CITI Technical Report 00-7

Analyzing the Overload Behavior of a Simple Web Server

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

Chuck Lever, AOL-Netscape, Incorporated
<chuckl@netscape.com>

Stephen Tweedie, Red Hat Software
<sct@redhat.com>

ABSTRACT

Linux introduces POSIX Real Time signals to report I/O activity on multiple connections with more scalability than traditional models. In this paper we explore ways of improving the scalability and performance of POSIX RT signals even more by measuring system call latency and by creating bulk system calls that can deliver multiple signals at once.

August 24, 2000

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via grants from the Sun-Netscape Alliance, Intel, Dell, and IBM. For more information, see our home page.

If you have comments or suggestions, email <linux-scalability@citi.umich.edu>

Analyzing the Overload Behavior of a Simple Web Server

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

Chuck Lever, AOL-Netscape, Incorporated
<chuckl@netscape.com>

Stephen Tweedie, Red Hat Software
<sct@redhat.com>

1. Introduction

Experts on network server architecture have argued that servers making use of I/O completion events are more scalable than today's servers [2, 3, 5]. In Linux, POSIX Real-Time (RT) signals can deliver I/O completion events. Unlike traditional UNIX signals, RT signals carry a data payload, such as a specific file descriptor that just completed. Signals with a payload can 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 pick up I/O completion events when convenient.

The RT signal queue is a limited resource. When it is exhausted, the kernel signals an application to switch to polling, which delivers multiple completion events at once. Even when no signal queue overflow happens, however, RT signals may have inherent limitations due to the number of system calls needed to manage events on a single connection. This number may not be critical if the queue remains short, for instance while server workload is easily handled. When the server becomes loaded, the signal queue can cause system call overhead to dominate server processing, with the result that events are forced to wait a long time in the signal queue.

Linux has been carefully designed so that system calls are not much more expensive than library calls. There are no more cache effects for a system call than there are for a library call, and few virtual memory effects because the kernel appears in every process's address space. However, added security checks during system calls and hardware overhead caused by crossing protection domains make it expedient to avoid multiple system calls when fewer will do. Process switching is still comparatively expensive, often resulting in TLB flushes and virtual memory overhead. If a system call must sleep, it increases the likelihood that the kernel will switch to a different process. By lowering the number of system calls required to accomplish a given task, we reduce the likelihood of harm to an application's cache resident set.

Improving the scalability and reducing the overhead of often-used system calls has a direct impact on the scalability of network servers [1, 4]. Reducing wait time for blocking system calls gives multithreaded server applications more control over when and where requested work is done. Combining several functions into fewer system calls has the same effect.

In this paper, we continue work begun in "Scalable Network I/O for Linux" by Provos, *et al.* [9]. We measure the effects of system call latency on the performance and scalability of a simple web server based on an RT signal event model. Of special interest is the way server applications gather pending RT signals. Today applications use `sigwaitinfo()` to dequeue pending signals one at a time. We create a new interface, called `sigtimedwait4()`, that is capable of delivering multiple signals to an application at once.

We use `phhttpd` as our web server. `Phhttpd` is a static-content caching front end for full-service web servers such as Apache [8]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism, added to Linux during the 2.1 development series and completed during the 2.3 series [2]. We drive our test server with `httperf` [6]. An added client creates high-latency, low-bandwidth connections, as in Banga and Druschel [7].

Section 2 introduces POSIX Real-Time signals and describes how server designers can employ them. It also documents the `phhttpd` web server. Section 3 motivates the creation of our new system call. We describe our benchmark in Section 4, and discuss the results of the benchmark in Section 5. We conclude in Section 6.

2. POSIX Real-Time Signals and the `phhttpd` Web Server

In this section, we introduce POSIX Real-Time signals (RT signals), and provide an example of their use in a network server.

2.1 Using SIGIO with non-blocking sockets

To understand how RT signals provide an event notification mechanism, we must first understand how signals drive I/O in a server application. We recapitulate Stevens' illustration of signal-driven I/O here [10].

