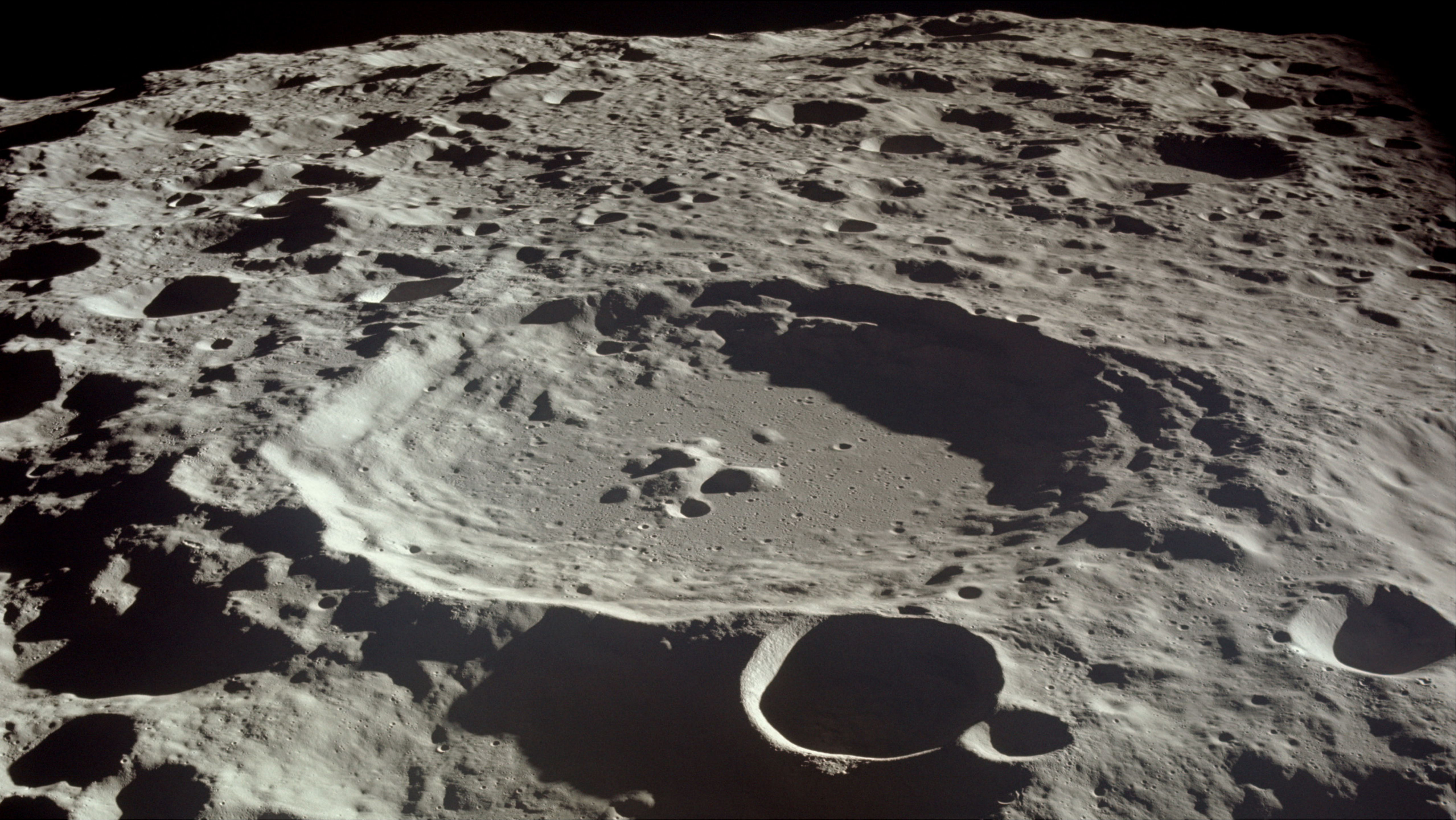
EuroBSDcon 2017

System Performance Analysis Methodologies

Brendan Gregg

Senior Performance Architect

NETFLIX



Apollo Lunar Module Guidance Computer performance analysis

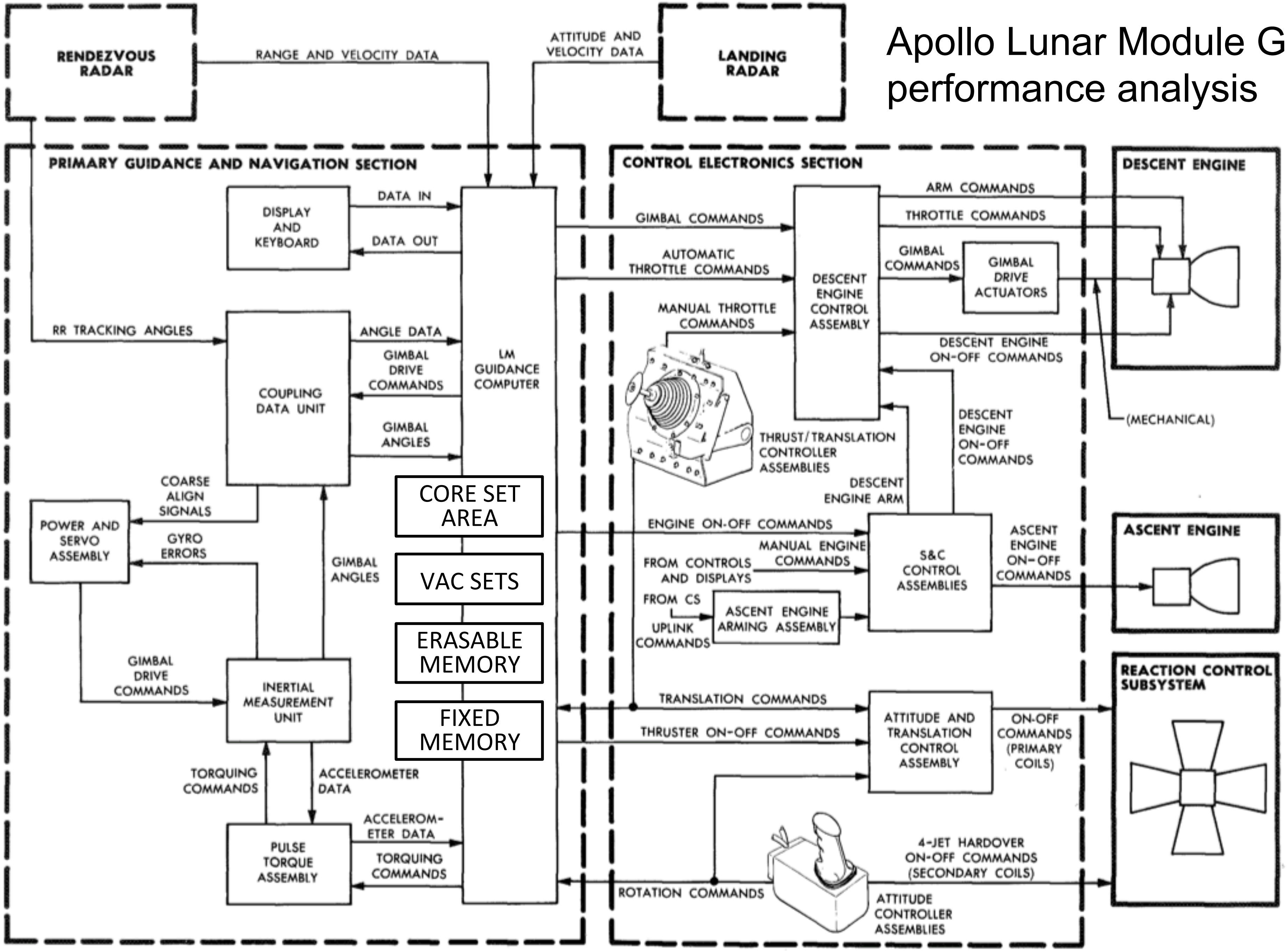
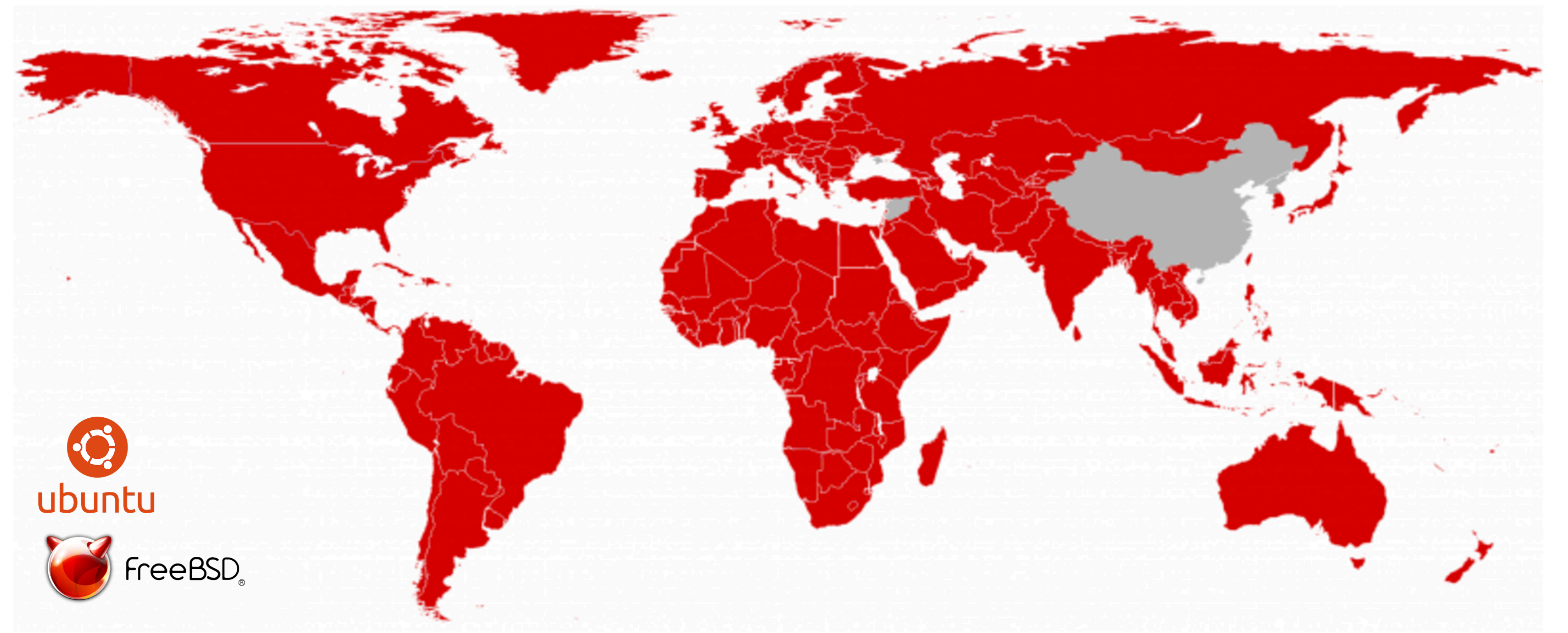


Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram

NETFLIX

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?

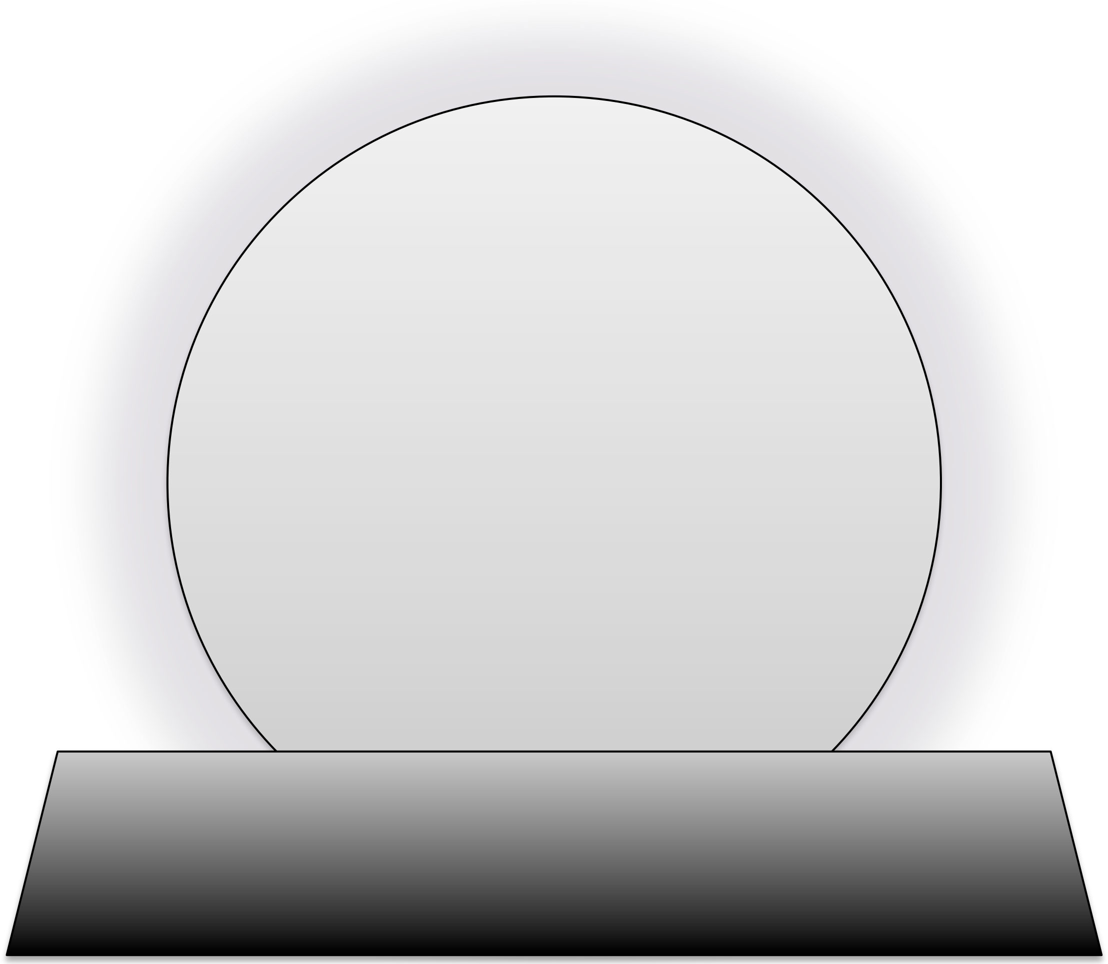
```
$ ps -auxw
USER    PID %CPU %MEM   VSZ  RSS TT  STAT STARTED     TIME COMMAND
root     11 99.9  0.0     0   16  -  RL   22:10   22:27.05 [idle]
root      0  0.0  0.0     0  176  -  DLs  22:10    0:00.47 [kernel]
root      1  0.0  0.2  5408 1040  -  ILs  22:10    0:00.01 /sbin/init --
[…]
```

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

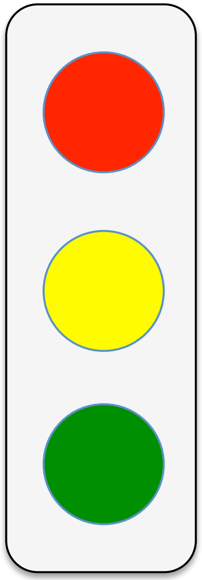
- ~~Drink~~ Tune things at random until the problem goes away

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
3. Everything green? No worries, mate.



