ACM Applicative 2016

Jun, 2016

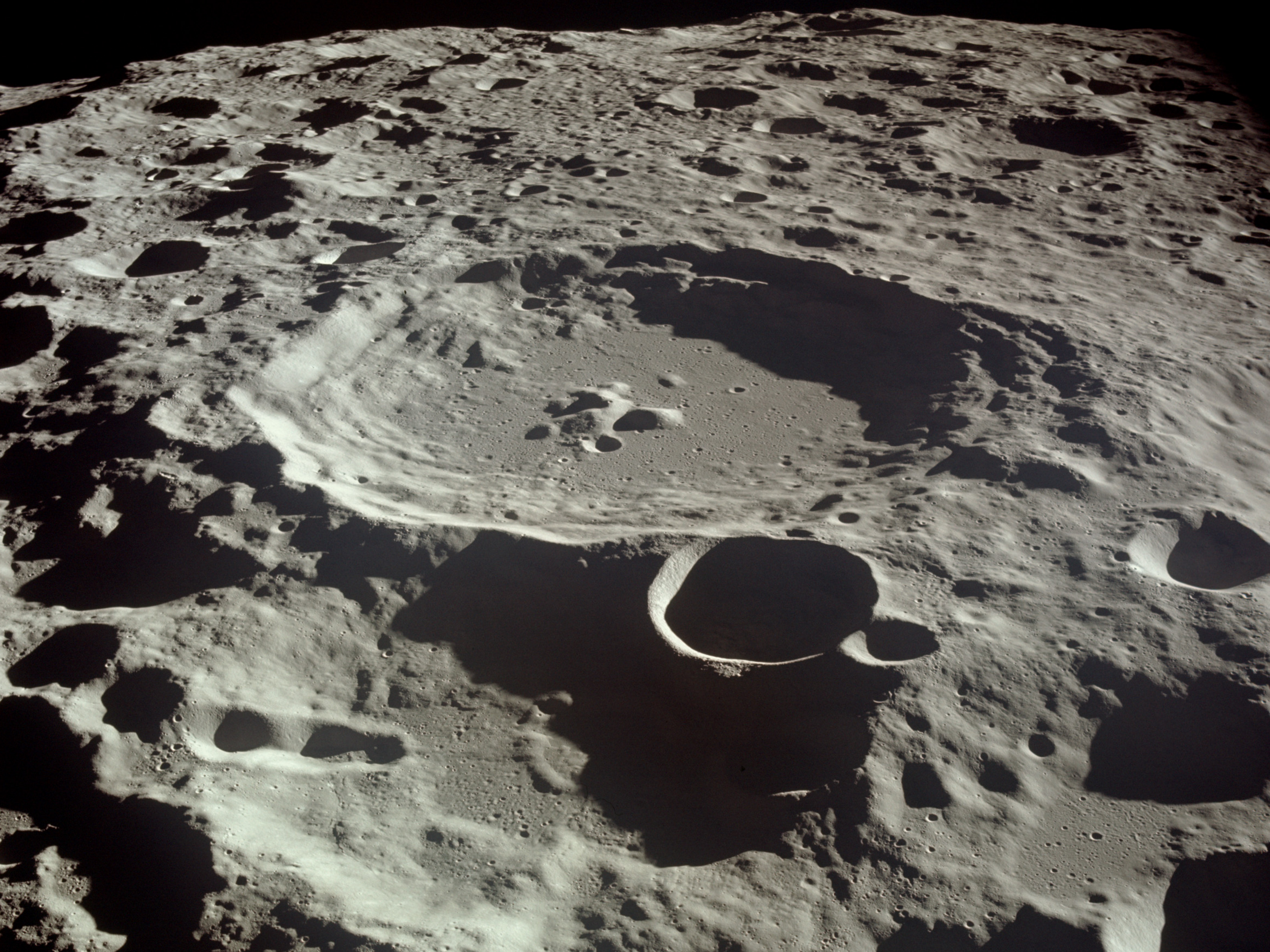
System Methodology

Holistic Performance Analysis on Modern Systems

Brendan Gregg

Senior Performance Architect

NETFLIX



Apollo LMGC performance analysis

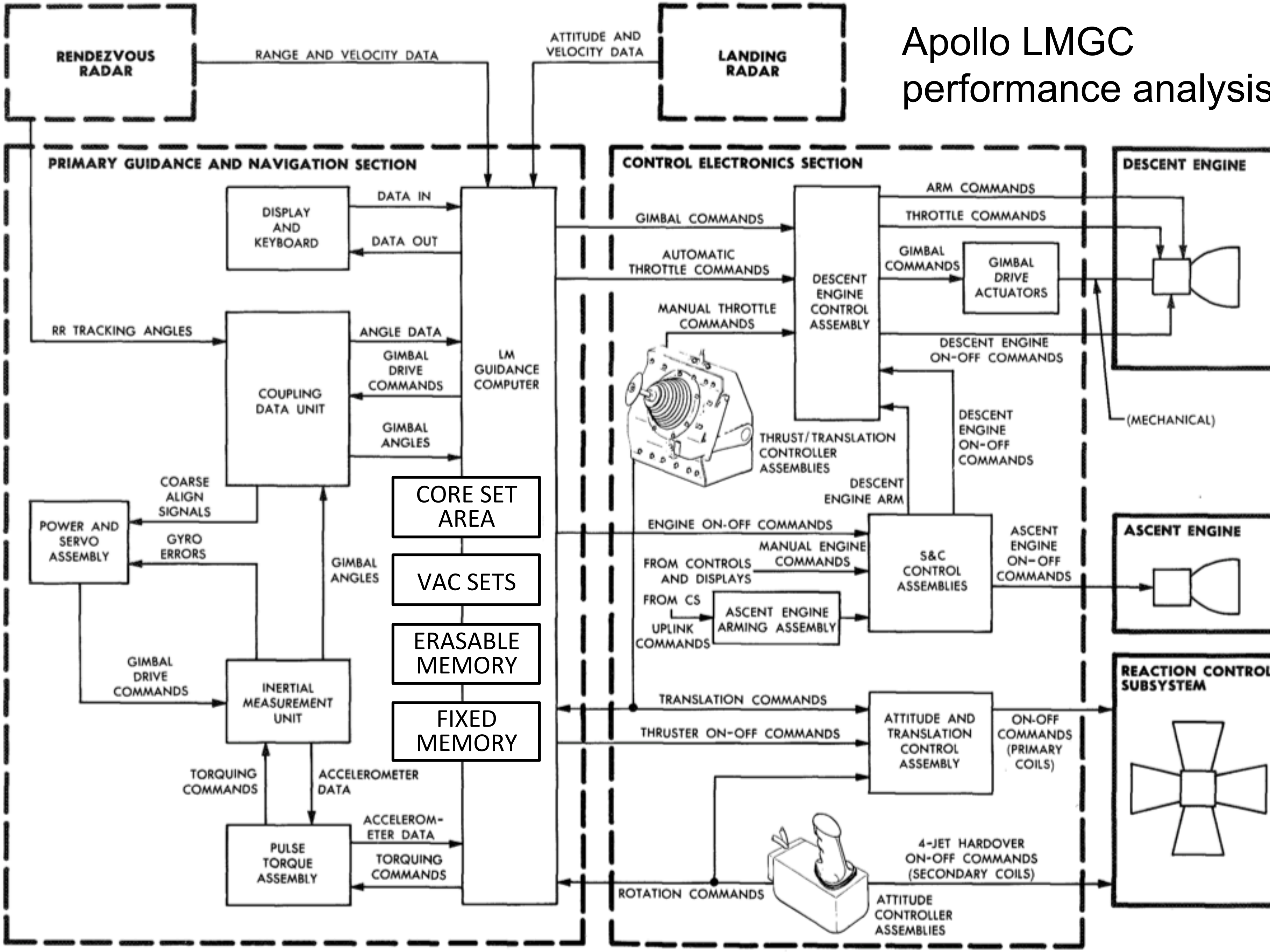
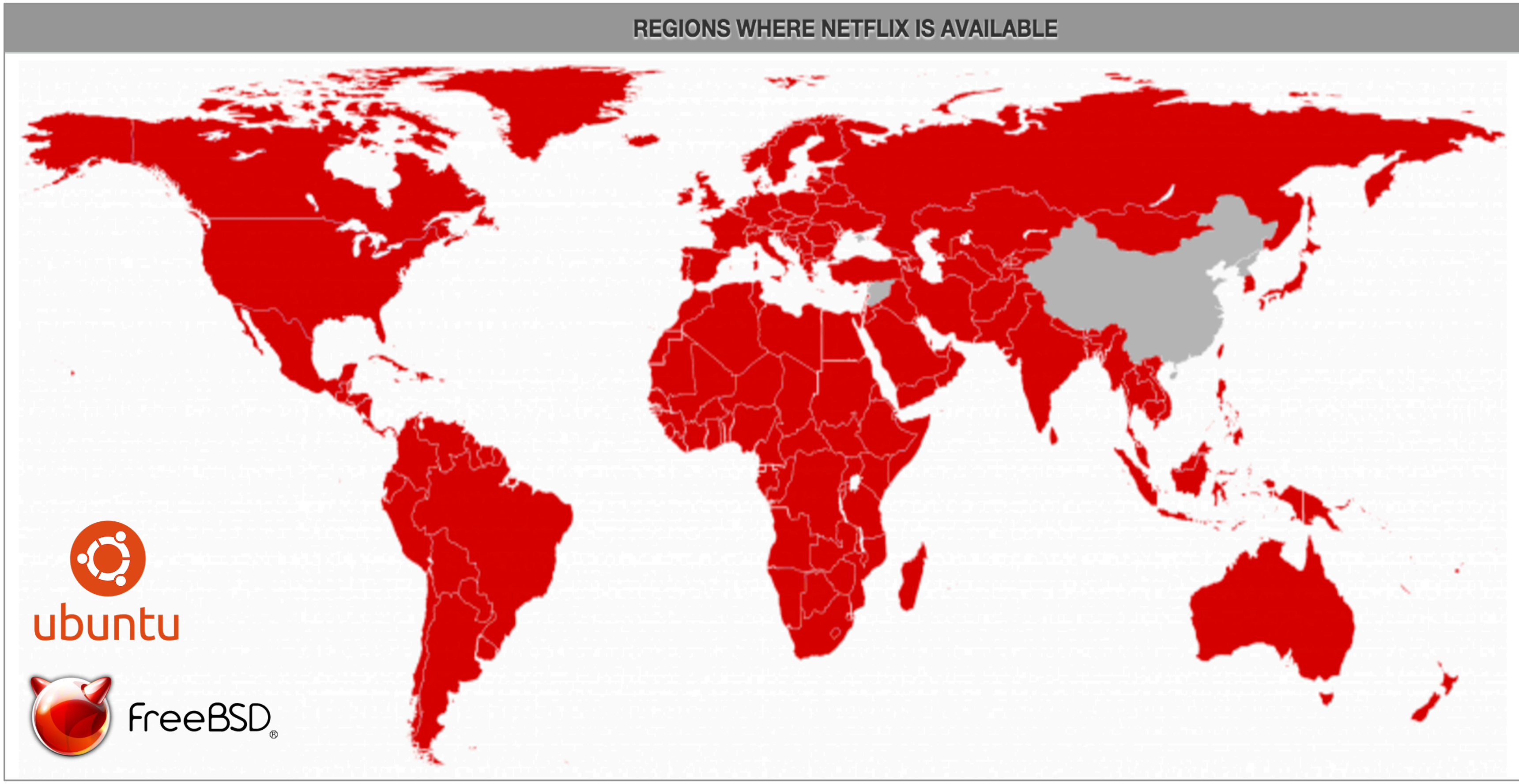


Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram

NETFLIX



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?

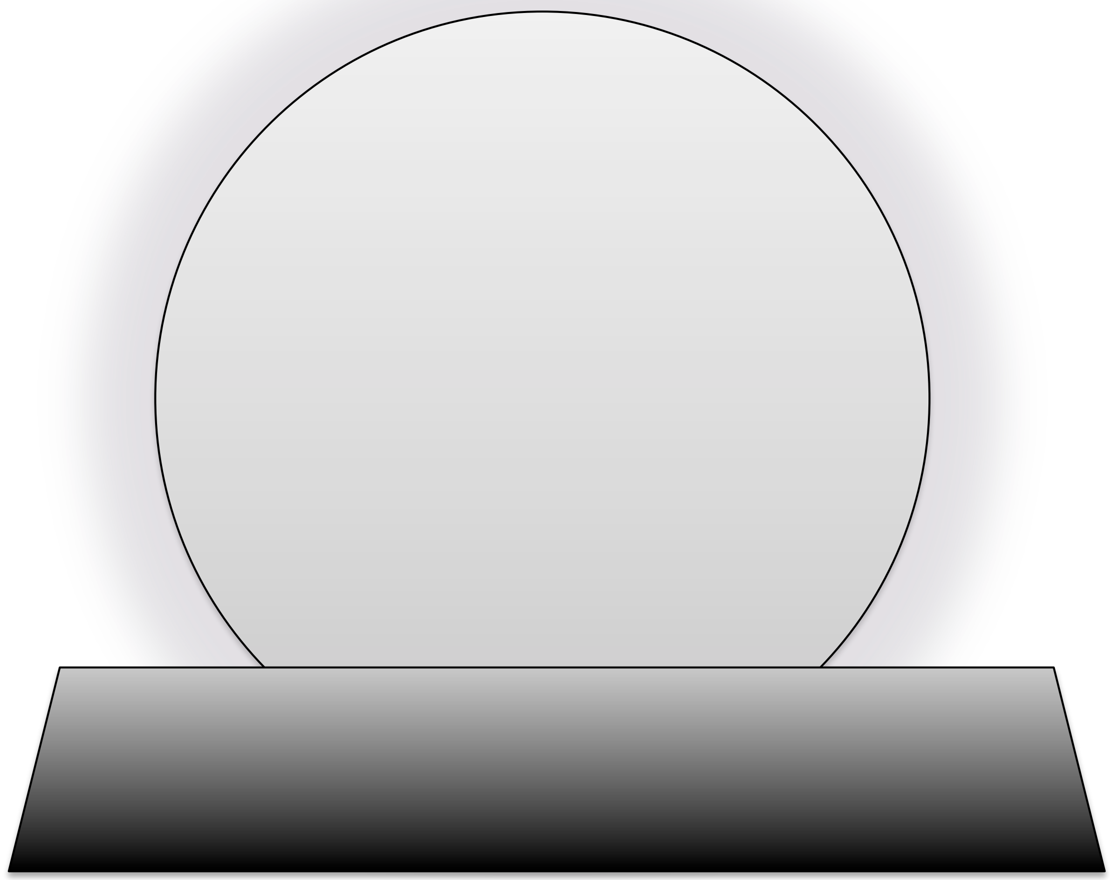
```
# ps alx
 F S UID   PID  PPID CPU PRI NICE  ADDR  SZ  WCHAN TTY TIME CMD
 3 S   0     0     0   0   0   20  2253   2   4412 ?  186:14 swapper
 1 S   0     1     0   0  30   20  2423   8  46520 ?   0:00 /etc/init
 1 S   0    16     1   0  30   20  2273  11  46554 co  0:00 –sh
[…]
```