An application follows these steps to enable signal-driven I/O:

1. The application assigns a `SIGIO` signal handler with `signal()` or `sigaction()`.
2. The application creates a new socket via `socket()` or `accept()`.
3. The application assigns an owner pid, usually its own pid, to the new socket with `fcntl(fd, F_SETOWN, newpid)`. The owner then receives signals for this file descriptor.
4. The application enables non-blocking I/O on the socket with `fcntl(fd, F_SETFL, O_ASYNC)`.
5. The application responds to signals either with its signal handler, or by masking these signals and picking them up synchronously with `sigwaitinfo()`.

The kernel raises `SIGIO` for a variety of reasons:

- A connection request has completed on a listening socket.
- A disconnect request has been initiated.
- A disconnect request has completed.
- Half of a connection has been shut down.
- Data has arrived on a socket.
- A write operation has completed.
- An out-of-band error has occurred.

When using old-style signal handlers, this mechanism has no way to inform an application which of these conditions occurred. POSIX defines the `siginfo_t` struct (see FIG. 1), which, when used with the `sigwaitinfo()` system call, supplies a signal reason code that distinguishes among the conditions listed above. Detailed signal information is also available for new-style signal handlers, as defined by the latest POSIX specification [15].

This mechanism cannot say what file descriptor caused the signal, thus it is not useful for servers that manage more than one TCP socket at a time. Since its inception, it has been used successfully only with UDP-based servers [10].

2.2 POSIX Real-Time signals

POSIX Real-Time signals provide a more complete event notification system by allowing an application to associate signals with specific file descriptors. For example, an application can assign signal numbers larger than `SIGRTMIN` to specific open file descriptors using `fcntl(fd, F_SETSIG, signum)`. The kernel raises the assigned signal whenever there is new data to be read, a write operation completes, the remote end of the connection closes, and so on, as with the basic `SIGIO` model described in the previous section.

Unlike normal signals, however, RT signals can queue in the kernel. If a normal signal occurs more than once before the kernel can deliver it to an application, the kernel delivers only one instance of that signal. Other instances of the same signal are dropped. However, RT signals are placed in a FIFO queue, creating a stream of event notifications that can drive an application's response to incoming requests. Typically, to avoid complexity and race conditions, and to take advantage of the information available in `siginfo_t` structures, applications mask the chosen RT signals during normal operation. An application uses `sigwaitinfo()` or `sigtimedwait()` to pick up pending signals synchronously from the RT signal queue.

The kernel must generate a separate indication if it cannot queue an RT signal, for example, if the RT signal queue overflows, or kernel resources are temporarily exhausted. The kernel raises the normal signal `SIGIO` if this occurs. If a server uses RT signals to monitor incoming network activity, it must clear the RT signal queue and use another mechanism such as `poll()` to discover remaining pending activity when `SIGIO` is raised.