- 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

Performance 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

System Methodologies:

- Problem statement method
- Functional diagram method
- Workload analysis
- Workload characterization
- Resource analysis
- USE method
- Thread State Analysis
- On-CPU analysis
- CPU flame graph analysis
- Off-CPU analysis
- Latency correlations
- Checklists
- Static performance tuning
- Tools-based methods

…

Problem Statement Method

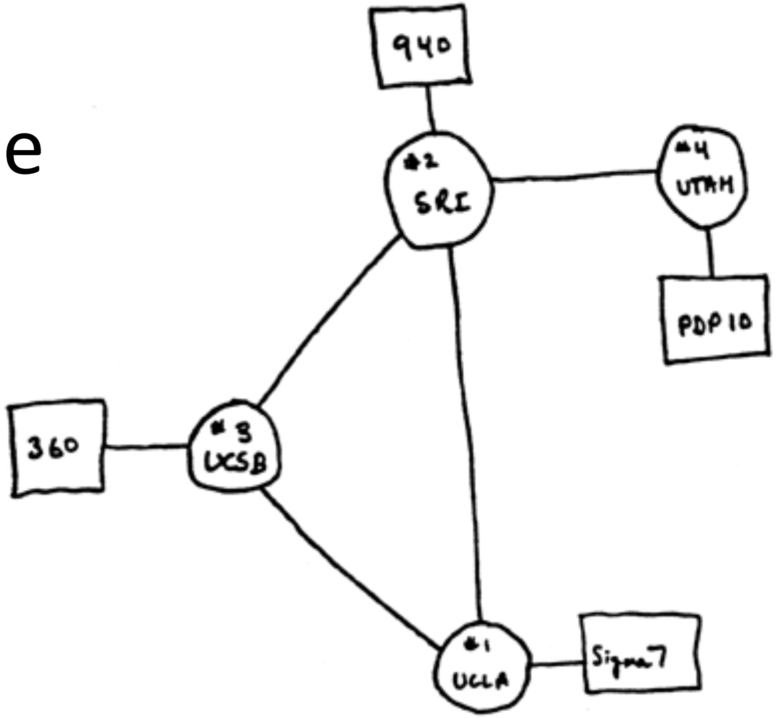
1. What makes you **think** there is a performance problem?
2. Has this system **ever** performed well?
3. What has **changed** recently?
   – software? hardware? load?
4. Can the problem be described in terms of **latency**?
   – or run time. not IOPS or throughput.
5. Does the problem affect **other** 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

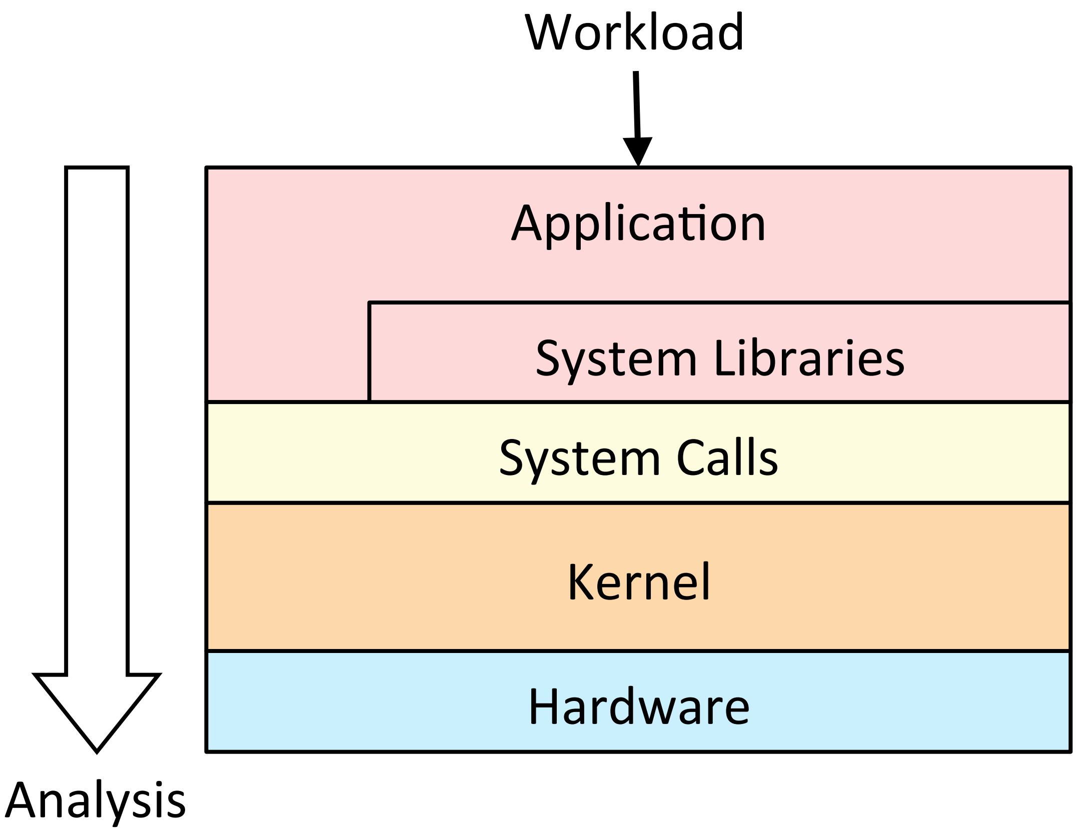
Eg, imagine throughput between the UCSB 360 and the UTAH PDP10 was slow…



ARPA Network 1969

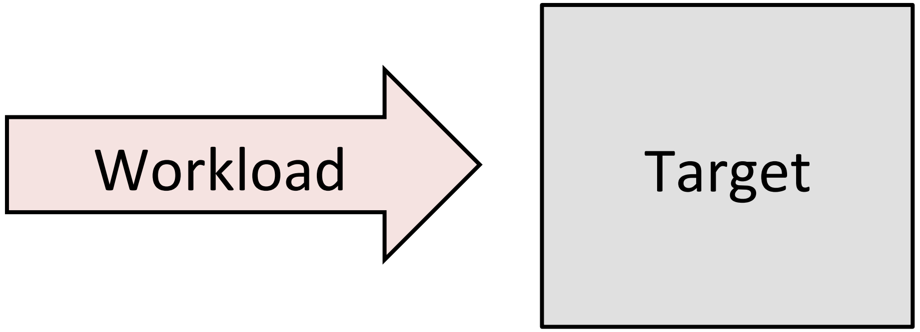
Workload Analysis

- 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

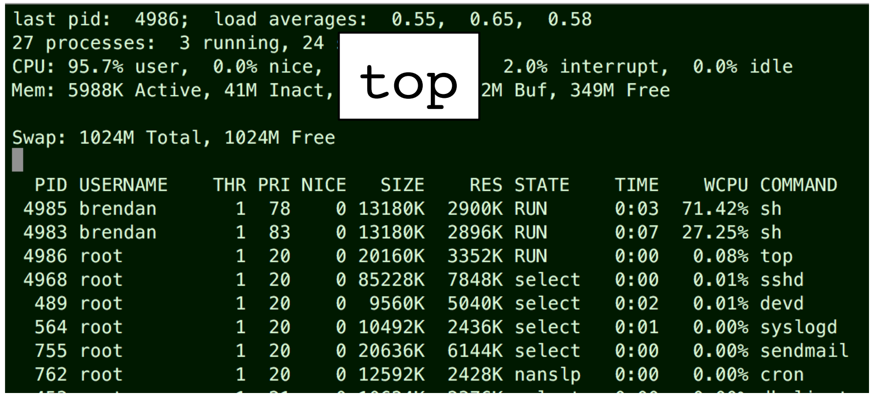
- Check the workload, not resulting performance



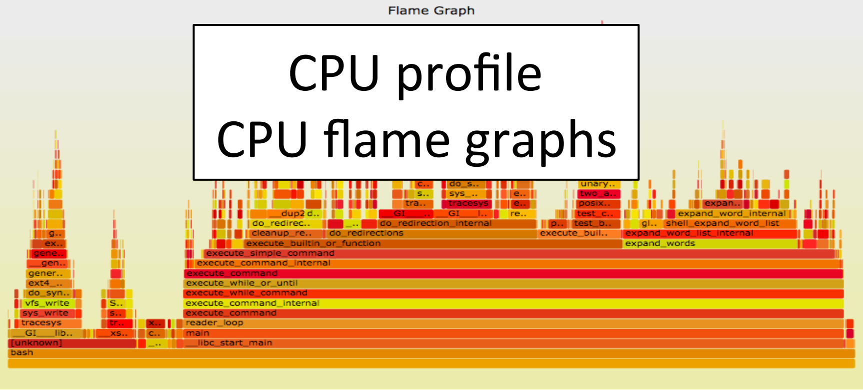
- Eg, for CPUs:
  1. **Who**: which PIDs, programs, users
  2. **Why**: code paths, context
  3. **What**: CPU instructions, cycles
  4. **How**: changing over time

Workload Characterization: CPUs

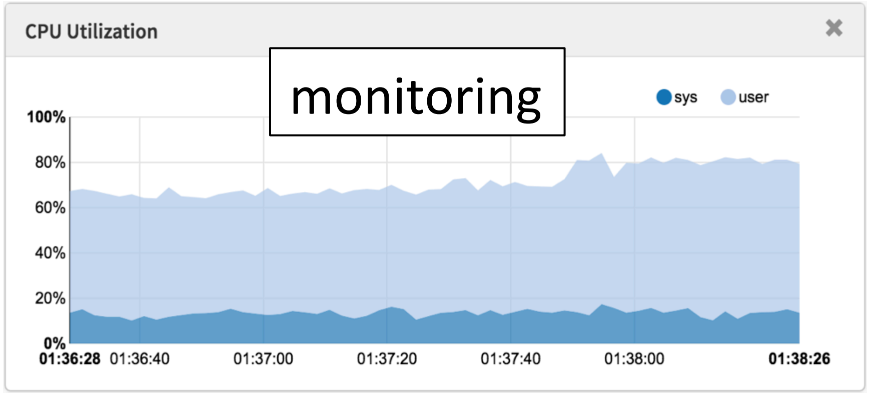
Who



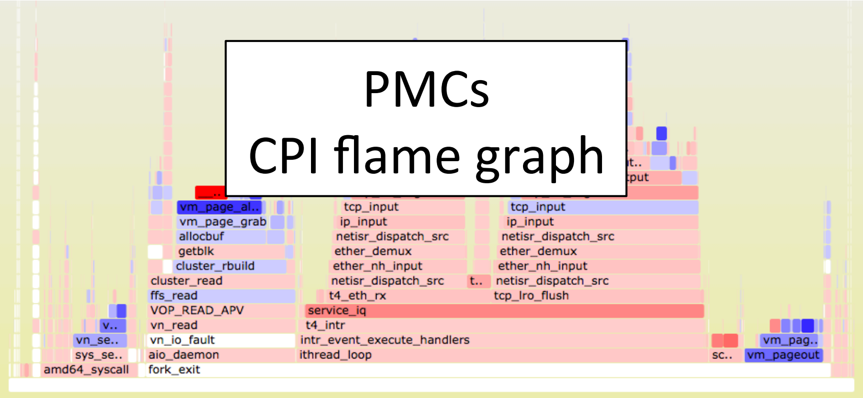
Why



How

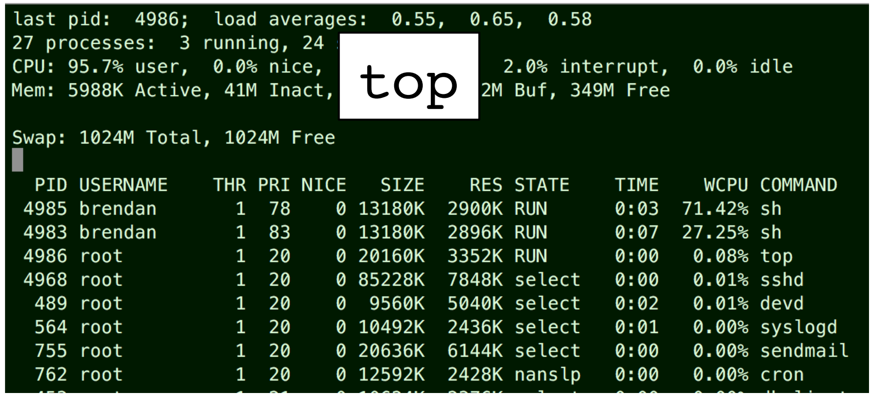


What

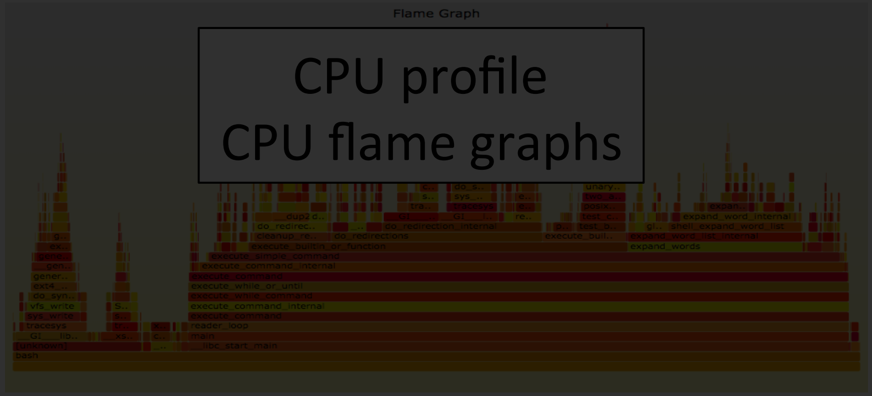


Most companies and monitoring products today

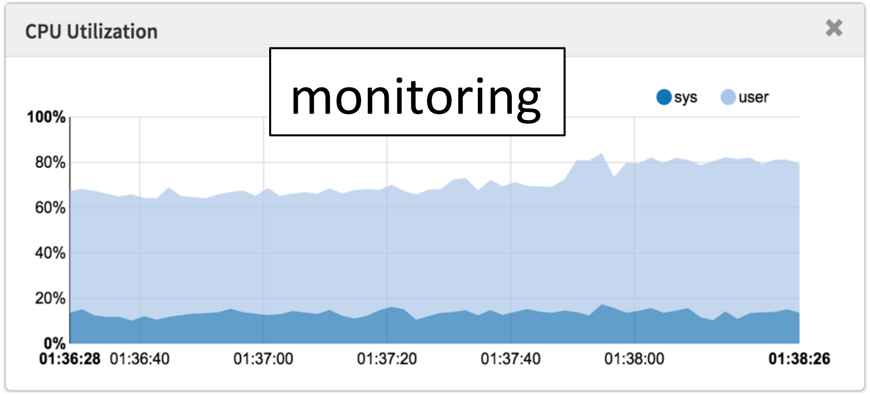
Who



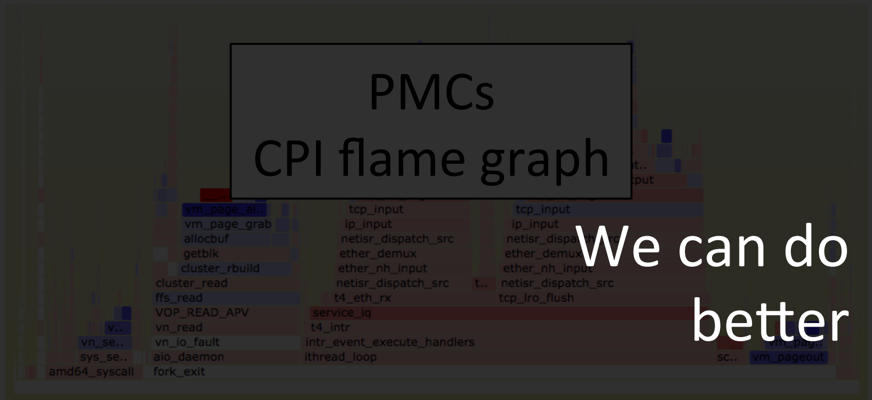
Why



How

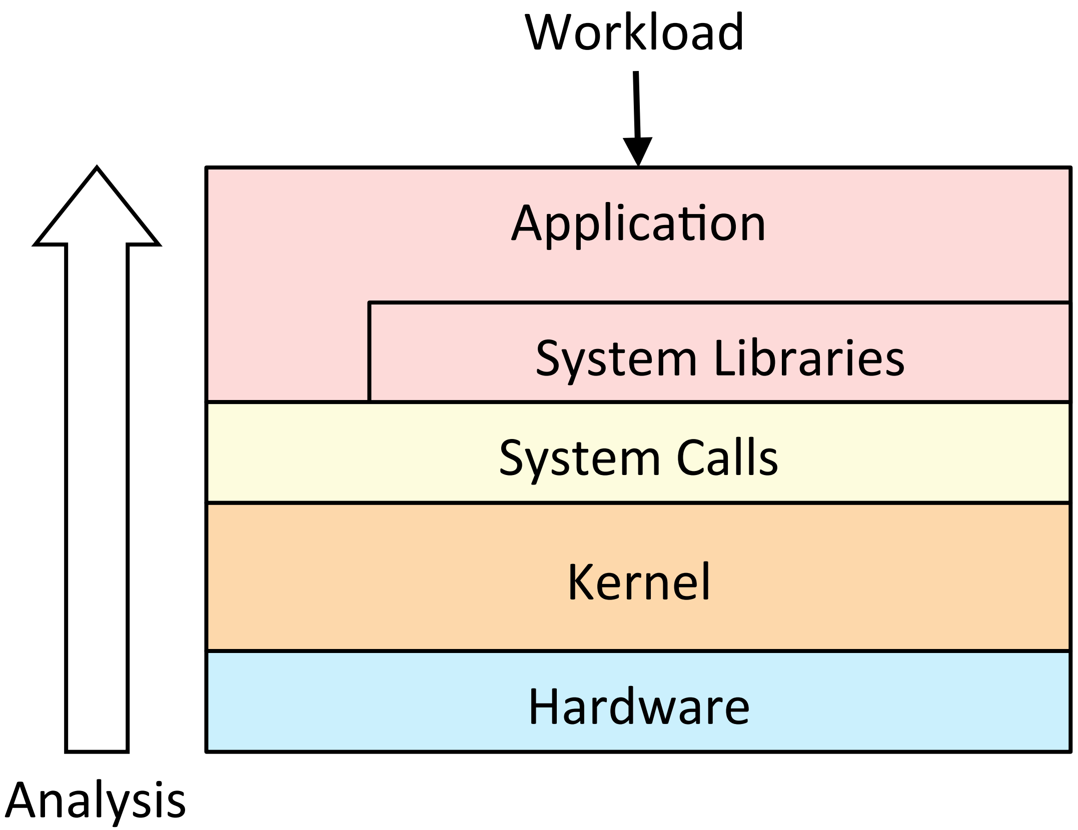


What



Resource Analysis

- Typical approach for system performance analysis: begin with system tools & metrics
- Pros:
  - Generic
  - Aids resource perf tuning
- Cons:
  - Uneven coverage
  - False positives