Today

1. Open source
   - Operating systems: Linux, BSDs, illumos, 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

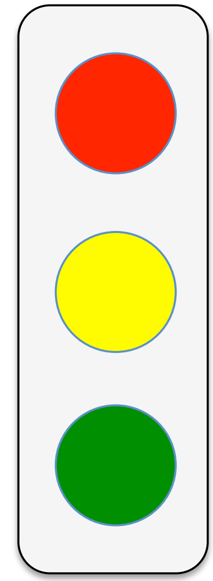
Drunk Man *Anti*-Method

- ~~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

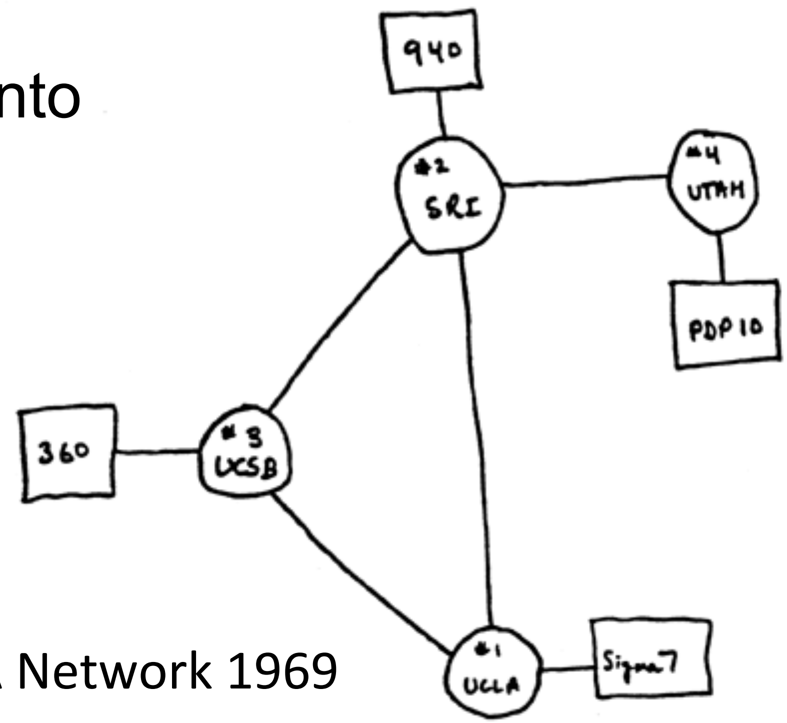
1. What makes you **think** there is a performance problem?
2. Has this system **ever** performed well?
3. What has **changed** recently?
   – software? hardware? load?
4. Can the problem be described in terms of **latency**?
   – or run time. not IOPS or throughput.
5. Does the problem affect **other**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

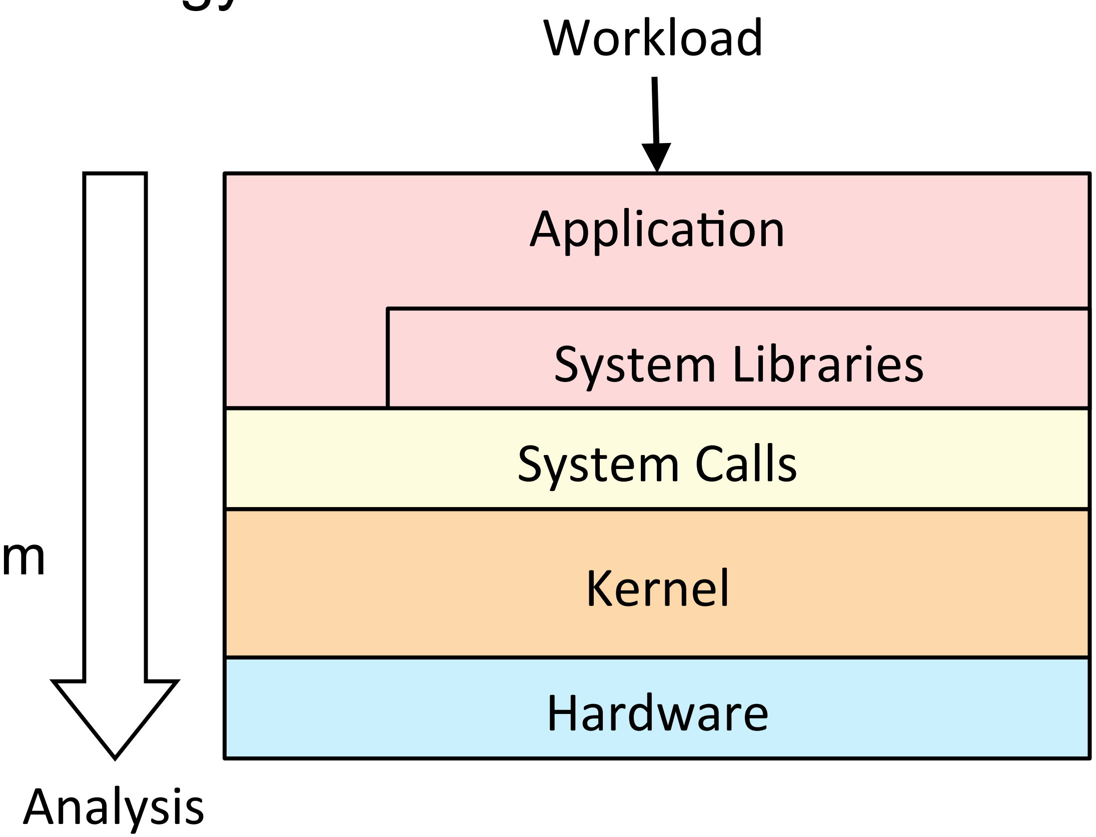
Eg, imagine throughput between the UCSB 360 and the UTAH PDP10 was slow…



ARPA Network 1969

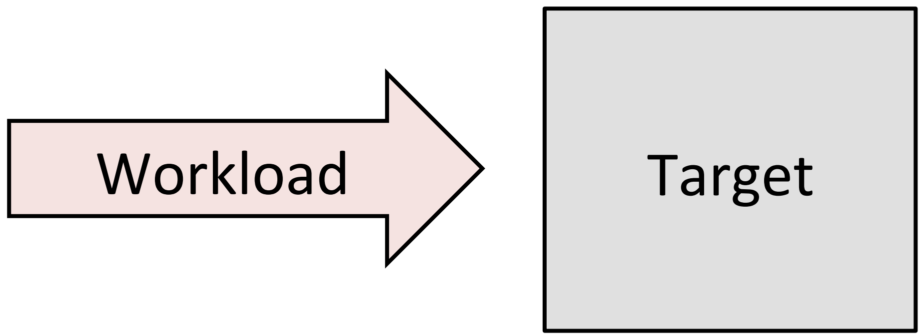
Workload Analysis

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



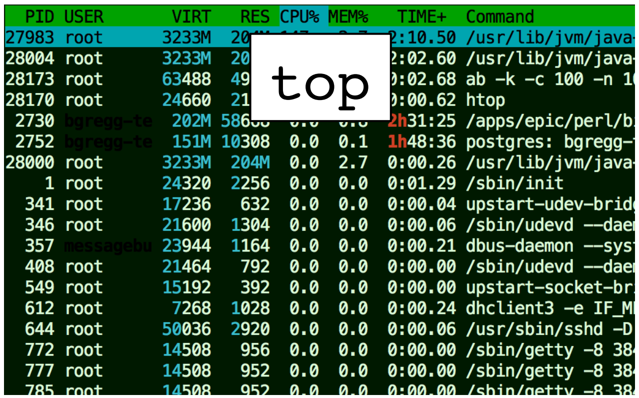
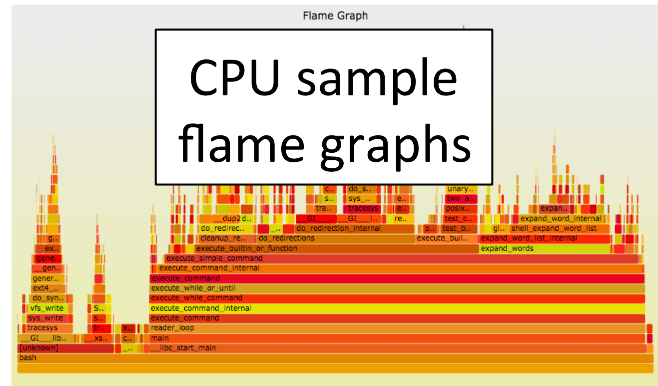
Workload Characterization

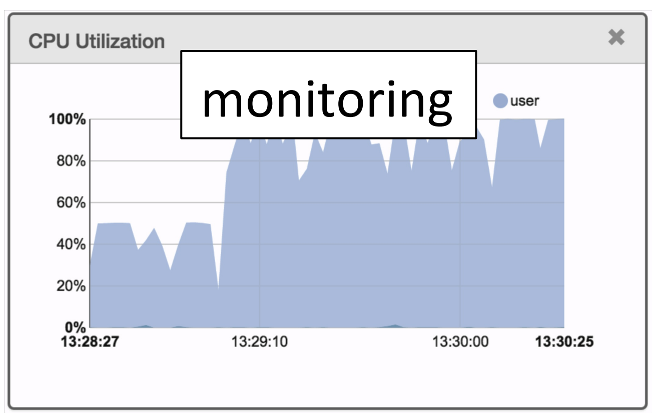
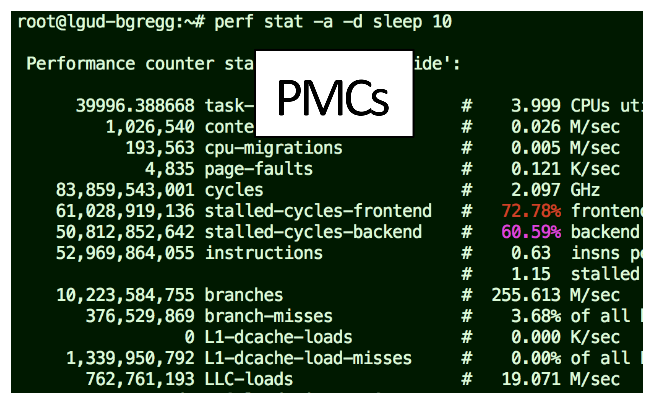
- Check the workload: **who, why, what, how**
  - not resulting performance



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

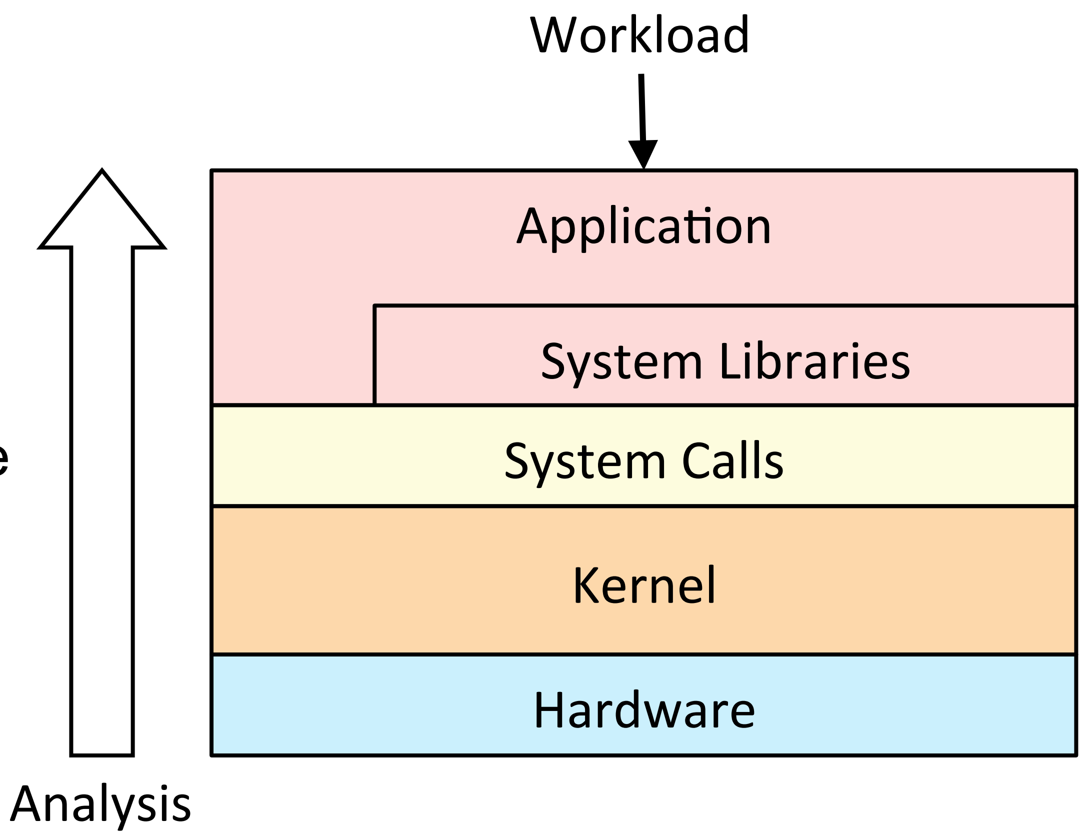
Workload Characterization: CPUs

Who	Why
	

How	What
	

Resource Analysis

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

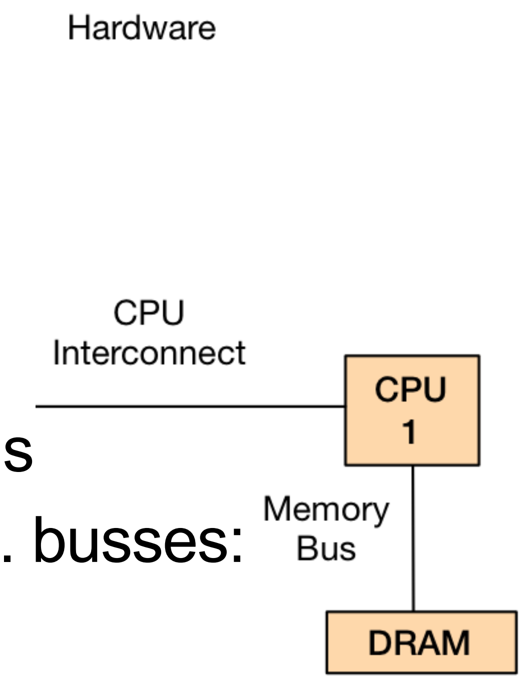


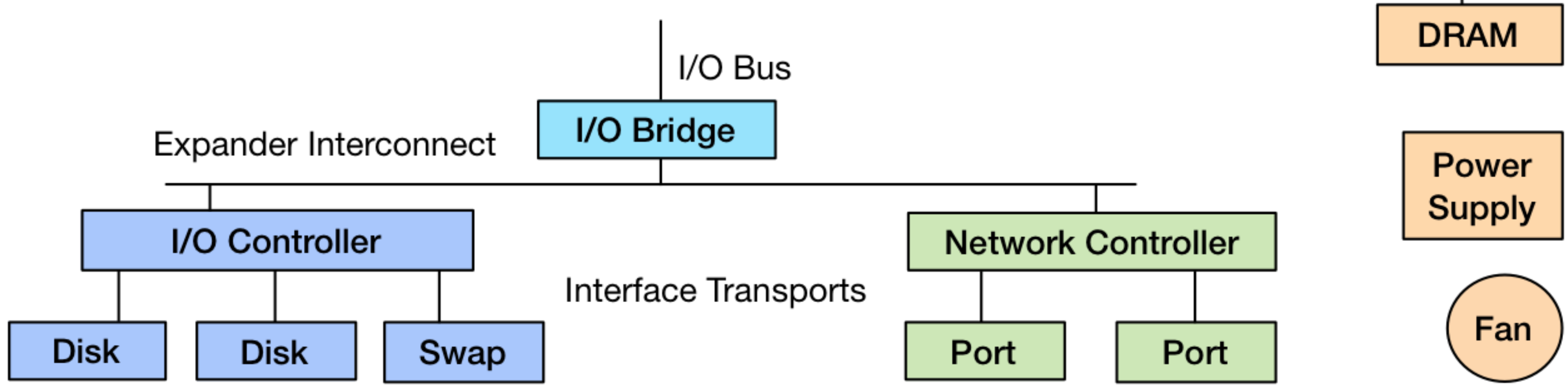
The USE Method

- 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.

