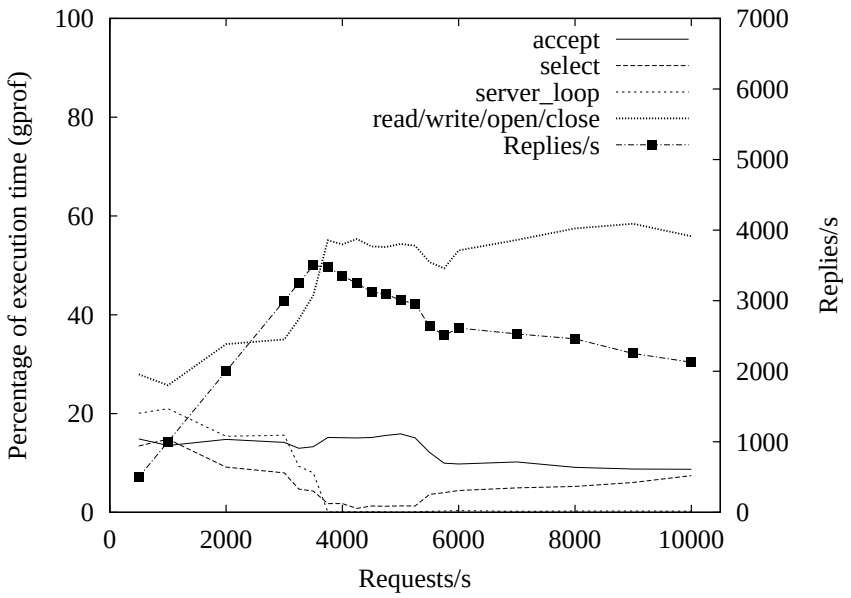

Figure 6: *Examining where the fraction of execution time is spent in the* `[-m 0 -r -w -s]` *server as the load increases.*

Our approach is to use the `[-m 0 -s]` server but to limit the number of consecutive connections that are accepted. As a first approximation of a limit we note that when the throughput peaks at 4250 requests per second the maximum number of consecutive connections accepted by the server was 73. We conducted a series of experiments with limits of 25, 50, 75, 100, and 125. Additionally, we conducted experiments with and without the `[-s]` option because the point of this series of experiments is to reduce and limit the number of new connections. We did find that in this case the `[-s]` option degrades throughput slightly. We also found that the `[-m 25]` server resulted in quite good peak throughput and did a slightly better job of avoiding the drop in throughput that others experienced when the request rate slightly exceeds the server's saturation point.

The graph in Figure 7 compares the throughput of the `[-m 25]`, `[-m 75]`, `[-m 0 -s]`, and the multi-accept servers. These experiments show that if one is willing to give up a minor amount of peak throughput the drawbacks resulting from the excessive emphasis on obtaining new connections by the `[-m 0 -s]` server can be ameliorated by limiting the number of new connections using the `[-m #]` option. As a result throughput improved significantly when the request rate is just slightly higher than the saturation point. One can also see that the `[-m 75]` server obtains slightly lower throughput than the `[-m 25]` server at 4250 and 4500 requests per second. This is because it is more aggressively accepting connections than the `[-m 25]` server and isn't expending enough resources in making forward progress on existing connections.

Figure 8 shows another view of how each of the different options influences the server's ability to successfully accept incoming requests for new connections. In this case the graph shows both the response rate and the rate at which the kernel drops incoming TCP SYN packets (QDrops/s). A TCP SYN packet is the first packet from a client initiating the three-way handshake required to establish a TCP connection. When the queue is full the kernel drops incoming packets. As mentioned previously, the size of the queue used in our experiments is 1024.