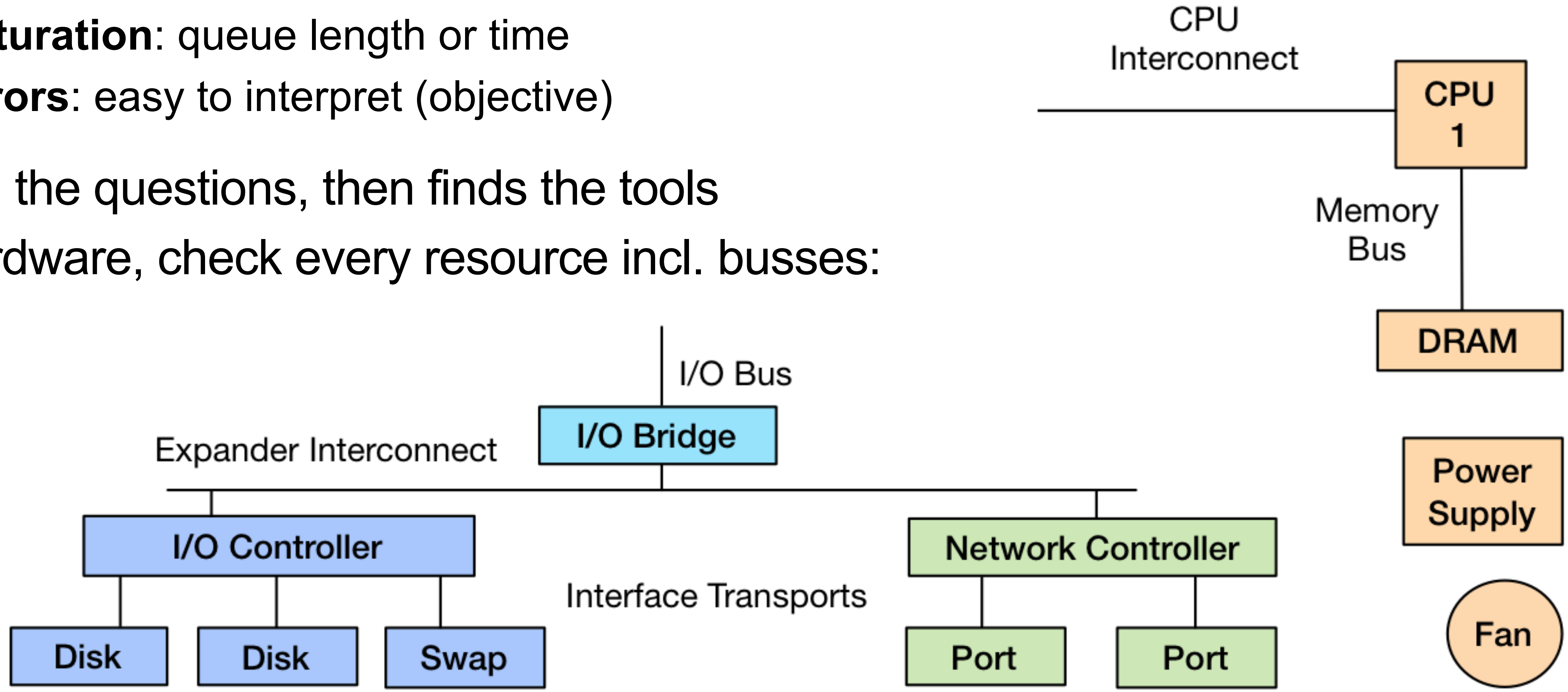


The USE Method

- 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.

- [x] Linux
- [ ] Solaris
- [x] FreeBSD
- [x] Mac OS X

Redraw

http://www.brendangregg.com/USEmethod/use-rosetta.html

Hardware Resources

Resource	Metric	Linux	FreeBSD	Mac OS X
CPU	errors	`perf` (LPE) if processor specific error events (CPC) are available; eg, AMD64's "04Ah Single-bit ECC Errors Recorded by Scrubber" [4]	`dmesg`; /var/log/messages; `pmcstat` for PMC and whatever error counters are supported (eg, thermal throttling)	`dmesg`; /var/log/system.log; Instruments → Counters, for PMC and whatever error counters are supported (eg, thermal throttling)
CPU	saturation	system-wide: `vmstat 1`, "r" > CPU count [2]; `sar -q`, "runq-sz" > CPU count; `dstat -p`, "run" > 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]	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]	system-wide: `uptime`, "load averages" > CPU count; `latency`, "SCHEDULER" and "INTERRUPTS"; per-cpu: `dispqlen.d` (DTT), non-zero "value"; `runocc.d` (DTT), non-zero "%runocc"; per-process: Instruments → Thread States, "On run queue"; DTrace [2]
CPU	utilization	system-wide: `vmstat 1`, "us" + "sy" + "st"; `sar -u`, sum fields except "%idle" and "%iowait"; `dstat -c`, sum fields except "idl" and "wai"; per-cpu: `mpstat -P ALL 1`, sum fields except "%idle" and "%iowait"; `sar -P ALL`, same as `mpstat`; per-process: `top`, "%CPU"; `htop`, "CPU%"; `ps -o pcpu`; `pidstat 1`, "%CPU"; per-kernel-thread: `top`/`htop` ("K" to toggle), where VIRT == 0 (heuristic). [1]	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"	system-wide: `iostat 1`, "us" + "sy"; per-cpu: DTrace [1]; Activity Monitor → CPU Usage or Floating CPU Window; per-process: `top -o cpu`, "%CPU"; Activity Monitor → Activity Monitor, "%CPU"; per-kernel-thread: DTrace profile stack()
CPU interconnect	errors	LPE (CPC) for whatever is available	`pmcstat` and relevant PMCs for whatever is available	Instruments → Counters, and relevent PMCs for whatever is available
CPU interconnect	saturation	LPE (CPC) for stall cycles	`pmcstat` and relevant PMCs for CPU interconnect stall cycles	Instruments → Counters, and relevent PMCs for stall cycles

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.

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

Physical Resources

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

USE Method: Unix 7th Edition Performance Checklist


PDP 11/70 front panel (similar to the 11/45)

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 "unknowns".

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

component	type	metric
CPU	utilization	system-wide: `iostat 1`, utilization is "user" + "nice" + "systm"; per-process: `ps alx`, "CPU" shows recent CPU usage (max 255), and "TIME" shows cumulative minutes:seconds of CPU time
CPU	saturation	`ps alx | awk '$2 == "R" { r++ } END { print r - 1 }'`, shows the number of runnable processes
CPU	errors	console message if lucky, otherwise panic
Memory capacity	utilization	system-wide: unknown [1]; per-type: unknown [2]; per-process: `ps alx`, "SZ" is the in-core (main memory) in blocks (512 bytes); `pstat -p`, "SIZE" is in-core size, in units of core clicks (64 bytes) and printed in octal!
Memory capacity	saturation	system-wide: `iostat 1`, sustained "tpm" may be caused by swapping to disk; significant delays as processes wait for space to swap in
Memory	errors	malloc() returns 0; ENOMEM

Apollo Lunar Module Guidance Computer performance analysis

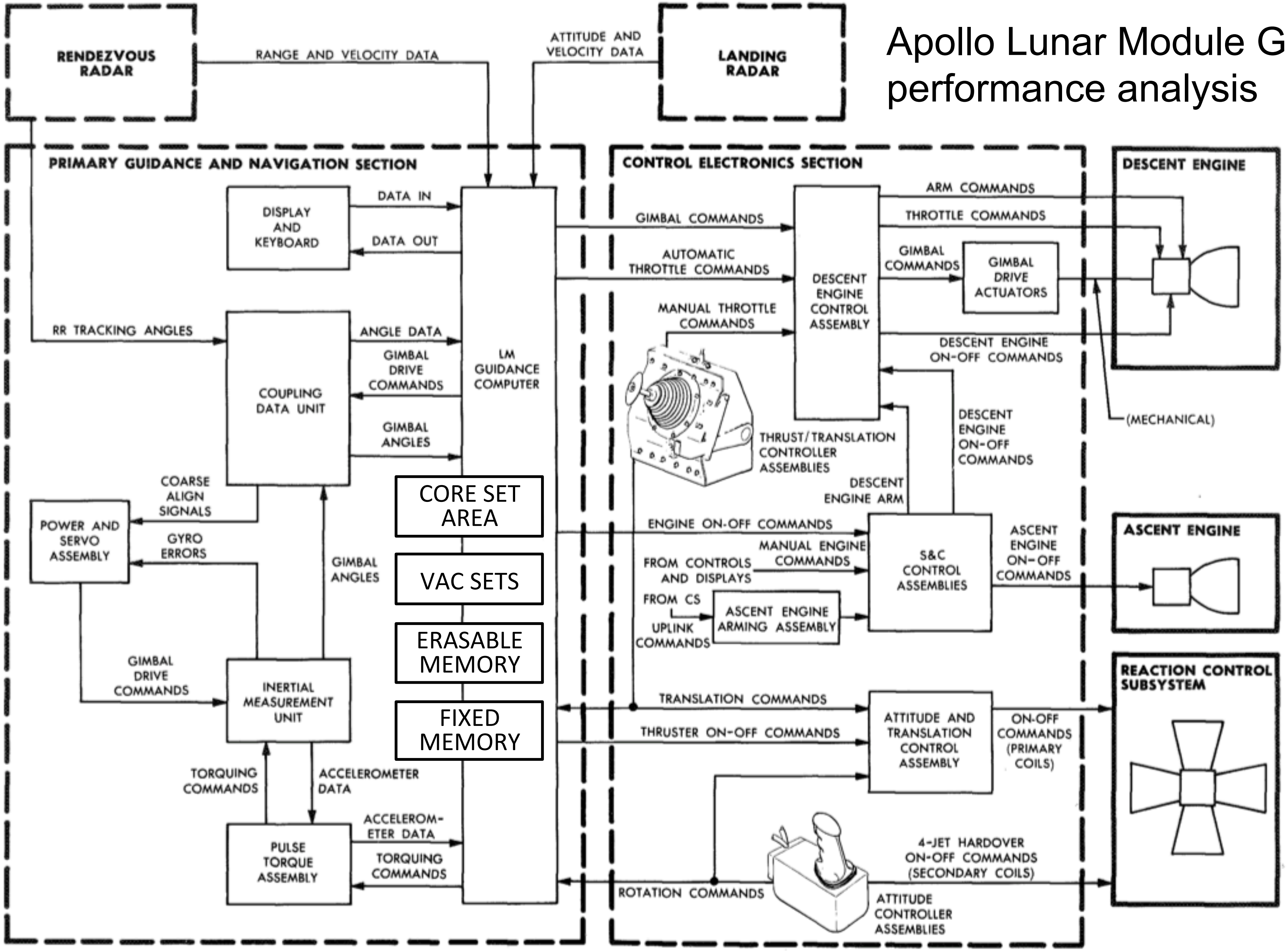
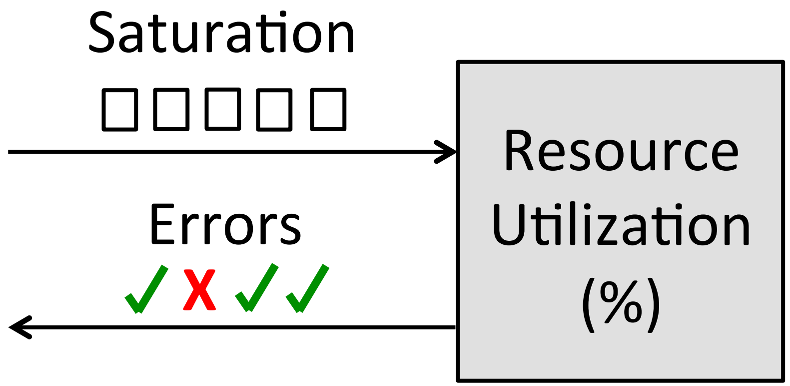


Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram

USE Method: Software

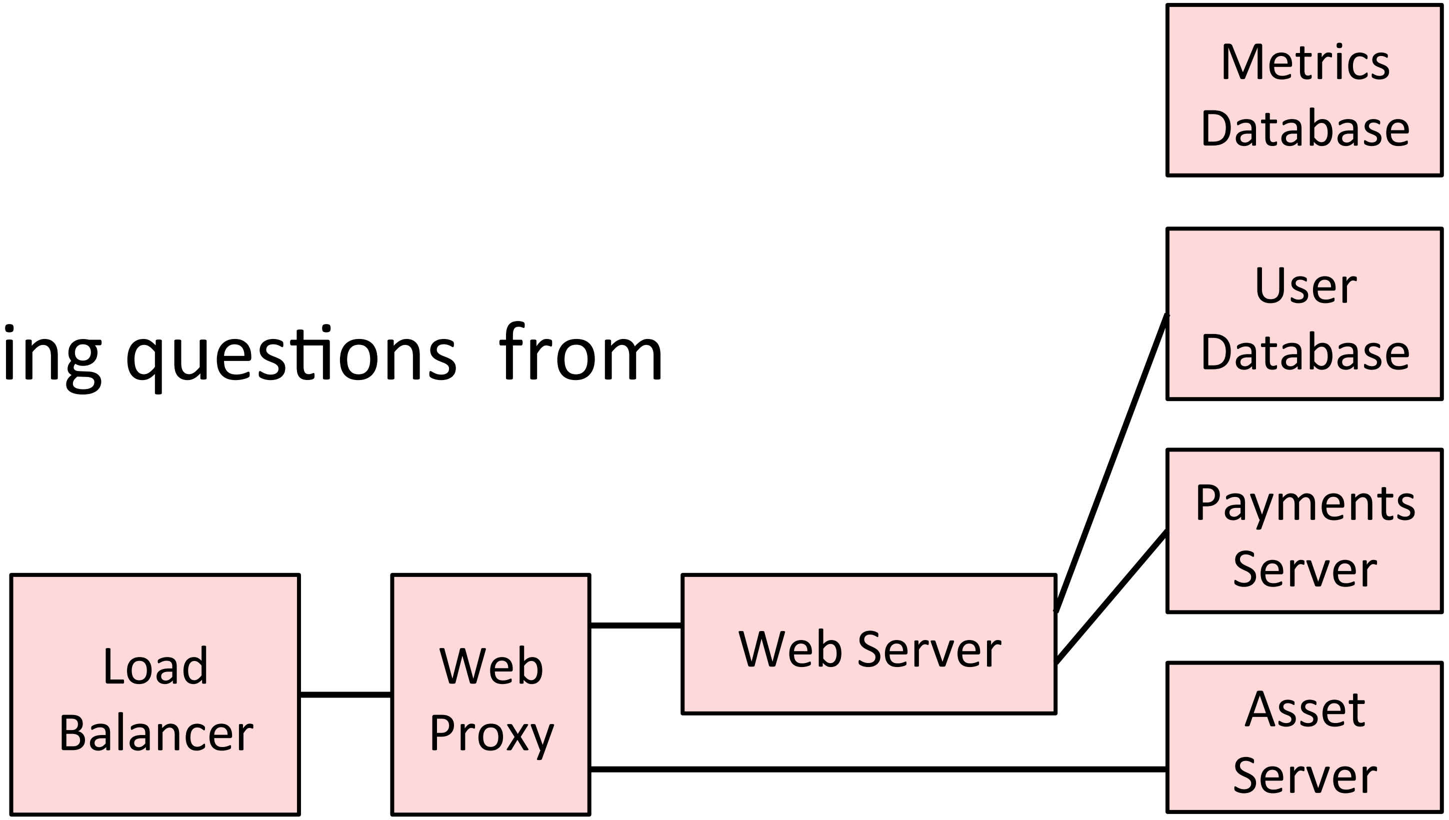
- USE method can also work for software resources
  - kernel or app internals, cloud environments
  - small scale (eg, locks) to large scale (apps). Eg:
- Mutex locks:
  - utilization → lock hold time
  - saturation → lock contention
  - errors → any errors
- Entire application:
  - utilization → percentage of worker threads busy
  - saturation → length of queued work
  - errors → request errors



