QCon SAN FRANCISCO Nov 2015

NETFLIX

Broken Performance Tools

Brendan Gregg

Senior Performance Architect, Netflix

CAUTION: PERFORMANCE TOOLS

NETFLIX

- Over 60 million subscribers
- AWS EC2 Linux cloud
- FreeBSD CDN
- Awesome place to work

This Talk

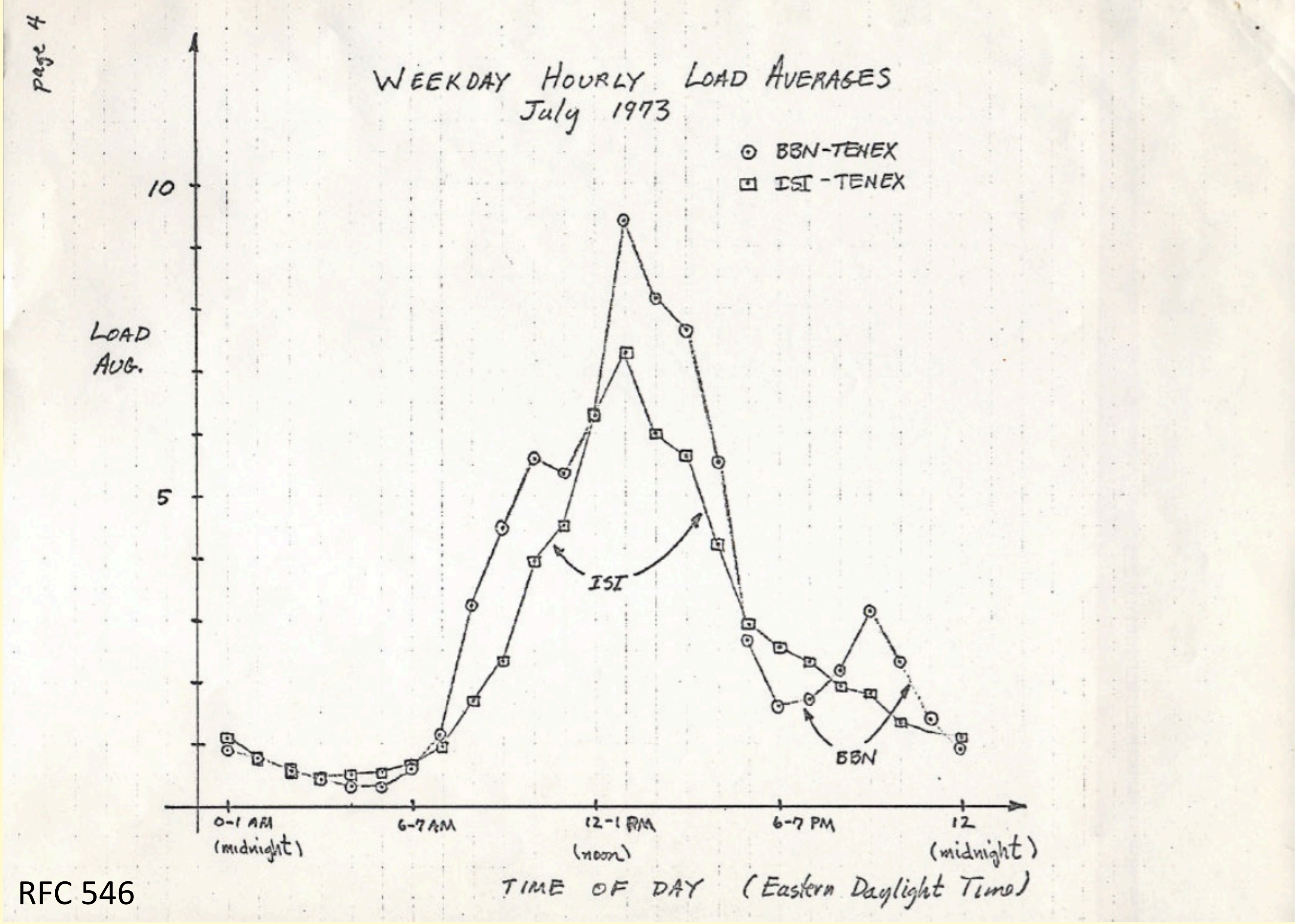
- Observability, benchmarking, anti-patterns, and lessons
- Broken and misleading things that are surprising

Note: problems with current implementations are discussed, which may be fixed/improved in the future

Observability:

System Metrics

LOAD AVERAGES

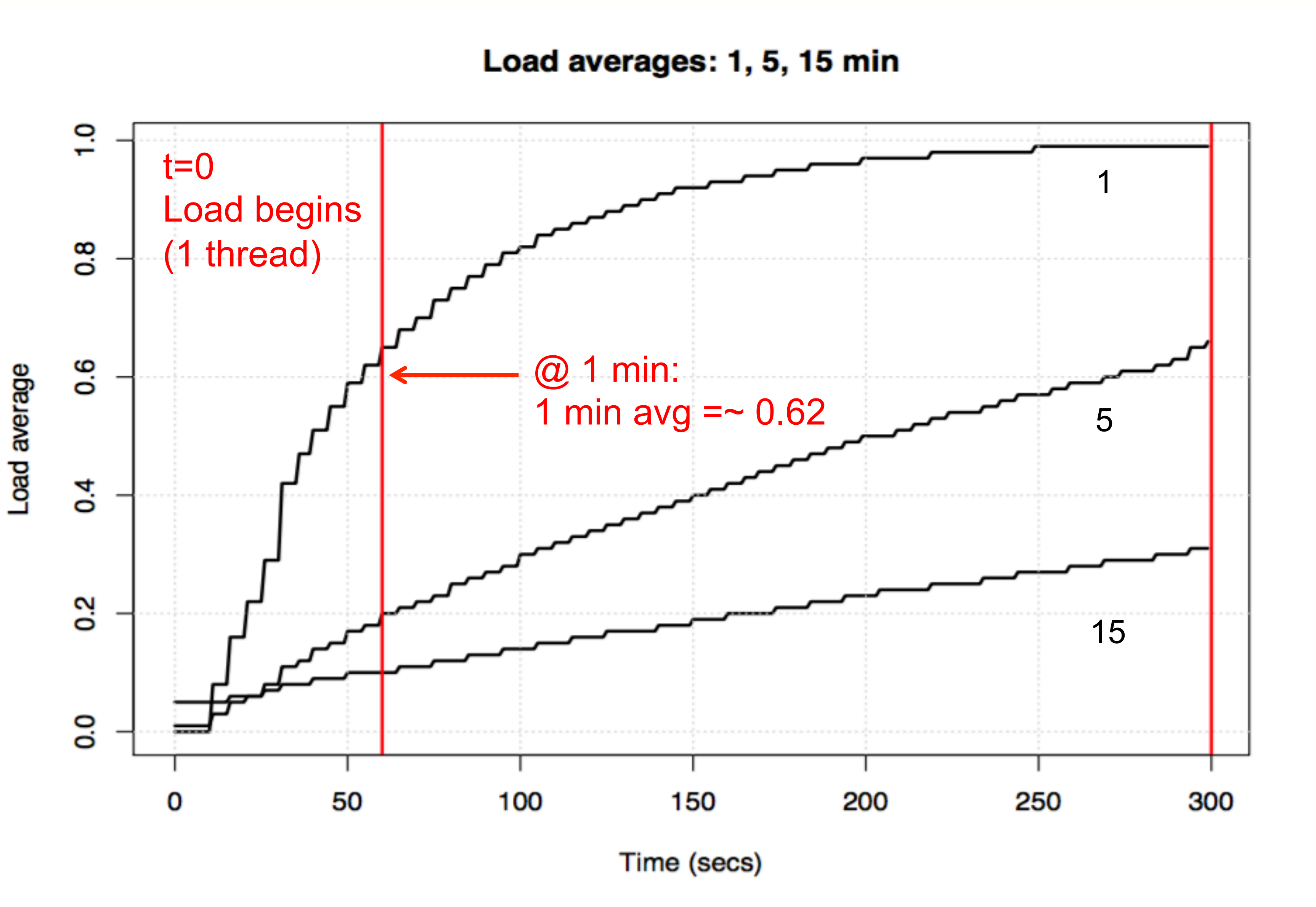

page 4
WEEKDAY HOURLY LOAD AVERAGES
July 1973
BBN-TENEX
ISI-TENEX
10
LOAD AVG.
5
ISI
BBN
0-1 AM (midnight)
6-7AM
12-1 PM (noon)
6-7 PM
12 (midnight)
TIME OF DAY (Eastern Daylight Time)

RFC 546

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 (scheduler run queue length/latency)
  - On Linux, task demand: CPU + uninterruptible disk I/O (?)
- "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
0.0
0.2
0.4
0.6
0.8
1.0
0
50
100
150
200
250
300
Time (secs)

Load Average

"1 minute load average"

really means…

"*The exponentially damped moving sum of CPU + uninterruptible disk I/O that uses a value of 60 seconds in its equation*"

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)
- Stop clearing the screen!
  - No option to turn this off. Your eyes can miss updates.
  - I often use pidstat(1) on Linux instead. Scroll back for history.

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?

