Nov, 2014

Performance Analysis

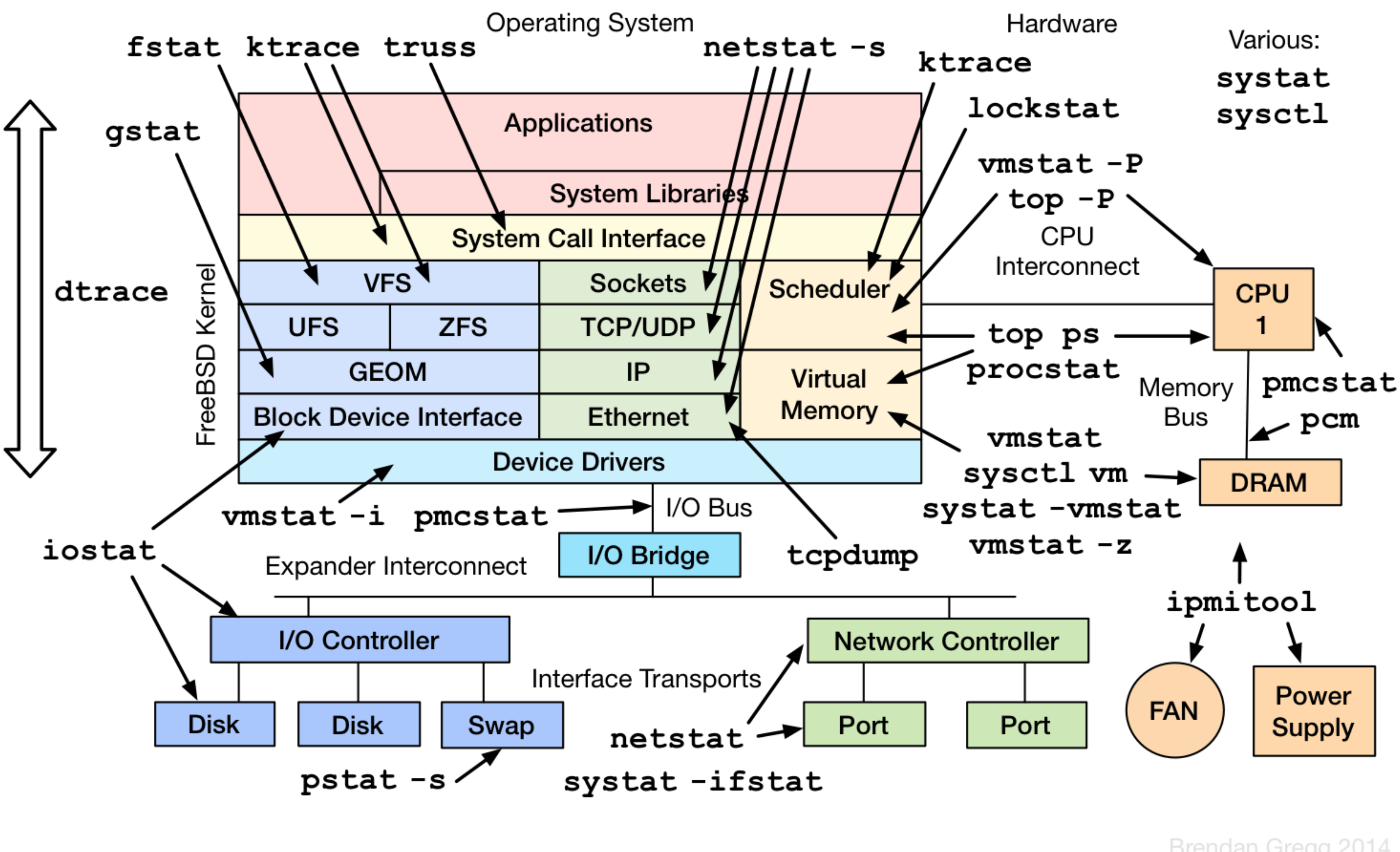
Brendan Gregg

Senior Performance Architect

NETFLIX

meetBSD California 2014

BSD Observability

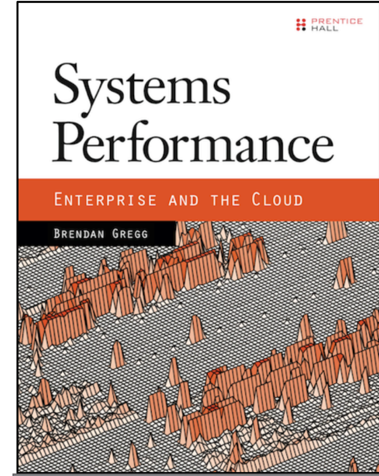


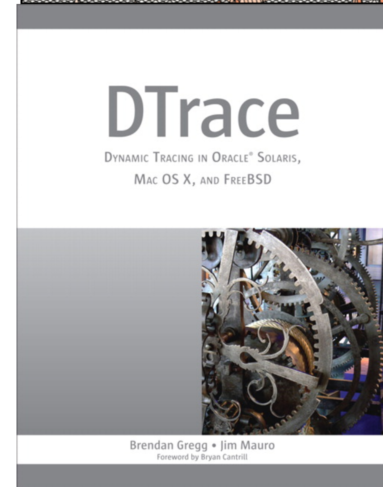
NETFLIX

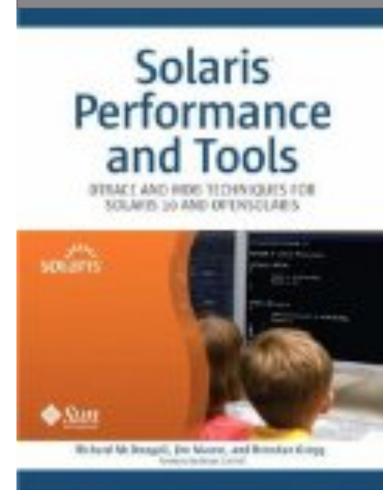
- FreeBSD for content delivery
  - Open Connect Appliances
  - Approx 33% of US Internet traffic at night
- AWS EC2 Linux cloud for interfaces
  - Tens of thousands of instances
  - CentOS and Ubuntu
- Performance is critical
  - Customer satisfaction: >50M subscribers
  - $$$ price/performance