RED Method

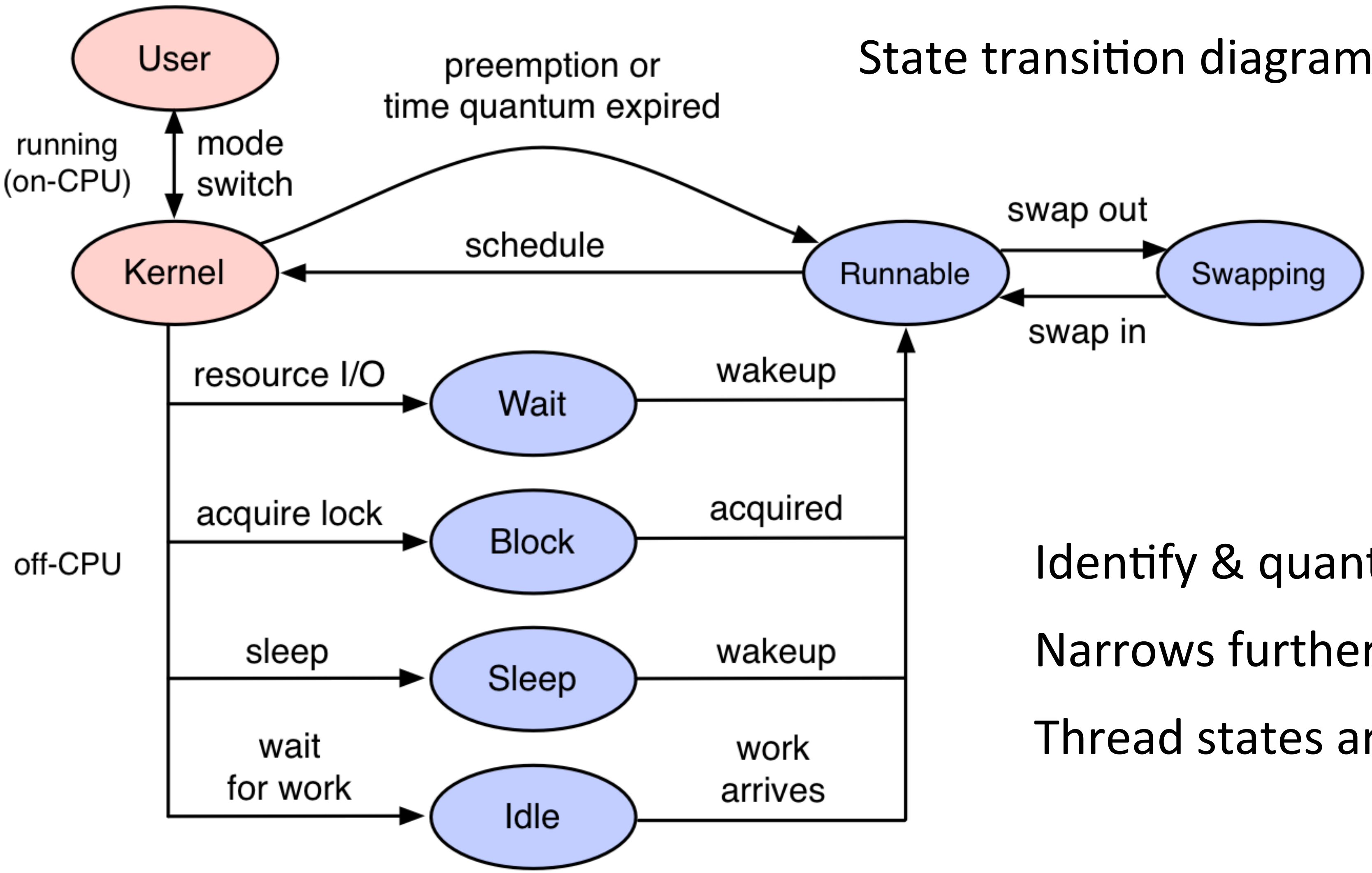
- For every service, check these are within SLO/A:
  1. **Request** rate
  2. **Error** rate
  3. **Duration** (distribution)

Another exercise in posing questions from functional diagrams



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

Thread State Analysis



State transition diagram

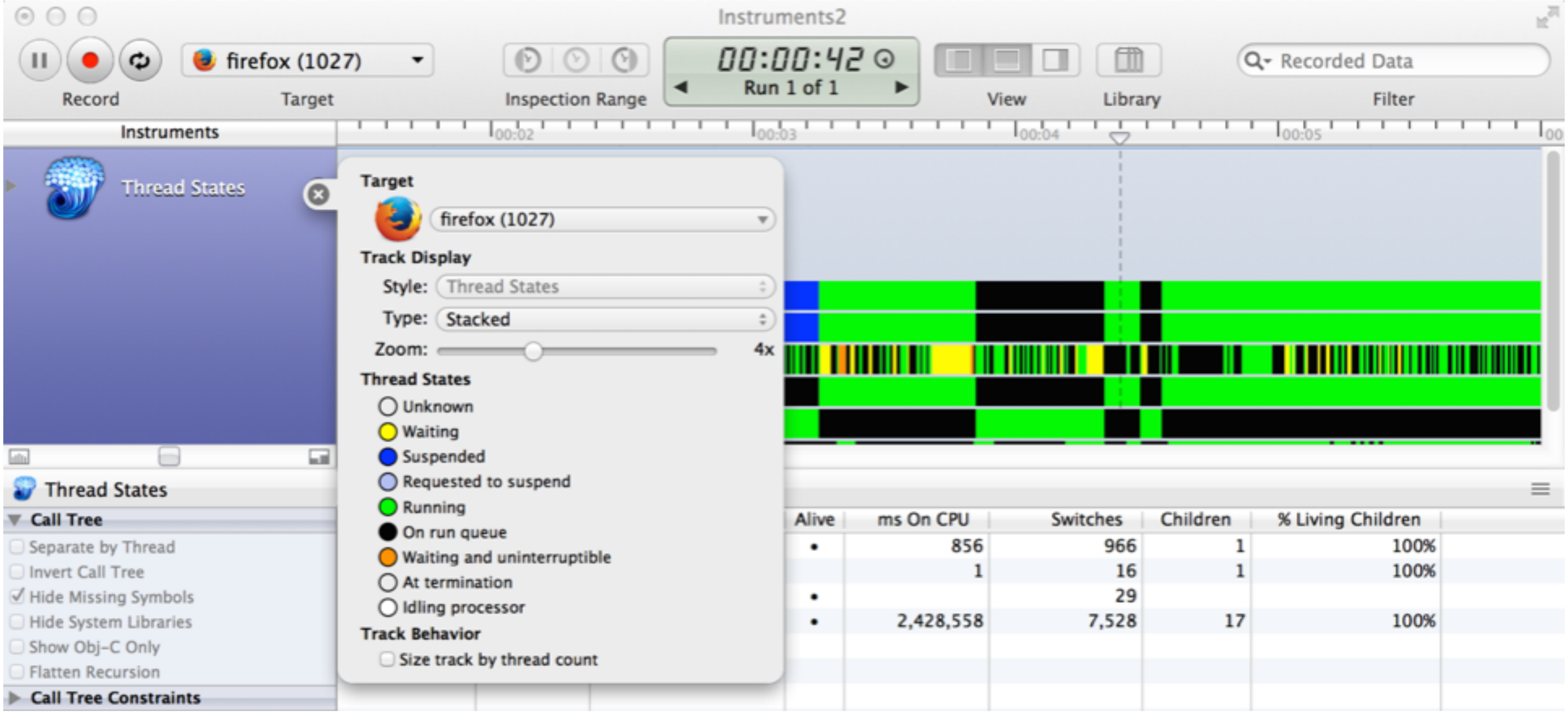
Identify & quantify time in states

Narrows further analysis to state

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

State Column (Job Status)		
RN	Run	Job is running or waiting to run.
RS	Residency	Job is waiting for residency. (The job has been swapped out of memory and is waiting to be swapped back in.)
BF	Buffers	Job is waiting for buffers (no space is available for I/O buffers).
SL	Sleep	Job is sleeping (SLEEP statement).
SR	Send/Receive	Job is sleeping and is a message receiver.
FP	File Processor	Job is waiting for file processing by the system (opening or closing a file, file search).
TT	Terminal	Job is waiting to perform output to a terminal.
HB	Hibernating	Job is detached and waiting to perform I/O to or from a terminal. (Someone must attach to the job before it can resume execution.)
KB	Keyboard	Job is waiting for input from a terminal.
^C	CTRL/C	Job is at command level, awaiting a command. (In other words, the keyboard monitor has displayed its prompt and is waiting for input.)
CR	Card Reader	Job is waiting for input from a card reader.
MT,MM, or MS	Magnetic Tape	Job is waiting for magnetic tape I/O.
LP	Line Printer	Job is waiting to perform line printer output.
DT	DECtape	Job is waiting for DECtape I/O.
DK,DM,DB, DP,DL,DR	Disk	Job is waiting to perform disk I/O.

TSA: Finding FreeBSD Thread States

```
# dtrace -ln sched:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
56622      sched            kernel                              none preempt
56627      sched            kernel                              none dequeue
56628      sched            kernel                              none enqueue
56631      sched            kernel                              none off-cpu
56632      sched            kernel                              none on-cpu
56633      sched            kernel                              none remain-cpu
56634      sched            kernel                              none surrender
56640      sched            kernel                              none sleep
56641      sched            kernel                              none wakeup
[…]
```

probes

```
struct thread {
[…]
        enum {
                TDS_INACTIVE = 0x0,
                TDS_INHIBITED,
                TDS_CAN_RUN,
                TDS_RUNQ,
                TDS_RUNNING
        } td_state;
[…]
#define KTDSTATE(td)                                                \
        (((td)->td_inhibitors & TDI_SLEEPING) != 0 ? "sleep"  :     \
        ((td)->td_inhibitors & TDI_SUSPENDED) != 0 ? "suspended" :  \
        ((td)->td_inhibitors & TDI_SWAPPED) != 0 ? "swapped" :      \
        ((td)->td_inhibitors & TDI_LOCK) != 0 ? "blocked" :         \
        ((td)->td_inhibitors & TDI_IWAIT) != 0 ? "iwait" : "yielding")
```

thread flags

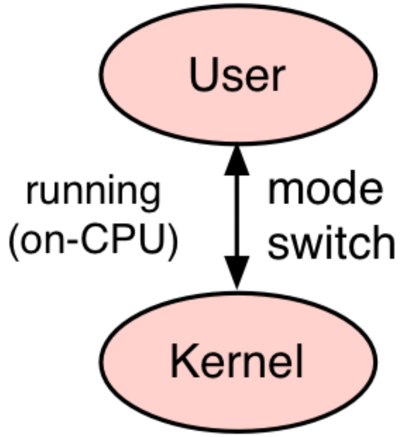
TSA: FreeBSD

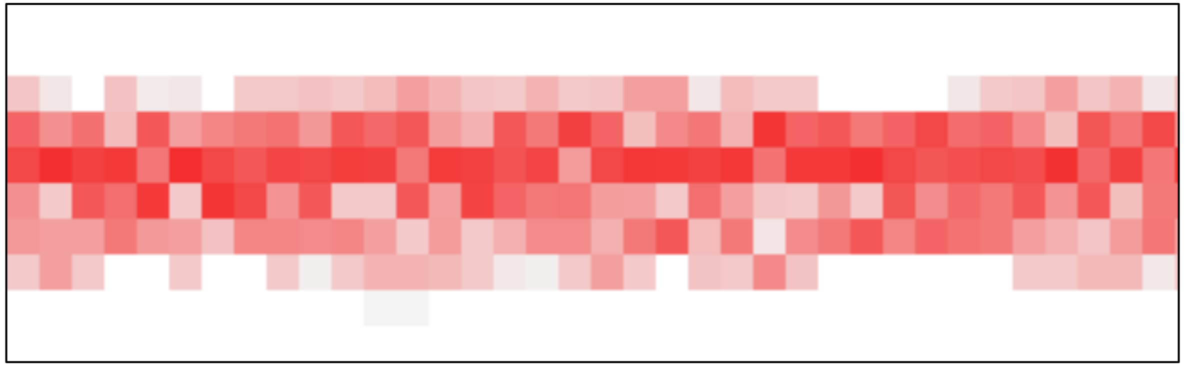
DTrace proof of concept

```
# ./tstates.d
Tracing scheduler events... Ctrl-C to end.
^C
Time (ms) per state:
COMM                PID         CPU    RUNQ     SLP     SUS     SWP     LCK     IWT     YLD
irq14: ata0         12            0       0       0       0       0       0       0       0
irq15: ata1         12            0       0       0       0       0       0    9009       0
swi4: clock (0)     12            0       0       0       0       0       0    9761       0
usbus0              14            0       0    8005       0       0       0       0       0
[...]
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
```

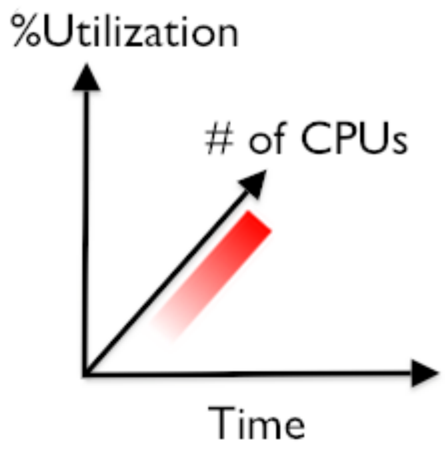
https://github.com/brendangregg/DTrace-tools/blob/master/sched/tstates.d

On-CPU Analysis





CPU Utilization
Heat Map



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

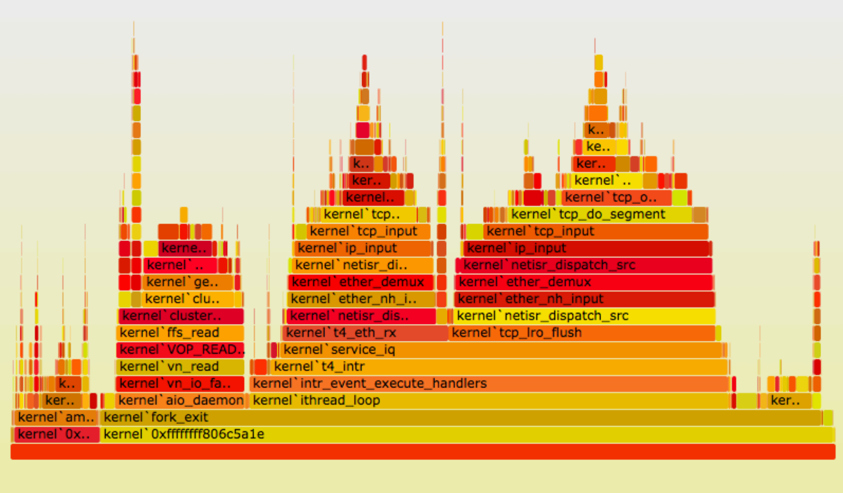
1. Take a CPU profile
2. Render it as a flame graph
3. Study largest "towers" first

Discovers issues by their CPU usage

- Directly: CPU consumers
- Indirectly: initialization of I/O, locks, times, ...

