Linux Performance Analysis and Tools

Brendan Gregg
Lead Performance Engineer
brendan@joyent.com
@brendangregg

SCaLE11x
February, 2013
Find the Bottleneck

Operating System
- Applications
  - DBs, all server types, ...
- System Libraries
- System Call Interface
  - VFS
  - Sockets
  - TCP/UDP
  - IP
  - Ethernet
  - Scheduler
  - ZFS
  - Virtual Memory
- LVM
- Block Device Interface
- Device Drivers
- I/O Bus

Linux Kernel
- Device Drivers
- I/O Bridge

Hardware
- CPU Interconnect
- Memory Bus
- CPU 1
- DRAM
- I/O Controller
  - Disk
  - Disk
- Network Controller
  - Port
  - Port

Interface Transports

Sunday, February 24, 13
whoami

- Lead Performance Engineer
- Work/Research: tools, visualizations, methodologies
- Was Brendan@Sun Microsystems, Oracle, now Joyent
• High-Performance Cloud Infrastructure
  • Compete on cloud instance/OS performance
• Public/private cloud provider
• OS-Virtualization for bare metal performance (Zones)
• Core developers of SmartOS and node.js
• KVM for Linux guests
SCaLE10x: Cloud Performance Analysis

- Example perf issues, including new tools and visualizations:

http://dtrace.org/blogs/brendan/2012/01/30/performance-analysis-talk-at-scale10x/
SCaLE11x: Linux Performance Analysis

- The primary operating system for my next book: (secondary is the OpenSolaris-illumos-based SmartOS)
Agenda

• Background
• Linux Analysis and Tools
  • Basic
  • Intermediate
  • Advanced
• Methodologies
• Challenges
Performance

• Why do performance analysis?
  
  • **Reduce IT spend** – find and eliminate waste, find areas to tune, and do more with less
  
  • **Build scalable architectures** – understand system limits and develop around them
  
  • **Solve issues** – locate bottlenecks and latency outliers
Systems Performance

• Why study the operating system?
  • Find and fix kernel-based perf issues
    • 2-20% wins: I/O or buffer size tuning, NUMA config, etc
    • 2-200x wins: bugs, disabled features, perturbations causing latency outliers
  • Kernels change, new devices are added, workloads scale, and new perf issues are encountered.
• Analyze application perf from kernel/system context
  • 2-2000x wins: identifying and eliminating unnecessary work
Perspectives

- System analysis can be top-down, or bottom-up:
Kernel Internals

- Eventually you’ll need to know some kernel internals

Operating System

System Libraries

Applications
- DBs, all server types, ...

System Call Interface
- VFS
- Sockets
- ext3/...
- ZFS
- Scheduler
- TCP/UDP
- IP
- Virtual Memory
- Ethernet

Device Drivers
- Block Device Int.
- Linux Kernel

user-level
kernel-level
Common System Metrics

It’s also worth studying common system metrics (iostat, ...), even if you intend to use a monitoring product. Monitoring products often use the same metrics, read from /proc.
Analysis and Tools
Analysis and Tools

• A quick tour of tools, to show what can be done
• Then, some methodologies for applying them
Analysis and Tools

Operating System

Applications
DBs, all server types, ...

System Libraries

System Call Interface

VFS
Sockets
Scheduler

ext3/...
ZFS
TCP/UDP

LVM
IP
Virtual Memory

Block Device Interface
Ethernet

Device Drivers

I/O Bridge

I/O Controller
Disk
Disk

Network Controller
Port
Port
Analysis and Tools

Operating System

Applications
DBs, all server types, ...

System Libraries

System Call Interface

VFS
Sockets
Schedulers
TCP/UDP
IP
Ethernet
Virtual Memory

Device Drivers

I/O Bridge

I/O Controller

Network Controller

Disk

Port

Various:
sar
/proc

perf dtrace stap

perf

top

pidstat

mpstat
dstat

vmstat

slabtop
dstat

free
top

netstat

tcpdump ip
nicstat
dtrace

perf

Iostat

Iotop

Blktrace
dtrace

ping
Tools: Basic

• uptime
• top or htop
• mpstat
• iostat
• vmstat
• free
• ping
• nicstat
• dstat
uptime

• Shows load averages, which are also shown by other tools:

```
$ uptime
16:23:34 up 126 days,  1:03,  1 user,  load average: 5.09, 2.12, 1.82
```

• This counts runnable threads (tasks), on-CPU, or, runnable and waiting. Linux includes tasks blocked on disk I/O.

• These are exponentially-damped moving averages, with time constants of 1, 5 and 15 minutes. With three values you can see if load is increasing, steady, or decreasing.

• If the load is greater than the CPU count, it might mean the CPUs are saturated (100% utilized), and threads are suffering scheduler latency. Might. There’s that disk I/O factor too.

• This is only useful as a clue. Use other tools to investigate!
System-wide and per-process summaries:

$ top

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11721 web       20   0  623m  50m 4984 R   93  0.1   0:59.50 node
11715 web       20   0  619m  20m 4916 S   25  0.0   0:07.52 node
10 root      20   0     0    0    0 S    1  0.0 248:52.56 ksoftirqd/2
51 root      20   0     0    0    0 S    0  0.0   0:35.66 events/0
11724 admin     20   0 19412 1444 960 R  100 0.0   0:00.07 top
1 root      20   0 23772 1948 1296 S    0  0.0   0:04.35 init

%CPU = interval sum for all CPUs (varies on other OSes)

top can consume CPU (syscalls to read /proc)

Straight-forward. Or is it?
Interview questions:

1. Does it show all CPU consumers?
2. A process has high %CPU – next steps for analysis?
top, cont.

1. top can miss:
   - short-lived processes
   - kernel threads (tasks), unless included (see top options)

2. analyzing high CPU processes:
   - identify why – profile code path
   - identify what – execution or stall cycles

High %CPU time may be stall cycles on memory I/O – upgrading to faster CPUs doesn’t help!
htop

- Super top. Super configurable. Eg, basic CPU visualization:
mpstat

• Check for hot threads, unbalanced workloads:

```bash
$ mpstat -P ALL 1
02:47:49   CPU    %usr  %nice   %sys %iowait   %irq  %soft %steal %guest  %idle
02:47:50   all   54.37   0.00  33.12    0.00   0.00   0.00   0.00   0.00  12.50
02:47:50     0   22.00   0.00  57.00    0.00   0.00   0.00   0.00   0.00  21.00
02:47:50     1   19.00   0.00  65.00    0.00   0.00   0.00   0.00   0.00  16.00
02:47:50     2   24.00   0.00  52.00    0.00   0.00   0.00   0.00   0.00  24.00
02:47:50     3  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     4  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     5  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     6  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     7   16.00   0.00  63.00    0.00   0.00   0.00   0.00   0.00  21.00
02:47:50     8  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
[...]```

• Columns are summarized system-wide in top(1)’s header
• Disk I/O statistics. 1st output is summary since boot.