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







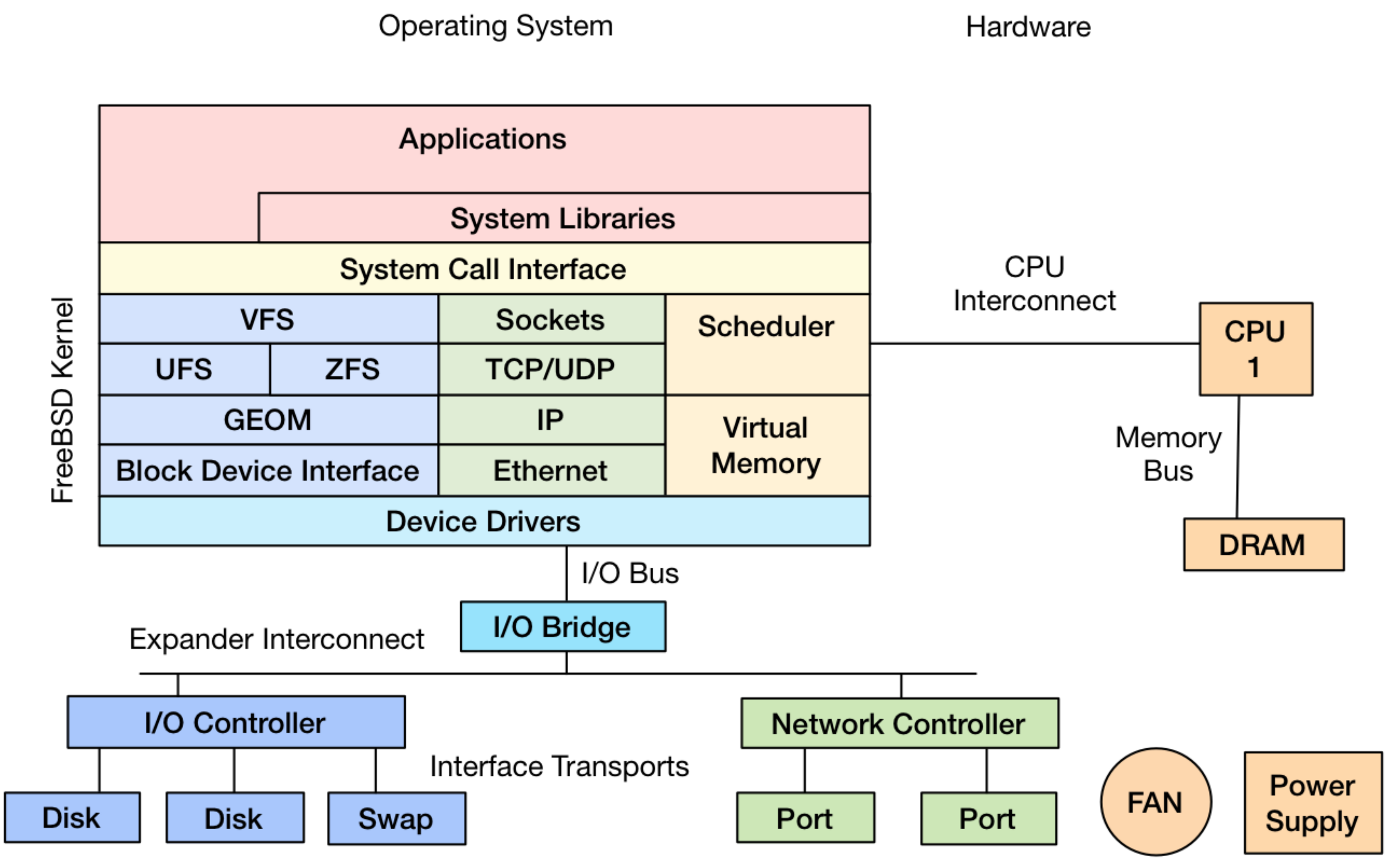
Agenda

A brief discussion of 5 facets of performance analysis on FreeBSD

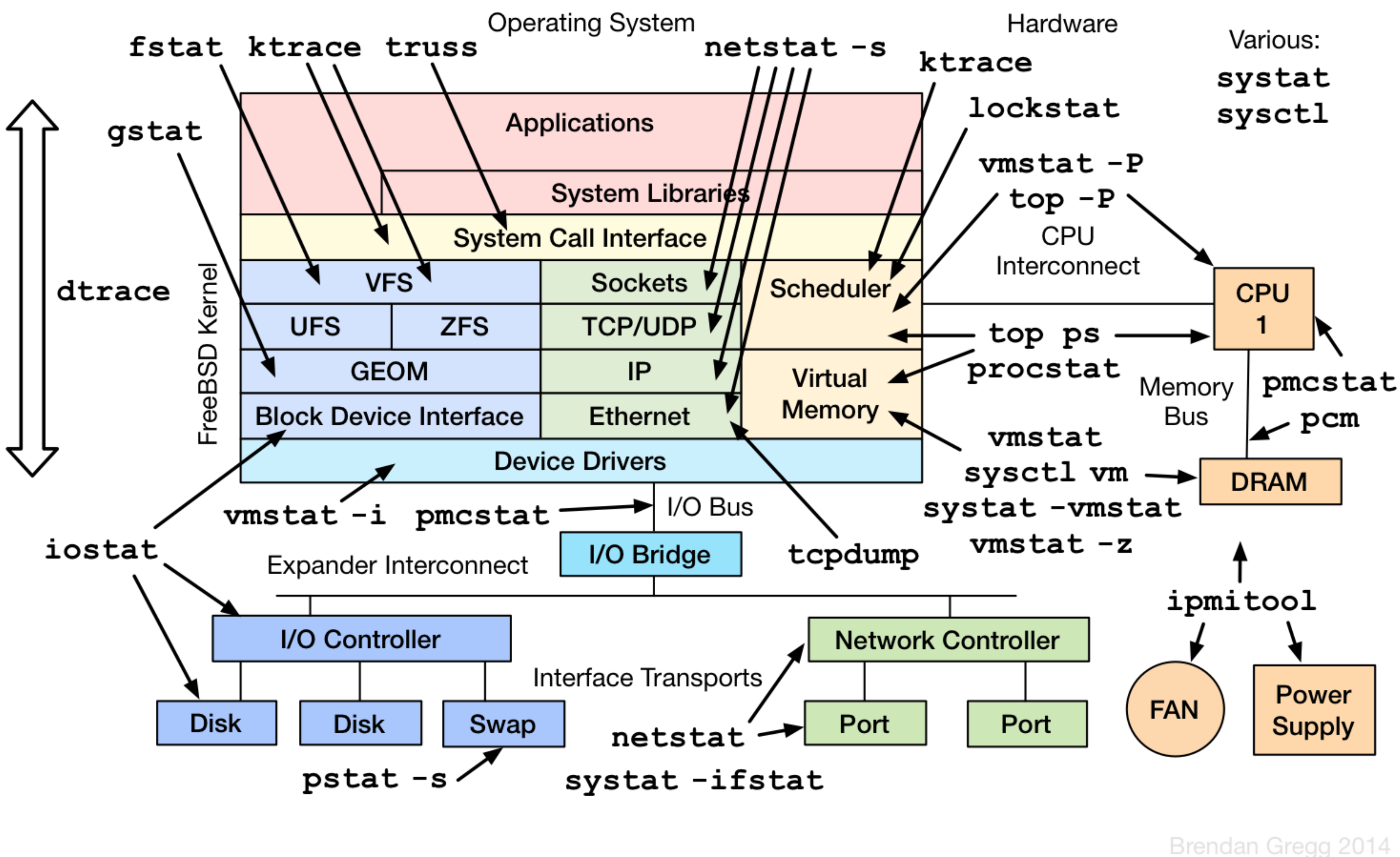
1. Observability Tools
2. Methodologies
3. Benchmarking
4. Tracing
5. Counters

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…

uptime

- One way to print *load averages*:

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

- 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

top

- Includes -P to show processors:

```
# last pid: 32561;  load averages:  2.67,  3.20,  3.03 up 6+17:13:49  19:20:59
70 processes:  1 running, 69 sleeping
CPU 0:   0.8% user,  0.0% nice,  4.7% system, 19.5% interrupt, 75.0% idle
CPU 1:   2.3% user,  0.0% nice,  2.3% system, 17.2% interrupt, 78.1% idle
CPU 2:   2.3% user,  0.0% nice,  6.3% system, 21.1% interrupt, 70.3% idle
CPU 3:   0.8% user,  0.0% nice,  9.4% system, 14.1% interrupt, 75.8% idle
CPU 4:   0.8% user,  0.0% nice,  8.6% system, 12.5% interrupt, 78.1% idle
CPU 5:   1.6% user,  0.0% nice,  3.9% system, 15.6% interrupt, 78.9% idle
[…]
Mem: 295M Active, 236G Inact, 9784M Wired, 1656M Buf, 3704M Free
Swap: 32G Total, 108M Used, 32G Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU COMMAND
 1941 www           1   4   -4 55512K 26312K kqread  9 512:43  4.98% nginx
 1930 www           1   4   -4 55512K 24000K kqread  3 511:34  4.44% nginx
 1937 www           1   4   -4 51416K 22648K kqread  4 510:32  4.35% nginx
 1937 www           1   4   -4 51416K 22648K kqread 10 510:32  4.10% nginx
 […]
```

- WCPU: weighted CPU, another decaying average

vmstat

- Virtual memory statistics and more:

```
$ 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:

```
# iostat –xz 1
[…]
                              extended device statistics
device     r/s   w/s    kr/s     kw/s  qlen svc_t  %b
ada4       5.0   0.0  5087.8      0.0     0   3.8   2
da1        6.0   0.0  6105.3      0.0     0   7.7   3
da8        4.0   0.0  4070.2      0.0     0   1.9   1
da18       3.0   0.0  2098.7      0.0     0   7.4   2
da19       3.0   0.0  3052.7      0.0     0   1.9   1
da25       3.0   0.0  3052.7      0.0     0   1.9   1
da31       3.0   0.0  2989.1      0.0     0   5.3   2
```

workload analysis

resulting performance

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

systat -ifstat

- Network interface throughput:

```
# systat –ifstat
 /0   /1   /2   /3   /4   /5   /6   /7   /8   /9   /10
     Load Average   |||||||||||||||||

      Interface           Traffic               Peak                Total
            lo0  in      0.000 KB/s        16.269 KB/s          2.314 GB
                 out     0.000 KB/s        16.269 KB/s          2.314 GB

           cxl0  in     31.632 MB/s        31.632 MB/s         19.346 TB
                 out   800.456 MB/s       800.456 MB/s        786.230 TB
```

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

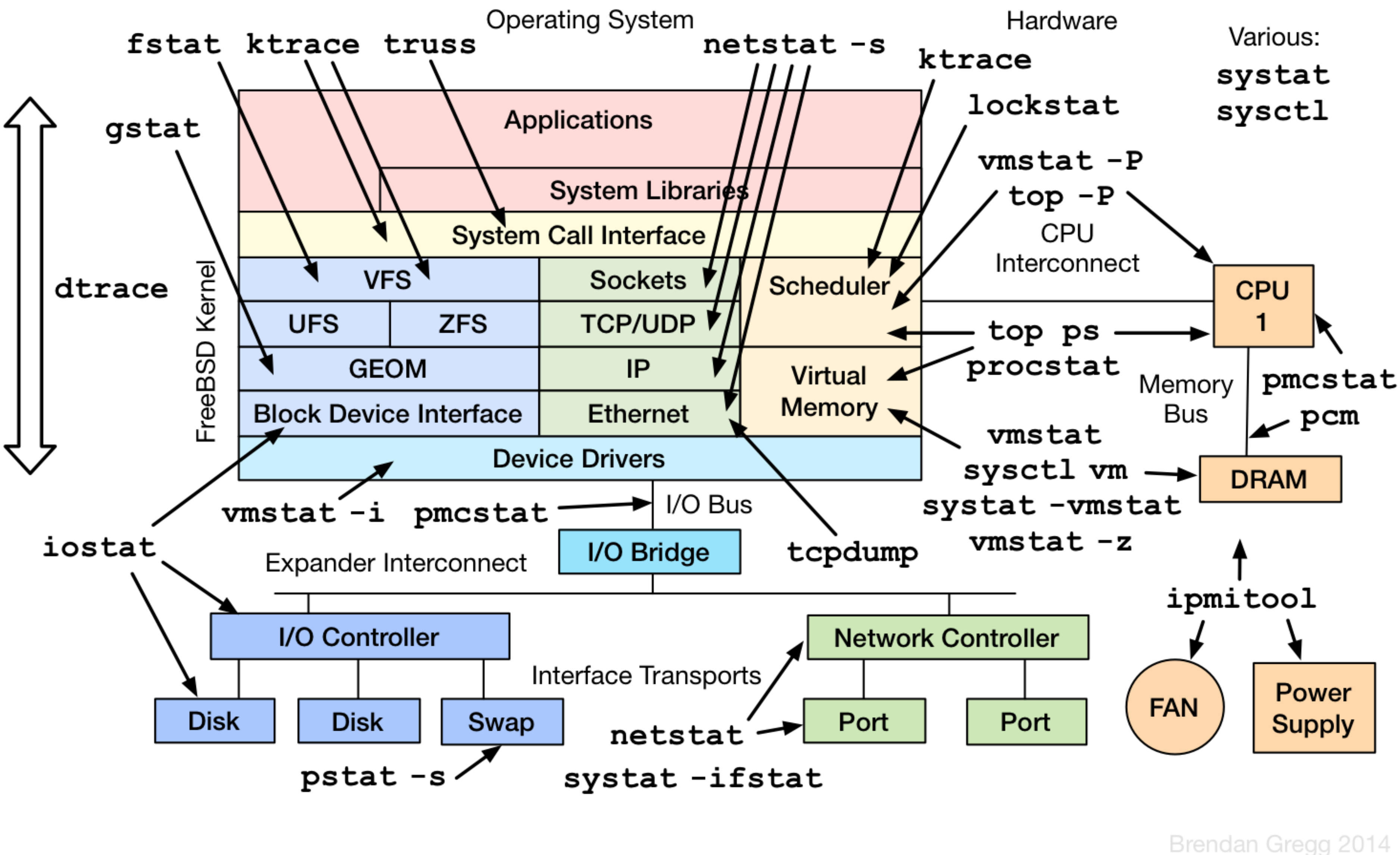
systat -vmstat

```
# systat -vmstat
  1 users    Load  2.86  2.99  3.03                  Oct 30 19:57
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:                                                            Interrupts
  r   p   d   s   w   Csw  Trp  Sys  Int  Sof  Flt        ioflt 88456 total
         10  65      400k  24k  56k  74k 3503  129     29 cow         uart2 10
                                                       92 zfod      1 ehci0 16
 5.7%Sys  18.8%Intr  2.1%User  0.0%Nice 73.4%Idle         ozfod     2 ehci1 23
|    |    |    |    |    |    |    |    |    |           %ozfod  1129 cpu0:timer
===+++++++++>                                             daefr     1 igb0:que 0
                                            25 dtbuf     5 prcfr     1 igb0:que 1
Namei     Name-cache   Dir-cache    2621440 desvn   285817 totfr     1 igb0:que 2
   Calls    hits   %    hits   %      70104 numvn          react     1 igb0:que 3
  182004  182004 100                  40558 frevn          pdwak     1 igb0:que 4
                                                       104 pdpgs     1 igb0:que 5
Disks   md0   md1   md2   md3  ada0  ada1  ada2             intrn     1 igb0:que 6
KB/t   0.00  0.00  0.00  0.00   564   546   520   10409576 wire      1 igb0:que 7
tps       0     0     0     0   131   180   158     251280 act         igb0:link
MB/s   0.00  0.00  0.00  0.00 72.14 95.77 80.11    248112k inact       t5nex0:evt
%busy     0     0     0     0    17    22    32            cache  5720 t5nex0:0.0
                                                   2727140 free   5652 t5nex0:0.1
                                                   1696608 buf    5648 t5nex0:0.2

