SCALE 14x
The Fourteenth Annual
Southern California Linux Expo

Jan 2016

NETFLIX

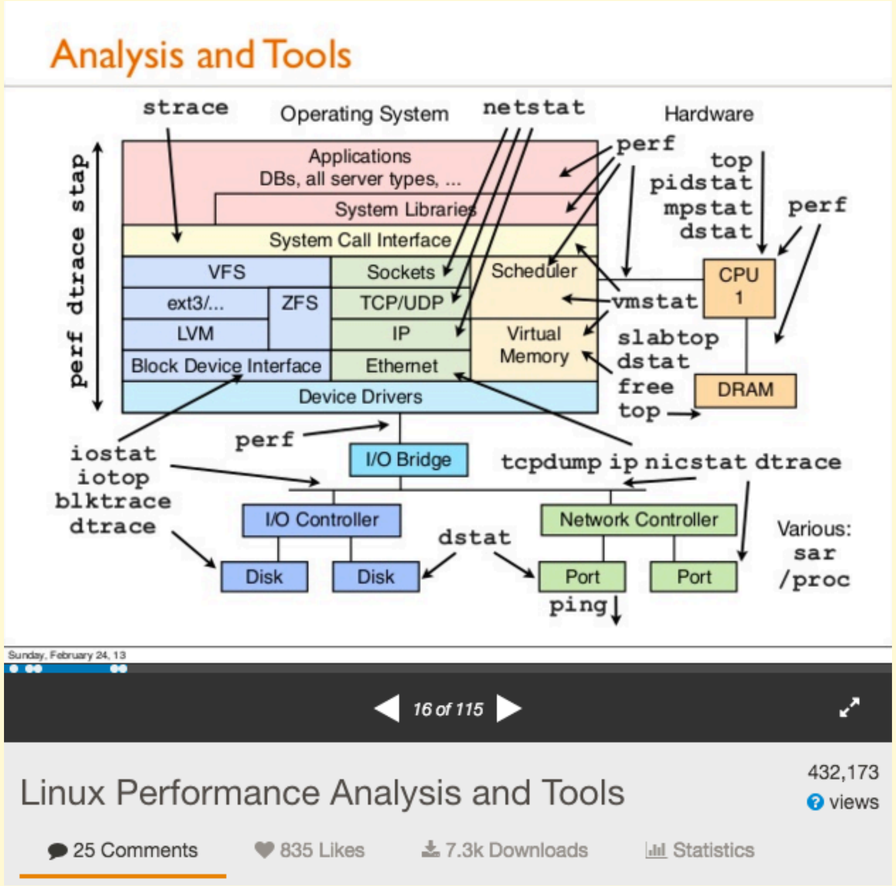
Broken Linux Performance Tools

Brendan Gregg

Senior Performance Architect, Netflix

Previously (SCaLE11x)

Working Linux performance tools:



This Talk (SCaLE14x)

Broken Linux performance tools:


Observability

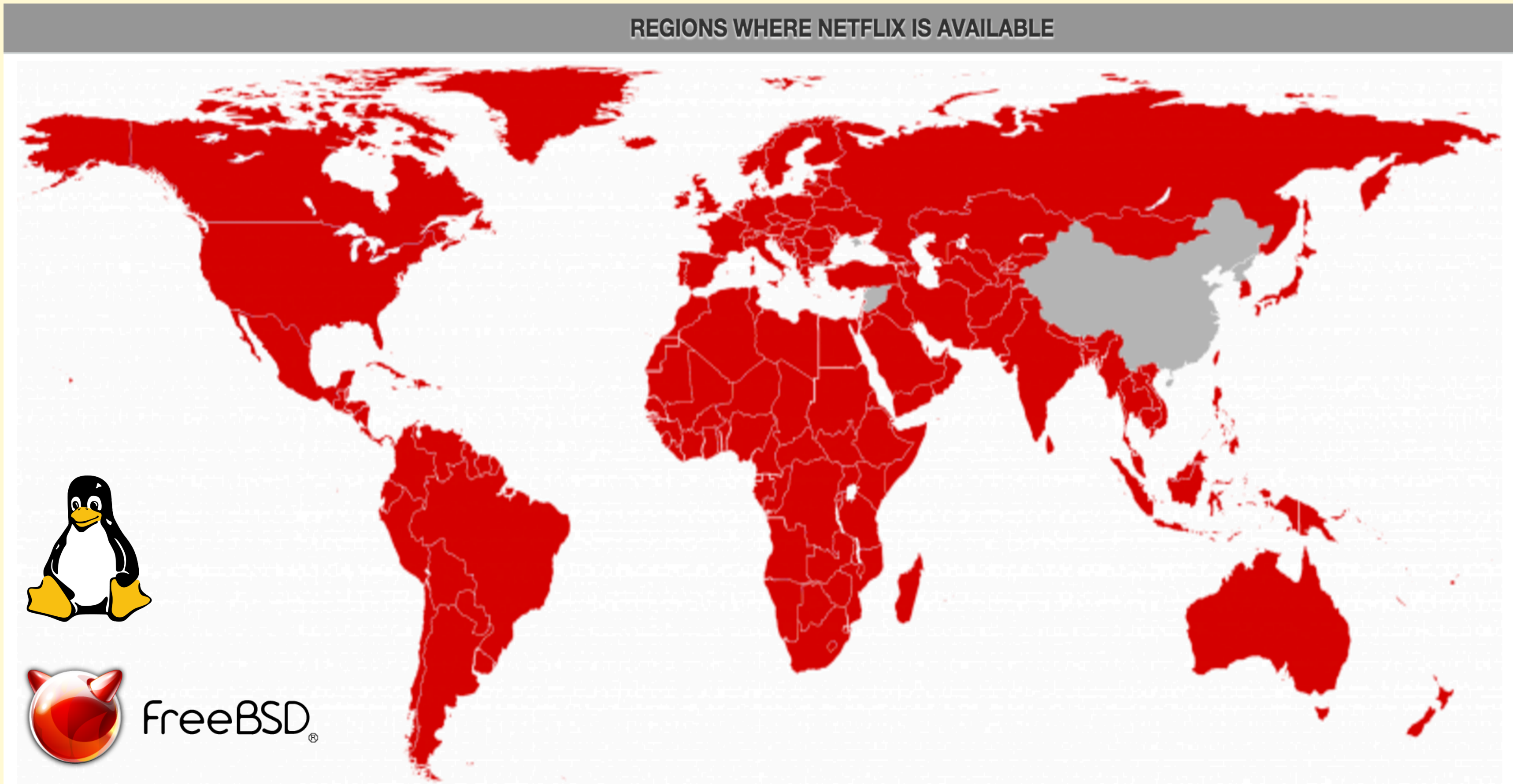

Benchmarking

Objectives:

- Bust assumptions about tools and metrics
- Learn how to verify and find missing metrics
- Avoid the common mistakes when benchmarking

Note: Current software is discussed, which could be fixed in the future (by you!)

NETFLIX



OBSERVABILITY

Load Averages

top %CPU

iowait

vmstat

Overhead

strace

Java Profilers

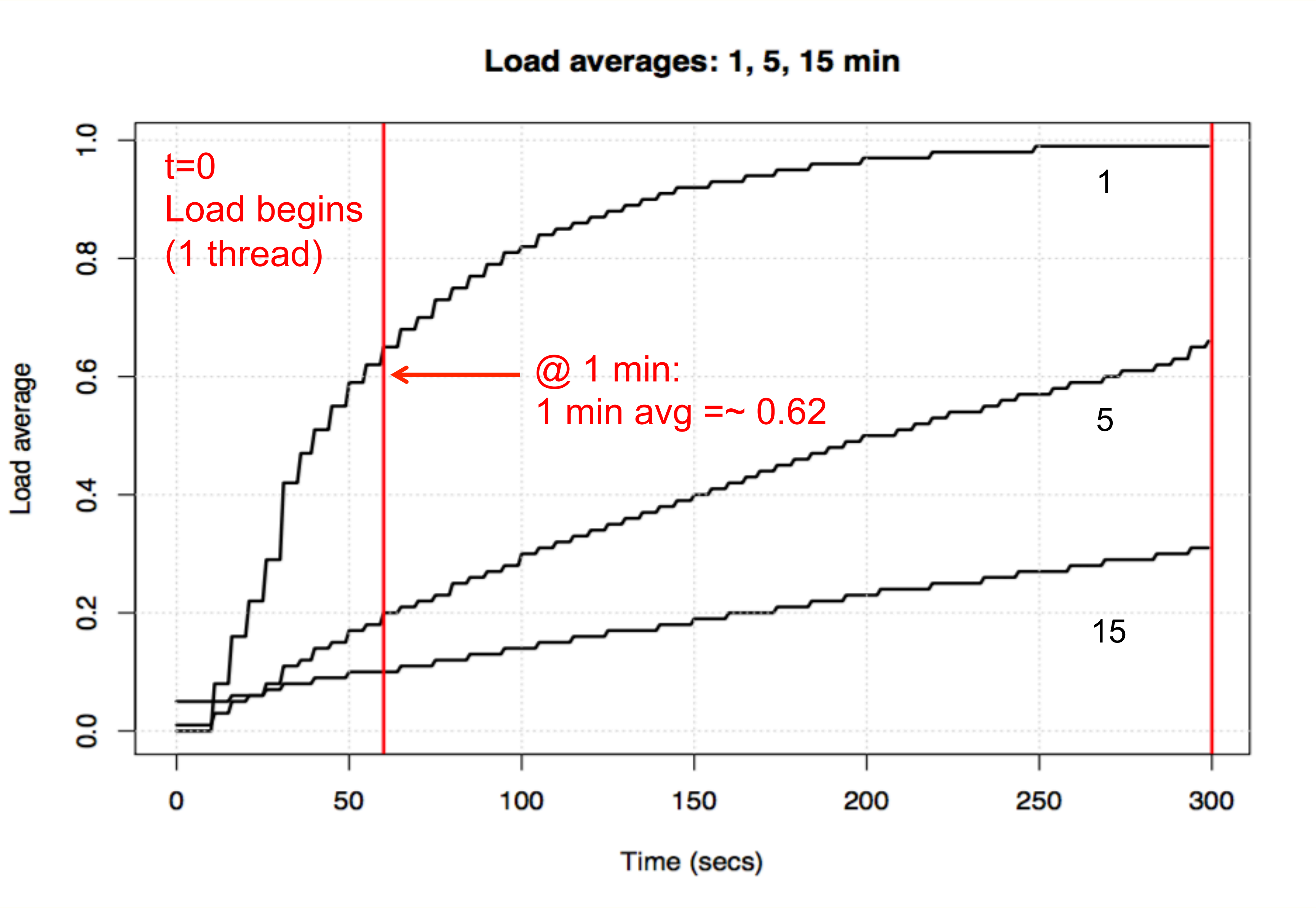
Monitoring

LOAD AVERAGES

Load Averages (1, 5, 15 min)

```
$ uptime
 22:08:07 up  9:05,  1 user,  load average: 11.42, 11.87, 12.12
```

