Performance Analysis

Brendan Gregg
Senior Performance Architect

NETFLIX

meet BSD
California 2014
BSD Observability
• FreeBSD for content delivery
  – Open Connect Appliances
  – Approx 33% of US Internet traffic at night

• AWS EC2 Linux cloud for interfaces
  – Tens of thousands of instances
  – CentOS and Ubuntu

• Performance is critical
  – Customer satisfaction: >50M subscribers
  – $$$ price/performance
Brendan Gregg

- Senior Performance Architect, Netflix
  - Linux and FreeBSD performance
  - Performance Engineering team (@coburnw)
- Recent work:
  - New Flame Graph types with pmcstat
  - DTrace tools for FreeBSD OCAs
- Previous work includes:
  - Solaris performance, DTrace, ZFS, methodologies, visualizations, findbill
A brief discussion of 5 facets of performance analysis on FreeBSD

1. Observability Tools
2. Methodologies
3. Benchmarking
4. Tracing
5. Counters
1. Observability Tools
How do you measure these?
FreeBSD Observability Tools
Observability Tools

• Observability tools are generally safe to use
  – Depends on their resource overhead

• The BSDs have awesome observability tools
  – DTrace, pmcstat, systat

• Apart from utility, an OS competitive advantage
  – Solve more perf issues instead of wearing losses

• Some examples...
One way to print load averages:

- CPU demand: runnable + running threads
  - Not confusing (like Linux and nr_uninterruptible)
- Exponentially-damped moving averages with time constants of 1, 5, and 15 minutes
  - Historic trend without the line graph
- Load > # of CPUs, may mean CPU saturation
  - Don’t spend more than 5 seconds studying these

$ uptime
7:07PM  up 18 days, 11:07, 1 user, load averages: 0.15, 0.26, 0.25
top

• Includes -P to show processors:

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>THR</th>
<th>PRI</th>
<th>NICE</th>
<th>SIZE</th>
<th>RES</th>
<th>STATE</th>
<th>C</th>
<th>TIME</th>
<th>WCPU</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1941</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>-4</td>
<td>55512K</td>
<td>26312K</td>
<td>kqread</td>
<td>9</td>
<td>512:43</td>
<td>4.98%</td>
<td>nginx</td>
</tr>
<tr>
<td>1930</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>-4</td>
<td>55512K</td>
<td>24000K</td>
<td>kqread</td>
<td>3</td>
<td>511:34</td>
<td>4.44%</td>
<td>nginx</td>
</tr>
<tr>
<td>1937</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>-4</td>
<td>51416K</td>
<td>22648K</td>
<td>kqread</td>
<td>4</td>
<td>510:32</td>
<td>4.35%</td>
<td>nginx</td>
</tr>
<tr>
<td>1937</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>-4</td>
<td>51416K</td>
<td>22648K</td>
<td>kqread</td>
<td>10</td>
<td>510:32</td>
<td>4.10%</td>
<td>nginx</td>
</tr>
</tbody>
</table>

• WCPU: weighted CPU, another decaying average
vimstat

- Virtual memory statistics and more:

```bash
$ vmstat 1

procs  memory  page  disks  faults  cpu
r b w  avm fre flt re pi po fr sr md0 md1 in sy cs us sy id
3 11 0  2444M 4025M   1106 0 1980 0 3188 899 0 0 294 5140 2198 2 25 73
0 11 0  2444M 3955M   30 0 2324 0 299543 105 0 0 75812 53510 397345 2 25 73
1 11 0  2444M 3836M   432 0 2373 0 295671 105 0 0 76689 53980 411422 2 24 74
0 11 0  2444M 3749M  19508 0 2382 0 308611 105 0 0 76586 56501 430339 3 26 71
0 11 0  2444M 3702M   28 0 2373 0 303591 105 0 0 75732 55629 403774 2 23 75
[..]
```

- USAGE: `vmstat [interval [count]]`
- First output line shows summary since boot
- High level system summary
  - scheduler run queue, memory, syscalls, CPU states
# iostat