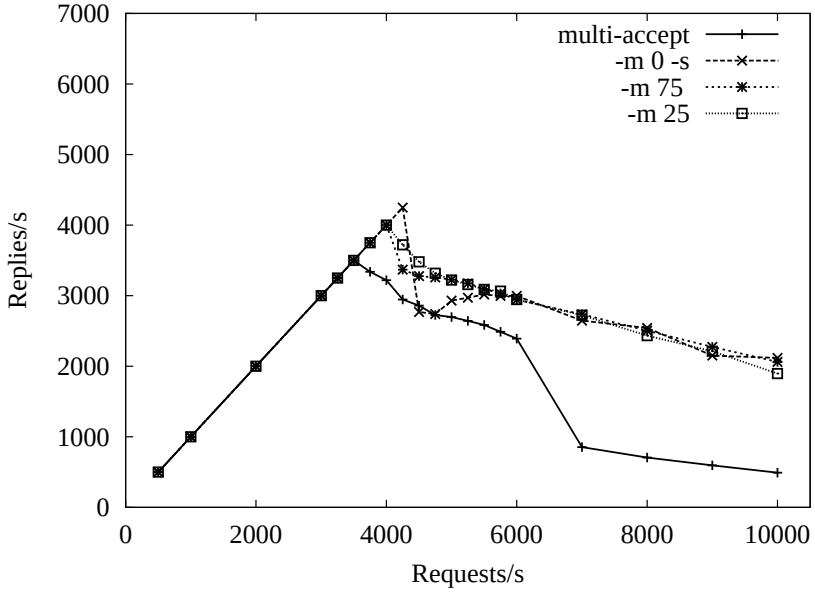

Figure 7: *Limiting the number of consecutive connections the server permits during a multi-accept and the influence on server throughput.*

By comparing where the reply rate drops with where the TCP SYN queue drop rate increases we can see that for some of these servers there is quite a large difference in the number of incoming TCP SYN packets that are dropped. All servers are able to avoid dropping TCP SYN packets until the saturation point is reached. But as can be seen in the graph this point is reached at different points for different servers.

With this comparison of reply rates and TCP SYN queue drop rates we are also able to see another reason that although the `[-m 0 -s]` version of the server aggressively accepts new connections its throughput suffers. As mentioned previously when the server is more aggressive about accepting new connections the period between successive calls to `select` is longer. As a result the number of file descriptors and events that must be handled by `select` is larger and the amount of work to process before returning to accept new connections increases. The result is that when the period between doing multi-accepts becomes too large, the TCP SYN queue quickly fills and the kernel must drop incoming TCP SYN packets. When using the `[-m 0 -s]` version of the server the kernel is forced to drop significantly more TCP SYN packets than when using other servers at the same request rate. This is another view of what causes the serious drop in throughput for the `[-m 0 -s]` version of the server.

Note that the time the kernel spends handling interrupts for packets that will be discarded because the TCP SYN queue is full also decreases the amount of time that the application can spend doing its work. We believe that this is one of the main reasons that none of the servers is able to sustain its peak response rate after the saturation point is reached. Note that the rate at which TCP SYN packets are dropped is very high relative to the request rate. We believe this is caused by `httperf` attempting to achieve the target request rate because it must be more aggressive when attempting to establish new connections when the failure rate is high.

