Scalable Network I/O in Linux

Niels Provos, *University of Michigan*
provos@citi.umich.edu

Chuck Lever, *Sun-Netscape Alliance*
chuckl@netscape.com

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu
http://www.citi.umich.edu/projects/linux-scalability

Abstract

Recent highly publicized benchmarks have suggested that Linux systems do not scale as well as other systems, such as Windows NT, when used as network servers. Windows NT contains features such as I/O completion ports that help boost network server performance and scalability. In this paper we focus on improving the Linux implementation of `poll()` *to reduce the expense of managing large numbers of network connections. We also explore the newer POSIX RT signal API that will help network servers scale into the next decade. A comparison between the two interfaces shows that a server using our* `/dev/poll` *interface scales better than a server using RT signals.*

1. Introduction

Many traditional web server benchmarks have focused on improving throughput for clients attached to the server via a high-speed local area network [13]. Recent studies have shown, however, that the difference between 32 high performance clients connected via gigabit Ethernet, and 32,000 high latency, low bandwidth connections from across the Internet, is extremely important to server scalability [8]. Connections that last only seconds do not place the same load on a server that slow error-prone connections do, due to resources consumed by error recovery, and the expense of managing many connections at once, most of which are idle.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

Experts on server architecture have argued that servers making use of a combination of asynchronous events and `poll()` are significantly more scalable than today's servers in these environments [2, 3, 5]. In Linux, signals can deliver I/O-completion events. Unlike traditional UNIX signals, POSIX Real-Time (RT) signals carry a data payload, such as a specific file descriptor that recently changed state. Signals with a payload enable network server applications to respond immediately to network requests, as if they were event driven. An added benefit of RT signals is that they can be queued in the kernel and delivered to an application one at a time, in order, leaving an application free to collect and process events when convenient.

The RT signal queue is a limited resource. When it is exhausted, the kernel signals a server to switch to polling, which delivers multiple completion events at once. Normally in a server like this, polling is simply an error recovery mechanism. However, the size of the RT signal queue might also be used as a load threshold to help network servers determine whether RT signals or the `poll()` interface is more efficient.

We have identified two areas of study. First, we demonstrate several modifications that improve `poll()`'s scalability and performance when a large proportion of a server's connections are inactive. Second, finding the right combination of RT signals and polling might allow network servers to leverage the latency advantages of completion notification against the throughput boosts of using `poll()`.

In this paper, we outline several improvements to the `poll()` interface, and measure the performance change of application using the improved `poll()`. We describe a test harness that simulates loads consisting of many inactive connections. We use this test harness to measure changes in application throughput as we vary the server's event model. Finally we report on our experiences using the new POSIX RT signals.

2. Using POSIX RT Signals: Introducing `phhttpd`

`Phhttpd` is a static-content caching front end for full-service web servers such as Apache [2, 10]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism added to the Linux kernel starting in the late 2.1.x kernel development series, and completed during the 2.3.x series. POSIX RT signals provide an event delivery system by allowing an application to assign unique signal numbers to each open file descriptor using `fcntl(fd, F_SETSIG, signum)`. The kernel raises the assigned signal whenever a `read()`, `write()`, or `close()` operation completes.

```
struct pollfd {
        int fd;
        short events;
        short revents;
};
```

Figure 1. Standard `pollfd` struct

```
struct siginfo {
        int si_signo;
        int si_errno;
        int si_code;
        union {
                /* other members elided */
                struct {
                        int _band;
                        int _fd;
                } _sigpoll;
        } _sifields;
} siginfo_t;
```

Figure 2. Simplified `siginfo` struct.

To avoid complexity and race conditions, the chosen RT signals are masked during normal server operation. An application uses `sigwaitinfo()` to pick up pending signals from the RT signal queue. `Sigwaitinfo()` returns a `siginfo` struct (see FIG. 2) for a single event. The `_fd` and `_band` fields in this struct contain the same information as the `fd` and `revents` fields in a `pollfd` struct (see FIG. 1).

The kernel raises `SIGIO` if the RT signal queue overflows. An application then flushes pending signals by changing their signal handler to `SIG_DFL`, and to recover, it uses `poll()` to discover any remaining pending activity.

Events queued before an application closes a connection will remain on the RT signal queue, and must be processed and/or ignored by applications. For instance, when a socket closes, a server application may receive and try to process previously queued read or write events before it picks up the close event, causing it to attempt inappropriate operations on the closed socket.

3. `poll()` Optimizations

There are two motivations for improving `poll()`. First, many legacy applications can benefit, with little or no modification, from performance and scalability improvements in `poll()`. While the changes necessary to take advantage of a new interface might be few, an overall architectural update for legacy applications is usually unnecessary. Yet updating an application to use POSIX RT signals is a major overhaul with a concomitant increase in complexity. A second purpose for improving `poll()` is that even with POSIX RT signals, `poll()` is still required to handle special cases such as RT signal queue overflows. An efficient `poll()` implementation helps performance and scalability of the new paradigm.

For the `poll()` interface to maintain performance comparable to the newer POSIX RT signal interfaces, it needs a face lift. We've enhanced `poll()` by making the following optimizations:

- We provide an interface that maintains state in the kernel, so state doesn't have to be passed in during every `poll()` invocation
- We allow device drivers to post completion events to `poll()`, reducing the need to invoke device-specific poll operations when scanning for events
- We eliminate result copying when `poll()` returns by creating a special address space mapping that is shared between the kernel and the application

In this section we describe these changes and evaluate their respective performance implications.

