EuroBSDcon 2017

System Performance Analysis Methodologies

Brendan Gregg
Senior Performance Architect
REGIONS WHERE NETFLIX IS AVAILABLE
Background
History

• System Performance Analysis up to the '90s:
  – Closed source UNIXes and applications
  – Vendor-created metrics and performance tools
  – Users interpret given metrics

• Problems
  – Vendors may not provide the best metrics
  – Often had to infer, rather than measure
  – Given metrics, what do we do with them?

$ ps -auxw
USER   PID %-CPU %-MEM  VSZ RSS TT STAT STARTED   TIME COMMAND
root 11 99.9  0.0     0 16  -  RL 22:10  22:27.05 [idle]
root  0  0.0  0.0     0 176  -  DLs 22:10  0:00.47 [kernel]
root  1  0.0  0.2  5408 1040  -  ILs 22:10  0:00.01 /sbin/init --
[...]

Today

1. Open source
   - Operating systems: Linux, BSD, etc.
   - Applications: source online (Github)

2. Custom metrics
   - Can patch the open source, or,
   - Use dynamic tracing (open source helps)

3. Methodologies
   - Start with the questions, then make metrics to answer them
   - Methodologies can pose the questions

Biggest problem with dynamic tracing has been what to do with it. Methodologies guide your usage.
Crystal Ball Thinking
Anti-Methodologies
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

- Drink Tune things at random until the problem goes away
Blame Someone Else *Anti*-Method

1. Find a system or environment component you are not responsible for
2. Hypothesize that the issue is with that component
3. Redirect the issue to the responsible team
4. When proven wrong, go to 1
Traffic Light *Anti*-Method

1. Turn all metrics into traffic lights
2. Open dashboard

- **Type I errors**: red instead of green
  - team wastes time
- **Type II errors**: green instead of red
  - performance issues undiagnosed
  - team wastes more time looking elsewhere

Traffic lights are suitable for *objective* metrics (eg, errors), not *subjective* metrics (eg, IOPS, latency).
Methodologies
For system engineers:
- ways to analyze unfamiliar systems and applications

For app developers:
- guidance for metric and dashboard design

Collect your own toolbox of methodologies
Problem Statement Method

1. What makes you **think** there is a performance problem?
2. Has this system **ever** performed well?
3. What has **changed** recently?
   - software? hardware? load?
4. Can the problem be described in terms of **latency**?
   - or run time. not IOPS or throughput.
5. Does the problem affect **other** people or apps?
6. What is the **environment**?
   - software, hardware, instance types? versions? config?
Functional Diagram Method

1. Draw the functional diagram
2. Trace all components in the data path
3. For each component, check performance

Breaks up a bigger problem into smaller, relevant parts

Eg, imagine throughput between the UCSB 360 and the UTAH PDP10 was slow...

ARPA Network 1969
Workload Analysis

• Begin with application metrics & context
• A **drill-down** methodology
• Pros:
  – Proportional, accurate metrics
  – App context
• Cons:
  – Difficult to dig from app to resource
  – App specific
Workload Characterization

• Check the workload, not resulting performance

• Eg, for CPUs:
  1. **Who**: which PIDs, programs, users
  2. **Why**: code paths, context
  3. **What**: CPU instructions, cycles
  4. **How**: changing over time
Workload Characterization: CPUs

**Who**
- `top`

**Why**
- CPU profile
- CPU flame graphs

**How**
- Monitoring

**What**
- PMCs
- CPI flame graph
Most companies and monitoring products today

**Who**

```
last pid: 4988; load averages: 0.55, 0.65, 0.58
27 processes: 3 running, 24
CPU: 95.7% user, 0.0% nice, 2.0% interrupt, 0.0% idle
Mem: 5988K Active, 41M Inact,
    2M Buf, 349M Free
Swap: 1824M Total, 1824M Free
```

**Why**

- CPU profile
- CPU flame graphs

**What**

- PMCs
- CPI flame graph

**How**

- monitoring

We can do better
Resource Analysis

• Typical approach for system performance analysis: begin with system tools & metrics

• Pros:
  – Generic
  – Aids resource perf tuning

• Cons:
  – Uneven coverage
  – False positives
The USE Method

• For every resource, check:
  1. **Utilization**: busy time
  2. **Saturation**: queue length or time
  3. **Errors**: easy to interpret (objective)

Starts with the questions, then finds the tools
Eg, for hardware, check every resource incl. busses:
# USE Method: Rosetta Stone of Performance Checklists

The following USE Method example checklists are automatically generated from the individual pages for: Linux, Solaris, Mac OS X, and FreeBSD. These analyze the performance of the physical host. You can customize this table using the checkboxes on the right.

