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

Linux Scalability Project

Center for Information Technology Integration

University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu

Overview

1. Introduction
2. POSIX RT Signals
3. `phhttpd`
4. `sigtimedwait4()`
5. Benchmark
6. Discussion
7. Conclusion

Introduction

- Experts argue that a combination of I/O completion events are more scalable than today's servers.
- In Linux, I/O completion events are delivered via POSIX RT signals; the signals can also be queued and delivered to an application one at a time.
- When the signal queue is exhausted, a server needs to switch to polling.

Introduction

- Even without signal queue overflows, a long signal queue can cause system call overhead to dominate server processing.
- In Linux, system calls are designed to be inexpensive.
- Lowering the number of system calls required for queue management can improve the scalability of network servers.
- We created a new interface `sigtimedwait4()` that delivers multiple signals to an application simultaneously.

POSIX RT Signals

- We illustrate how signal driven I/O works in general.
- An application enables signal-driven I/O as follows:
  - Assign a SIGIO handler with `sigaction()`.
  - Assign an owner pid to the socket with `fcntl()`.
  - Respond to signals with the signal handler or by masking them and picking them up with `sigwaitinfo()`.
- The kernel raises `SIGIO` for various reasons:
  - A connection request has completed,
  - data has arrived on a socket,
  - a write operation has completed, etc...

POSIX RT Signals

- In POSIX, more detailed signal information is available in the `siginfo_t` struct.
- However, there is no information about the file descriptor that caused the signal.
- POSIX RT signals can associate a signal with specific file descriptors using `fcntl(fd, F_SETSIG, signum)`.

POSIX RT Signals

- POSIX Real-Time signals can be queued in the kernel and delivered one at a time using `sigwaitinfo()` or `sigtimedwait()`.
- Additionally, the `siginfo_t` struct returned by `sigwaitinfo()` contains information about the file descriptor and the reason for the signal.
- Under POSIX semantics, a threaded application should handle all signals in one thread and have all others block them.
- However, Linux threads have their own PIDs, and signals can be directed to particular threads.

phhttpd

- **phhttpd** is a static-content caching web server. It uses Apache as its back end.
- **phhttpd** uses Linux's POSIX RT signal mechanism.
- It uses one RT signal number for all file descriptors for the following reasons:
  - Lowest RT signals are delivered first. Using the same signal number ensures in-order delivery.
  - There is no standard way to allocate signal numbers atomically across threads.
- **phhttpd** uses RT signal worker threads that mask off the RT signals and pick them up with `sigwaitinfo()`.

phhttpd

- To reduce system call rate, `phhttpd` reads on a new connection directly after accepting it, and sends an immediate response.
- Load balancing is implemented by passing the listen socket to the next worker thread after accepting a new connection.
- This is done by reassigning the owner with `fcntl(fd, F_SETOWN, newpid)`.

sigtimedwait4()

- To reduce system call overhead and remove unnecessary system calls, we want the kernel to deliver more than one signal at once.
- Our new system call `sigtimedwait4()` has the following arguments:

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

- It returns the number of signals retrieved or an error.

sigtimedwait4()

- Retrieving more than one signal at a time has the following benefits:
  - It reduces the number of transitions between user and kernel space.
  - It reduces the number of times the per-task signal spinlock is acquired.
  - It allows for a single pass through the signal queue for all pending signals.

Benchmark

- Our test harness consists of two Linux machines connected via 100 MBit/s switched Ethernet.
- The workload is driven by an Intel SC450NX with four 500 MHz Xeon Pentium III processors with 512 MByte memory.
- The web server runs on a 400 MHz AMD K6-2 processor with 64 MByte memory and a single 8 GByte 7.2 KRPM IDE drive.