```
$ iostat -xkdz 1

Linux 2.6.35-32-server (prod21)       02/20/13     _x86_64_    (16 CPU)
Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s  
... 
sda               0.00     0.00    0.00    0.00     0.00     0.00  
... 
sdb               0.00     0.35    0.00    0.05     0.10     1.58  
... 
Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s  
... 
sdb               0.00     0.00  591.00    0.00  2364.00     0.00  
... 
```

workload input

```
... \   avgqu-sz   await r-await w-await  svctm  %util
... /     0.00  0.84   0.84    0.00   0.84   0.00
... \     0.00  3.82   3.47    3.86   0.30   0.00
... /     0.00  2.31   2.31    0.00   2.31   0.00
... \     0.95  1.61   1.61    0.00   1.61  95.00
```

resulting performance
iostat, cont.

• %util: usefulness depends on target – virtual devices backed by multiple disks may accept more work a 100% utilization
• Also calculate I/O controller stats by summing their devices
• One nit: would like to see disk errors too. Add a “-e”? 
vmstat

• Virtual-Memory statistics, and other high-level summaries:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r  b  swpd   free  buff  cache  si  so  bi  bo  in  cs  us  sy  id  wa
15  0  2852 46686812 279456 1401196  0  0  0  0  0  0  0  0  0  100  0
16  0  2852 46685192 279456 1401196  0  0  0  0 2136 36607 56 33 11  0
15  0  2852 46685952 279456 1401196  0  0  0  56 2150 36905 54 35 11  0
15  0  2852 46685960 279456 1401196  0  0  0  0 2173 36645 54 33 13  0
[...]```

• First line of output includes *some* summary-since-boot values
• “r” = total number of runnable threads, *including* those running
• Swapping (aka paging) allows over-subscription of main memory by swapping pages to disk, but costs performance
free

• Memory usage summary (Kbytes default):

$ free

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>used</th>
<th>free</th>
<th>shared</th>
<th>buffers</th>
<th>cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem:</td>
<td>49548744</td>
<td>32787912</td>
<td>16760832</td>
<td>0</td>
<td>61588</td>
<td>342696</td>
</tr>
<tr>
<td>-/+ buffers/cache:</td>
<td>32383628</td>
<td>17165116</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap:</td>
<td>100663292</td>
<td>0</td>
<td>100663292</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• buffers: block device I/O cache
• cached: virtual page cache
ping

• Simple network test (ICMP):

$ ping www.hilton.com
PING a831.b.akamai.net (63.234.226.9): 56 data bytes
64 bytes from 63.234.226.9: icmp_seq=0 ttl=56 time=737.737 ms
Request timeout for icmp_seq 1
64 bytes from 63.234.226.9: icmp_seq=2 ttl=56 time=819.457 ms
64 bytes from 63.234.226.9: icmp_seq=3 ttl=56 time=897.835 ms
64 bytes from 63.234.226.9: icmp_seq=4 ttl=56 time=669.052 ms
64 bytes from 63.234.226.9: icmp_seq=5 ttl=56 time=799.932 ms
^C
--- a831.b.akamai.net ping statistics ---
6 packets transmitted, 5 packets received, 16.7% packet loss
round-trip min/avg/max/stddev = 669.052/784.803/897.835/77.226 ms

• Used to measure network latency. Actually kernel <-> kernel IP stack latency, including how the network handles ICMP.

• Tells us some, but not a lot (above is an exception). Lots of other/better tools for this (eg, hping). Try using TCP.
nicstat

- Network statistics tool, ver 1.92 on Linux:

```
# nicstat -z 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Int</th>
<th>rKB/s</th>
<th>wKB/s</th>
<th>rPk/s</th>
<th>wPk/s</th>
<th>rAvs</th>
<th>wAvs</th>
<th>%Util</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:20:58</td>
<td>eth0</td>
<td>0.07</td>
<td>0.00</td>
<td>0.95</td>
<td>0.02</td>
<td>79.43</td>
<td>64.81</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>01:20:58</td>
<td>eth4</td>
<td>0.28</td>
<td>0.01</td>
<td>0.20</td>
<td>0.10</td>
<td>1451.3</td>
<td>80.11</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>01:20:58</td>
<td>vlan123</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>42.00</td>
<td>64.81</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>01:20:58</td>
<td>br0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>42.00</td>
<td>42.07</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Int</th>
<th>rKB/s</th>
<th>wKB/s</th>
<th>rPk/s</th>
<th>wPk/s</th>
<th>rAvs</th>
<th>wAvs</th>
<th>%Util</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:20:59</td>
<td>eth4</td>
<td>42376.0</td>
<td>974.5</td>
<td>28589.4</td>
<td>14002.1</td>
<td>1517.8</td>
<td>71.27</td>
<td>35.5</td>
<td>0.00</td>
</tr>
<tr>
<td>01:21:00</td>
<td>eth0</td>
<td>0.05</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>56.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>01:21:00</td>
<td>eth4</td>
<td>41834.7</td>
<td>977.9</td>
<td>28221.5</td>
<td>14058.3</td>
<td>1517.9</td>
<td>71.23</td>
<td>35.1</td>
<td>0.00</td>
</tr>
<tr>
<td>01:21:01</td>
<td>eth4</td>
<td>42017.9</td>
<td>979.0</td>
<td>28345.0</td>
<td>14073.0</td>
<td>1517.9</td>
<td>71.24</td>
<td>35.2</td>
<td>0.00</td>
</tr>
</tbody>
</table>
```

- This was the tool I wanted, and finally wrote it out of frustration (Tim Cook ported and enhanced it on Linux)

- Calculate network controller stats by summing interfaces
A better vmstat-like tool. Does coloring (FWIW).