There are some additional USE Method example checklists not included in this table: the SmartOS checklist, which is for use within an OS virtualized guest, and the Unix 7th Edition checklist for historical interest.

For general purpose operating system differences, see the Rosetta Stone for Unix, which was the inspiration for this page.

## Hardware Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Metric</th>
<th>Linux</th>
<th>FreeBSD</th>
<th>Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>errors</td>
<td>perf (LPE) if processor specific error events (CPC) are available; eg, AMD64's &quot;04Ah Single-bit ECC Errors Recorded by Scrubber&quot; [4]</td>
<td>dmesg; /var/log/messages; pmstat for PMC and whatever error counters are supported (eg, thermal throttling)</td>
<td>dmesg; /var/log/system.log; Instruments → Counters, for PMC and whatever error counters are supported (eg, thermal throttling)</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>system-wide: vnstat 1, &quot;r&quot; &gt; CPU count [2]; sar - q, &quot;runq-sz&quot; &gt; CPU count; dstat - p, &quot;run&quot; &gt; CPU count; per-process: /proc/PID/schedstat 2nd field (sched_info.run_delay); perf sched 1 latency (shows &quot;Average&quot; and &quot;Maximum&quot; delay per schedule); dynamic tracing, eg, SystemTap schedtimes.st &quot;queued(us)&quot; [3]</td>
<td>system-wide: uptime, &quot;load averages&quot; &gt; CPU count; vnstat 1, &quot;process&quot; &gt; CPU count; per-cpu: DTrace to profile CPU run queue lengths [1]; per-process: DTrace of scheduler events [2]</td>
<td>system-wide: uptime, &quot;load averages&quot; &gt; CPU count; latency, &quot;SCHEDULER&quot; and &quot;INTERRUPTS&quot;; per-cpu: dispgen.d (DTI), non-zero &quot;value&quot;; runnoc.d (DTI), non-zero &quot;%runnoc&quot;; per-process: Instruments → Thread States, &quot;On run queue&quot;; DTrace [2]</td>
</tr>
<tr>
<td>CPU</td>
<td>utilization</td>
<td>system-wide: vnstat 1, &quot;us&quot; + &quot;sy&quot; + &quot;st&quot;; sar - u, sum fields except &quot;%idle&quot; and &quot;%iowait&quot;; dstat - c, sum fields except &quot;iidl&quot; and &quot;waii&quot;; per-cpu: mpsstat -F ALL 1, sum fields except &quot;%idle&quot; and &quot;%iowait&quot;; sar - p ALL, same as mpsstat; per-process: top, &quot;%CPU&quot;; htop, &quot;CPU%&quot;; ps -o popcpu; pidstat 1, &quot;%CPU&quot;; per-kernel-thread: top/htop (&quot;K&quot; to toggle), where VIRT = 0 (heuristic) [1]</td>
<td>system-wide: vnstat 1, &quot;us&quot; + &quot;sy&quot;; per-cpu: mpsstat -p, per-process: top, &quot;WCPU&quot; for weighted and recent usage; per-kernel-process: top -s = &quot;WCPU&quot;</td>
<td>system-wide: iostat 1, &quot;us&quot; + &quot;sy&quot;; per-cpu: DTrace [1]; Activity Monitor → CPU Usage or Floating CPU Window; per-process: top -o cpu, &quot;%CPU&quot;; Activity Monitor → Activity Monitor, &quot;%CPU&quot;; per-kernel-thread: DTrace profile stack</td>
</tr>
<tr>
<td>CPU interconnect</td>
<td>errors</td>
<td>LPE (CPC) for whatever is available</td>
<td>pmstat and relevant PMCs for whatever is available</td>
<td>Instruments → Counters, and relevant PMCs for whatever is available</td>
</tr>
<tr>
<td>CPU interconnect</td>
<td>saturation</td>
<td>LPE (CPC) for stall cycles</td>
<td>pmstat and relevant PMCs for CPU interconnect stall cycles</td>
<td>Instruments → Counters, and relevant PMCs for stall cycles</td>
</tr>
</tbody>
</table>
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: vmstat 1, &quot;us&quot; + &quot;sy&quot;; per-cpu: vmstat -P; per-process: top, &quot;WCPU&quot; for weighted and recent usage; per-kernel-process: top -S, &quot;WCPU&quot;</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>system-wide: uptime, &quot;load averages&quot; &gt; CPU count; 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>dmesg; /var/log/messages; pmstat for PMC and whatever error counters are supported (eg, thermal throttling)</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>utilization</td>
<td>system-wide: vmstat 1, &quot;fre&quot; is main memory free; top, &quot;Mem&quot;; per-process: top -o res, &quot;RES&quot; is resident main memory size, &quot;SIZE&quot; is virtual memory size; 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: vmstat 1, &quot;sr&quot; for scan rate, &quot;w&quot; for swapped threads (was saturated, may not be now); 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: dmesg; /var/log/messages; virtual: DTrace failed malloc()s</td>
</tr>
</tbody>
</table>
USE Method: Unix 7th Edition Performance Checklist