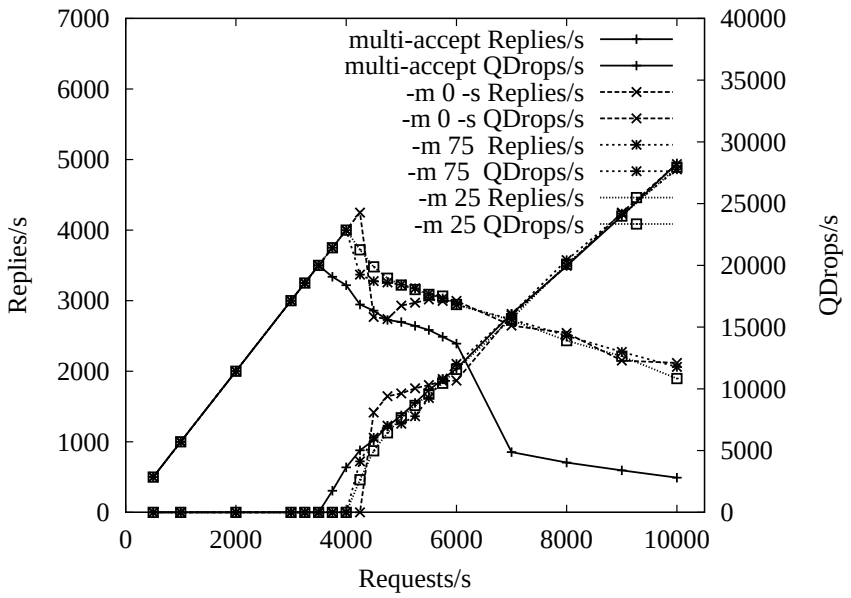

Figure 8: *Limiting the number of consecutive connections the server permits during a multi-accept and the influence on throughput and TCP SYN queue drop rates. The lines representing QDrops/s are those that start at zero on the right y-axis and rise between 3500 and 4500 requests per second. The lines representing Replies/s start at 500 on the left y-axis and drop between 3500 and 4500 requests per second.*

Figure 9 shows the `gprof` output for the `[-m 25]` server. This graph shows that as hoped, the proportion of time the `[-m 25]` server spends processing `select` calls avoids the sharp increase seen in the `[-m 0 -s]` server in Figure 5. This helps the `[-m 25]` server to avoid the drop in throughput that is seen by the `[-m 0 -s]` server when the load is slightly beyond the server's saturation point. We also see that the proportion of time spent in `accept` is reduced in the `[-m 25]` server (Figure 9) when compared with the `[-m 0 -r -w -s]` server (Figure 6) thus permitting the `[-m 25]` server to spend a greater portion of its execution time processing existing connections, making more forward progress on existing connections, and improving throughput.

Figure 10 shows the mean response times observed using the servers discussed in this section as a function of the request rate. Note that the mean recorded response times are only accurate to one millisecond. This graph is shown mainly to show that the increases in throughput are obtained without significant sacrifices in mean response times.

The mean response time provided by each server significantly degrades when that server reaches its saturation point. However, the multi-accept server attempts to make as much forward progress on new incoming connections as possible and maintains lower mean response times right up until reaching the saturation point. The other servers make forward progress on existing connections only following each call to select. Since forward progress is being made in what are essentially batches, the response times increase slightly as the saturation point is approached. Because the number of consecutive connections permitted with the `[-m 75]` option is slightly larger than with the `[-m 25]` option, its batches are larger and its response times are slightly higher just prior to reaching the saturation point than when using the `[-m 25]` option.