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

Redraw

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

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

Resource	Metric	Linux	Solaris	FreeBSD
CPU	errors	`perf` (LPE) if processor specific error events (CPC) are available; eg, AMD64's "04Ah Single-bit ECC Errors Recorded by Scrubber" [4]	`fmadm faulty`; `cpustat` (CPC) for whatever error counters are supported (eg, thermal throttling)	`dmesg`; /var/log/messages; `pmcstat` 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; `vmstat 1`, "r"; DTrace dispqlen.d (DTT) for a better "vmstat r"; per-process: `prstat -mLc 1`, "LAT"	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	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]	per-cpu: `mpstat 1`, "usr" + "sys"; system-wide: `vmstat 1`, "us" + "sy"; per-process: `prstat -c 1` ("CPU" == recent), `prstat -mLc 1` ("USR" + "SYS"); per-kernel-thread: `lockstat -Ii` *`rate`*, DTrace profile stack()	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 interconnect	errors	LPE (CPC) for whatever is available	`cpustat` (CPC) for whatever is available	`pmcstat` and relevant PMCs for whatever is available
CPU	saturation	LPE (CPC) for stall cycles	`cpustat` (CPC) for stall	`pmcstat` and relevant PMCs for

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 often work on the illumos kernel, a direct descendant of Unix which contains some original AT&T code. It's been interesting to study this earlier version, and see familiar code that has survived over 30 years of development.

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 capacity	errors	malloc() returns 0; ENOMEM
Disk I/O	utilization	system-wide: `iostat -i 1`, "IO active" plus "IO wait" percents; per-disk-controller: `iostat -i 1`, RF, RK, RP "active" percents; rough estimate using `iostat 1`, and "tpm" for transactions per minute on expected max; per-disk: listen to each rattle; unknown from Unix, unless only 1 disk per controller; per-process: unknown
Disk I/O	saturation	unknown [3]
Disk I/O	errors	might get a console message, eg, "err on dev", "ECC on dev" or "no space on dev", otherwise unknown [4]
Tape I/O	utilization	look at tape drives and watch them spin [5]

Apollo Guidance Computer

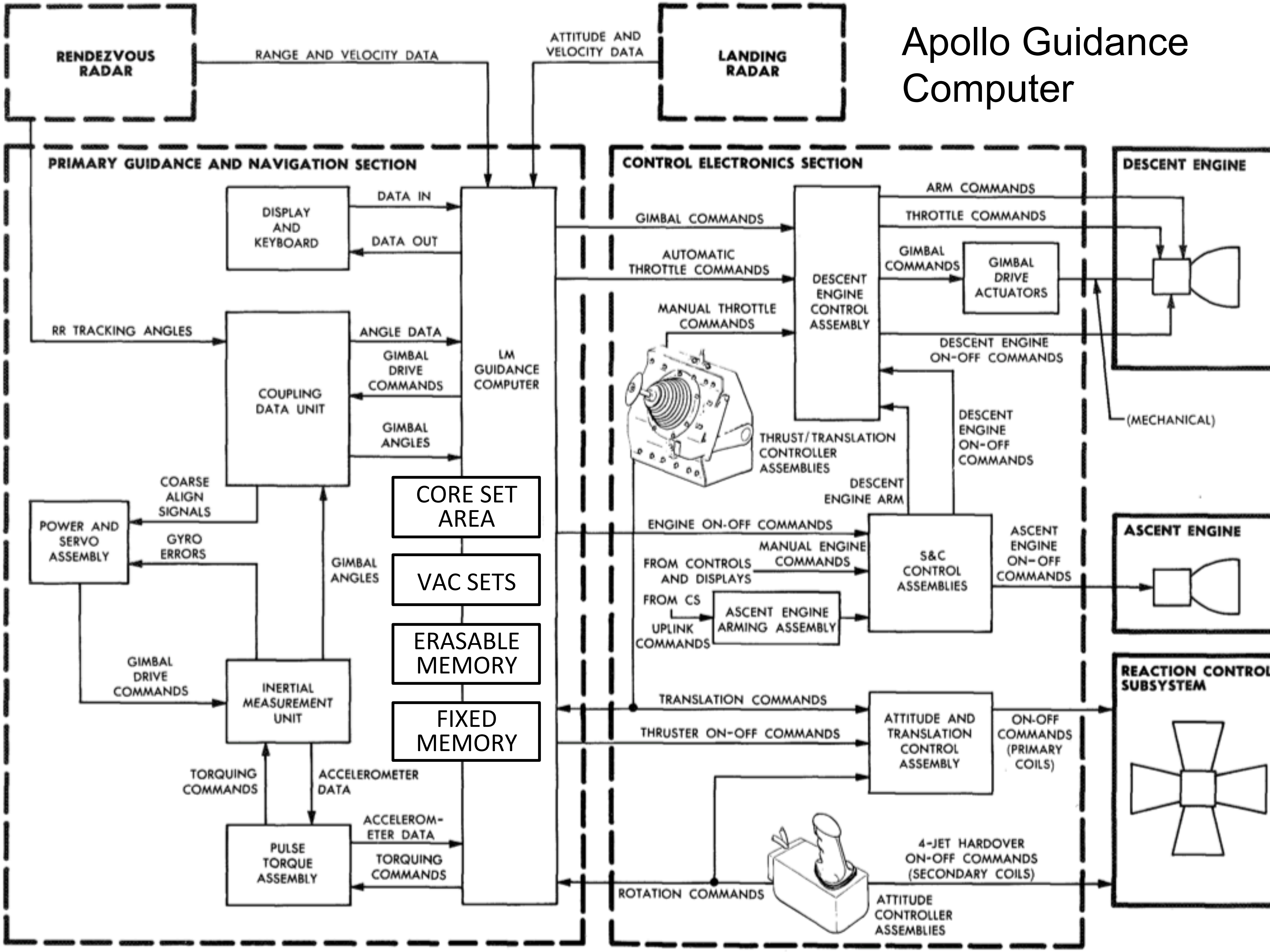
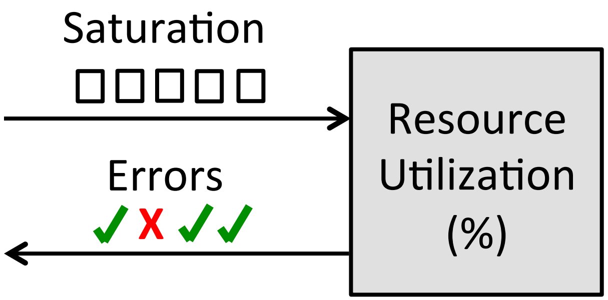


Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram

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

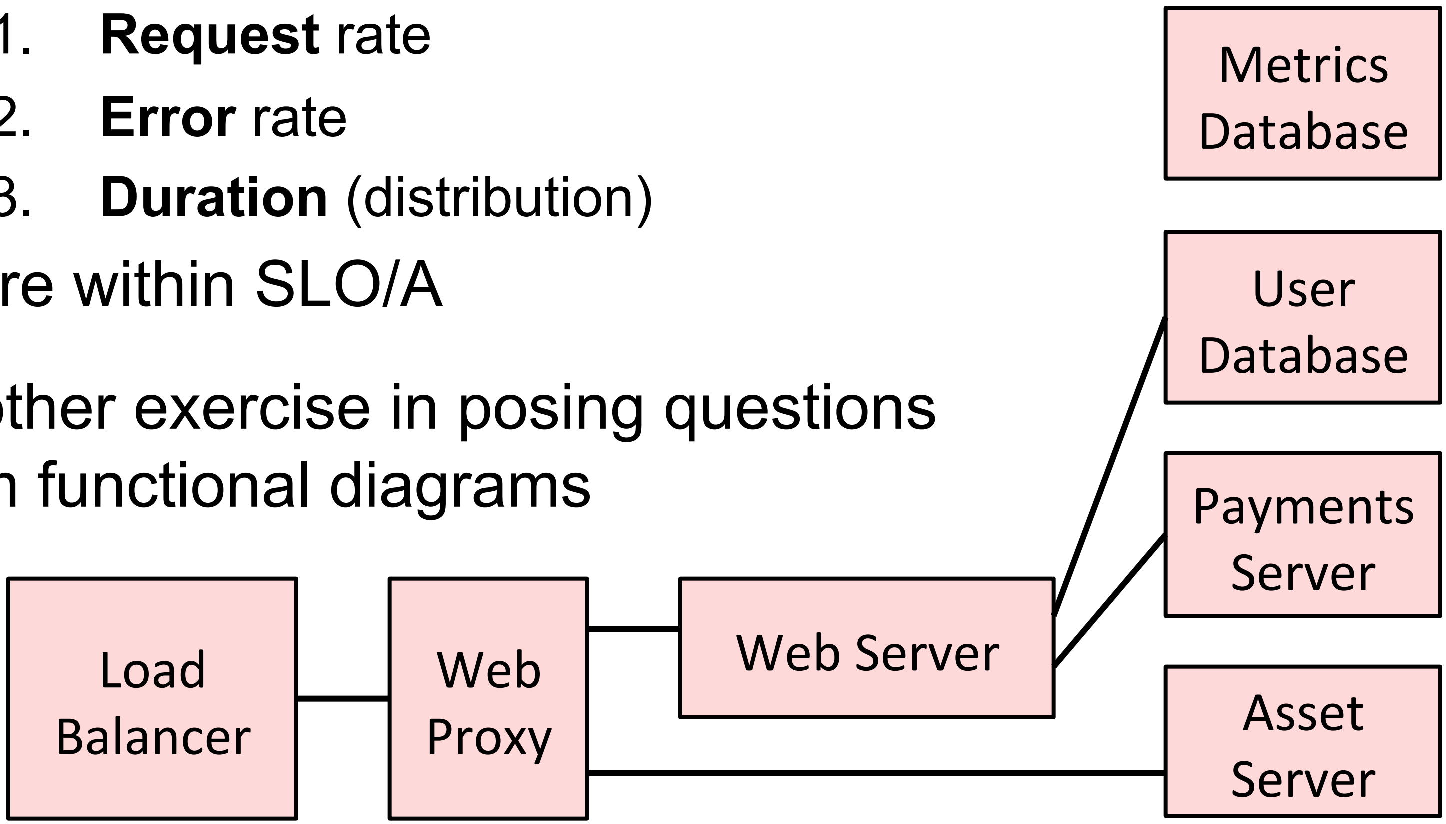


RED Method

- For every service, check that:
  1. **Request** rate
  2. **Error** rate
  3. **Duration** (distribution)

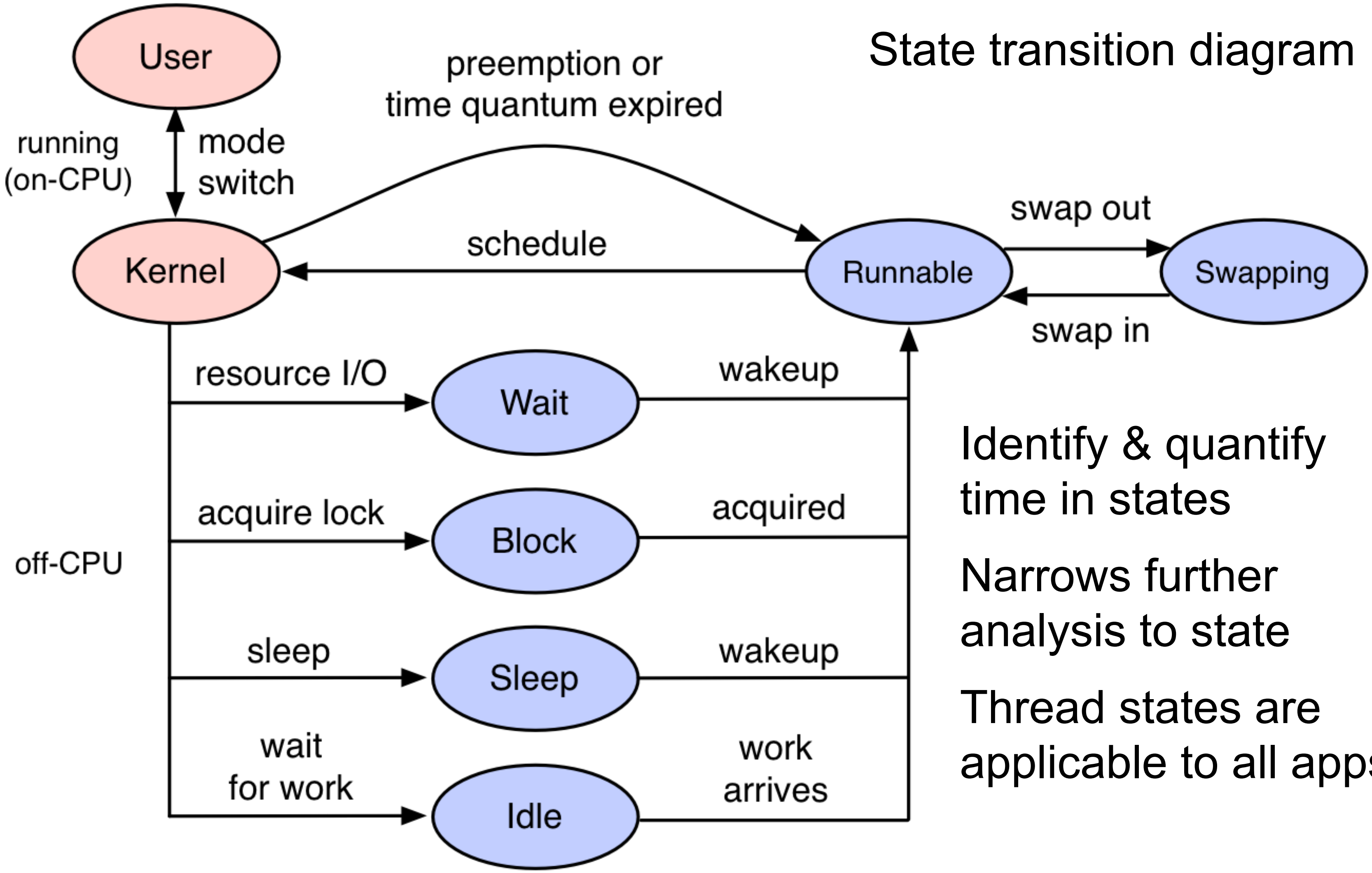
  are within SLO/A

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, Solaris

1) `$ prstat -mLc 1`

```
   PID USERNAME  USR  SYS  TRP  TFL  DFL  LCK  SLP  LAT  VCX  ICX  SCL  SIG PROCESS/LWPID
 45747 1000       35   28  0.0  0.0   22  0.0   16  0.1  216   93  38K    0 beam.smp/192
[...]
```