[…]
```

DTrace

```
# kldload dtrace
# dtrace -ln 'fbt:::entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
    4        fbt            kernel                camstatusentrycomp entry
    6        fbt            kernel            cam_compat_handle_0x17 entry
    8        fbt            kernel                   cam_periph_done entry
    9        fbt            kernel                     camperiphdone entry
   11        fbt            kernel                         heap_down entry
   13        fbt            kernel               cam_ccbq_remove_ccb entry
   15        fbt            kernel          cam_module_event_handler entry
   17        fbt            kernel                   camisr_runqueue entry
   19        fbt            kernel          xpt_alloc_device_default entry
   21        fbt            kernel                 xpt_async_process entry
   22        fbt            kernel             xpt_async_process_dev entry
   24        fbt            kernel             xpt_async_process_tgt entry
   26        fbt            kernel                   xpt_boot_delay entry
   27        fbt            kernel                        xpt_config entry
   29        fbt            kernel                xpt_destroy_device entry
   30        fbt            kernel            xpt_dev_async_default entry
   32        fbt            kernel                  xpt_done_process entry
   34        fbt            kernel                       xpt_done_td entry
   35        fbt            kernel            xpt_finishconfig_task entry
   37        fbt            kernel                      xpt_modevent entry
   39        fbt            kernel                  xpt_periph_init entry
   40        fbt            kernel                  xpt_release_bus entry
[…28472 lines truncated…]
```

run <u>all</u> the things?



2. Methodologies

Methodologies & Tools

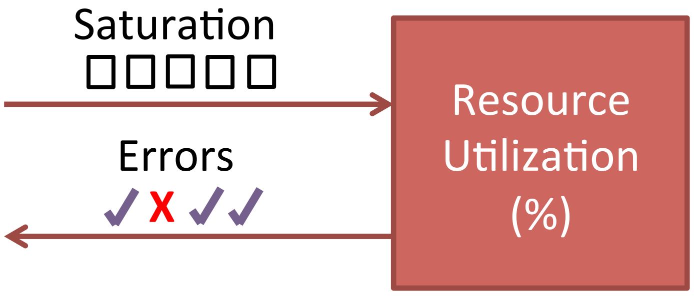
- Many awesome tools
  - Only awesome if you actually use them
  - The real problem becomes how to use them
- Methodologies can guide usage

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

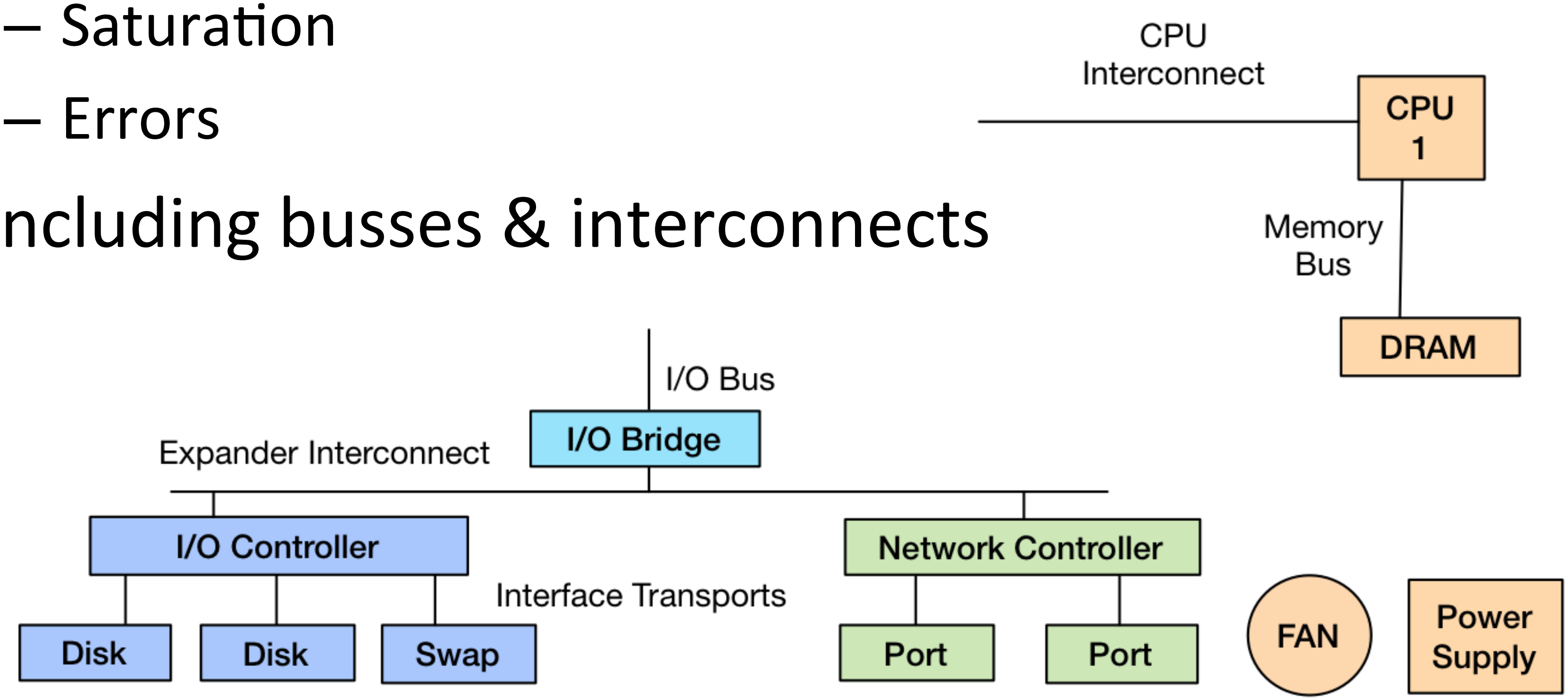
- For example, the USE Method:
  - For every resource, check:
    - Utilization
    - Saturation
    - Errors



- 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

(http://www.brendangregg.com/USEmethod/use-freebsd.html)

component	type	metric
CPU	utilization	system-wide: `vmstat 1`, "us" + "sy"; per-cpu: `vmstat -P`; per-process: `top`, "WCPU" for weighted and recent usage; per-kernel-process: `top -S`, "WCPU"
CPU	saturation	system-wide: `uptime`, "load averages" > CPU count; `vmstat 1`, "procs:r" > CPU count; per-cpu: DTrace to profile CPU run queue lengths [1]; per-process: DTrace of scheduler events [2]
CPU	errors	`dmesg`; /var/log/messages; `pmcstat` for PMC and whatever error counters are supported (eg, thermal throttling)
Memory capacity	utilization	system-wide: `vmstat 1`, "fre" is main memory free; `top`, "Mem:"; per-process: `top -o res`, "RES" is resident main memory size, "SIZE" is virtual memory size; `ps -auxw`, "RSS" is resident set size (Kbytes), "VSZ" is virtual memory size (Kbytes)
Memory capacity	saturation	system-wide: `vmstat 1`, "sr" for scan rate, "w" for swapped threads (was saturated, may not be now); `swapinfo`, "Capacity" also for evidence of swapping/paging; per-process: DTrace [3]
Memory capacity	errors	physical: `dmesg`?; /var/log/messages?; virtual: DTrace failed malloc()s
Network Interfaces	utilization	system-wide: `netstat -i 1`, assume one very busy interface and use input/output "bytes" / known max (note: includes localhost traffic); per-interface: `netstat -I` *`interface`* `1`, input/output "bytes" / known max
Network Interfaces	saturation	system-wide: `netstat -s`, for saturation related metrics, eg `netstat -s | egrep 'retrans|drop|out-of-order|memory problems|overflow'`; per-interface: DTrace
Network Interfaces	errors	system-wide: `netstat -s | egrep 'bad|checksum'`, for various metrics; per-interface: `netstat -i`, "Ierrs", "Oerrs" (eg, late collisions), "Colls" [5]
Storage device I/O	utilization	system-wide: `iostat -xz 1`, "%b"; per-process: DTrace io provider, eg, iosnoop or iotop (DTT, needs porting)
Storage device I/O	saturation	system-wide: `iostat -xz 1`, "qlen"; DTrace for queue duration or length [4]

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
  - http://www.brendangregg.com/blog/2014-05-03/the-benchmark-paradox.html
- 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 is CPU saturated, but the 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

https://www.rfc-editor.org/rfc/museum/tcp-ip-digest/tcp-ip-digest.v1n6.1

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:

```
# pmcstat –S RESOURCE_STALLS.ANY -O out.pmc sleep 10
# pmcstat -R out.pmc -z 32 -G out.stacks
CONVERSION STATISTICS:
 #exec/elf                                        25
 #samples/total                                   107362
 #samples/unknown-function                        244
 #callchain/dubious-frames                        89
# more out.stacks
@ RESOURCE_STALLS.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

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

Can also emit gprof/Kcallgrind output

PMC Counters

- Profile based on any counter:

```
# 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.LOADS
        MISALIGN_MEM_REF.STORES
        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
[…]
```

