Scalable Network Programming
Or: The Quest For A Good Web Server (That Survives Slashdot)

Felix von Leitner
felix-linuxkongress@fefe.de

2003-10-16

Abstract
How do you write a fast network server?
How do you write a network server that can handle 10000 clients?
What bottlenecks are there and how do you avoid them?

About me

I do security consulting for a living (my company is called Code Blau).

I am particularly interested in small and high performance code.

Besides writing web servers, I also wrote an ftp server, a very fast LDAP server (read-only so far, I'm waiting for someone to pay me to complete it), and the smallest Unix libc (which I also used to compile and run all the programs mentioned in this talk).

I'm also known in some circles for porting djbdns to IPv6, writing the FAQ for the mutt mail client, and for writing inflammatory bug reports on the Linux kernel mailing list.

Why care about high performance network code?

Apache ist fast enough for most things.

But if you read Slashdot a lot, you will see that the sites they link to are often down, because they can't handle the load.

Slashdot itself uses 8 P3/600 with 1 Gig RAM and 10k rpm SCSI disks.

www.heise.de uses 4 P3/650 with 1 Gig RAM.

ftp.fu-berlin.de is an SGI Origin 200 with 2 R10k/225 with 1 Gig RAM and 10k rpm SCSI disks.

It's obviously possible to buy performance with more hardware, but how can we make sure the software is not the bottleneck?

Why is it important to handle many connections?

I was called by an internet toy shop once. It was a rainy day in December.

They are making most of their money in the Christmas season.

They called because their web server was under attack. Some sort of distributed attack had opened 100.000 HTTP connections to their web server.

This caused 20.000 Apache processes to be spawned on each of their load balanced Solaris back-ends. The machines were desparation swapping.

It was impossible to buy anything from them.

I think they were actually afraid of bankruptcy.

First, let’s write a web client

```
char buf[4096];
int len;
int fd=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);
struct sockaddr_in si;

si.sin_family=PF_INET;
inet_aton("127.0.0.1",&si.sin_addr);
si.sin_port=htons(80);
connect(fd,(struct sockaddr*)si,sizeof si);
write(fd,"GET / HTTP/1.0\r\n\r\n");
len=read(fd,buf,sizeof buf);
close(fd);
```

That’s it?

Well,

1. I didn’t `#include` all the headers
2. there is no error handling
3. the client can only fetch up to 4k (including HTTP headers)
4. you can’t actually specify the URL

but other than that: yeah, that’s pretty much it.

OK, then let’s write a web server!

```
int cfd,fd=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);
struct sockaddr_in si;

si.sin_family=PF_INET;
inet_aton("127.0.0.1",&si.sin_addr);
si.sin_port=htons(80);
bind(fd,(struct sockaddr*)si,sizeof si);
listen(fd);
while ((cfd=accept(fd,(struct sockaddr*)si,sizeof si)) != -1) {
  read_request(cfd);   /* read(cfd,...) until "\r\n\r\n" */
  write(cfd,"200 OK HTTP/1.0\r\n\r\n"
            "That’s it.  You’re welcome.",19+27);
  close(cfd);
}
```

This server sucks!!!

This server (besides not actually implementing the protocol) has the drawback that it can only handle one client at a time. Here is a better one:

```
while ((cfd=accept(fd,(struct sockaddr*)si,sizeof si)) != -1) {
  if (fork()>0) continue;  /* handle connection in a child process */
  read_request(cfd);   /* read(cfd,...) until "\r\n\r\n" */
  write(cfd,"200 OK HTTP/1.0\r\n\r\n"
          "That’s it.  You’re welcome.",19+27);
  close(cfd);
  exit(0);
}
```

One process per connection – is that a good idea?

That’s the old school way. It’s been done this way since the 70ies.

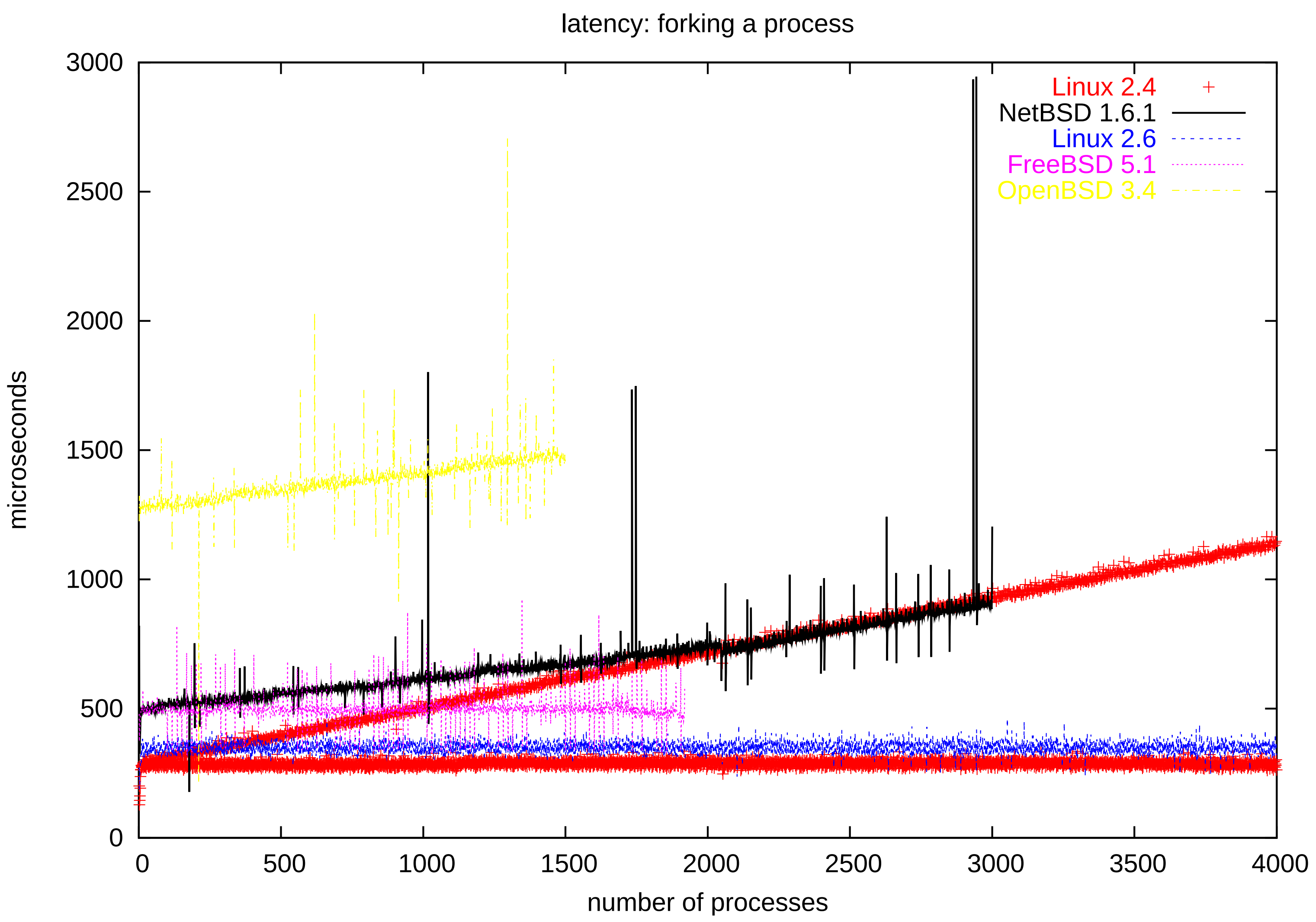
The first web server from CERN used this method.

One process per connection – is that a good idea?

Creating one process per connection does have scalability issues.

It is difficult to implement `fork()` properly. Here is a benchmark:

```
pipe(pfd);
for (i=0; i<4000; ++i) {
  gettimeofday(&a,0);
  if (fork()>0) {
    write(pfd[1],"+",1); block(); exit(0);
  }
  read(pfd[0],buf,1);
  gettimeofday(&b,0);
  printf("%llu\n",difference(&a,&b));
}
```


latency: forking a process
microseconds
number of processes
0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000
3500
4000
Linux 2.4
NetBSD 1.6.1
Linux 2.6
FreeBSD 5.1
OpenBSD 3.4

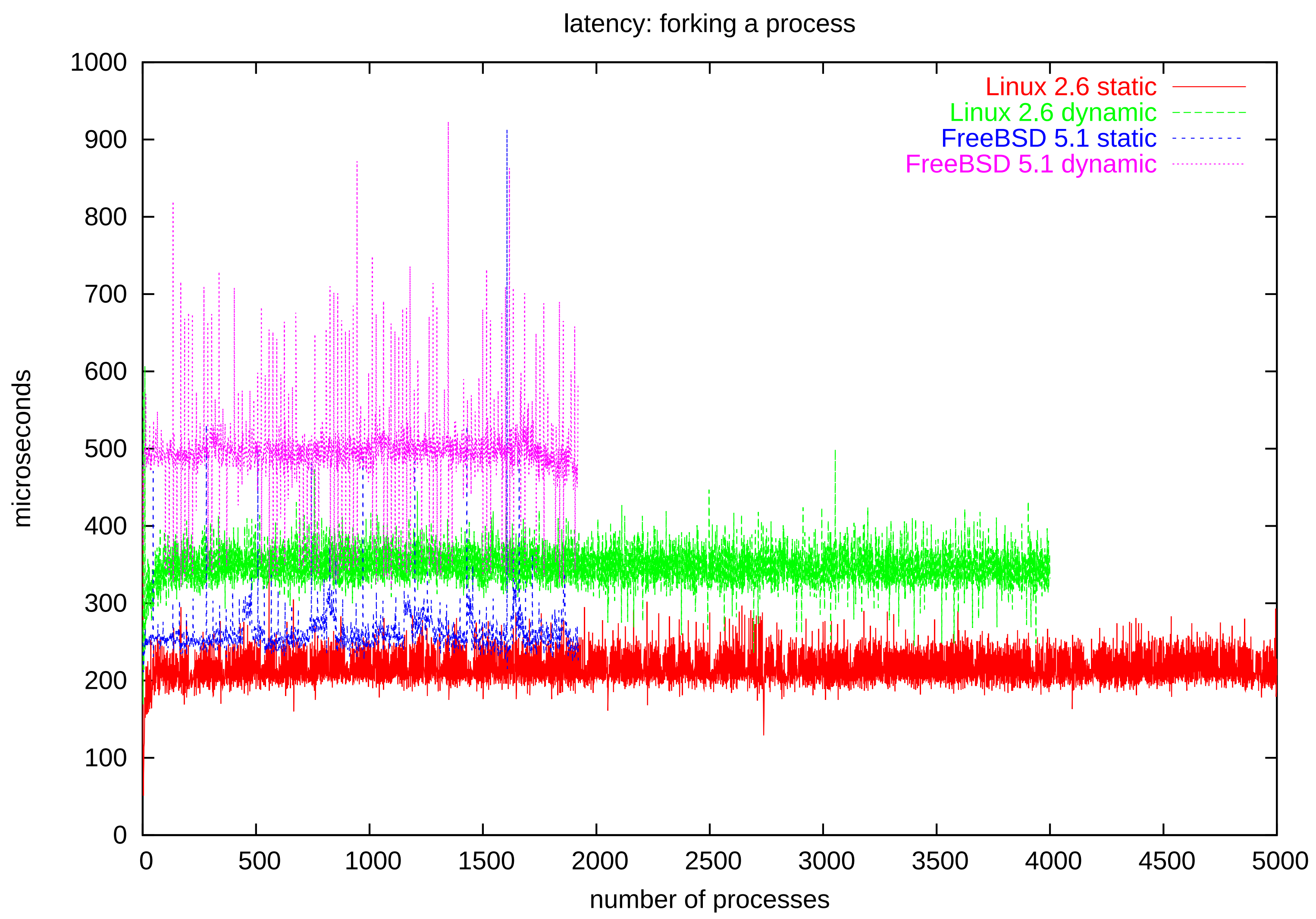
fork: dynamic linking versus static linking

Another important fork performance factor is how much fork has to do.