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

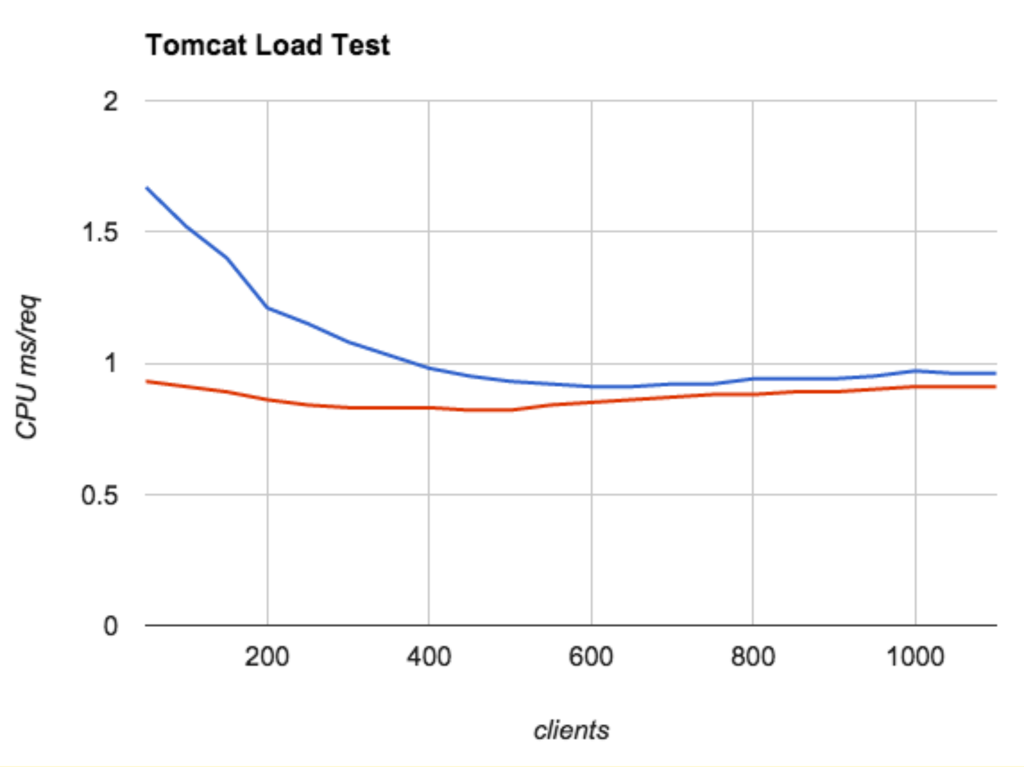
%CPU

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…

A CPU Mystery…

- As load increased, CPU ms per request lowered (blue)
  - up to 1.84x faster
- Was it due to:
  - Cache warmth? no
  - Different code? no
  - Turbo boost? no
- (Same test, but problem fixed, is shown in red)

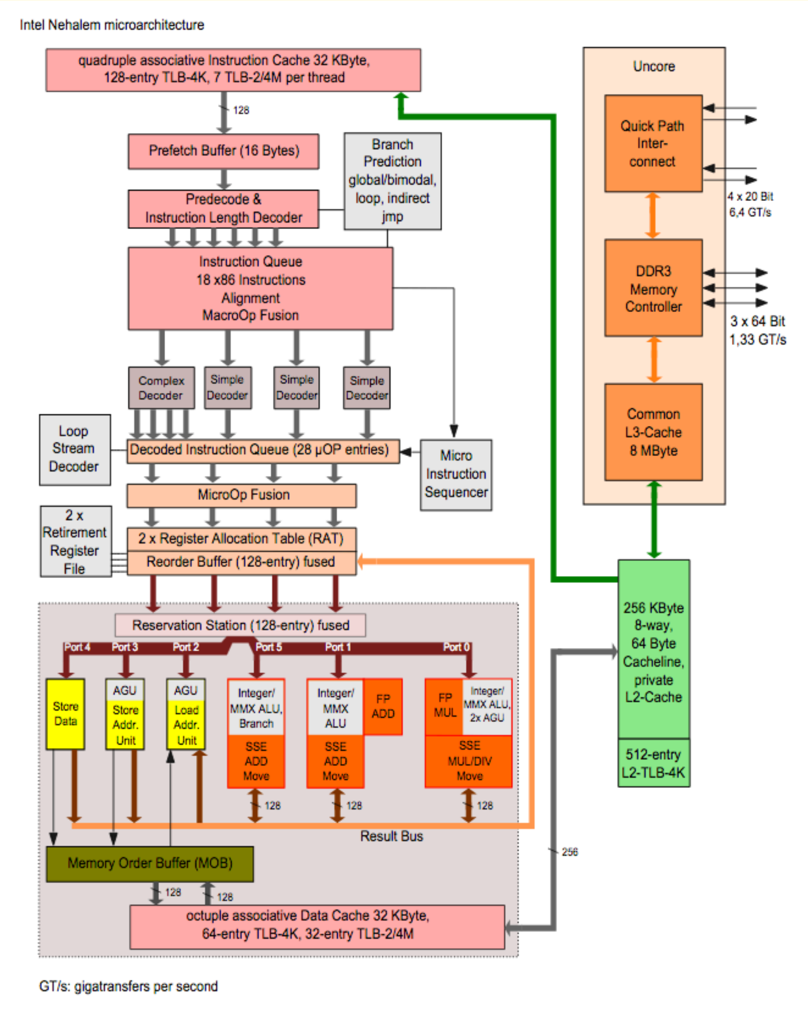


CPU Speed Variation

- **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. Need to know the clock speed as well
- 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
- Nowadays it's a lot of work to truly understand what CPUs are doing