Narrows target of study

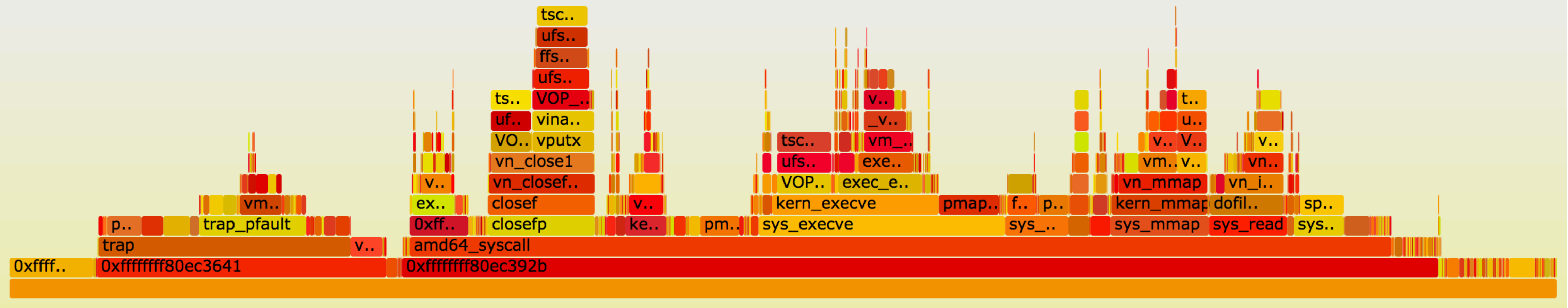
Flame Graph



CPU Flame Graphs: FreeBSD

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

```
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:

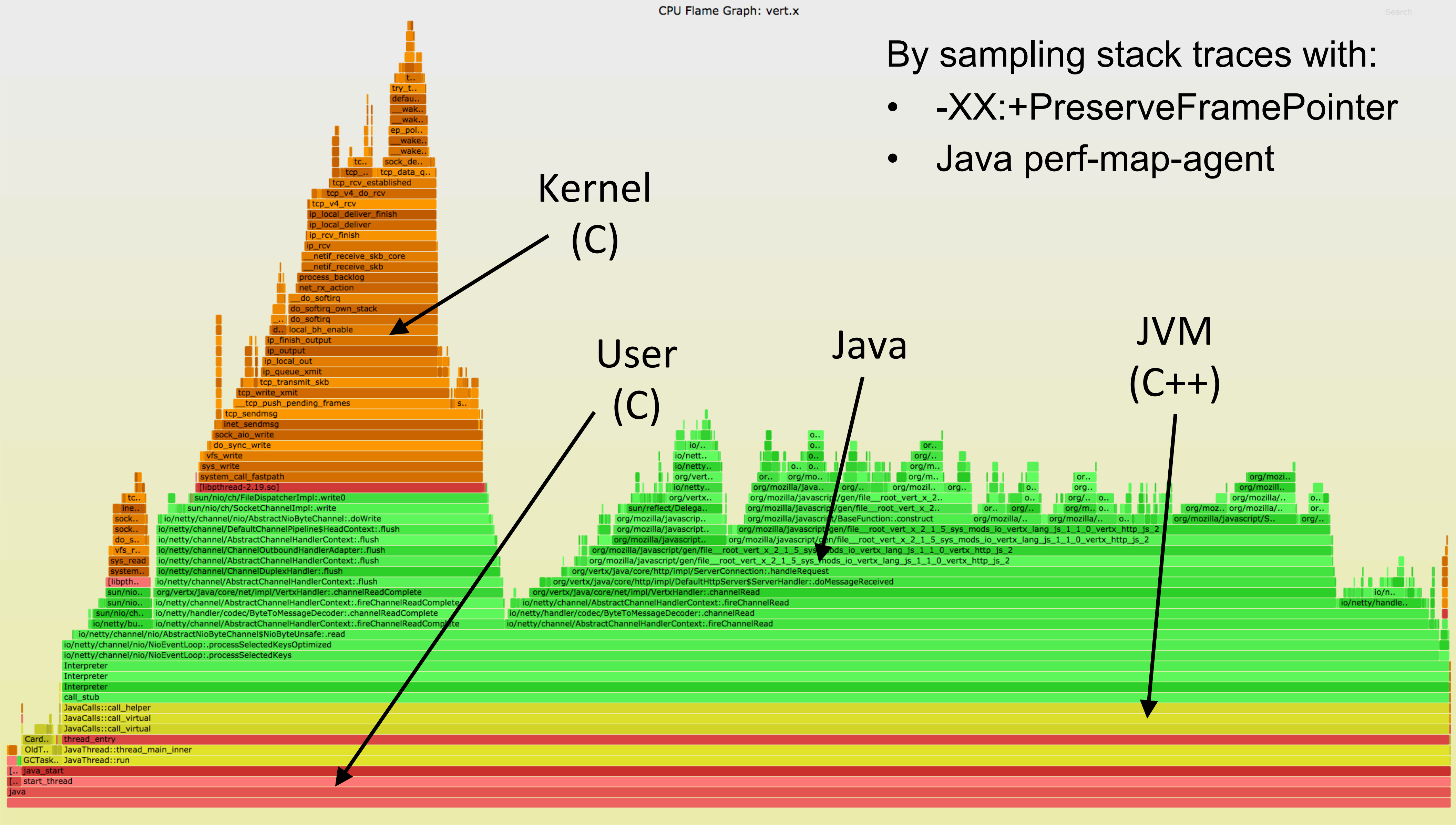
```
dtrace -x ustackframes=100 -x stackframes=100 -n '
    profile-99 { @[stack(), ustack(), execname] = sum(1); }
    tick-30s,END { printa("%k-%k%s\n%@d\n", @); trunc(@); exit(0); }' > stacks02
```

http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#DTrace

Java Mixed-Mode CPU Flame Graph

By sampling stack traces with:

- -XX:+PreserveFramePointer
- Java perf-map-agent



CPI Flame Graph: BSD

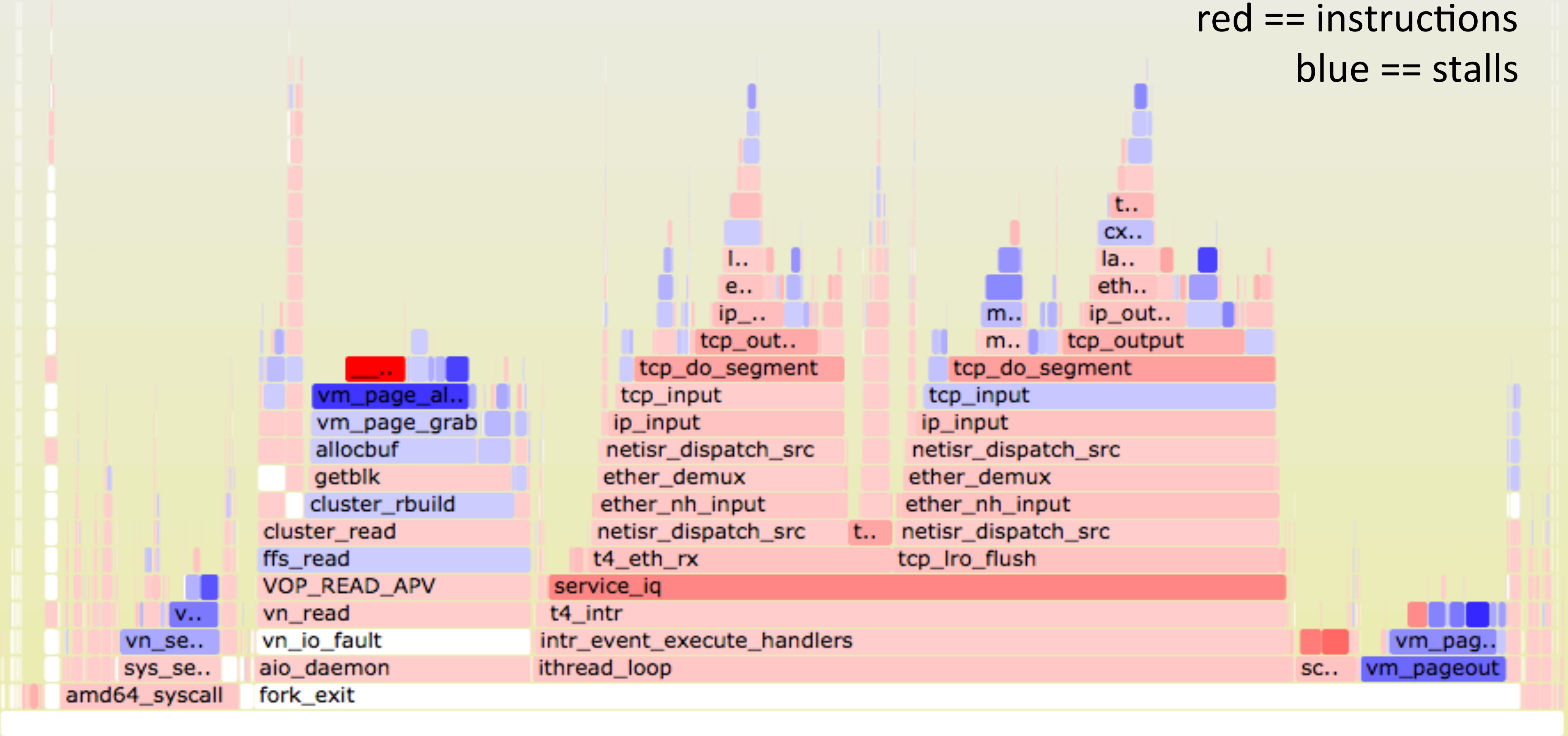
http://www.brendangregg.com/blog/2014-10-31/cpi-flame-graphs.html

A CPU flame graph (cycles) colored using instructions/stall profile data

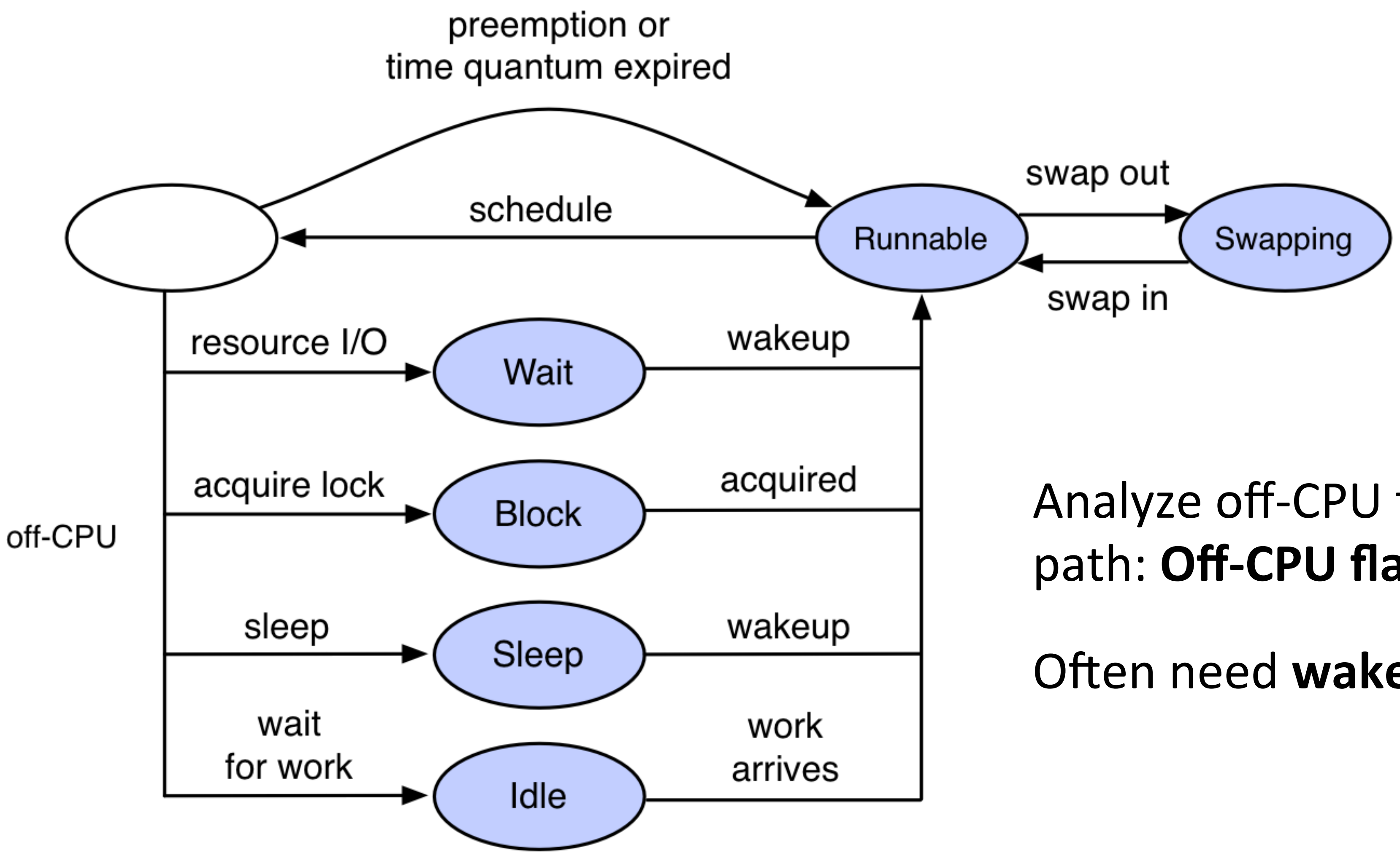
eg, using FreeBSD pmcstat:

red == instructions

blue == stalls



Off-CPU Analysis

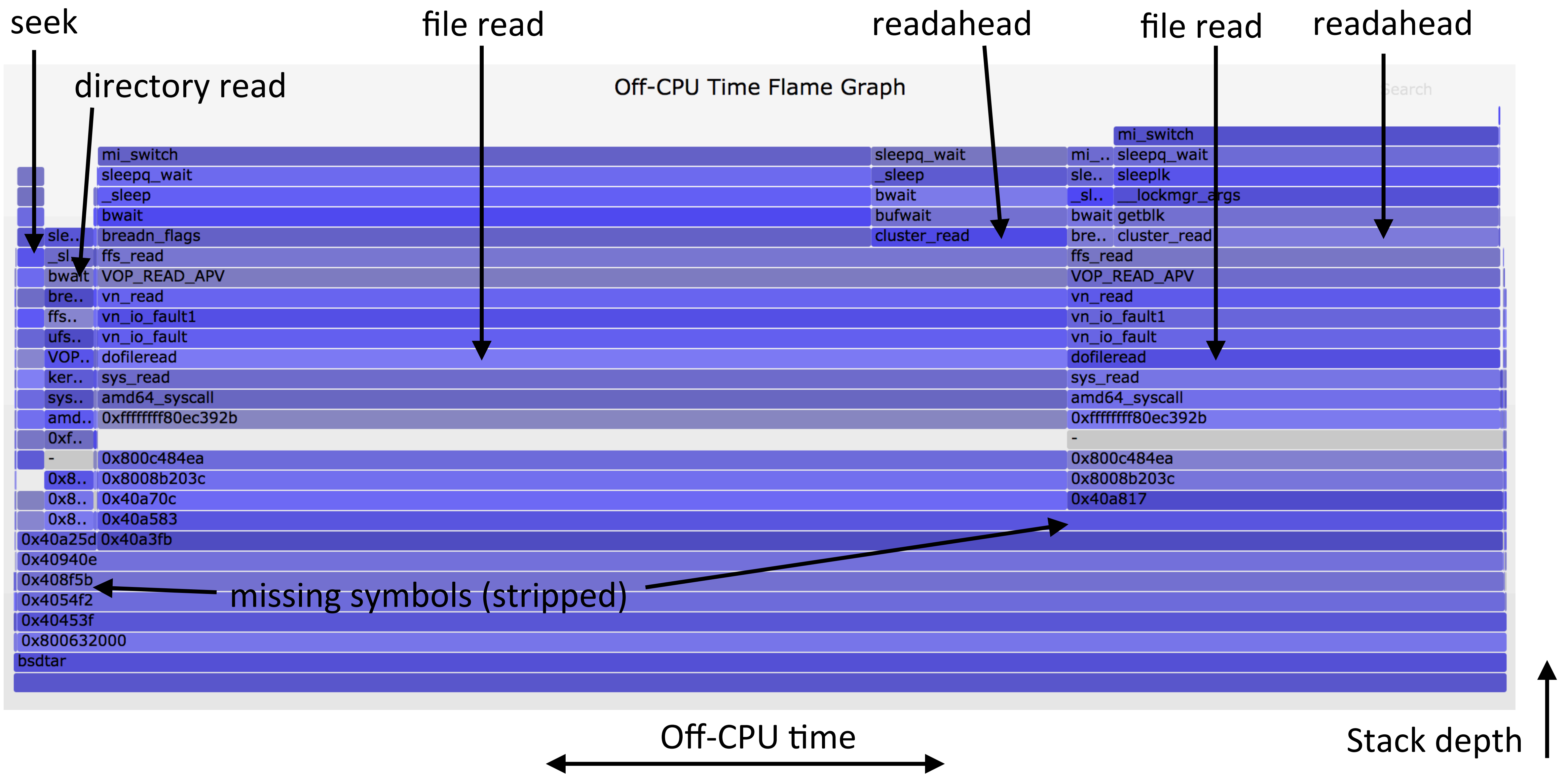


Analyze off-CPU time via blocking code path: **Off-CPU flame graph**

Often need **wakeup** code paths as well…

Off-CPU Time Flame Graph: FreeBSD

tar … > /dev/null



Off-CPU Profiling: FreeBSD

offcpu.d
Uses DTrace