- Storage device I/O statistics:

```plaintext
extended device statistics

<table>
<thead>
<tr>
<th>device</th>
<th>r/s</th>
<th>w/s</th>
<th>kr/s</th>
<th>kw/s</th>
<th>qlen</th>
<th>svc_t</th>
<th>%b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ada4</td>
<td>5.0</td>
<td>0.0</td>
<td>5087.8</td>
<td>0.0</td>
<td>0</td>
<td>3.8</td>
<td>2</td>
</tr>
<tr>
<td>da1</td>
<td>6.0</td>
<td>0.0</td>
<td>6105.3</td>
<td>0.0</td>
<td>0</td>
<td>7.7</td>
<td>3</td>
</tr>
<tr>
<td>da8</td>
<td>4.0</td>
<td>0.0</td>
<td>4070.2</td>
<td>0.0</td>
<td>0</td>
<td>1.9</td>
<td>1</td>
</tr>
<tr>
<td>da18</td>
<td>3.0</td>
<td>0.0</td>
<td>2098.7</td>
<td>0.0</td>
<td>0</td>
<td>7.4</td>
<td>2</td>
</tr>
<tr>
<td>da19</td>
<td>3.0</td>
<td>0.0</td>
<td>3052.7</td>
<td>0.0</td>
<td>0</td>
<td>1.9</td>
<td>1</td>
</tr>
<tr>
<td>da25</td>
<td>3.0</td>
<td>0.0</td>
<td>3052.7</td>
<td>0.0</td>
<td>0</td>
<td>1.9</td>
<td>1</td>
</tr>
<tr>
<td>da31</td>
<td>3.0</td>
<td>0.0</td>
<td>2989.1</td>
<td>0.0</td>
<td>0</td>
<td>5.3</td>
<td>2</td>
</tr>
</tbody>
</table>
```

- First output is summary since boot
- Excellent metric selection
- Wish it had -e for an error column
systat -ifstat

- Network interface throughput:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Traffic</th>
<th>Peak</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo0</td>
<td>0.000 KB/s</td>
<td>16.269 KB/s</td>
<td>2.314 GB</td>
</tr>
<tr>
<td>out</td>
<td>0.000 KB/s</td>
<td>16.269 KB/s</td>
<td>2.314 GB</td>
</tr>
<tr>
<td>cxl0</td>
<td>31.632 MB/s</td>
<td>31.632 MB/s</td>
<td>19.346 TB</td>
</tr>
<tr>
<td>out</td>
<td>800.456 MB/s</td>
<td>800.456 MB/s</td>
<td>786.230 TB</td>
</tr>
</tbody>
</table>