- "load"
  - Usually CPU demand (run queue length/latency)
  - On Linux: CPU + uninterruptible I/O (e.g., disk)
- "average"
  - Exponentially damped moving sum
- "1, 5, and 15 minutes"
  - Constants used in the equation
- Don't study these for longer than 10 seconds



Load averages: 1, 5, 15 min
t=0
Load begins
(1 thread)
@ 1 min:
1 min avg =~ 0.62
1
5
15
Load average
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
Time (secs)

TOP %CPU

top %CPU

```
$ top - 20:15:55 up 19:12,  1 user,  load average: 7.96, 8.59, 7.05
Tasks: 470 total,   1 running, 468 sleeping,   0 stopped,   1 zombie
%Cpu(s): 28.1 us, 0.4 sy, 0.0 ni, 71.2 id, 0.0 wa, 0.0 hi, 0.1 si, 0.1 st
KiB Mem:  61663100 total, 61342588 used,   320512 free,     9544 buffers
KiB Swap:        0 total,        0 used,        0 free.  3324696 cached Mem

  PID USER      PR  NI    VIRT    RES   SHR S  %CPU %MEM    TIME+ COMMAND
11959 apiprod   20   0 81.731g 0.053t 14476 S 935.8 92.1 13568:22 java
12595 snmp      20   0   21240   3256  1392 S   3.6  0.0  2:37.23 snmp-pass
10447 snmp      20   0   51512   6028  1432 S   2.0  0.0  2:12.12 snmpd
18463 apiprod   20   0   23932   1972  1176 R   0.7  0.0  0:00.07 top
[…]
```

- Who is consuming CPU?
- And by how much?

top: Missing %CPU

- **Short-lived processes can be missing entirely**
  - Process creates and exits in-between sampling /proc.
    e.g., software builds.
  - Try atop(1), or sampling using perf(1)
- Short-lived processes may vanish on screen updates
  - I often use pidstat(1) on Linux instead, for concise scroll back

top: Misinterpreting %CPU

- Different top(1)s use **different calculations**
  - On different OSes, check the man page, and run a test!
- %CPU can mean:
  - A) Sum of per-CPU percents (0-Ncpu x 100%) consumed during the last interval
  - B) Percentage of total CPU capacity (0-100%) consumed during the last interval
  - C) (A) but historically damped (like load averages)
  - D) (B) " " "

top: %Cpu vs %CPU

```
$ top - 15:52:58 up 10 days, 21:58, 2 users, load average: 0.27, 0.53, 0.41
Tasks: 180 total,   1 running, 179 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.2 us, 24.5 sy, 0.0 ni, 67.2 id, 0.2 wa, 0.0 hi, 6.6 si, 0.4 st
KiB Mem:   2872448 total,  2778160 used,    94288 free,    31424 buffers
KiB Swap:  4151292 total,       76 used,  4151216 free.  2411728 cached Mem

  PID USER      PR  NI    VIRT   RES   SHR S  %CPU %MEM     TIME+ COMMAND
12678 root      20   0   96812  1100   912 S 100.4  0.0   0:23.52 iperf
12675 root      20   0  170544  1096   904 S  88.8  0.0   0:20.83 iperf
  215 root      20   0       0     0     0 S   0.3  0.0   0:27.73 jbd2/sda1-8
[…]
```

- This 4 CPU system is consuming:
  - 130% total CPU, via %Cpu(s)
  - 190% total CPU, via %CPU
- Which one is right? Is either?
  - "A man with one watch knows the time; with two he's never sure"

CPU Summary Statistics

- %Cpu row is from /proc/stat
- linux/Documentation/cpu-load.txt:

```
In most cases the `/proc/stat' information reflects
the reality quite closely, however due to the nature
of how/when the kernel collects this data
sometimes it can not be trusted at all.
```

- /proc/stat is used by everything for CPU stats

%CPU

What is %CPU anyway?

- "Good" %CPU:
  - **Retiring instructions** (provided they aren't a spin loop)
  - High IPC (Instructions-Per-Cycle)
- "Bad" %CPU:
  - **Stall cycles** waiting on resources, usually memory I/O
  - Low IPC
  - Buying faster processors may make little difference
- %CPU alone is ambiguous
  - Would love top(1) to split %CPU into cycles retiring vs stalled
  - Although, it gets worse…

CPU Speed Variation

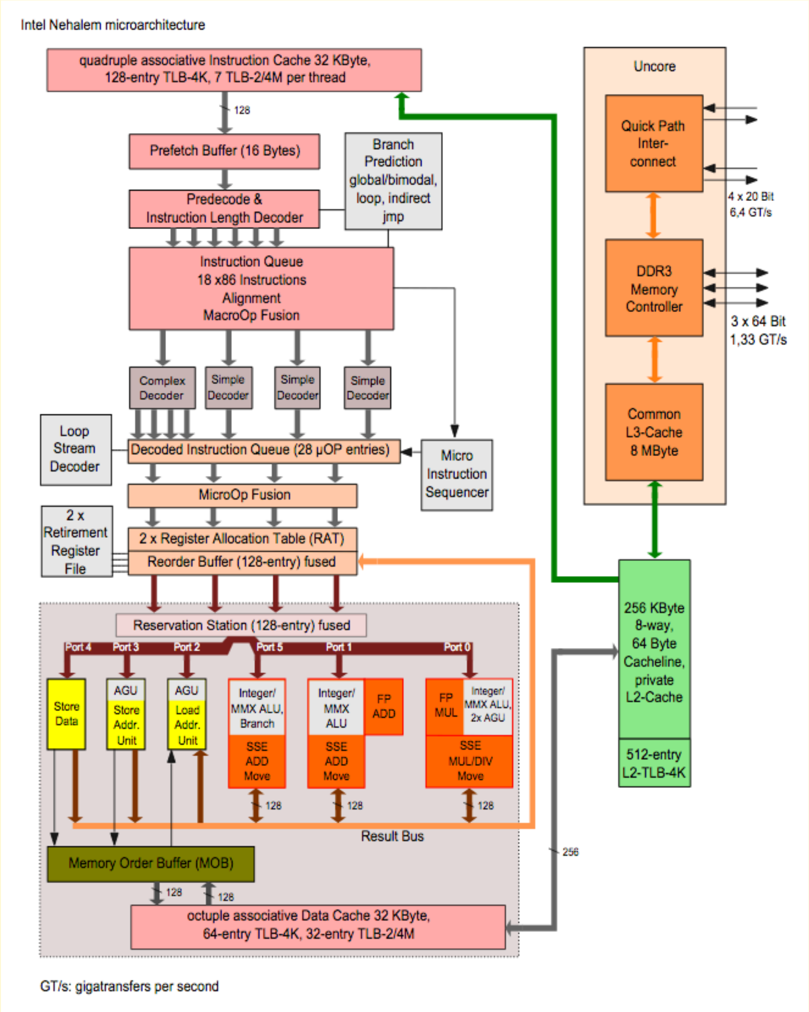
- **Clock speed can vary** thanks to:
  - Intel Turbo Boost: by hardware, based on power, temp, etc
  - Intel Speed Step: by software, controlled by the kernel
- %CPU is still ambiguous, given IPC

80% CPU (1.6 IPC)	may not ==	4 x 20% CPU (1.6 IPC)

- Need to know the clock speed as well
  - 80% CPU (@3000MHz) != 4 x 20% CPU (@1600MHz)
- CPU counters nowadays have "reference cycles"

Out-of-order Execution

- CPUs execute uops out-of-order and in parallel across multiple functional units
- %CPU doesn't account for how many units are active
- Accounting each cycles as "stalled" or “retiring" is a simplification



https://upload.wikimedia.org/wikipedia/commons/6/64/Intel_Nehalem_arch.svg

I/O WAIT

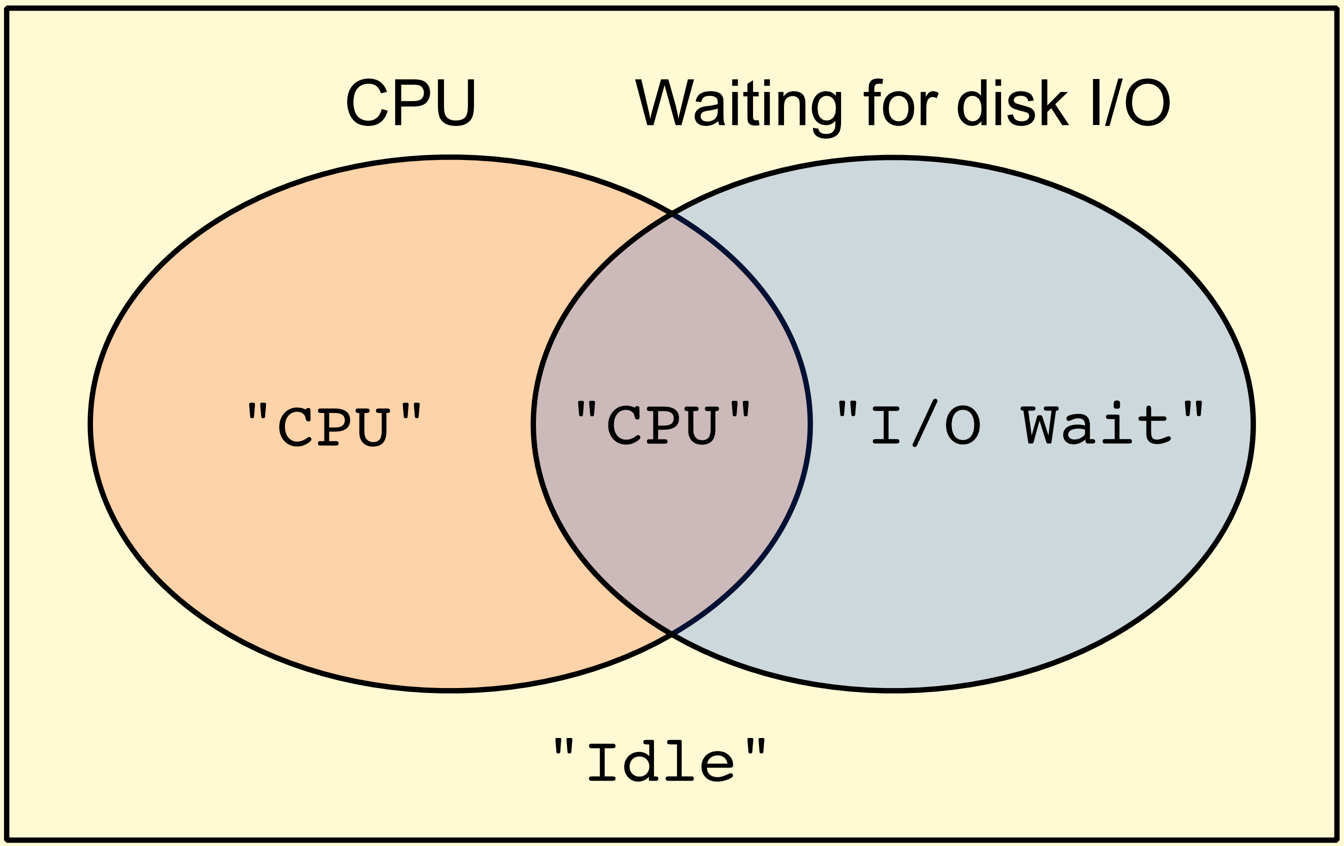
I/O Wait

```
$ mpstat -P ALL 1
08:06:43 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
08:06:44 PM  all   53.45    0.00    3.77    0.00    0.00    0.39    0.13    0.00   42.26
[…]
```