3.1 Maintaining State in the Kernel

To invoke `poll()` an application must build a set of interests, where an interest is a file descriptor that may have I/O ready, then notify the kernel of all interests by passing it the complete set of interests via `poll()`. As the number of interests increases, this mechanism becomes unwieldy and inefficient. The entire set must be copied into the kernel upon system call entry. The kernel must parse the entire interest set to carry out the request. Then each interest must be checked individually to assess its readiness.

Banga, Druschel, and Mogul have described new operating system features to mitigate these overheads [4]. They suggest that the `poll()` interface itself can be broken into one interface used to incrementally build an application's interest set within the kernel, and another interface used to wait for the next event. They refer to the first interface as `declare_interest()`, while the second is much like today's `poll()`. Using `declare_interest()`, an application can build its interest set inside the kernel as connections are set up and torn down. The complete interest set is never copied between user space and kernel space, completely eliminating unnecessary data copies (for instance, when there is no change in the interest set between two `poll()` invocations).

Recent versions of Solaris include a similar interface called `/dev/poll` [6]. This character device allows an application to notify the kernel of event interests and to build a (potentially) very large set of interests while reducing data copying between user space and kernel space. As far as we know this is the first real implementation of `declare_interest()`. We chose to implement this because, if it is effective, it will allow easier portability of high-performance network applications between Solaris and Linux.

An application opens `/dev/poll` and receives a file descriptor. The kernel associates an interest set with this file descriptor. A process may open `/dev/poll` more than once to build multiple independent interest sets. An application uses `write()` operations on `/dev/poll` to maintain each interest set.

Writing to `/dev/poll` allows an application to add, modify, and remove interests from an interest set. Applications construct an array of standard `pollfd` structs, one for each file descriptor in which it is interested (see Figure 1). Enabling the `POLLREMOVE` flag in the `events` field indicates the removal of an interest. Specifying a file descriptor the kernel already knows about allows an application to modify its interest. The contents of the events field replace the previous interest, unlike the Solaris implementation, where the `events` field is OR'd with the current interest. If complete Solaris compatibility is desired, this behavior can be adjusted with a minor modification to the device driver.

```
struct dvpoll {
        struct pollfd* dp_fds;
        int dp_nfds;
        int dp_timeout;
}
```

Figure 3. `dvpoll` struct

To wait for I/O events, an application issues an `ioctl()` with a `dvpoll` struct (see FIG. 3). This struct indicates how long to wait, and specifies a return area for the results of the poll operation. In general, only a small subset of an application's interest set becomes ready for I/O during a given poll request, so this interface tends to scale well.

A hash table contains each interest set within the kernel. On average, hash tables provide fast lookup, insertion, and deletion. For simplicity, when the average bucket size is two, the number of buckets in the hash table is doubled. The hash table is never shrunk.

3.2 Device Driver Hints

When an application registers interest in events on file descriptors with the `poll()` system call, the kernel passes this information to device drivers and puts the process to sleep until a relevant event occurs. When an application process wakes up, the kernel must scan all file descriptors in which the application has registered interest to check for status changes. This is the case even though the status of only one file descriptor in hundreds or thousands might have changed.

Now that we have an efficient mechanism for applications to indicate their interests, it would be useful if device drivers could indicate efficiently which file descriptors changed their status. We extend the `/dev/poll` implementation to make file descriptor information available to device drivers. The `/dev/poll` implementation maintains this information in a backmapping list. When an event occurs, the driver marks the appropriate file descriptor for each process in its backmapping list. When `poll()` scans an interest set to pick up new work, it uses this hint to avoid an expensive call to the device driver's poll callback. When managing a large number of high latency connections, this greatly reduces the number of driver poll operations that show that nothing has changed.

Hints allow `poll()` to determine if a cached result from a previous poll call is still valid. Specifically, a hint indicates a change in the socket's status, so it is time to invoke the device driver's poll callback. This also erases the current hint. We cache the result returned by the device driver, in the hope that we can reuse it without having to invoke the poll callback again soon. We do not receive hints that indicate the change from ready to not-ready, however. This means that a cached result indicating readiness has to be reevaluated each time.

To prevent the hinting system from requiring every device driver to be modified, device drivers indicate whether they support hinting. In this way, only essential drivers must be modified, *e.g.* network device drivers.

At this point, all backmapping lists are protected by a single read-write lock. Hints require only a read lock, so the lock itself is generally not contended. The lock is held for writing only when the interest set is modified. Each socket gets its own backmap, so ideally the backmap lock should be added in a per-socket structure to reduce lock contention and improve cache line sharing characteristics on SMP hardware. Each per-socket lock requires an extra 8 bytes.

3.3 Reducing Result Copy Operations

As the list of file descriptors passed to `poll()` grows, the overhead to copy out and parse the results also increases. To reduce this overhead, we need to improve the way the kernel reports the results of a `poll()` operation. The safest and most efficient way to do this is to create a memory map shared between the application and kernel where the kernel may deposit the results of the `poll()` operation.

We added this feature to our `/dev/poll` implementation. The application invokes `mmap()` on `/dev/poll` to create the mapping. Results from `poll()` for that process are reported in that area until it is `munmap()`'d by the application. Usually, the size of the result set is small compared to the size of the entire interest set, so we do not expect this modification to make as significant an impact as `/dev/poll` and device driver hints.