- systat is a multi-tool with other modes:
  - -tcp: TCP statistics
  - -iostat: storage I/O, with histogram
```plaintext
# systat -vmstat

Oct 30 19:57

1 users    Load  2.86  2.99  3.03

Mem:KB     REAL      VIRTUAL     VN  PAGER     SWAP    PAGER
         Tot  Share  Tot  Share  Free   in   out  in   out
Act  358036  9040  2443624  12360 2723532  count  2246
All 2408200  9576  3548292  46648     pages  306k

Proc:
   r  p  d  s   w  Csw  Trp  Sys  Int  Sof  Flt
10  65  400k  24k  56k  74k 3503  129

5.7%Sys  18.8%Intr  2.1%User  0.0%Nice  73.4%Idle

|    |    |    |    |    |    |    |    |    |           |
|    |    |    |    |    |    |    |    |    |  ioflt  88456 total
|    |    |    |    |    |    |    |    |    |  29  cow  uart2  10
|    |    |    |    |    |    |    |    |    |  92  zfod  1 ehci0  16
|    |    |    |    |    |    |    |    |    |  ozfod  2 ehci1  23
|    |    |    |    |    |    |    |    |    |  %ozfod  1129 cpu0:timer
|    |    |    |    |    |    |    |    |    |  daefr  1 igb0:que  0
|    |    |    |    |    |    |    |    |    |  5 prcfr  1 igb0:que  1
|    |    |    |    |    |    |    |    |    |  25 dtbuf
|    |    |    |    |    |    |    |    |    |  1 igb0:que  5
|    |    |    |    |    |    |    |    |    |  285817 totfr  1 igb0:que  2
|    |    |    |    |    |    |    |    |    |  react  1 igb0:que  3
|    |    |    |    |    |    |    |    |    |  pdwak  1 igb0:que  4
|    |    |    |    |    |    |    |    |    |  104 pdpgs  1 igb0:que  5
|    |    |    |    |    |    |    |    |    |  intrn  1 igb0:que  6
|    |    |    |    |    |    |    |    |    |  40558 frevn
|    |    |    |    |    |    |    |    |    |  70104 numvn
|    |    |    |    |    |    |    |    |    |  182004 182004 100

Namei     Name-cache  Dir-cache  2621440 desvn
Calls  hits  %    hits  %     70104 numvn
      182004  182004  100

Disks  md0  md1  md2  md3  ada0  adal  ada2
KB/t    0.00  0.00  0.00  0.00  564  546  520
tps     0   0   0   0  131  180  158
MB/s    0.00  0.00  0.00  0.00  72.14  95.77  80.11
%busy   0   0   0   0  17   22   32

[...]
```
# kldload dtrace
# dtrace -ln 'fbt:::entry'