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

I/O WAIT

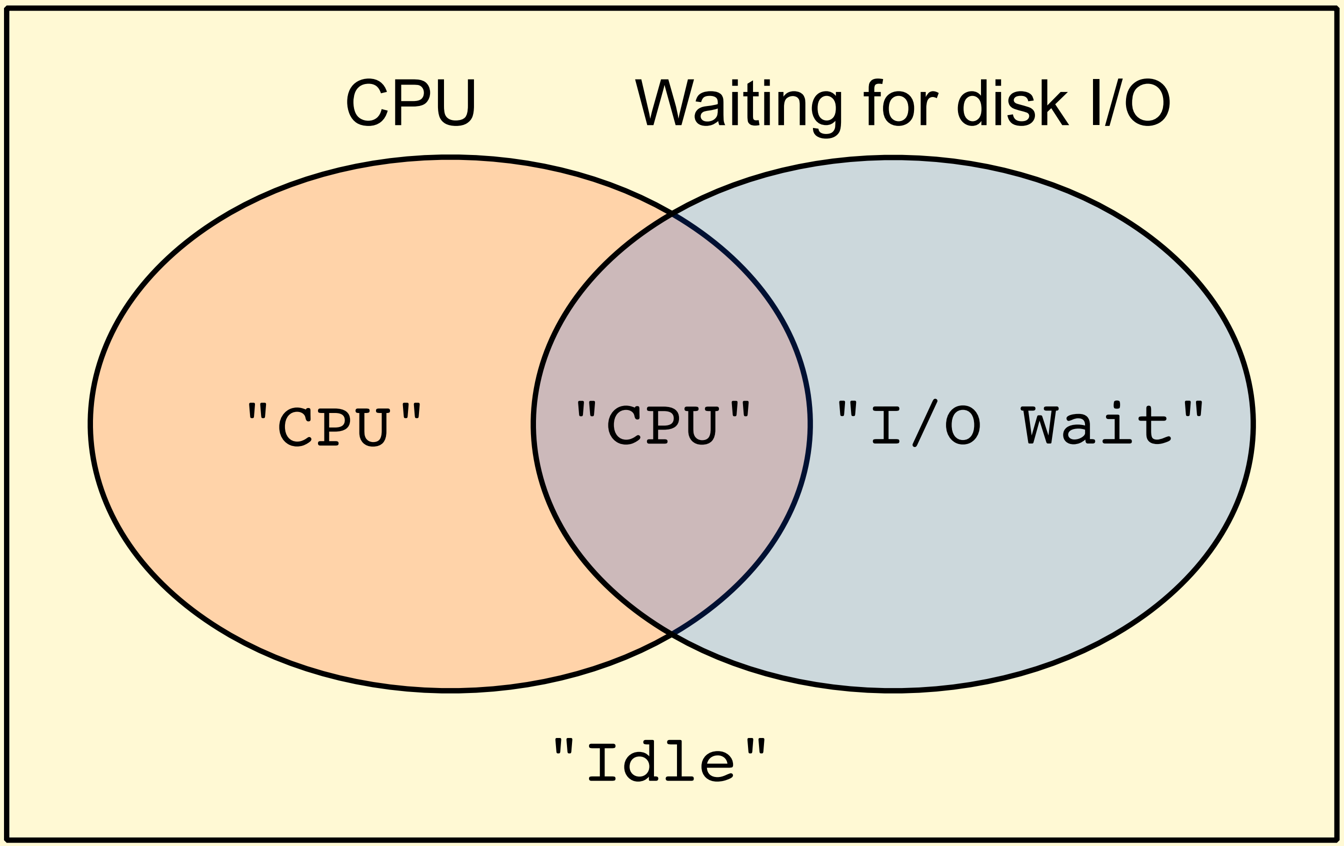
I/O Wait

```
$ 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
- Solaris implementation was also broken and later hardwired to zero
- Can be very useful when understood: another idle state

I/O Wait Venn Diagram

Per CPU:

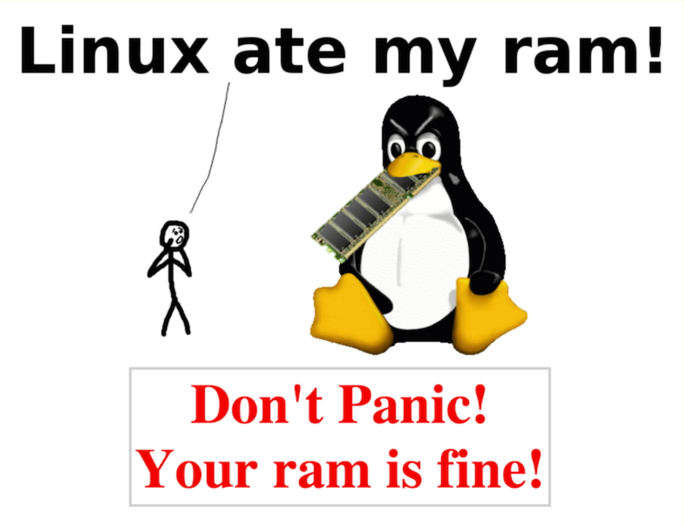


FREE MEMORY

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!
- Other implementations:
  - "r" may be sampled once per second. Almost useless.
  - Columns like "de" for deficit, making much less sense for non-page scanned situations

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