- Suggests system is disk I/O bound, but often misleading
- Comparing I/O wait between system A and B:
  - **higher might be bad**: slower disks, more blocking
  - **lower might be bad**: slower processor and architecture consumes more CPU, obscuring I/O wait
- Can be very useful when understood: another idle state

I/O Wait Venn Diagram

Per CPU:

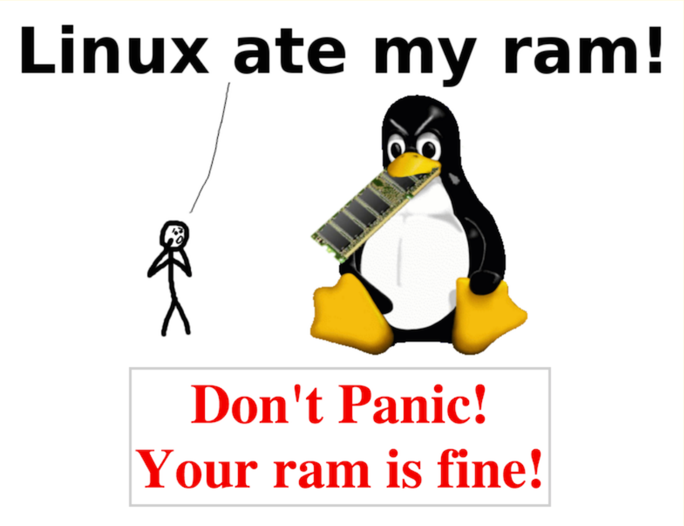


FREE MEMORY

Free Memory

```
$ free -m
                    total       used       free     shared    buffers     cached
Mem:                 3750       1111       2639          0        147        527
-/+ buffers/cache:               436       3313
Swap:                   0          0          0
```

- "free" is near-zero: I'm running out of memory!
  - No, it's in the file system cache, and is still free for apps to use
- Linux free(1) explains it, but other tools, e.g. vmstat(1), don't
  - Some file systems (e.g., ZFS) may not be shown in the system's cached metrics at all



www.linuxatemyram.com

VMSTAT

vmstat(1)

```
$ vmstat –Sm 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 8  0      0   1620    149    552    0    0     1   179   77   12 25 34  0  0
 7  0      0   1598    149    552    0    0     0     0  205  186 46 13  0  0
 8  0      0   1617    149    552    0    0     0     8  210  435 39 21  0  0
 8  0      0   1589    149    552    0    0     0     0  218  219 42 17  0  0
[…]
```

- Linux: first line has *some* summary since boot values — confusing!
- This system-wide summary is missing networking

NETSTAT -S

netstat -s

```
$ netstat -s
Ip:
    7962754 total packets received
    8 with invalid addresses
    0 forwarded
    0 incoming packets discarded
    7962746 incoming packets delivered
    8019427 requests sent out
Icmp:
    382 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 125
        timeout in transit: 257
    3410 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 3410
IcmpMsg:
        InType3: 125
        InType11: 257
        OutType3: 3410
Tcp:
    17337 active connections openings
    395515 passive connection openings
    8953 failed connection attempts
    240214 connection resets received
    3 connections established
    7198375 segments received
    7504939 segments send out
    62696 segments retransmited
    10 bad segments received.
 1072 resets sent
    InCsumErrors: 5
Udp:
    759925 packets received
    3412 packets to unknown port received.
    0 packet receive errors
    784370 packets sent
UdpLite:
TcpExt:
    858 invalid SYN cookies received
    8951 resets received for embryonic SYN_RECV sockets
    14 packets pruned from receive queue because of socket buffer overrun
    6177 TCP sockets finished time wait in fast timer
    293 packets rejects in established connections because of timestamp
    733028 delayed acks sent
    89 delayed acks further delayed because of locked socket
    Quick ack mode was activated 13214 times
    336520 packets directly queued to recvmsg prequeue.
    43964 packets directly received from backlog
    11406012 packets directly received from prequeue
    1039165 packets header predicted
    7066 packets header predicted and directly queued to user
```

```
    1428960 acknowledgments not containing data received
    1004791 predicted acknowledgments
    1 times recovered from packet loss due to fast retransmit
    5044 times recovered from packet loss due to SACK data
    2 bad SACKs received
    Detected reordering 4 times using SACK
    Detected reordering 11 times using time stamp
    13 congestion windows fully recovered
    11 congestion windows partially recovered using Hoe heuristic
    TCPDSACKUndo: 39
    2384 congestion windows recovered after partial ack
    228 timeouts after SACK recovery
    100 timeouts in loss state
    5018 fast retransmits
    39 forward retransmits
    783 retransmits in slow start
    32455 other TCP timeouts
    TCPLossProbes: 30233
    TCPLossProbeRecovery: 19070
    992 sack retransmits failed
    18 times receiver scheduled too late for direct processing
    705 packets collapsed in receive queue due to low socket buffer
    13658 DSACKs sent for old packets
    8 DSACKs sent for out of order packets
    13595 DSACKs received
    33 DSACKs for out of order packets received
    32 connections reset due to unexpected data
    108 connections reset due to early user close
    1608 connections aborted due to timeout
    TCPSACKDiscard: 4
    TCPDSACKIgnoredOld: 1
    TCPDSACKIgnoredNoUndo: 8649
    TCPSpuriousRTOs: 445
    TCPSackShiftFallback: 8588
    TCPRcvCoalesce: 95854
    TCPOFOQueue: 24741
    TCPOFOMerge: 8
    TCPChallengeACK: 1441
    TCPSYNChallenge: 5
    TCPSpuriousRtxHostQueues: 1
    TCPAutoCorking: 4823
IpExt:
    InOctets: 1561561375
    OutOctets: 1509416943
    InNoECTPkts: 8201572
    InECT1Pkts: 2
    InECT0Pkts: 3844
    InCEPkts: 306
```

netstat -s

- Many metrics on Linux (can be over 200)
  - Still doesn't include everything: getting better, but don't assume everything is there
- Includes typos & inconsistencies
  - Might be more readable to:
    `cat /proc/net/snmp /proc/net/netstat`
- Totals since boot can be misleading
  - On Linux, -s needs -c support
- Often no documentation outside kernel source code
  - Requires expertise to comprehend

DISK METRICS

Disk Metrics

- **All disk metrics are misleading**
- Disk %utilization / %busy
  - Logical devices (volume managers) and individual disks can process I/O in parallel, and may accept more I/O at 100%
- Disk IOPS
  - High IOPS is "bad"? That depends…
- Disk latency
  - Does it matter? File systems and volume managers try hard to hide latency and make it asynchronous
  - Better measuring latency via application->FS calls

FS CACHE METRICS

FS Cache Metrics

- Size metrics exist: free -m
- Activity metrics are missing: e.g., hit/miss ratio
- Hacking stats using ftrace (/eBPF):

```
# ./cachestat 1
Counting cache functions... Output every 1 seconds.
    HITS   MISSES  DIRTIES    RATIO   BUFFERS_MB   CACHE_MB
     210      869        0    19.5%            2        209
     444     1413        0    23.9%            8        210
     471     1399        0    25.2%           12        211
     403     1507        3    21.1%           18        211
     967     1853        3    34.3%           24        212
[...]
```

What You Can Do

- Verify and understand existing metrics
  - Even %CPU can be misleading
  - Cross check with another tool & backend
  - Test with known workloads
  - Read the source, including comments
  - Use "known to be good" metrics to sanity test others
- Find missing metrics
  - Follow the USE Method, and other methodologies
  - Draw a functional diagram
- Burn it all down and start again from scratch?