Finally, RT signals can deliver a payload. `Sigwaitinfo()` returns a `siginfo_t` struct (see FIG. 1) for each signal. The `_fd` and `_band` fields in this structure contain the same information as the `fd` and `revents` fields in a `pollfd` struct (see FIG. 2).

```
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 1. Simplified `siginfo_t` struct.

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

Figure 2. Standard `pollfd` struct

2.2.1 Mixing threads and RT signals

According to the GNU `info` documentation that accompanies `glibc`, threads and signals can be mixed reliably by blocking all signals in all threads, and picking them up using one of the system calls from the `sigwait()` family [16].

POSIX semantics for signal delivery do not guarantee that threads waiting in `sigwait()` will receive particular signals. According to the standard, an external signal is addressed to the whole process (the collection of all threads), which then delivers it to one particular thread. The thread that actually receives the signal is any thread that does not currently block the signal. Thus, only one thread in a process should wait for normal signals while all others should block them.

In Linux, however, each thread is a kernel process with its own PID, so external signals are always directed to one particular thread. If, for instance, another thread is blocked in `sigwait()` on that signal, it will not be restarted.

This is an important element of the design of servers using an RT signals-based event core. All normal signals should be blocked and handled by one thread. On Linux, other threads may handle RT signals on file descriptors, because file descriptors are "owned" by a specific thread. The kernel will always direct signals for that file descriptor to its owner.

2.2.2 Handling a socket close operation

Signals 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 previously queued read or write events before it picks up the close event, causing it to attempt inappropriate operations on the closed socket.

When a socket is closed on the remote end, the local kernel queues a `POLL_HUP` event to indicate the remote hang-up. `POLL_IN` signals occurring earlier in the event stream usually cause an application to read a socket, and when it does in this case, it receives an EOF. Applications that close sockets when they receive `POLL_HUP` must ignore any later signals for that socket. Likewise, applications must be prepared for reads to fail at any time, and not depend only on RT signals to manage socket state.

Because RT signals queue unlike normal signals, server applications cannot treat these signals as interrupts. The kernel can immediately re-use a freshly closed file descriptor, confusing an application that then processes (rather than ignores) `POLL_IN` signals queued by previous operations on an old socket with the same file descriptor number. This introduces to the unwary application designer significant vulnerabilities to race conditions.

2.3 Using RT Signals in a Web Server

`Phhttpd` is a static-content caching front end for full-service web servers such as Apache [2, 8]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism, added to the Linux kernel during the 2.1.x kernel development series and completed during the 2.3.x series. We describe it here to document its features and design, and to help motivate the design of `sigtimedwait4()`. Our discussion focuses on how `phhttpd` makes use of RT signals.

2.3.1 Assigning RT signal numbers

Even though a unique signal number could be assigned to each file descriptor, `phhttpd` uses one RT signal number for all file descriptors in all threads for two reasons.

1. Lowest numbered RT signals are delivered first. If all signals use the same number, the kernel always delivers RT signals in the order in which they arrive.

2. There is no standard library interface for multithreaded applications to allocate signal numbers atomically. Allocating a single number

once during startup and giving the same number to all threads alleviates this problem.

2.3.2 Threading model

`Phhttpd` operates with one or more worker threads that handle RT signals. Additionally, an extra thread is created for managing logs. A separate thread prepopulates the file data cache, if requested.

Instead of handling incoming requests with signals, `phhttpd` may use polling threads instead. Usually, though, `phhttpd` creates a set of RT signal worker threads, and a matching set of polling threads known as *sibling* threads. The purpose of sibling threads is described later.

Each RT signal worker thread masks off the file descriptor signal, then iterates, picking up each RT signal via `sigwaitinfo()` and processing it, one at a time. To reduce system call rate, `phhttpd read()`s on a new connection as soon as it has `accept()`ed it. Often, on high-bandwidth connections, data is ready to be read as soon as a connection is `accept()`ed. `Phhttpd` reads this data and sends a response immediately to prevent another trip through the "event" loop, reducing the negative cache effects of handling other work in between the accept and the read operations.

Because the read operation is non-blocking, it fails with `EAGAIN` if data is not immediately present. The thread proceeds normally back to the "event" loop in this case to wait for data to become available on the socket.

2.3.3 Load balancing

When more than one thread is available, a simple load balancing scheme passes listening sockets among the threads by reassigning the listener's owner via `fcntl(fd, F_SETOWN, newpid)`. After a thread accepts an incoming connection, it passes its listener to the next worker thread in the chain of worker threads. This mechanism requires that each thread have a unique pid, a property of the Linux threading model.

2.3.4 Caching responses

Because `phhttpd` is not a full-service web server, it must identify requests as those it can handle itself, or those it must pass off to its backing server. Local files that `phhttpd` can access are cached by mapping them and storing the map information in a hash, along with a pre-formed http response. When a cached file is requested, `phhttpd` sends the cached response via `write()` along with the mapped file data.

Logic exists to handle the request via `sendfile()` instead. In the long run, this may be more efficient for several reasons. First, there is a limited amount of address space per process. This limits the total number of cached bytes, especially because these bytes share the address space with the pre-formed responses, hash information, heap and stack space, and program text. Using `sendfile()` allows data to be cached in extended memory (memory addressed higher than one or two gigabytes). Next, as the number of mapped objects grows, mapping a new object takes longer. On Linux, finding an unused area of an address space requires at least one search that is linear in the number of mapped objects in that address space. Finally, creating these maps requires expensive page table and TLB flush operations that can hurt system-wide performance, especially on SMP hardware.

2.3.5 Queue overflow recovery

The original `phhttpd` web server recovered from signal queue overflow by passing all file descriptors owned by a signal handling worker thread to a preexisting poll-based worker thread, known as its *sibling*. The sibling then cleans up the signal queue, polls over all the file descriptors, processes remaining work, and passes all the file descriptors back to the original signal worker thread.

On a server handling perhaps thousands of connections, this creates considerable overhead during a period when the server is already overloaded. We modified the queue overflow handler to reduce this overhead. The server now handles signal queue overflow in the same thread as the RT signal handler; sibling threads are no longer needed. This modification appears in `phhttpd` version 0.1.2. It is still necessary, however, to build a fresh `poll_fd` array completely during overflow processing. This overhead slows the server during overflow processing, but can be reduced by maintaining the `poll_fd` array concurrently with signal processing.

RT signal queue overflow is probably not as rare as some would like. Some kernel designs have a single maximum queue size for the entire system. If any aberrant application stops picking up its RT signals (the thread that picks up RT signals may cause a segmentation fault, for example, while the rest of the application continues to run), the system-wide signal queue will fill. All other applications on the system that use RT signals will eventually be unable to proceed without recovering from a queue overflow, even though they are not the cause of it.

It is well known that Linux is not a real-time operating system, and that unbounded latencies sometimes occur. Application design may also prohibit a latency upper bound guarantee. These latencies can delay RT signals, causing the queue to grow long enough that recovery is required even when servers are fast enough to handle heavy loads under normal circumstances.

3. New interface: `sigtimedwait4()`

To reduce system call overhead and remove a potential source of unnecessary system calls, we'd like the kernel to deliver more than one signal per system call. One mechanism to do this is implemented in the `poll()` system call. The application provides a buffer for a vector of results. The system call returns the number of results it stored in the buffer, or an error.

Our new system call interface combines the multiple result delivery of `poll()` with the efficiency of POSIX RT signals. The interface prototype appears in FIG. 3.