To create the result area, an application invokes `ioctl(DP_ALLOC)` to allocate room for a specific number of file descriptor results. This is followed by an `mmap()` call on a previously opened `/dev/poll` file descriptor to share the mapping between the kernel and the application. When polling, an application uses `ioctl(DP_POLL)` and specifies a `NULL` in the `dp_fds` field (see FIG. 2). When the application is finished, it uses `munmap()` to deallocate the area, then it closes `/dev/poll` normally.

4. POSIX Real-Time Signals v. `poll()`

A fundamental question is how great a server workload is required before polling is a more efficient way to gather requests. Are there *any* times when polling is a better choice? In this section we address the following questions:

- How big is the difference in performance between POSIX RT signals and `poll()`. Naturally this depends on the application implementation, but something as crude as an order of magnitude number would still be useful. What are the factors that determine this difference in performance— inefficiencies in `poll()` itself, argument copying, and so on?
- How important is an efficient `poll()` implementation for good overall performance of an implementation based on POSIX RT signals?
- How complete is the POSIX RT signal interface and implementation? Is it easy to use? Are there races or performance issues? Is it easy to use in combination with threads and black-box libraries?

To study these questions, we compare the performance of polling and event-driven architectures with a benchmark. The benchmark indicates which parts of the performance curve are served better by a particular event model. Imagine a hybrid server that can switch between polling and processing incoming requests via RT signals.

- To reduce the latencies of polling models, the server uses RT signals to process incoming requests and to handle them as soon as they arrive.
- To manage resource exhaustion in the kernel, the server uses RT signals until the signal queue reaches its maximum length.
- To overcome the inefficiencies of one-at-a-time event handling, the server uses polling after its workload becomes heavy.

Such a server might use the RT signal queue maximum as a crossover point for two reasons. First, it is built into the RT signal interface. When the signal queue overflows, the application receives a signal indicating that the overflow occurred. A `poll()` is necessary at this point to make sure that no requests are dropped. Second, the queue length tracks server workload fairly well. As server workload increases, so does the RT signal queue length. Thus it becomes an obvious indicator to cause a workload-triggered switch between event-driven and polling modes.

By studying the behavior and performance at the crossover point between RT signals and polling in a hybrid server, we gain an understanding of each design's relative advantages. Before creating such an imaginary hybrid, we can run specific tests that show whether each model has appropriate complementary performance and scalability characteristics.

Additionally, in real servers using the RT signal queue, we'd like to be sure that queue overload recovery mechanisms (*i.e.* invoking `poll()` to clean up) do not make the overload situation worse due to poor performance. Even better, perhaps `poll()` can perform well enough relative to POSIX RT signals that we don't have to relegate it to managing overloads. Note that the RT signal queue maximum length is normally set high enough (1024 by default) that it is never exceeded in today's implementations.

5. Benchmark

Our test harness consists of two machines running Linux connected via a 100 Mbit/s Ethernet switch. The workload is driven by an Intel SC450NX with four 500MHZ Xeon Pentium III processors (512Kb of L2 cache each), 512Mb of RAM, and a pair of SYMBIOS 53C896 SCSI controllers managing several LVD 10KRPM drives. Our web server runs on custom-built hardware equipped with a single 400MHZ AMD K6-2 processor, 64Mb of RAM, and a single 8G 7.2KRPM IDE drive. The server hardware is small so that we can easily drive the server into overload. We also want to eliminate any SMP effects on our server, so it is has only a single CPU.

The benchmark clients are driven by `httperf` running on the four-way Pentium III [7]. The web servers are `thttpd`, a simple single-process event-driven web server that is easy to modify, and `phhttpd`, an experimental server created to demonstrate the POSIX RT signal interface [9, 2].

The `httperf` benchmark client provides repeatable server workloads. We vary the server implementation and try each new idea with fixed workloads. We are most interested in static content delivery as that exercises the system components we are interested in improving. A side benefit of these improvements is better dynamic content service.

Scalability is especially critical to modern network service when serving many high-latency connections. Most clients are connected to the Internet via high-latency connections, such as modems, whereas servers are usually connected to the Internet via a few high bandwidth low-latency connections. This creates resource contention on servers because connections to high-latency clients are relatively long-lived, tying up server resources, and they induce a bursty and unpredictable interrupt load on the server [8].

Most web server benchmarks don't simulate high-latency connections, which appear to cause difficult-to-handle load on real-world servers [5]. We've modified the `httperf` benchmark to simulate these slower connections to examine the effects of our improvements on more realistic server workloads. We add client programs that do not complete an http request. To keep the number of high-latency clients constant, these clients reopen their connection if the server times them out.

There are several system limitations that influence our benchmark procedures. There are only a limited number of file descriptors available for single processes; `httperf` assumes that the maximum is 1024. We modified `httperf` to cope dynamically with a large number of file descriptors. Additionally, we can have only about 60000 open sockets at a single point in time. When a socket closes it enters the TIMEWAIT state for sixty seconds, so we must avoid reaching the port number limitation. We therefore run each benchmark for 35,000 connections, and then wait for all sockets to leave the TIMEWAIT state before we continue with the next benchmark run.

Our benchmark configuration contains only a single client host and a single server host, which makes the simulated workload less realistic. However, our benchmark results are strictly for comparing relative performance among our implementations. We believe the results also give an indication of real-world server performance.