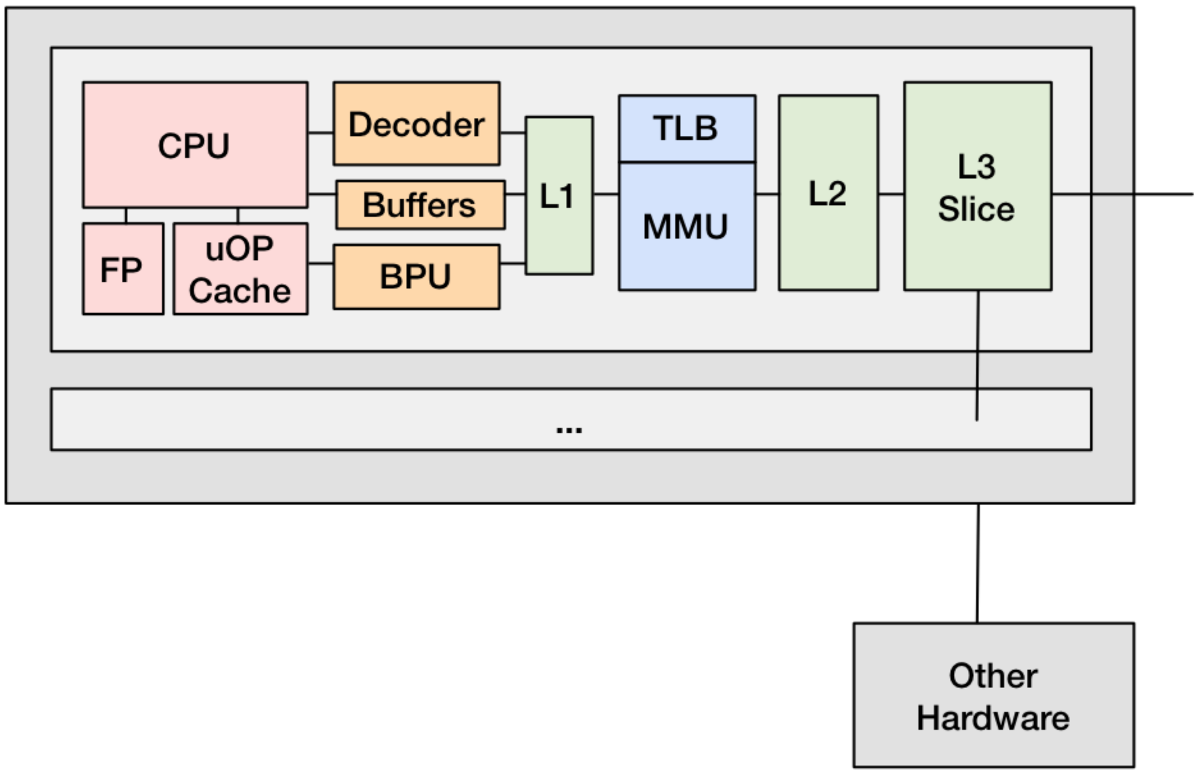
Beware of high frequency events, and use -n to limit samples

PMC Counter Groups

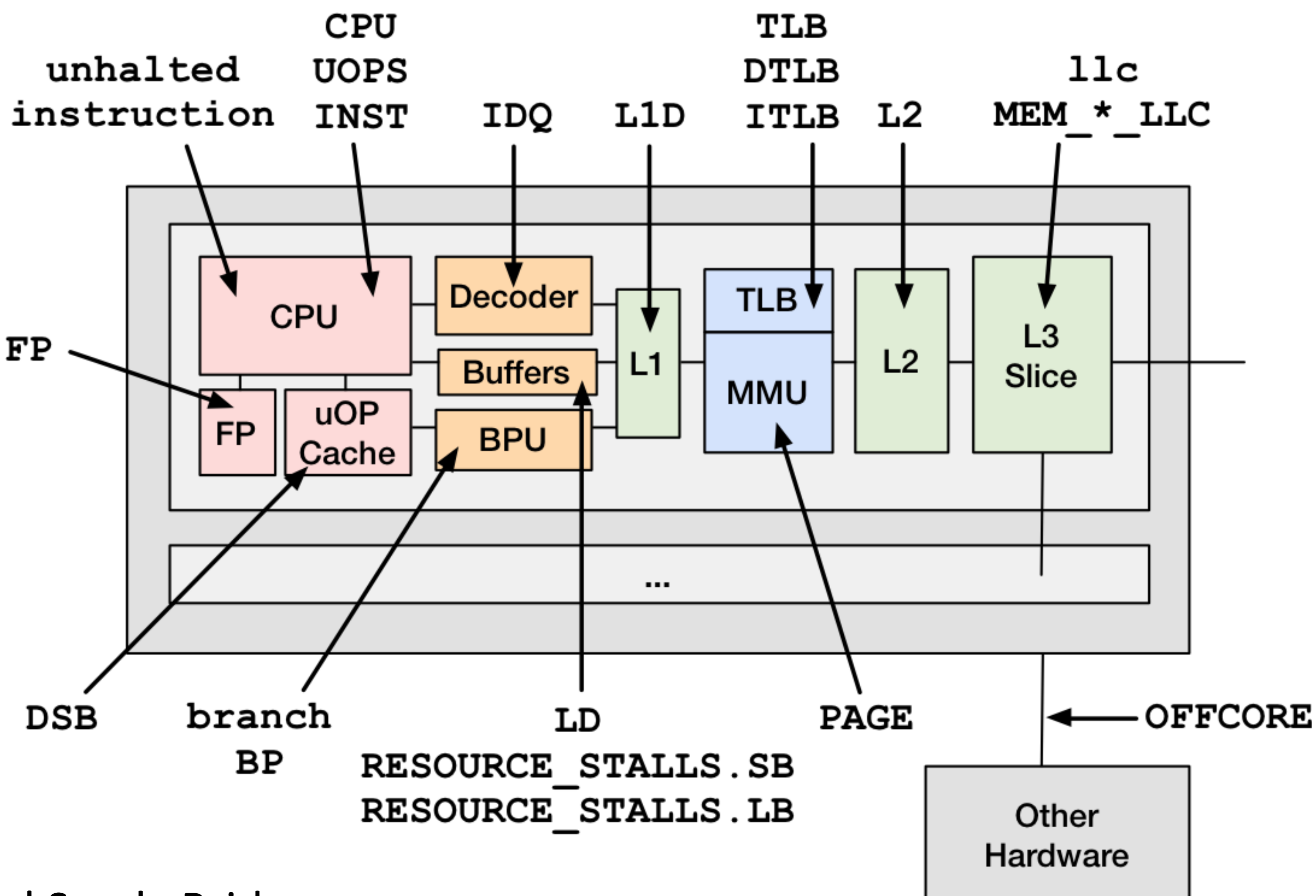
- Counters by group (eg, Intel Sandy Bridge):

```
# pmccontrol -L | sed -n '/[_\.-]/s/[_\.-].*//p' | sort | \
    uniq –c | sort -n | pr -t3
   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:

```
# 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:

```
# 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

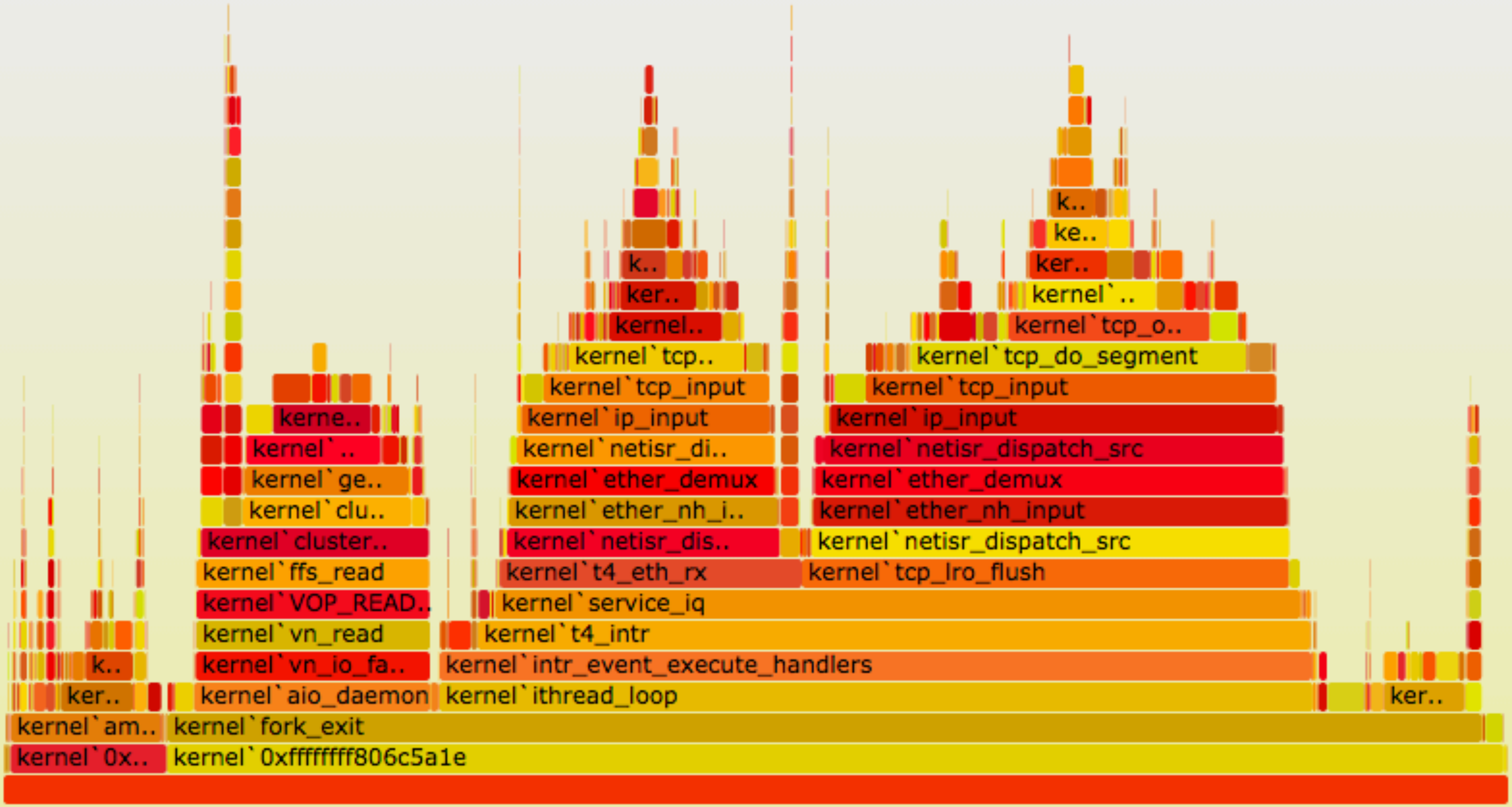
- CPU flame graph (using DTrace):

```
# 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
```

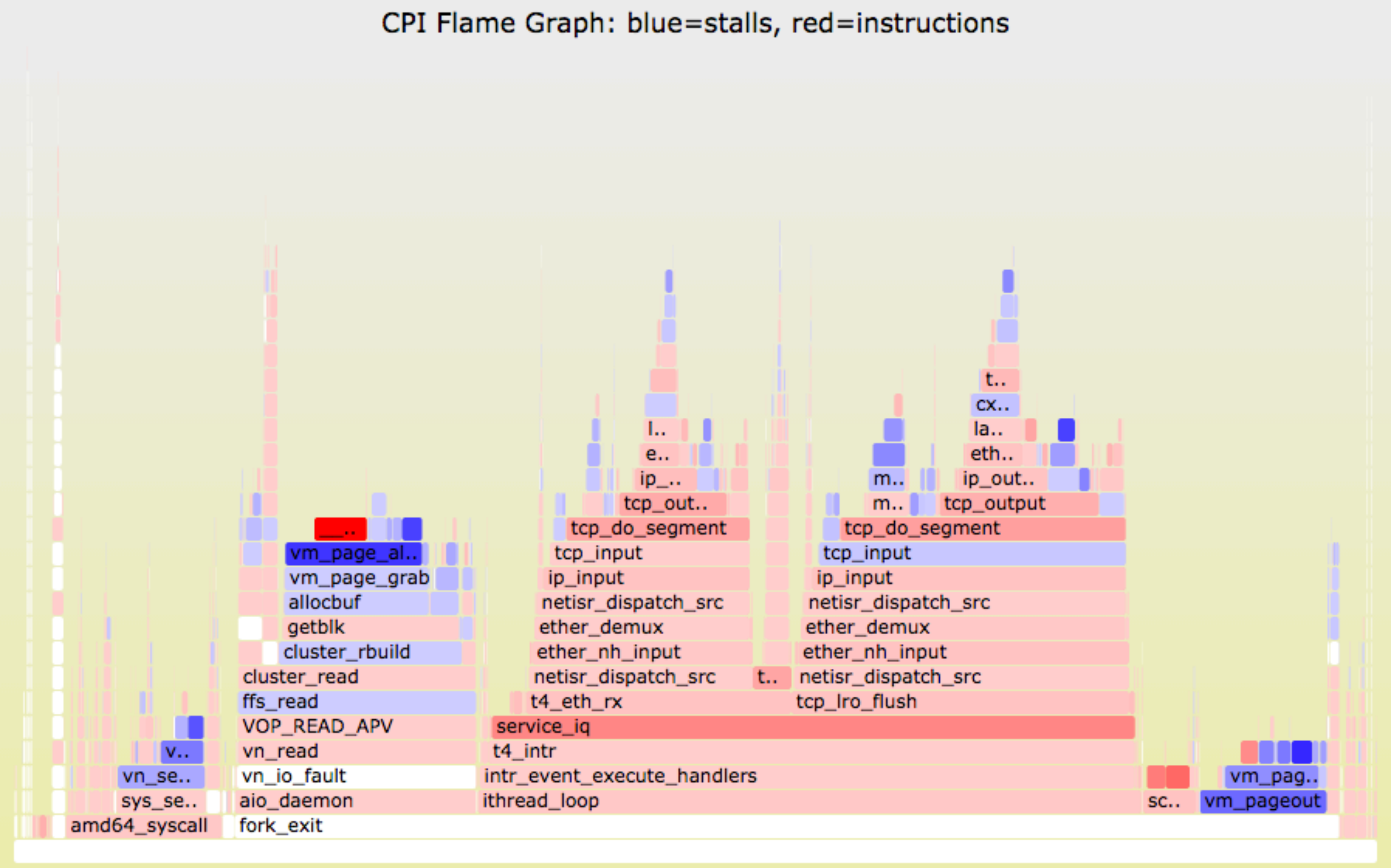
- Stall cycle flame graph (using pmcstat):