USR SYS: Executing; DFL: Anon. Paging; LCK: Lock +Idle; SLP: Sleep +Idle; LAT: Runnable

2)

Fields	States	Analysis	Actions
USR+SYS	Executing	1. Profile stacks using DTrace; Flame Graphs 2. Check CPU stall cycles: cpustat, DTrace 3. If SYS time, analyze syscalls using DTrace	Look for inefficiencies Look for tunables/ config in active code
DFL	Anon. Paging	1. Confirm using: vmstat -p 1, "api" 2. Check system-wide memory free: vmstat 1 3. Check any resource controls; eg: zonememstat	Upgrade memory Increase memory limits Look for leaks/growth
LCK	Lock + Idle	1. Coarse: profile CPU stacks and look for spins 2. Analyze using DTrace [p]lockstat providers 3. Separate locks and the Idle state using DTrace sched:::off-cpu with ustack()	Check config
SLP	Sleep + Idle	1. Quick resource check: iostat -xnz 1, nicstat 1 2. Identify both sleep reason and separate from Idle: DTrace sched:::off-cpu with ustack() and stack()	Tune or upgrade resource
LAT	Runnable	1. Check system CPU usage: mpstat 1 2. Check any resource controls; eg, prctl, kstat -p caps::cpucaps_zone*: 3. Check for pbind/psets limiting migrations	Upgrade CPUs Increase CPU limits Move/tune other load Unbind apps

TSA: eg, RSTS/E

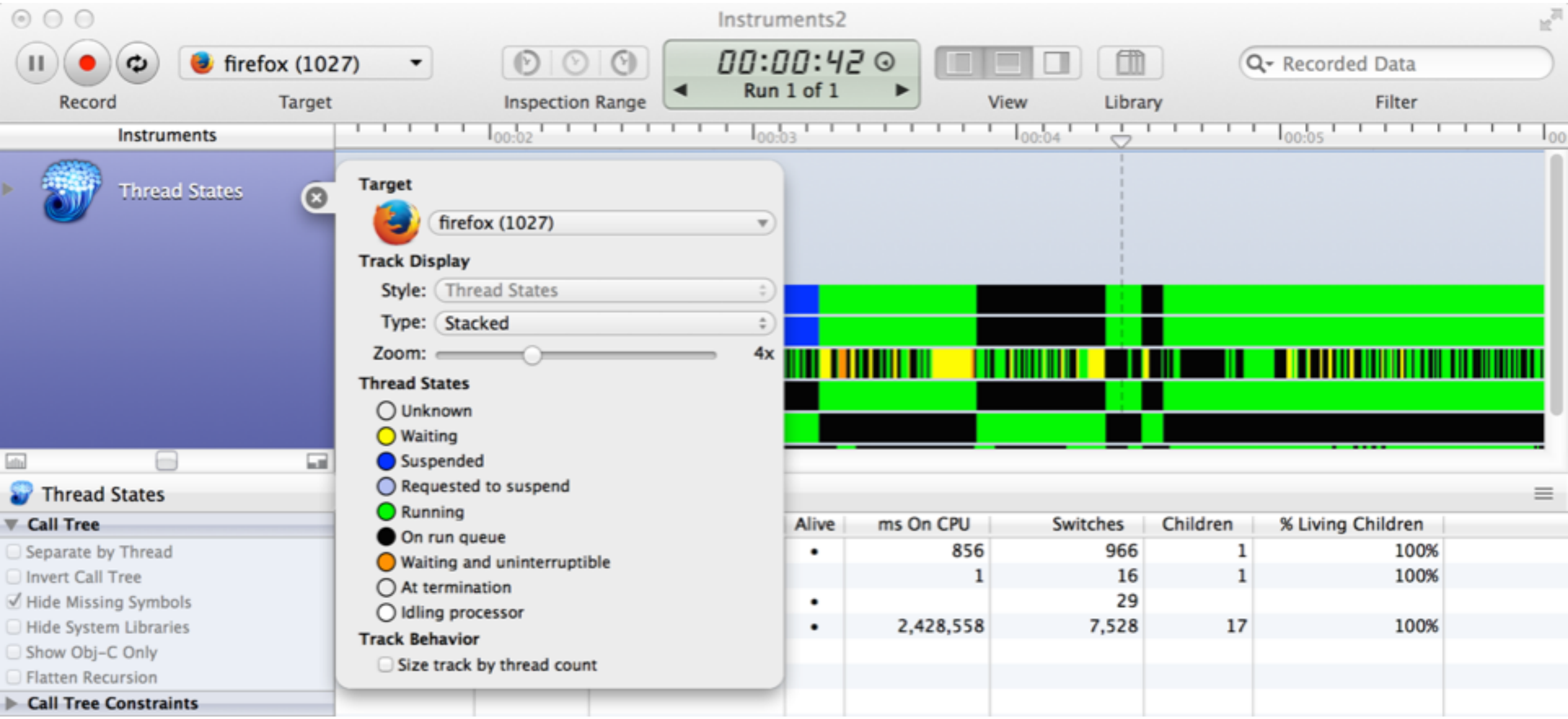
RSTS: DEC OS from the 1970's

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

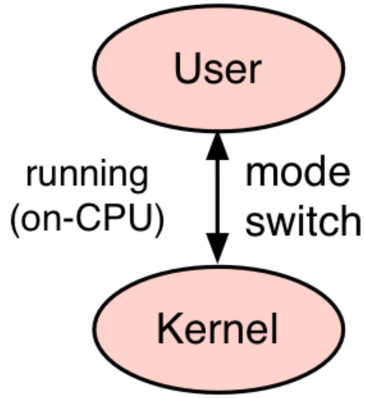
State Column (Job Status)		
RN	Run	Job is running or waiting to run.
RS	Residency	Job is waiting for residency. (The job has been swapped out of memory and is waiting to be swapped back in.)
BF	Buffers	Job is waiting for buffers (no space is available for I/O buffers).
SL	Sleep	Job is sleeping (SLEEP 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.

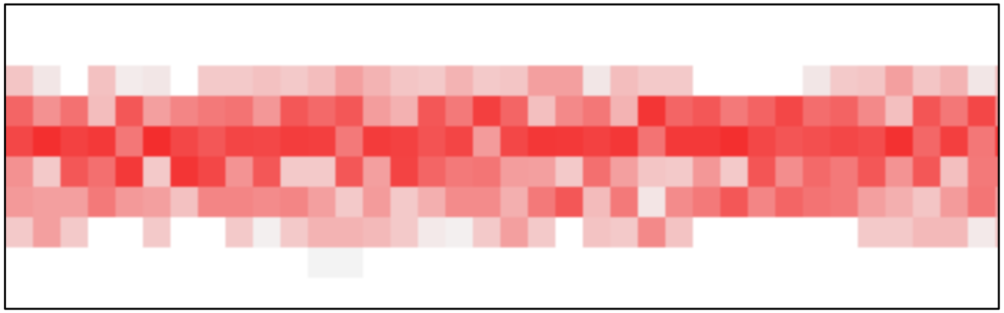
TSA: eg, OS X

Instruments: Thread States

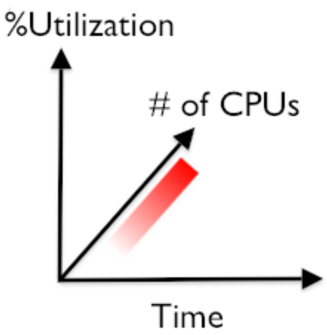


On-CPU Analysis





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. Understand all software that is in >1% of samples

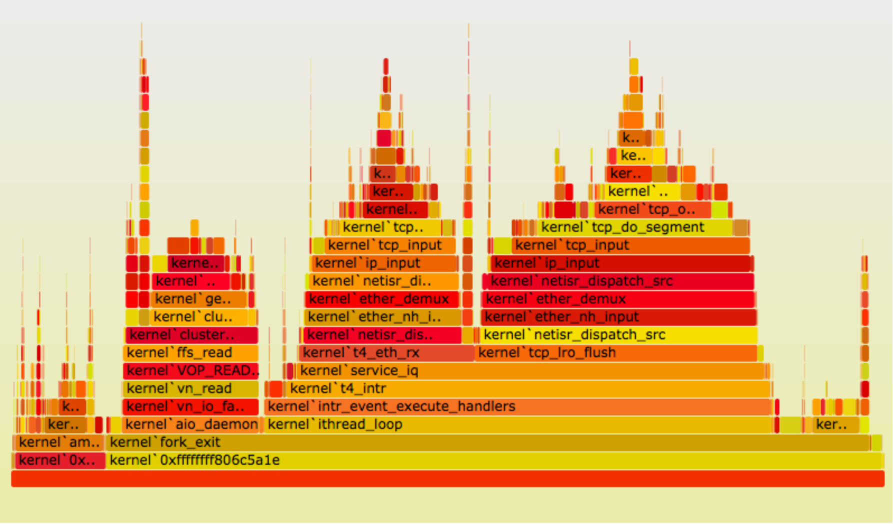
Discovers issues by their CPU usage

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

Narrows target of study to only running code

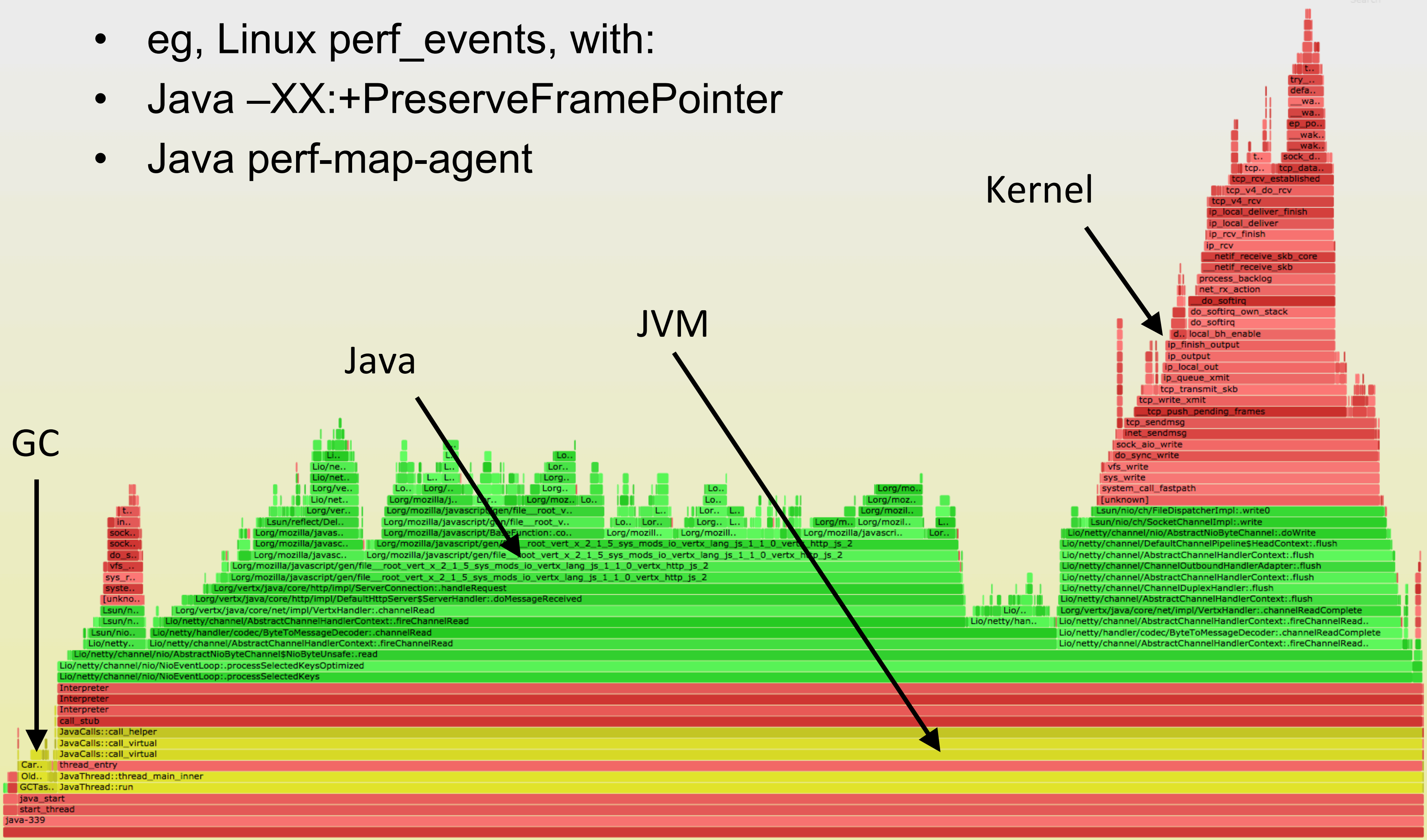
- See: "The Flame Graph", CACM, June 2016

Flame Graph



Java Mixed-Mode CPU Flame Graph

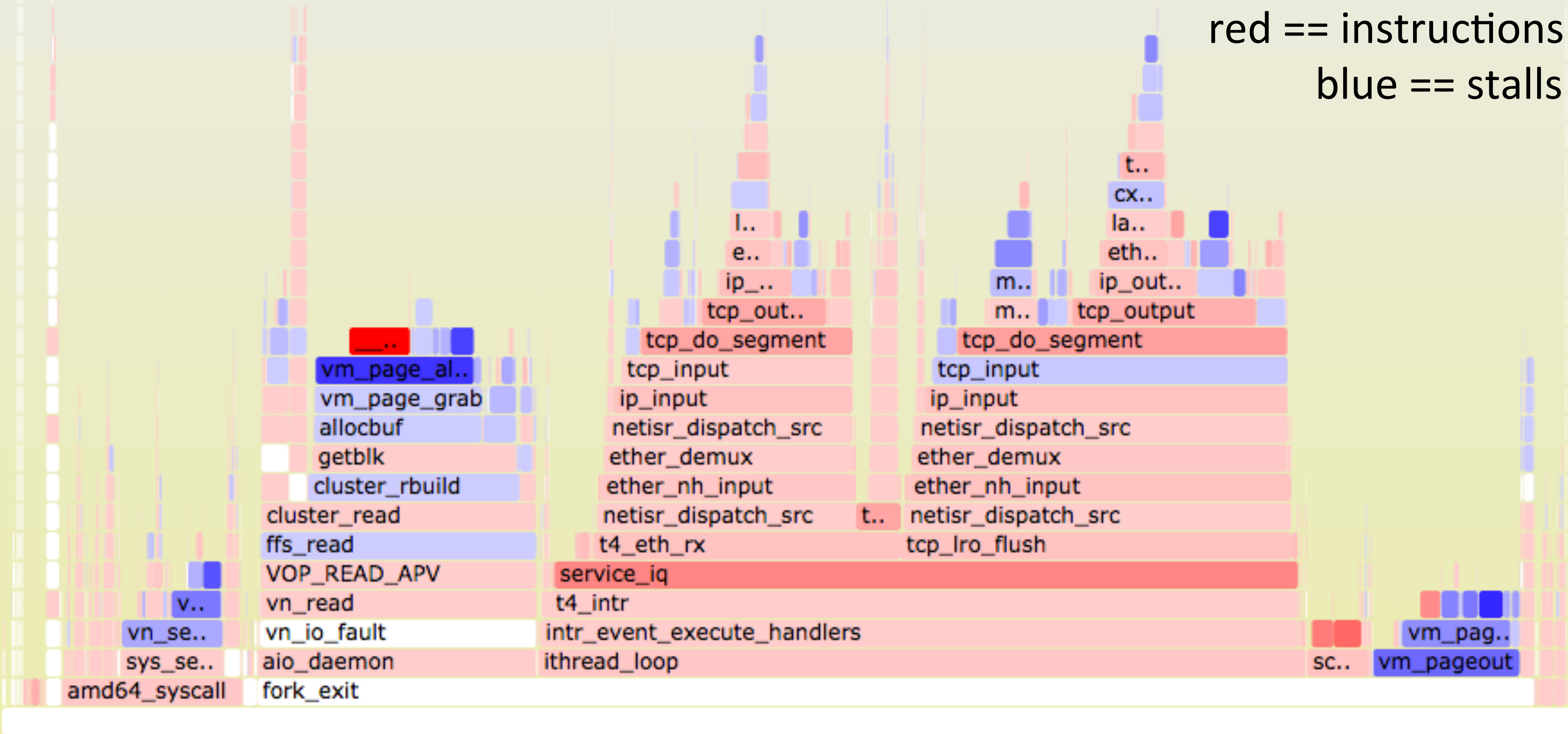
- eg, Linux perf_events, with:
- Java –XX:+PreserveFramePointer
- Java perf-map-agent