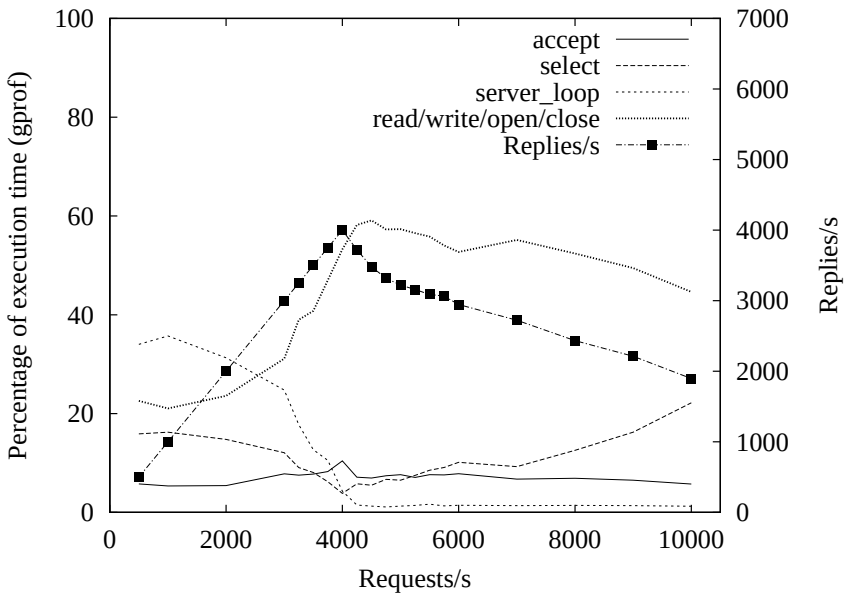

Figure 9: *Examining where execution time is being spent in the* `[-m 25]` *server as the load increases.*

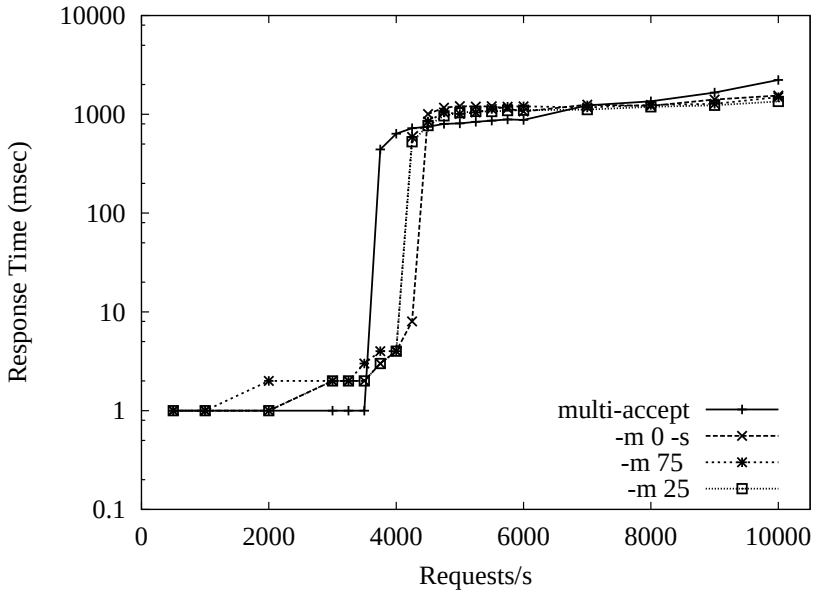

Figure 10: *Comparing the mean response times for different server options while increasing the load.*

7 Discussion

There is a similarity between the observations we make in this paper and observations made in work related to “receive livelock” [15] [9]. Receive livelock is referred to [15] as a condition where no useful progress

is being made in a system because a required resource is entirely consumed by the processing of receiver interrupts. Mogul and Ramakrishnan [9] demonstrate how this situation can be eliminated by carefully scheduling how work is performed. A fundamental difference between receive livelock and the problems we observe with not being able to make forward progress on existing connections is that receive livelock is a result of the kernel implementation (i.e., it is caused by the kernel) while our lack of forward progress is caused by the application itself. Therefore, the techniques used to improve performance in our case must be different from those used to eliminate receive livelock.

More recently, Provos *et al.* [14] observed a similar problem while studying the performance of their improved event-dispatch mechanism, `sigtimedwait4`. They found that while trying to reduce the overhead incurred in obtaining event information from the kernel that no noticeable improvement in performance could be seen when compared with the original approach. They were able to improve performance by modifying the application to observe the number of signals obtained on each call to `sigtimedwait4` and when the weighted average of the number of signals returned was too high (indicating an overload condition) they reset incoming connections instead of processing the request. In their environment, this is another method for improving the balance between accepting new and processing existing connections. The main difference between their work and our work in this paper is that they concentrate on trying to improve performance by reducing the amount of time required to obtain events from the kernel, while our work takes a different approach by trying to determine how a server should best apportion its time (instead of trying to reduce the time spent in individual elements of the server).