A web server's static performance depends on the size distribution of requested documents. Larger documents cause sockets and their corresponding file descriptors to remain active over a longer time period. As a result the web server and kernel have to examine a larger set of descriptors, making the amortized cost of polling on a single file descriptor larger. In our tests, we request a 6 Kbyte document, a typical `index.html` file from the CITI web site.

5.1 `/dev/poll` benchmark results

Our first series of benchmarks measures the scalability of using `/dev/poll` instead of the stock version of `poll()`. We use `httperf` to drive a uniprocessor web server running `thttpd`.

We run two series of tests. First, we test stock `thttpd` running on stock Linux 2.2.14, varying the load offered by `httperf` by adjusting the number of inactive connections. The second test is the same, but replaces the kernel with a 2.2.14 kernel that supports `/dev/poll`, and replaces `thttpd` with a version modified to use `/dev/poll` instead of `poll()`. A subset of the results of these two series of tests is shown in FIGS. 4 through 10. Each of these graphs represents data from a single run of the benchmark.

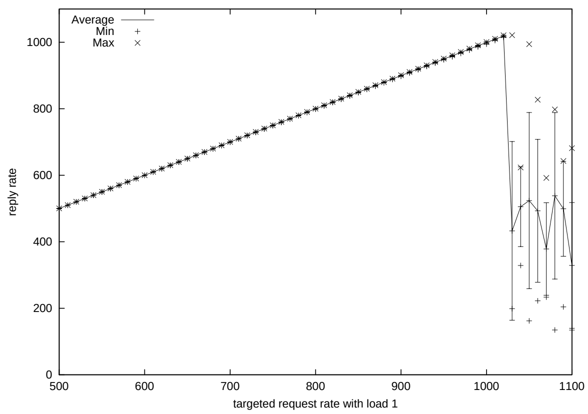

FIGURE 4. Normal `thttpd` using normal `poll()`, with one extra inactive connection. As expected, the server performs well when processing only active connections. After reaching a high enough request rate however, server performance breaks down as processing latency begins to exceed request rate.

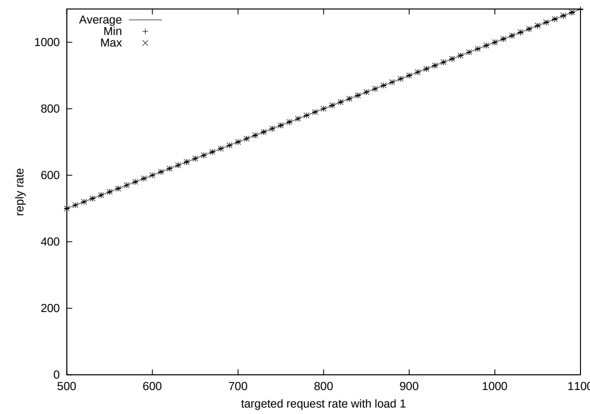

FIGURE 5. `thttpd` modified to use `/dev/poll`, with one extra inactive connection. With no inactive connections, the modified server performs well at all request rates. Unlike stock `thttpd`, there does not appear to be any point where processing latency exceeds request rate.

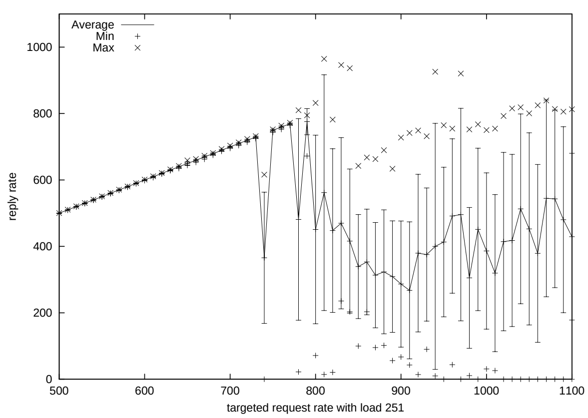

FIGURE 6. Normal `thttpd` using normal `poll()`, with 251 extra inactive connections. As load caused by inactive connections increases, processing latencies likewise increase. Server performance breaks down sooner, causing minimum response rates of zero in several places.

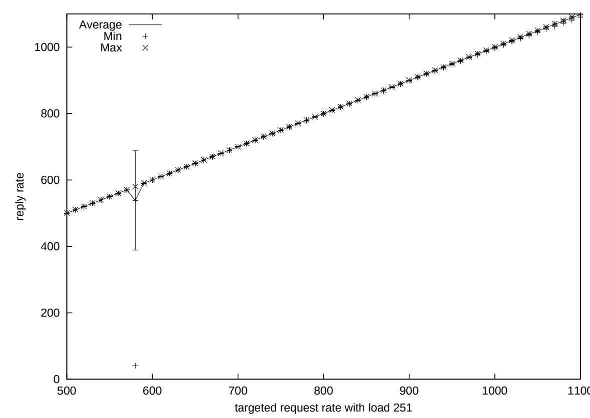

FIGURE 7. `thttpd` modified to use `/dev/poll`, with 251 extra inactive connections. With some inactive connections, the modified server performs almost as well as a server with no inactive connections.

