Using Currying and process-private system calls to break the one-microsecond system call barrier

Ronald G. Minnich[1] and John Floren and Jim McKie[2]

January, 2009

[1]Sandia National Labs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy National Nuclear Security Administration under contract DEÂAC04Â94AL85000. SAND- 2009-5156C.

[2]Bell Laboratories, Murray Hill, NJ, USA

The Problem

- As we saw, we can figure out where the time goes

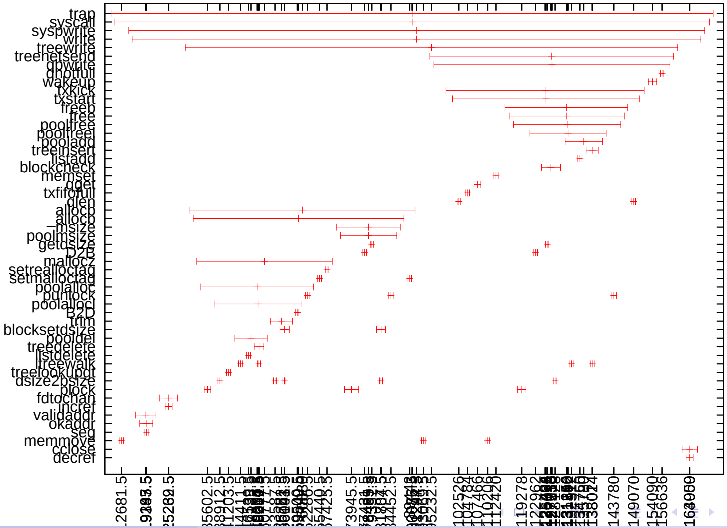
The Problem

- As we saw, we can figure out where the time goes
- And we can fix some of it

The Problem

- As we saw, we can figure out where the time goes
- And we can fix some of it
- But we can't get there just by hacking the kernel

Timing example to one fast network



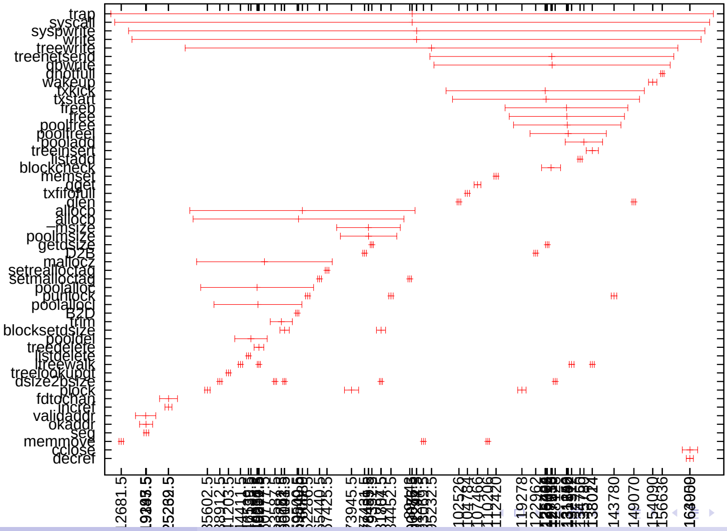
er, what?

- Drop into device write
- Allocate
- Push into the queue
- kick the tree
- send it out
- And all we really had to do was push it into a FIFO

So we killed the generality

- But it's still too slow
- don't seem to be able to get under 3-4 microseconds
- fdtochan
- Validaddr
- Sort of an irreducible problem
- Should we optimize these?

Timing example to one fast network



Should we optimize fdtochan and validaddr

- Cache the 32 or so addresses most programs use?
- Cache channels in the proc struct?
- We explored these options
- Makes code in port more complex
- End up with management of va cache in proc struct

Such micro-optimization is a common approach

- You see it all over, e.g., Linux
- Lots of heuristics in the code
- Which lead to creation of lots more timing-driven cleanup
- Gets really complex
- Can fail pathologically when programs do not follow common path
- Kernel is trying to intuit your intentions
- Got so crazy that Lustre just went ahead and added intents to ops
- I.e. when you do 'walk' you say 'and I intend to open this file after the walk'
- And all systems pay the costs of micro-optimization, even if only one subsystem needs it

Some numbers from Linux

- grep -r inline .—wc
- 69649 440053 6904168
- grep -r likely .—wc
- 18668 108045 1538433
- Note: includes likely and unlikely!
- e.g. `kernel/workqueue.c:  if (unlikely(cpu >= 0))`
- `kernel/workqueue.c:  while (unlikely(ret < 0));`
- grep -r heurist . — wc
- 197 2052 20268

HPC gave up long ago

- Around 1986, created interfaces that let programs write direct to network[1]
- These were later called *OS bypass*
- OS did not, could not, know when or how network I/O occurred
- Ubiquotous on top supercomputers
- Problem: Any program, to get good performance, has to be rewritten to use HPC networks
- HPC network libraries starting to look like OSes

But OS bypass is a trap

- Sure it works
- Just need to write an OS in your library
- But most interfaces run an OS anyway
- Bypass host OS, but not bypass the network OS
- Threading in support libraries continues to be a source of bugs
- And these libraries have complexity usually seen only in the OS
  - page colors
  - cache alignment
  - Playing games with interrupts