Out of curiosity, I've developed a USE Method-based performance checklist for Unix 7th Edition on a PDP-11/45, which I've been running via a PDP simulator. 7th Edition is from 1979, and was the first Unix with iostat(1M) and pstat(1M), enabling more serious performance analysis from shipped tools. Were I to write a checklist for earlier Unixes, it would contain many more "unkowns".

I've worked on various Unix derivatives over the years, and it's been interesting to study this earlier version and see so many familiar areas.

Example screenshots from various tools are shown at the end of this page.

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: iostat 1, utilization is &quot;user&quot; + &quot;nice&quot; + &quot;sysm&quot;; per-process: ps aIx, &quot;CPU&quot; shows recent CPU usage (max 255), and &quot;TIME&quot; shows cumulative minutes:seconds of CPU time</td>
</tr>
<tr>
<td>CPU</td>
<td>saturation</td>
<td>ps aIx</td>
</tr>
<tr>
<td>CPU</td>
<td>errors</td>
<td>console message if lucky, otherwise panic</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>utilization</td>
<td>system-wide: unknown [1]; per-type: unknown [2]; per-process: ps aIx, &quot;SZ&quot; is the in-core (main memory) in blocks (512 bytes); pstat -p, &quot;SIZE&quot; is in-core size, in units of core clicks (64 bytes) and printed in octal!</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>saturation</td>
<td>system-wide: iostat 1, sustained &quot;ipm&quot; may be caused by swapping to disk; significant delays as processes wait for space to swap in</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>errors</td>
<td>malloc() returns 0; ENOMEM</td>
</tr>
</tbody>
</table>
USE Method: Software

• USE method can also work for software resources
  – kernel or app internals, cloud environments
  – small scale (eg, locks) to large scale (apps). Eg:

• Mutex locks:
  – utilization → lock hold time
  – saturation → lock contention
  – errors → any errors

• Entire application:
  – utilization → percentage of worker threads busy
  – saturation → length of queued work
  – errors → request errors
**RED Method**

- For every service, check these are within SLO/A:
  1. Request rate
  2. Error rate
  3. Duration (distribution)

Another exercise in posing questions from functional diagrams

By Tom Wilkie: http://www.slideshare.net/weaveworks/monitoring-microservices
Thread State Analysis

Identifier & quantify in states
Narrows further analysis to states
Thread states are applicable to all apps
TSA: eg, OS X