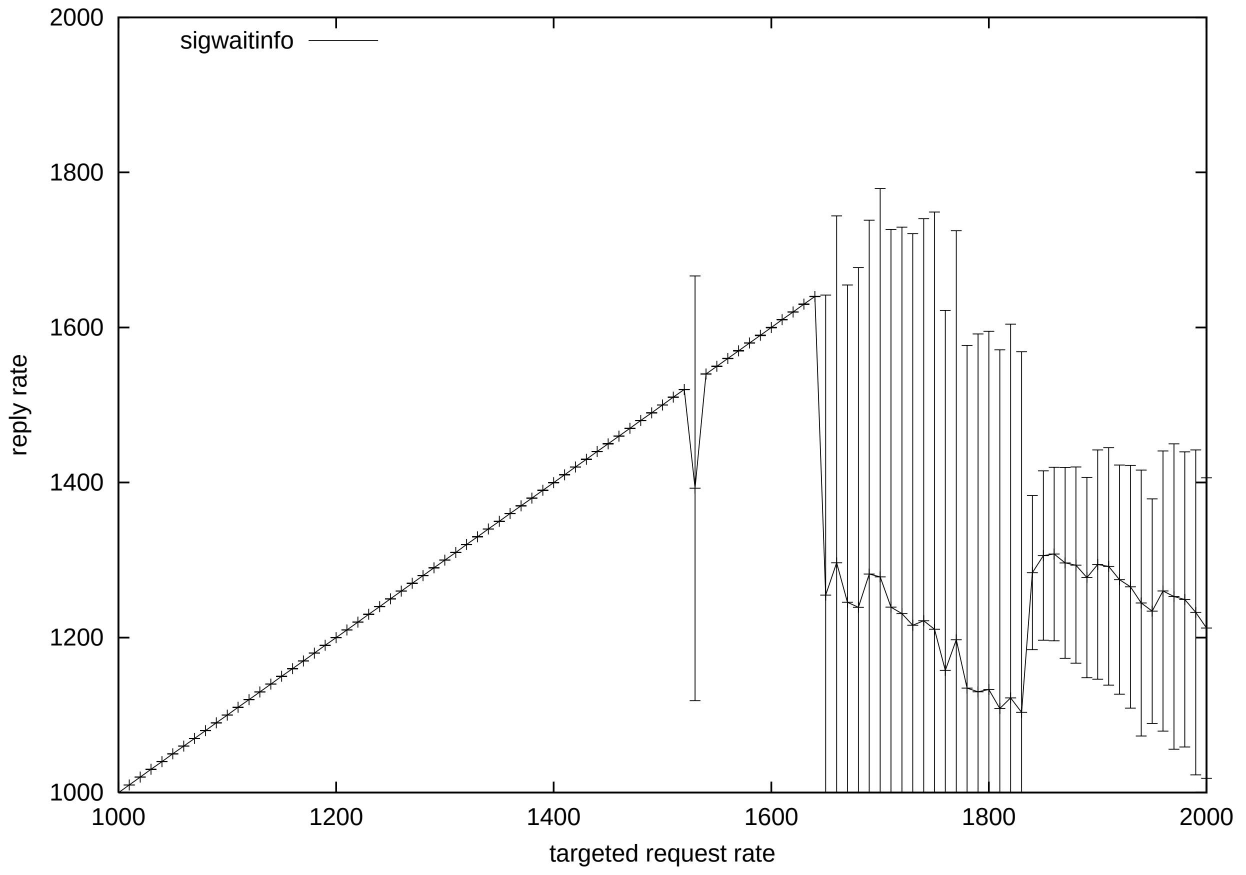
Benchmark

- The server hardware is small, so that it is easy to drive the server into overload.
- The benchmark is driven by `httperf` providing repeatable server workloads.
- We use only static content as it exercises the system components that interested us.