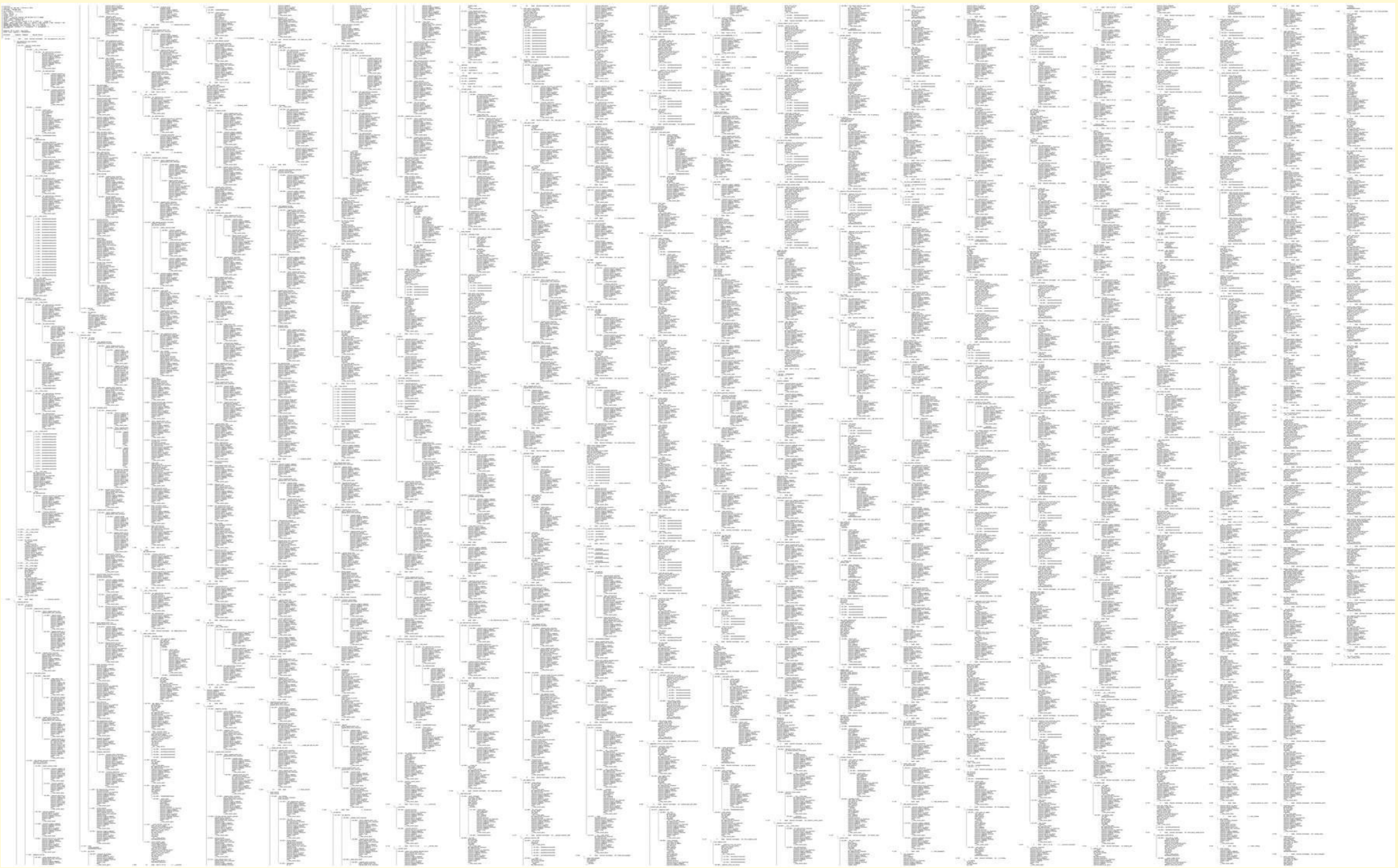
PROFILERS

Linux `perf`

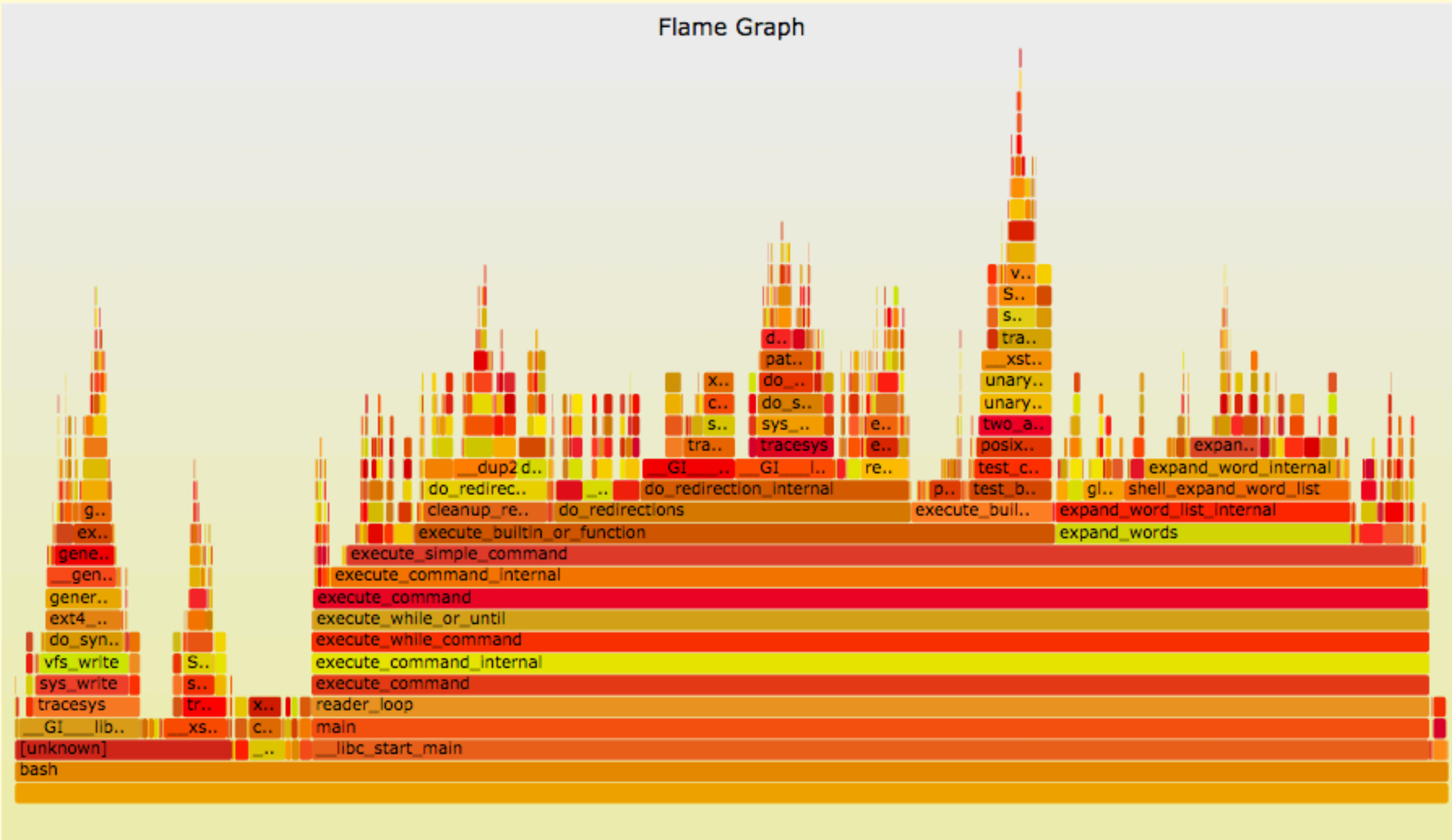
- Can sample stack traces and summarize output:

```
# perf report -n -stdio
[…]
# Overhead       Samples  Command      Shared Object                             Symbol
# ........  ............  .......  .................  ..............................
#
    20.42%           605     bash  [kernel.kallsyms]  [k] xen_hypercall_xen_version
                     |
                     --- xen_hypercall_xen_version
                         check_events
                        |
                        |--44.13%-- syscall_trace_enter
                        |           tracesys
                        |           |
                        |           |--35.58%-- __GI___libc_fcntl
                        |           |           |
                        |           |           |--65.26%-- do_redirection_internal
                        |           |           |           do_redirections
                        |           |           |           execute_builtin_or_function
                        |           |           |           execute_simple_command
[… ~13,000 lines truncated …]
```