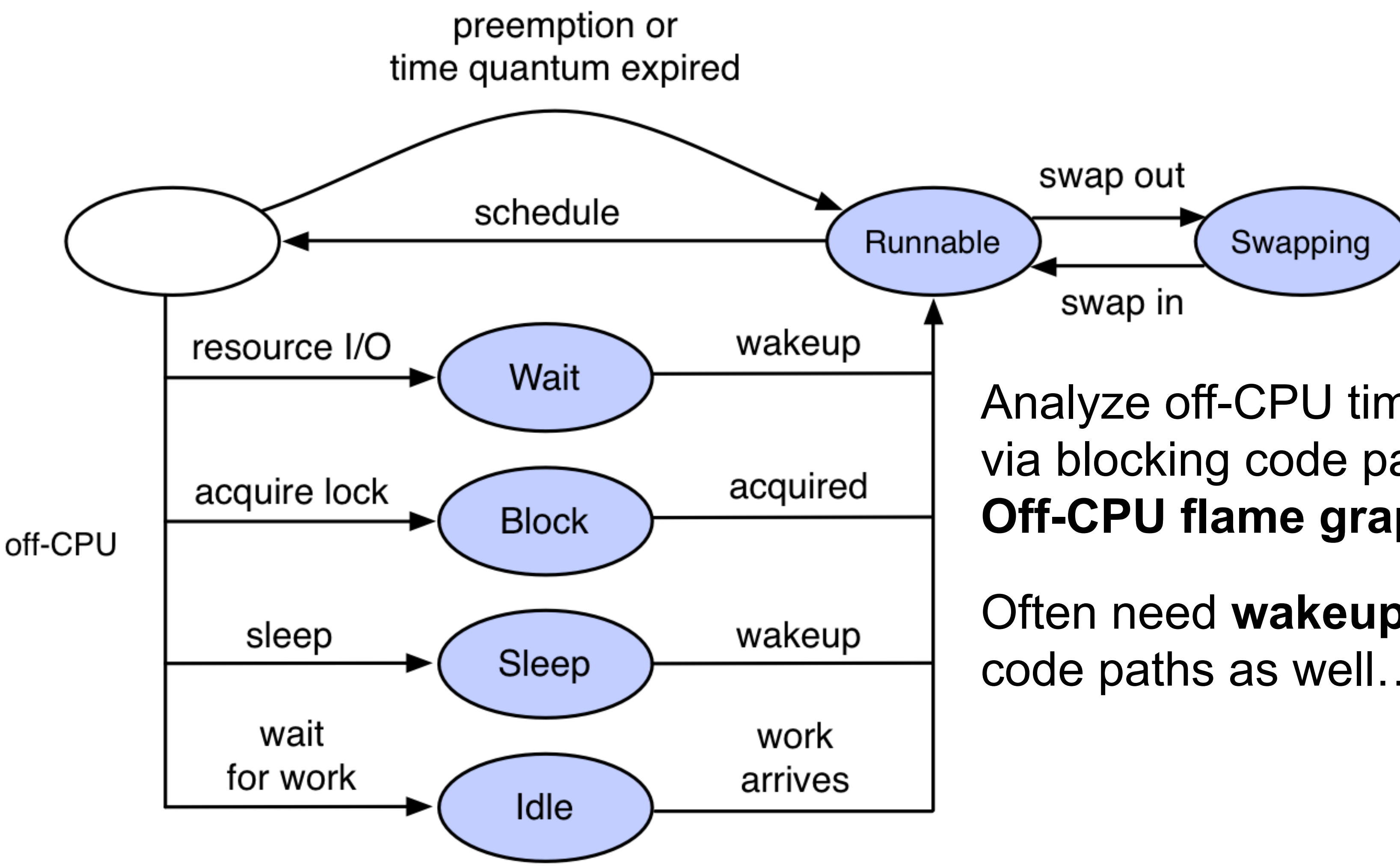
CPI Flame Graph

- Profile cycle stack traces and instructions or stalls separately
- Generate CPU flame graph (cycles) and color using other profile
- eg, FreeBSD: pmcstat

red == instructions
blue == stalls



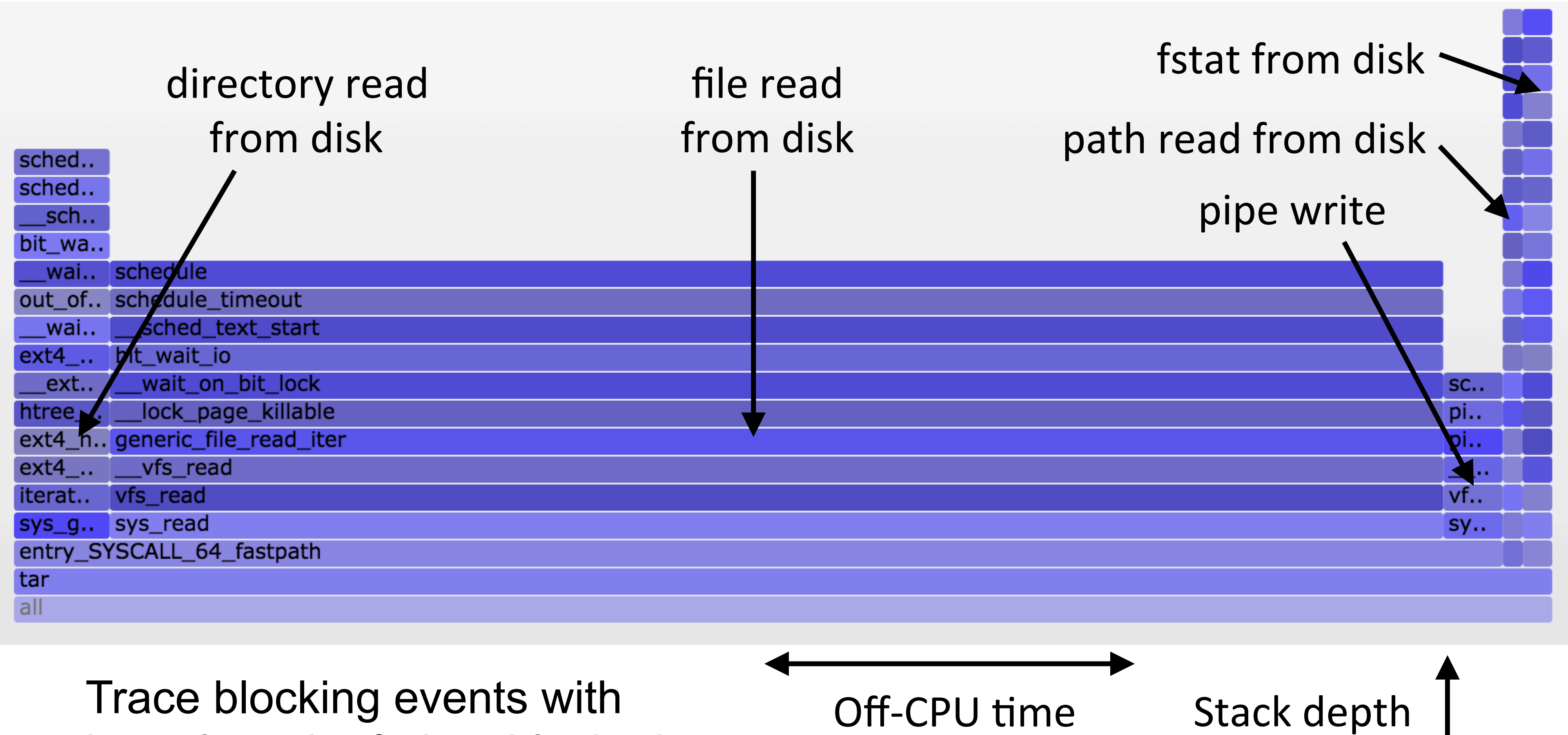
Off-CPU Analysis



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

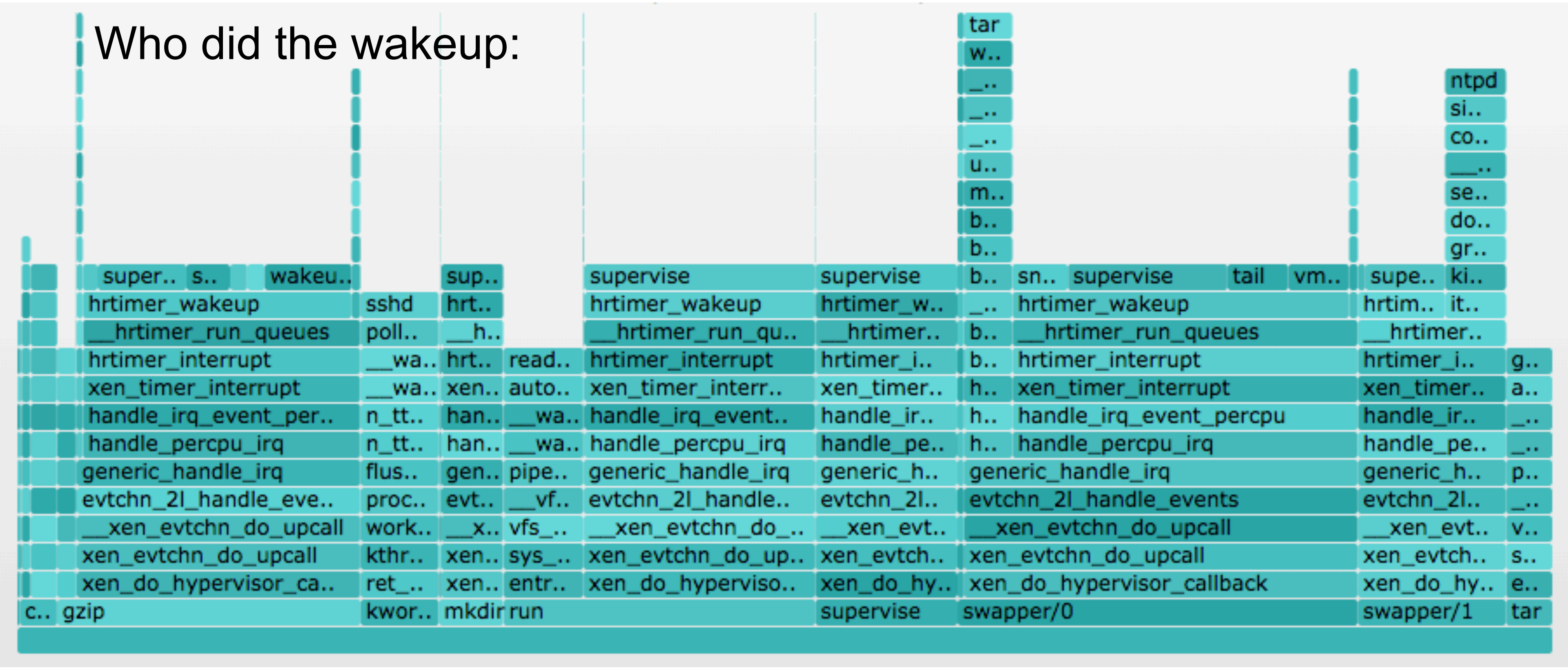
Often need **wakeup** code paths as well…

Off-CPU Time Flame Graph



Trace blocking events with kernel stacks & time blocked (eg, using Linux BPF)

Wakeup Time Flame Graph



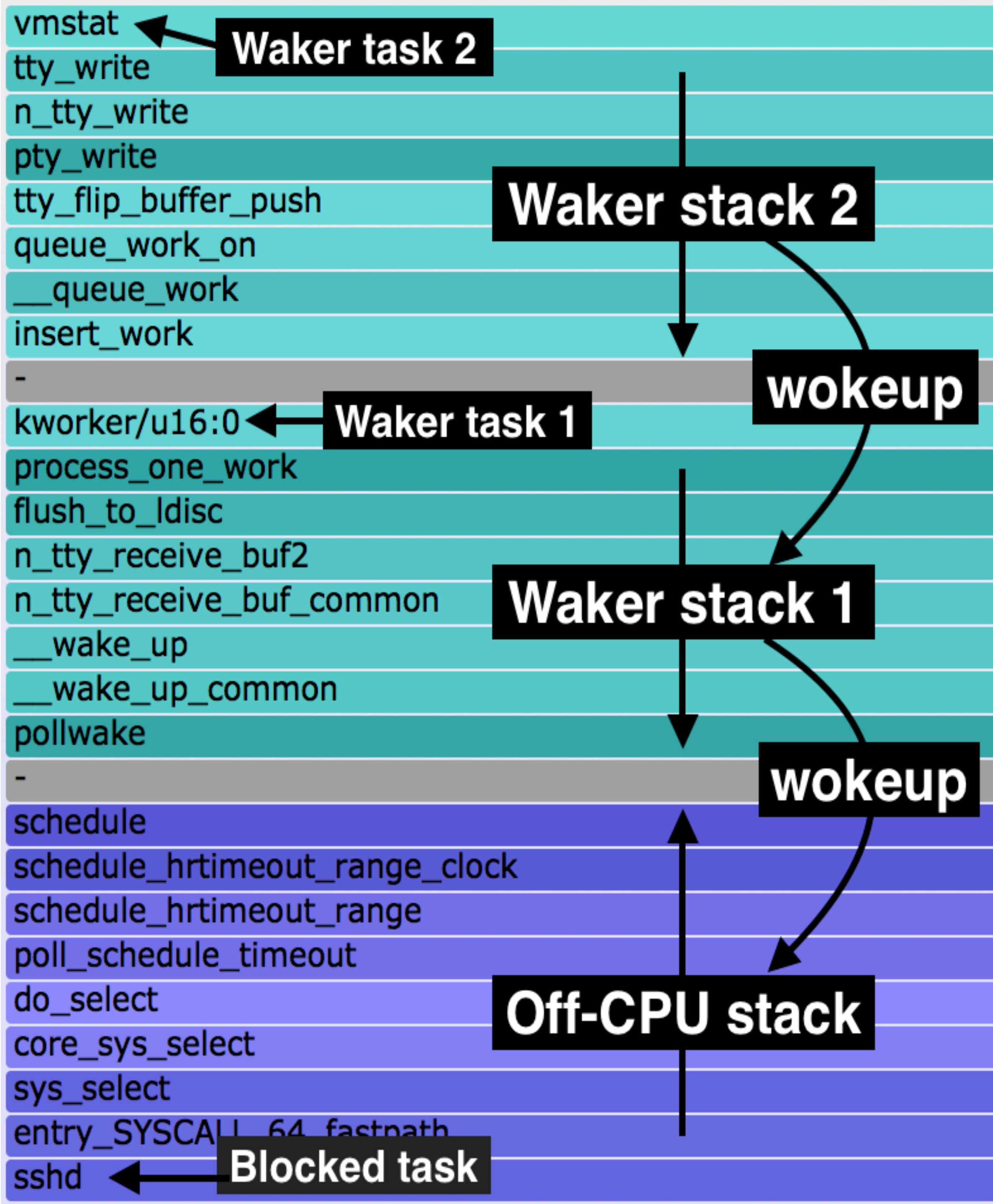
… can also associate wake-up stacks *with* off-CPU stacks
(eg, Linux 4.6: samples/bpf/offwaketime*)