Benchmark

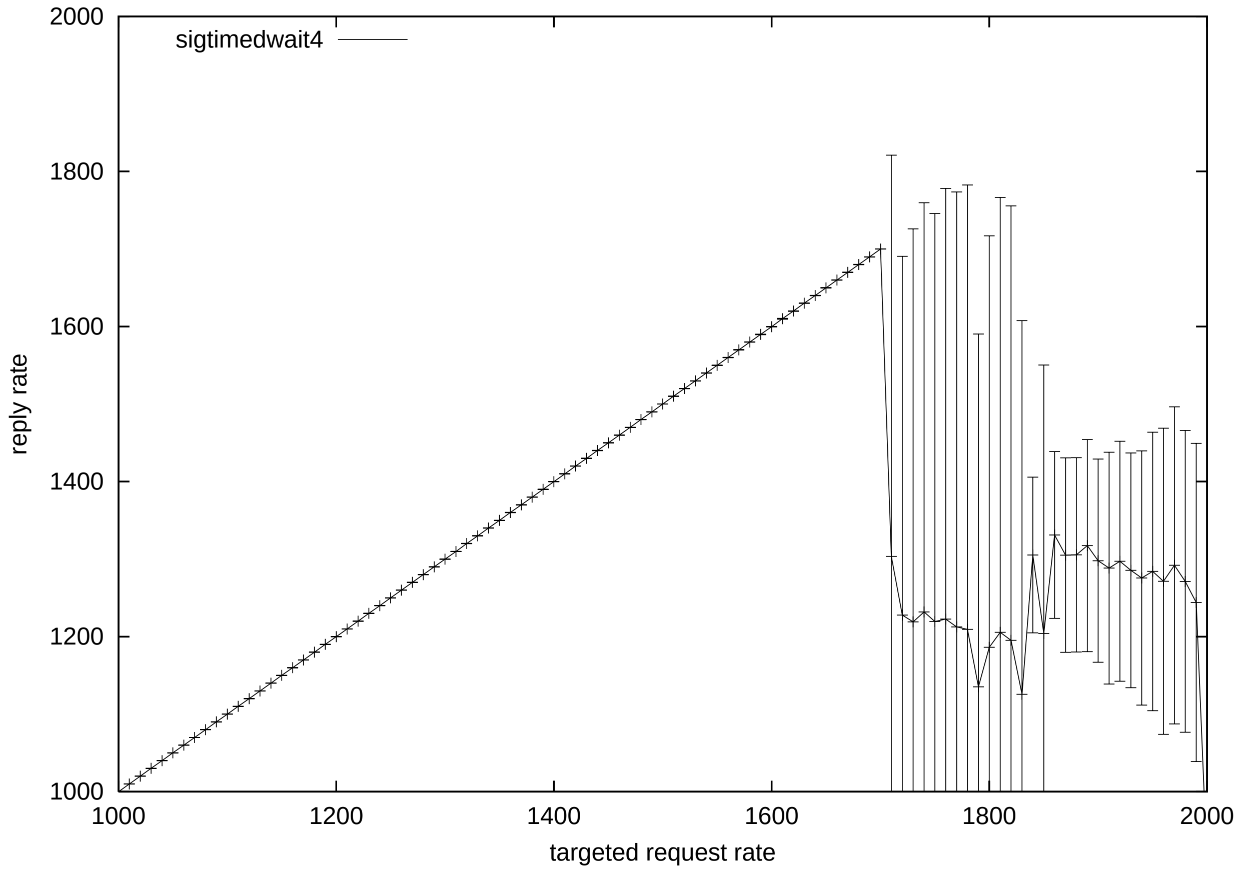
- There are several limitations influencing the benchmark:
  - We had to modify `httperf` to deal with more than 1024 file descriptors.
  - We can have only 60K open sockets at any time, so each benchmark is limited to 35K connections.
  - Before the next benchmark starts, we wait for all sockets to leave `TIMEWAIT`.
- `httperf` runs with 4096 file descriptors and `phhttpd` with five thousand.

sigwaitinfo() with phhttpd web



- At higher request rates, server performance breaks down.

sigtimedwait4() retrieving up to 500 signals



- Compared to `sigwaitinfo()` only small improvement.

Discussion

- Only minor improvements occur when varying the number of signals that `sigtimedwait4()` can pick up.
- `sigtimedwait4()` with 500 signals has a slightly higher maximum throughput compared to `sigwaitinfo()`.
- The results indicate that picking up the signals does not constitute the significant part of the server overhead.

Discussion

- However, `sigtimedwait4()` provides new information that we can use to improve scalability.
- Mogul, *et al.*, call "receive livelock" the situation where the server make no progress because it accepts new requests before it completes older ones.
- In such an overload situation, dropping requests early results in more request completions.