Too Much Output

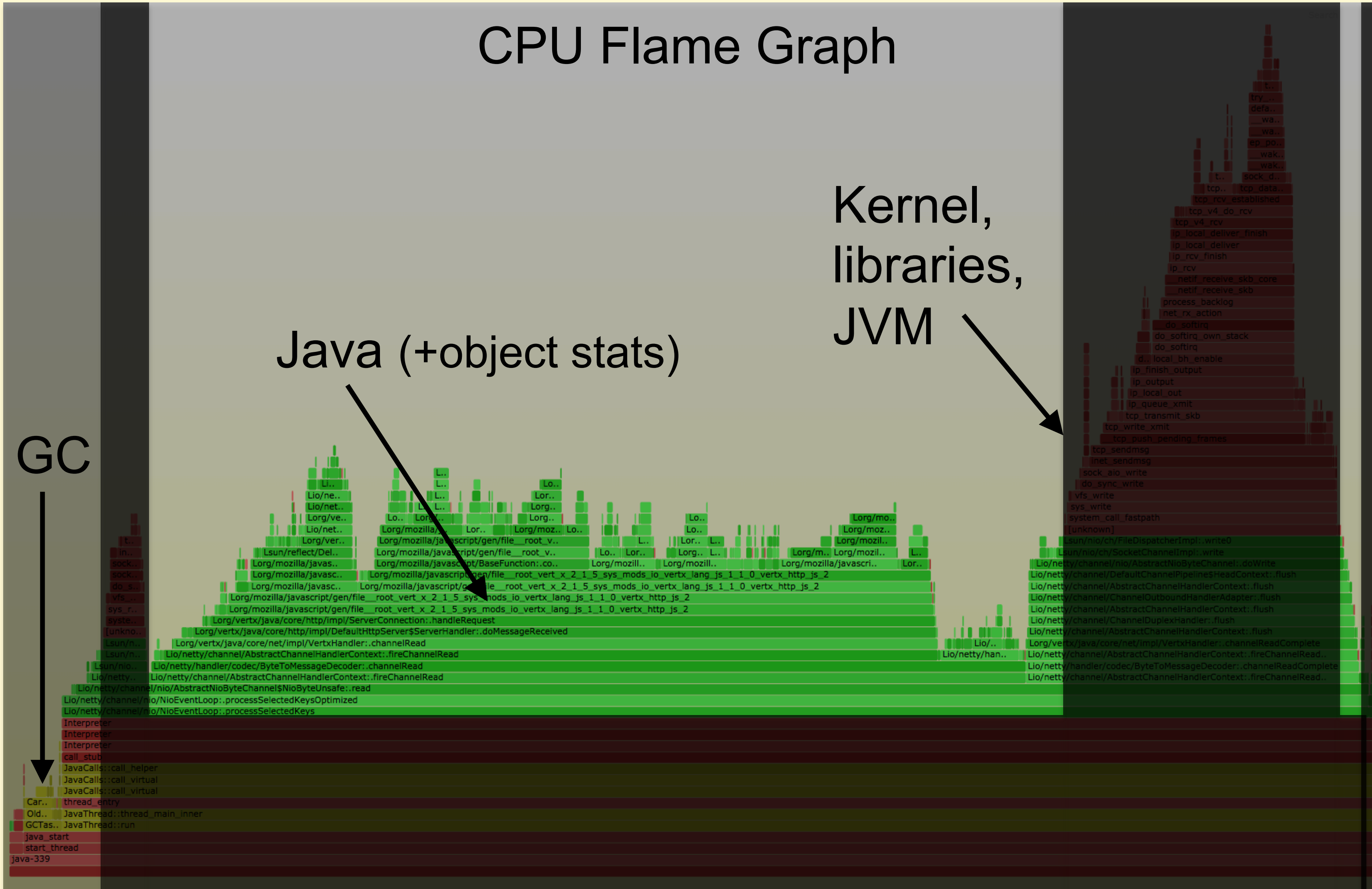


… as a Flame Graph



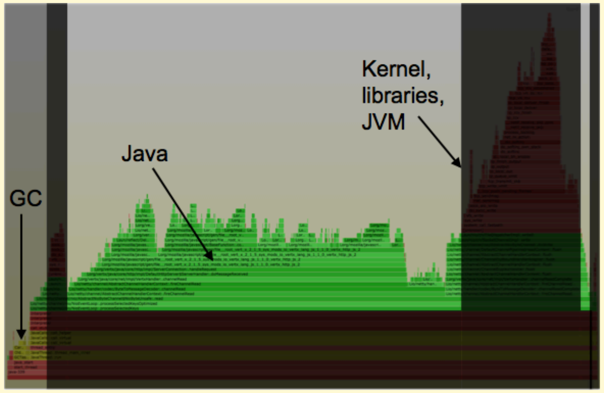
PROFILER VISIBILITY

Java Profilers

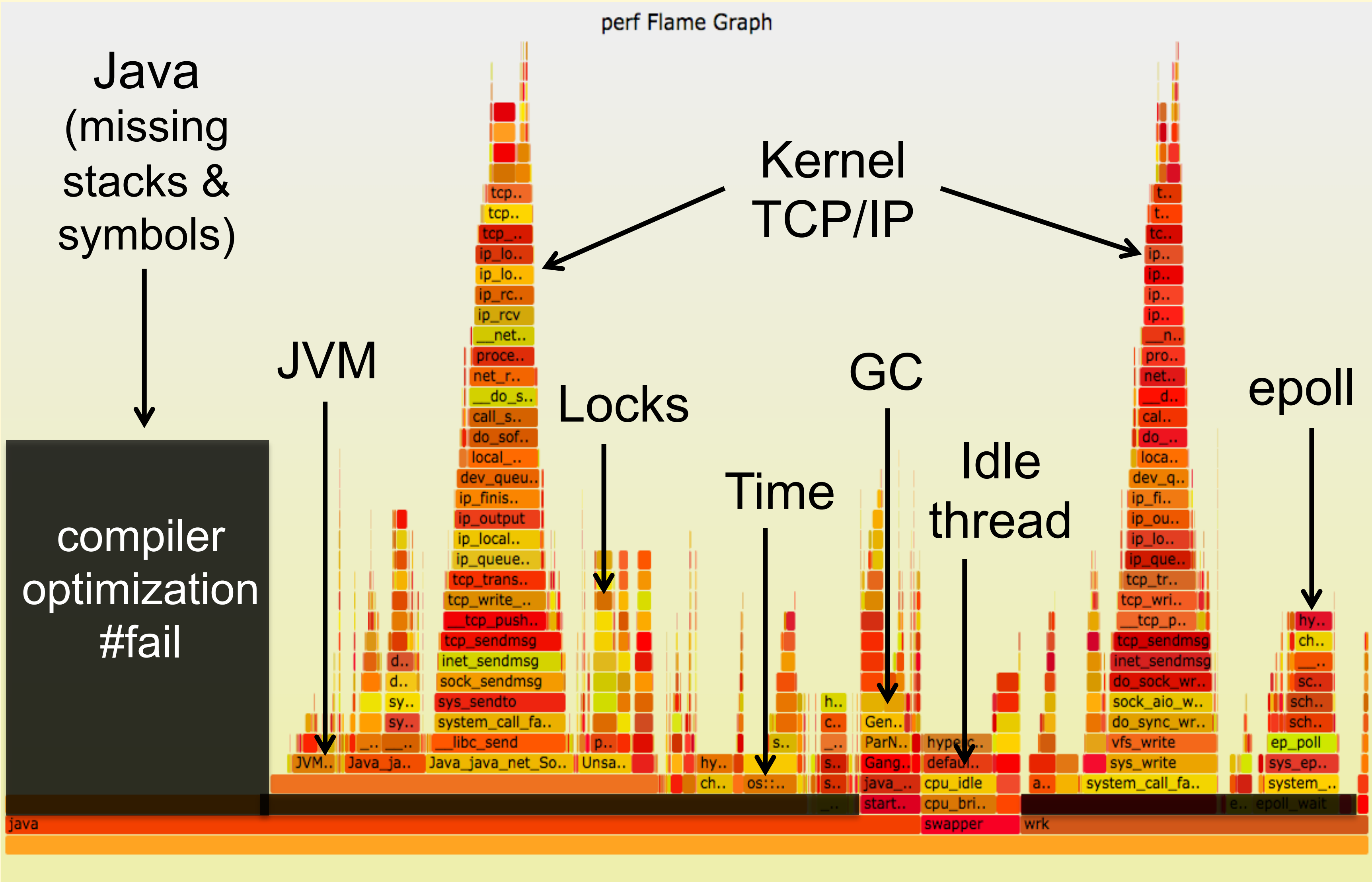


Java Profilers

- Typical problems:
  - Sampling at safepoints (skew)
  - Method tracing observer effect
  - RUNNING != on-CPU (e.g., epoll)
  - Missing GC or JVM CPU time
- **Inaccurate** (skewed) and **incomplete** profiles
- Let's try a system profiler?



System Profilers with Java (x86)



COMPILER OPTIMIZATIONS

Broken System Stack Traces

- Broken stacks (1 or 2 levels deep, junk values):

```
# perf record –F 99 –a –g – sleep 30; perf script
[…]
java  4579 cpu-clock:
  ffffffff8172adff tracesys ([kernel.kallsyms])
      7f4183bad7ce pthread_cond_timedwait@@GLIBC_2…

java  4579 cpu-clock:
      7f417908c10b [unknown] (/tmp/perf-4458.map)

java  4579 cpu-clock:
      7f4179101c97 [unknown] (/tmp/perf-4458.map)
```

- On x86 (x86_64), hotspot reuses the frame pointer register (RBP) as general purpose (a "compiler optimization"), which *once upon a time* made sense
- gcc has **`-fno-omit-frame-pointer`** to avoid this
  - JDK8u60+ now has this as -XX:+PreserveFramePoiner

Missing Symbols

- Missing symbols may show up as hex; e.g., Linux perf:

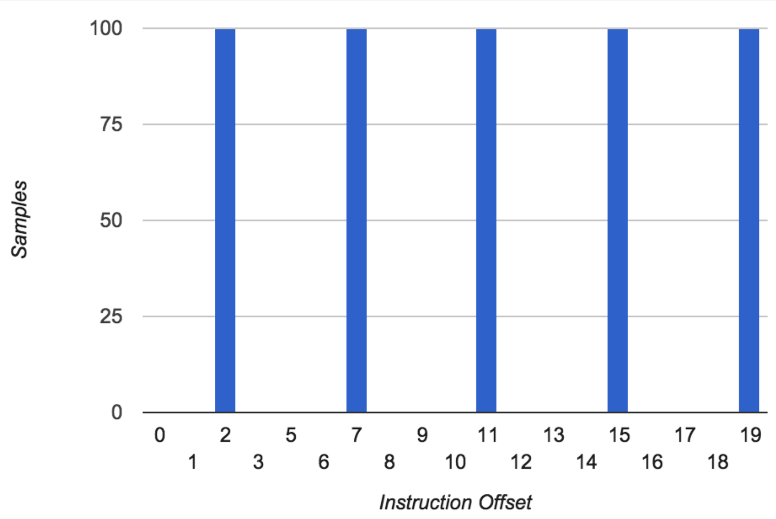
```
# perf script
Failed to open /tmp/perf-8131.map, continuing without symbols
[…]
java 8131 cpu-clock:
    7fff76f2dce1 [unknown] ([vdso])
    7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm…
    7fd301861e46 [unknown] (/tmp/perf-8131.map)
[…]
```

- For applications, install debug symbol package
- For JIT'd code, Linux perf already looks for an externally provided symbol file: /tmp/perf-PID.map
  - Find a way to do this for your runtime

INSTRUCTION PROFILING

Instruction Profiling

```
# perf annotate -i perf.data.noplooper --stdio
 Percent |   Source code & Disassembly of noplooper
---------------------------------------------------
         :   Disassembly of section .text:
         :
         :   00000000004004ed <main>:
    0.00 :     4004ed:    push   %rbp
    0.00 :     4004ee:    mov    %rsp,%rbp
   20.86 :     4004f1:    nop
    0.00 :     4004f2:    nop
    0.00 :     4004f3:    nop
    0.00 :     4004f4:    nop
   19.84 :     4004f5:    nop
    0.00 :     4004f6:    nop
    0.00 :     4004f7:    nop
    0.00 :     4004f8:    nop
   18.73 :     4004f9:    nop
    0.00 :     4004fa:    nop
    0.00 :     4004fb:    nop
    0.00 :     4004fc:    nop
   19.08 :     4004fd:    nop
    0.00 :     4004fe:    nop
    0.00 :     4004ff:    nop
    0.00 :     400500:    nop
   21.49 :     400501:    jmp    4004f1 <main+0x4>
```



- Often broken nowadays due to skid, out-of-order execution, and sampling the resumption instruction
- Better with PEBS support

What You Can Do

- Do stack trace profiling
  - Get stack traces to work
  - Get symbols to work
  - This all may be a lot of work. It's worth it!
- Make CPU flame graphs!