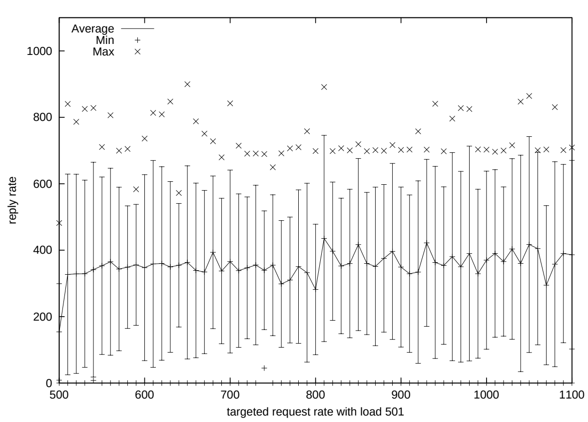

FIGURE 8. Normal `thttpd` using normal `poll()`, with 501 extra inactive connections. Latency due to processing inactive connections dominates server performance for all request rates, causing poor performance and high error rates.

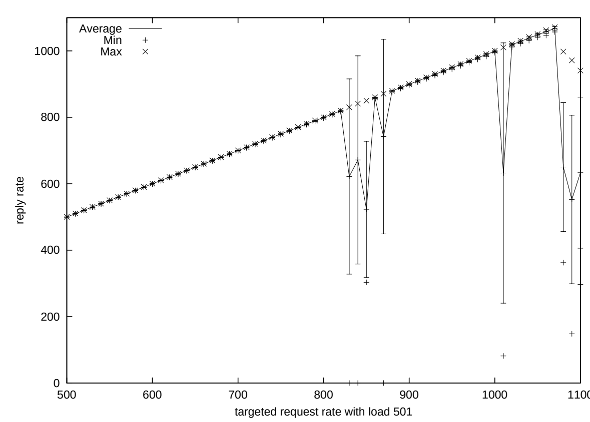

FIGURE 9. `thttpd` modified to use `/dev/poll`, with 501 extra inactive connections. Despite some response rate anomalies, the modified server manages a high inactive connection load with ease. Performance begins to break down at extreme high request rates.

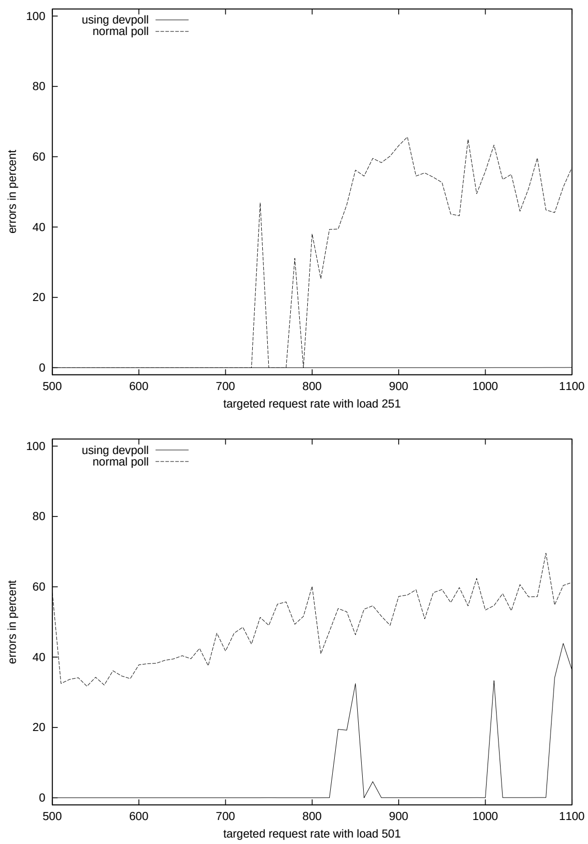

FIGURE 10. Error rate reported by `httperf` for stock `thttpd` and for `thttpd` modified to use `/dev/poll`. `httperf` maintains 251 inactive connections during the test shown in the top graph, and 501 inactive connections during the test shown in the bottom graph. `thttpd` using `/dev/poll` runs the test with 251 inactive connections with no errors whatsoever.

To simulate more realistic load on the server, we use an extra program to create inactive server connections. FIGS. 4 through 9 show the results of the benchmark for stock and modified `thttpd` with 1, 251 and 501 inactive connections. The graphs on the left show the results for stock `thttpd` using normal `poll()`. The graphs on the right the results for `thttpd` modified to use `/dev/poll`. Each graph plots the average response rate with error bars showing standard deviation against the request rate generated by the benchmark client. Ideally the generated request rate should match the server's response rate. The minimum and the maximum response rate for each run are also provided for comparison.

We observe a decrease in the average response rate as the number of inactive connections increases for both versions of `thttpd`. Some graphs show jumps in the maximum measured response rate while the minimum rate approaches zero, indicating that the server starves some connections.

`thttpd` using `/dev/poll` fully or partially achieves the desired response rate for all offered loads, as indicated by the data points showing maximum achieved response rate. On the other hand, the unmodified server is unable to maintain its throughput with increasing inactive connection load or increasing request rate. Its average response rate is smaller in all cases compared to `/dev/poll`. Banga and Drushel obtain a similar result [8].

FIG. 10 plots the percentage of connections aborted due to errors during runs with 251 and 501 inactive connections. Connection errors can result when the client runs out of file descriptors, when connections time out, or when the server refuses connections for some reason. For stock `thttpd`, the error rate increases slowly to 60% of all connections. `thttpd` using `/dev/poll` experiences only sporadic errors. In fact, when using `/dev/poll,` we measured no connection errors for benchmarks with fewer than 501 inactive connections.