On modern platforms with a hardware memory management unit, fork only copies the page mappings.

Dynamic linking creates many page mappings for the ELF sections in the shared libraries, and the Global Offset Table and so on.

So, linking the fork benchmark statically ought to improve performance noticeably.


latency: forking a process
microseconds
number of processes
0
100
200
300
400
500
600
700
800
900
1000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Linux 2.6 static
Linux 2.6 dynamic
FreeBSD 5.1 static
FreeBSD 5.1 dynamic

Just to make sure you understood these numbers

The fork-and-do-something latency on my notebook on Linux 2.6 is 200 microseconds.

That means my notebook can create 5.000 processes per second.

Thus my notebook can handle about 13 billion forks per month.

My Athlon XP 2000+ desktop can do 10.000 processes per second, or 26 billion per month.

Heise Online, the biggest German site, had 118 million page impressions in September.

Scheduling

Why does the fork benchmark include writing to the pipe?

Because having many processes not only makes creating more processes more difficult, it also makes choosing which process to run more difficult.

The part of the operating system that chooses which process to run next is called the *scheduler*.

A typical workload is having two dozen processes, with one or two of them actually having something to do (they are *runnable*).

Scheduling

Linux interrupts running processes every 1/100th second (1/1000th on Alpha; this value is traditionally called HZ on Unix and Linux and is a compile time constant) to give others a chance to run.

The scheduler is also activated when a process blocks (waiting for input, for example).

The scheduler's job is to select the process that should run next. What makes this hard is the *fairness* requirement: all processes should get an equal share of the CPU. In particular, no process should starve.

It obviously makes sense to have two lists: one for the runnable processes, one for the blocked ones.

Scheduling

Unix has a mechanism to favour interactive processes over batch jobs. To that end, the kernel calculates a *nice* value for each process once per second.

When there are 10.000 processes, this thrashes the 2nd level cache.

This is especially bad for SMP, because the process table must be protected by a spinlock, i.e. the other CPUs can't switch processes while the first CPU calculates the *nice* value.

The typical solution from the commercial Unices is to have one run queue per CPU. Further optimizations include keeping the run queues sorted, for example with a heap, or having one run queue per priority.

The Linux 2.4 Scheduler

The Linux scheduler has one unsorted run queue with all runnable processes and one task list for all sleeping processes and zombies.

All operations on the run queue are protected by a spin lock.

There are several new experimental schedulers for Linux 2.4, with heap based priority queues or multiple run queues, but the most amazing scheduler is the O(1) scheduler by Ingo Molnar.

The O(1) scheduler is the default scheduler in Linux 2.6.

The Linux 2.6 Scheduler

The O(1) scheduler has two arrays of pointers to a linked list for each CPU. The arrays have one entry for each possible priority.

The linked lists contain all the tasks for the given priority. Since all the tasks in each list have the same priority, putting a process of a given priority into this data structure takes constant time.

One of the arrays is the current run queue; all the processes in it are run in turn, and when they used up their time slice they are moved to the other array. When the run queue is empty, the arrays are swapped.

Processes that the scheduler interrupted are assumed to be batch processes and punished. That is cheaper than favouring interactive processes.

How important is scheduler performance?

The scheduler is run 100 (HZ, actually) times per second, plus each time a process is blocking.

Each time the scheduler is suspending one process and switching to another one, the operating system increases a counter.

This ”context switch” counter can be seen using vmstat.

It is architecture dependent how expensive a context switch is. Basically, the more registers, the more expensive the context switch. So x86 actually has an unfair advantage over RISC, in particular over SPARC and IA64.

How important is scheduler performance?

Here is a vmstat, running an MP3 player:

```
  procs                memory       swap            io      system      cpu
r  b  w   free buff  cache   si    so     bi    bo    in     cs us sy id
0  0  0  60316 9136 316280    0     0    135   113    47      8  8  9 82
0  0  0  60296 9152 316280    0     0      0     0   200     54  3  0 97
```

And now with httpbench running over lo and eth0:

```
4  0  0  56136 9236 317936    0     0     24     0   228   7953  6 59 35
0  0  0  56308 9236 317936    0     0      0     0   221   1590  4 10 86

4  0  0  53476 9304 320180    0     0      0     0 16416  12319 26 74  0
5  0  0  52552 9304 320180    0     0      0     0 16391  12307 20 80  0
```

Other problems with running many processes

Creating very many processes has another problem: memory consumption.

Each process takes up memory. When a process forks a child, all the pages are set to copy-on-write in both processes. As soon as a process writes to a page, it gets copied.

This is good, but most people don't realize just how much their programs write all over the main memory.

For example, calling malloc or free may cause a chain reaction of coalescing data structures or rebalancing some tree in the free list.

Other problems with running many processes

Another example is the dynamic linker, which maintains the Global Offset Table, but does so lazily per default, i.e. the offsets in the table are updated only when the function is first called.

So if you call a function the first time after a fork, and you fork 1000 children, then you will have wasted 1000 4k pages.

You can tell the dynamic loader to update all offsets at program start by setting the `$LD_BIND_NOW` environment variable.

Please note that cleaning up 1000 processes is usually slower than forking 1000 processes, and *all* the tested systems are unresponsive during that time.

Memory Consumption

Measuring your own memory consumption is not so easy on Unix.

There is a `getrusage` system call to do it, but Linux always returns 0 for the memory usage, and FreeBSD only returns data for dynamically linked programs.

On Linux, `/proc/self/statm` contains the data:

```
% ../show/memusage-glibc-c
1164K total size, 324K resident, 1132K shared
% ../show/memusage-glibc-static-c
364K total size, 112K resident, 348K shared
% ../show/memusage-diet-c
32K total size, 32K resident, 16K shared
```

The one-process-per-connection model

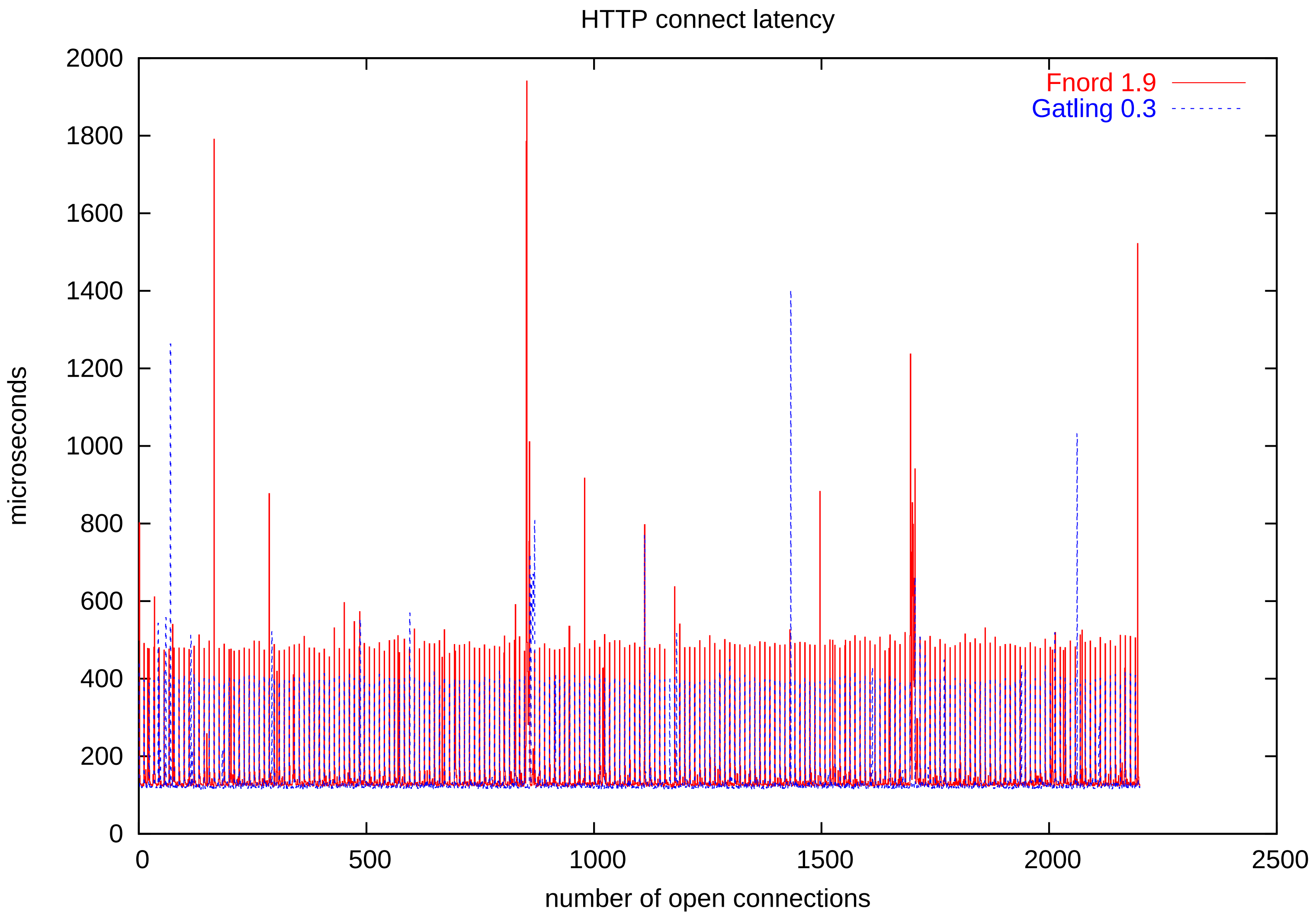
This programming model works so well, there are standard tools for it.

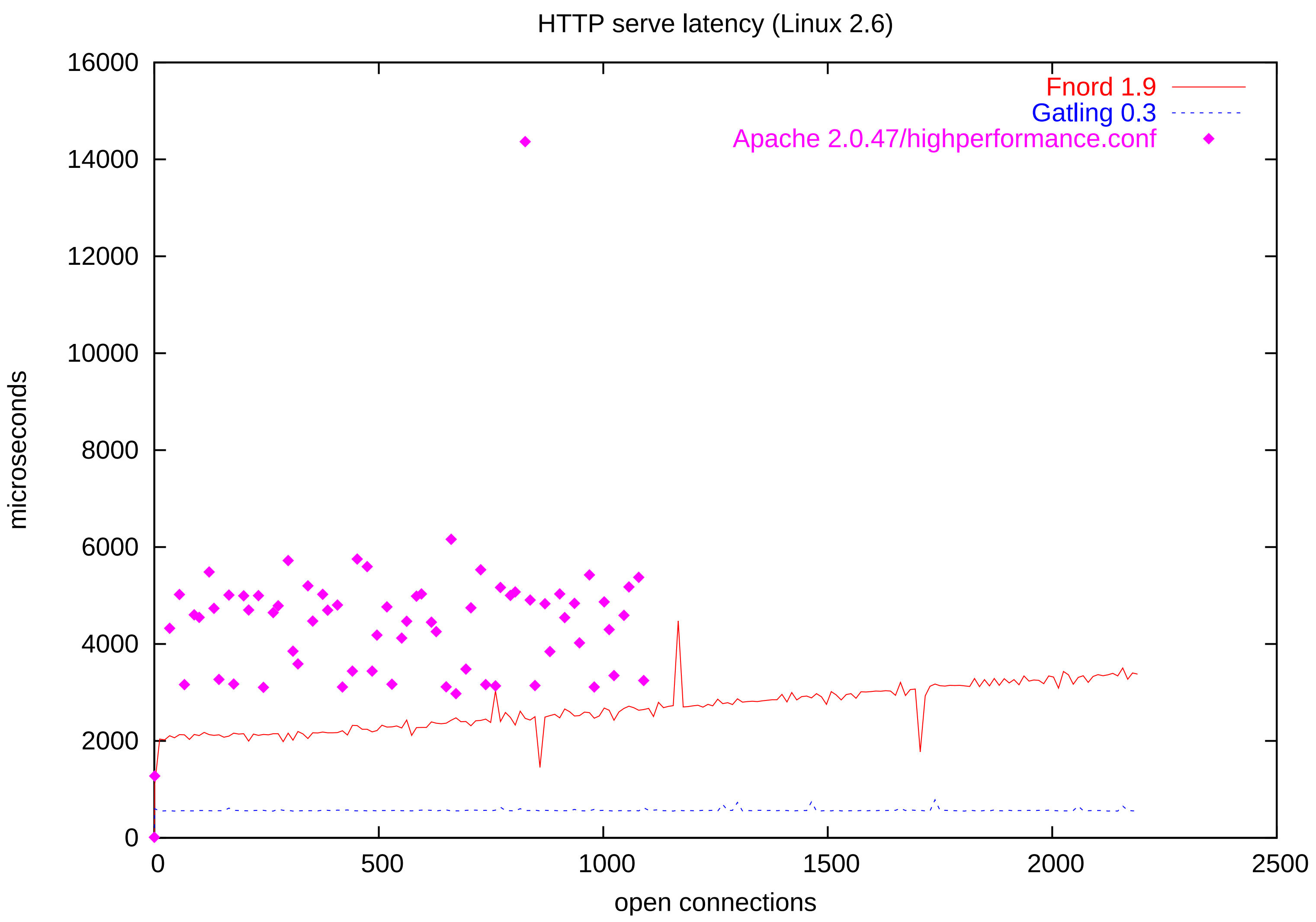
Unix comes with the `inetd` server that not only forks a new process for each connection, it also executes an external program for each connection.