```
# dstat 1
You did not select any stats, using -cdngy by default.
---total-cpu-usage--- -dsk/total- -net/total- ---paging--- ---system---
usr  sys  idl  wai  hiq  sig|  read  writ|  recv  send|  in  out|  int  csw
 0  0  100  0  0  0|  13  10k|  0  0|  158  968|  7  14
25  27  0  0  11  37|  0  0|  224 122|  0  0| 12333 1426
22  23  9  0  13  32|  0  53M| 19M| 143|  0|  508k|2037 1377
22  26  1  0  12  38|  0  208k|23M| 174|  0|  0|2425 1649
14  12  40  1  13  19|  0  36k|13M| 127|  0|  0|1164 1045
18  16  16  0  24 25M|4096| 16M| 18M| 265|  0|  0|1584 1822
13  14  47  0  6  21|  0  39M|13M| 105|  0|  0|1253 857
23  27  0  0  12  37|  0  0| 23M| 113|  0|  0|2248 1432
23  30  0  0  10  37|  0  20k|23M| 113|  0|  0|12305 1424
12  11  48  0  9  19|  0  16M|11M| 128|  0|  0|1133 959
19  19  17  0  15  31|  0  56M|18M| 189|  0|  0|1717 1388
 3  1  92  2  1  1|  1  428 |  0| 787k|5576| 24k|  0|136 216
 0  1  99  0  0  0|  0  0| 108k| 66|  0|  0|  8  9
```
Tools: Basic, recap

- uptime
- top or htop
- mpstat
- iostat
- vmstat
- free
- ping
- nicstat
- dstat
Tools: Basic, recap

Operating System

- Applications
  - DBs, all server types, ...

System Libraries

- System Call Interface
  - VFS
  - Sockets
  - Scheduler
  - ZFS
  - TCP/UDP
  - IP
  - Virtual Memory
  - Ethernet

Device Drivers

- Block Device Interface
  - Disk
  - Port

Hardware

- CPU
  - top
  - mpstat
  - dstat

- Virtual Memory
  - vmstat
  - dstat
  - free
  - top

- DRAM
  - nicstat

- I/O Bridge

- I/O Controller
  - Disk
  - Port

- Network Controller
  - Port

- System Call Interface
  - CPU
  - DRAM

- Device Drivers
  - Disk
  - Port

- Hardware
  - top
  - mpstat
  - dstat

- I/O Bridge
  - iostat
  - infer

- I/O Controller
  - Disk
  - ping

- Network Controller
  - Port
  - dstat

- System Call Interface
  - CPU
  - DRAM

- Device Drivers
  - Disk
  - Port

- Hardware
  - top
  - mpstat
  - dstat

- I/O Bridge
  - iostat
  - infer

- I/O Controller
  - Disk
  - ping

- Network Controller
  - Port
  - dstat
Tools: Intermediate

- sar
- netstat
- pidstat
- strace
- tcpdump
- blktrace
- iotop
- slabtop
- sysctl
- /proc
sar

- System Activity Reporter. Eg, paging statistics -B:

```
$ sar -B 1
Linux 3.2.6-3.fc16.x86_64 (node104) 02/20/2013 _x86_64_ (1 CPU)
05:24:34 PM  pgpgin/s pgpgout/s   fault/s  majflt/s  pgfree/s pgscank/s pgscand/s pgsteal/s  %vmeff
05:24:35 PM      0.00      0.00    267.68      0.00     29.29      0.00      0.00      0.00    0.00
05:24:36 PM     19.80      0.00    265.35      0.99     28.71      0.00      0.00      0.00    0.00
05:24:37 PM     12.12      0.00   1339.39      1.01   2763.64      0.00   1035.35   1035.35  100.00
05:24:38 PM      0.00      0.00    534.00      0.00     28.00      0.00      0.00      0.00    0.00
05:24:39 PM    220.00      0.00    644.00      3.00     74.00      0.00      0.00      0.00    0.00
05:24:40 PM   2206.06      0.00   6188.89     17.17   5222.22   2919.19      0.00   2919.19  100.00
[...]
```

- Configure to archive statistics from cron

- Many, many statistics available:
  - -d: block device statistics, -q: run queue statistics, ...

- Same statistics as shown by other tools (vmstat, iostat, ...)
netstat

* Various network protocol statistics using -s:

```
$ netstat -s
 [...]  
Tcp:  
  127116 active connections openings
  165223 passive connection openings
  12904 failed connection attempts
  19873 connection resets received
  20 connections established
  662889209 segments received
  354923419 segments send out
  405146 segments retransmitted
  6 bad segments received.
  26379 resets sent
 [...]  
TcpExt:
  2142 invalid SYN cookies received
  3350 resets received for embryonic SYN_RECV sockets
  7460 packets pruned from receive queue because of socket buffer overrun
  2932 ICMP packets dropped because they were out-of-window
  96670 TCP sockets finished time wait in fast timer
  86 time wait sockets recycled by time stamp
  1007 packets rejects in established connections because of timestamp
 [...] many...]
```
pidstat

- Very useful process breakdowns:

# pidstat 1
Linux 3.2.6-3.fc16.x86_64 (node107) 02/20/2013 _x86_64_ (1 CPU)

<table>
<thead>
<tr>
<th>Time</th>
<th>PID</th>
<th>%usr</th>
<th>%system</th>
<th>%guest</th>
<th>%CPU</th>
<th>CPU</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:55:18 PM</td>
<td>PID</td>
<td>%usr</td>
<td>%system</td>
<td>%guest</td>
<td>%CPU</td>
<td>CPU</td>
<td>Command</td>
</tr>
<tr>
<td>05:55:19 PM</td>
<td>12642</td>
<td>0.00</td>
<td>1.01</td>
<td>0.00</td>
<td>1.01</td>
<td>0</td>
<td>pidstat</td>
</tr>
<tr>
<td>05:55:20 PM</td>
<td>12643</td>
<td>15.05</td>
<td>11.11</td>
<td>0.00</td>
<td>16.16</td>
<td>0</td>
<td>cksum</td>
</tr>
</tbody>
</table>

# pidstat -d 1
Linux 3.2.6-3.fc16.x86_64 (node107) 02/20/2013 _x86_64_ (1 CPU)

<table>
<thead>
<tr>
<th>Time</th>
<th>PID</th>
<th>kB_rd/s</th>
<th>kB_wr/s</th>
<th>kB_ccwr/s</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:55:22 PM</td>
<td>PID</td>
<td>0.00</td>
<td>61.90</td>
<td>0.00</td>
<td>jbd2/vda2-8</td>
</tr>
<tr>
<td>05:55:23 PM</td>
<td>12643</td>
<td>151985.71</td>
<td>0.00</td>
<td>0.00</td>
<td>cksum</td>
</tr>
</tbody>
</table>

disk I/O (yay!)
strace

- System call tracer:

$ strace -tttT -p 12670
1361424797.229550 read(3, "REQUEST 1888 CID 2"..., 65536) = 959 <0.009214>
1361424797.239053 read(3, ",", 61440) = 0 <0.000017>
1361424797.239406 close(3) = 0 <0.000016>
1361424797.239738 munmap(0x7f8b22684000, 4096) = 0 <0.000023>
1361424797.240145 fstat(1, 
...]

- -ttt: microsecond timestamp since epoch (left column)
- -T: time spent in syscall (<seconds>)
- -p: PID to trace (or provide a command)
- Useful – high application latency often caused by resource I/O, and most resource I/O is performed by syscalls
strace, cont.

• -c: print summary:

```bash
# strace -c dd if=/dev/zero of=/dev/null bs=512 count=1024k
[...]
% time  seconds  usecs/call  calls  errors syscall
------  ---------  -----------  -------  -----------
51.32   0.028376   0           1048581  read
48.68   0.026911   0           1048579  write
0.00    0.000000   0           7       open
[...]
```

• This is also a (worst case) demo of the strace overhead:

```bash
# time dd if=/dev/zero of=/dev/null bs=512 count=1024k
[...]
536870912 bytes (537 MB) copied, 0.35226 s, 1.5 GB/s
real 0m0.355s
user 0m0.021s
sys 0m0.022s
# time strace -c dd if=/dev/zero of=/dev/null bs=512 count=1024k
[...]
536870912 bytes (537 MB) copied, 71.9565 s, 7.5 MB/s
real 1m11.969s
user 0m3.179s
sys 1m6.346s
```

200x slower
tcpdump

- Sniff network packets, dump to output files for post analysis:
  # tcpdump -i eth4 -w /tmp/out.tcpdump
  tcpdump: listening on eth4, link-type EN10MB (Ethernet), capture size 65535 bytes
  ^C33651 packets captured
  34160 packets received by filter
  508 packets dropped by kernel

  # tcpdump -nr /tmp/out.tcpdump
  reading from file /tmp/out.tcpdump, link-type EN10MB (Ethernet)
  06:24:43.908732 IP 10.2.0.2.55502 > 10.2.203.2.22: Flags [], ack ...
  06:24:43.908922 IP 10.2.0.2.55502 > 10.2.203.2.22: Flags [], ack ...
  06:24:43.908943 IP 10.2.203.2.22 > 10.2.0.2.55502: Flags [], seq ...
  06:24:43.909061 IP 10.2.0.2.55502 > 10.2.203.2.22: Flags [], ack ...

- Output has timestamps with microsecond resolution
- Study odd network latency packet-by-packet
- Import file into other tools (wireshark)
tcpdump, cont.

• Does have overhead in terms of CPU and storage; previous example dropped packets
  • Should be using socket ring buffers to reduce overhead
  • Can use filter expressions to also reduce overhead
  • Could still be problematic for busy interfaces
**blktrace**

- Block device I/O event tracing. Launch using btrace, eg:

  ```
  # btrace /dev/sdb
  8,16 3 1 0.429604145 20442 A R 184773879 + 8 <- (8,17) 184773816
  8,16 3 2 0.429604569 20442 Q R 184773879 + 8 [cksum]
  8,16 3 3 0.429606014 20442 G R 184773879 + 8 [cksum]
  8,16 3 4 0.429607624 20442 P N [cksum]
  8,16 3 5 0.429608804 20442 I R 184773879 + 8 [cksum]
  8,16 3 6 0.429610501 20442 U N [cksum] 1
  8,16 3 7 0.429611912 20442 D R 184773879 + 8 [cksum]
  8,16 1 1 0.440227144 0 C R 184773879 + 8 [0]
  [...]
  ```

- Above output shows a single disk I/O event. Action time is highlighted (seconds).

- Use for investigating I/O latency outliers
• Disk I/O by process:

```bash
# iotop -bod5
Total DISK READ: 35.38 M/s | Total DISK WRITE: 39.50 K/s
TID   PRIO  USER DISK READ DISK WRITE SWAPIN  IO COMMAND
12824 be/4 root  35.35 M/s  0.00 B/s  0.00 %  80.59 % cksum ...
279   be/3 root   0.00 B/s  27.65 K/s  0.00 %  2.21 % [jbd2/vda2-8]
12716 be/4 root  28.44 K/s  0.00 B/s  2.35 %  0.00 % sshd: root@pts/0
12816 be/4 root  6.32 K/s  0.00 B/s  0.89 %  0.00 % python /usr/bin/
iotop -bod5

[...]```

• IO: time thread was waiting on I/O (this is even more useful than pidstat’s Kbytes)

• Needs CONFIG_TASK_IO_ACCOUNTING or something similar enabled to work.
slabtop

- Kernel slab allocator usage top:

```
# slabtop -sc
Active / Total Objects (% used) : 900356 / 1072416 (84.0%)
Active / Total Slabs (% used)   : 29085 / 29085 (100.0%)
Active / Total Caches (% used)  : 68 / 91 (74.7%)
Active / Total Size (% used)    : 237067.98K / 260697.24K (90.9%)
Minimum / Average / Maximum Object: 0.01K / 0.24K / 10.09K
```

<table>
<thead>
<tr>
<th>OBJS</th>
<th>ACTIVE</th>
<th>USE</th>
<th>OBJ SIZE</th>
<th>SLABS</th>
<th>OBJ/SLAB</th>
<th>CACHE SIZE</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>112035</td>
<td>110974</td>
<td>99%</td>
<td>0.91K</td>
<td>3201</td>
<td>35</td>
<td>102432K</td>
<td>ext4_inode_cache</td>
</tr>
<tr>
<td>726660</td>
<td>579946</td>
<td>79%</td>
<td>0.11K</td>
<td>20185</td>
<td>36</td>
<td>80740K</td>
<td>buffer_head</td>
</tr>
<tr>
<td>4608</td>
<td>4463</td>
<td>96%</td>
<td>4.00K</td>
<td>576</td>
<td>8</td>
<td>18432K</td>
<td>kmalloc-4096</td>
</tr>
<tr>
<td>83496</td>
<td>76878</td>
<td>92%</td>
<td>0.19K</td>
<td>1988</td>
<td>42</td>
<td>15904K</td>
<td>dentry</td>
</tr>
<tr>
<td>23809</td>
<td>23693</td>
<td>99%</td>
<td>0.55K</td>
<td>821</td>
<td>29</td>
<td>13136K</td>
<td>radix_tree_node</td>
</tr>
<tr>
<td>11016</td>
<td>9559</td>
<td>86%</td>
<td>0.62K</td>
<td>216</td>
<td>51</td>
<td>6912K</td>
<td>proc_inode_cache</td>
</tr>
<tr>
<td>3488</td>
<td>2702</td>
<td>77%</td>
<td>1.00K</td>
<td>109</td>
<td>32</td>
<td>3488K</td>
<td>kmalloc-1024</td>
</tr>
<tr>
<td>510</td>
<td>431</td>
<td>84%</td>
<td>5.73K</td>
<td>102</td>
<td>5</td>
<td>3264K</td>
<td>task_struct</td>
</tr>
<tr>
<td>10948</td>
<td>9054</td>
<td>82%</td>
<td>0.17K</td>
<td>238</td>
<td>46</td>
<td>1904K</td>
<td>vm_area_struct</td>
</tr>
<tr>
<td>2585</td>
<td>1930</td>
<td>74%</td>
<td>0.58K</td>
<td>47</td>
<td>55</td>
<td>1504K</td>
<td>inode_cache</td>
</tr>
</tbody>
</table>

...
sysctl

- System settings:

```bash
# sysctl -a
[...]
net.ipv4.tcp_fack = 1
net.ipv4.tcp_reordering = 3
net.ipv4.tcp_ecn = 2
net.ipv4.tcp_dsack = 1
net.ipv4.tcp_mem = 24180 32240 48360
net.ipv4.tcp_wmem = 4096 16384 1031680
net.ipv4.tcp_rmem = 4096 87380 1031680
[...]
```