Instruments: Thread States
TSA: eg, RSTS/E

RSTS: DEC OS from the 1970's

TENEX (1969-72) also had Control-T for job states

<table>
<thead>
<tr>
<th>State Column (Job Status)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RN</strong></td>
</tr>
<tr>
<td><strong>RS</strong></td>
</tr>
<tr>
<td><strong>BF</strong></td>
</tr>
<tr>
<td><strong>SL</strong></td>
</tr>
<tr>
<td><strong>SR</strong></td>
</tr>
<tr>
<td><strong>FP</strong></td>
</tr>
<tr>
<td><strong>TT</strong></td>
</tr>
<tr>
<td><strong>HB</strong></td>
</tr>
<tr>
<td><strong>KB</strong></td>
</tr>
<tr>
<td><strong>^C</strong></td>
</tr>
<tr>
<td><strong>CR</strong></td>
</tr>
<tr>
<td><strong>MT, MM, or MS</strong></td>
</tr>
<tr>
<td><strong>LP</strong></td>
</tr>
<tr>
<td><strong>DT</strong></td>
</tr>
<tr>
<td><strong>DK, DM, DB, DP, DL, DR</strong></td>
</tr>
</tbody>
</table>
TSA: Finding FreeBSD Thread States

```c
# dtrace -ln sched:::

<table>
<thead>
<tr>
<th>ID</th>
<th>PROVIDER</th>
<th>MODULE</th>
<th>FUNCTION NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>56622</td>
<td>sched</td>
<td>kernel</td>
<td>preempt</td>
</tr>
<tr>
<td>56627</td>
<td>sched</td>
<td>kernel</td>
<td>dequeue</td>
</tr>
<tr>
<td>56628</td>
<td>sched</td>
<td>kernel</td>
<td>enqueue</td>
</tr>
<tr>
<td>56631</td>
<td>sched</td>
<td>kernel</td>
<td>off-cpu</td>
</tr>
<tr>
<td>56632</td>
<td>sched</td>
<td>kernel</td>
<td>on-cpu</td>
</tr>
<tr>
<td>56633</td>
<td>sched</td>
<td>kernel</td>
<td>remain-cpu</td>
</tr>
<tr>
<td>56634</td>
<td>sched</td>
<td>kernel</td>
<td>surrender</td>
</tr>
<tr>
<td>56640</td>
<td>sched</td>
<td>kernel</td>
<td>sleep</td>
</tr>
<tr>
<td>56641</td>
<td>sched</td>
<td>kernel</td>
<td>wakeup</td>
</tr>
</tbody>
</table>

```
# ./tstates.d

Tracing scheduler events... Ctrl-C to end.

^C

Time (ms) per state:

<table>
<thead>
<tr>
<th>COMM</th>
<th>PID</th>
<th>CPU</th>
<th>RUNQ</th>
<th>SLP</th>
<th>SUS</th>
<th>SWP</th>
<th>LCK</th>
<th>IWT</th>
<th>YLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq14: ata0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>irq15: ata1</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9009</td>
<td>0</td>
</tr>
<tr>
<td>swi4: clock (0)</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9761</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>usbus0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>8005</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

[...]

| sshd           | 807 | 0   | 0    | 10011| 0   | 0   | 0   | 0   | 0   |
| devd           | 474 | 0   | 0    | 9009 | 0   | 0   | 0   | 0   | 0   |
| dtrace         | 1166| 1   | 4    | 10006| 0   | 0   | 0   | 0   | 0   |
| sh             | 936 | 2   | 22   | 5648 | 0   | 0   | 0   | 0   | 0   |
| rand_harvestq  | 6   | 5   | 38   | 9889 | 0   | 0   | 0   | 0   | 0   |
| sh             | 1170| 9   | 0    | 0    | 0   | 0   | 0   | 0   | 0   |
| kernel         | 0   | 10  | 13   | 0    | 0   | 0   | 0   | 0   | 0   |
| sshd           | 935 | 14  | 22   | 5644 | 0   | 0   | 0   | 0   | 0   |
| intr           | 12  | 46  | 276  | 0    | 0   | 0   | 0   | 0   | 0   |
| cksum          | 1076| 929 | 28   | 0    | 480 | 0   | 0   | 0   | 0   |
| cksum          | 1170| 1499| 1029 | 0    | 0   | 0   | 0   | 0   | 0   |
| cksum          | 1169| 1590| 1144 | 0    | 0   | 0   | 0   | 0   | 0   |
| idle           | 11  | 5856 | 999 | 0   | 0   | 0   | 0   | 0   | 0   |

DTrace proof of concept

https://github.com/brendangregg/DTrace-tools/blob/master/sched/tstates.d
On-CPU Analysis

1. Split into user/kernel states
   – /proc, vmstat(1)
2. Check CPU balance
   – mpstat(1), CPU utilization heat map
3. Profile software
   – User & kernel stack sampling (as a **CPU flame graph**)
4. Profile cycles, caches, busses
   – PMCs, CPI flame graph
CPU Flame Graph Analysis

1. Take a CPU profile
2. Render it as a flame graph
3. Study largest "towers" first

Discovers issues by their CPU usage
- Directly: CPU consumers
- Indirectly: initialization of I/O, locks, times, ...