OVERHEAD

tcpdump

```
$ tcpdump -i eth0 -w /tmp/out.tcpdump
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
^C7985 packets captured
8996 packets received by filter
1010 packets dropped by kernel
```

- **Packet tracing doesn't scale**. Overheads:
  - CPU cost of per-packet tracing (improved by [e]BPF)
    - Consider CPU budget per-packet at 10/40/100 GbE
  - Transfer to user-level (improved by ring buffers)
  - File system storage (more CPU, and disk I/O)
  - Possible additional network transfer
- Can also drop packets when overloaded
- You should only trace send/receive as a last resort
  - I solve problems by tracing lower frequency TCP events

STRACE

strace

- Before:

```
$ dd if=/dev/zero of=/dev/null bs=1 count=500k
[…]
512000 bytes (512 kB) copied, 0.103851 s, 4.9 MB/s
```

- After:

```
$ strace -eaccept dd if=/dev/zero of=/dev/null bs=1 count=500k
[…]
512000 bytes (512 kB) copied, 45.9599 s, 11.1 kB/s
```

- 442x slower. This is worst case.
- strace(1) pauses the process twice for each syscall. This is like putting metering lights on your app.
  - "BUGS: A traced process runs slowly." – strace(1) man page

PERF_EVENTS

perf_events

- Buffered tracing helps, but you can still trace too much:

```
# perf record -e sched:sched_switch -a -g -- sleep 1
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 100.212 MB perf.data (486550 samples) ]
```

- Overhead = event instrumentation cost X event frequency
- Costs
  - Higher: event dumps (perf.data), stack traces, copyin/outs
  - Lower: counters, in-kernel aggregations (ftrace, eBPF)
- Frequencies
  - Higher: instructions, scheduler, malloc/free, Java methods
  - Lower: process creation & destruction, disk I/O (usually)

VALGRIND

Valgrind

- A suite of tools including an extensive leak detector

    "Your program will run much slower
    (eg. 20 to 30 times) than normal"

    – http://valgrind.org/docs/manual/quick-start.html

- To its credit it does warn the end user

JAVA PROFILERS

Java Profilers

- Some Java profilers have two modes:
  - Sampling stacks: eg, at 100 Hertz
  - Tracing methods: instrumenting and timing every method
- Method timing has been described as "highly accurate", despite slowing the target by **up to 1000x**!
- For more about Java profiler issues, see Nitsan Wakart's QCon2015 talk "Profilers are Lying Hobbitses"

What You Can Do

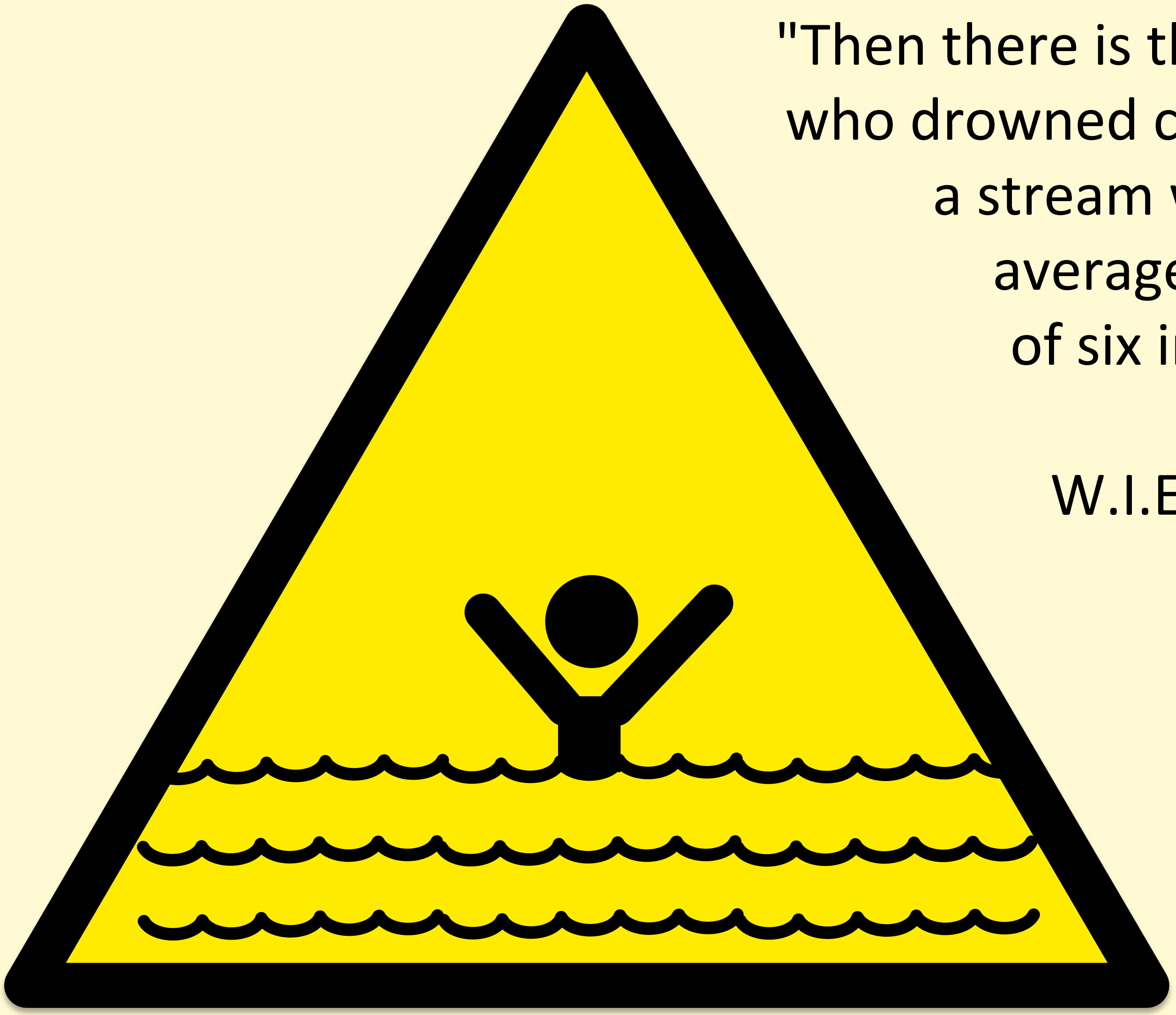
- Understand how the profiler works
  - Measure overhead
  - Know the frequency of instrumented events
- Use in-kernel summaries (ftrace, eBPF)
  - < 10,000 events/sec, probably ok
  - > 100,000 events/sec, overhead may start to be measurable

MONITORING

Monitoring

- By now you should recognize these pathologies:
  - Let's just graph the system metrics!
    - That's not the problem that needs solving
  - Let's just trace everything and post process!
    - Now you have one million problems per second
- Monitoring adds additional problems:
  - Let's have a cloud-wide dashboard update per-second!
    - From every instance? Packet overheads?
  - Now we have billions of metrics!

STATISTICS



"Then there is the man who drowned crossing a stream with an average depth of six inches."

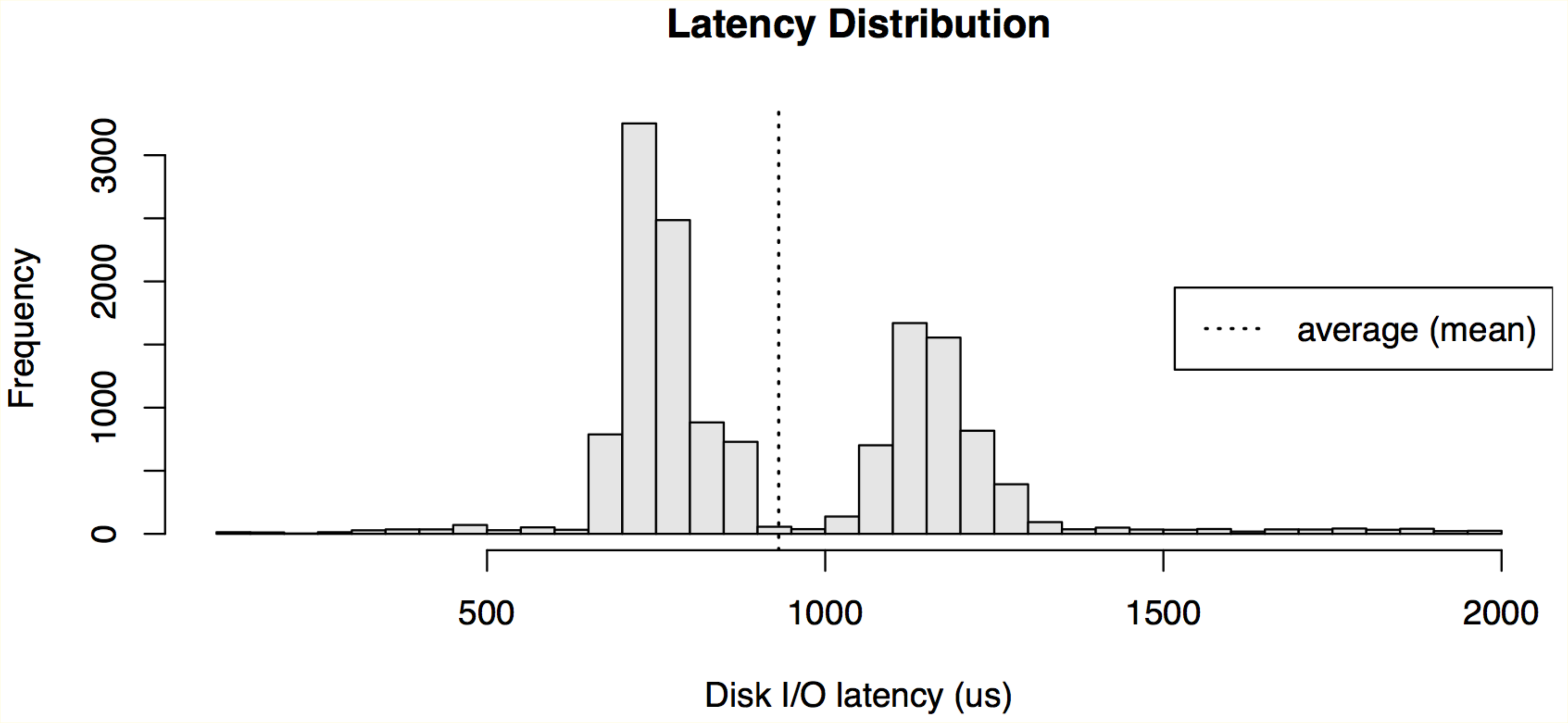
– W.I.E. Gates

Statistics

- Averages can be misleading
  - Hide latency outliers
  - Per-minute averages can hide multi-second issues
- Percentiles can be misleading
  - Probability of hitting 99.9th latency may be more than 1/1000 after many dependency requests
- Show the distribution:
  - Summarize: histogram, density plot, frequency trail
  - Over-time: scatter plot, heat map

Average Latency

- When the index of central tendency isn't…



VISUALIZATIONS

Traffic Lights

RED == bad, GREEN == good

```
$ dstat 1
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  3   0  97   0   0   0|  48B  842B|   0     0 |   0     0 |  12    10
 24  76   0   0   0   0|   0     0 | 104B  892B|   0     0 | 255    20
 14  86   0   0   0   0|   0     0 | 285B  899B|   0     0 | 264    29
 20  80   0   0   0   0|   0     0 | 104B  428B|   0     0 | 258    24
 20  80   0   0   0   0|   0     0 | 156B  738B|   0     0 | 258    22
```