Plan A and B fail

- Optimize all we want, we can not get the OS faster
- Use OS bypass, end up recreating all the problems in the libraries
- and making the HPC interface unusable for non-HPC apps
- The price is too high to pay in either case
- That is where we were last summer
- Clearly in need of Plan C

One idea – Currying

- Useful technique
- $f(x, y) = y/x$
- if you know y, e.g. if y is 2
- Create $g(x)$ s.t. $g(x) = f(x, 2)$

Exploiting Currying in the kernel interface

- Leave information around about, e.g., past write behavior, past addresses
- take a chance that it will happen again
- Keep results fo validaddr, fdtochan, etc., around
- Add all the extra book-keeping needed to make it go
- Results from tomorrow's talk show that 32 addresses and a few cached channels might be enough
- need to clean them up when process exits

Currying alone is not enough

- We are still back to intuiting program behavior
- Kernel would have to look for patterns, cache arguments, etc.
- and it still might be wrong
- We got stuck on this problem last summer
- At some point, Currying just looks like more micro-optimizations

Another approach: Synthesis[2]

- Once a function and parameters are known
- Generate code *on the fly* to implement that code
- Can take this pretty far
- In fact, to the point that repeated reading of a file turns into repeated reading of a disk block
- Not a great general approach because code synthesis can have bugs
- and debugging is near-impossible
- Perhaps new techniques such as those found in vx32 might help

Create a struct which defines a fast path

```
struct Fastcall {
int scnum;
Chan *c;
long (*fun)(Chan*, void*, long, vlong);
void *buf;
int n;
vlong off;
};
```

- Holds all you need to know about a read/write system call
- a pointer to a function
- Also add an empty Fastcall * to proc struct
- Used as pointer to base of dynamically allocated array
- Add counter to proc struct for size

Modify syscall code

- There is a path in syscall to cover "syscall number too high"
- Simple change: if number too high, search array of Fastcall structs
- If found: `up->fc[i].fun(ar0, up->fc[i].c, up->fc[i].buf, up->fc[i].n, up->fc[i].off);`

Fast call setup function

- Find data chan given ctl file fd and incref it
- Do validaddr for (va, len)
- Other validation as needed for device (e.g. offset)
- Check for collisions in user-provided system call number
- Realloc up->fc struct, fill in new data
- Errors: read/write syscall errrors
- Other errors related to fastcall

Fast call function

- Passed a channel, data pointer, length, offset just as a normal call
- The overhead saved is in the generic read and write syscall
- Fastcall function can use none, some, or all of these
- E.g. on BG/P barrier, we only use the channel and the pointer
- Key is that these are validated
- void * always works because in Plan 9, *segments never shrink*
- Would be a nightmare in Unix

Special syscall code

```
TEXT mysyscall+0(SB),0,$8
MOVL a+0(FP),AX
BYTE $0xcd; BYTE $0x40
RET
```

- Not a lot there
- There's only one argument: the system call number

fastpath command

```
ctl = open(TEMP "/ctl", ORDWR);
cmd = smprint("fastwrite 256 %d 0x%p %d", fd0, data,
sizeof(data));
res = write(ctl, cmd, strlen(cmd));
```

- Must have syscall number and fd
- Typically there would a pointer and size
- And maybe an offset
- On success, invoke with `mysyscall(256);`

Blue Gene barrier network

- Normal path: too long
- More optimized path: 3.6 microseconds
- Fastcall: 770 nanoseconds
- Five-fold performance improvement
- With very little code change

But it's not a panacea

- We modified the pipe device
- Performance was no better and in some cases *worse*
- Why? queues are slow, slow, slow
- For slow interfaces, current interface is fine
- Shouldn't we fix those however?

Kernels have been stuck in a rut

- We know they're too slow
- People in HPC gave up years ago, moved to OS bypass
- Kernel path has just gotten slower
- App libraries have gotten more like kernels

Key contribution: private system calls + compile-time fastpath + run-time currying

- Private system calls support the fastpath function
- But could do much more; it is up to the driver
- Compile-time fastpath eliminates Synthesis issues with runtime code generation
- Run-time currying parameterizes compile-time fastpath
- Result, overall, is a new way to provide a fast path to programs
- Allows us to focus optimizations where needed, ignore them where not
- i.e. not all devices need this, so no need to modify them all

Conclusions

- We feel this can enable much higher I/O performance
- Need faster and pipe device
- Should be able to make fossil to venti to sdC path faster
- Could we have a fastpath through the network stack?

Next steps

- Make kernel-based comms on supercomputers run as well as OS-bypass-based comms
- On non-supercomputers, provide IPC that makes user-level-servers run as well as in-kernel servers
- No fastpath for read done yet
- Do not see that as a huge problem

Further reading

Stephen Menke, Mark Moir, and Srikanth Ramamurthy.
Synchronization mechanisms for scramnet+ systems.
In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 71–80, New York, NY, USA, 1998. ACM Press.

Calton Pu, Henry Massalin, and John Ioannidis.
The synthesis kernel.
Computing Systems, 1:11–32, 1988.