Narrows target of study
CPU Flame Graphs: FreeBSD

- Use either DTrace or pmcstat. Eg, kernel CPU with DTrace:

```bash
git clone https://github.com/brendangregg/FlameGraph; cd FlameGraph
dtrace -n 'profile-99 /arg0/ { @[stack()] = count(); } tick-30s { exit(0); }' > stacks01
stackcollapse.pl < stacks01 | sed 's/kernel`//g' | ./flamegraph.pl > stacks01.svg
```

- Both user & kernel CPU:

```bash
dtrace -x ustackframes=100 -x stackframes=100 -n '
profile-99 { @[stack(), ustack(), execname] = sum(1); }
tick-30s,END { printa("%k-%k%s\n@d\n", @); trunc(@); exit(0); }' > stacks02
```

[http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#DTrace](http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#DTrace)
Java Mixed-Mode CPU Flame Graph

By sampling stack traces with:

- `-XX:+PreserveFramePointer`
- `Java perf-map-agent`
A CPU flame graph (cycles) colored using instructions/stall profile data eg, using FreeBSD pmcstat:

red == instructions  
blue == stalls

Off-CPU Analysis

Analyze off-CPU time via blocking code path: **Off-CPU flame graph**

Often need *wakeup* code paths as well...
Off-CPU Time Flame Graph: FreeBSD

- Off-CPU Time
- Stack depth

- seek
- directory read
- file read
- readahead
- file read
- readahead

- missing symbols (stripped)

- Off-CPU time

- tar ... > /dev/null
# Off-CPU Profiling: FreeBSD

```bash
#!/usr/sbin/dtrace -s
#pragma D option ustackframes=100
#pragma D option dynvarsize=32m

sched:::off-cpu /execname == "bsdtar"/ { self->ts = timestamp; }

sched:::on-cpu /self->ts/
{
    @[stack()], ustack(), execname] = sum(timestamp - self->ts);
    self->ts = 0;
}

dtrace:::END
{
    normalize(@, 1000000);
    printa("%k-%k%s
%@d
", @);
}

# Uses DTrace

Warning: can have significant overhead (scheduler events can be frequent)

# .=./offcpu.d > out.stacks
# git clone https://github.com/brendangregg/FlameGraph; cd FlameGraph
# stackcollapse.pl < ../out.stacks | sed 's/kernel`//g' | 
    ./flamegraph.pl --color=io --title="Off-CPU Flame Graph" --countname=ms > out.svg
```
Off-CPU Time Flame Graph: FreeBSD

- file read
- readahead
- pipe write

Off-CPU Time Flame Graph

- mi_switch
- sleepq_wait
- sleep
- bwait
- breadn_flags
- ffs_read
- VOP_READ_APV
- vn_read
- vn_io_fault1
- vn_io_fault
- dofileread
- sys_read
- amd64_syscall
- 0x800c484a
- 0xfffff80ec392b
- 0x8008469ad
- 0x80087f9f
- 0x80087c5d7
- 0x800851d10
- 0x40a70c
- 0x40a583
- 0x40a3fb
- 0x409940e
- 0x40885b
- 0x4054f2
- 0x40453f
- 0x800632000
- bsd.tar

- tar ...
- | gzip
Wakeup Time Flame Graph: FreeBSD

Who did the wakeup:

- kernel-stack
- wakee
- user-stack
- waker
Wakeup Profiling: FreeBSD

#! /usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option usstackframes=100
#pragma D option dynvarsize=32m

sched:::sleep /execname == "bsdtar"/ { ts[curlwpsinfo->pr_addr] = timestamp; }

sched:::wakeup /ts[arg0]/ {
    this->delta = timestamp - ts[arg0];
    @[args[1]->p_comm, stack(), ustack(), execname] = sum(this->delta);
    ts[arg0] = 0;
}

dtrace:::END {
    normalize(@, 1000000);
    printa("\n%s%k-%%k%s%n%@d\n", @);
}