Unfortunately, inetd has certain shortcomings that have given it (and the whole network programming model) a bad name, so some people wrote their own replacements, for example tcpserver, xinetd, ipsvd.

This is slightly less efficient than simply forking, but it has the advantage that you can standardize the IP access control mechanism.

I wrote my first web server, *fnord*, using this model. It still serves www.fefe.de.


HTTP connect latency
microseconds
number of open connections
Fnord 1.9
Gatling 0.3
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
500
1000
1500
2000
2500


HTTP serve latency (Linux 2.6)
Fnord 1.9
Gatling 0.3
Apache 2.0.47/highperformance.conf
microseconds
open connections
0
2000
4000
6000
8000
10000
12000
14000
16000
0
500
1000
1500
2000
2500

The one-process-per-connection model

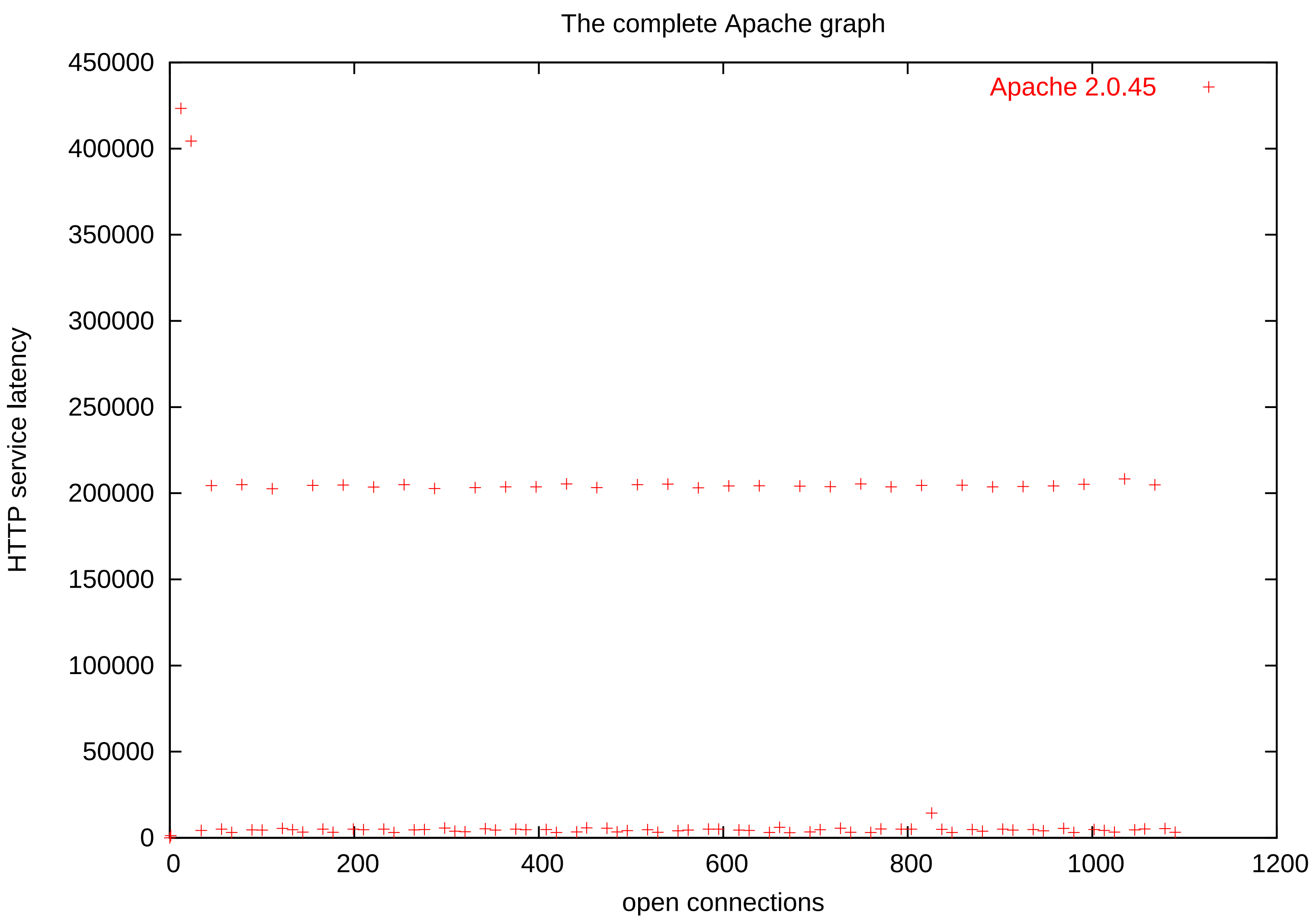
Apache also uses the one process per connection model.

However, to save the fork latency, Apache pre-forks, i.e. it forks first and then accepts and delegates the HTTP connection to the child.

But wait! You haven’t seen the whole Apache graph yet!

I clipped away most of it, because then the gatling and fnord measurements wouldn’t have been visible on the combined graph.

In conclusion: Apache performance sucks.


The complete Apache graph
Apache 2.0.45
HTTP service latency
open connections
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200

Since processes are so slow, why not use threads instead?

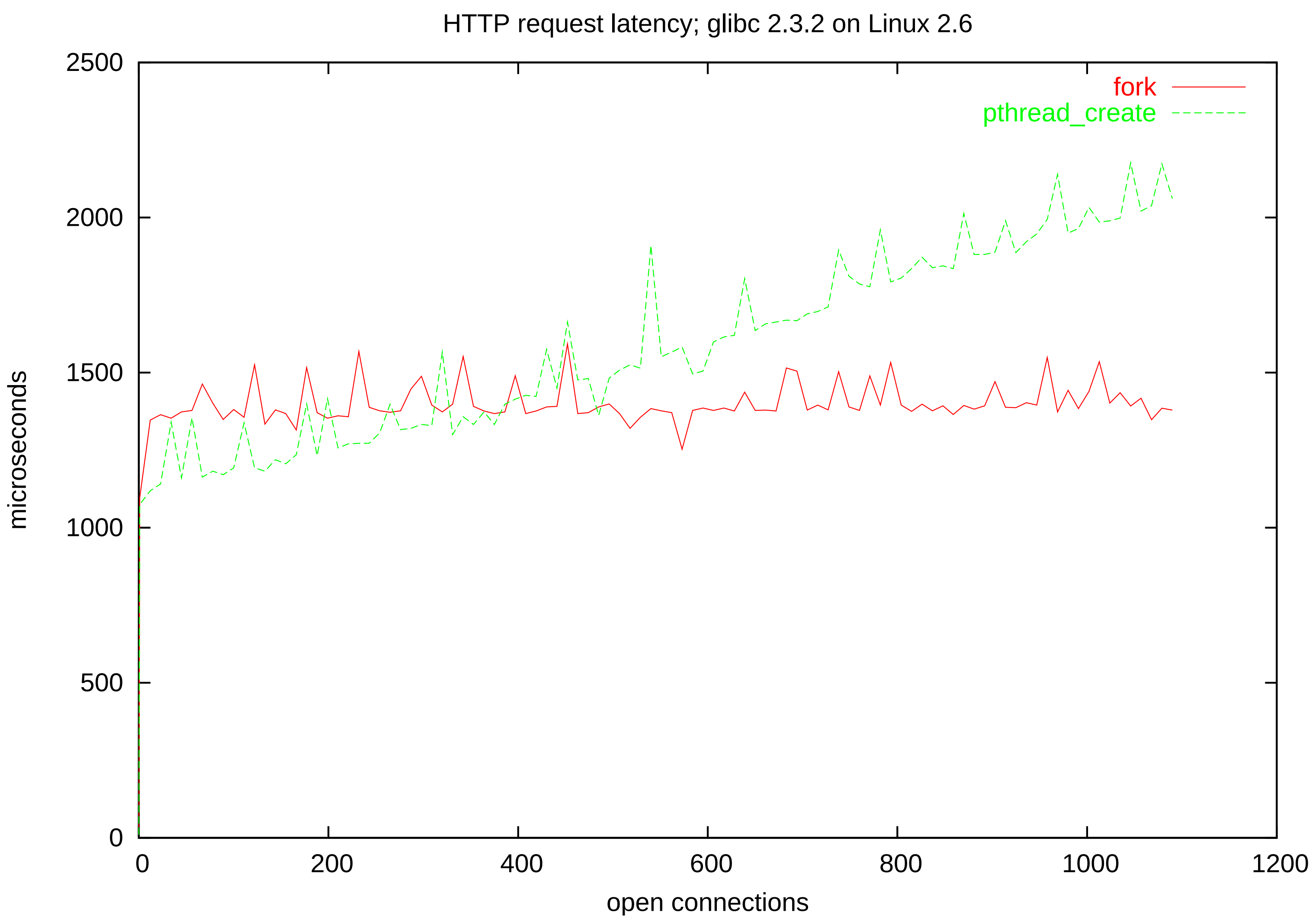
It is common knowledge that fork is slow and we should use threads instead.

The truth is more complex than that.

fork is not universally slow, as you have seen. But on some systems, fork is **really** slow. One of these systems is Solaris.

Apparently it was too difficult to fix `fork` (or all the engineers were busy inventing Java), so threads were invented instead.

Two years ago, I measured that `pthread_create` on Solaris is slower than `fork` on Linux on the same hardware. However, I lost access to that hardware, so I don't have that graph with me today. I do have this:


HTTP request latency; glibc 2.3.2 on Linux 2.6
microseconds
open connections
fork
pthread_create
0
500
1000
1500
2000
2500
0
200
400
600
800
1000
1200

Multithreading performance

One problem with threads is that on a bad OS, creating threads is still slow. Solaris and Windows may create threads faster than processes, but it is still way too slow.

So what did they do? They invented *thread pools*.

A thread pool works like Apache’s pre-forking. The threads are created en block at the program start, and then the work load is distributed evenly.

When there are more connections than threads, the new connections have to wait. But on the plus side, you save the cost for the thread creation.

I don’t know about you, but I wouldn’t call that ”scalable”.

Positive aspects of threading

There are some good aspects to Java and threading as well! We got new, faster hardware, and the RAM prices are on an all-time low!