```
int sigtimedwait4(const sigset_t *set,
       siginfo_t *infos, int nsiginfos,
       const struct timespec *timeout);
```

Figure 3. `sigtimedwait4()` prototype

Like its cousin `sigtimedwait()`, `sigtimedwait4()` provides the kernel with a set of signals in which it is interested, and a timeout value that is used when no signals are immediately ready for delivery. The kernel selects queued pending signals from the signal set specified by `set`, and returns them in the array of `siginfo_t` structures specified by `infos` and `nsiginfos`.

Providing a buffer with enough room for only one `siginfo_t` struct forces `sigtimedwait4()` to behave almost like `sigtimedwait()`. The only difference is that specifying a negative timeout value causes `sigtimedwait4()` to behave like `sigwaitinfo()`. The same negative timeout instead causes an error return from `sigtimedwait()`.

Retrieving more than one single signal at a time has important benefits. First and most obviously, it reduces the average number of transitions between user space and kernel space required to process a single server request. Second, it reduces the number of times per signal the per-task signal spinlock is acquired and released. This improves concurrency and reduces cache ping-ponging on SMP hardware. Third, it amortizes the cost of verifying the user's result buffer, although some believe this is insignificant. Finally, it allows a single pass through the signal queue for all pending signals that can be returned, instead of a pass for each pending signal.

The `sigtimedwait4()` system call enables efficient server implementations by allowing the server to "compress" signals- if it sees multiple read signals on a socket, for instance, it can empty that socket's read buffer just once.

The `sys_rt_sigtimedwait()` function is a moderate CPU consumer in our benchmarks, according to the results of kernel EIP histograms. About three fifths of the time spent in the function occurs in the second critical section in FIG. 4.

```
spin_lock_irq(&current->sigmask_lock);
sig = dequeue_signal(&these, &info);
if (!sig) {
  sigset_t oldblocked = current-
>blocked;
  sigandsets(&current->blocked,
              &current->blocked,
&these);
  recalc_sigpending(current);
  spin_unlock_irq(&current->
                         sigmask_lock);

  timeout = MAX_SCHEDULE_TIMEOUT;
  if (uts)
    timeout = (timespec_to_jiffies(&ts)
          + (ts.tv_sec || ts.tv_nsec));

  current->state = TASK_INTERRUPTIBLE;
  timeout = schedule_timeout(timeout);