Both the effective reply rate and the percent of connection errors demonstrate that `thttpd` using `/dev/poll` scales better than the unmodified version using `poll()`.

5.2 Comparing event models

Our second series of benchmarks is designed to compare the benefits of an RT signal-based event core with an event core designed around `poll()`. If they scale complementarily, it makes sense to try a hybrid server that switches between the two, triggered by server load.

FIGS. 11 through 13 illustrate the scalability of an unmodified single-threaded `phhttpd` server running on custom-built hardware equipped with a single 400MHZ AMD K6-2 processor, 64Mb of RAM, and a single 8G 7.2KRPM IDE drive. Our modified `httperf` client runs on an Intel SC450NX with four 500MHZ Xeon Pentium III processors (512Kb of L2 cache each), 512Mb of RAM, and a pair of SYMBIOS 53C896 SCSI controllers managing several LVD 10KRPM drives. Both machines are attached to a 100 Mbit/s Ethernet switch. The web server runs Linux 2.2.14 with complete support for RT signals back-ported from the 2.3 kernel series. The benchmark client runs stock Linux 2.2.14. Both machines are loaded with the Red Hat 6.1 distribution.

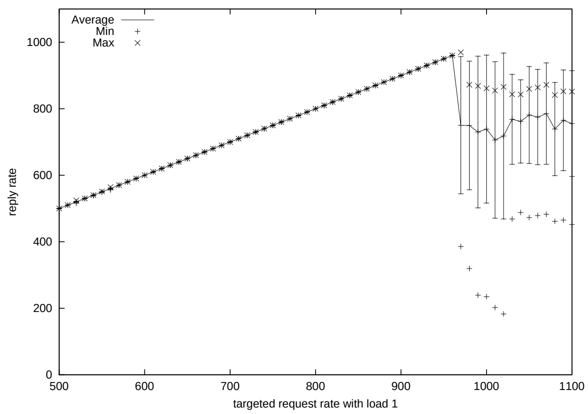

FIGURE 11. `phhttpd` with 1 extra inactive connection. Performance at lower request rates compares with the best performance of other servers. Very high request rates cause the server to falter, however. We believe this is due to the system call overhead of processing RT signals. During high loads, this overhead slows the server's ability to process requests.

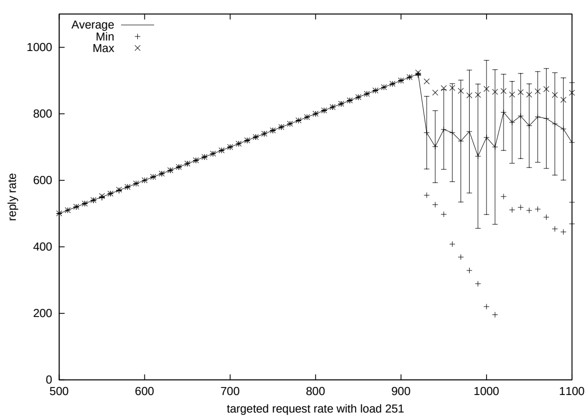

FIGURE 12. `phhttpd` with 251 extra inactive connections. With some inactive connections present, the server reaches its performance knee sooner. Inactive connections appear to increase the overhead of handling active connections, something that we didn't expect to find in a signals-based server implementation. This may be a problem with RT signals or with the `phhttpd` implementation itself.

As with the earlier `/dev/poll` benchmarks, we vary offered load by fixing the number of inactive connections, then we gradually increase the client request rate and record the corresponding server response rate. We compare a single-threaded `phhttpd` configuration against `thttpd`, a single process web server. Comparing FIGS. 11 through 13 with FIGS. 4, 6, and 8, clearly `phhttpd` outperforms the stock version of `thttpd`. However, comparing FIG. 11 to FIG. 5, we see that on the same hardware with few inactive connections, `thttpd` using `/dev/poll` responds more scalably to a higher load of active connections than does `phhttpd`.

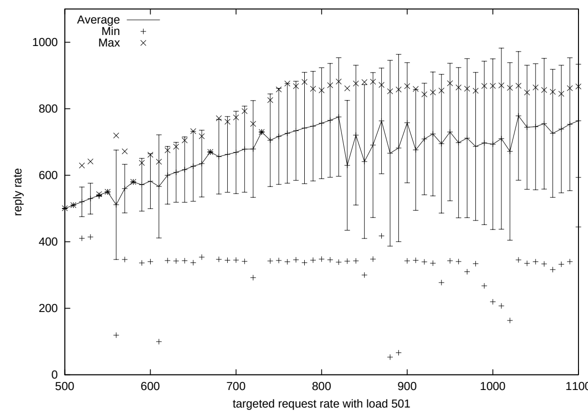

FIGURE 13. `phhttpd` with 501 extra inactive connections. In this test, load due to inactive connections appears to affect server throughput at all request rate levels. Compared to the throughput of `thttpd` using `/dev/poll`, this server scales less well.

The disparity between request and response rate increases markedly as more inactive connections are added to `phhttpd`'s load.

As FIG. 13 demonstrates, a heavy load of inactive connections causes `phhttpd` to perform worse than `thttpd` using `/dev/poll`, even at low request rates. Because `phhttpd` is unfinished and experimental, we believe that further refinements to `phhttpd` can improve its performance and scalability, but it is not clear whether it will perform better than `thttpd` based on `/dev/poll`.