Same for software: the cretins on the application layer forced great innovations on the OS layer. Because Lotus Notes keeps one TCP connection per client open, IBM contributed major optimizations for the ”one process, 100.000 open connections” case to Linux.

And the O(1) scheduler was originally created to score well on some irrelevant Java benchmark.

The bottom line is that this bloat benefits all of us. We just need to make sure that there are always small and efficient alternatives to all the crapware.

How is timeout handling done?

One important question for network servers is: how do you detect timeouts?

A network server needs to detect when a client connects and then does nothing with the connection.

No matter how well optimized your network server is, you (and the OS) will always have to keep state for established connections, and those system ressources are better spent on handling active connections.

How is timeout handling done?

Unix has a standard function called `alarm` that is well suited for this:

```
alarm(23);   /* deliver SIGALRM in 23 seconds */
```

Receiving an unhandled signal terminates the program.

So, in 23 seconds, the kernel will terminate the web server.

Later calls to `alarm` override the previous alarm.

You simply call alarm every time you get activity on the socket.

Timeout handling with select()

`select` waits for events on one or more file descriptors (select appeared 1983 in 4.2BSD):

```
fd_set rfd;
struct timeval tv;
FD_ZERO(&rfd); FD_SET(0,&rfd); /* fd 0 */
tv.tv_sec=5; tv.tv_usec=0;     /* 5 seconds */
if (select(1,&rfd,0,0,&tv)==0) /* read, write, error, timeout */
  handle_timeout();
if (FD_ISSET(0, &rfd))
  can_read_on_fd_0();
```

The timeout value is given in microsecond resolution, but no Unix actually offers this accuracy. The first argument is the highest fd plus one. *Puke*

Disadvantages of select()

select does not say how long it waited for the events. So you still have to call `gettimeofday()` manually. Actually, select does say on Linux, but that is not portable and actually breaks portable software.

select works on bit vectors. The size of these is OS dependent. If you are lucky, you get 1024 (on Linux, for example). Many other Unices have less.

This is worse than it looks. Some bad DNS libraries use select for their timeout handling. If you open 1024 files, DNS suddenly stops working, because the DNS socket is over 1024! Apache works around this by manually keeping the descriptors below 15 free (with `dup2`).

Timeout handling with poll()

poll is a variant on select (poll appeared 1986 in System V R3):

```
struct pollfd pfd[2];
pfd[0].fd=0; pfd[0].events=POLLIN;
pfd[1].fd=1; pfd[1].events=POLLOUT|POLLERR;
if (poll(pfd,2,1000)==0) /* 2 records, 1000 milliseconds timeout */
  handle_timeout();
if (pfd[0].revents&POLLIN) can_read_on_fd_0();
if (pfd[1].revents&POLLOUT) can_write_on_fd_1();
```

Pro: no limit on the number of records and descriptors.

Con: poll is not available on some museum exhibitions. Only one current "Unix" disqualifies itself for networking by not offering poll: MacOS X.

Disadvantes of poll()

The whole array is unnecessarily copied around between user and kernel space. The kernel then finds out about the events and sets the corresponding `revents`.

Current CPUs spend most of their time waiting for memory anyway. poll makes that even worse, and the wasted effort rises linearly with the number of file descriptors.

This is not as bad as it looks. If poll takes longer, the events are not lost. The kernel will queue the events and signal them on the next poll.

On the other hand, on Linux and FreeBSD a fork based web server actually scales better than a poll based one.

Linux 2.4: SIGIO

Linux 2.4 can tell you about poll events via signals as well.

```
int sigio_add_fd(int fd) {
  static const int signum=SIGRTMIN+1;
  static pid_t mypid=0;
  if (!mypid) mypid=getpid();
  fcntl(fd,F_SETOWN,mypid);
  fcntl(fd,F_SETSIG,signum);
  fcntl(fd,F_SETFL,fcntl(fd,F_GETFL)|O_NONBLOCK|O_ASYNC);
}

int sigio_rm_fd(struct sigio* s,int fd) {
  fcntl(fd,F_SETFL,fcntl(fd,F_GETFL)&(~O_ASYNC));
}
```

Linux 2.4: SIGIO

SIGIO is *no* direct poll replacement. poll tells you when a descriptor **is** readable. SIGIO tells you when it **becomes** readable.

If poll says you can read from file descriptor 3 and you don't do anything about it, the next call to poll will tell you again. SIGIO won't. The poll way is called *level triggered*, the SIGIO way is called *edge triggered*.

The best way to get the events is `sigtimedwait`, after you block `SIGIO` (the signal) and `signum`. That restores synchronicity and avoids the need for locking and reentrant functions.

Linux 2.4: SIGIO

```
for (;;) {
  timeout.tv_sec=0;
  timeout.tv_nsec=10000;
  switch (r=sigtimedwait(&s.ss,&info,&timeout)) {
  case -1: if (errno!=EAGAIN) error("sigtimedwait");
  case SIGIO: puts("SIGIO queue overflow!"); return 1;
  }
  if (r==signum) handle_io(info.si_fd,info.si_band);
}
```

`info.si_band` is identical to `pollfd.revents`.

Disadvantages of SIGIO

SIGIO has disadvantes, too. The kernel has an event queue, and the events are delivered in order. The kernel may signal an event on a file descriptor you already closed.

The queue has a fixed length, and you can't say how large you need it to be. If the queue is full, you get a `SIGIO` signal and no events at all.

You are then expected to flush the queue (by setting the handler for `signum` to `SIG_DFL`) and get the events using poll.

This API reduces the memory traffic, but now you have one syscall per event. Syscalls are cheap on Linux, but not free.

Handling queue overflows is very annoying. One of the problems with poll is that you don't want to maintain the pollfd array.

What’s wrong with the pollfd array?

We write a web server. `poll` tells us to read from fd 5. `read` tells us the connection was dropped. We now `close` the descriptor and remove the record from the array.

How do we do that? We can’t just set the events in the record to 0, because poll will then abort with EBADF and set revents to POLLNVAL. So we need to copy the last descriptor in the array over this one and reduce the array size.

Now entry 5 in the array is no longer the entry for file descriptor 5! So we need an extra index array to find the pollfd for a given descriptor. This index needs to be shrunk and grown and maintained. That is ugly and error prone.

And it is perfectly superfluous.

/dev/poll

Sun added a new poll style API to Solaris a few years ago. You open the device `/dev/poll`, and then `write` the pollfds to that device. You wait for events with `ioctl`.

The `ioctl` then returns how many events are there, and this many pollfds you can then `read` from the device. The device only gives you pollfds which had actual activity.

This means you don't have to scan the array for the ones with events.

There were several approaches to create such a device for Linux. None of it made it into the kernel proper.

Oh yes, and the patches were unstable under high load.

/dev/epoll

There is a patch for Linux 2.4 that adds a `/dev/epoll` device.

```
int epollfd=open("/dev/misc/eventpoll",O_RDWR);
char* map;
ioctl(epollfd,EP_ALLOC,maxfds); /* hint: number of descriptors */
map=mmap(0, EP_MAP_SIZE(maxfds), PROT_READ, MAP_PRIVATE, epollfd, 0);
```

You signal interest in an event by writing the pollfd to the device. You signal disinterest by writing a pollfd with events==0 to the device.

/dev/epoll

The events are fetched with an `ioctl`:

```
struct evpoll evp;
for (;;) {
  int n;
  evp.ep_timeout=1000;
  evp.ep_resoff=0;
  n=ioctl(e.fd,EP_POLL,&evp);
  pfds=(struct pollfd*)(e.map+evp.ep_resoff);
  /* now you have n pollfds with events in pfds */
}
```

Because of `mmap`, there is no actual copying between kernel and user space.

/dev/epoll

The disadvantage of /dev/epoll is that it is just a patch. Linus does not like new pseudo devices in the kernel.

He says we already have a dispatcher for the syscalls in the kernel, so if we want to add something, we add a syscall and not a device or an ioctl.

So the author of /dev/epoll redid the API using syscalls. This API is was merged into Linux 2.5 (since 2.5.51 even in the documented form ;-)).

It is now the recommended API for event notification in Linux 2.6.

epoll

```
int epollfd=epoll_create(maxfds);
struct epoll_event x;
x.events=EPOLLIN|EPOLLERR;
x.data.ptr=whatever; /* you can put some cookie here */
epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&x);
/* changing is analogous: */
epoll_ctl(epollfd,EPOLL_CTL_MOD,fd,&x);
/* deleting -- only x.fd has to be set */
epoll_ctl(epollfd,EPOLL_CTL_DEL,fd,&x);
```

The EPOLLIN,... constants have the same value as POLLIN,... but the author wanted to keep all options open. epoll started out edge triggered but is now level triggered. Add `|EPOLLET` to switch back.

epoll

This is how you fetch the events:

```
for (;;) {
  struct epoll_event x[100];
  int n=epoll_wait(epollfd,x,100,1000); /* 1000 milliseconds */
  /* x[0] .. x[n-1] are the events */
}
```

Note that `epoll_event` does not contain the actual descriptor!

That's why the API has a cookie. You can put the file descriptor there, but you can also put a pointer to some struct with the context data for this connection there.

FreeBSD: kqueue

kqueue is a cross between epoll and SIGIO. Like epoll you can have edge or level triggered events. kqueue can also do file and directory status notification. The problem is: the API is unwieldly and not well documented.

kqueue is older than epoll. I think Linux should simply have implemented the kqueue API instead of inventing epoll, but the Linux people insist on doing all the mistakes of the other people again. For example, the epoll guy initially thought he could get away without level triggering.

The performance of epoll and kqueue is very similiar.

kqueue is also implemented in OpenBSD, but not in NetBSD.

FreeBSD: kqueue

This is how you signal interest or disinterest in an event:

```
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>
int kq=kqueue();
struct timespec ts;
EV_SET(&kev, fd, EVFILT_READ, EV_ADD|EV_ENABLE, 0, 0, 0);
ts.tv_sec=0; ts.tv_nsec=0;
kevent(kq, &kev, 1, 0, 0, &ts);  /* I want to read */

EV_SET(&kev, fd, EVFILT_READ, EV_DELETE, 0, 0, 0);
ts.tv_sec=0; ts.tv_nsec=0;
kevent(kq, &kev, 1, 0, 0, &ts);  /* I'm done reading */
```

FreeBSD: kqueue

Here is how you fetch the events:

```
struct kevent ev[100];
struct timespec ts;
ts.tv_sec=milliseconds/1000;
ts.tv_nsec=(milliseconds%1000)*1000000;
if ((n=kevent(io_master,0,0,y,100,milliseconds!=-1?&ts:0))==-1)
  return -1;
for (i=0; i<n; ++i) {
  if (ev[i].filter==EVFILT_READ) can_read(ev[i].ident);
  if (ev[i].filter==EVFILT_WRITE) can_write(ev[i].ident);
}
```

Windoze: Completion Ports

Even Microsoft needs to offer something in this field.

The Microsoft solution is a thread pool, each thread running something SIGIO-like mechanism.

This combines the disadvantages of threads with the disadvantages of SIGIO. It is amazing how Microsoft's marketing machinery managed to blow this up into an epic innovation.

I found this quote in the documentation: *"First of all, threads are system resources that are neither unlimited nor cheap."*

Huh? Not cheap? I thought that was the reason for their existance!