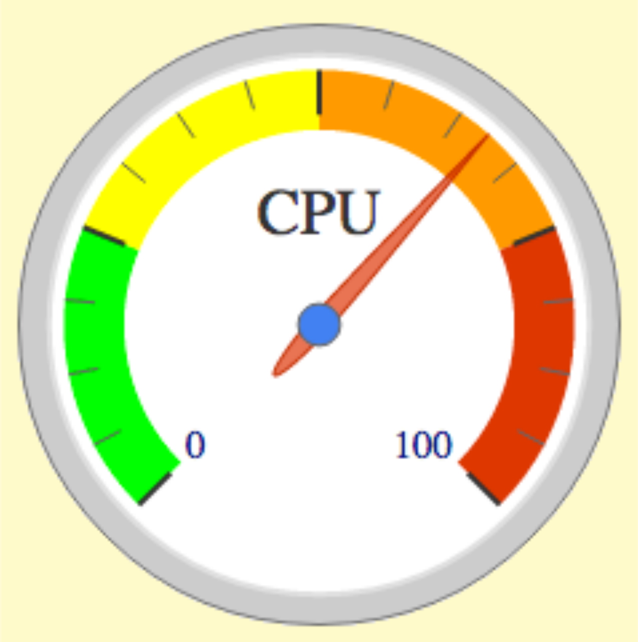
```
$ htop

  CPU[||||||||||||||||||||||||100.0%]     Tasks: 38, 5 thr; 2 running
  Mem[|||||||||||||||||||||133/592MB]     Load average: 0.59 0.18 0.10
  Swp[                         0/0MB]     Uptime: 367 days(!), 10:38:17
```

…misleading for *subjective* metrics

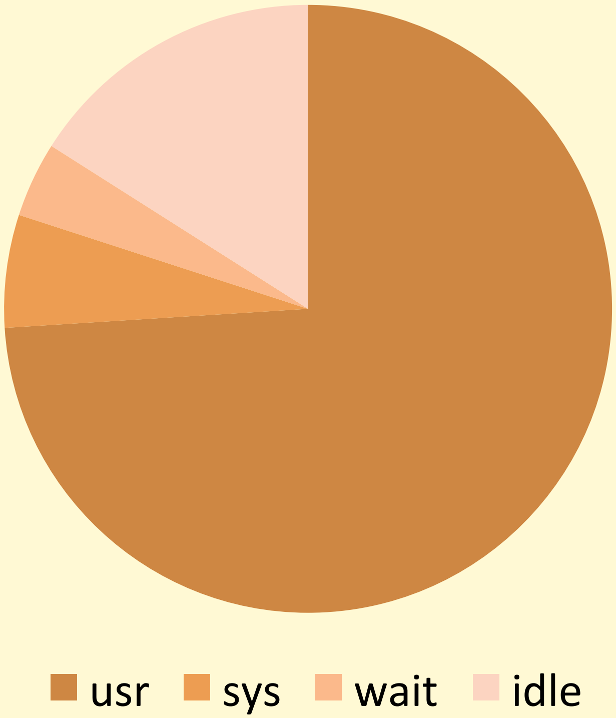
Better suited for *objective* metrics

Tachometers



…especially with arbitrary color highlighting

Pie Charts



…for real-time metrics

What You Can Do

- Monitoring:
  - Verify metrics, test overhead (same as tools)
- Statistics:
  - Ask how is this calculated?
  - Study the full distribution
- Visualizations:
  - Use histograms, heat maps, flame graphs

BENCHMARKING

Benchmarks

Common Mistakes

Micro

Macro

Kitchen-Sink

bonnie++

Apache Bench

BENCHMARKS

~100% of Benchmarks are Wrong

- "Most popular benchmarks are flawed"
  - Traeger, A., E. Zadok, N. Joukov, and C. Wright. "**A Nine Year Study of File System and Storage Benchmarking**," ACM Transactions on Storage, 2008.
- All alternates can also be flawed

COMMON MISTAKES

Common Mistakes

1. Testing the wrong target
   – eg, FS cache instead of disk; misconfiguration
2. Choosing the wrong target
   – eg, disk instead of FS cache … doesn’t resemble real world
3. Invalid results
   – benchmark software bugs
4. Ignoring errors
   – error path may be fast!
5. Ignoring variance or perturbations
   – real workload isn't steady/consistent, which matters
6. Misleading results
   – Casual benchmarking: you benchmark A, but actually measure B, and conclude you measured C

MICRO BENCHMARKS

Micro Benchmarks

- Test a specific function in isolation. e.g.:
  - File system maximum cached read ops/sec
  - Network maximum throughput
- Examples of bad microbenchmarks:
  - gitpid() in a tight loop
  - speed of /dev/zero and /dev/null
- Common problems:
  - Testing a workload that is not very relevant
  - Missing other workloads that are relevant

MACRO BENCHMARKS

Macro Benchmarks

- Simulate application user load. e.g.:
  - Simulated web client transaction
- Common problems:
  - Misplaced trust: believed to be realistic, but misses variance, errors, perturbations, etc.
  - Complex to debug, verify, and root cause

KITCHEN SINK BENCHMARKS

Kitchen Sink Benchmarks

- Run everything!
  - Mostly random benchmarks found on the Internet, where most are are broken or irrelevant
  - Developers focus on collecting more benchmarks than verifying or fixing the existing ones
- Myth that more benchmarks == greater accuracy
  - No, use active benchmarking (analysis)

BONNIE++

bonnie++

- "simple tests of hard drive and file system performance"
- First metric printed: **per character sequential output**
- What I found it actually tested:
  - 1 byte writes to libc (via putc())
  - 4 Kbyte writes from libc -> FS (depends on OS; see setbuffer())
  - 128 Kbyte async writes to disk (depends on storage stack)
  - Any file system throttles that may be present (eg, ionice)
  - C++ code, to some extent (bonnie++ 10% slower than Bonnie)
- Actual limiter:
  - Single threaded write_block_putc() and putc() calls
- Now thankfully fixed

APACHE BENCH

Apache Bench

- HTTP web server benchmark
- Single thread limited (use wrk for multi-threaded)
- Keep-alive option (-k):
  - without: Can become an unrealistic TCP session benchmark
  - with: Can become an unrealistic server throughput test
- Performance issues of ab's own code

UNIXBENCH

UnixBench

- The original kitchen-sink micro benchmark from 1984, published in BYTE magazine
- Results summarized as "The BYTE Index". Including:

```
 system:
    dhry2reg         Dhrystone 2 using register variables
    whetstone-double Double-Precision Whetstone
    syscall          System Call Overhead
    pipe             Pipe Throughput
    context1         Pipe-based Context Switching
    spawn            Process Creation
    execl            Execl Throughput
    fstime-w         File Write 1024 bufsize 2000 maxblocks
    fstime-r         File Read 1024 bufsize 2000 maxblocks
    fstime           File Copy 1024 bufsize 2000 maxblocks
    fsbuffer-w       File Write 256 bufsize 500 maxblocks
    fsbuffer-r       File Read 256 bufsize 500 maxblocks
    fsbuffer         File Copy 256 bufsize 500 maxblocks
    fsdisk-w         File Write 4096 bufsize 8000 maxblocks
[…]
```

- Many problems, starting with…

UnixBench Makefile

- Default (by ./Run) for **Linux**. Would you edit it? Then what?
- I "fixed" it and "improved" Dhrystone 2 performance by 64%

```
## Very generic
#OPTON = -O

## For Linux 486/Pentium, GCC 2.7.x and 2.8.x
#OPTON = -O2 -fomit-frame-pointer -fforce-addr -fforce-mem -ffast-math \
#  -m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2

## For Linux, GCC previous to 2.7.0
#OPTON = -O2 -fomit-frame-pointer -fforce-addr -fforce-mem -ffast-math -m486

#OPTON = -O2 -fomit-frame-pointer -fforce-addr -fforce-mem -ffast-math \
#  -m386 -malign-loops=1 -malign-jumps=1 -malign-functions=1

## For Solaris 2, or general-purpose GCC 2.7.x
OPTON = -O2 -fomit-frame-pointer -fforce-addr -ffast-math -Wall

## For Digital Unix v4.x, with DEC cc v5.x
#OPTON = -O4
#CFLAGS = -DTIME -std1 -verbose -w0
```

UnixBench Documentation

"The results will depend not only on your hardware, but on your **operating system, libraries, and even compiler.**"

"So you may want to make sure that all your test systems are running the same version of the OS; or **at least publish the OS and compuiler versions with your results.**"

… UnixBench was innovative & useful, but it's time has passed

What You Can Do

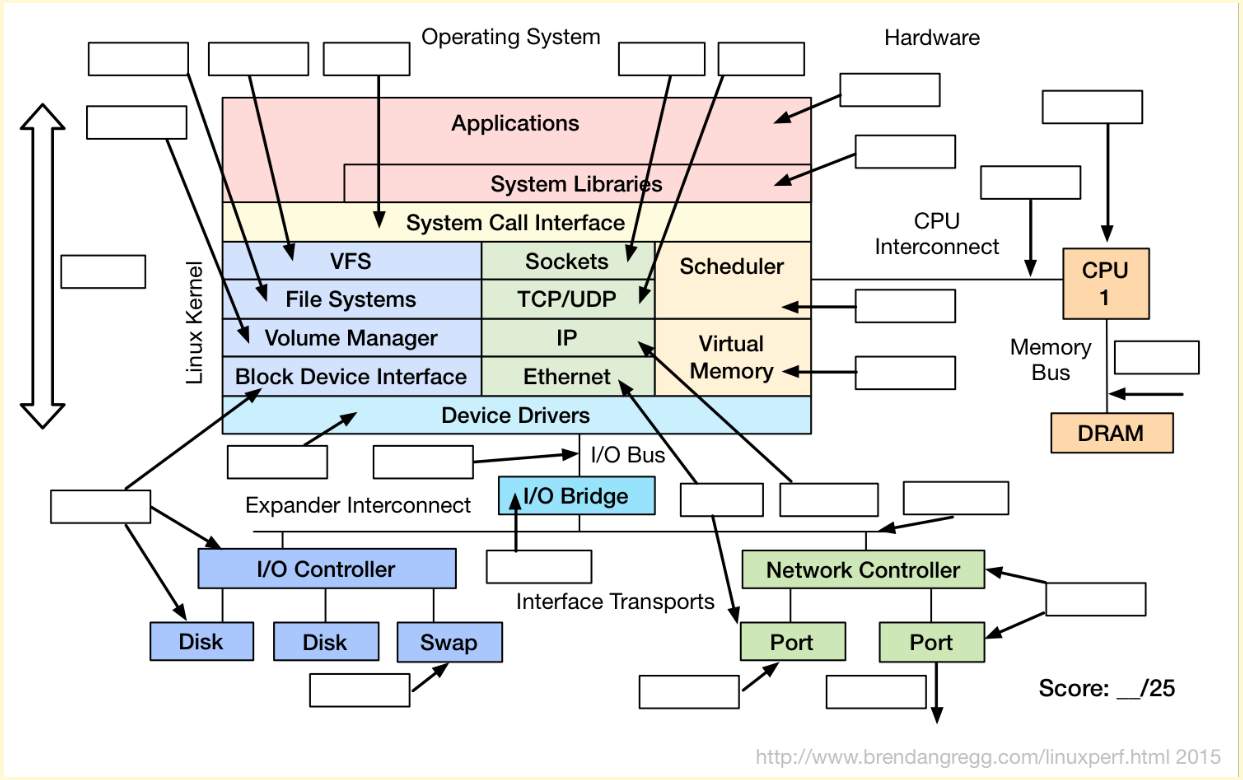
- Match the benchmark to your workload
- Active Benchmarking
  1. Configure the benchmark to run in steady state, 24x7
  2. Do root-cause analysis of benchmark performance
  3. Answer: why X and not 10X? Limiting factor?