- Static performance tuning: check the config of the system
• Read statistic sources directly:

```bash
$ cat /proc/meminfo
MemTotal:  8181740 kB
MemFree: 71632 kB
Buffers: 163288 kB
Cached: 4518600 kB
SwapCached: 7036 kB
Active: 4765476 kB
Inactive: 2866016 kB
Active(anon): 2480336 kB
Inactive(anon): 478580 kB
Active(file): 2285140 kB
Inactive(file): 2387436 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 2932728 kB
SwapFree: 2799568 kB
Dirty: 76 kB
Writeback: 0 kB
[...]```

• Also see /proc/vmstat
Tools: Intermediate, recap.

- sar
- netstat
- pidstat
- strace
- tcpdump
- blktrace
- iotop
- slabtop
- sysctl
- /proc
Tools: Advanced

• perf
• DTrace
• SystemTap
• and more ...
• Originally Performance Counters for Linux (PCL), focusing on CPU performance counters (programmable registers)
• Now a collection of profiling and tracing tools, with numerous subcommands, including:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmem</td>
<td>Trace/measure kernel memory (slab) properties</td>
</tr>
<tr>
<td>kvm</td>
<td>Trace/measure KVM guest OS</td>
</tr>
<tr>
<td>list</td>
<td>List available events (targets of instrumentation)</td>
</tr>
<tr>
<td>lock</td>
<td>Analyze lock events</td>
</tr>
<tr>
<td>probe</td>
<td>Create dynamic probe points (dynamic tracing!)</td>
</tr>
<tr>
<td>record</td>
<td>Run a command and record profile data (as perf.data)</td>
</tr>
<tr>
<td>report</td>
<td>Read perf.data and summarize, has an interactive mode</td>
</tr>
<tr>
<td>sched</td>
<td>Trace/measure kernel scheduler statistics</td>
</tr>
<tr>
<td>stat</td>
<td>Run a command, gather, and report perf counter stats</td>
</tr>
</tbody>
</table>
**perf: Performance Counters**

- **Key performance counter summary:**

  ```bash
  $ perf stat gzip file1
  Performance counter stats for 'gzip file1':
  2294.924314  task-clock-msecs # 0.901 CPUs
  62  context-switches # 0.000 M/sec
  0  CPU-migrations # 0.000 M/sec
  265  page-faults # 0.000 M/sec
  5496871381  cycles # 2395.230 M/sec
  12210601948  instructions # 2.221 IPC
  1263678628  branches # 550.641 M/sec
  13037608  branch-misses # 1.032 %
  4725467  cache-references # 2.059 M/sec
  2779597  cache-misses # 1.211 M/sec
  2.546444859  seconds time elapsed
  ```

- Low IPC (<0.2) means stall cycles (likely memory); look for ways to reduce memory I/O, and improve locality (NUMA)
perf: Performance Counters, cont.

Can choose different counters:

```
$ perf list | grep Hardware
  cpu-cycles OR cycles                      [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend  [Hardware event]
  instructions                                [Hardware event]
  cache-references                            [Hardware event]

[...]
$ perf stat -e instructions,cycles,L1-dcache-load-misses,LLC-load-misses,dTLB-load-misses gzip file1

Performance counter stats for 'gzip file1':

  12278136571 instructions        #        2.199 IPC
    5582247352 cycles
    90367344 L1-dcache-load-misses
    1227085 LLC-load-misses
    685149 dTLB-load-misses

  2.3324925555 seconds time elapsed
```

Supports additional custom counters (in hex or a desc) for whatever the processor supports. Examine bus events.
perf: Performance Counters, cont.

advanced activity: refer to the processor manuals
perf: Profiling

- Profiling (sampling) CPU activity:
  
  ```bash
  # perf record -a -g -F 997 sleep 10
  [ perf record: Woken up 44 times to write data ]
  ```

- -a: all CPUs
- -g: call stacks
- -F: Hertz
- sleep 10: duration to sample (dummy command)
- Generates a perf.data file
- Can profile other hardware events too, with call stacks
perf: Profiling, cont.

- Reading perf.data, forcing non-interactive mode (--stdio):

```bash
# perf report --stdio
[...]
# Overhead          Command          Shared Object                          Symbol
#  .........  ............  .................  ..............................
#
72.98%       swapper  [kernel.kallsyms]  [k] native_safe_halt
    |-- native_safe_halt
    |    default_idle
    |    cpu_idle
    |    rest_init
    |    start_kernel
    |    x86_64_start_reservations
    |    x86_64_start_kernel

9.43%         dd  [kernel.kallsyms]  [k] acpi_pm_read
    |-- acpi_pm_read
    |    ktime_get_ts
    |    |--87.75%-- __delayacct_blkio_start
    |    |    io_schedule_timeout
    |    |    balance.dirty_pages_ratelimited_nr
    |    |    generic_file_buffered_write
[...]
```
perf: Profiling, cont.

• Flame Graphs support perf profiling data:

• Interactive SVG. Navigate to quantify and compare code paths
perf: Static Tracing

• Listing static tracepoints for block I/O:

```bash
$ perf list | grep block:
  block:block_rq_abort                           [Tracepoint event]
  block:block_rq_requeue                         [Tracepoint event]
  block:block_rq_complete                        [Tracepoint event]
  block:block_rq_insert                          [Tracepoint event]
  block:block_rq_issue                           [Tracepoint event]
  block:block_bio_bounce                         [Tracepoint event]
  block:block_bio_complete                       [Tracepoint event]
  block:block_bio_backmerge                      [Tracepoint event]
  block:block_bio_frontmerge                     [Tracepoint event]
  block:block_bio_queue                          [Tracepoint event]
  block:block_getrq                              [Tracepoint event]
  block:block_sleeprq                            [Tracepoint event]
  block:block_plug                               [Tracepoint event]
  block:block_unplug                             [Tracepoint event]
  block:block_split                              [Tracepoint event]
  block:block_bio_remap                          [Tracepoint event]
  block:block_rq_remap                           [Tracepoint event]
```

• Many useful probes already provided for kernel tracing:

```bash
$ perf list | grep Tracepoint | wc -l
840
```
## perf: Dynamic Tracing

- Define custom probes from kernel code; eg, tcp_sendmsg():

```
# perf probe --add='tcp_sendmsg'
Add new event:
    probe:tcp_sendmsg    (on tcp_sendmsg)
[...]