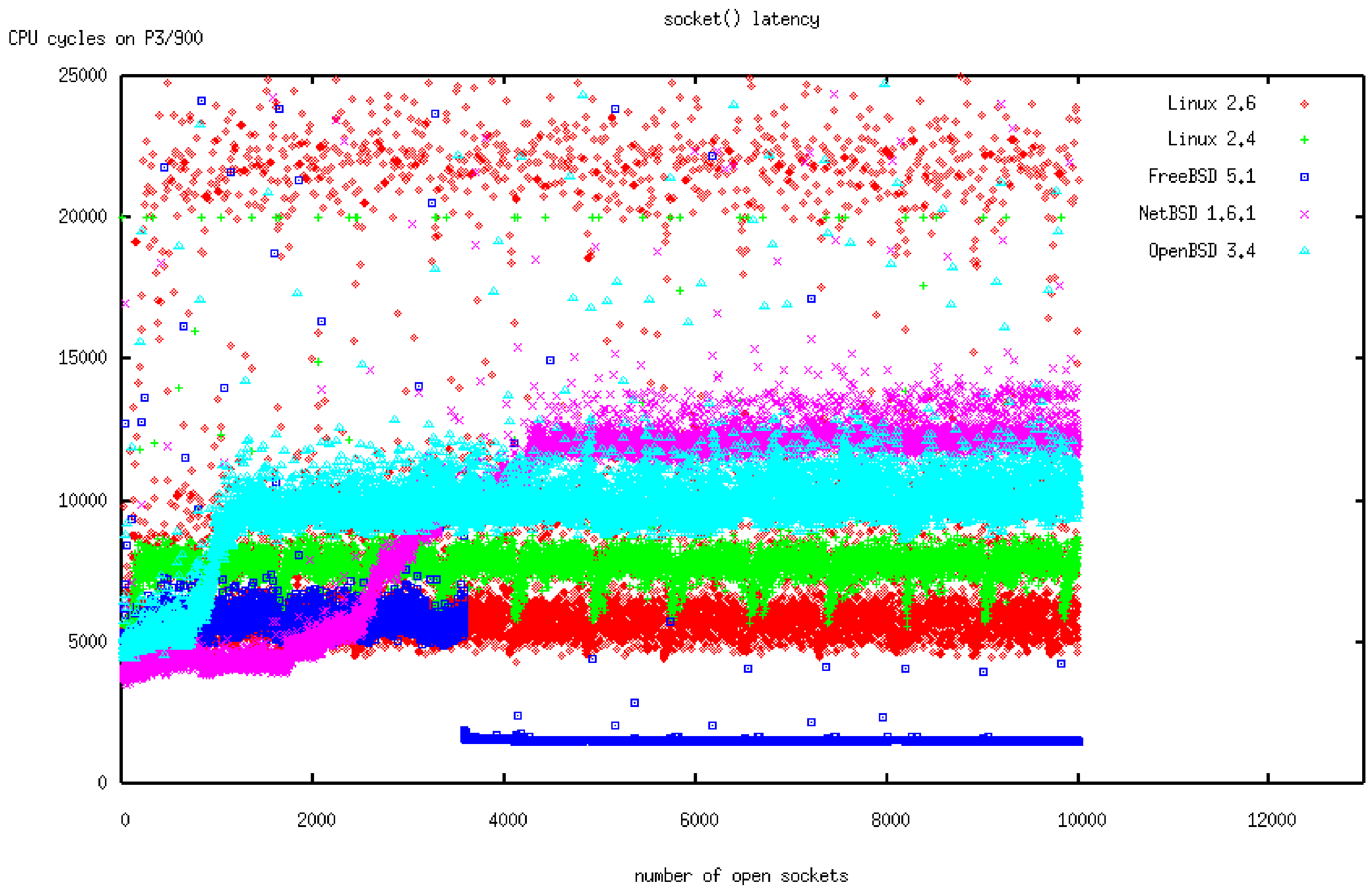
Other reasons for high latency

The POSIX API says that the kernel always has to use the lowest unused file descriptor when creating a new one (i.e. when accepting a connection, opening a file, creating a socket).

There is no known way to do this in O(1). Linux uses bit vectors.

The Linux kernel mailing list has carried numerous discussions about adding an open flag allowing the kernel to return a higher file descriptor, but it hasn't happened yet. And it wouldn't help for sockets.

For Linux and all BSDs this scales well. Here is a graph:


socket() latency
CPU cycles on P3/900
25000
20000
15000
10000
5000
0
0
2000
4000
6000
8000
10000
12000
number of open sockets
Linux 2.6
Linux 2.4
FreeBSD 5.1
NetBSD 1.6.1
OpenBSD 3.4

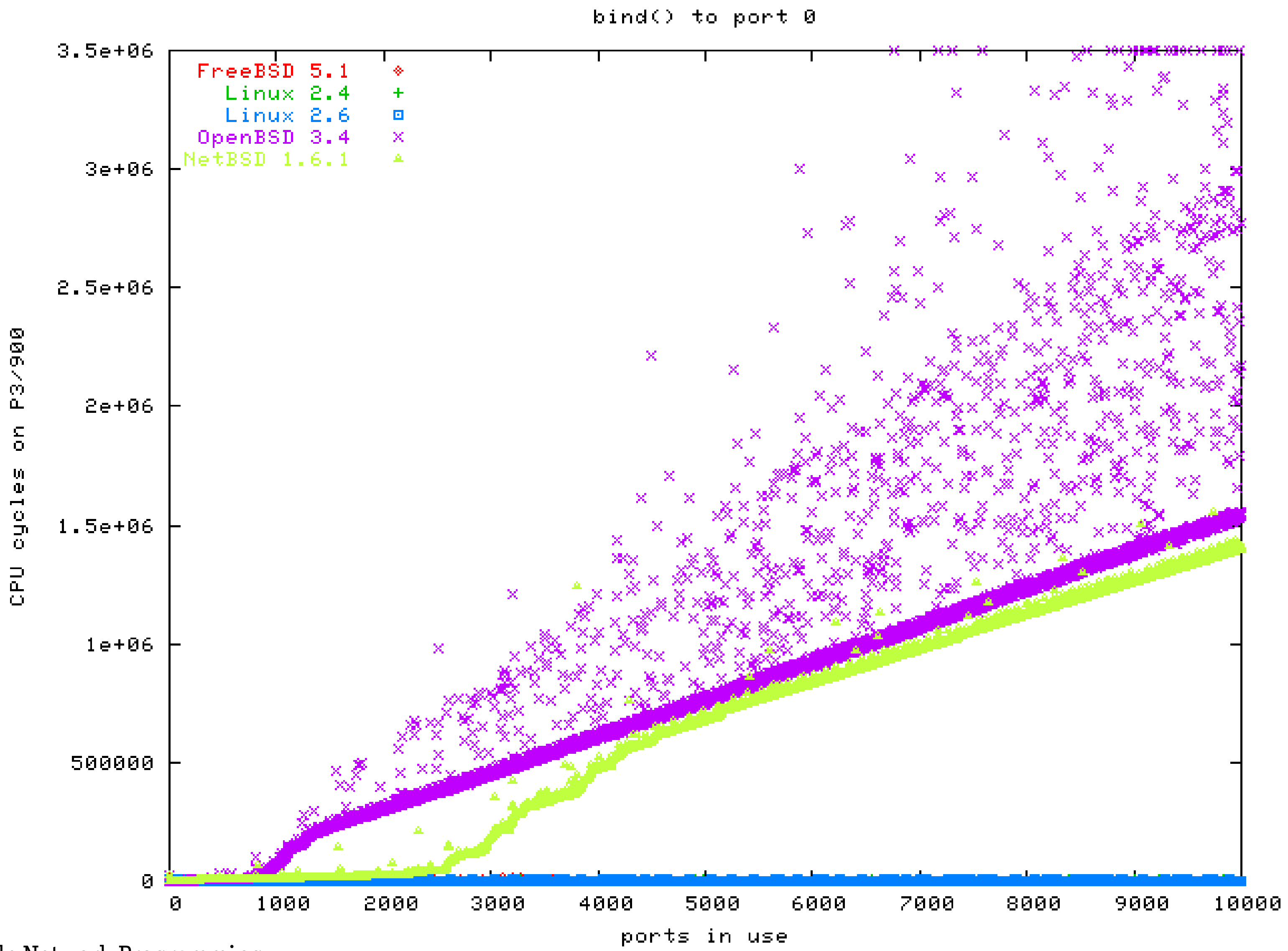
Other reasons for high latency

There is a similar problem for clients and proxies.

When you open a client socket to connect somewhere, and you don't explicitly choose a port, the kernel has to choose a free one for you.

No standard says the kernel has to return the lowest port, but this is often overlooked as problem.

My tests turned out that there is no significant difference between IPv4 and IPv6 latencies on any system, so I only include a graph for IPv4:


bind() to port 0
FreeBSD 5.1
Linux 2.4
Linux 2.6
OpenBSD 3.4
NetBSD 1.6.1
CPU cycles on P3/900
ports in use
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Fragmentation and seeking

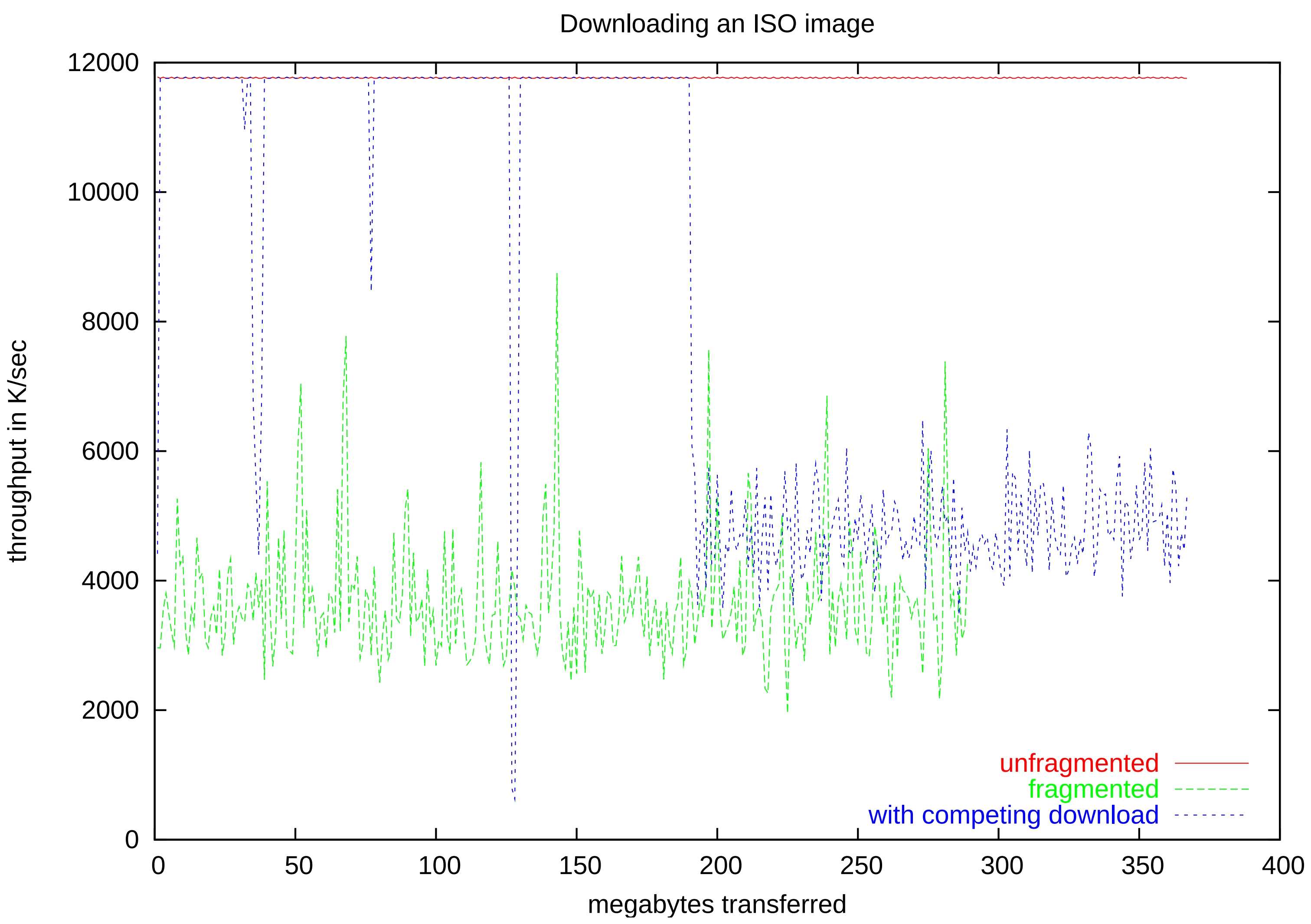
Hard disks are fast when reading sequentially, but slow when they have to move the head (i.e. *seek*).