```
…
# 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
```

FreeBSD Kernel CPU Flame Graph (no cpu_idle)



cpu-freebsd02.svg



cpi-flamegraph-01.svg

5. Tracing

Tracing Tools

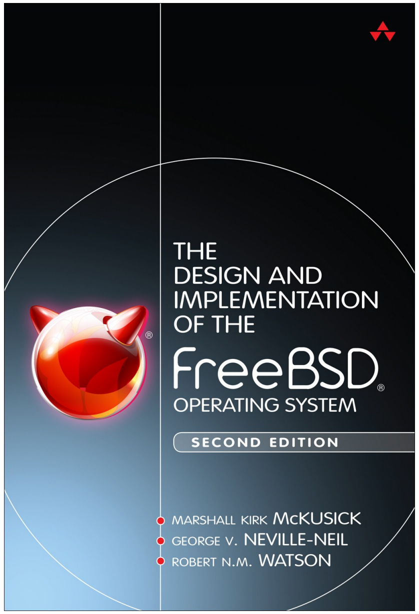
- truss
- tcpdump
- ktrace
- DTrace

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

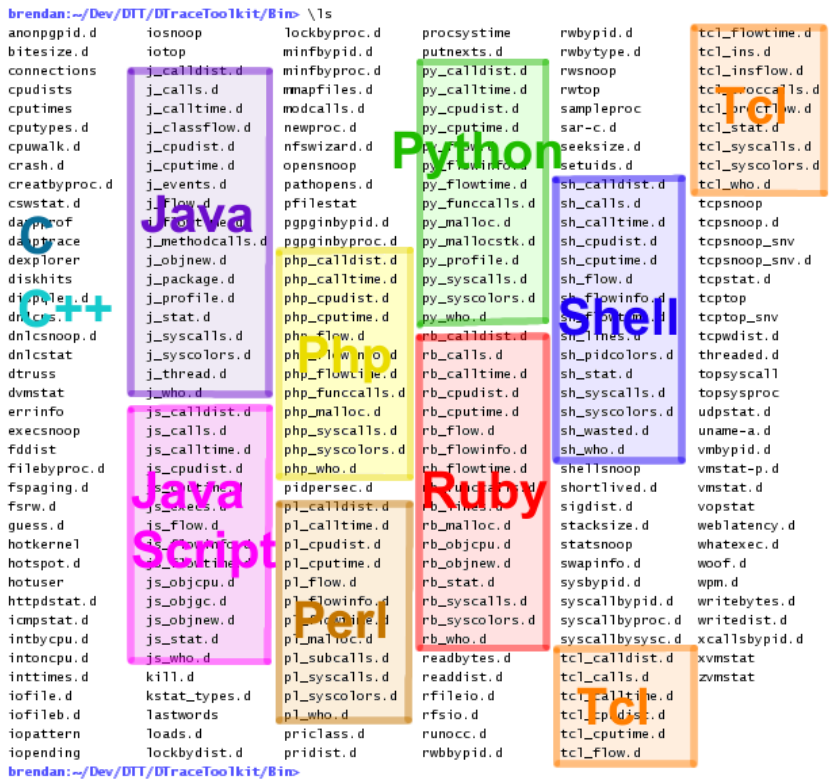
- Practical usage for most companies:
  - A) A performance team (or person)
    - Acquires useful one-liners & scripts
    - Develops custom one-liners & scripts
  - B) The rest of the company asks (A) for help
    - They need to know what's possible, to know to ask
  - Or, you buy/develop a GUI that everyone can use
- There are some exceptions
  - Team of kernel/driver developers, who will all write custom scripts

DTrace One-liners

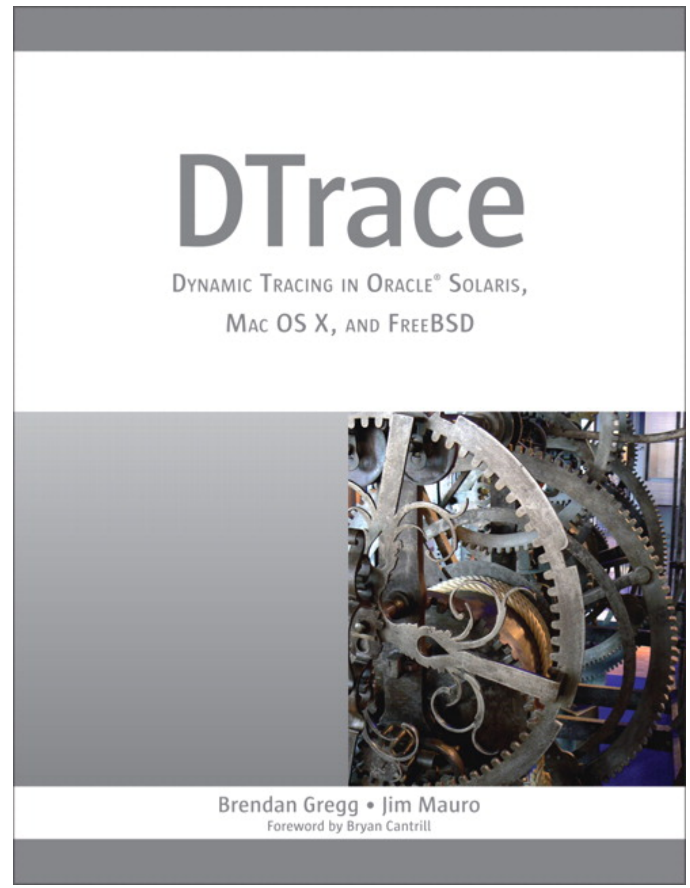
```
# Trace file opens with process and filename:
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'