  spin_lock_irq(&current->sigmask_lock);
  sig = dequeue_signal(&these, &info);
  current->blocked = oldblocked;
  recalc_sigpending(current);
}
spin_unlock_irq(&current->sigmask_lock);
```

Figure 4. This excerpt of the `sys_rt_sigtimedwait()` kernel function shows two critical sections. The most CPU time is consumed in the second critical section.

The `dequeue_signal()` function contains some complexity that we can amortize across the total number of dequeued signals. This function walks through the list of queued signals looking for the signal described in `info`. If we have a list of signals to dequeue, we can walk the signal queue once picking up all the signals we want.

4. Benchmark description

In this section, we measure and report several aspects of server performance.

Our test harness consists of two machines running Linux connected via a 100 Mb/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 has only a single CPU.

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 naturally 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 1 Kbyte document, a typical `index.html` file from the `monkey.org` web site.

4.1 Offered load

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. They also induce a bursty and unpredictable interrupt load on the server [7].

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 added an extra client that runs in conjunction with the `httperf` benchmark to simulate these slower connections to examine the effects of our improvements on more realistic server workloads [6]. This client program opens a connection, but does 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.

In previous work, we noticed server performance change as the number of inactive connections varied [9]. As a result of this work, one of the authors modified `phhttpd` to correct this problem. The latest version of `phhttpd` (0.1.2 as of this writing) does not show significant performance degradation as the number of inactive connections increases. Therefore, the benchmark results we present here show performance with no extra inactive connections.

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, because we use only a single client and server in our test harness, we can have only about 60,000 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. In the following tests, we run `httperf` with 4096 file descriptors, and `phhttpd` with five thousand file descriptors.

4.2 Execution Profiling

To assess our modifications to the kernel, we use the EIP sampler built in to the Linux kernel. This sampler checks the value of the instruction pointer (EIP register) at fixed intervals, and populates a hash table with the number of samples it finds at particular addresses. Each bucket in the hash table reports the results of a four-byte range of instruction addresses.

A user-level program later extracts the hash data and creates a histogram of CPU time matched against the kernel's symbol table. The resulting histogram demonstrates which routines are most heavily used, and how efficiently they are implemented. The granularity of the histogram allows us to see not only which functions are heavily used, but also where the most time is spent in each function.

5. Results and Discussion

In this section we present the results of our benchmarks, and describe some new features that our new system call API enables.

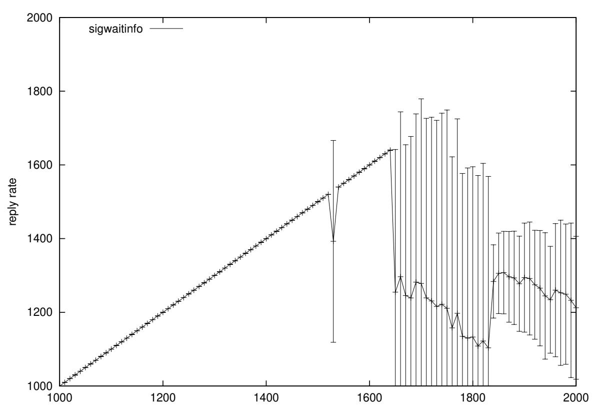
5.1 Basic performance and scalability results

As described previously, our web server is a single processor host running a Linux 2.2.16 kernel modified to include our implementation of `sigtimedwait4()`. The web server application is phhttpd version 0.1.2. We compare an unmodified version with a version modified to use `sigtimedwait4()`. Our benchmark driver is a modified version of `httperf 0.6` running on a four-processor host.

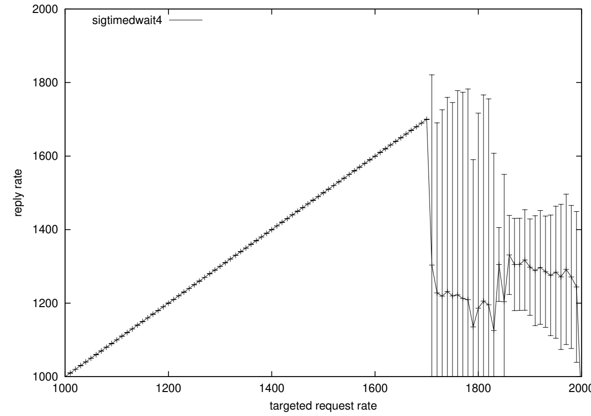
Our first test compares the scalability of unmodified `phhttpd` using `sigwaitinfo()` to collect one signal at a time with the scalability of `phhttpd` using `sigtimedwait4()` to collect many signals at once. The modified version of `phhttpd` picks up as many as 500 signals at once during this test.

Graphs 1 and 2 show that picking up more than one RT signal at a time gains little. Only minor changes in behavior occur when varying the maximum number of signals that can be picked up at once. The maximum throughput attained during the test increases slightly.