This is why big server irons use hard disks with 10k or even 15k rpm.

Disk throughput has ceased to be a major concern quite a while ago.

Here is a graph. The hard disk can do 25 MB/sec, i.e. more than twice the Fast Ethernet capacity. I started a competing download limited to 5 MB/sec on loopback, for another file. The throughput for the Ethernet download halves.

The graph also shows how file fragmentation can slow down read access.


Downloading an ISO image
throughput in K/sec
megabytes transferred
0
2000
4000
6000
8000
10000
12000
0
50
100
150
200
250
300
350
400
unfragmented
fragmented
with competing download

sendfile

`sendfile()` is like `write()`, only directly from a file descriptor to a socket, i.e. no buffer and no `read()`.

That eliminates copying the file data from the buffer cache to a buffer in user space.

Current NICs (all gigabit ethernet NICs) can do scatter-gather I/O, i.e. they can take the packet header from a kernel buffer, but the packet contents from the buffer cache (note: csum_partial_copy_from_user).

The result is called *Zero Copy TCP* and it is the ultimate goal.

Linux and FreeBSD have (different) sendfile syscalls.

NetBSD and OpenBSD don't have sendfile.

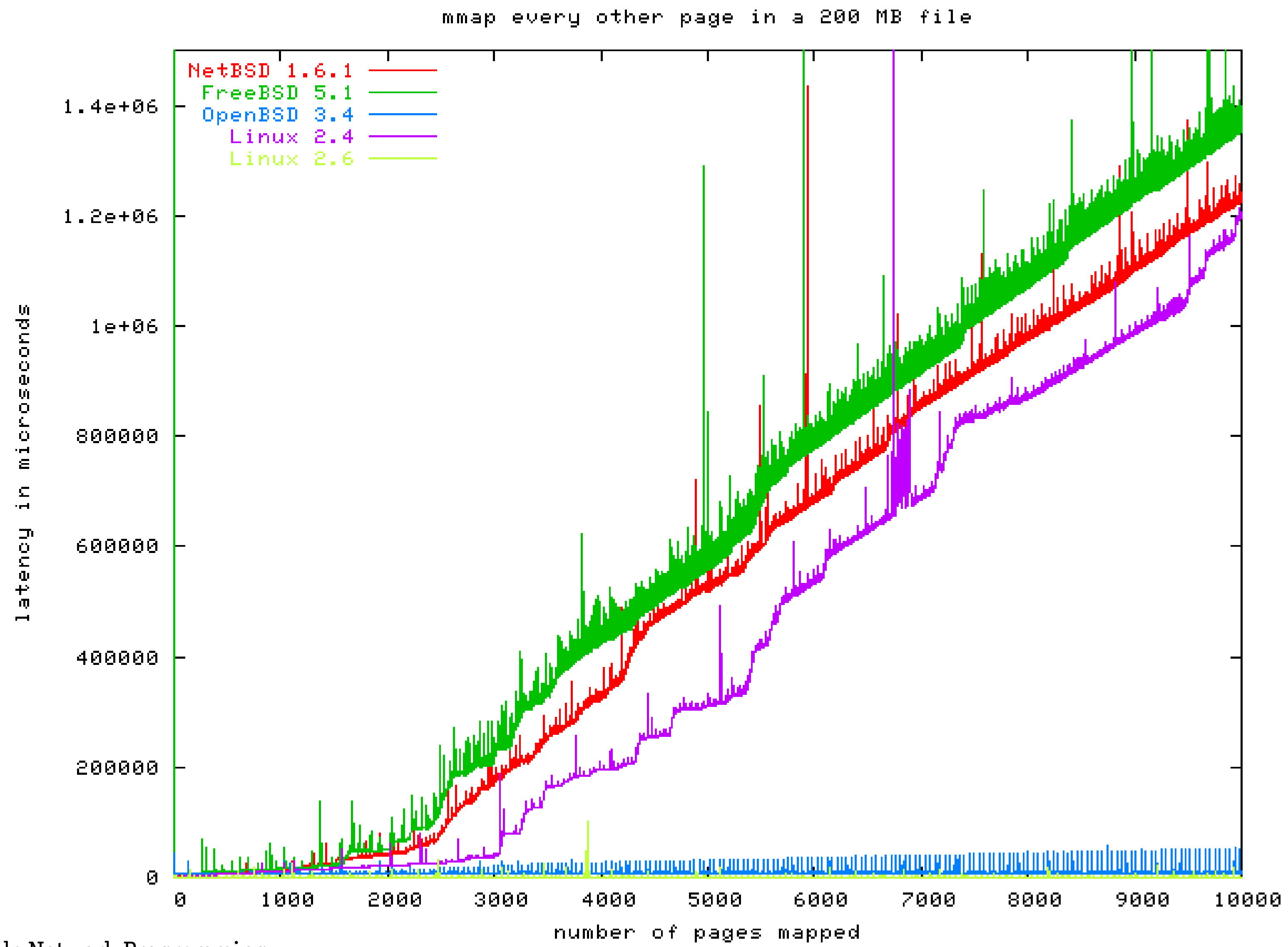
Memory Mapped I/O

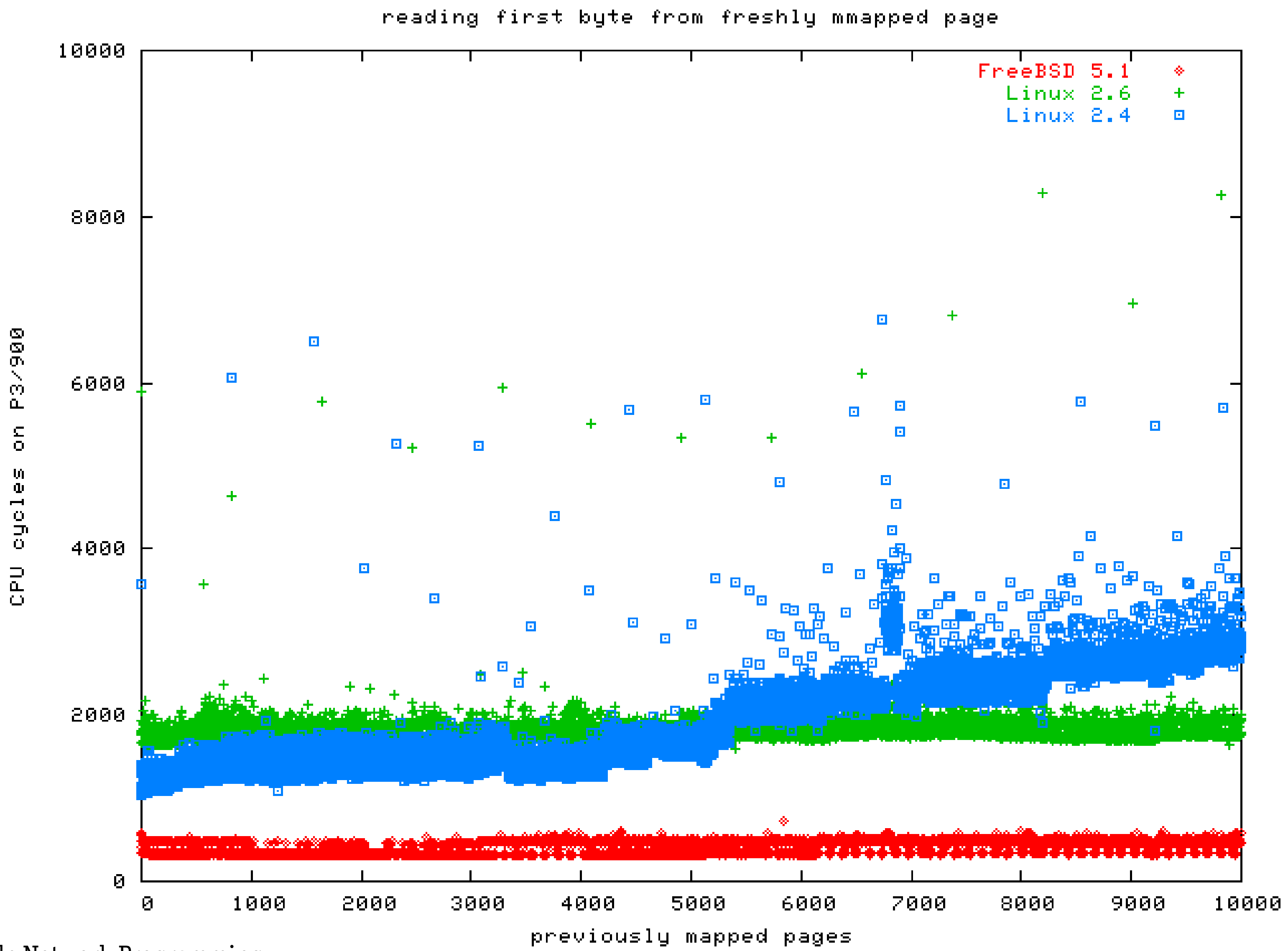
Instead of reading from a file and then doing something read-only with the file contents in the buffer, it is also possible to map the file into user space memory.

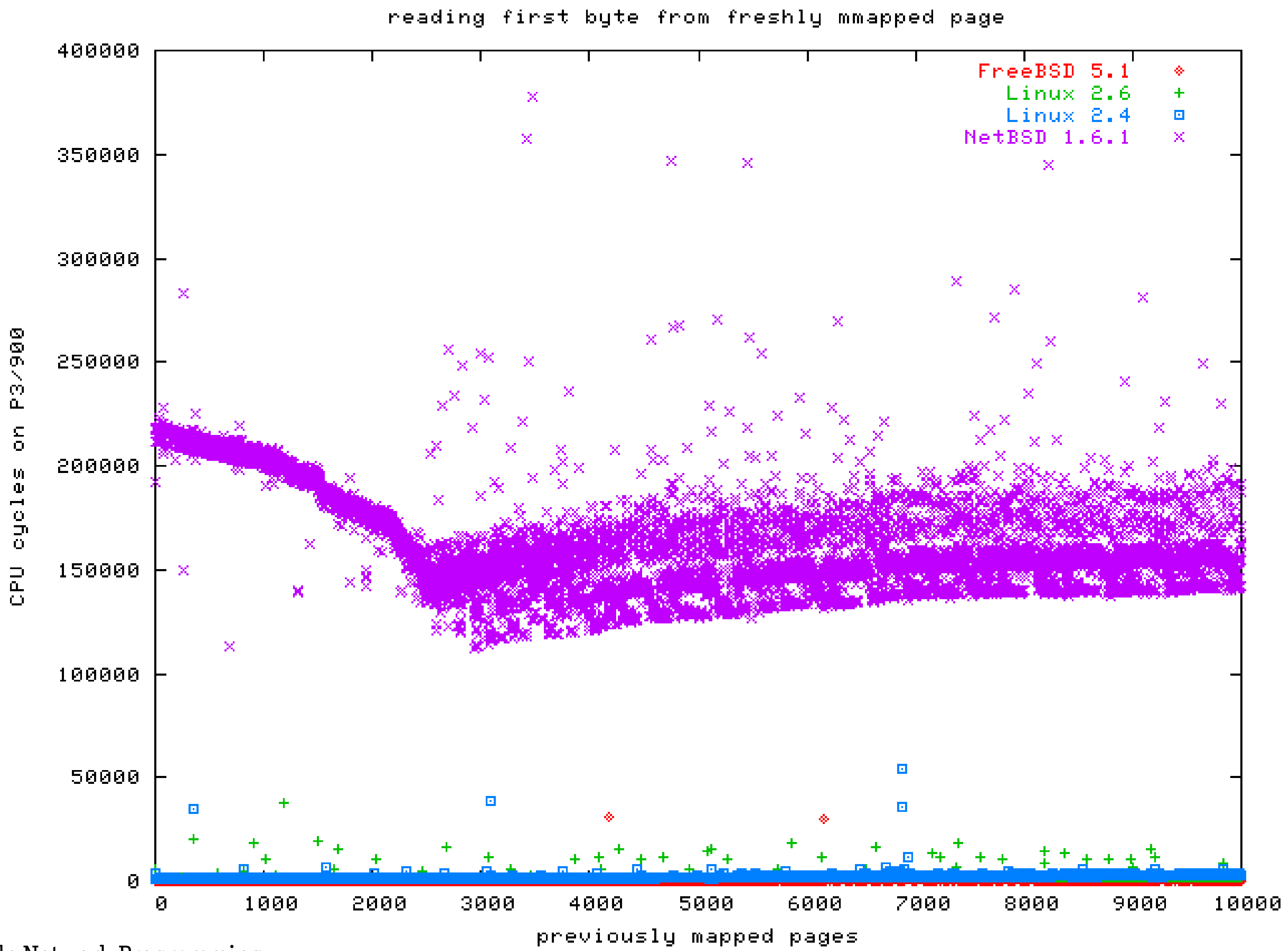
The syscall for this is called mmap, and it is very important for scalable network I/O, because it eliminates the buffer.