```
[…]
Tcp:
    17337 active connections openings
    395515 passive connection openings
    8953 failed connection attempts
    240214 connection resets received
    3 connections established
    7198870 segments received
    7505329 segments send out
    62697 segments retransmited
    10 bad segments received.
    1072 resets sent
    InCsumErrors: 5
[…]
```

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) can process requests 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 latency asynchronous
  – Better measuring latency via application->FS calls

Rules for Metrics Makers

- They must work
  - As well as possible. Clearly document caveats.
- They must be useful
  - Document a real use case (eg, my example.txt files). If you get stuck, it's not useful – ditch it.
- Aim to be intuitive
  - Document it. If it's too weird to explain, redo it.
- As few as possible
  - Respect end-user's time
- Good system examples:
  - iostat -x: workload columns, then resulting perf columns
  - Linux sar: consistency, units on columns, logical groups

Observability:

Profilers

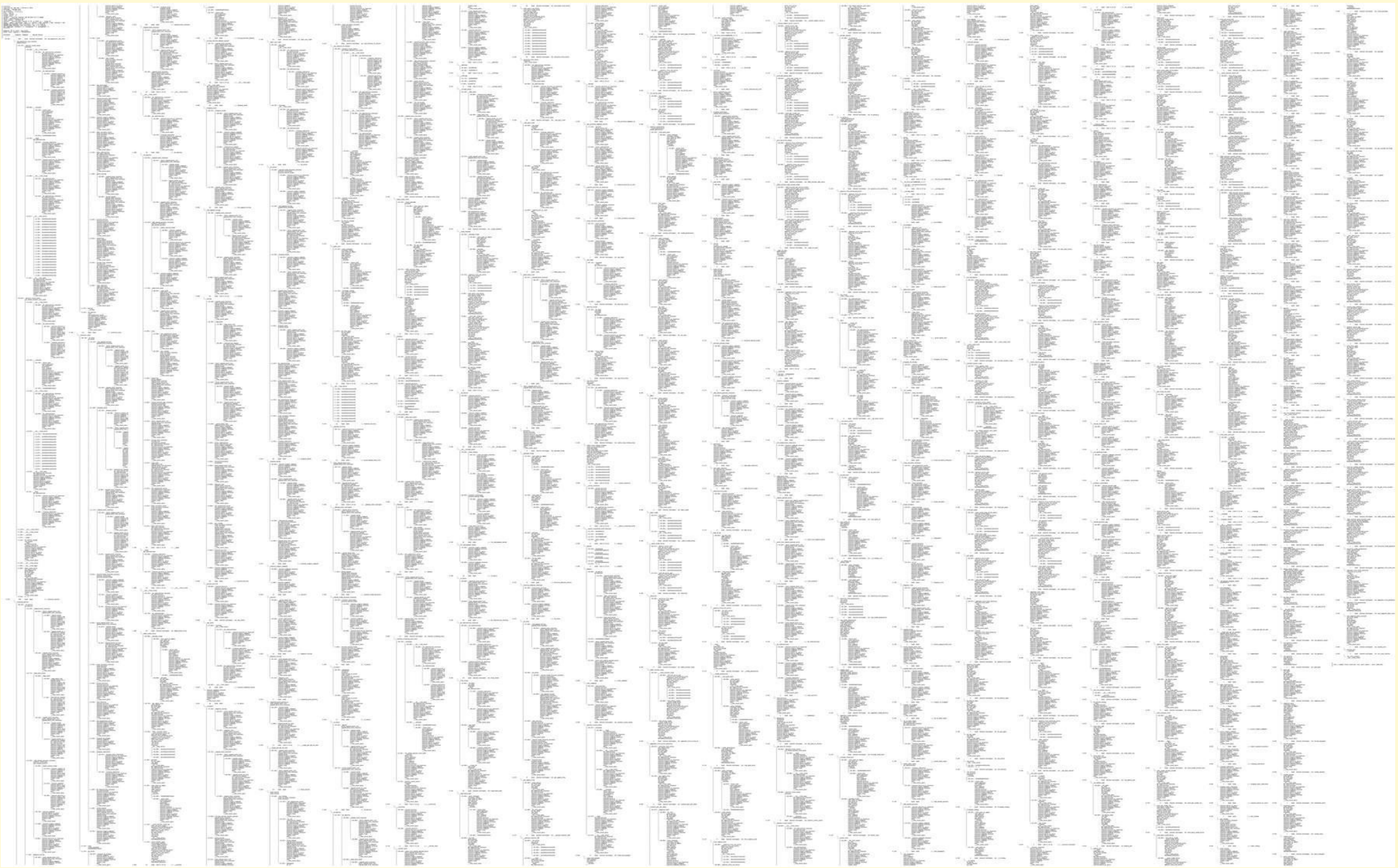
PROFILERS

Linux `perf`

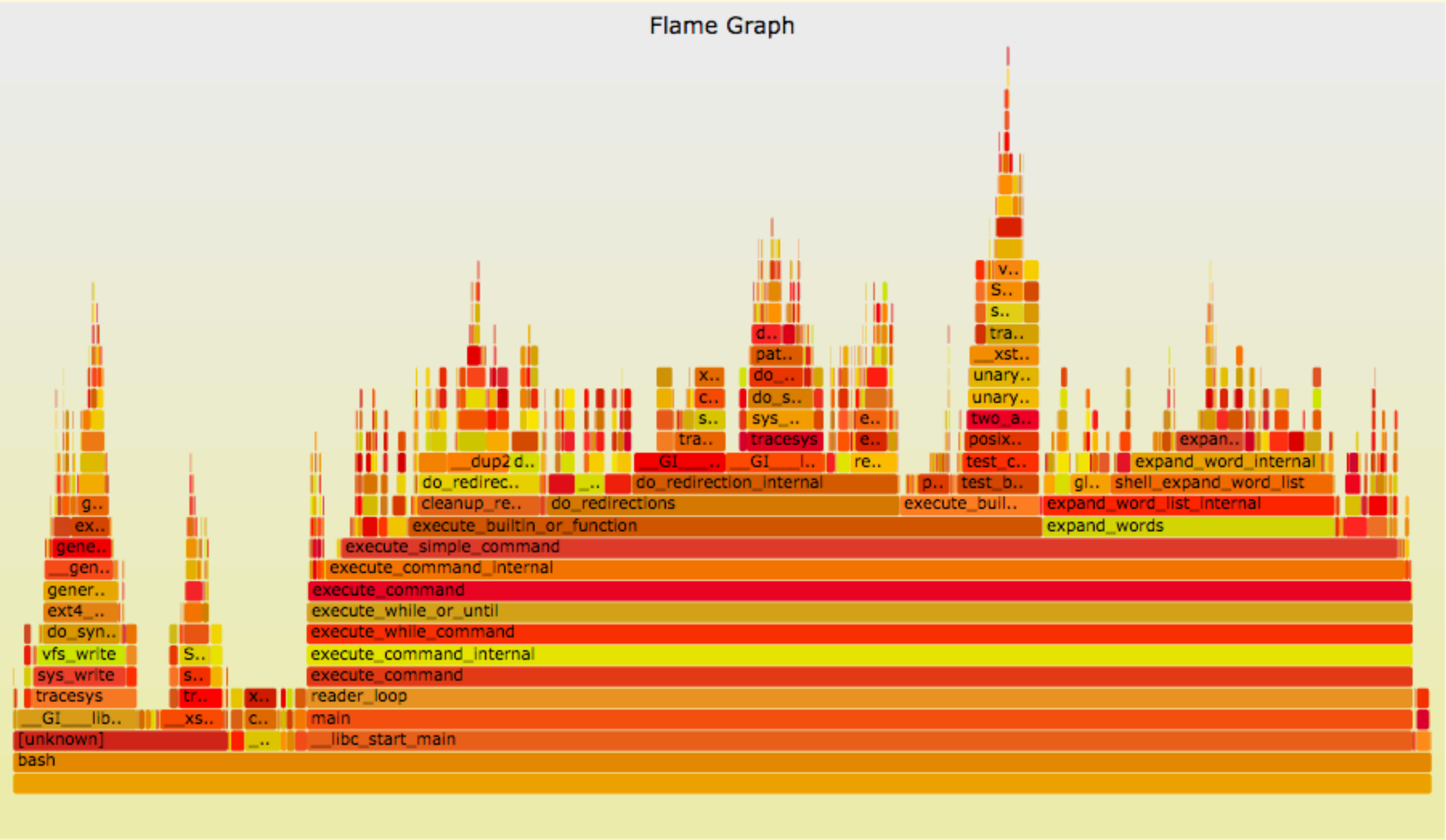
- Can sample stack traces and summarize output:

```
# 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

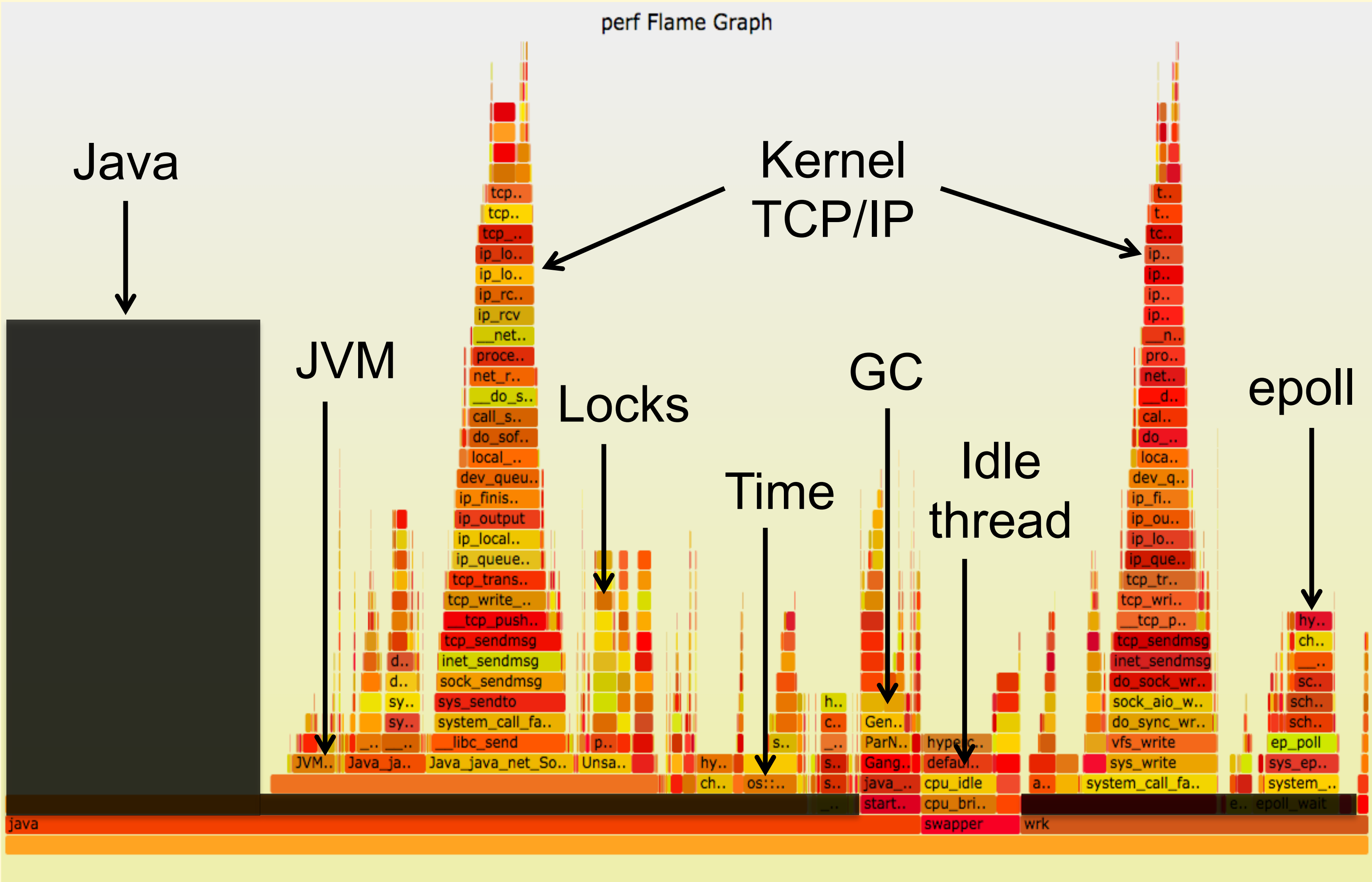


… as a Flame Graph



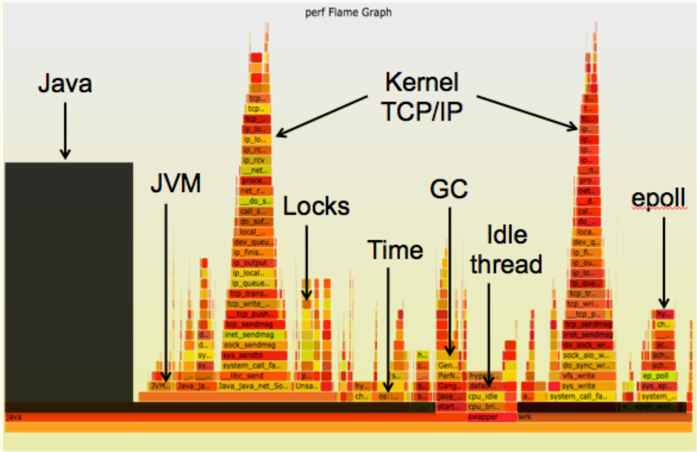
PROFILER VISIBILITY

System Profilers with Java



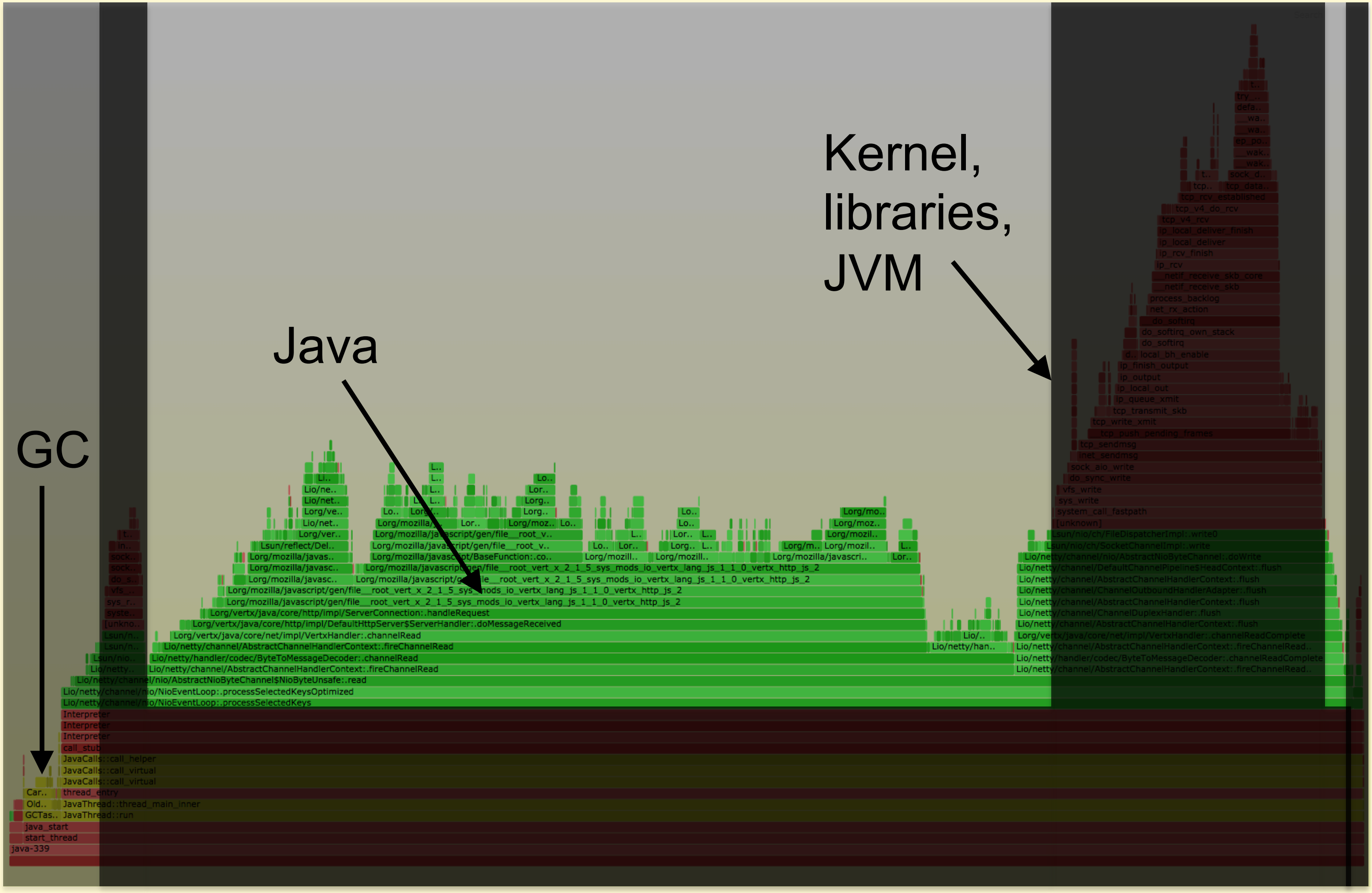
System Profilers with Java