# perf record -e probe:tcp_sendmsg -aR -g sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.091 MB perf.data (~3972 samples) ]

# perf report --stdio
[...]
# Overhead  Command      Shared Object       Symbol
# ........  .......  .................  ...........
#
100.00%     sshd  [kernel.kallsyms]  [k] tcp_sendmsg
  | active traced call stacks from
  | arbitrary kernel locations!
```

```
--- tcp_sendmsg
    sock_aio_write
    do_sync_write
    vfs_write
    sys_write
    system_call
    __GI___libc_write
```
perf: Dynamic Tracing, cont.

Advanced activity: refer to the kernel source code.
perf: Dynamic Tracing, cont.

• Fills in kernel observability gaps
• Awesome capability
  • Takes some effort to use (waiting for the trace-dump-analyze cycle, and using post-processors to rework the output, or the post-scripting capability)
• Would be the awesomest tool ever, if it wasn’t for ...
DTrace

dtrace
DTrace

- *Programmable, real-time*, dynamic and static tracing
- Perf analysis and troubleshooting, without restarting anything
- Used on Solaris, illumos/SmartOS, Mac OS X, FreeBSD, ...
- Two ports in development for Linux (that we know of):
  - 1. dtrace4linux
    - Mostly by Paul Fox
  - 2. Oracle Enterprise Linux DTrace
    - Steady progress

There are a couple of awesome books about DTrace too
DTrace: Installation

• dtrace4linux version:

  1. https://github.com/dtrace4linux/dtrace
  2. README:

```bash
tools/get-deps.pl   # if using Ubuntu
tools/get-deps-fedora.sh  # RedHat/Fedora
make all
make install
make load           (need to be root or have sudo access)
```

```bash
# make load
tools/load.pl
13:40:14 Syncing...
13:40:14 Loading: build-3.2.6-3.fc16.x86_64/driver/dtracedrv.ko
13:40:15 Preparing symbols...
13:40:15 Probes available: 281887
13:40:18 Time: 4s
```

• WARNING: still a prototype, can panic/freeze kernels.
  I’m using it the lab to solve replicated production perf issues
DTrace: Programming

- Programming capabilities allow for powerful, efficient, one-liners and scripts. In-kernel custom filtering and aggregation.

```bash
# dtrace -n 'fbt::tcp_sendmsg:entry /execname == "sshd"/{
    @["bytes"] = quantize(arg3); }

dtrace: description 'fbt::tcp_sendmsg:entry' matched 1 probe

bytes

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1869</td>
</tr>
<tr>
<td>64</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1490</td>
</tr>
<tr>
<td>128</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>355</td>
</tr>
<tr>
<td>256</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>461</td>
</tr>
<tr>
<td>512</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>373</td>
</tr>
<tr>
<td>1024</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>95</td>
</tr>
<tr>
<td>2048</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>4</td>
</tr>
<tr>
<td>4096</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- Example shows `tcp_sendmsg()` size dist for “sshd” PIDs
Programming capabilities allow for powerful, efficient, one-liners and scripts. In-kernel custom filtering and aggregation.

```
# dtrace -n 'fbt::tcp_sendmsg:entry /execname == "sshd"/ {
    @"bytes" = quantize(arg3); }
```

```
dtrace: description 'fbt::tcp_sendmsg:entry ' matched 1 probe
^C
```

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1869</td>
</tr>
<tr>
<td>64</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1490</td>
</tr>
<tr>
<td>128</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>355</td>
</tr>
<tr>
<td>256</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>461</td>
</tr>
<tr>
<td>512</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>373</td>
</tr>
<tr>
<td>1024</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>95</td>
</tr>
<tr>
<td>2048</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>4</td>
</tr>
<tr>
<td>4096</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>0</td>
</tr>
</tbody>
</table>

Example shows tcp_sendmsg() size dist for “sshd” PIDs

these examples use dtrace4linux
**DTrace: Real-Time**

- Multiple GUIs use DTrace for real-time statistics. Eg, Joyent Cloud Analytics, showing real-time cloud-wide syscall latency:
DTrace, cont.

- Has advanced capabilities, but not necessarily difficult; You may just:
  - use one-liners (google “DTrace one-liners”)
  - use scripts (DTraceToolkit; DTrace book; google)
  - tweak one-liners or scripts a little
  - ask someone else to write the scripts you need
- Ideally, you learn DTrace and write your own
DTrace: Scripts

```c
#!/usr/sbin/dtrace -s

fbt::vfs_read:entry
{
    self->start = timestamp;
}

fbt::vfs_read:return
/self->start/
{
    @[execname, "ns"] = quantize(timestamp - self->start);
    self->start = 0;
}
```

13 line script to time VFS reads by process name

```bash
# ./vfsread.d
dtrace: script './vfsread.d' matched 2 probes

cksum
    value  ------------------ Distribution ------------------ count

[...]

    262144 | 0
    524288 |@@@@@@@@@@ 834
    1048576 | 8
    2097152 | 30
    4194304 | 40
    8388608 |@ 66
   16777216 | 28
   33554432 | 1
```

read latency distribution,
0.5ms -> 33ms (disks)
DTrace: Basics

• CLI syntax:

\[
dtrace -n 'provider:module:function:name /predicate/ \{ action \}'
\]

\[
\text{probe description} \quad \text{optional} \quad \text{filter} \quad \text{do this when probe “fires”}
\]

• provider – library of related probes
• module:function – shows where probe is located (for debug)
• name – name of probe

• Online reference and tutorial: http://dtrace.org/guide
DTrace: Providers

- Applications
  - DBs, all server types, ...
- System Libraries
  - System Call Interface
    - VFS
    - ext3/...
    - ZFS
    - LVM
- Block Device Interface
  - VFS
  - Sockets
  - Scheduler
  - TCP/UDP
  - IP
  - Virtual Memory
  - Ethernet
- Device Drivers
  - I/O Bridge
    - I/O Controller
      - Disk
      - Disk
    - Network Controller
      - Port
      - Port
- CPU 1
- DRAM

Sunday, February 24, 13
DTrace: Providers

- syscall
- plockstat
- tcp
- udp
- ip
- Applications
- DBs, all server types, ...
- System Libraries
- System Call Interface
- VFS
- Sockets
- Scheduler
- TCP/UDP
- IP
- Virtual Memory
- Ethernet
- Block Device Interface
- Device Drivers
- I/O Bridge
- I/O Controller
- Disk
- Disk
- Network Controller
- Port
- Port
- CPU
- DRAM
- sched
- proc
- profile
- cpc
- vminfo
- java
- javascript
- node
- perl
- php
- ruby
- erlang
- objc
- tcl
- ...
- mysql
- postgres
- ...
- fbt
- pid
- profile
fbt and pid are dynamic

Sunday, February 24, 13
DTrace: Linux Examples

• Following examples use fbt – kernel dynamic tracing
DTrace: ext4slower.d

- Show me:
  - ext4 reads and writes
  - slower than a specified latency (milliseconds)
  - with time, process, direction, size, latency, and file name

```plaintext
# ./ext4slower.d 10
Tracing ext4 read/write slower than 10 ms

<table>
<thead>
<tr>
<th>TIME</th>
<th>PROCESS</th>
<th>D</th>
<th>KB</th>
<th>ms</th>
<th>FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013 Feb 22 17:17:02</td>
<td>cksum</td>
<td>R</td>
<td>64</td>
<td>35</td>
<td>100m</td>
</tr>
<tr>
<td>2013 Feb 22 17:17:02</td>
<td>cksum</td>
<td>R</td>
<td>64</td>
<td>16</td>
<td>1m</td>
</tr>
<tr>
<td>2013 Feb 22 17:17:03</td>
<td>cksum</td>
<td>R</td>
<td>64</td>
<td>18</td>
<td>data1</td>
</tr>
<tr>
<td>2013 Feb 22 17:17:03</td>
<td>cksum</td>
<td>R</td>
<td>64</td>
<td>23</td>
<td>data1</td>
</tr>
</tbody>
</table>
```

- I wrote this to answer: is ext4 to blame for latency outliers?
- Argument is latency you are looking for: here, 10+ ms
DTrace: ext4slower.d, cont.