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>fbt</td>
<td>kernel</td>
<td>camstatusentrycomp</td>
</tr>
<tr>
<td>6</td>
<td>fbt</td>
<td>kernel</td>
<td>cam_compatible_handle_0x17</td>
</tr>
<tr>
<td>8</td>
<td>fbt</td>
<td>kernel</td>
<td>cam_periph_done</td>
</tr>
<tr>
<td>9</td>
<td>fbt</td>
<td>kernel</td>
<td>camperiphdone</td>
</tr>
<tr>
<td>11</td>
<td>fbt</td>
<td>kernel</td>
<td>heap_down</td>
</tr>
<tr>
<td>13</td>
<td>fbt</td>
<td>kernel</td>
<td>cam_ccbq_remove_ccb</td>
</tr>
<tr>
<td>15</td>
<td>fbt</td>
<td>kernel</td>
<td>cam_module_event_handler</td>
</tr>
<tr>
<td>17</td>
<td>fbt</td>
<td>kernel</td>
<td>camisr_runqueue</td>
</tr>
<tr>
<td>19</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_alloc_device_default</td>
</tr>
<tr>
<td>21</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_async_process</td>
</tr>
<tr>
<td>22</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_async_process_dev</td>
</tr>
<tr>
<td>24</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_async_process_tgt</td>
</tr>
<tr>
<td>26</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_boot_delay</td>
</tr>
<tr>
<td>27</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_config</td>
</tr>
<tr>
<td>29</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_destroy_device</td>
</tr>
<tr>
<td>30</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_dev_async_default</td>
</tr>
<tr>
<td>32</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_done_process</td>
</tr>
<tr>
<td>34</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_done_td</td>
</tr>
<tr>
<td>35</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_finishconfig_task</td>
</tr>
<tr>
<td>37</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_modevent</td>
</tr>
<tr>
<td>39</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_periph_init</td>
</tr>
<tr>
<td>40</td>
<td>fbt</td>
<td>kernel</td>
<td>xpt_release_bus</td>
</tr>
</tbody>
</table>
```

[...28472 lines truncated...]

run all the things?
2. Methodologies
Methodologies & Tools

• Many awesome tools
  – Only awesome if you actually use them
  – The real problem becomes how to use them

• Methodologies can guide usage
Anti-Methodologies

• The lack of a deliberate methodology...

• Street Light Anti-Method:
  – 1. Pick observability tools that are
    • Familiar
    • Found on the Internet
    • Found at random
  – 2. Run tools
  – 3. Look for obvious issues

• Drunk Man Anti-Method:
  – Tune things at random until the problem goes away
Methodologies

• For example, the USE Method:
  – For every resource, check:
    • Utilization
    • Saturation
    • Errors
  
• 5 Whys: Ask “why?” 5 times

• Other methods include:
  – Workload characterization, drill-down analysis, event tracing, baseline stats, static performance tuning, ...

• Start with the questions, then find the tools
USE Method for Hardware

• For every resource, check:
  – Utilization
  – Saturation
  – Errors

• Including busses & interconnects
# USE Method: FreeBSD Performance Checklist

This page contains an example USE Method-based performance checklist for FreeBSD, for identifying common bottlenecks and errors. This is intended to be used early in a performance investigation, before moving onto more time consuming methodologies. This should be helpful for anyone using FreeBSD, especially system administrators.

This was developed on FreeBSD 10.0 alpha, and focuses on tools shipped by default. With DTrace, I was able to create a few new one-liners to answer some metrics. See the notes below the tables.

## Physical Resources


<table>
<thead>
<tr>
<th>component</th>
<th>type</th>
<th>metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>utilization</td>
<td>system-wide: <code>vmstat 1, &quot;us&quot; + &quot;sy&quot;</code>; per-cpu: <code>vmstat -P</code>; per-process: <code>top, &quot;WCPU&quot;</code> for weighted and recent usage; per-kernel-process: <code>top -s, &quot;WCPU&quot;</code></td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>system-wide: <code>uptime&quot;, &quot;load averages&quot; &gt; CPU count; </code>vmstat 1, &quot;procs&quot;: &gt; CPU count; per-cpu: DTrace to profile CPU run queue lengths [1]; per-process: DTrace of scheduler events [2]</td>
</tr>
<tr>
<td>CPU</td>
<td>errors</td>
<td><code>dmesg; /var/log/messages; pncstat</code> for PMC and whatever error counters are supported (eg, thermal throttling)</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>utilization</td>