Discussion

- To determine that a server is overloaded, we use a weighted load average of the number of signals returned.
- When the load average exceeds a predetermined value, the server rejects new requests.
- The moving average allows us to smooth out temporary bursts.

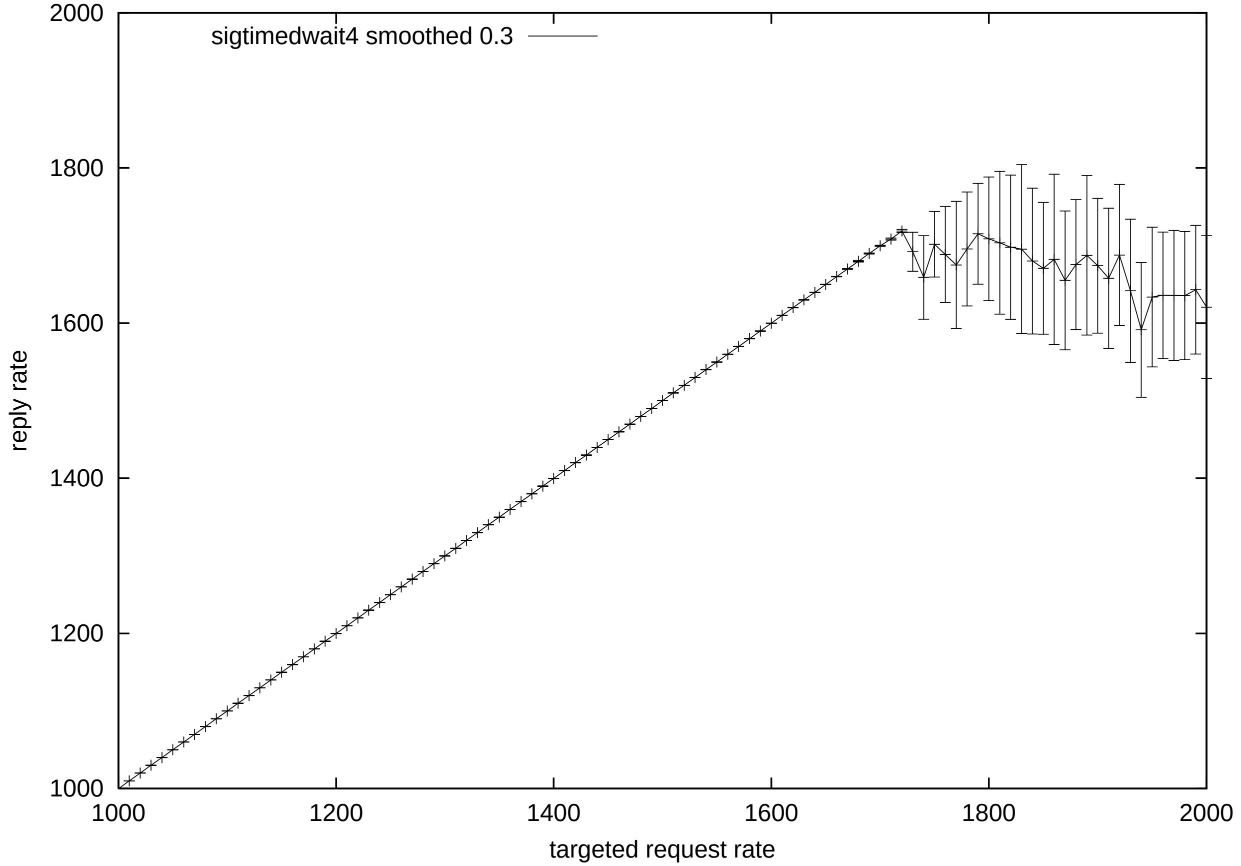
Discussion

- The number of signals S returned by `sigtimedwait4()` is averaged over time as follows:

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

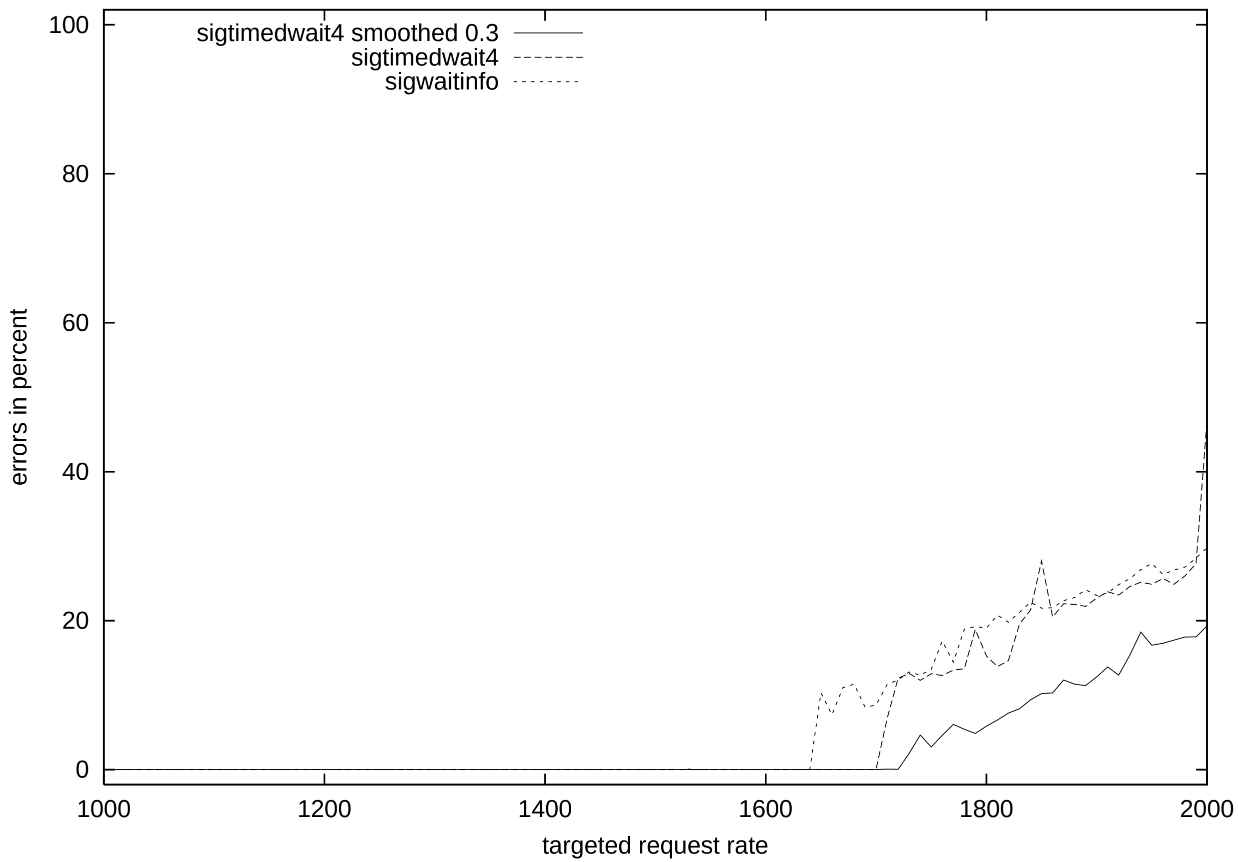
- In our implementation, `phhttpd` picks up a maximum of 23 signals.
- The threshold for the average at which `phhttpd` starts resetting connections is set to 18.
- We found that $\alpha = 0.3$ caused a good reaction time to overload situations while still smoothing out bursts.

sigtimedwait4() with averaged load limiting



- Overload behavior improves considerably.

Error rate in comparison



- Error rate is reduced when server drops requests on purpose.

Conclusion

- Lowering the signal processing overhead in the Linux kernel has little effect on server performance.
- `sigtimedwait4()` enables a new way to deal with web server overload.
- By choosing to reset connections, our modified web server achieves considerably better performance during overload.
- Software enhancements are freely available, contact the authors.
- Acknowledgments:
  - Peter Honeyman and Andy Adamson for their guidance,
  - Zach Brown for his insights,
  - Intel Corporation for equipment loans.