Chain Graphs

Associate more than one waker: the full chain of wakeups

With enough stacks, all paths lead to metal

An approach for analyzing *all* off-CPU issues

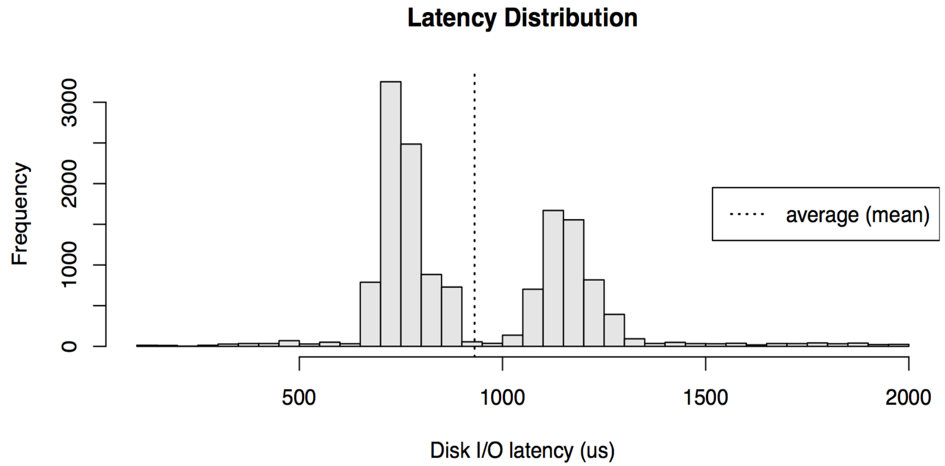


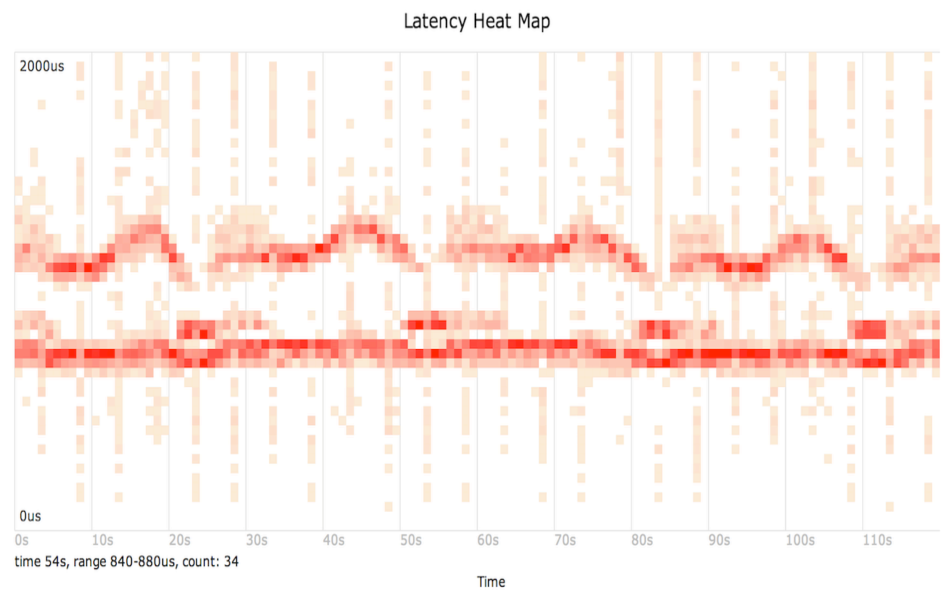
Latency Correlations

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



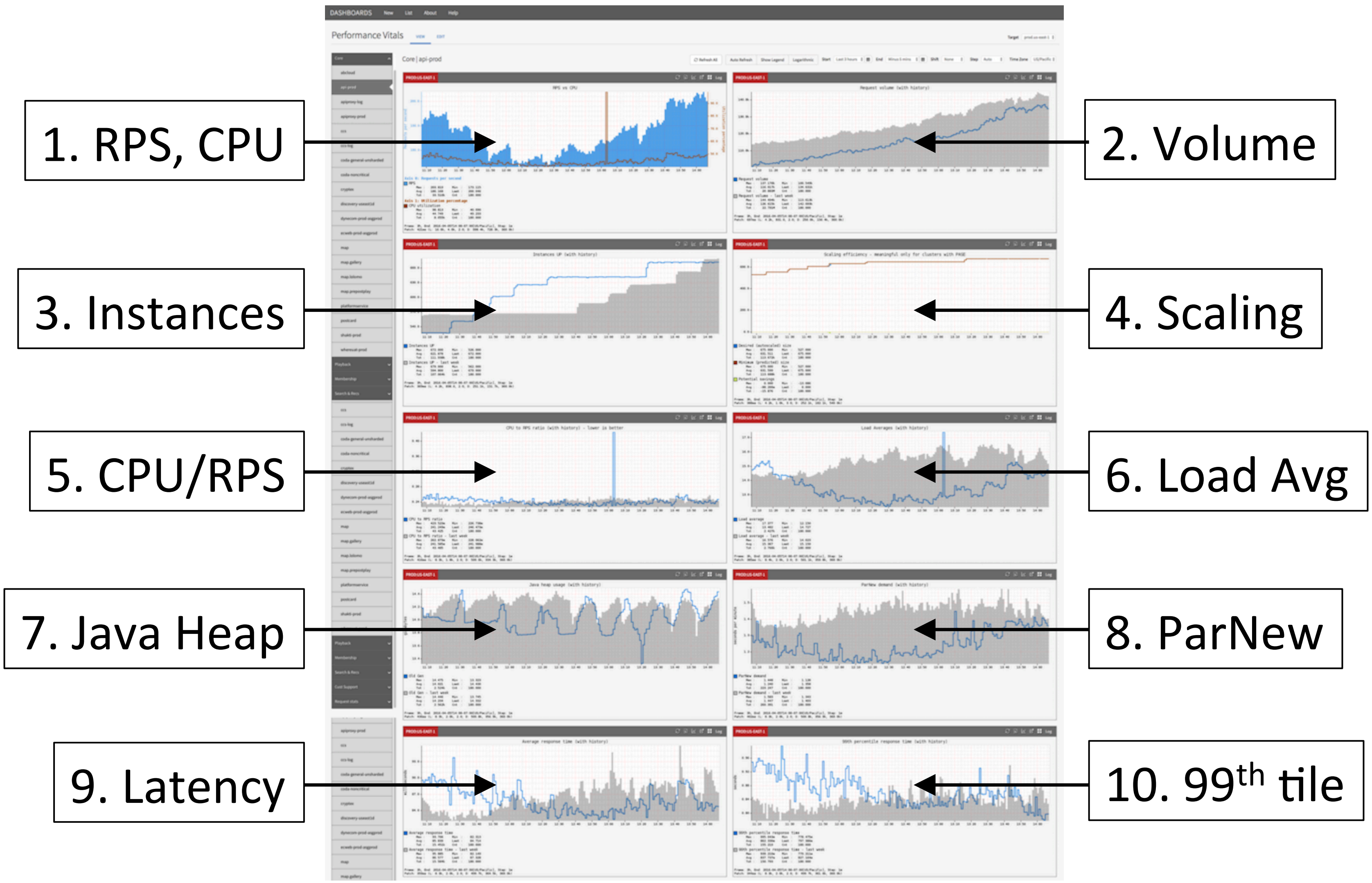


Checklists: eg, Linux Perf Analysis in 60s

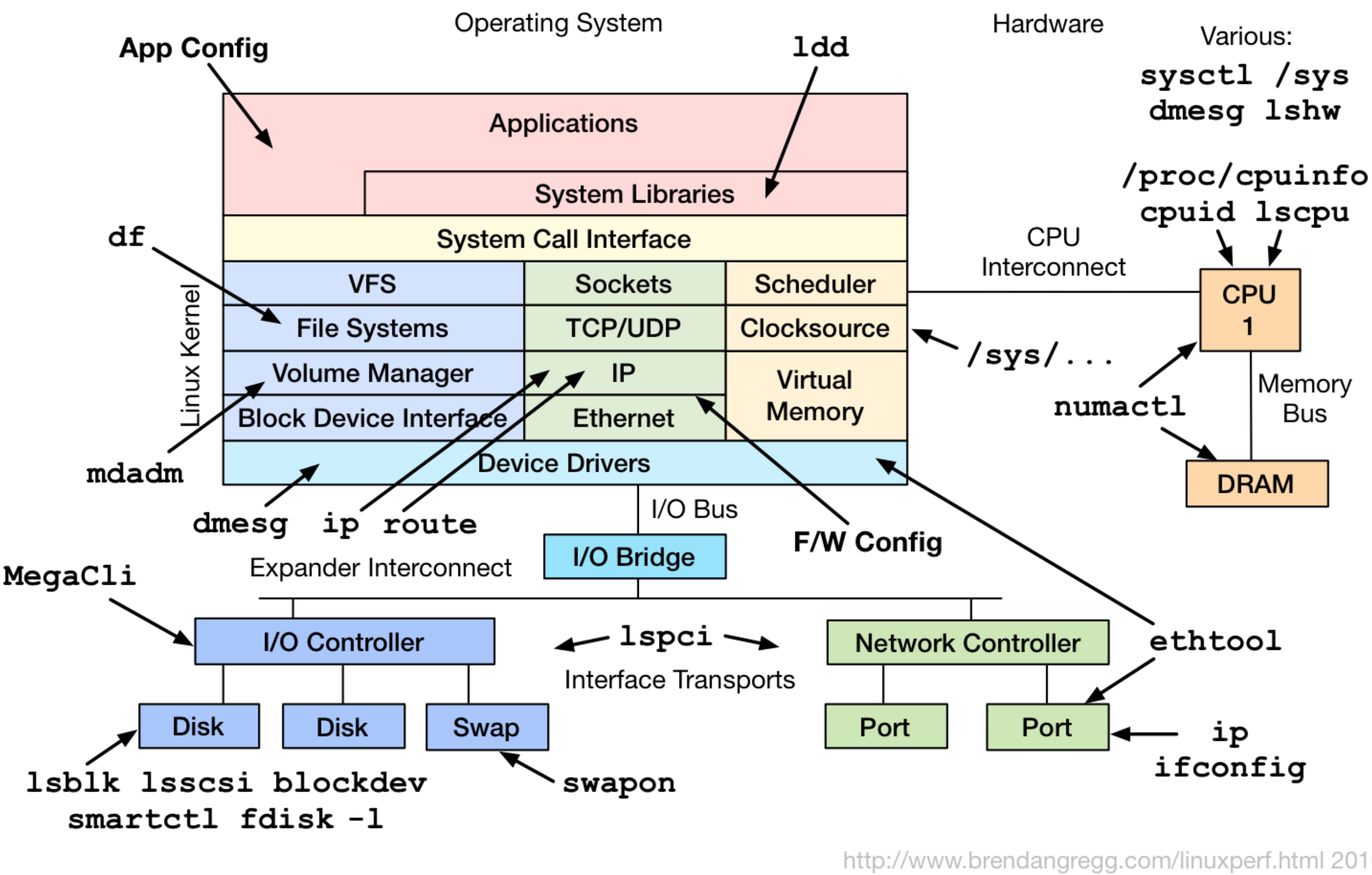
1. `uptime` ------------------------------► load averages
2. `dmesg | tail` ------------------------► kernel errors
3. `vmstat 1` ----------------------------► overall stats by time
4. `mpstat -P ALL 1` ---------------------► CPU balance
5. `pidstat 1` ---------------------------► process usage
6. `iostat -xz 1` ------------------------► disk I/O
7. `free -m` -----------------------------► memory usage
8. `sar -n DEV 1` ------------------------► network I/O
9. `sar -n TCP,ETCP 1` -------------------► TCP stats
10. `top` --------------------------------► check overview

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

Checklists: eg, Netflix perfvitals Dashboard

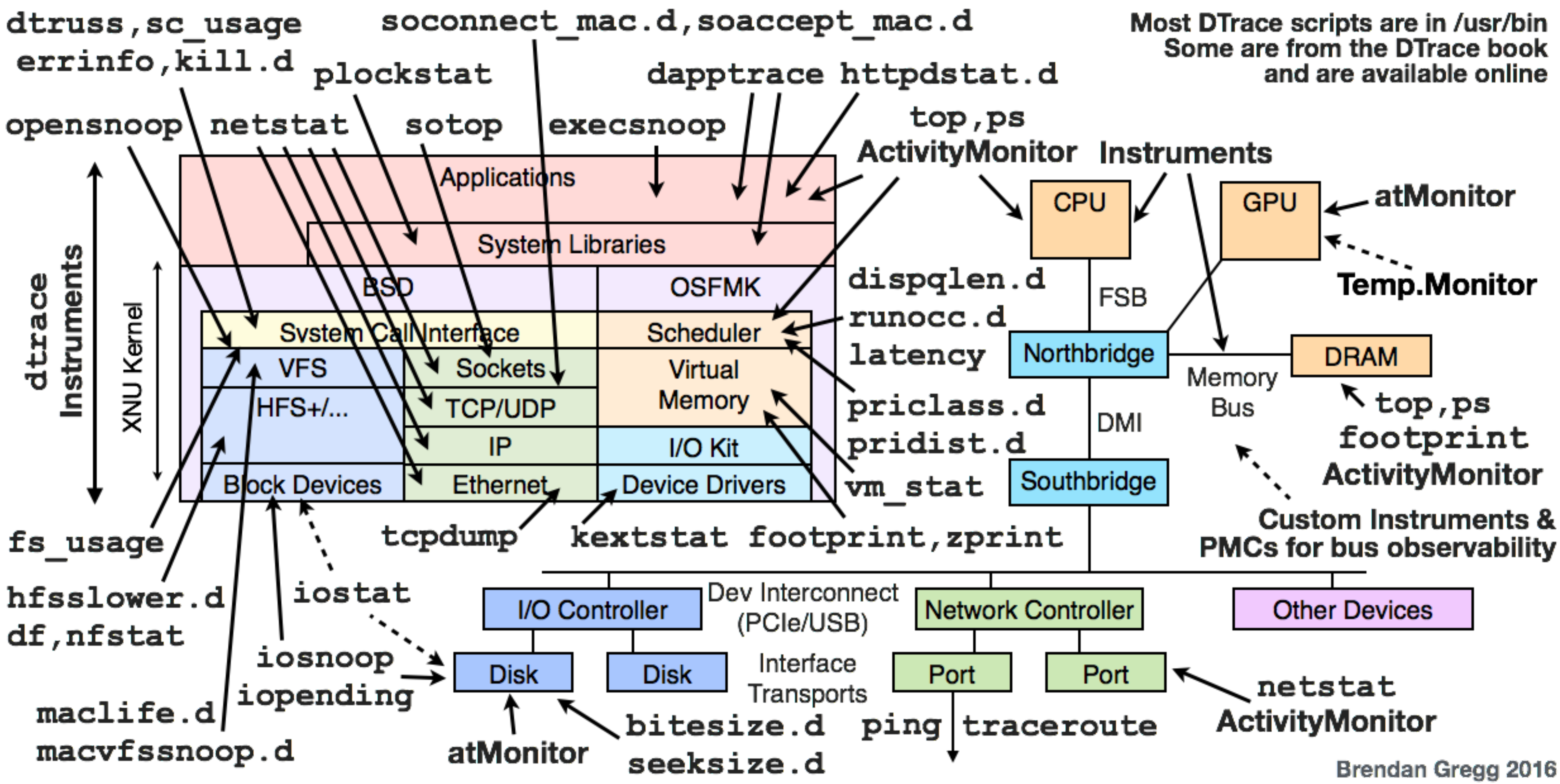


Static Performance Tuning: eg, Linux



Tools-Based Method

1. Try all the tools! May be an anti-pattern. Eg, OS X:



Other Methodologies

- Scientific method
- 5 Why's
- Process of elimination
- Intel's Top-Down Methodology
- Method R