Warning: can have significant overhead (scheduler events can be frequent)
Merging Stacks with eBPF: Linux

- Using enhanced Berkeley Packet Filter (eBPF) to merge stacks in kernel context
- Not available on BSD (yet)
Ye Olde BPF

Berkeley Packet Filter

# tcpdump host 127.0.0.1 and port 22 -d
(000) ldh [12] Optimizes packet filter
(001) jeq #0x800 performance
(002) ld [26]
(003) jeq #0x7f000001
(004) ld [30]
(005) jeq #0x7f000001
(006) ldb [23] 2 x 32-bit registers
(007) jeq #0x84 & scratch memory
(008) jeq #0x6
(009) jeq #0x11
(010) ldh [20] User-defined bytecode
(011) jset #0x1fff executed by an in-kernel
(012) ldxb 4*([14]&0xf) sandboxed virtual machine
(013) ldh [x + 14] Steven McCanne and Van Jacobson, 1993

[...]
Enhanced BPF

aka eBPF or just "BPF"

```
struct bpf_insn prog[] = {
    BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
    BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto*/),
    BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
    BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
    BPF_LD_MAP_FD(BPF_REG_1, map_fd),
    BPF_RAWInsn(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
    BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
    BPF_RAWInsn(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
    BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
    BPF_EXIT_insn(),
};
```

10 x 64-bit registers
maps (hashes)
stack traces
actions

Alexei Starovoitov, 2014+
# load BPF program
b = BPF(text=""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
BPF_HISTOGRAM(dist);
int kprobe__blk_account_io_completion(struct pt_regs *ctx, 
    struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""
"
"
"
# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print

# output
b["dist"].print_log2_hist("kbytes")

bcc examples/tracing/bitehist.py
Latency Correlations

1. Measure latency histograms at different stack layers

2. Compare histograms to find latency origin

Even better, use latency heat maps

- Match outliers based on both latency and time
Checklists: eg, BSD Perf Analysis in 60s

1. uptime -------------------------------------------------> load averages
2. dmesg -a | tail ----------------------------------------> kernel errors
3. vmstat 1 ------------------------------------------> overall stats by time
4. vmstat -P ----------------------------------------> CPU balance
5. ps -auxw ------------------------------------------> process usage
6. iostat -xz 1 --------------------------------------> disk I/O
7. systat -ifstat ------------------------------------> network I/O
8. systat -netstat ------------------------------------> TCP stats
9. top -------------------------------------------------> process overview
10. systat -vmstat ------------------------------------> system overview

Checklists: eg, Netflix perfvitals Dashboard

1. RPS, CPU
2. Volume
3. Instances
4. Scaling
5. CPU/RPS
6. Load Avg
7. Java Heap
8. ParNew
9. Latency
10. 99th tile
Static Performance Tuning: FreeBSD

Operating System
- ps
- service -e
- sockstat
- ldd

System Libraries
- Applications

System Call Interface
- VFS
- UFS
- ZFS
- GEOM

Block Device Interface
- Device Drivers

FreeBSD Kernel
- sysctl kenv
- devinfo
- dmesg
- kldstat

Hardware
- cpuset -g
- vmstat [-moz]
- CPU
- Memory Bus
- DRAM

I/O Bus
- I/O Bridge
- ipf pfctl

I/O Controller
- Disk
- Disk
- Swap

Expander Interconnect
- mptutil

Network Controller
- Port
- Port

Device Interconnect
- camcontrol devlist
- swapinfo
- ifconfig

Power Supply
- FAN

Brendan Gregg 2017
Tools-Based Method: FreeBSD

Try all the tools!
May be an anti-pattern
Tools-Based Method: DTrace FreeBSD

Just my new BSD tools
Other Methodologies

• Scientific method
• 5 Why's
• Process of elimination
• Intel's Top-Down Methodology
• Method R
What You Can Do
What you can do

1. Know what's now possible on modern systems
   - Dynamic tracing: efficiently instrument any software
   - CPU facilities: PMCs, MSRs (model specific registers)
   - Visualizations: flame graphs, latency heat maps, ...