• Extending vfs_read() example:

```c
#!/usr/sbin/dtrace -s
#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=5

dtrace:::BEGIN
{
    min_ns = $1 * 1000000;
    printf("Tracing ext4 read/write slower than %d ms\n", $1);
    printf("%-20s %-16s %s %4s %6s %s\n", "TIME", "PROCESS", "D", "KB", "ms", "FILE");
}

fbt::vfs_read:entry, fbt::vfs_write:entry
{
    this->file = (struct file *)arg0;
    this->fs = this->file->f_path.dentry->d_inode->i_sb->s_type->name;
}
```

• ... continued:
DTrace: ext4slower.d, cont.

```
fbt::vfs_read:entry, fbt::vfs_write:entry
/stringof(this->fs) == "ext4"/
{
    self->start = timestamp;
    self->name = this->file->f_path.dentry->d_name.name;
}

fbt::vfs_read:return, fbt::vfs_write:return
/self->start && (this->delta = timestamp - self->start) > min_ns/
{
    this->dir = probefunc == "vfs_read" ? "R" : "W";
    printf("%-20Y %-16s %ls %4d %6d %s\n", walltimestamp,
            execname, this->dir, arg1 / 1024, this->delta / 1000000,
            stringof(self->name));
}

fbt::vfs_read:return, fbt::vfs_write:return
{
    self->start = 0;
    self->name = 0;
}
```

- Immediately exonerate or blame ext4.
  ... should add more vfs_*() calls; or trace ext4 funcs directly
DTrace: tcpretransmit.d

• Show me:
  • TCP retransmits
  • destination IP address
  • kernel stack (shows why)
  • in real-time
• Don’t sniff all packets – *only* trace retransmits, to minimize overhead
DTrace: tcpretransmit.d, cont.

... can trace those stack functions directly for more detail
DTrace: tcpretransmit.d, cont.

- Source:

```c
#!/usr/sbin/dtrace -s
#pragma D option quiet
dtrace:::BEGIN { trace("Tracing TCP retransmits... Ctrl-C to end.\n"); }

fbt::tcp_retransmit_skb:entry {
    this->so = (struct sock *)arg0;
    this->d = (unsigned char *)&this->so->__sk_common.skc_daddr;
    printf("%Y: retransmit to %d.%d.%d.%d, by:", walltimestamp,
           this->d[0], this->d[1], this->d[2], this->d[3]);
    stack(99);
}
```
DTrace: Current State

• This was demoed on a prototype DTrace for Linux
  • Right now (Feb 2013) not stable – will panic/freeze
  • Needs other handholding to work around nits/bugs
  • AFAIK, both DTrace ports welcome help (that means you!)
• Those examples were also fbt-based:
  • Will probably need tweaks to match different kernels, since
    the API is dynamically built from the kernel code
  • DTrace stable providers solve that problem – but many
    aren’t there on Linux yet
DTrace: Trying it out

- All providers are available to try on illumos/SmartOS
  - illumos is the on-going fork of the OpenSolaris kernel
  - SmartOS is Joyent’s illumos-based cloud OS (distro)
- Rough translation guide:
  - `kernel: linux == illumos`
  - `distros: {ubuntu,CentOS,Fedora} == {SmartOS,OmniOS,OpenIndiana}`
- DTrace implementation mature
- Joyent uses SmartOS as a hypervisor for running KVM Linux on ZFS
DTrace: Other Capabilities

- Trace short lived processes
- Profile CPU usage
- Time any thread blocking event
- Investigate disk I/O latency
- Investigate network I/O latency
- Examine cache activity
- Investigate memory allocation: growth or leaks
- Investigate swapping (paging) in detail
- Follow network packets through the stack
- Examine lock contention
- ...

Sunday, February 24, 13
SystemTap

- Created when there wasn’t DTrace for Linux ports
- Static and dynamic tracing, probes, tapsets, scripts, ...
- I’ve used it a lot:
  - panics/freezes
  - slow startups
  - for Linux only
  - incompatible with D
Given the tools to see everything, how do you use them?
And More ...

- Other observability tools at all levels include:
  - ps, pmap, traceroute, ntop, ss, lsof, oprofile, gprof, kcachegrind, valgrind, google profiler, nfsiostat, cifsiostat, latencytop, powertop, LLTng, ktap, ...
- And many experimental tools: micro-benchmarks
- So many tools it gets confusing – where do you start?
Methodologies

- Selected four:
  - Streetlight Anti-Method
  - Workload Characterization Method
  - Drill-Down Analysis Method
  - USE Method

- Methodologies give beginners a starting point, casual users a checklist, and experts a reminder
Streetlight Anti-Method
Streetlight Anti-Method

1. Pick observability tools that are
   - familiar
   - found on the Internet
   - found at random
2. Run tools
3. Look for obvious issues

Included for comparison (don’t use this methodology)
Streetlight Anti-Method, cont.

- Named after an observational bias called the *streetlight effect*

A policeman sees a drunk looking under a streetlight, and asks what he is looking for. The drunk says he has lost his keys. The policeman can't find them either, and asks if he lost them under the streetlight. The drunk replies: “No, but this is where the light is best.”
Why are you *still* running `top`?
Streetlight Anti-Method, cont.

- Tools-based approach
- Inefficient:
  - can take time before the right tool is found
  - can be wasteful when investigating false positives
- Incomplete:
  - don’t find the right tool, or,
  - the right tool doesn’t exist
Workload Characterization Method
Workload Characterization Method

• 1. Who
• 2. Why
• 3. What
• 4. How
Workload Characterization Method

• 1. Who is causing the load? PID, UID, IP addr, ...
• 2. Why is the load called? code path
• 3. What is the load? IOPS, tput, direction, type
• 4. How is the load changing over time?
Workload Characterization Method, cont.

- Identifies issues of load
- Best performance wins are from *eliminating unnecessary work*
- Don’t assume you know what the workload is – characterize
- Many of the previous analysis tools included workload statistics
Workload Characterization Method, cont.