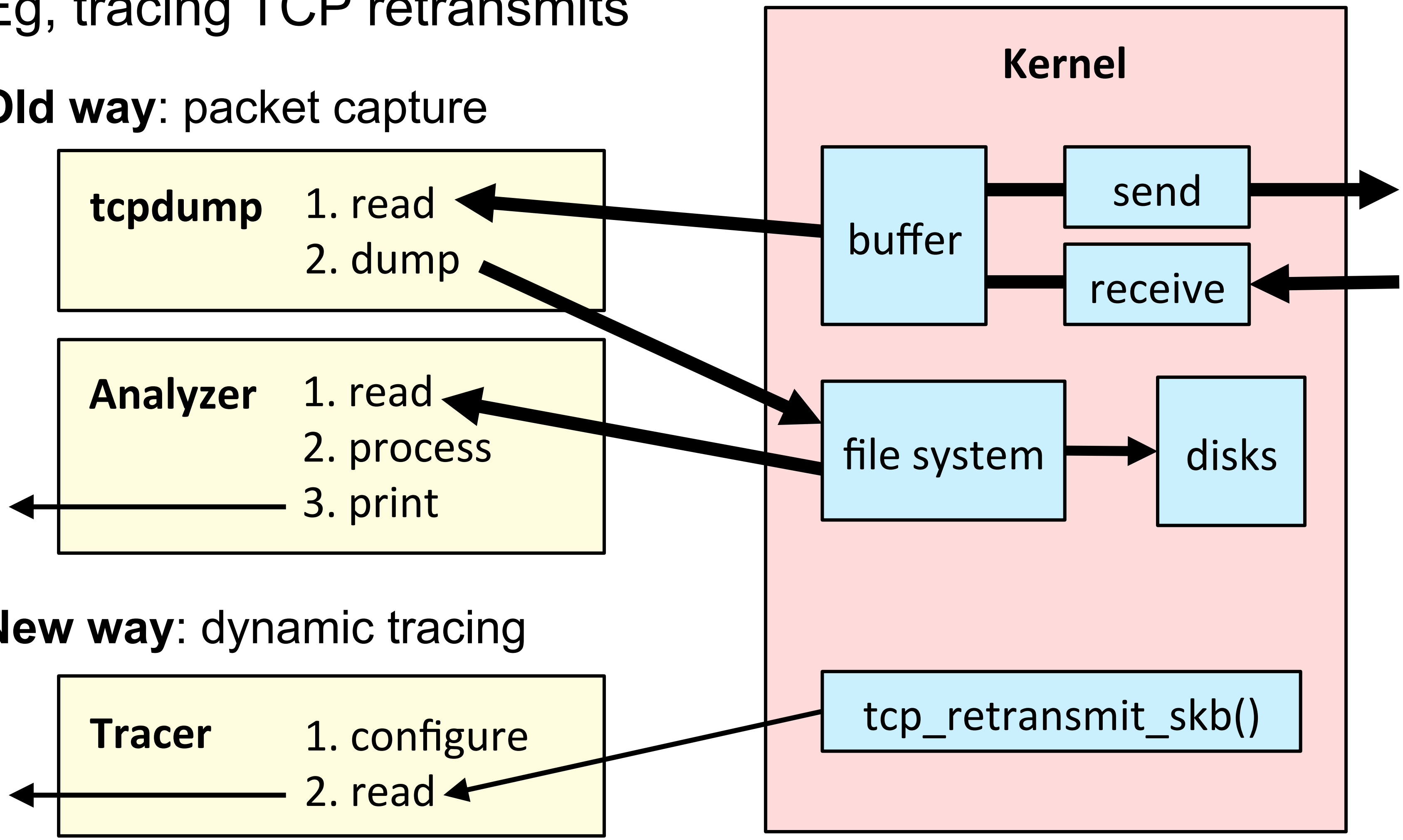
What You Can Do

What you can do

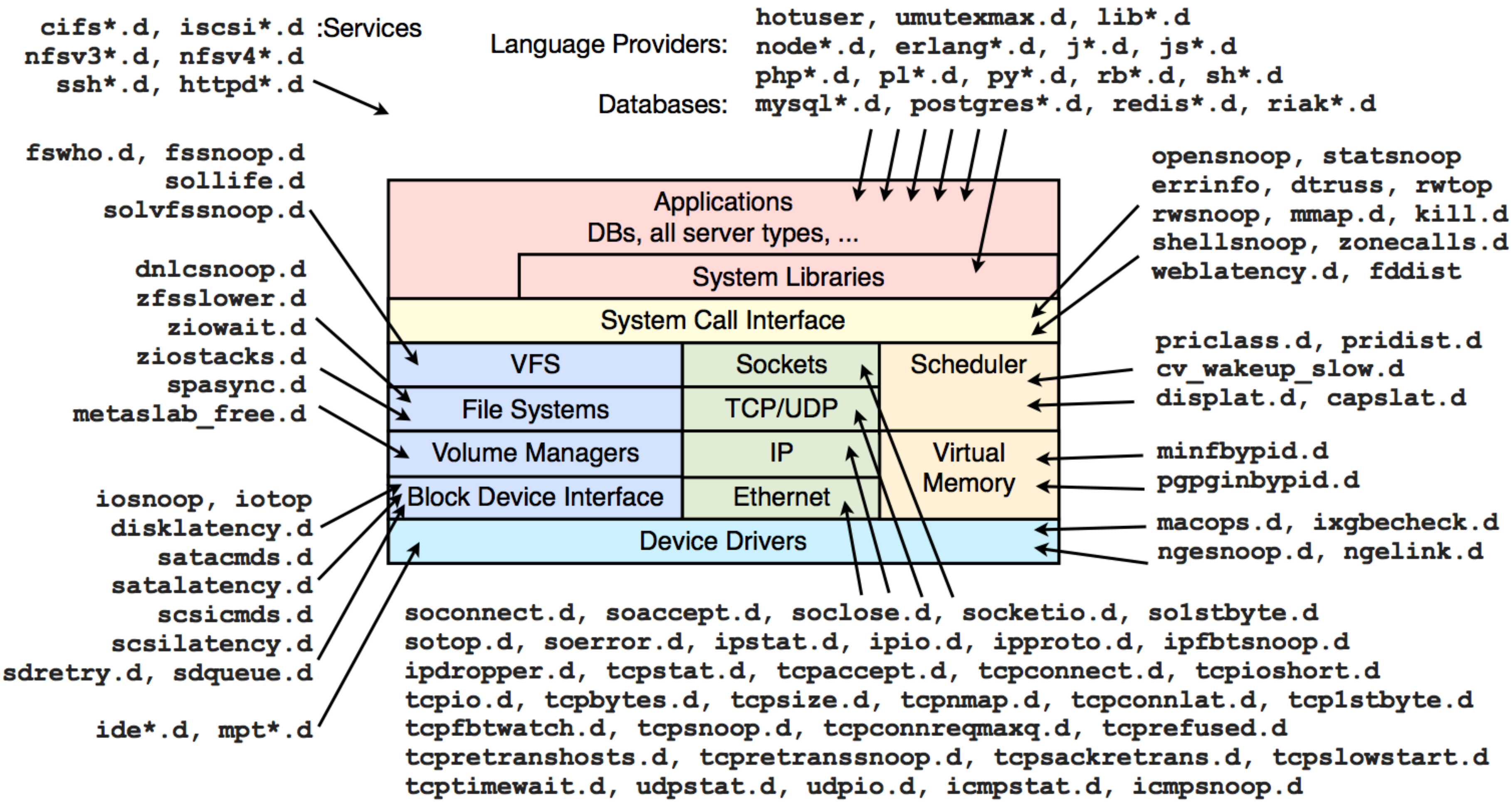
1. Know what's now possible on modern systems
   – Dynamic tracing: efficiently instrument any software
   – CPU facilities: PMCs, MSRs (model specific registers)
   – Visualizations: flame graphs, latency heat maps, …
2. Ask questions first: use methodologies to ask them
3. Then find/build the metrics
4. Build or buy dashboards to support methodologies

Dynamic Tracing: Efficient Metrics

Eg, tracing TCP retransmits



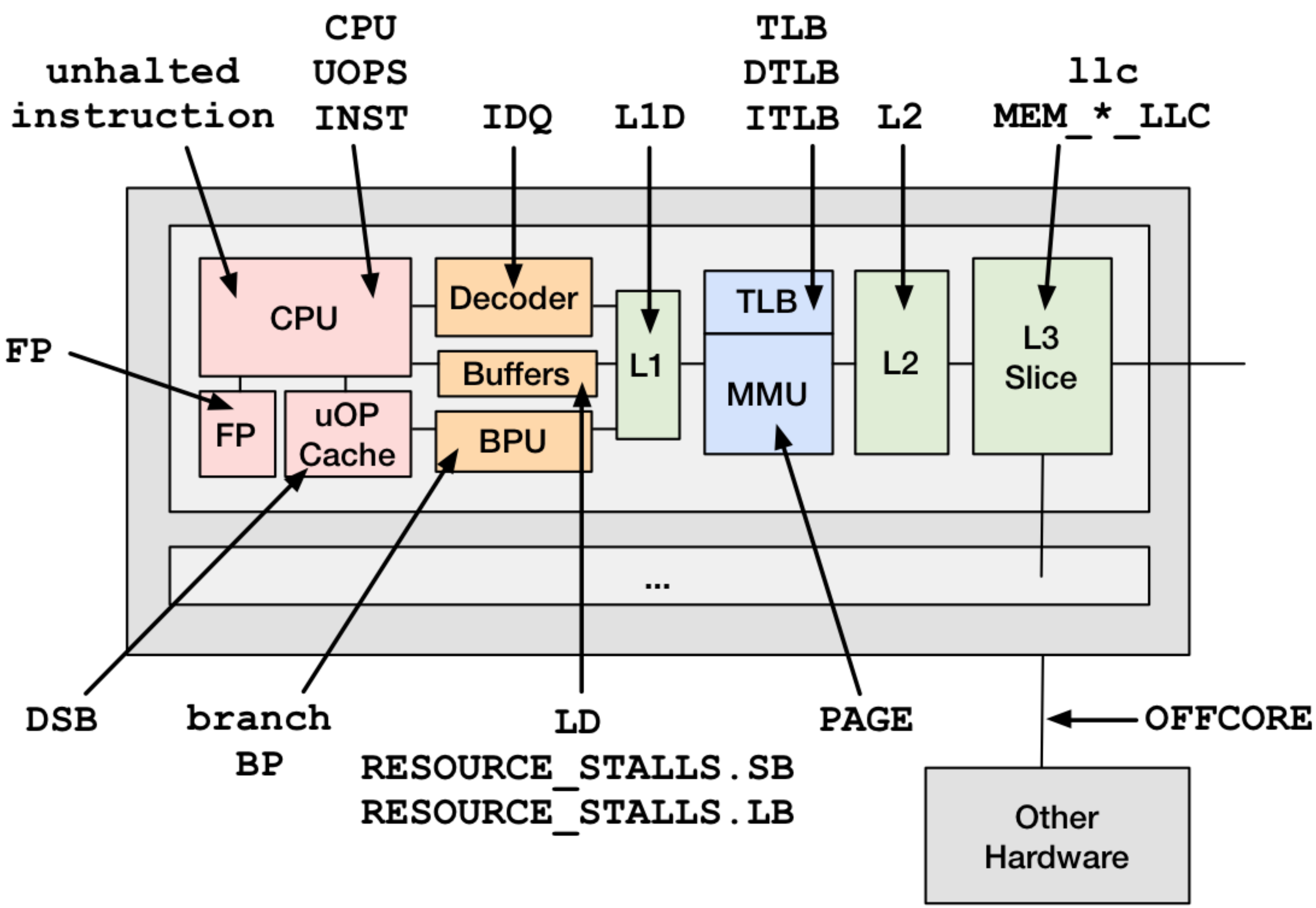
Dynamic Tracing: Measure Anything



Those are Solaris/DTrace tools. Now becoming possible on all OSes: FreeBSD & OS X DTrace, Linux BPF, Windows ETW