2. Ask questions first: use methodologies to ask them

3. Then find/build the metrics

4. Build or buy dashboards to support methodologies
Dynamic Tracing: Efficient Metrics

Eg, tracing TCP retransmits

**Old way:** packet capture

- **tcpdump**
  - 1. read
  - 2. dump

- **Analyzer**
  - 1. read
  - 2. process
  - 3. print

**New way:** dynamic tracing

- **Tracer**
  - 1. configure
  - 2. read

- **Kernel**
  - buffer
  - send
  - receive
  - file system
  - disks

- **tcp_retransmit_skb()**
Dynamic Tracing: Instrument Most Software

My Solaris/DTrace tools (many already work on BSD/DTrace):

cifs*.d, iscsi*.d, services
nfsv3*.d, nfsv4*.d
ssh*.d, httpd*.d
fsho.d, fssnoop.d
solife.d
solvfssnoop.d
dnlcsnoop.d
zfsslower.d
ziowait.d
ziostacks.d
spasync.d
metaslab_free.d
iosnoop, iotop
disklatency.d
satacmd.s
satalatency.d
scsicmds.d
scsilatency.d
sdretry.d, squeue.d
ide*.d, mpt*.d
hotuser, umutexmax.d, lib*.d
node*.d, erlang*.d, j*.d, js*.d
php*.d, pl*.d, py*.d, rb*.d, sh*.d
mysql*.d, postgres*.d, redis*.d, riak*.d
opensnoop, statsnoop
errinfo, dtruss, rwtop
rwsnoop, mmap.d, kill.d
shellsnoop, zonecalls.d
weblatency.d, fddist
priclass.d, pridist.d
cv_wakeup_slow.d
display.d, capslat.d
minfiber.d
pgpginbid.p.d
macops.d, ixgbecheck.d
ngesnoop.d, ngelink.d
soconnect.d, soaccept.d, soclose.d, socketio.d, solstbyte.d
sotop.d, soerror.d, ipstat.d, ipio.d, ipproto.d, ipfbsnoop.d
ipdropper.d, tcpstat.d, tcpaccept.d, tcpconnect.d, tcpioshort.d
tcpio.d, tcpbytes.d, tcpsize.d, tcpmmmap.d, tcpconnlat.d, tcplstbyte.d
tcpftbwatch.d, tcpsnnoop.d, tcpconnreqmaxq.d, tcprefused.d
tcpretranshosts.d, tcpretranssnoop.d, tcpsackretrans.d, tcpslowstart.d
tcptimewait.d, udpstat.d, udpio.d, icmpstat.d, icmmpsnoop.d
Performance Monitoring Counters

Eg, BSD PMC groups for Intel Sandy Bridge:
Visualizations

Eg, Disk I/O latency as a heat map, quantized in kernel:

Post processing the output of my iosnoop tool: [www.brendangregg.com/HeatMaps/latency.html](http://www.brendangregg.com/HeatMaps/latency.html)
Summary

• It is the crystal ball age of performance observability
• What matters is the questions you want answered
• Methodologies are a great way to pose questions
References & Resources

- FreeBSD @ Netflix:
  - https://openconnect.itp.netflix.com/
  - http://www.youtube.com/watch?v=FL5U4wr86L4
- USE Method
  - http://queue.acm.org/detail.cfm?id=2413037
- TSA Method
- Off-CPU Analysis
  - http://www.brendangregg.com/offcpuanalysis.html
  - http://www.brendangregg.com/blog/2016-02-05/ebpf-chaintagigraph-prototype.html
- Static Performance Tuning, Richard Elling, Sun blueprint, May 2000
- RED Method: http://www.slideshare.net/weaveworks/monitoring-microservices
- Other system methodologies
  - Systems Performance: Enterprise and the Cloud, Prentice Hall 2013
  - The Art of Computer Systems Performance Analysis, Jain, R., 1991
- Flame Graphs
  - http://queue.acm.org/detail.cfm?id=2927301
  - http://www.brendangregg.com/flamegraphs.html
- Latency Heat Maps
  - http://queue.acm.org/detail.cfm?id=1809426
- ARPA Network: http://www.computerhistory.org/internethistory/1960s
- DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD, Prentice Hall 2011
Thank You

- http://slideshare.net/brendangregg
- http://www.brendangregg.com
- bgregg@netflix.com
- @brendangregg