• Pros:
  • Potentially largest wins

• Cons:
  • Only solves a class of issues – load
  • Time consuming, and can be discouraging – most attributes examined will not be a problem
Drill-Down Analysis Method
Drill-Down Analysis Method

1. Start at highest level
2. Examine next-level details
3. Pick most interesting breakdown
4. If problem unsolved, go to 2
Drill-Down Analysis Method, cont.: Example

- For example, ext4 – identify latency origin top-down:

  Dynamic Tracing / DTrace is well suited for this, as it can dig through all layers with custom detail.
Drill-Down Analysis: ext4

- eg, ext4_readpages() latency distribution (microseconds):

```c
# dtrace -n 'fbt::ext4_readpages:entry { self->ts = timestamp; }
    fbt::ext4_readpages:return /self->ts/ {
        @["us"] = lquantize((timestamp - self->ts) / 1000, 0, 10000, 250);
        self->ts = 0;
    }'
dtrace: description 'fbt::ext4_readpages:entry ' matched 2 probes ^C

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>@@@@@@@@@@@@@</td>
<td>303</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>750</td>
<td>@@@@</td>
<td>88</td>
</tr>
<tr>
<td>1000</td>
<td>@@@@@@@@@@@@@@@@@@@@@@</td>
<td>335</td>
</tr>
<tr>
<td>1250</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1750</td>
<td>@@@@</td>
<td>107</td>
</tr>
<tr>
<td>2000</td>
<td>@@@@@@@</td>
<td>144</td>
</tr>
<tr>
<td>2250</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
[...]```

cache hits

disk I/O
Drill-Down Analysis: ext4

• ... can dig out more details as needed: file name, code path:

```
# dtrace -n 'fbt::ext4_readpages:entry {
    this->file = (struct file *)arg0;
    this->name = this->file->f_path.dentry->d_name.name;
    @[stringof(this->name), stack()] = count();
}
```

dtrace: description 'fbt::ext4_readpages:entry ' matched 1 probe

```
^C[...]
foo8
```

```
kernel`__do_page_cache_readahead+0x1c7
kernel`ra_submit+0x21
kernel`ondemand_readahead+0x115
kernel`page_cache_async_readahead+0x80
kernel`radix_tree_lookup_slot+0xe
kernel`find_get_page+0x1e
kernel`generic_file_aio_read+0x48b
kernel`vma_merge+0x121
kernel`do_sync_read+0xd2
kernel`__switch_to+0x132
kernel`security_file_permission+0x93
kernel`rw_verify_area+0x61
kernel`vfs_read+0xb0
kernel`sys_read+0x4a
kernel`system_call_fastpath+0x16
122
```
Drill-Down Analysis Method, cont.

- Moves from higher- to lower-level details based on findings: environment-wide down to metal
- Pros:
  - Will identify root cause(s)
- Cons:
  - Time consuming – especially when drilling in the wrong direction
USE Method
USE Method

• For every resource, check:
  • 1. Utilization
  • 2. Saturation
  • 3. Errors
USE Method, cont.

- For every resource, check:
  - 1. Utilization: time resource was busy, or degree used
  - 2. Saturation: degree of queued extra work
  - 3. Errors: any errors
USE Method, cont.

- Hardware Resources:
  - CPUs
  - Main Memory
  - Network Interfaces
  - Storage Devices
  - Controllers
  - Interconnects

- Find the *functional diagram* and examine every item in the *data path*...
For each check:
1. Utilization
2. Saturation
3. Errors
USE Method, cont.: Example Linux Checklist

http://dtrace.org/blogs/brendan/2012/03/07/the-use-method-linux-performance-checklist

<table>
<thead>
<tr>
<th>Resource</th>
<th>Type</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Utilization</td>
<td>per-cpu: mpstat -P ALL 1, “%idle”; sar -P ALL, “%idle”; system-wide: vmstat 1, “id”; sar -u, “%idle”; dstat -c, “idl”; per-process: top, “%CPU”; htop, “CPU %”; ps -o pcpu; pidstat 1, “%CPU”; per-kernel-thread: top/htop (“K” to toggle), where VIRT == 0</td>
</tr>
<tr>
<td>CPU</td>
<td>Saturation</td>
<td>system-wide: vmstat 1, “r” &gt; CPU count [2]; sar -q, “runq-sz” &gt; CPU count; dstat -p, “run” &gt; CPU count; per-process: /proc/PID/schedstat 2nd field (sched_info.run_delay); perf sched latency (shows “Average” and “Maximum” delay per-schedule); dynamic tracing, eg, SystemTap schedtimes.stp “queued(us)” [3]</td>
</tr>
<tr>
<td>CPU</td>
<td>Errors</td>
<td>perf (LPE) if processor specific error events (CPC) are available; eg, AMD64’s “04Ah Single-bit ECC Errors Recorded by Scrubber” [4]</td>
</tr>
</tbody>
</table>

... etc for all combinations (would fill a dozen slides)
Some software resources can also be studied:

- Mutex Locks
- Thread Pools
- Process/Thread Capacity
- File Descriptor Capacity

Consider possible USE metrics for each
USE Method, cont.

• This process may reveal *missing metrics* – those not provided by your current toolset
  • They are your *known unknowns*
  • Much better than *unknown unknowns*

• More tools can be installed and developed to help
  • So many top(1)s, but where is the *interconnect*-top?

• Full USE Method checklist may, practically, only be used for critical issues
USE Method, cont.

• Resource-based approach
• Quick system health check, early in an investigation
• Pros:
  • Complete: all resource bottlenecks and errors
  • Not limited in scope by your current toolset
  • No unknown unknowns – at least known unknowns
  • Efficient: picks three metrics for each resource – from what may be dozens available
• Cons:
  • Limited to a class of issues
Other Methodologies

• Include:
  • Blame-Someone-Else Anti-Method
  • Tools Method
  • Ad-Hoc Checklist Method
  • Problem Statement Method
  • Scientific Method
  • Latency Analysis
  • Stack Profile Method

Challenges

- Performance counter analysis (eg, bus or interconnect port analysis) is time consuming – would like tools for convenience
  - How about a “bustop” subcommand for perf?
- DTrace for Linux ports still in progress – will be awesome when complete
Cloud Computing

- Performance may be limited by cloud resource controls, rather than physical limits
- Hardware Virtualization complicates things – as a guest you can’t analyze down to metal directly
  - Hopefully the cloud provider provides an API for accessing physical statistics, or does the analysis on your behalf
- We do analysis at Joyent (and our hypervisors have DTrace!)
  - Free trial for new customers: good for $125 of usage value (~ one Small 1GB SmartMachine for 60 days). All prices subject to change. Limited time only. Sign up at joyent.com
References

- Linux man pages, source, /Documentation
- USE Method: http://queue.acm.org/detail.cfm?id=2413037
- Upcoming: “Systems Performance” (Prentice Hall)
Thank you!

• email: brendan@joyent.com
• twitter: @brendangregg
• blog: http://dtrace.org/blogs/brendan

• blog resources:
  • http://dtrace.org/blogs/brendan/tag/linux-2/
  • http://dtrace.org/blogs/brendan/2012/02/29/the-use-method/
  • http://dtrace.org/blogs/brendan/2012/03/07/the-use-method-linux-performance-checklist/
  • http://dtrace.org/blogs/brendan/2011/12/16/flame-graphs/
  • http://dtrace.org/blogs/brendan/2012/03/17/linux-kernel-performance-flame-graphs/
  • http://dtrace.org/blogs/brendan/2011/10/15/using-systemtap/