```
#!/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\n%@d\n", @);
}
```

Change/remove as desired

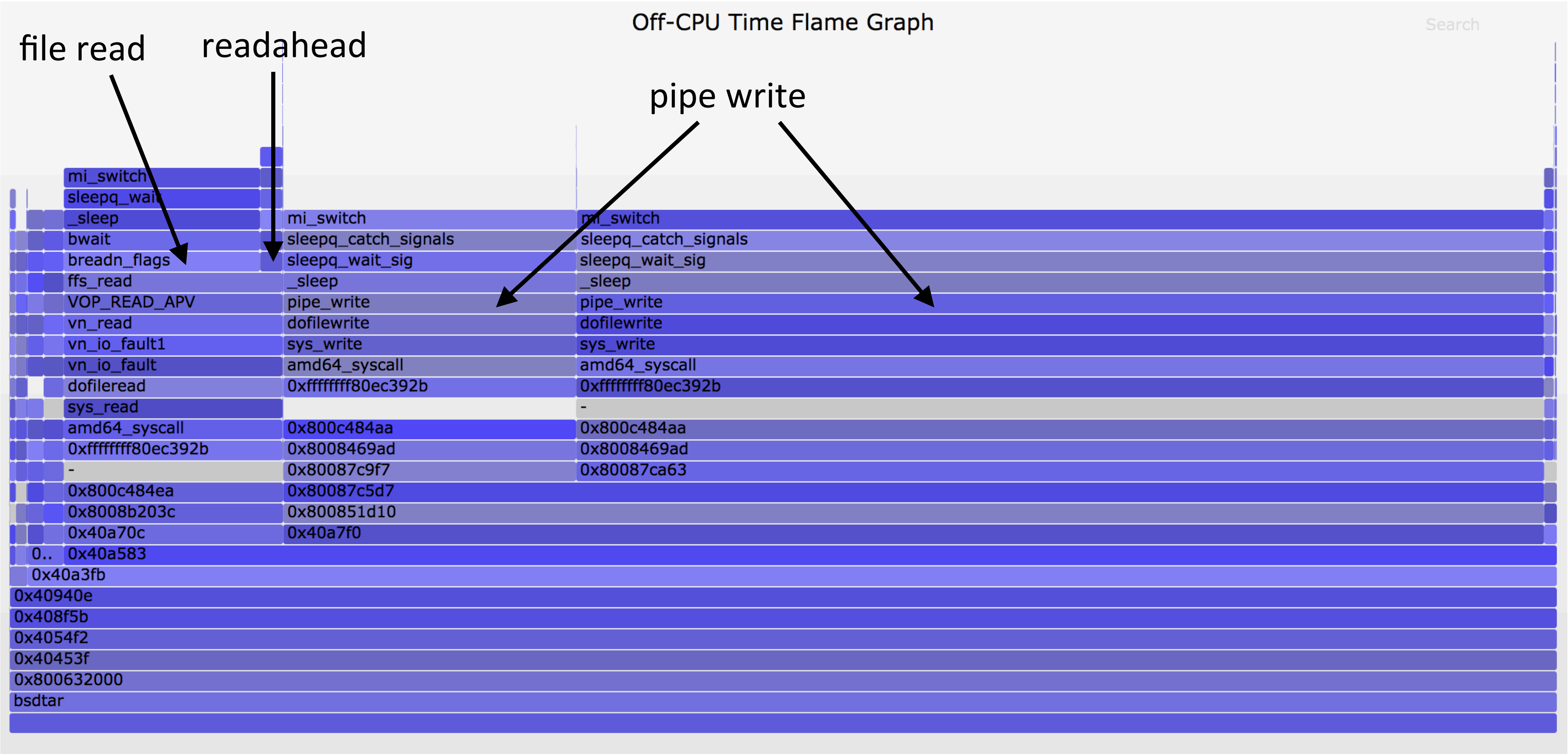
eg, add /curthread->td_state <= 1/ to exclude preempt, otherwise sees iCsw

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
```

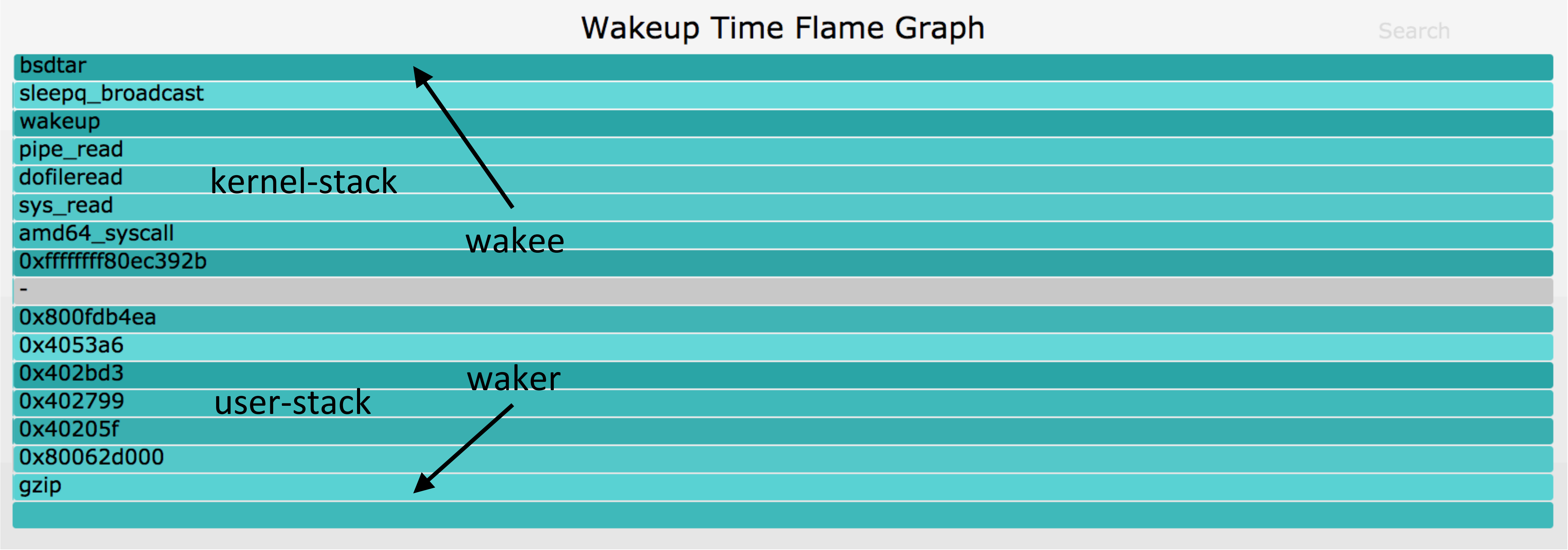
Off-CPU Time Flame Graph: FreeBSD

tar … | gzip



Wakeup Time Flame Graph: FreeBSD

Who did the wakeup:



Wakeup Profiling: FreeBSD

wakeup.d

Uses DTrace

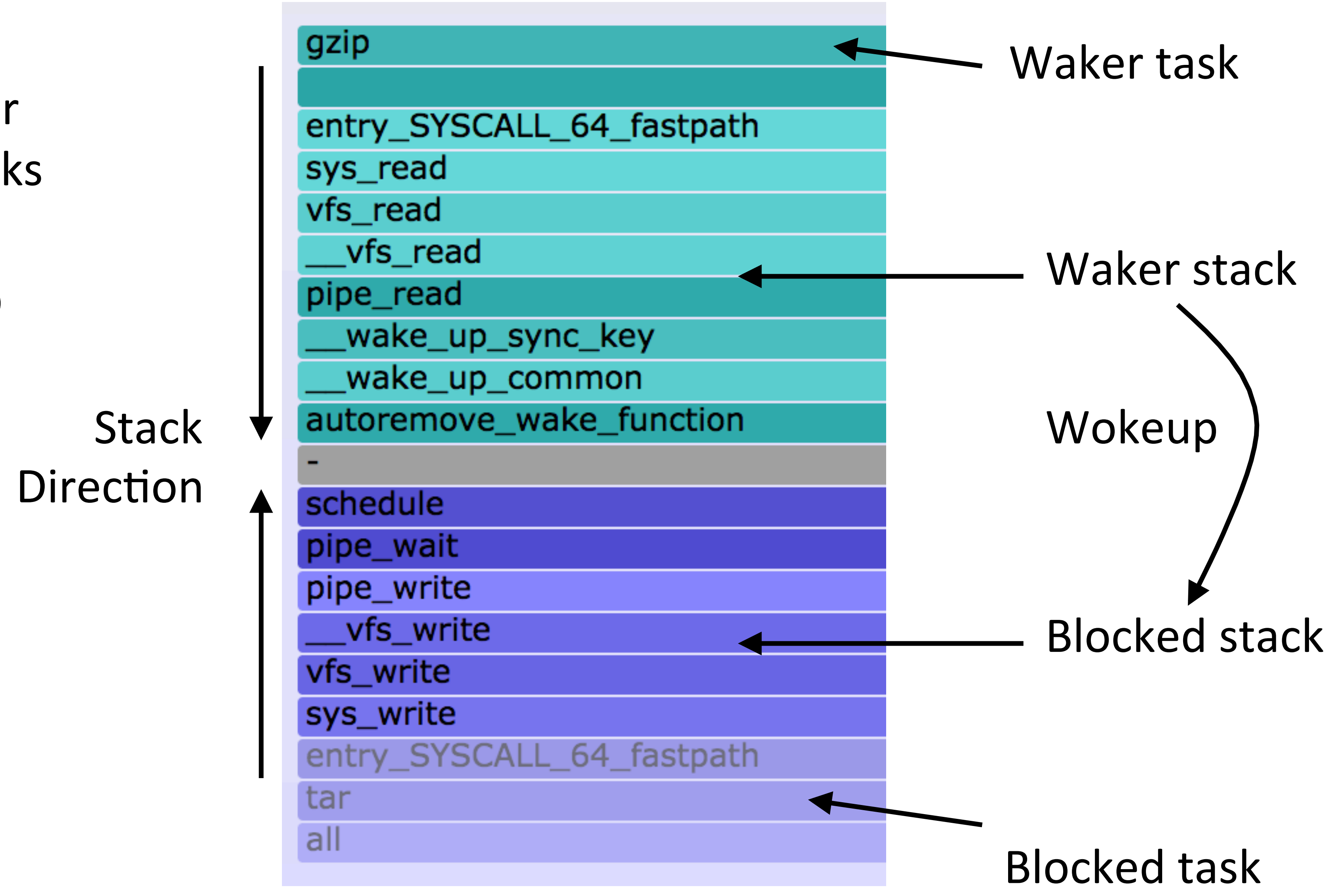
```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option ustackframes=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", @);
}
```

Change/remove as desired

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]
(001) jeq      #0x800          jt 2    jf 18
(002) ld       [26]
(003) jeq      #0x7f000001     jt 6    jf 4
(004) ld       [30]
(005) jeq      #0x7f000001     jt 6    jf 18
(006) ldb      [23]
(007) jeq      #0x84           jt 10   jf 8
(008) jeq      #0x6            jt 10   jf 9
(009) jeq      #0x11           jt 10   jf 18
(010) ldh      [20]
(011) jset     #0x1fff         jt 18   jf 12
(012) ldxb     4*([14]&0xf)
(013) ldh      [x + 14]
[...]
```

Optimizes packet filter performance

2 x 32-bit registers & scratch memory

User-defined bytecode executed by an in-kernel sandboxed virtual machine

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_RAW_INSN(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_RAW_INSN(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+

bcc/BPF front-end (C & Python)