This result suggests that the largest system call bottleneck is not where we first assumed. Picking up signals appears to be an insignificant part of server overhead. We hypothesize that *responding to requests*, rather than picking them up, is where the server spends most of its effort.



Graph 1. Scalability of the `phhttpd` web server. This graph shows how a single threaded `phhttpd` web server scales as request rate increases. The axes are in units of requests per second.



Graph 2. Scalability of `phhttpd` using `sigtimedwait4()`. The signal buffer size was five hundred signals, meaning that the web server could pick up as many as five hundred events at a time. Compared to Graph 1, there is little improvement.

5.2 Improving overload performance

While the graphs for `sigtimedwait4()` and `sigwaitinfo()` look disappointingly similar, `sigtimedwait4()` provides new information that we can leverage to improve server scalability.

Mogul, *et al.*, refer to "receive livelock," a condition where a server is not deadlocked, but makes no forward progress on any of its scheduled tasks [12]. This is a condition that is typical of overloaded interrupt-driven servers: the server appears to be running flat out, but is not responding to client requests. In general, receive livelock occurs because processing a request to completion takes longer than the time between requests.

Mogul's study finds that dropping requests as early as possible results in more request completions on overloaded servers. While the study recommends dropping requests in the hardware interrupt level or network protocol stack, we instead implement this scheme at the application level. When the web server becomes overloaded, it resets incoming connections instead of processing the requests.

To determine that a server is overloaded, we use a weighted load average, essentially the same as the TCP round trip time estimator [11, 13, 14]. Our new `sigtimedwait4()` system call returns as many signals as can fit in the provided buffer. The number of signals returned each time `phhttpd` invokes `sigtimedwait4()` is averaged over time. When the load average exceeds a predetermined value, the server begins rejecting requests.

Instead of dropping requests at the application level, using the listen backlog might allow the kernel to drop connections even before the application becomes involved in handling a request. Once the backlog overflows, the server's kernel can refuse connections, not even passing connection requests to the server application, further reducing the workload the web server experiences. However, this solution does not handle bursty request traffic gracefully. A moving average such as the RTT estimator smoothes out temporary traffic excesses, providing a better indicator of server workload over time.

The smoothing function is computed after each invocation of `sigtimedwait4()`. The number of signals picked up by `sigtimedwait4()` is one of the function's parameters:

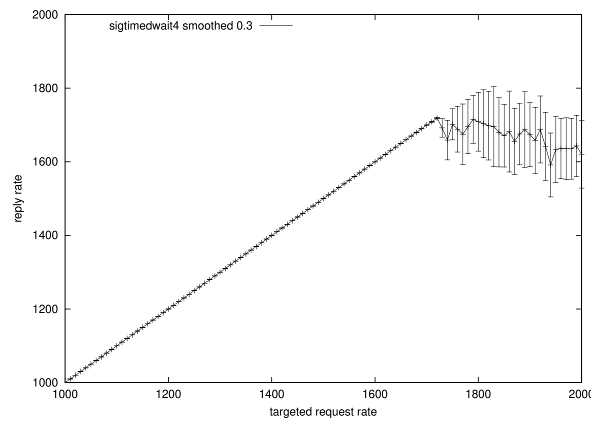
$$Avg_t = \alpha S + (1-\alpha) Avg_{t-1}$$

where S is the number of signals picked up by the most recent invocation of `sigtimedwait4()`; Avg is the moving load average; α is the gain value, controlling how much the current signal count influences the load average; and t is time.

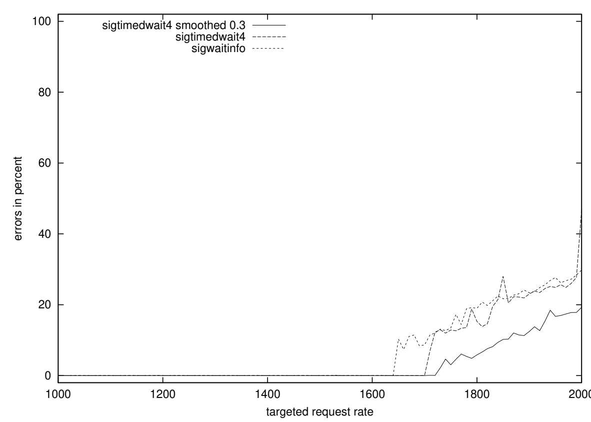
In our implementation, `phhttpd` picks up a maximum of 23 signals. If Avg exceeds 18, `phhttpd` begins resetting incoming connections. Experimentation and the following reasoning influenced the selection of these values. As the server picks up fewer signals at once, the sample rate is higher but the sample quantum is smaller. Only picking up one signal, for example, means we're either overloaded, or we're not. This doesn't give a good indication of the server's load. As we increase the signal buffer size, the sample rate goes down (it takes longer before the server calls `sigtimedwait4()` again), but the sample quantum improves. At some point, the sample rate becomes too slow to adequately detect and handle overload. That is, if we pick up five hundred signals at once, the server either handles or rejects connections for all five hundred signals.

The gain value determines how quickly the server reacts to full signal buffers (our "overload" condition). When the gain value approaches 1, the server begins resetting connections almost immediately during bursts of requests. Reducing the gain value allows the server to ride out smaller request bursts. If it is too small, the server may fail to detect overload, resulting in early performance degradation. We found that a gain value of 0.3 was the best compromise between smooth response to traffic bursts and overload reaction time.