<td>system-wide: <code>vmstat 1, &quot;fre&quot; is main memory free; </code>top, &quot;Mem&quot;:<code>; per-process: </code>top -o res, &quot;RES&quot;<code>is resident main memory size, &quot;SIZE&quot; is virtual memory size;</code>ps -auxw, &quot;RSS&quot;` is resident set size (Kbytes), &quot;VSZ&quot; is virtual memory size (Kbytes)</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>saturation</td>
<td>system-wide: <code>vmstat 1, &quot;sr&quot; for scan rate, &quot;w&quot; for swapped threads (was saturated, may not be now); </code>swapinfo, &quot;Capacity&quot; also for evidence of swapping/paging; per-process: DTrace [3]</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>errors</td>
<td>physical: <code>dmesg?; /var/log/messages?; virtual: DTrace failed malloc()</code></td>
</tr>
<tr>
<td>Network Interfaces</td>
<td>utilization</td>
<td>system-wide: <code>netstat -i 1</code>, assume one very busy interface and use input/output &quot;bytes&quot; / known max (note: includes localhost traffic); per-interface: <code>netstat -I interface 1</code>, input/output &quot;bytes&quot; / known max</td>
</tr>
<tr>
<td>Network Interfaces</td>
<td>saturation</td>
<td>system-wide: <code>netstat -s</code>, for saturation related metrics, eg `netstat -s</td>
</tr>
<tr>
<td>Network Interfaces</td>
<td>errors</td>
<td>system-wide: `netstat -s</td>
</tr>
<tr>
<td>Storage device I/O</td>
<td>utilization</td>
<td>system-wide: <code>iostat -xz 1, &quot;%b&quot;</code>; per-process: DTrace io provider, eg, iosnoop or iotop (DTR, needs porting)</td>
</tr>
<tr>
<td>Storage</td>
<td>saturation</td>
<td>system-wide: <code>iostat -xz 1, &quot;block&quot;:</code> DTrace for queue duration or length [4]</td>
</tr>
</tbody>
</table>
3. Benchmarking
~100% of benchmarks are wrong
The energy needed to refute benchmarks is multiple orders of magnitude bigger than to run them.
Benchmarking

• Apart from observational analysis, benchmarking is a useful form of experimental analysis
  – Try observational first; benchmarks can perturb

• However, benchmarking is error prone:
  – Testing the wrong target: eg, FS cache instead of disk
  – Choosing the wrong target: eg, disk instead of FS cache
    … doesn’t resemble real world usage
  – Invalid results: eg, bugs
  – Misleading results: you benchmark A, but actually measure B, and conclude you measured C

• FreeBSD has ministat for statistical analysis
Benchmark Examples

• Micro benchmarks:
  – File system maximum cached read operations/sec
  – Network maximum throughput

• Macro (application) benchmarks:
  – Simulated application maximum request rate

• Bad benchmarks:
  – gitpid() in a tight loop
  – Context switch timing
The Benchmark Paradox

• Benchmarking is used for product evaluations
  – Eg, evaluating a switch to BSD

• The Benchmark Paradox:
  – If your product’s chances of winning a benchmark are 50/50, you’ll usually lose

• Solving this seeming paradox (and benchmarking in general)...

For any given benchmark result, ask: why isn’t it 10x?
Active Benchmarking

• Root cause performance analysis while the benchmark is still running
  – Use the observability tools mentioned earlier
  – Identify the limiter (or suspected limiter) and include it with the benchmark results
  – Answer: why not 10x?

• This takes time, but uncovers most mistakes
4. Profiling
Profiling

• Can you do this?

“As an experiment to investigate the performance of the resulting TCP/IP implementation ... the [redacted] is CPU saturated, but the [redacted] has about 30% idle time. The time spent in the system processing the data is spread out among handling for the Ethernet (20%), IP packet processing (10%), TCP processing (30%), checksumming (25%), and user system call handling (15%), with no single part of the handling dominating the time in the system.”
Profiling

• Can you do this?

“As an experiment to investigate the performance of the resulting TCP/IP implementation ... the 11/750 is CPU saturated, but the 11/780 has about 30% idle time. The time spent in the system processing the data is spread out among handling for the Ethernet (20%), IP packet processing (10%), TCP processing (30%), checksumming (25%), and user system call handling (15%), with no single part of the handling dominating the time in the system.”

– Bill Joy, 1981, TCP-IP Digest, Vol 1 #6

Profiling Tools

• pmcstat
• DTrace
• Application specific products
pmcstat

• pmcstat counts PMC events, or records samples of kernel or user stacks
  – Eg, kernel stack every 64k L2 misses
• Performance monitoring counter (PMC) events
  – Low level CPU behavior: cycles, stalls, instructions, cache hits/misses
• FreeBSD has great PMC docs
  • eg, PMC.SANDYBRIDGE(3), PMC.IVYBRIDGE(3), ...
pmcstat Profiling

• Sampling stall cycles:

```bash
# pmcstat -S RESOURCESTALLS.ANY -O out.pmc sleep 10
# pmcstat -R out.pmc -z 32 -G out.stacks
```

CONVERSION STATISTICS:

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#exec/elf</td>
<td>25</td>
</tr>
<tr>
<td>#samples/total</td>
<td>107362</td>
</tr>
<tr>
<td>#samples/unknown-function</td>
<td>244</td>
</tr>
<tr>
<td>#callchain/dubious-frames</td>
<td>89</td>
</tr>
</tbody>
</table>

`# more out.stacks`
@ RESOURCESTALLS.ANY [16561 samples]

18.25% [3023] copyout @ /boot/kernel/kernel
99.93% [3021] soreceive_generic
100.0% [3021] kern_recvit
100.0% [3021] sys_recvfrom
100.0% [3021] amd64_syscall
00.07% [2] amd64_syscall

Can also emit gprof/Kcallgrind output

13.28% [2200] copyin @ /boot/kernel/kernel
100.0% [2200] ffs_write
100.0% [2200] VOP_WRITE_APV

[...]
PMC Counters

• Profile based on any counter:

```bash
# pmccontrol -L

[...]

branch-instruction-retired
branch-misses-retired
instruction-retired
llc-misses
llc-reference
unhalted-reference-cycles
unhalted-core-cycles
LD_BLOCKS.DATA_UNKNOWN
LD_BLOCKS.STORE_FORWARD
LD_BLOCKS.NO_SR
LD_BLOCKS.ALL_BLOCK
MISALIGN_MEM_REF.LOAD
MISALIGN_MEM_REF.STORE
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS
LD_BLOCKS_PARTIAL.ALL_STA_BLOCK
DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK
DTLB_LOAD_MISSES.WALK_COMPLETED
DTLB_LOAD_MISSES.WALK_DURATION

[...]
```

Beware of high frequency events, and use -n to limit samples.
 PMC Counter Groups

• Counters by group (eg, Intel Sandy Bridge):

| 1   | AGU  | 2   | LOAD | 4   | CPU  |
| 1   | ARITH| 2   | LONGEST | 4   | OTHER |
| 1   | BACLEARS | 2   | MISALIGN | 5   | ITLB  |
| 1   | HW   | 2   | OFF   | 6   | L1D   |
| 1   | ICACHE | 2   | SIMD   | 6   | LD    |
| 1   | INSTR | 2   | TLB    | 7   | IDQ   |
| 1   | INSTS | 2   | branch | 7   | OFFCORE |
| 1   | ROB  | 2   | llc    | 8   | DTLB  |
| 1   | RS   | 2   | unhalted | 10  | FP    |
| 1   | SQ   | 3   | CLOCK  | 12  | RESOURCE |
| 1   | instruction | 3   | CYCLE | 15  | UOPS |
| 2   | CPL  | 3   | DSB    | 22  | MEM   |
| 2   | DSB2MITE | 3   | LOCK  | 31  | BR    |
| 2   | ILD  | 3   | MACHINE | 37  | L2    |
| 2   | INST | 3   | PAGE   |               |
| 2   | INT  | 3   | PARTIAL |               |
How do you measure these?
PMC groups

eg, Intel Sandy Bridge
DTrace Profiling

• Kernel stack sampling at 199 Hertz, 60 s:

```bash
# kldload dtraceall       # if needed
# dtrace -x stackframes=100 -n 'profile-199 /arg0/ {
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks
```

• User stack sampling at 99 Hertz, 60 s:

```bash
# dtrace -x ustackframes=100 -n 'profile-99 /arg1/ {
    @[ustack()] = count(); } tick-60s { exit(0); }' -o out.stacks
```

• Warnings:
  – ustack() can be expensive
  – Short-lived processes will miss symbol translation

DEMO
Flame Graphs

• CPU flame graph (using DTrace):

```bash
# git clone https://github.com/brendangregg/FlameGraph
# cd FlameGraph
# kldload dtraceall    # if needed
# dtrace -x stackframes=100 -n 'profile-197 /arg0/ {
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks
# ./stackcollapse.pl out.stacks | ./flamegraph.pl > out.svg
```

• Stall cycle flame graph (using pmcstat):

