Measuring kernel throughput on Blue Gene/P with the Plan 9 research operating system

Ronald G. Minnich, John Floren, Aki Nyrhinen

Fourth International Workshop on Plan 9, 2009
Outline

1 Context

2 Where is the time going?
   - Related Work

3 devtrace

4 Using it

5 Summary
Porting Plan 9 to supercomputers
Because it's a clean, small system
Flexibility comes in at user level
One question of the DOE work: can we remove OS bypass if the kernel is fast enough?
Simulations said “maybe”
Using IBM SystemSim, boot Plan 9, and run a program that does a single write

- acid: 0x0119dd39 n = r;==>/9k/port/sysfile.c:790
- acid: 0x0119dd3a n = r;==>/k/port/sysfile.c:790
- acid: 0x0119dd3b o = ~0LL;==>9k/port/sysfile.c:792
- acid: 0x0119dd3c off = ~0LL;==>9k/port/sysfile.c:792 etc.

- About 600 ticks
- About 180 lines
- Seemed like it would be quite fast
But not as fast as we want

- On simulation we had thought the path from user to kernel to wire was fast
- Certainly faster than MPI libraries (or so the MPI guys told us)
- Measurement on real hardware showed it was actually slower than sim by too much
Example: global barrier driver

dcrput(p->set, 1); /* signal */

- That’s all there is to it
- Across 128K CPUs, this op takes about 125 ns.
- Other networks are similar
- HPC approach: just let programs do it directly
- Our approach: go through a fast kernel
- But it was not fast enough ... took a significantly longer time
- What’s an acceptable time? Has to be well under 1 microsecond
Where is all the time going?

- Did not have a way to trace, function-by-function, where time was spent, and who called whom
- Can do profiling but that is really a “fraction of time spent”
- Hard to see relationships between events
- Profiling is a histogram tool
We would rather see

- Who calls whom
- What fraction of time I spend in “x” before I call “y”
- Not just “how much time spent in “x” and “y”
- Need to see relationships and ordering of calls
Other work

- dtrace[1], dkm, neat hardware hacks[3], kprobes[2], djprobes, jprobes[4], kernel markers, ftrace[http://lwn.net/Articles/270971/]
- First time I saw it was on SunOS ca. 1988, which used a kernel markers like approach
- Kernel markers are a lot of work, requires annotating thousands of points to really get coverage
- My reading: Linux community may find function tracing is “good enough” most of the time (see: ftrace)
- MacOS, however, has adopted dtrace, which is extremely powerful
- dtrace has two modes: enable always-compiled-in function traces, or:
- Rewrite running kernel binary for more complex tracing
Some tracing issues

- The obvious one: overhead
- Hence terms like “invasive” and “sample-based”
- The less obvious one: you just changed a kernel binary
- Was that safe?
- Do all the CPUs know?
- When is it safe to change it back? (answer: maybe never)
- SMP issues
Starting from earlier Dynamic Kernel Modifier work (2001?)

- Code rewriting at runtime
- Modify code by moving blocks and replacing them with jumps
- Given an address, rewrite the code at that address to jump to a “logger”
- If you know entry and exit addresses, you can trace a function
- Gets a little tricky if you don’t want to write an object-code-understander-relocater for CISC – and I don’t
- I only moved code known to be “safe” to move, i.e. register-register moves etc.
- See paper for details
- GCC function prologues are only a few specific types, so was easy – and you only need to move 5 bytes, around instruction boundaries
- 8c is not so gentle
DKM code inserted

Rewritten code with entry, exit modified

- jump to buffer
- function body
- jump to buffer

- Call user trigger code
- function entry
- Jump to body
- Call user function
- function exit
The fun bits

- Relocated code has to be position-independent
- There is stack fixup:
  - Have to maintain the stack correctly so calls from this function work
  - Have to ensure that the function, on exit, returns to the jump to your exit code
  - i.e. we don’t rewrite the exit code, we only rewrite the entry code
- It gets messy but it’s doable
- And it’s fun to disable gettimeofday() and watch how things slowly fall apart ...
Version 1 (Aki and Ron)

- Ron did an early cut based on the Dynamic Kernel Modifier work from 2001
- At IWP9 2, Aki adapted it to the Power PC on Jim’s desk
- We then further took it over to Blue Gene/P
IWP9 2 work

- Short form: on PPC it was pretty high overhead (although object-code-understander was not an issue)
- Worse, it required rewriting bits of the kernel memory image at run time
- Even worse, there is never a guarantee that you know when you can turn it off[4]
- Much less turn it on: are you sure that core 1 is not running code while you are busy rewriting it?
- You can’t ensure it by just making sure you write less than one cache line of code!
Version 2 goals

- Easily build into kernel
- Easy to control
- Can reliably turn tracing on and off
- No kernel rewrite
Devtrace in a nutshell

- **Plan 9 style text control**
  - Textual output
  - No kernel rewriting
  - Tracing on/off is always safe
  - Logic analyzer style interface
  - Not as powerful as dtrace
  - Not as informative as ftrace (I think?)
  - Ftrace info can be added
  - Could produce dtrace format data for dtrace function processing
Devtrace in a nutshell

- Plan 9 style text control
- Textual output
  - No kernel rewriting
  - Tracing on/off is always safe
  - Logic analyzer style interface
  - Not as powerful as dtrace
  - Not as informative as ftrace (I think?)
  - Ftrace info can be added
  - Could produce dtrace format data for dtrace function processing
Devtrace in a nutshell

- Plan 9 style text control
- Textual output
- No kernel rewriting
- Tracing on/off is always safe
- Logic analyzer style interface
- Not as powerful as dtrace
- Not as informative as ftrace (I think?)
- Ftrace info can be added
- Could produce dtrace format data for dtrace function processing