  It can take 1-2 weeks to debug a single benchmark

Summary

Observe Everything

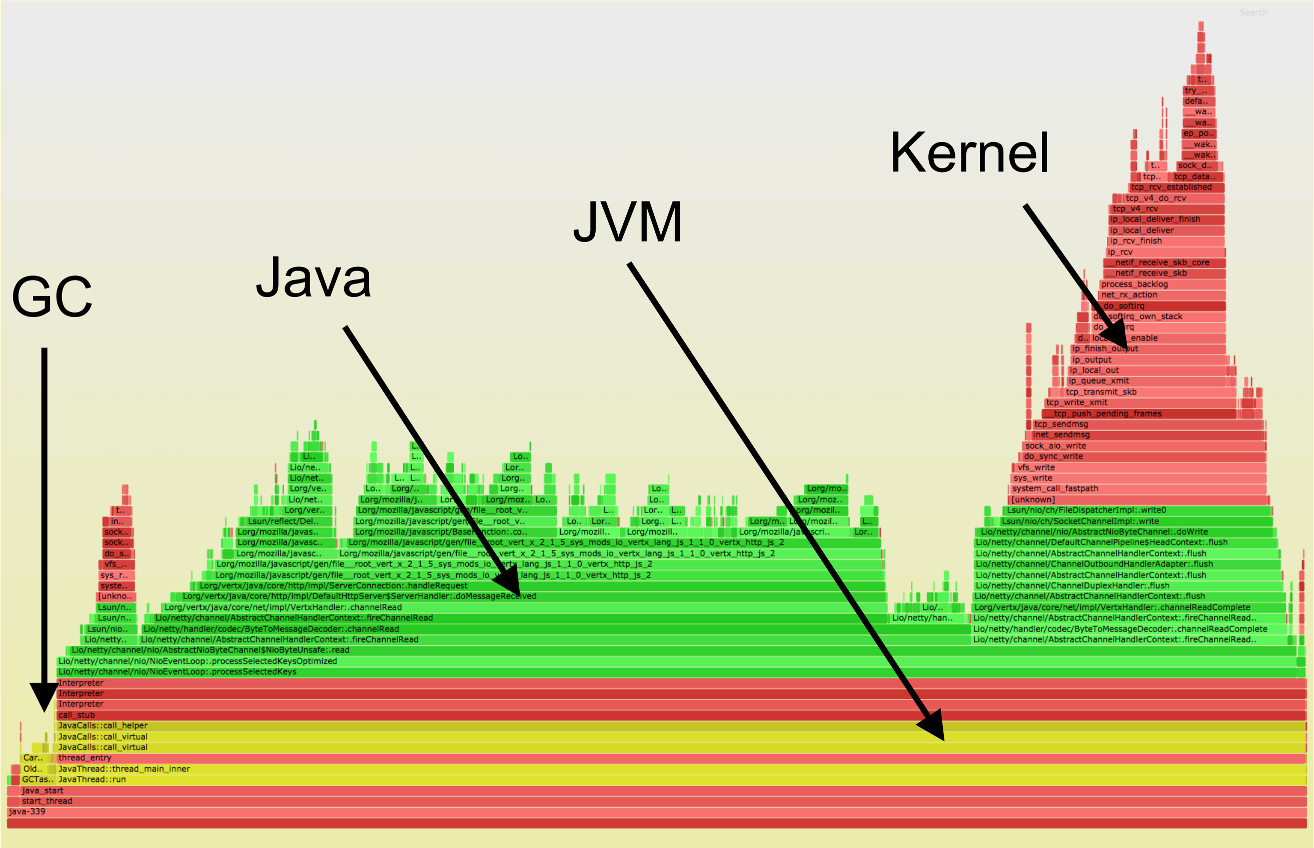
- Trust nothing. Verify. Write small tests.
- Pose Q's first then find the metrics. e.g., functional diagrams:



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

Profile Everything

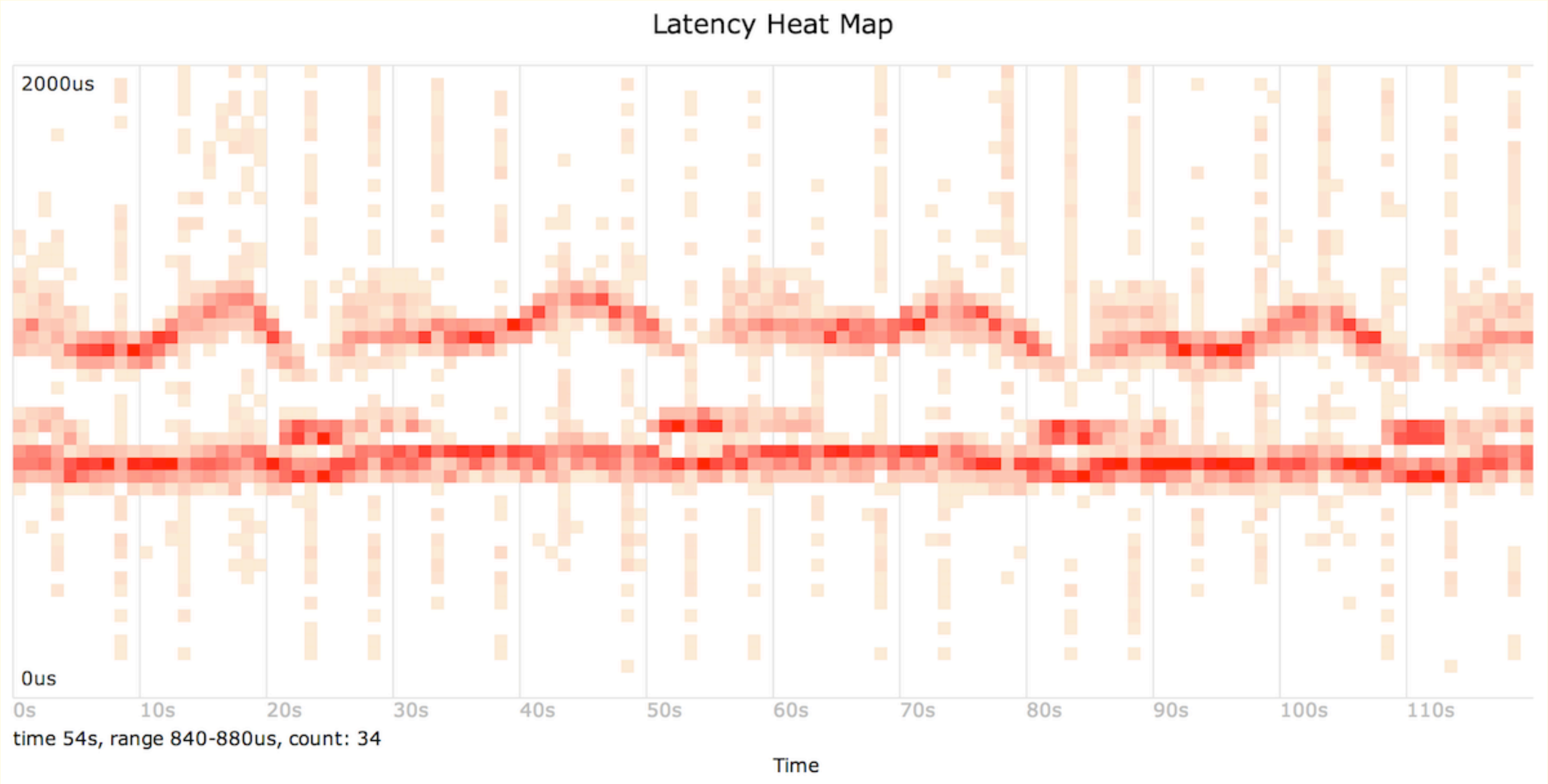
- e.g., Java Mixed-Mode Flame Graphs:



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

Visualize Everything

- Full distributions of latency. e.g., heat maps:



Reference: http://queue.acm.org/detail.cfm?id=1809426

Benchmark Nothing!

(if you must, use Active Benchmarking)

Links & References

- **Things that aren't broken**:
  - http://www.brendangregg.com/linuxperf.html
- References:
  - https://upload.wikimedia.org/wikipedia/commons/6/64/Intel_Nehalem_arch.svg
  - http://www.linuxatemyram.com/
  - Traeger, A., E. Zadok, N. Joukov, and C. Wright. “A Nine Year Study of File System and Storage Benchmarking,” ACM Trans- actions on Storage, 2008.
  - http://www.brendangregg.com/blog/2014-06-09/java-cpu-sampling-using-hprof.html
  - http://www.brendangregg.com/activebenchmarking.html
  - https://blogs.oracle.com/roch/entry/decoding_bonnie
  - http://www.brendangregg.com/blog/2014-05-02/compilers-love-messing-with-benchmarks.html
  - https://code.google.com/p/byte-unixbench/
  - https://qconsf.com/sf2015/presentation/how-not-measure-latency
  - https://qconsf.com/system/files/presentation-slides/profilers_are_lying_hobbitses.pdf
  - Caution signs drawn be me, inspired by real-world signs

Thanks

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