Are events fast?

Gabriel Kerneis, Juliusz Chroboczek

Are events fast?

Gabriel Kerneis and Juliusz Chroboczek
PPS, Université de Paris 7, Paris, France
kerneis@pps.jussieu.fr

12 January 2009

Abstract

We compare a set of web servers on a simple synthetic workload. We show that, on this particular benchmark, event-driven code is as fast or faster than the fastest implementations using thread libraries.

1 Introduction

Increasingly, the programs we write are *concurrent*: they simultaneously communicate with their environment over multiple channels, and need to react to input events as they arrive in a nondeterministic order. The most obvious examples of such programs are network daemons, that need to simultaneously service multiple clients or peers, programs with a graphical user interface (GUI), that need to simultaneously service multiple input devices, and networked GUI programs such as web browsers, that need to do both.

There are two main paradigms for writing such programs: *threads* and *event loops*.

Threads When using threads, a concurrent program is described as a set of sequential threads of control, communicating through shared memory. Threads may be created and destroyed dynamically, or there may be a *thread pool*, a statically sized set of threads that compete for the work to be performed.

Threads can be implemented in an operating system kernel, in which case we speak of *kernel threads*; they can also be implemented by a user-space library, in which case we speak of *user threads*. Kernel and user threads have different performance tradeoffs: kernel threads require a relatively expensive kernel intervention at context-switch time, but they allow the use of multiple CPUs or CPU cores. Additionally, kernel threads allow the direct use of native system services, while with user threads, precautions must be taken to avoid blocking the whole program when invoking a blocking system call.

Event-driven programs In an *event-driven* style, the concurrent program is described as a set of *event handlers* that handle the situations that can occur, such as receiving a request for a new connection, receiving a client request, or a client timeout.

The set of events, and the set of associated event handlers, can change over time; for example, after a new connection has been established, a new event handler is established for handling requests arriving on that particular connection.

Performance issues Event-driven programming is rumoured [8] to have a number of advantages over threads. First, event-handlers are tiny, heap-allocated data structures, unlike thread contexts, which include a large, statically allocated stack. A highly concurrent event-driven program will therefore likely use less memory than a threaded program. Second, switching between event handlers involves a mere (indirect) function call; a thread switch, on the other hand, may involve one or more system calls with the associated mode-switching penalty. Finally, since event handlers are scheduled within user code, an event-driven program is likely to provide more deterministic response than a threaded one.

The performance advantage of event-driven programs, however, is not as obvious as might appear

from the above description. A thread's state is split between a number of locations: the processor's registers (including the program counter), stack-allocated variables and heap-allocated structures. An event handler, on the other hand, receives much of its state in the form of a single heap-allocated data structure; this structure must be unpacked and its contents "spread" into the processor's registers before the event handler can be run, which involves at least one indirect load per register. Thus, event-driven programming avoids the localised cost associated to thread contexts, but pays a price that is spread throughout the program and hence difficult to quantify.

Automatic generation of threaded programs In a threaded program, the flow of control of every single thread is explicit. In an event-driven program, on the other hand, the flow of control is broken into a possibly large number of possibly tiny event handlers. Because of that, event-driven programming is difficult and error-prone, and most programmers choose to avoid it.

The authors are currently experimenting with automatic generation of event-driven programs from a threaded description [4]. Our software, called *CPC*, takes a threaded description of a program, and produces an event-driven program that is equivalent, in a suitable sense, to the original description. Since the main import of this work is to combine the rumoured efficiency of event-driven programs with the convenience and intuitiveness of thread-based programming, it is of the utmost importance for us to understand the effective efficiency of event-driven programs.

Experimental approach The aim of this work is to precisely quantify the relative performance of kernel threads, user-space threads, hand-written event-driven programs, and automatically generated event-driven programs.