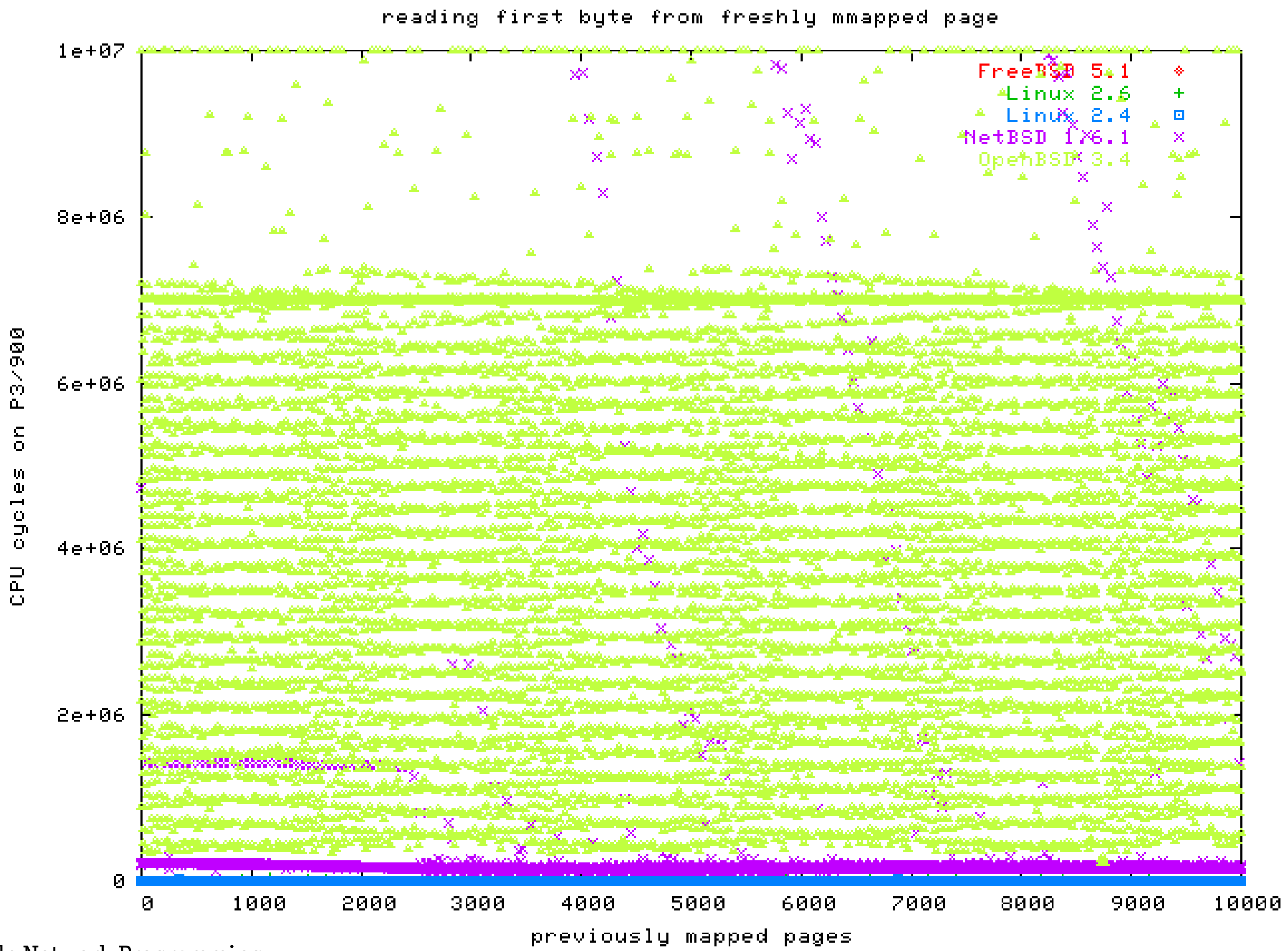
The OS can put that saved memory to good use in the buffer cache.

However, maintaining the page tables is difficult to get to scale properly. This will be even more important on 64-bit machines. Here are a few graphs:


mmap every other page in a 200 MB file
NetBSD 1.6.1
FreeBSD 5.1
OpenBSD 3.4
Linux 2.4
Linux 2.6
latency in microseconds
1.4e+06
1.2e+06
1e+06
800000
600000
400000
200000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
number of pages mapped


reading first byte from freshly mmapped page
CPU cycles on P3/900
previously mapped pages
FreeBSD 5.1
Linux 2.6
Linux 2.4
0
2000
4000
6000
8000
10000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


reading first byte from freshly mmapped page
CPU cycles on P3/900
previously mapped pages
0
50000
100000
150000
200000
250000
300000
350000
400000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
FreeBSD 5.1
Linux 2.6
Linux 2.4
NetBSD 1.6.1


reading first byte from freshly mmapped page
CPU cycles on P3/900
previously mapped pages
1e+07
8e+06
6e+06
4e+06
2e+06
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
FreeBSD 5.1
Linux 2.6
Linux 2.4
NetBSD 1.6.1
OpenBSD 3.4

Filesystem latency

This is usually not a problem these days, but there are pathological cases.

I once ran an FTP server with a big /incoming at a LAN party.

Someone decided to upload his whole porn collection as single JPEG files.

I noticed and kicked him out after 50.000 files, because the server had become very sluggish.

The OS spent close to 100% CPU time in the kernel, traversing the in-memory list of files in that directory over and over again.

These days we have advanced file systems like XFS and reiserfs specializing in this case, and there is directory hashing support for ext3 and FreeBSD UFS.

Asynchronous I/O

POSIX specifies an API for asynchronous I/O. Unfortunately almost nobody implements it. Linux doesn't, for example.

glibc has an emulation that creates one thread for each request. This is much worse than not having the API in the first place, so nobody uses the API.

The idea is to queue a read request. Later, you can then ask the OS if it has finished the request in the mean time. Or, you can get a signal when the kernel is done.

The problem is: you have no idea *which* of the queued requests was finished when you get the signal.

Asynchronous I/O

Async I/O is meant for files, not sockets. If you want to read 1000 blocks from a terabyte database, and you use `lseek` and `read`, you force the OS to read the data in the specified order. That forces the path of the hard disk head.

With async I/O, the system can sort the blocks so that the hard disk head will move only once over the whole disk. In principle, this is wonderful. In practice, it's worthless.

Solaris, for example, has their own proprietary async I/O API. They also offer the POSIX API, only to return ENOSYS for all calls.

The most important OS offering async I/O is FreeBSD.

writev, TCP_CORK and TCP_NOPUSH

An HTTP server writes a header and then the contents of a file to a socket. If you call `write` (and then `sendfile`), the kernel sends one TCP packet for the header and starts a fresh one for the body.

If the file is only 100 bytes large, it would have fit into one packet.

Obviously this can be solved by having a buffer and copying both the header and the file contents there. But we want zero-copy TCP, remember?

There are four solutions: `writev`, `TCP_CORK` (Linux), `TCP_NOPUSH` (BSD) and the FreeBSD `sendfile`.

writev

writev is like a batch write. You give it a vector of pointers and lengths, and it writes them all out.

The difference is too small to notice normally, except for TCP connections.

```
struct iovec x[2];
x[0].iov_base=header; x[0].iov_len=strlen(header);
x[1].iov_base=mapped_file; x[1].iov_len=file_size;
writev(sock,x,2); /* returns bytes written */
```

TCP_CORK

```
int null=0, eins=1;

/* put cork in bottle */
setsockopt(sock,IPPROTO_TCP,TCP_CORK,&eins,sizeof(eins));
write(sock,header,strlen(header));
write(sock,mapped_file,file_size);
/* pull cork out of bottle */
setsockopt(sock,IPPROTO_TCP,TCP_CORK,&null,sizeof(null));
```

TCP_NOPUSH from BSD does the same, but you have to set the flag to zero *before* writing the last write, which is a little awkward.

The sendfile from FreeBSD is like the Linux sendfile plus one writev style vector for headers and one for footers, so you don't need TCP_NOPUSH.

vfork

fork is very fast on modern Unices, because no actual memory contents is copied, only the page tables.

However, this can still be comparatively expensive, especially if you were going to exec a CGI program anyway. That's why Linux and the BSDs have implemented a traditional vfork again.

But vfork is not universally faster! On Linux 2.6, vfork is actually slower (250 ms vs. 180 ms) than fork if the forking program is a statically linked dietlibc program. For a small glibc program, the difference is 250 ms vs. 320 ms.

So which OS is the best one?

I recommend Linux 2.6. Linux scales O(1) in all benchmarks.

FreeBSD 5.1 is a close second. It is O(1) in all benchmarks except mmap.

Linux 2.4 scales badly for mmap and many processes, use 2.6 instead!