Graphs 3 and 4 reveal an improvement in overload behavior when an overloaded server resets connections immediately instead of trying to fulfill the requests. Server performance levels off then declines slowly, rather than dropping sharply. In addition, connection error rate is considerably lower.



Graph 3. Scalability of `phhttpd` with averaged load limiting. Overload behavior improves considerably over the earlier runs, which suggests that formulating and sending responses present much greater overhead for the server than handling incoming signals.



Graph 4. Error rate of `phhttpd` with averaged load limiting. When the server drops connections on purpose, it actually reduces its error rate.

6. Conclusions and Future Work

Using `sigtimedwait4()` enables a new way to throttle web server behavior during overload. By choosing to reset connections rather than respond to incoming requests, our modified web server survives considerable overload scenarios without encountering receive livelock. The `sigtimedwait4()` system call also enables additional efficiency: by gathering signals in bulk, a server application can "compress" signals. For instance, if the server sees multiple read signals on a socket, it can empty that socket's read buffer just once.

Further, we demonstrate that more work is done during request processing than in handling and dispatching incoming signals. Lowering signal processing overhead in the Linux kernel has little effect on server performance, but reducing request processing overhead in the web server produces a significant change in server behavior.

It remains to be seen whether this request processing latency is due to:

- accepting incoming connections (`accept()` and `read()` system calls)
- writing the response (nonblocking `write()` system call and accompanying data copy operations)
- managing the cache (server-level hash table lookup and `mmap()` system call)
- some unforeseen problem.

Even though sending the response back to clients requires a copy operation, it is otherwise nonblocking. Finding the response in the server's cache should also be fast, especially considering the cache in our test contains only a single document. Thus we believe future work in this area should focus on the performance of the system calls and server logic that accept and perform the initial read on incoming connections.

This paper considers server performance with a single thread on a single processor to simplify our test environment. We should also study how RT signals behave on SMP architectures. Key factors influencing SMP performance and scalability include thread scheduling policies, the cache-friendliness of the kernel implementation of RT signals, and how well the web server balances load among its worker threads.

6.1. Acknowledgements

The authors thank Peter Honeyman and Andy Adamson for their guidance. We also thank the reviewers for their comments. Special thanks go to Zach Brown for his insights, and to Intel Corporation for equipment loans.

7. 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*, `www.zabbo.net/phhttpd`, November 1999.

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

[4] G. Banga. P. Druschel. 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 Concurrency Models on Web Server Performance Over High-Speed Networks," *Proceedings of the 2nd IEEE Global Internet Conference*, November 1997.

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

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

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

[9] N. Provos and C. Lever, "Scalable Network I/O in Linux," *Proceedings of the USENIX Technical Conference, FREENIX track,* June 2000.

[10] W. Richard Stevens, *UNIX Network Programming, Volume I: Networking APIs: Sockets and XTI*, 2nd edition, Prentice Hall, 1998.

[11] W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, pp. 299-309, Addison Wesley professional computing series, 1994.

[12] J. C. Mogul, K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *Proceedings of USENIX Technical Conference*, January 1996.

[13] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *Computer Communication Review*, pp. 2-7, vol. 17, no. 5, August 1987.

[14] V. Jacobson, "Congestion Avoidance and Control," *Computer Communication Review*, pp. 314-329, vol. 18, no. 4, August 1988.

[15] 1003.1b-1993 POSIX – Part 1: API C Language – Real-Time Extensions (ANSI/IEEE), 1993. ISBN 1-55937-375-X.

[16] GNU `info` documentation for `glibc`.

Appendix A: Man page for `sigtimedwait4()`