Our goal being to compare implementations of concurrency, rather than to provide realistic benchmarks of concurrent programs, we have chosen to use a simple, well-understood and repeatable benchmark rather than a realistic one. We have benchmarked a number of HTTP servers written in different programming styles, serving a single short (305 bytes) page to varying numbers of simultaneous clients. The servers measured include on the one hand a number of high-quality production servers, and on the other hand a number of "toy" servers written for this exercice.

2 Experimental approach

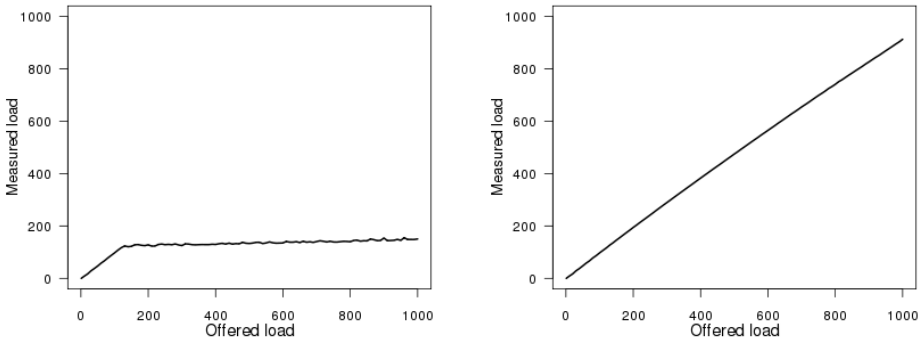
As noted above, we have used a simple, easily reproducible setup which would show how well a number of web server implementations handle multiple concurrent requests.

We have used the *ApacheBench* client which is designed to generate a constant load on a web server. Given a parameter l known as the *concurrency level* or simply *load*, ApacheBench tries to generate a self-clocking stream of requests timed so that the number of open connections at any given time is exactly l.

In practice, however, we have found that ApacheBench needs a period of time to "ramp up"; for this reason, we have discarded the first and last 1,000 samples from our measurements.

Tuning We were very much surprised by our first batch of experimental data. Below 128 simultaneous clients, the median and the average latency coincided and were linear in the number of simultaneous clients, just as expected. Above that threshold, however, the median latency remained at a constant value of 50 ms, while the average value became irregular and irreproducible. These results were caused by a small number of extreme outliers — requests that were being served in 200 ms or more.

In order to understand the phenomenon, we came up with the notion of *measured load* of a benchmark run. Consider an ideal run, in which there is no idle time: the number of in-flight requests at any given time is exactly l. Writing t for the total run time of the test, n the total number of requests serviced, and t_r the average servicing time for a request, we would then have $t = n \cdot t_r / l$. We therefore define the *measured load* l_m as $l_m = n \cdot t_r / t$; this value is always lower than the desired load l, and we would hope that



(a) `somaxconn` = 128 (b) `somaxconn` = 1024

Figure 1: Measured load against offered load, before and after tuning

it is very close to l in a run in which the client is able to saturate the server.

Plotting the measured load against the offered load (Fig. 1(a)) showed us that, however large the offered load, the effective load never exceeded roughly 130; obviously, something in our setup was limiting the number of connections to 130.

It turned out that the bottleneck was the kernel variable `somaxconn`, the value of which defaults to 128 [3]. The `listen` system call, which is used to establish a passive ("listening") socket, takes a parameter that indicates the length of the associated "accept queue"; when this queue becomes full, e.g. because the server doesn't accept connections fast enough, new requests are discarded, and will be resent by the client after a timeout. The variable `somaxconn` limits the size of this queue: the parameter to `listen` is silently limited to the value of `somaxconn`. Raising `somaxconn` to 1024 solves the problem (Fig 1(b)).

Other potential bottlenecks In order to ensure that our results apply more generally and are not specific to our particular setup, we repeated our benchmarks while varying other parameters, and found that they had no significant impact on the results. In particular, using different network cards and removing the switch between the client and the server yielded no measurable difference — hence, no hardware queues were being overflown. Using different computers (a faster client, a slower server) yielded slightly different figures, but didn't change the general conclusions. Finally, preloading the served file into memory only caused a slight additive difference. Tests with differently sized files (up to 100 kB) confirmed the general thrust of our results.