- e.g., Linux perf
- Visibility
  - JVM (C++)
  - GC (C++)
  - libraries (C)
  - kernel (C)



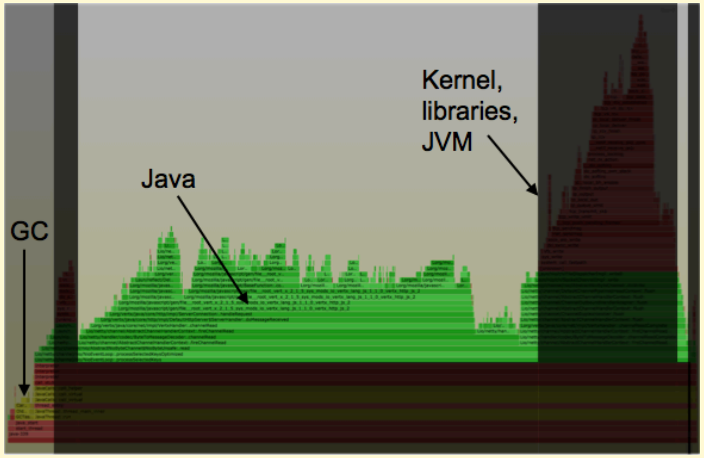
- Typical problems (x86):
  - Stacks missing for Java and other runtimes
  - Symbols missing for Java methods
- Profile everything **except Java** and similar runtimes