```
# 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")
```

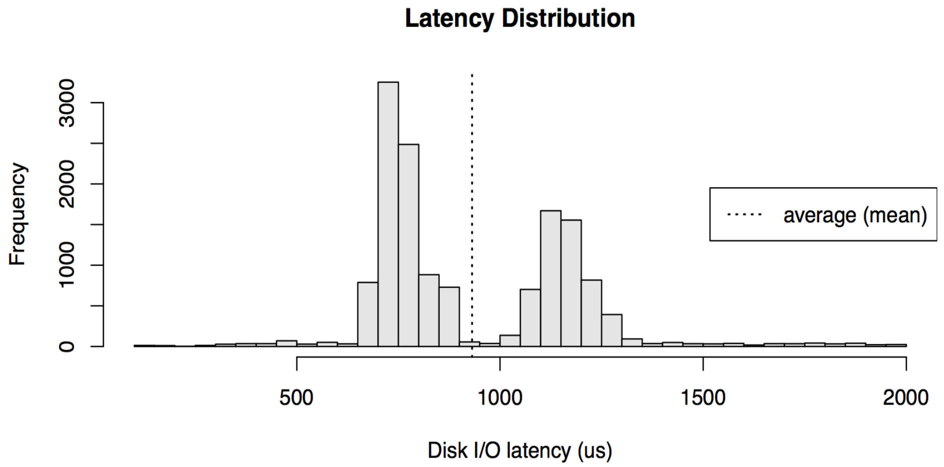
bcc examples/tracing/bitehist.py

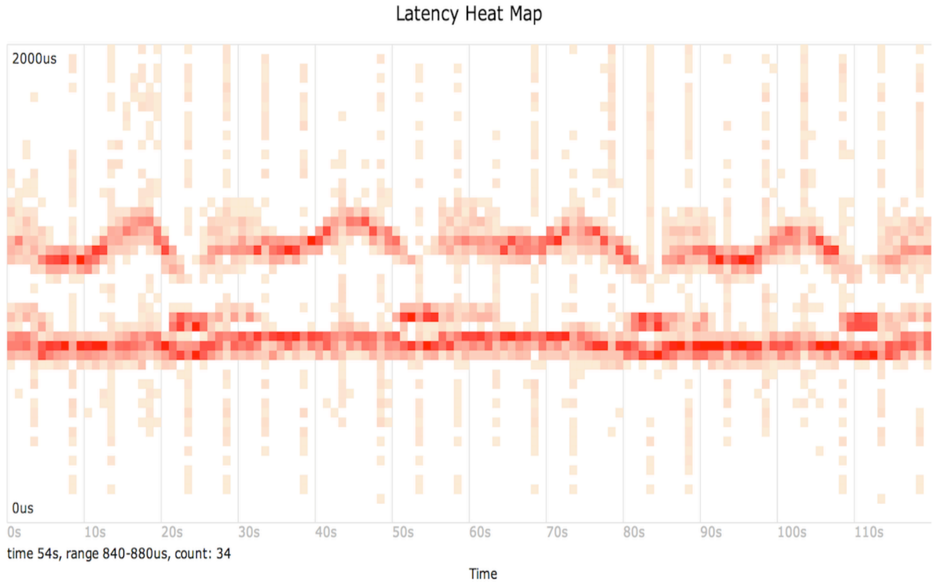
Latency Correlations

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



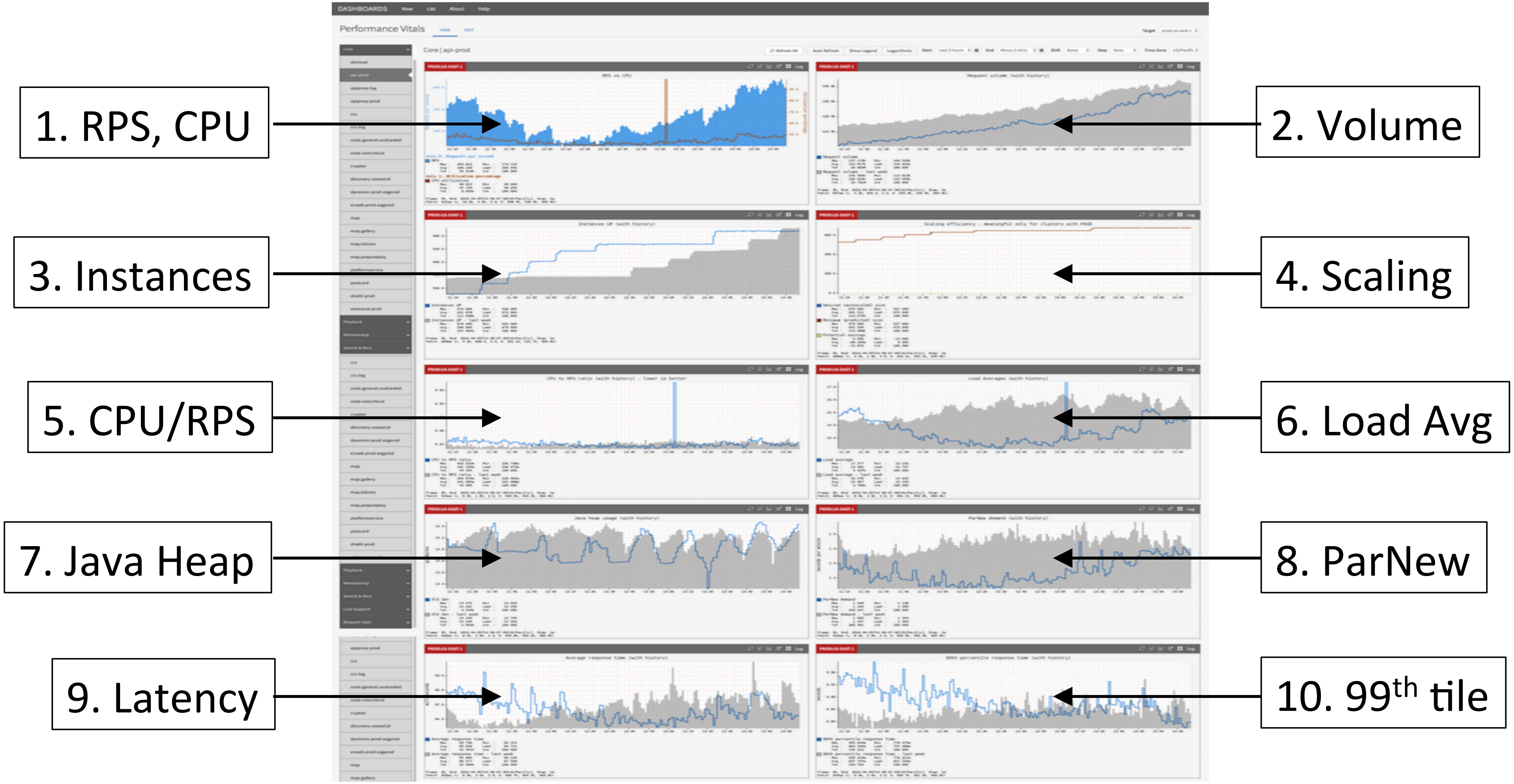


Checklists: eg, BSD Perf Analysis in 60s

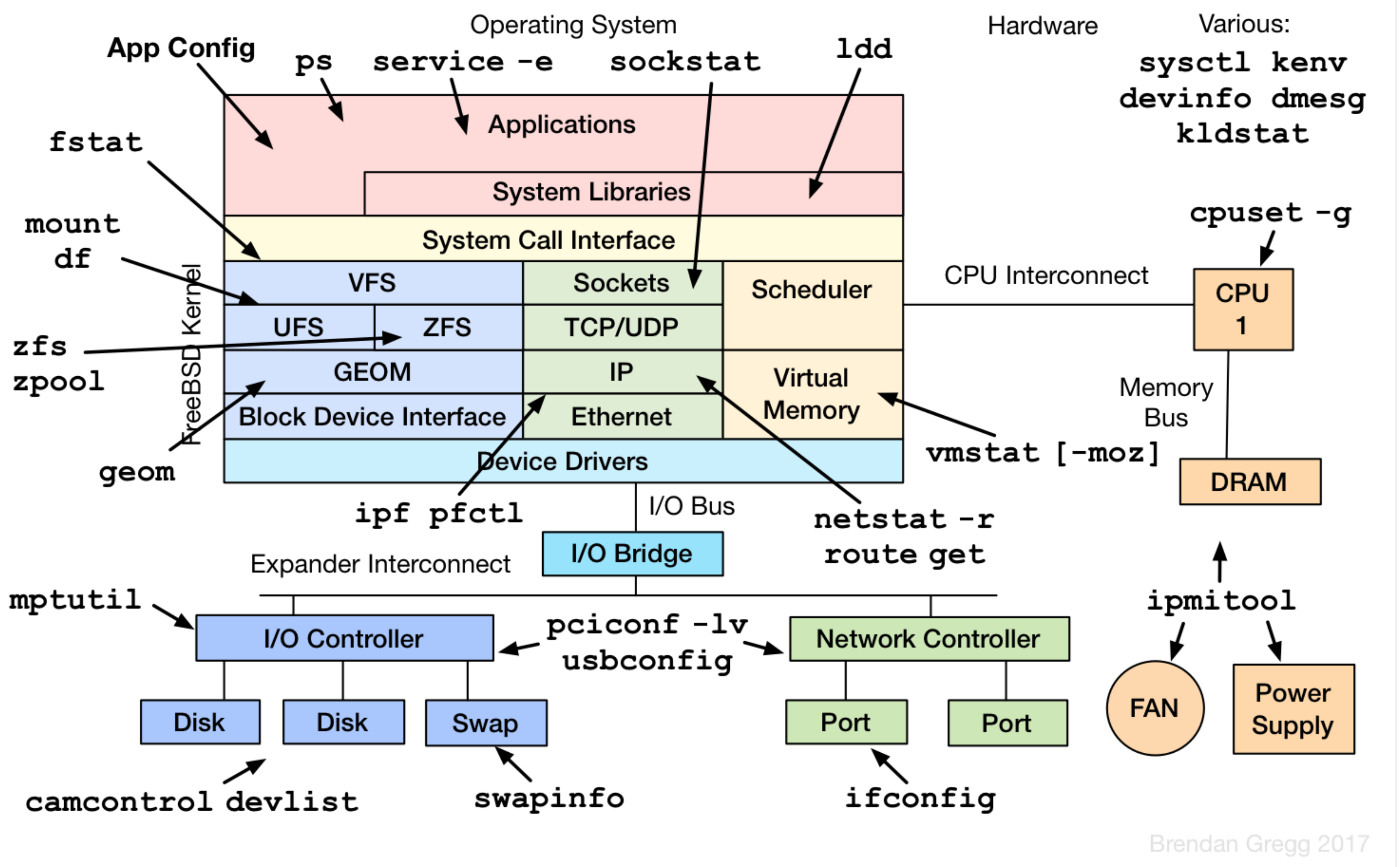
1. `uptime` → load averages
2. `dmesg -a | tail` → kernel errors
3. `vmstat 1` → overall stats by time
4. `vmstat -P` → CPU balance
5. `ps -auxw` → process usage
6. `iostat -xz 1` → disk I/O
7. `systat -ifstat` → network I/O
8. `systat -netstat` → TCP stats
9. `top` → process overview
10. `systat -vmstat`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

adapted from http://techblog.netflix.com/2015/11/linux-performance-analysis-in-60s.html

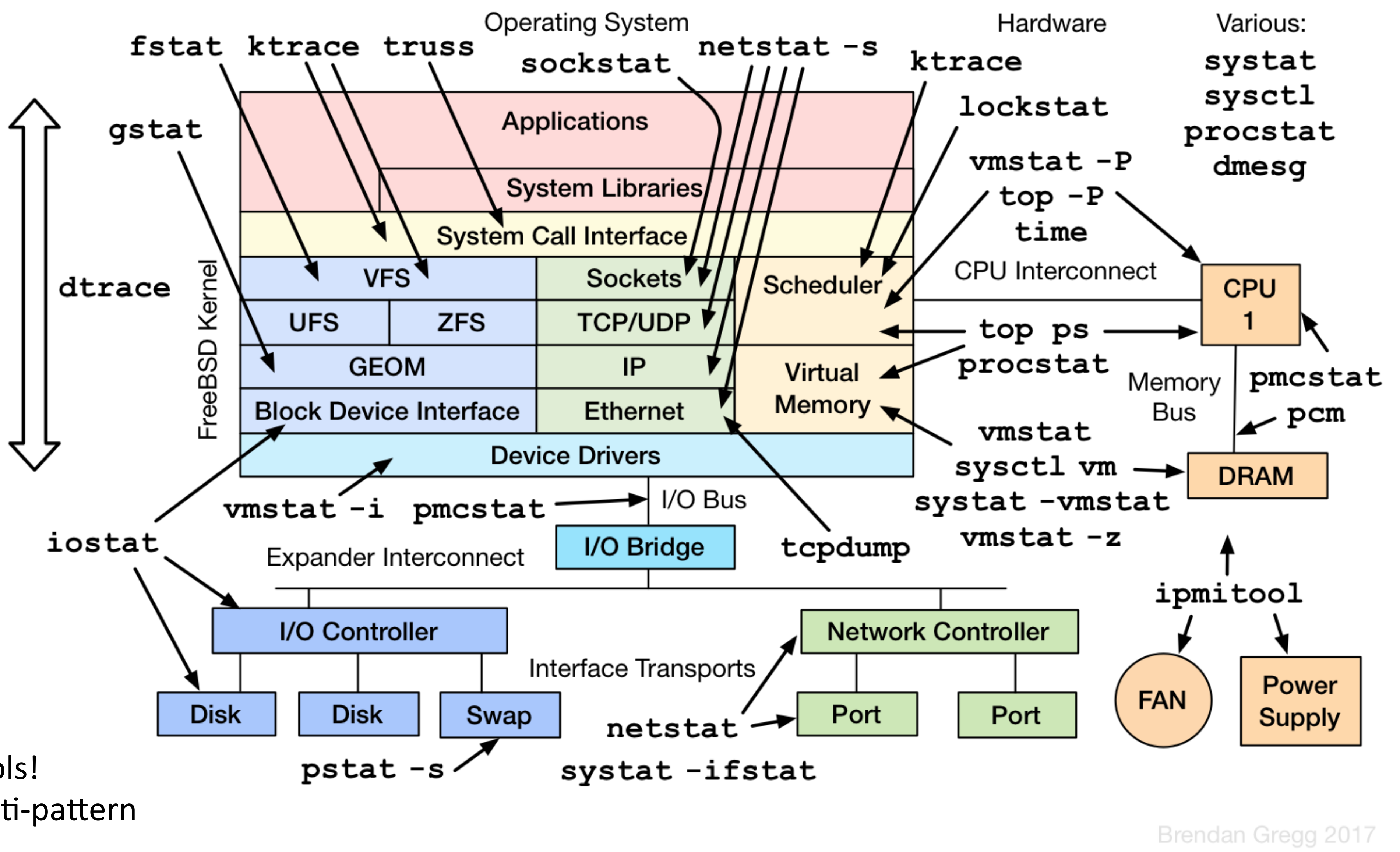
Checklists: eg, Netflix perfvitals Dashboard



Static Performance Tuning: FreeBSD



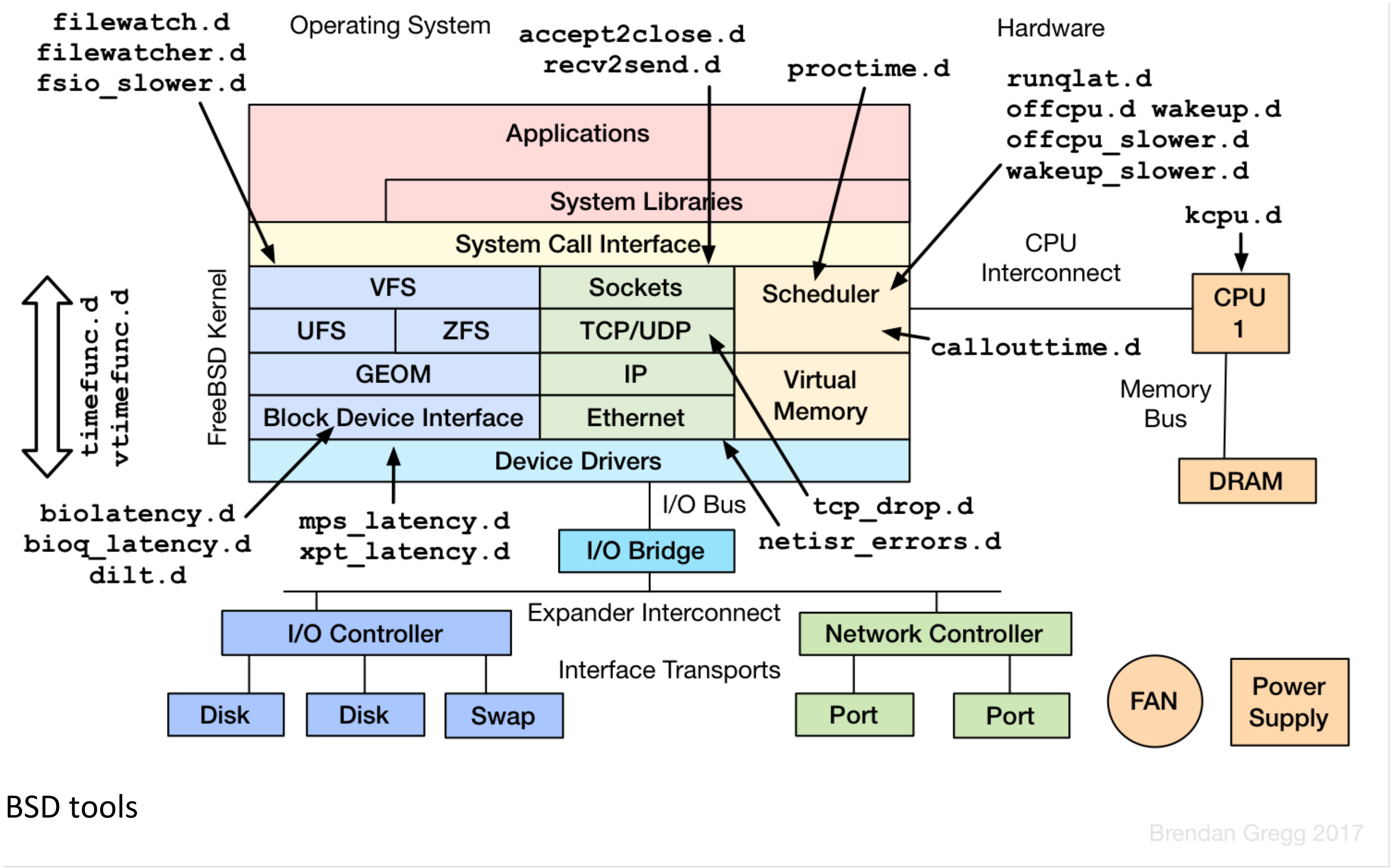
Tools-Based Method: FreeBSD



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

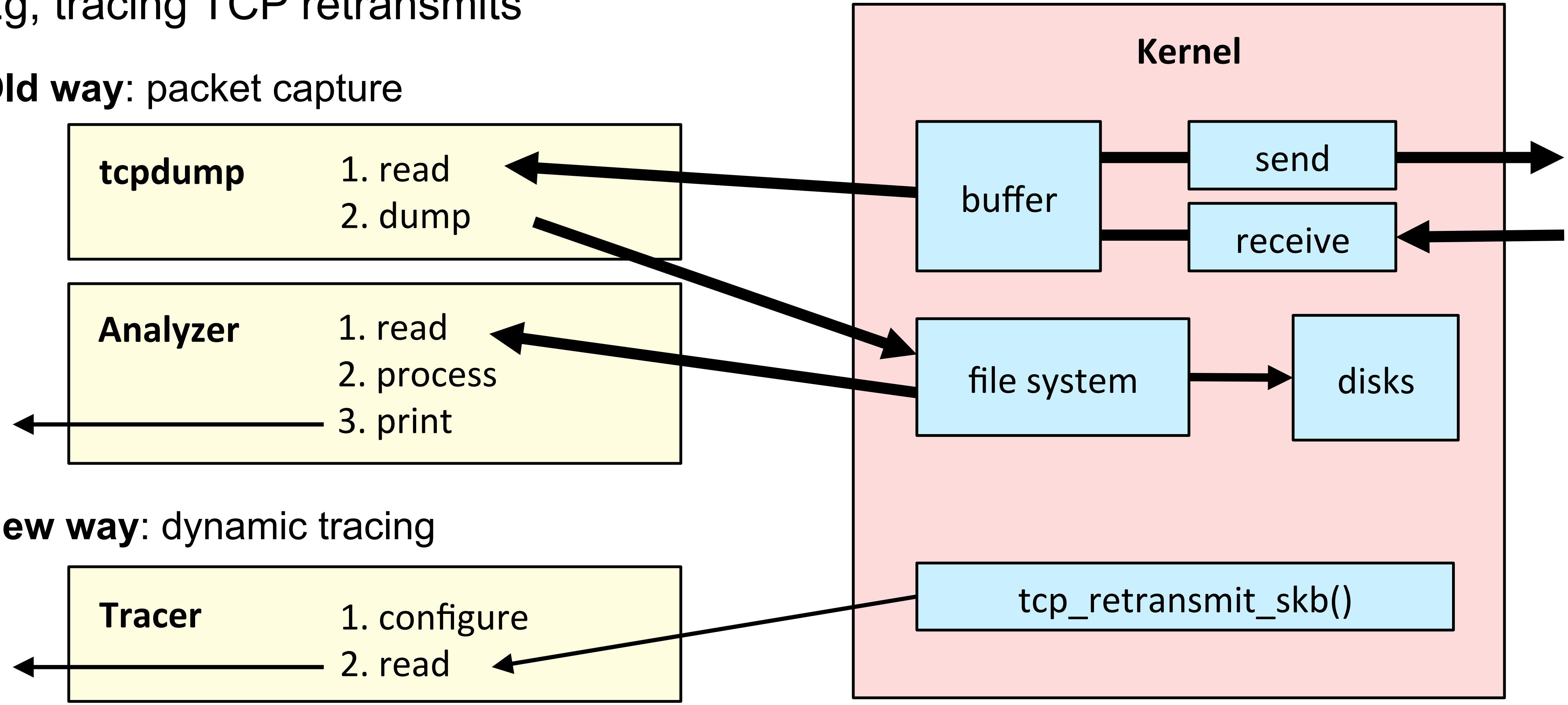
What You Can Do

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

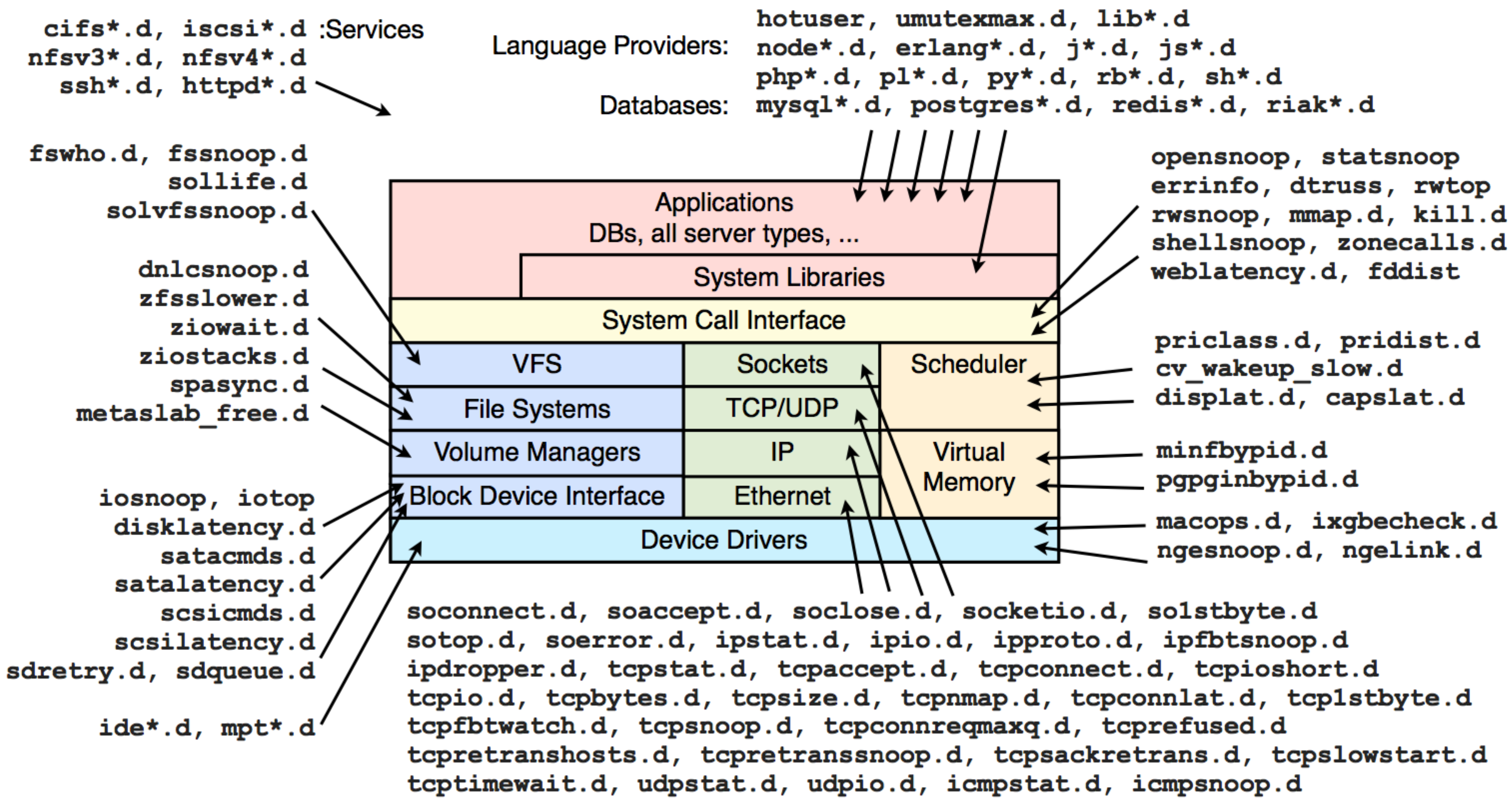
Dynamic Tracing: Efficient Metrics

Eg, tracing TCP retransmits



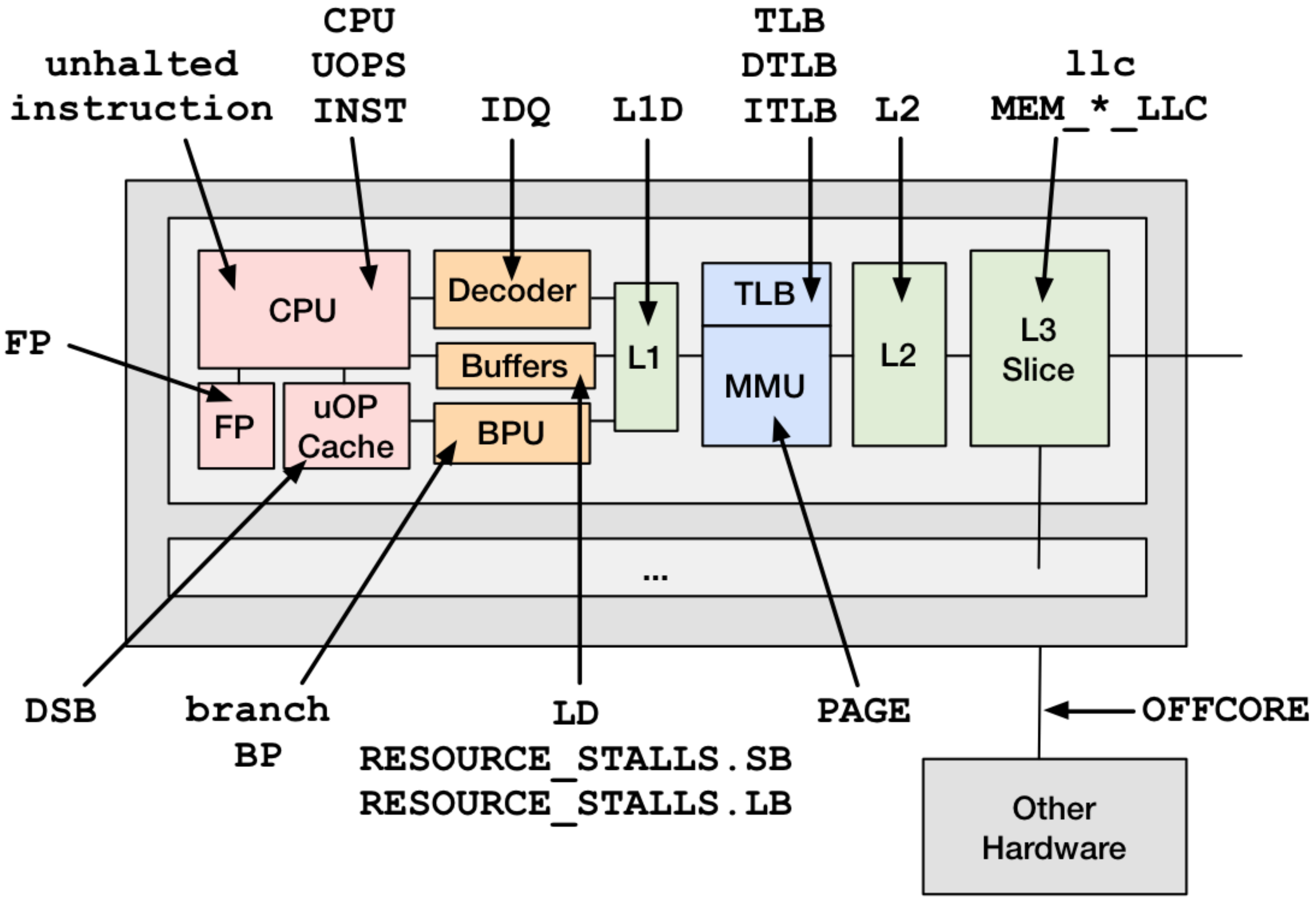
Dynamic Tracing: Instrument Most Software

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



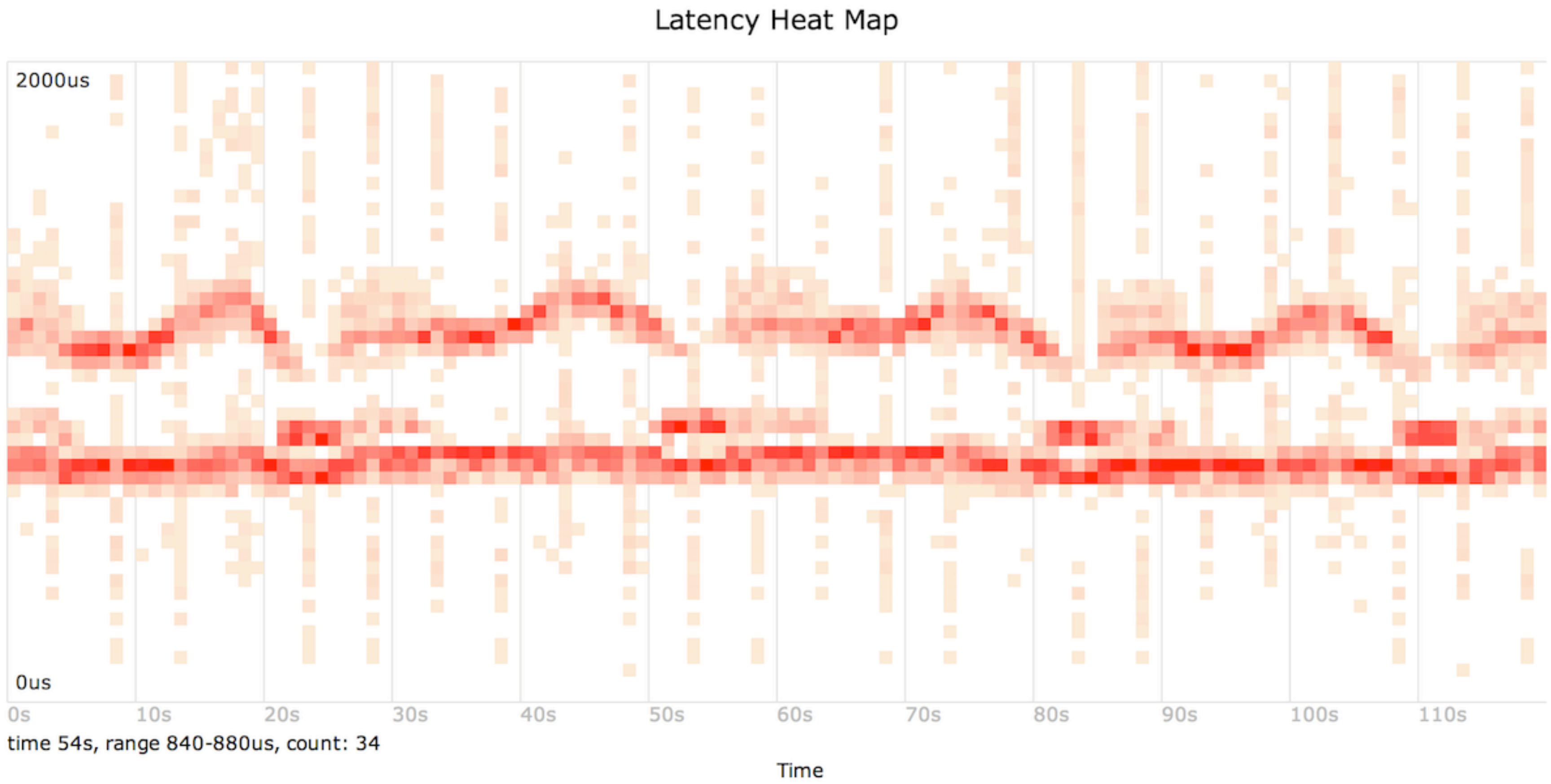
Performance Monitoring Counters

Eg, BSD PMC groups for Intel Sandy Bridge:



Visualizations

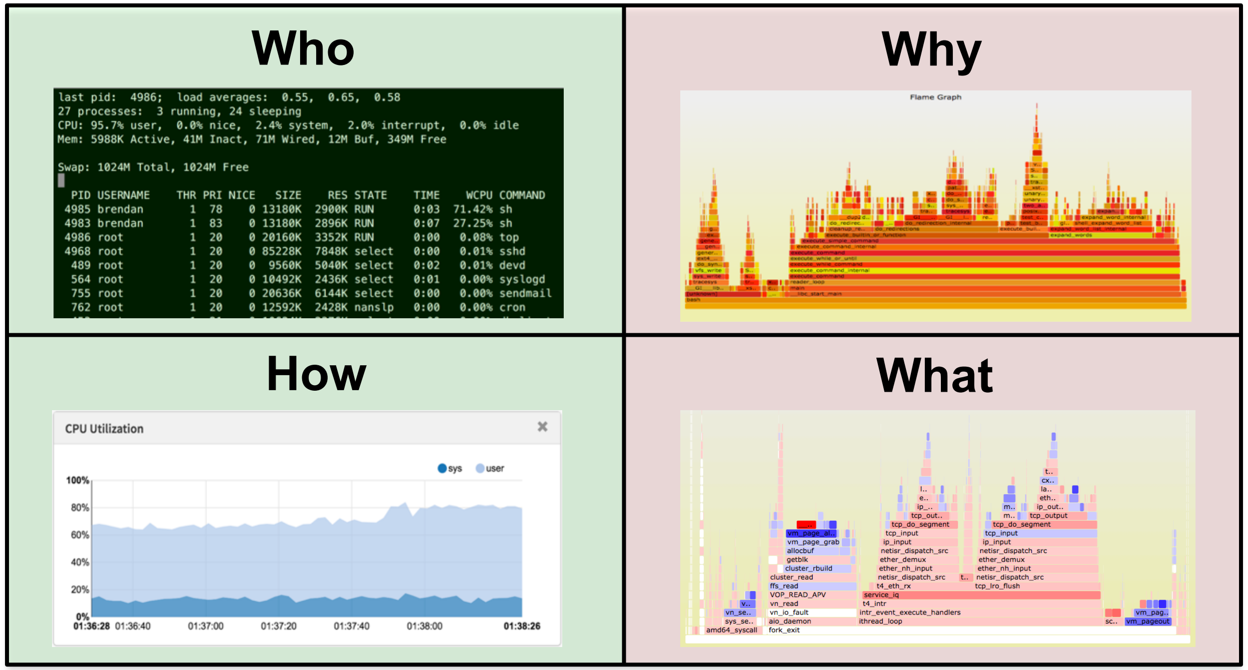
Eg, Disk I/O latency as a heat map, quantized in kernel:



Post processing the output of my iosnoop tool: www.brendangregg.com/HeatMaps/latency.html

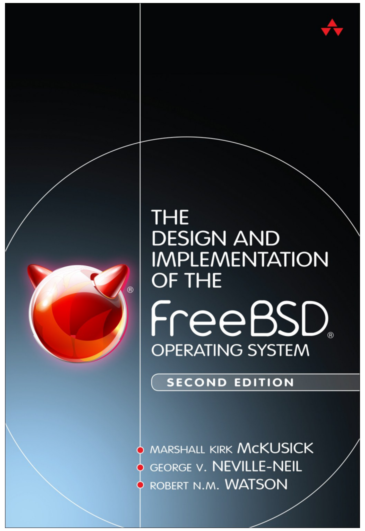
Summary

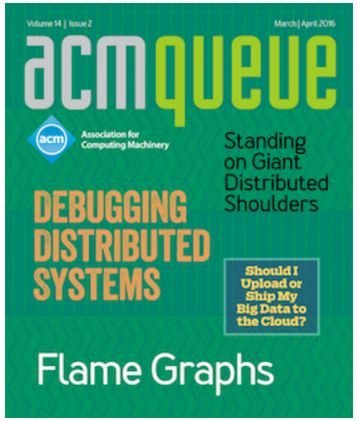
- It is the crystal ball age of performance observability
- What matters is the questions you want answered
- Methodologies are a great way to pose questions



References & Resources

- FreeBSD @ Netflix:
  - https://openconnect.itp.netflix.com/
  - http://people.freebsd.org/~scottl/Netflix-BSDCan-20130515.pdf
  - http://www.youtube.com/watch?v=FL5U4wr86L4
- USE Method
  - http://queue.acm.org/detail.cfm?id=2413037
  - http://www.brendangregg.com/usemethod.html
- TSA Method
  - http://www.brendangregg.com/tsamethod.html
- Off-CPU Analysis
  - http://www.brendangregg.com/offcpuanalysis.html
  - http://www.brendangregg.com/blog/2016-01-20/ebpf-offcpu-flame-graph.html
  - http://www.brendangregg.com/blog/2016-02-05/ebpf-chaingraph-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
  - http://www.brendangregg.com/methodology.html
  - 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
  - http://techblog.netflix.com/2015/07/java-in-flames.html
- Latency Heat Maps
  - http://queue.acm.org/detail.cfm?id=1809426
  - http://www.brendangregg.com/HeatMaps/latency.html
- ARPA Network: http://www.computerhistory.org/internethistory/1960s
- RSTS/E System User's Guide, 1985, page 4-5
- DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD, Prentice Hall 2011
- Apollo: http://www.hq.nasa.gov/office/pao/History/alsj/a11 http://www.hq.nasa.gov/alsj/alsj-LMdocs.html

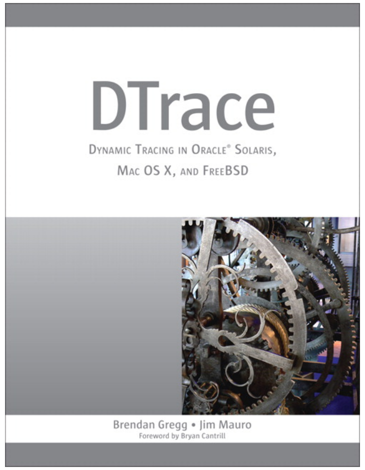


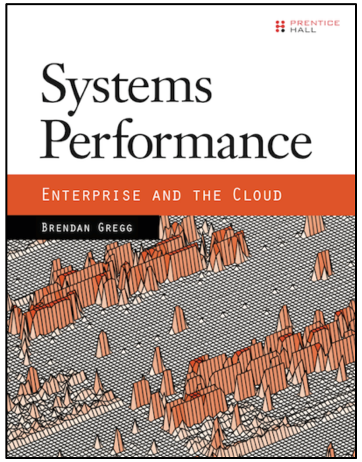


EuroBSDcon 2017

Thank You

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





NETFLIX