3 Implementations

We benchmarked four production web servers, and a set of "toy" web servers that were written for this particular purpose.

Full-fledged web servers Apache [2] is the most widely deployed web server in the Internet today; hence, a benchmark of web servers must include it as a comparison point. One of the claimed advantages of Apache 2 is its ability to run with different concurrency models; we measured two of Apache's concurrency models, the process-pool model *("prefork")* and the thread-pool model *("worker")*.

Thttpd [1] is a small event-driven server which was considered as one of the fastest open-source web servers in the late 1990s. It uses a standard event-driven model, with one minor twist: connections are accepted eagerly, and kept in a user-space queue of accepted connections until they are serviced.

Polipo [5] is an HTTP proxy written by the second author that can also act as a web server. It uses a fairly standard event-driven model.

Lighttpd [7] is a recent, highly optimised event-driven web server.

Toy servers We have written a set of toy web servers (less than 200 lines each) that share the exact same structure: a single thread or process waits for incoming connections, and spawns a new thread or process as soon as one is accepted; our servers do not use any clever implementation techniques, such as thread pools. Because of this simple structure, these servers can be directly compared, and we are able to benchmark the underlying implementation of concurrency rather than the implementation of the web server.

One of these web servers uses heavy-weight Unix processes, created using the `fork` system call. One is written using *NPTL*, the native thread library used in Linux version 2.6. Two are written using standard user-space thread libraries, called respectively *Pth* [6] and *ST* [9].

Finally, one uses *CPC* [4], our experimental source-to-source translator that converts programs written in a threaded style into event-driven programs. While CPC is at a very early stage, and doesn't yet contain many of the optimisations that are possible, we believe that the code that it generates is representative of naïve event-driven programming.

4 Experimental results

Fig. 2 presents the results of our experiment. It plots the average serving time per request against the desired load; a smaller slope indicates a faster server. With the exception of Apache, the curves are extremely regular (in each case, the correlation coefficient between the mean reponse time and the offered load is above 0.999).

Discussion Apache artificially limits the size of the accept queue to 512; hence, its results for more than 512 simultaneous requests should not be taken into account. Apache turned out to be the slowest amongst the production servers that we considered; moreover, we found that the process-pool *(prefork)* and the thread-pool *(worker)* implementations performed equally poorly.

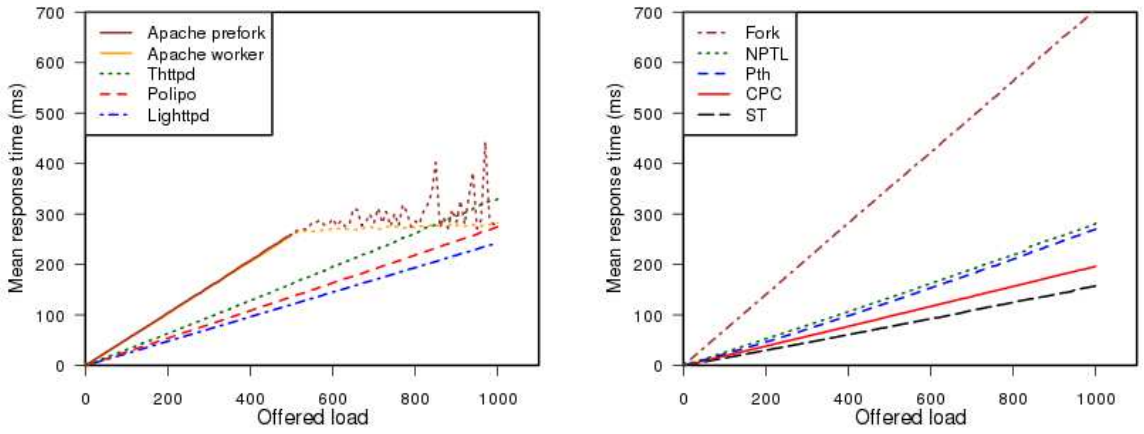
All three event-driven production servers were significantly faster than Apache, and their performance was roughly similar. Thttpd was somewhat slower than Polipo, and Lighttpd very slightly faster; we believe that the difference is due to different micro-optimisations rather than to any fundamental difference between the three servers. Incidentally, results when the accept queue did overflow (not shown) were much more regular for thttpd than for the other servers, which shows the effect of a user-space accept queue.