Other studies [5] [16] have also reset incoming connections in order to devote more time to processing existing connections. These and other studies centered around providing differentiated or improved quality of service are concerned with providing quality of service to some connections to the detriment of others. Unfortunately such approaches do not provide significant insights into improving the performance for all connections.

While we have examined techniques for managing the flow of work through an Internet server application, it is not our intent to determine an optimal strategy; our experience leads us to speculate that there is no silver bullet. Rather, our intent is to raise awareness of these issues; it is up to individual programmers to determine the best strategy for controlling the flow of work through their unique application. Our results in this paper demonstrate that even relatively simple server applications can experience significant changes in overload behavior with minor alterations. We also believe and emphasize that there is little to be gained in having highly efficient and scalable event management mechanisms in a server application if the application doesn't spend a sufficient, but not excessive, portion of its time making forward progress in servicing client requests.

8 Conclusions

In this paper we investigate the performance of a simple Internet server application built around the `select` system call. This is done in the context of a web-server application specifically implemented to permit us to study software architectures designed to avoid server meltdown during periods of high load.

We have constructed a parameterized micro web-server that permits us to explore a variety of server implementation options. This new server enables us to conduct fair and valid comparisons of different server

implementation options and ensures that any differences in performance are actually due to differences in the software architecture of the server and not due to other artifacts of the implementations being compared.

We have closely examined the performance of the `select`-based multi-accept server used by Chandra and Mosberger [6] and found that although its performance was better than methods that use alternate event-dispatch mechanisms, considerable improvements could still be made. First, significant improvements in peak throughput were obtained (14 – 21%) by more aggresively accepting incoming connections. We then show that servers that are overly aggressive in obtaining new connections can suffer from significant drops in performance after the saturation point of the server is reached. This is because insufficient resources are being applied toward making forward progress on existing connections. By limiting the number of consecutive connections accepted, we are able to strike a balance between rapidly accepting new incoming connections and making forward progress on existing connections. The resulting server not only improves peak throughput but also demonstrates a more gradual and stable deterioration of performance in the face of overload situations.

We believe that these results provide strong evidence that irrespective of the kernel event-dispatch method used, a balance must be maintained between accepting new connections, obtaining event information, and using that information to make forward progress on existing connections. When this balance is not maintained server throughput can degrade and in some cases this degradation can be substantial.

In the future we hope to clean up our code, improve the ease with which modifications can be made, and to make it available for others to use. We also hope to use it to study approaches to automatically controlling the flow of work through the server, to reexamine our results in the context of different and more realistic workloads, and to examine other techniques for providing kernel mechanisms that better support Internet-based server applications [11].

9 Acknowledgments

We wish to thank Abhishek Chandra and David Mosberger for providing us with the micro-server code used in their study [6]. We also wish to thank David Mosberger and Tai Jin for creating `httperf` [10], as well as Martin Arlitt for his updates and improvements and for numerous discussions related to improving Internet server performance. Thanks also to Todd Poynor and Brian Lynn of Hewlett Packard Research Labs for helpful comments on an earlier draft of this paper.

The authors also wish to thank the University of Waterloo (where much of this work got its start) and the Natural Sciences and Engineering Research Council of Canada for partial support for this research.

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] P. Bhoj, S Ramanathan, and S. Singhal. Web2K: Bringing QoS to web servers. Technical report, HP Laboratories, HPL-2000-61, May 2000.

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

[7] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPlan '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, 1982.

[8] 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.

[9] J. Mogul and K. Ramakrishnan. Eliminating receiver livelock in an interrupt-driven kernel. In *Proceedings of the USENIX Annual Technical Conference*, pages 99–111, San Diego, CA, 1996.

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

[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] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

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

[14] 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.

[15] K.K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proceedings of the IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, 1992.

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

[17] L.A. Wald and S. Schwarz. The 1999 Southern California seismic network bulletin. *Seismological Research Letters*, 71(4), July/August 2000.

[18] 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.