Ronald G. Minnich, John Floren, Aki Nyrhinen
Plan 9 trace device
Devtrace in a nutshell

- Plan 9 style text control
- Textual output
- No kernel rewriting
- Tracing on/off is always safe
- Logic analyzer style interface
- Not as powerful as dtrace
- Not as informative as ftrace (I think?)
- Ftrace info can be added
- Could produce dtrace format data for dtrace function processing
Devtrace in a nutshell

- Plan 9 style text control
- Textual output
- No kernel rewriting
- Tracing on/off is always safe
- Logic analyzer style interface
- Not as powerful as dtrace
- Not as informative as ftrace (I think?)
- Ftrace info can be added
- Could produce dtrace format data for dtrace function processing
We use -p infrastructure

- When you invoke `?l` with -p, functions look like this:
  
  ```
  0x00001020 CALL _profin(SB) f+0x5
  0x00001025 MOVL a+0x0(FP),AX f+0x9
  0x00001029 ADDL $0x5,AX f+0xc
  0x0000102c CALL _profout(SB) f+0x11
  0x00001031 RET
  ```

- `profin/out` give you arbitrary hooks
- Call sequence only lets you see the pc, no args
- Just gets a histogram, no time relationships
We just define our own profin

```
TEXT _profin(SB), 1, $0
TESTL probeactive(SB), AX
JZ inotready
MOVL 4(SP),AX
PUSHL AX
MOVL 4(SP),AX
PUSHL AX
CALL profin(SB)
POPL AX
POPL AX
POPL AX
inotready:   RET
```
and profout ...

TEXT _profout(SB), 1, $0
PUSHL AX
TESTL probeactive(SB), AX
JZ notready
MOVL 4(SP),AX
PUSHL AX
CALL profout(SB)
POPL AX
notready:  POPL AX
RET
A few details

- The stack frame already has some things you want
- Caller PC and some args
- Also, on some machines, one register has the “first parameter”
- Problem is to get them into a machine-independent format
- On x86, can trash ax on entry; must save it on return
- on PPC, must save it in both directions
- Finally, it’s important to disable tracing on certain functions
- Such as profin assembly and C code
- And anything the profin C code calls
Building into your kernel

- contrib/rminnich/9.probe
- Bind these directories over your /sys/src/9
- Note I left v1 code in there for your viewing pleasure
- mk ’CONF=pcprcpf’
- boot kernel and you’re ready to try it out
Trying it out

- See the 'probeit' file in 9.probe
  ```
  #!/bin/rc nm pc/9pcprcpf | grep $1 |
  awk '{print "probe 0x" $1 " new "$3}'
  > /dev/probectl
  ```
- That script will set up tracing for one symbol at entry
- Example in 'probeit' shows a real trace
- Probing syschdir and namec
- Showing arguments and so on
Output

- E or X
- PC
- Time in ticks
- PID
- Three args for E; return value for X
- Fixed, easy to parse format

E f01b626e 000000226c00e7d4 00009ed0 00000000 00000000 00000000 00000000
Viewing it

'plotme' using 1:2:3:ytic(5):xtic(1)

Ronald G. Minnich, John Floren, Aki Nyrhinen

Plan 9 trace device
What we learned

- We were able to measure where the time was spent
- There were some real time-wasters (inref/decref)
- There were some problems hard to see a way around (okaddr, fdtochan)
- We could go to elaborate and complex translation and other caching to try to shorten time
- But that seems the wrong path
devtrace can let you see where the time is going
Simple textual control and data interface
You can see relationships between calls
Exploits existing profiling architecture
Thanks to SP9SSS for help and advice
Can we make it less intrusive?

- Yes. And safer.
- Consider this code:
  
  ```
  a: JMP 2f
  call profin
  2: ...
  ```

- It becomes:
  
  ```
  EB05 a: jmp 2f
  E8F9FFFFFF call profin
  C3 ...
  ```

- So, actually, we can change one byte and enable/disable profiling on this function.
ret
call profout
ret

- Same deal: NOP and RET are same size, one byte
- So one-byte change can enable/disable profout in this function
- And it’s easy to find the code signature!
Have to modify 8l

- 8l builds “instructions” (prg()) as part of creating linked binary
- They form a linked list
- If profiling is enabled, 8l does a final-pass walk of list and inserts calls to profin/profout on function entry/exit
- How do you know what is entry/exit?
- prg() struct is marked as such
- So, given this least, need only modify how code is inserted
- In code shown below, we have the current entry/exit pointed to by ‘p’
Copy doprof() in obj.c to doprof2

q = prg();
q2 = prg();
q->line = p->line;  q->pc = p->pc;
q->link = p->link;  p->link = q2;
q2->link = q;
q2->line = p->line; q2->pc = p->pc;
q2->as = AJMP; q2->to.type = D_BRANCH;
q2->to.sym = p->to.sym; q2->pcond = q->link;
p = q;
p->as = ACALL;  p->to.type = D_BRANCH;
p->pcond = ps2;  p->to.sym = s2;
/* * RET */
qu = prg();
q->as = ARET; q->from = p->from; q->to = p->to; q->link = p->link; p->link = q;

/* * JAL
profout */
p->as = ACALL; p->from = zprg.from; p->to = zprg.to; p->to.type = D_BRANCH; p->pcond = ps4; p->to.sym = s4;
p = q;


Andrew McRae. Hardware profiling of kernels, or: How to look under the hood while the engine is running. 1993.

Satoshi Oshima. Djprobes status.