The production servers were generally slower than the toy servers, as the former need to perform additional activities such as security checks, monitoring user files for changes, etc.

The implementation using full-fledged processes, created using the `fork` system call, was slower than any other of the implementations that we benchmarked, while the version implemented using NPTL, the library based on the native (kernel) threads of the Linux operating system, turned out to be surprisingly fast. The good showing of NPTL indicates that, even on a modern virtual-memory system, fork's overhead is due to manipulating virtual memory structures rather than to kernel-side scheduling; in fact, NPTL's performance is close to that of the poorer user-space libraries.

The user-space threading libraries, Pth and ST, behaved quite differently. Pth yielded results similar to those of NPTL, while ST yielded excellent results; a cursory examination of ST's sources indicates that it uses some rather clever data structures (e.g. heaps for storing timeouts) and a number of micro-optimisations, some of which could easily be reused in other user-space implementations of threads.

Finally, the version implemented using *CPC*, our source-to-source transformation framework that converts threaded programs to event-driven ones, gave results that were better than all of the other implementations save the one using ST. Since CPC's scheduler is not as clever as the one in ST, and that many of the optimisations used by the latter can be used in the former, we believe that this is an encouraging result.



(a) Full-fledged web servers

(b) Toy web servers

Figure 2: Web servers comparison

5 Conclusion

Our results indicate that, for one particular class of realistic programs, event-driven programs are as fast as the fastest user-space thread libraries. Since events have much smaller memory requirements than threads, this indicates that they are an interesting technique for a certain class of environments. This encourages us to continue our research about automatic generation of event-driven programs from threaded descriptions.

References

[1] *The thttpd man page*, v. 2.25b. December 2003.

[2] *Apache HTTP Server Version 2.2 Documentation*, v. 2.2.9. June 2006.

[3] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *Proc. USITS'97*, pp. 61–71, Berkeley, CA, USA, 1997. USENIX Association.

[4] Juliusz Chroboczek. *The CPC manual*, preliminary edition. June 2008. Available online at `http://www.pps.jussieu.fr/~jch/software/cpc/cpc-manual.pdf`.

[5] Juliusz Chroboczek. *The Polipo manual*, v. 1.0.4. January 2008.

[6] Ralf S. Engelschall. *The GNU Portable Threads manual*, v. 2.0.7. June 2006.

[7] *The Lighttpd manual*, v. 1.4.19. March 2008.

[8] John Ousterhout. Why threads are a bad idea (for most purposes). January 1996.

[9] Gene Shekhtman and Mike Abbott. *The State Threads Library Documentation*, v. 1.7. June 2006.

A Experimental setup

The server is a Pentium-M laptop, downclocked to 600 MHz to ensure that it is slower than the client; CPU usage was close to 100 % during all of the tests. The client is a standard workstation using an AMD Athlon 64 at 2.2 GHz, with power-saving features turned off; in none of our tests did its CPU usage rise above 20 %. Both machines have Gigabit Ethernet interfaces, and were connected through a dedicated Gigabit Ethernet switch. We used the standard Ethernet MTU of 1500 bytes.

The server and the client were both running Linux kernel 2.6.24. We used Apache 2.2, Thttpd 2.25b, Polipo 1.0.4 and Lighttpd 1.4.19, and the version of ApacheBench included with Apache 2.2; the libraries used were ST 1.7 and Pth 2.0.7.