```bash
...  
# pmcstat -S RESOURCE_STALLS.ANY -O out.pmcstat sleep 10
# pmcstat -R out.pmcstat -z100 -G out.stacks
# ./stackcollapse-pmc.pl out.stacks | ./flamegraph.pl > out.svg
```
5. Tracing
Tracing Tools

- truss
- tcpdump
- ktrace
- DTrace
DTrace

• Kernel and user-level tracing, programmatic
• Instruments probes provided by providers
• Stable interface providers:
  – io, ip, lockstat, proc, sched, tcp, udp, vfs
• Unstable interface providers:
  – pid: user-level dynamic tracing
  – fbt: (function boundary tracing) kernel dynamic tracing
  – syscall: system calls (maybe unstable)
• Providers should be developed/enhanced on BSD
Learning DTrace on FreeBSD

- https://wiki.freebsd.org/DTrace
- https://wiki.freebsd.org/DTrace/Tutorial
- https://wiki.freebsd.org/DTrace/One-Liners

- There’s also a good reference on how the kernel works, for when using kernel dynamic tracing:
Using DTrace

• Practical usage for most companies:
  – A) A performance team (or person)
    • Acquires useful one-liners & scripts
    • Develops custom one-liners & scripts
  – B) The rest of the company asks (A) for help
    • They need to know what’s possible, to know to ask
  – Or, you buy/develop a GUI that everyone can use
• There are some exceptions
  – Team of kernel/driver developers, who will all write custom scripts
# Trace file opens with process and filename:
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'

# Count system calls by program name:
dtrace -n 'syscall:::entry { @[execname] = count(); }'

# Count system calls by syscall:
dtrace -n 'syscall:::entry { @[probefunc] = count(); }'

# Count system calls by syscall, for PID 123 only:
dtrace -n 'syscall:::entry /pid == 123/ { @[probefunc] = count(); }'

# Count system calls by syscall, for all processes with a specific program name ("nginx"):
dtrace -n 'syscall:::entry /execname == "nginx"/ { @[probefunc] = count(); }'

# Count system calls by PID and program name:
dtrace -n 'syscall:::entry { @[pid, execname] = count(); }'

# Summarize requested read() sizes by program name, as power-of-2 distributions (bytes):
dtrace -n 'syscall:::read:entry { @[execname] = quantize(arg2); }'

# Summarize returned read() sizes by program name, as power-of-2 distributions (bytes or error):
dtrace -n 'syscall:::read:return { @[execname] = quantize(arg1); }'

# Summarize read() latency as a power-of-2 distribution by program name (ns):
dtrace -n 'syscall:::read:entry { self->ts = timestamp; } syscall:::read:return /self->ts/ {
    @[execname, "ns"] = quantize(timestamp - self->ts); self->ts = 0; }'

[...]

For more, see https://wiki.freebsd.org/DTrace/One-Liners
Brendan’s Scripts

DTraceToolkit

DTrace
Dynamic Tracing in Oracle Solaris,
Mac OS X, and FreeBSD

Brendan Gregg • Jim Mauro
Foreword by Bryan Cantrill
Brendan’s New FreeBSD Scripts (so far)

https://github.com/brendangregg/DTrace-tools
DEMO
Heat Maps

- Study latency distributions by-time:

[Image of Latency Heat Map]

https://github.com/brendangregg/HeatMap
Summary

A brief discussion of 5 facets of performance analysis on FreeBSD

1. Observability Tools
2. Methodologies
3. Benchmarking
4. Tracing
5. Counters
More Links

• FreeBSD @ Netflix:
  • https://openconnect.itp.netflix.com/
  • http://people.freebsd.org/~scottl/Netflix-BSDCan-20130515.pdf
  • http://www.youtube.com/watch?v=FL5U4wr86L4

• Flame Graphs:
  • http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  • http://www.brendangregg.com/blog/2014-10-31/cpi-flame-graphs.html

• USE Method FreeBSD:
  • http://www.brendangregg.com/USEmethod/use-freebsd.html

• FreeBSD Performance:
  • http://people.freebsd.org/~kris/scaling/Help_my_system_is_slow.pdf
  • https://wiki.freebsd.org/BenchmarkAdvice
  • http://www.brendangregg.com/activebenchmarking.html

• All the things meme:
  • http://hyperboleandahalf.blogspot.com/2010/06/this-is-why-ill-never-be-adult.html
Thanks

• Questions?
• http://slideshare.net/brendangregg
• http://www.brendangregg.com
• bgregg@netflix.com
• @brendangregg