An important benefit of using `/dev/poll` is that it scales well when a large number of inactive connections is present. However, even without any inactive connections `/dev/poll` scales better for high request rates compared to either stock `thttpd` or `phhttpd` using RT signals.

Another assumed advantage of RT signals is low latency. FIG. 14 shows median server response latency, in milliseconds. Median response latency evenly divides all measured responses at that load into half that are slower than the indicated result, and half that are faster. This measurement is a good reflection of a client's experience of a server's responsiveness. We see in FIG. 14 that `phhttpd` indeed serves requests with a median latency 1-3 milliseconds faster than the `/dev/poll`-based `thttpd` server across a wide range of offered load. After sufficiently high load, however, `phhttpd`'s median response latency leaps to over 120ms per request, while `thttpd`'s response increases only slightly.

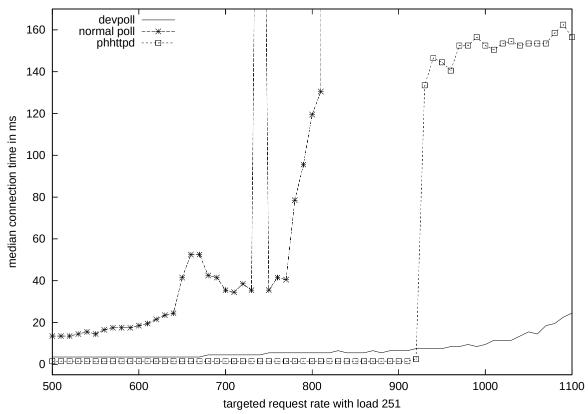

FIGURE 14. Median latency results of `phhttpd` with 251 extra inactive connections. For loads up to 900 concurrent `httperf` connections, `phhttpd` responds slightly faster than `thttpd` using `/dev/poll`. Above 900 concurrent connections, `phhttpd`'s connection latency jumps to over 120ms, whereas `thttpd`'s latency remains fairly steady. This is another indication that `thttpd` scales better than `phhttpd`.

6. Discussion and Future work

Originally we intended to modify `phhttpd` to use `/dev/poll` for these tests. After examining `phhttpd`, however, we saw that it completely rebuilds its poll interest set when recovering from RT signal queue overflow, negating any benefit to maintaining interest set state in the kernel rather than at the application level. Each thread that manages an RT signal queue for a listener socket has a partner thread that waits to handle RT signal queue overflow. When an overflow signal is raised, the thread managing the RT signal queue passes all of its current connections, including its listener socket, to its poll sibling, via a special UNIX domain socket. Considering that the server load is heavy enough to cause a queue overflow, the added work and inefficiency of transferring each connection one at a time and building a `pollfd` array from scratch will probably result in server meltdown.

When load subsides, the current `phhttpd` server does not switch from polling mode back to RT signal queue mode. Brown never implemented this logic [11].

To use either `poll()` or `/dev/poll` efficiently in `phhttpd`, we need to re-architect it. The RT signal queue overflow recovery mechanism should operate in the same thread as the RT signal queue handler. Additionally, RT signal queue processing should maintain its `pollfd` array (or corresponding kernel state) concurrently with RT signal queue activity. This would allow switching between polling and signal queue mode with very little overhead. Using `/dev/poll` without re-architecting this server won't help it scale unless it maintains its interest set concurrently with RT signal queue activity. Completely re-architecting `phhttpd` is beyond the scope of this paper. Future work may include a reworked server based on RT signals and `/dev/poll`.

Thus, modifying applications to use the `/dev/poll` interface efficiently requires more extensive changes to legacy applications than we had hoped. Applications of this type often entirely rebuild their `pollfd` array each time they invoke `poll()`, as `phhttpd` does.

Application developers may be tempted to treat POSIX RT signals like an interrupt delivery system. When used with signal handlers, signal delivery is immediate and asynchronous. However, when they are left masked and are picked up via `sigwaitinfo()`, POSIX RT signals behave much like `poll()`. The information delivered by a `siginfo` struct is the same as that in a `pollfd` struct, and, like `poll()`, it is provided synchronously when the application asks for it.

With `poll()`, however, the amount of data stored in the kernel is always bounded, because information about current activity on a file descriptor *replaces* previous information. However, managing this data in the kernel can become complex and inefficient as an application's interest set increases in size.

The POSIX RT signal queue receives a *new* item for any connection state change in a given interest set, and this item is simply added to the end of a queue. This necessitates a maximum queue limit and a special mechanism for recovering from queue overflow. Quite a bit of time can pass between when the kernel queues an RT signal and when an application finally picks it up. Sources of latency are varied: the kernel may need to swap in a stack frame to deliver a signal, lock contention can delay an application's response, or an application may be busy filling other requests. This means that a server picking up a signal must be prepared to find the corresponding connection in a different state. Later state changes that reflect the current state of the connection may be farther down the queue.

So, like the information contained in `pollfd` structs, events generated by `sigwaitinfo()` can be treated only as hints. Several connection state changes can occur before an application gets the first queued event indicating activity on a connection. Signals dequeue in order of their assigned signal number, thus activity on lower-numbered connections can cause longer delays for activity reports on higher-numbered connections.