```
SIGTIMEDWAIT4(2)        Linux Programmer's Manual       SIGTIMEDWAIT4(2)

NAME
     sigtimedwait4 - wait for queued signals

SYNOPSIS
     #include <signal.h>

     int sigtimedwait4(const sigset_t *set, siginfo_t *infos,
           int nsiginfos, const struct timespec *timeout);

     typedef struct siginfo {
        int            si_signo; /* signal from signal.h */
        int            si_code;  /* code from above      */
      ...
        int            si_value;
      ...
     } siginfo_t;

     struct timespec {
        time_t         tv_sec;   /* seconds */
        long           tv_nsec;  /* and nanoseconds */
     };
DESCRIPTION
     sigtimedwait4() selects queued pending signals from the  set
     specified by  set,  and  returns  them  in  the  array  of
     siginfo_t structs   specified   by   infos   and  nsiginfos.
     When multiple signals are pending,  the lowest numbered ones
     are selected.  The selection order between realtime and non-
     realtime signals, or between multiple  pending  non-realtime
     signals, is unspecified.

     sigtimedwait4() suspends  itself  for  the   time   interval
     specified in the timespec  structure  referenced by timeout.
     If  timeout  is  zero-valued,   or  no  timespec  struct  is
     specified, and if none of  the  signals  specified by set is
     pending, then sigtimedwait4()  returns  immediately with the
     error EAGAIN.   If  timeout   contains  a negative value, an
     infinite timeout is specified.

     If no signal in set is pending  at  the time  of  the  call,
     sigtimedwait4() suspends  the  calling  process until one or
     more signals  in  set  become  pending,  until  it is inter-
     rupted by an unblocked, caught signal,  or until the timeout
     specified   by   the   timespec   structure  pointed  to  by
     timeout expires.

     If, while sigtimedwait4() is waiting, a signal  occurs which
     is eligible for delivery (i.e., not blocked by  the  process
     signal mask), that  signal  is  handled  asynchronously  and
     the wait is interrupted.

     If  infos  is  non-NULL,  sigtimedwait4()  returns  as  many
     queued signals  as  are  ready  and  will  fit  in the array
     specified by infos. In  each  siginfo_t struct, the selected
     signal number is stored in si_signo,  and the cause  of  the
```

signal is stored in the si_code. If a payload is queued with the signal, the payload value is stored in si_value.

If the value of si_code is SI_NOINFO, only the si_signo member of a siginfo_t struct is meaningful, and the value of all other members of that siginfo_t struct is unspecified.

If no further signals are queued for the selected signal, the pending indication for that signal is reset.

RETURN VALUES

sigtimedwait4() returns the count of siginfo_t structs it was able to store in the buffer specified by _infos_ and _nsiginfos_. Otherwise, the function returns -1 and sets errno to indicate any error condition.

ERRORS

EINTR — The wait was interrupted by an unblocked, caught signal.

ENOSYS — sigtimedwait4() is not supported by this implementation.

EAGAIN — No signal specified by _set_ was delivered within the specified timeout period.

EINVAL — _timeout_ specified a tv_nsec value less than 0 or greater than 1,000,000,000.

EFAULT — The array of siginfo_t structs specified by _infos_ and _nsiginfos_ was not contained in the calling program's address space.

CONFORMING TO

Linux

AVAILABILITY

The sigtimedwait4() system call was introduced in Linux 2.4.

SEE ALSO

time(2), sigqueue(2), sigtimedwait(2), sigwaitinfo(2)

Linux 2.4.0 Last change: 23 August 2000 1