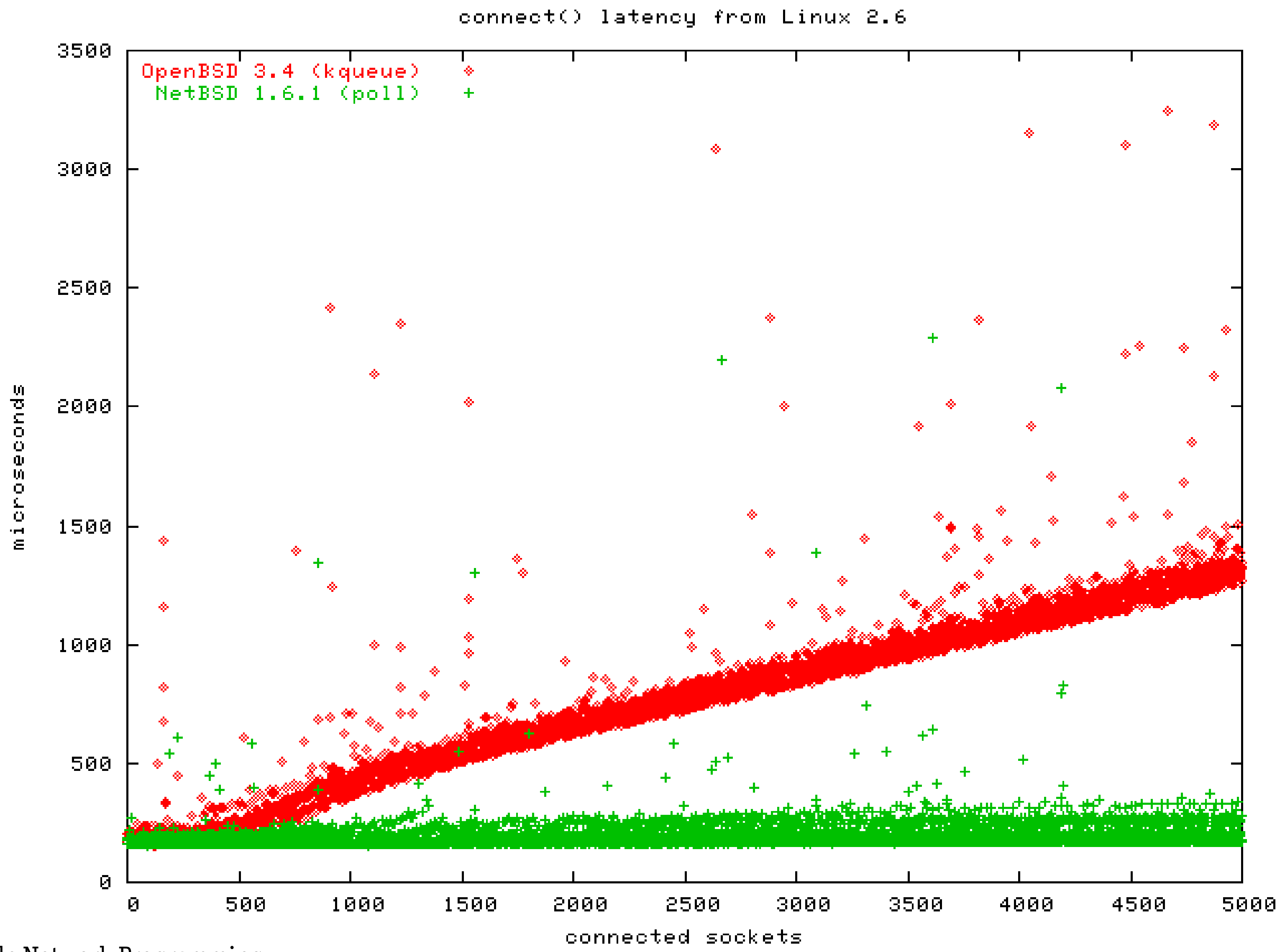
NetBSD does not have kqueue or sendfile, only poll. However, it is still a high performance operating system.

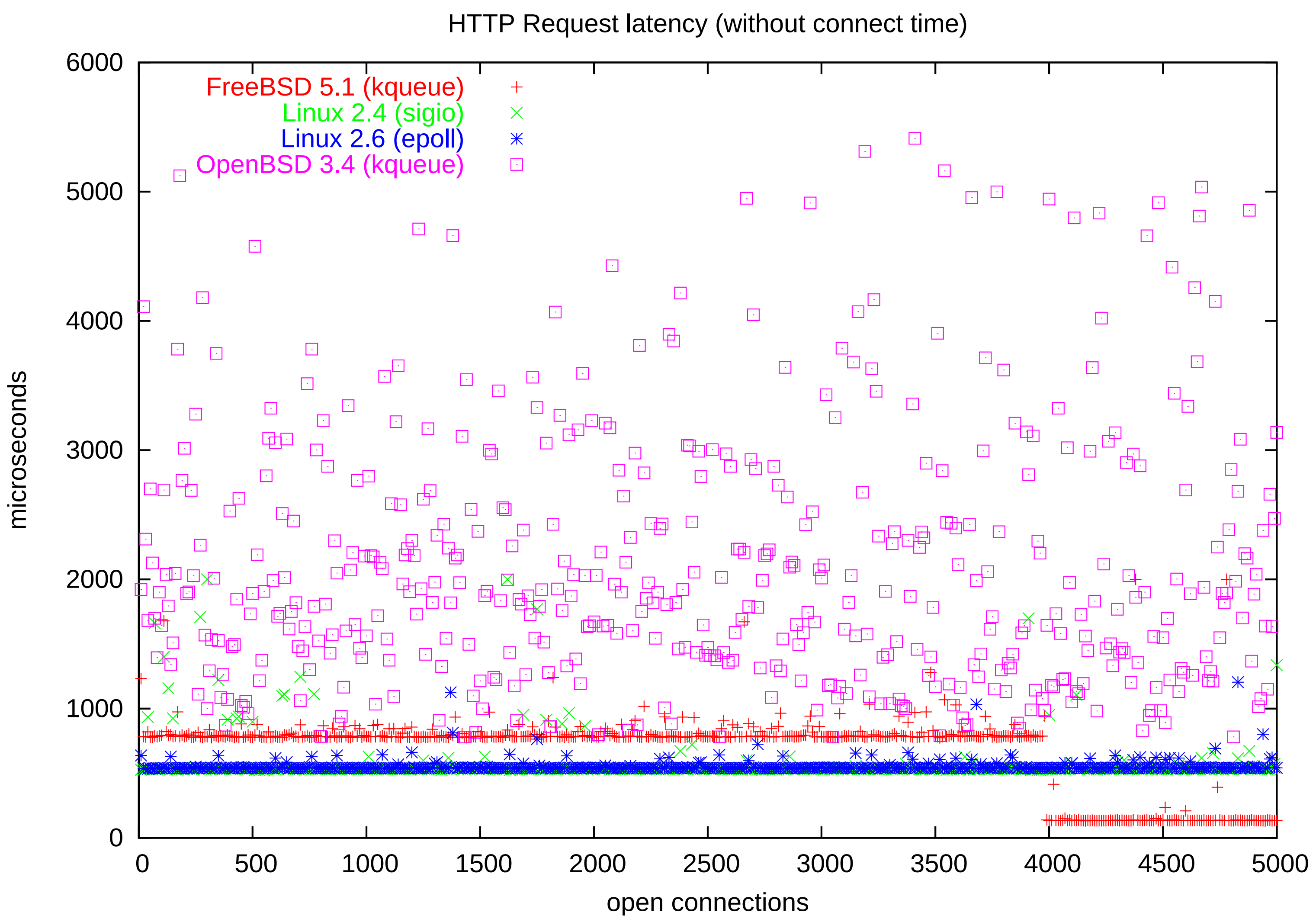
I found OpenBSD thoroughly disappointing. The disk performance sucks, OpenBSD can't even serve a single file with sustained 11 MB/sec on Fast Ethernet. And they deliberately broke their IPv6 stack.

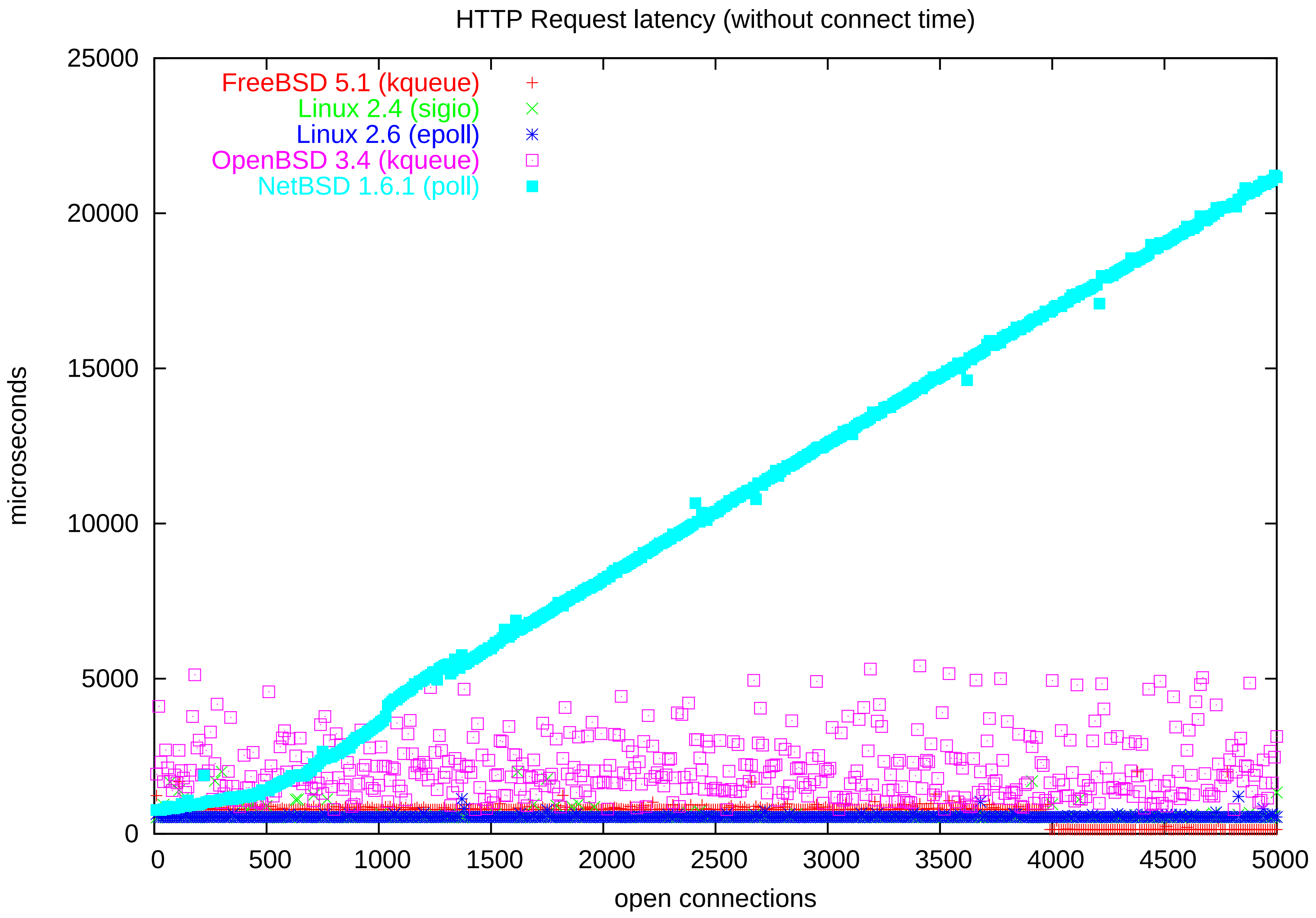
So which OS is the best one?

From a stability point of view, Linux and NetBSD worked stable all the time, FreeBSD 5.1-RELEASE panicked under load (that went away with 5.1-CURRENT) and OpenBSD crashed and panicked even in 3.4-CURRENT. OpenBSD also surprised me with ″interesting″ syslog messages like ″/bsd: full″.

Here are some more ″big picture″ graphs. I'll start with the most surprising result I encountered during my tests.


connect() latency from Linux 2.6
OpenBSD 3.4 (kqueue)
NetBSD 1.6.1 (poll)
microseconds
connected sockets
0
500
1000
1500
2000
2500
3000
3500
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


HTTP Request latency (without connect time)
microseconds
open connections
FreeBSD 5.1 (kqueue)
Linux 2.4 (sigio)
Linux 2.6 (epoll)
OpenBSD 3.4 (kqueue)
0
1000
2000
3000
4000
5000
6000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


HTTP Request latency (without connect time)
microseconds
open connections
FreeBSD 5.1 (kqueue)
Linux 2.4 (sigio)
Linux 2.6 (epoll)
OpenBSD 3.4 (kqueue)
NetBSD 1.6.1 (poll)
0
5000
10000
15000
20000
25000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

Questions

Thanks for sitting through this with me in this ungodly hour!

You can get the source code for all the benchmarks via anonymous cvs:

```
% cvs -d:pserver:cvs@cvs.fefe.de:/cvs -z9 co libowfat
% cvs -d:pserver:cvs@cvs.fefe.de:/cvs -z9 co gatling
```

My web page is at `http://www.fefe.de/`.

You can email me at `felix-linuxkongress@fefe.de`.