Another difficulty arises from the fact that the Linux threading model is incompatible with POSIX threads when it comes to catching signals. POSIX threads run together in the same process and catch the same signals, whereas Linux threads are each mapped to their own `pid`, and receive their own resources, such as signals. It is not clear how RT signal queuing should behave in a non-Linux `pthread` implementation. Certainly there are some interesting portability issues here.

Several developers have observed that it is difficult to share a thread's POSIX RT signal queue among non-cooperative or black-box libraries [10, 11]. For instance, glibc's `pthread` implementation uses signal 32. If an application starts using pthreads after it has assigned signal 32 to a file descriptor via `fcntl()`, application behavior is undetermined. There appears to be no standard externalized function available to allocate signal numbers atomically in a non-cooperative environment.

Even when no signal queue overflow happens, the RT signal model may have an inherent inefficiency due to the number of system calls needed to handle an event on a single connection. This number may not be critical while server workload is easily handled. When the server becomes loaded, system call overhead may dominate server processing and cause enough latency that events wait a long time in the signal queue. To optimize signal handling, the kernel and the application can dequeue signals in groups instead of singly (similar to `poll()` today). We plan to implement a `sigtimedwait4()` system call which would allow the kernel to return more than one `siginfo` struct per invocation.

Future work in this area includes the addition of support in `phhttpd` for efficiently recovering from RT signal queue overflow to the signal worker thread. A closer look at `phhttpd`'s overall design may reveal weaknesses that could account for its performance in our tests. The use of specialized system calls such as `sendfile()` might also be interesting to study in combination with the new RT signal model.

There are several possible improvements to `/dev/poll`. Applications wishing to update their interest set and immediately poll on that set must use a pair of system calls, `write()` followed by `ioctl()`. A single `ioctl()` that handles both operations at once could improve efficiency. Our backmap scheme could benefit from finer grained locking, as described earlier in this paper. Sharing the result map among several threads may make a shared work queue possible. Also, improving hint caching can reduce even further the number of device driver poll operations required to obtain accurate `poll()` results.

A careful review of the current poll `wait_queue` mechanism might reveal areas for improved performance and scalability. Brown postulates that expensive `wait_queue` manipulation is where POSIX RT signals have an advantage over `poll()` [11]. The `wait_queue` mechanism is only invoked while no internal poll operation returns an event that would cause the process to wake up. Once such an event is found and it is known that the process will be awakened, the `wait_queue` is not manipulated further. To avoid `wait_queue` operations, file descriptors that have events pending should be polled first. We plan to modify our hinting system so that active connections are checked first during a poll operation. Managing each interest set with more efficient data structures in the kernel could improve performance even further. It may also help to provide the option of waking only one thread, instead of all of them.

7. Conclusion

Because of the amount of work required to poll efficiently in `phhttpd`, we were unable to directly test our theories about hybrid web servers for this paper. However, it is clear that, for our benchmark, `thttpd` using `/dev/poll` scales better than single-threaded `phhttpd` using RT signals at both low and high inactive connection loads. Once the number of inactive connections becomes large relative to the number of active connections, the difference in performance between polling and signaling exposes itself across all request rates. Latency results at lower loads favor `phhttpd`. As load increases, however, `thttpd` using `/dev/poll` maintains stable median response time, while `phhttpd` median response time increases by more than an order of magnitute. Surprisingly, it may never be better to use RT signals over a properly architected server using `/dev/poll`.

The POSIX RT signal interface is young, and still evolving. Today's signals-based servers are complicated by extra processing that may be unnecessary once developers understand RT signals better, and when OS implementations have improved. We expect further work in this area will improve their ease of use, performance, and scalability.

Software enhancements described herein are freely available. Please contact the authors for more information.

7.1. Acknowledgements

The authors thank Peter Honeyman and Stephen Tweedie for their guidance. We also thank the reviewers for their comments. Special thanks go to Zach Brown and Dan Kegel for their insights, and to Intel Corporation for equipment loans.

8. References

[1] G. Banga and J. C. Mogul, "Scalable Kernel Performance for Internet Servers Under Realistic

Load," *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[2] Z. Brown, *phhttpd*, people.redhat.com/zab/phhttpd, November 1999.

[3] Signal driven IO (*thread*), linux-kernel mailing list, November 1999.

[4] G. Banga. P. Drushel. J. C. Mogul, "Better Operating System Features for Faster Network Servers," *SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[5] J. C. Hu, I. Pyarali, D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrecy Models on Web Server Performance Over High-Speed Networks," *Proceedings of the 2nd IEEE Global Internet Conference*, November 1997.

[6] Solaris 8 man pages for poll(7d). docs.sun.com:80/ab2/coll.40.6/REFMAN7/@Ab2PageView/55123?Ab2Lang=C&Ab2Enc=iso-8859-1

[7] D. Mosberger and T. Jin, "httperf – A Tool for Measuring Web Server Performance," *SIGMETRICS Workshop on Internet Server Performance, June 1998*.

[8] G. Banga and P. Druschel, "Measuring the Capacity of a Web Server," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[9] thttpd - tiny/turbo/throttling web server. www.acme.com/software/thttpd

[10] Apache Server, The Apache Software Foundation. www.apache.org

[11] Z. Brown, personal communication, April 2000.

[12] J. Meyers, personal communication, May 1999.

[13] B. Weiner, "Open Benchmark: Windows NT Server 4.0 and Linux," www.mindcraft.com/whitepapers/openbench1.html