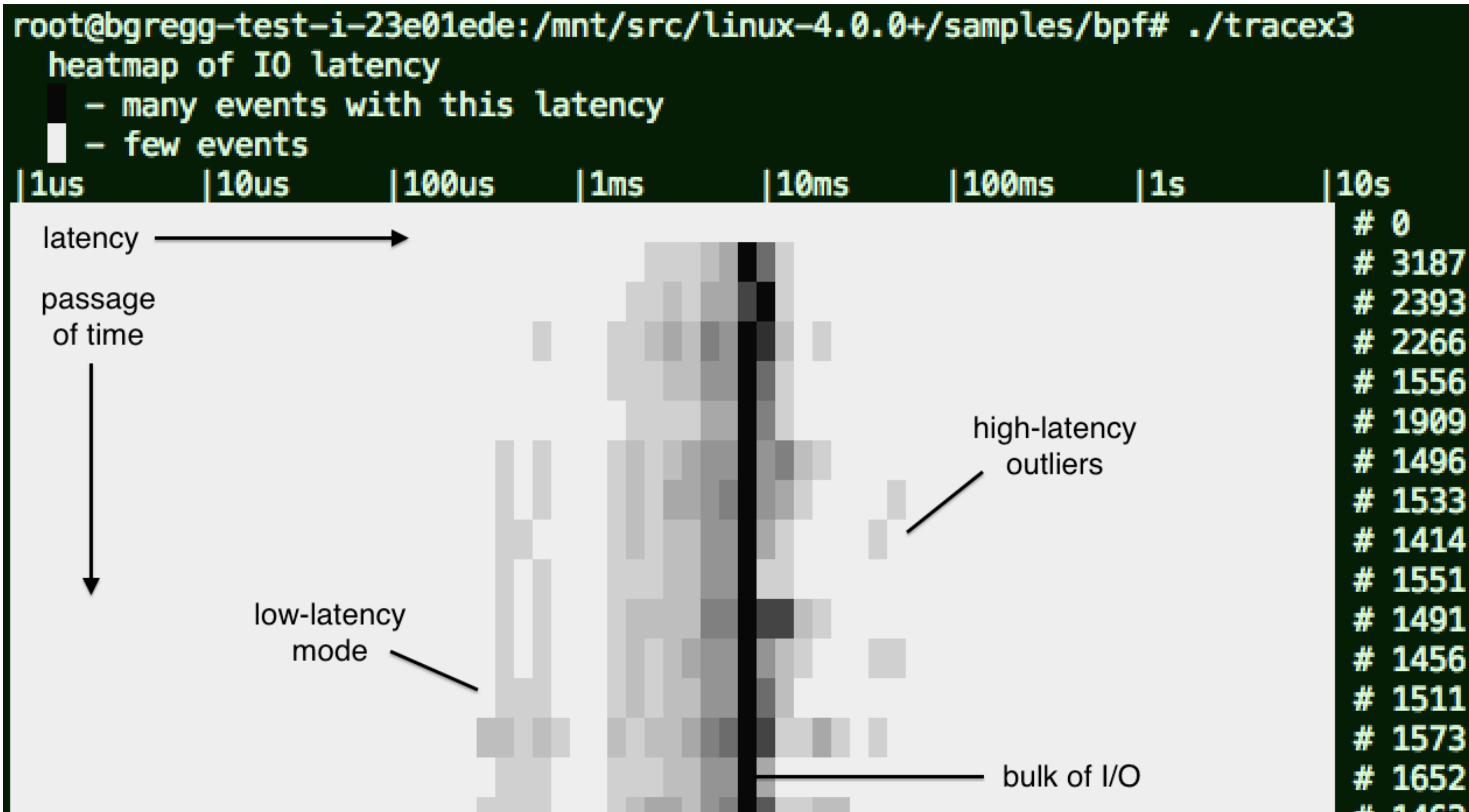
Performance Monitoring Counters

Eg, FreeBSD PMC groups for Intel Sandy Bridge:

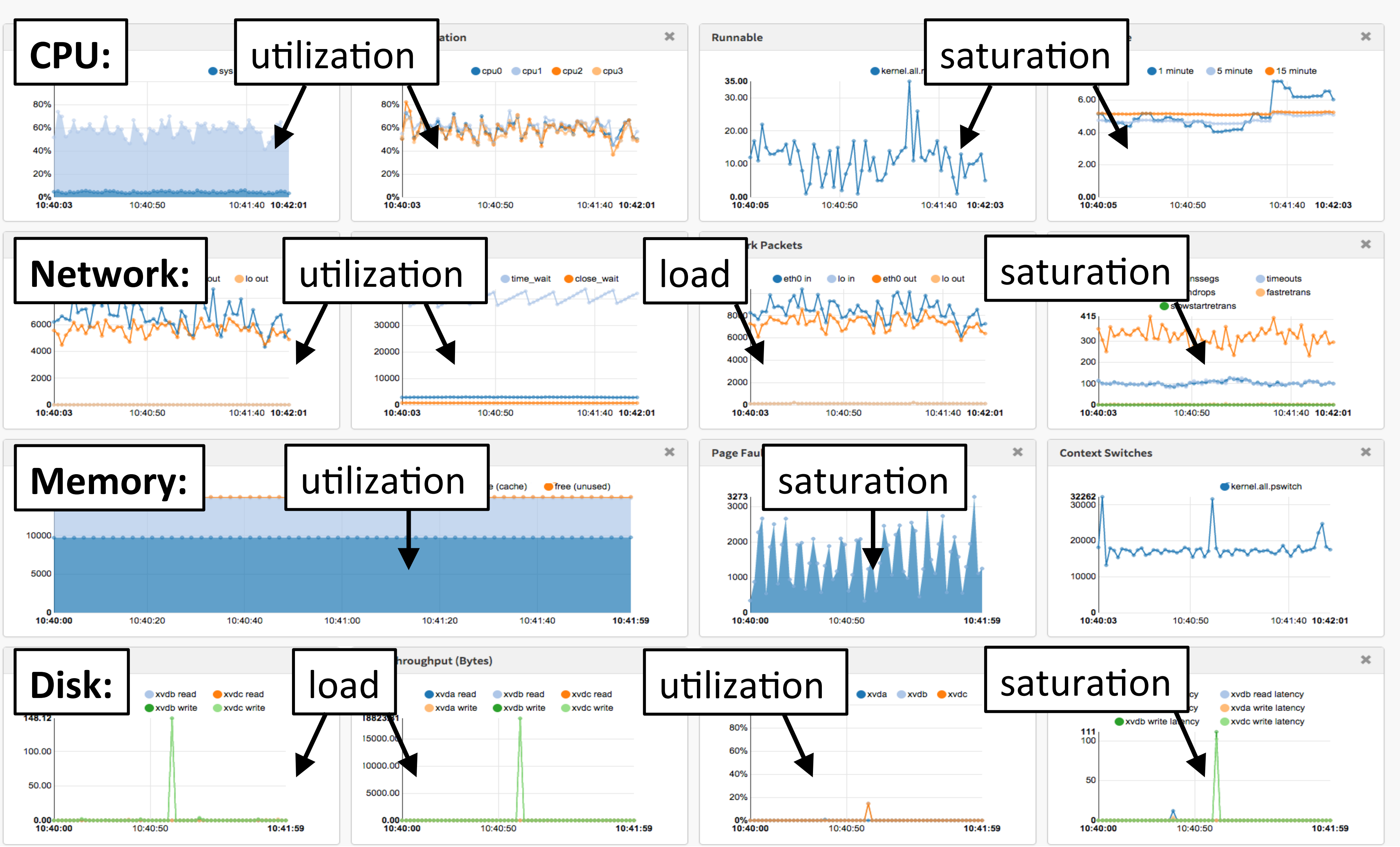


Visualizations

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



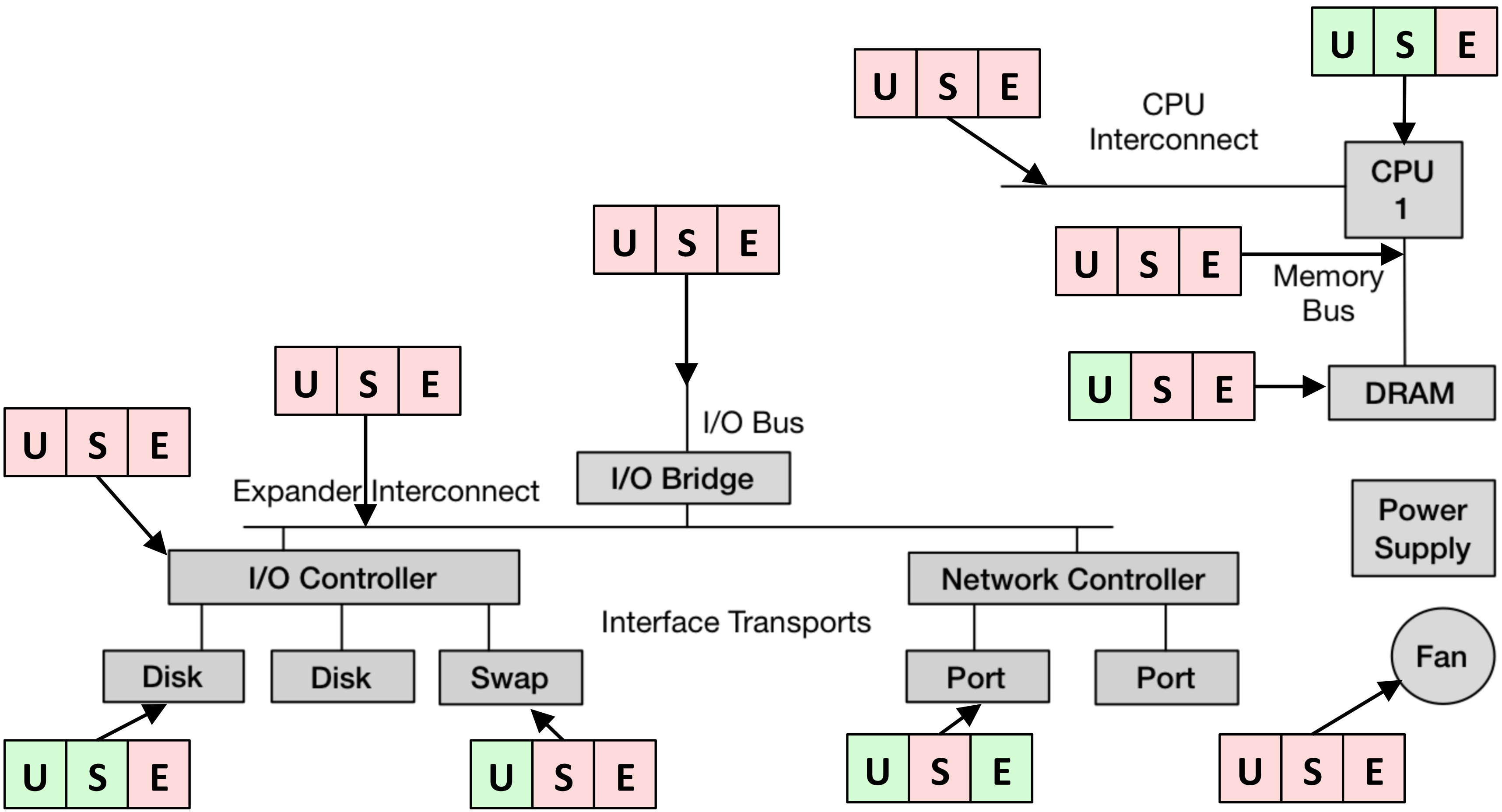
USE Method: eg, Netflix Vector



USE Method: To Do

Hardware

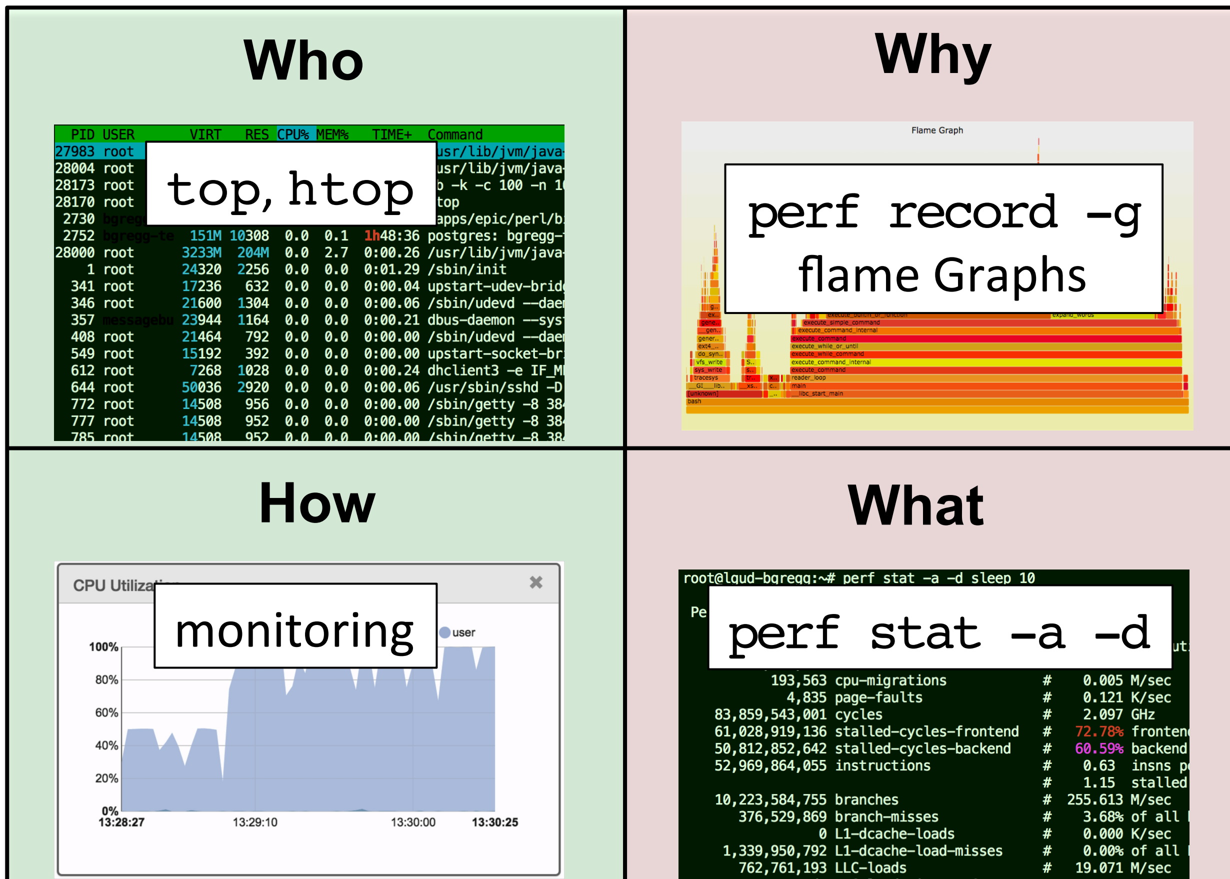
Showing what is and is not commonly measured



http://www.brendangregg.com/linuxperf.html 2015

CPU Workload Characterization: To Do

Showing what is and is not commonly measured

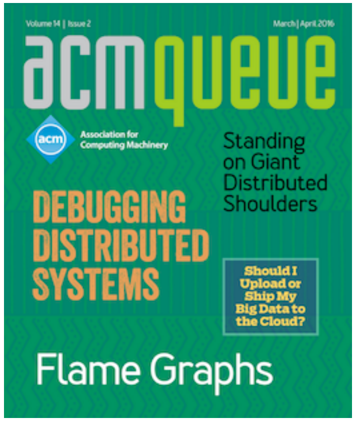


Summary

- 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

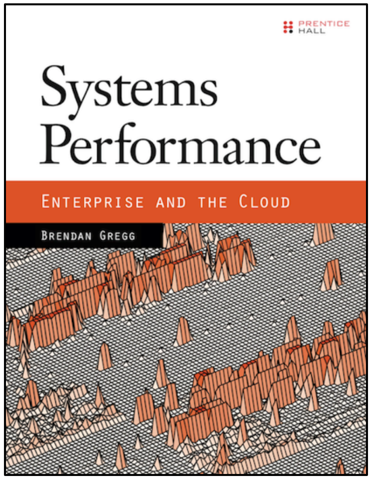
- USE Method
  - http://queue.acm.org/detail.cfm?id=2413037
  - http://www.brendangregg.com/usemethod.html
- TSA Method
  - http://www.brendangregg.com/tsamethod.html
- Off-CPU Analysis
  - http://www.brendangregg.com/offcpuanalysis.html
  - http://www.brendangregg.com/blog/2016-01-20/ebpf-offcpu-flame-graph.html
  - 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



ACM Applicative 2016

Jun, 2016

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



NETFLIX