# 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

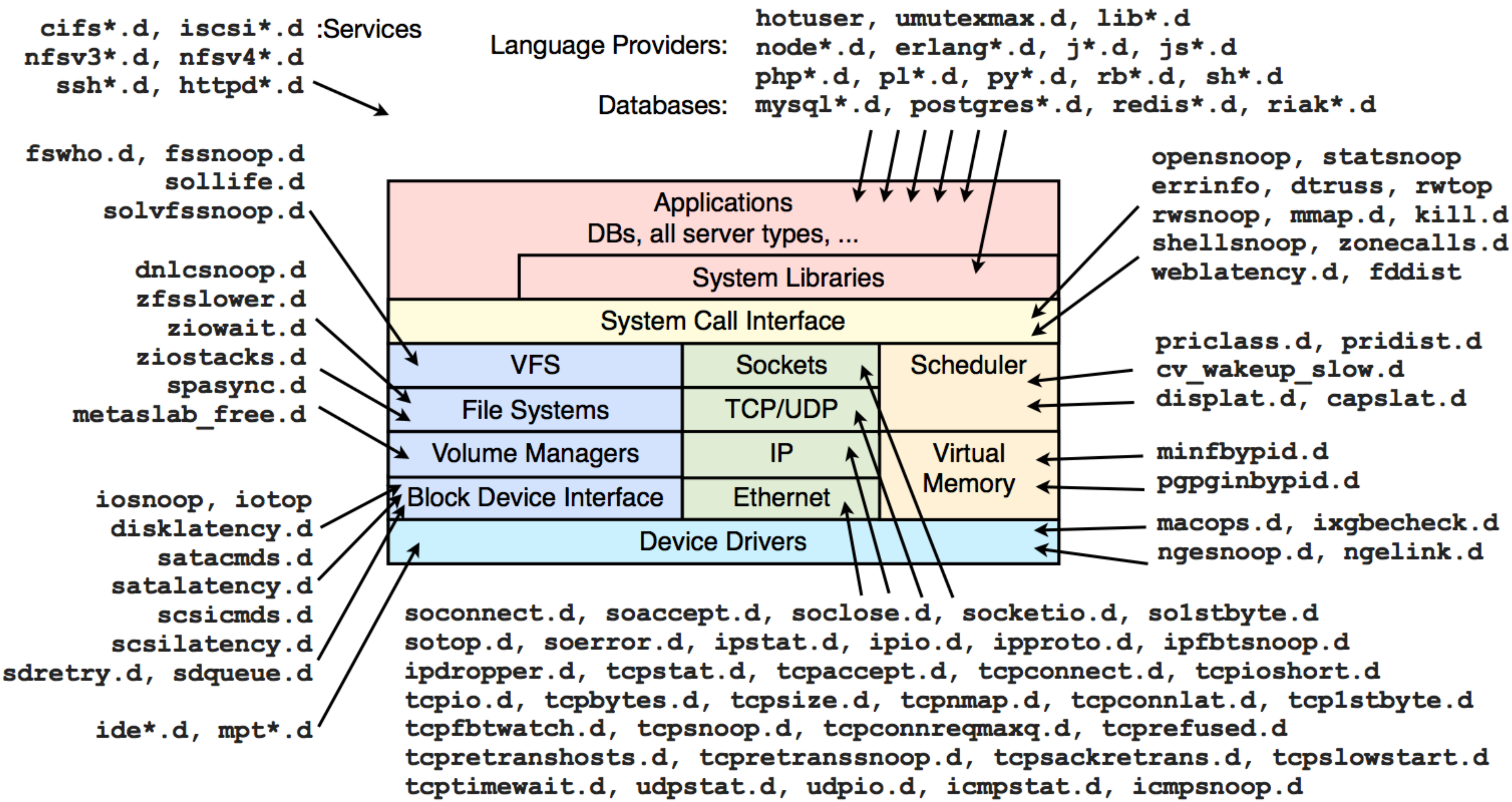
Brendan's Scripts



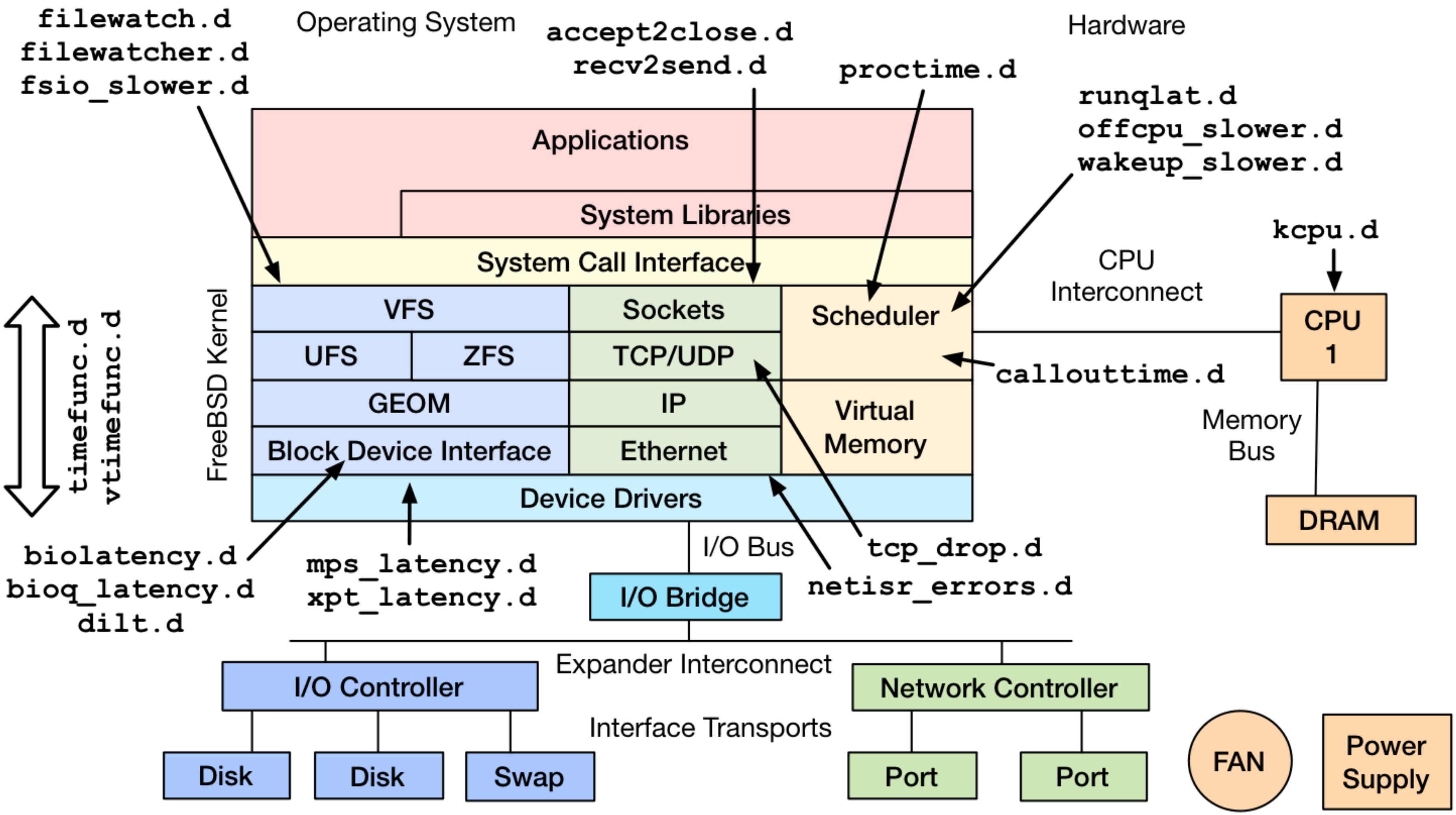
DTraceToolkit



Brendan's Scripts



Brendan's New FreeBSD Scripts (so far)

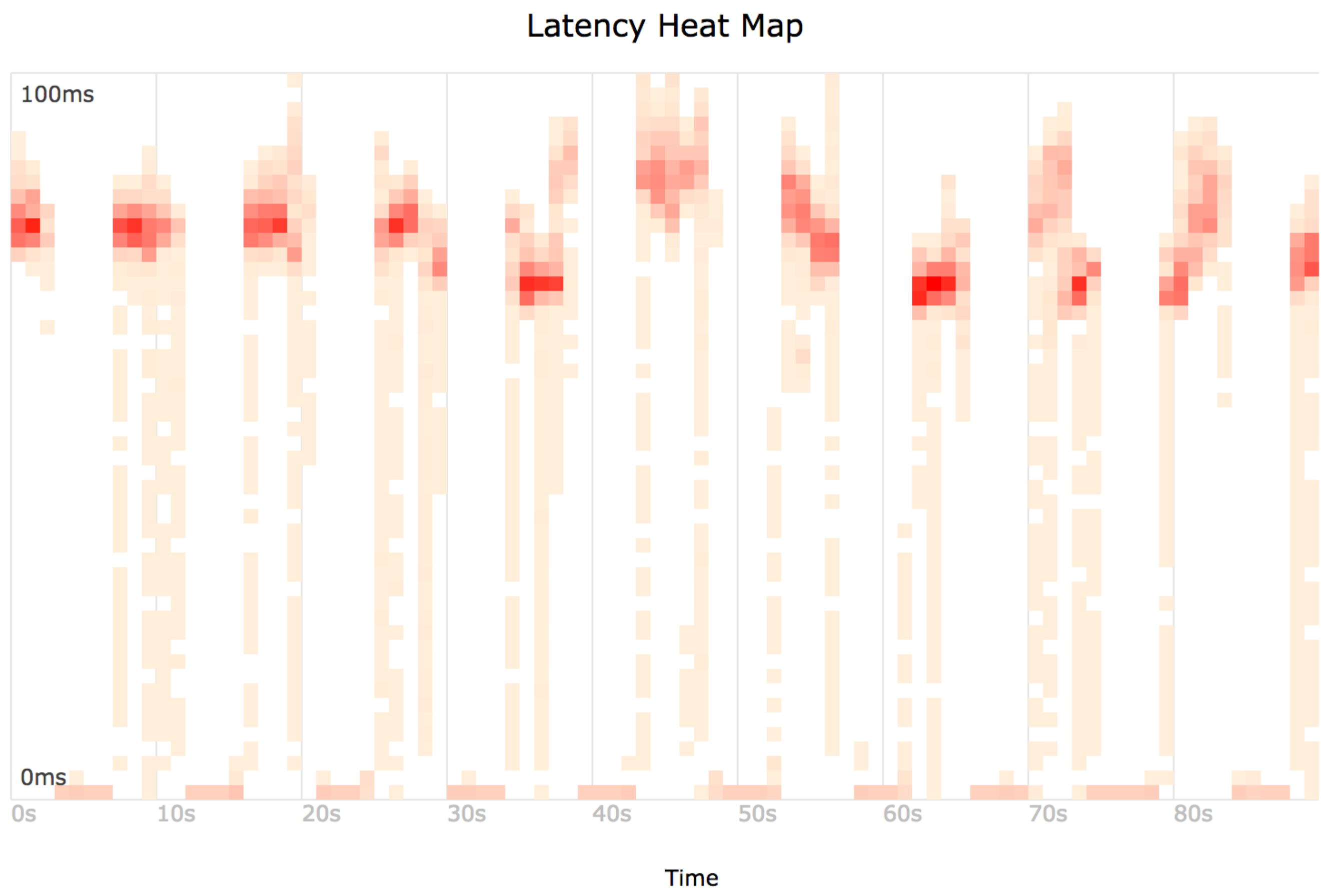


https://github.com/brendangregg/DTrace-tools

DEMO

Heat Maps

- Study latency distributions by-time:



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://lists.freebsd.org/pipermail/freebsd-current/2006-February/061096.html (sixty second pmc how to, by Robert Watson)
  - 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