Java Profilers



Java Profilers

- Visibility
  - Java method execution
  - Object usage
  - GC logs
  - Custom Java context
- Typical problems:
  - Sampling often happens at safety/yield points (skew)
  - Method tracing has massive observer effect
  - Misidentifies RUNNING as on-CPU (e.g., epoll)
  - Doesn't include or profile GC or JVM CPU time
  - Tree views not quick (proportional) to comprehend
- **Inaccurate** (skewed) and **incomplete** profiles



COMPILER OPTIMIZATIONS

Broken System Stack Traces

- Profiling Java on x86 using perf
- The stacks are 1 or 2 levels deep, and have 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)

java  4579 cpu-clock:
      7f41792fc65f [unknown] (/tmp/perf-4458.map)
  a2d53351ff7da603 [unknown] ([unknown])

java  4579 cpu-clock:
      7f4179349aec [unknown] (/tmp/perf-4458.map)

java  4579 cpu-clock:
      7f4179101d0f [unknown] (/tmp/perf-4458.map)
[…]
```

Why Stacks are Broken

- On x86 (x86_64), hotspot uses the frame pointer register (RBP) as general purpose
- This "**compiler optimization**" breaks stack walking
- *Once upon a time*, x86 had fewer registers, and this made much more sense
- gcc provides **`-fno-omit-frame-pointer`** to avoid doing this
  - JDK8u60+ now has this as -XX:+PreserveFramePoiner

Missing Symbols

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

```
71.79%            334     sed   sed                [.] 0x000000000001afc1
                   |
                   |--11.65%-- 0x40a447
                   |          0x40659a
                   |          0x408dd8
                   |          0x408ed1
                   |          0x402689
                   |          0x7fa1cd08aec5
```

broken

```
12.06%             62      sed  sed                [.] re_search_internal
                    |
                    --- re_search_internal
                       |
                       |--96.78%-- re_search_stub
                       |          rpl_re_search
                       |          match_regex
                       |          do_subst
                       |          execute_program
                       |          process_files
                       |          main
                       |          __libc_start_main
```

not broken

Fixing Symbols

- For applications, install debug symbol package
- For JIT'd code, Linux perf already looks for an externally provided symbol file: /tmp/perf-PID.map

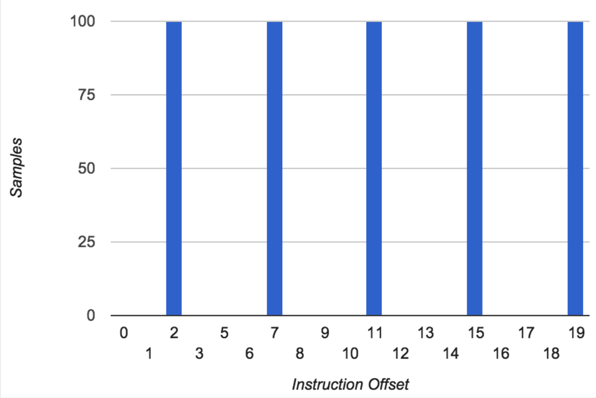
```
# perf script
Failed to open /tmp/perf-8131.map, continuing without symbols
[…]
java 8131 cpu-clock:
    7fff76f2dce1 [unknown] ([vdso])
    7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm…
    7fd301861e46 [unknown] (/tmp/perf-8131.map)
[…]
```

- Find for a way to create 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

Observability:

Overhead

TCPDUMP

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
  - Use buffered tracing / in-kernel counters instead, e.g. DTrace

DTRACE

DTrace

- Overhead often negligible, but not always
- Before:

```
# time wc systemlog
  262600  2995200 23925200 systemlog
real    0m1.098s
user    0m1.085s
sys     0m0.012s
```

- After:

```
# time dtrace -n 'pid$target:::entry { @[probefunc] = count(); }' -c 'wc systemlog'
dtrace: description 'pid$target:::entry ' matched 3756 probes
  262600  2995200 23925200 systemlog
[…]
real    7m2.896s
user    7m2.650s
sys     0m0.572s
```

- 384x slower. Fairly worst case: frequent pid probes.

Tracing Dangers

- Overhead potential exists for **all tracers**
  - Overhead = event instrumentation cost X frequency of event
- Costs
  - Lower: counters, in-kernel aggregations
  - Higher: event dumps, stack traces, string copies, copyin/outs
- Frequencies
  - Lower: process creation & destruction, disk I/O (usually), …
  - Higher: instructions, functions in I/O hot path, malloc/free, Java methods, …
- Advice
  - < 10,000 events/sec, probably ok
  - > 100,000 events/sec, overhead may start to be measurable

DTraceToolkit

- My own tools that can cause massive overhead:
  - dapptrace/dappprof: can trace all native functions
  - Java/j_flow.d, ...: can trace all Java methods with +ExtendedDTraceProbes

```
# j_flow.d
  C    PID TIME(us)         -- CLASS.METHOD
  0 311403 4789112583163    -> java/lang/Object.<clinit>
  0 311403 4789112583207      -> java/lang/Object.registerNatives
  0 311403 4789112583323      <- java/lang/Object.registerNatives
  0 311403 4789112583333    <- java/lang/Object.<clinit>
  0 311403 4789112583343    -> java/lang/String.<clinit>
  0 311403 4789112583732      -> java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583743        -> java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583752          -> java/lang/Object.<init>
[...]
```

- Useful for debugging, but should warn about overheads

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**!
- Issues & advice already covered at QCon:
  - Nitsan Wakart "Profilers are Lying Hobbitses" earlier today
  - Java track tomorrow

Observability:

Monitoring

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!

Observability:

Statistics

STATISTICS

Statistics

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

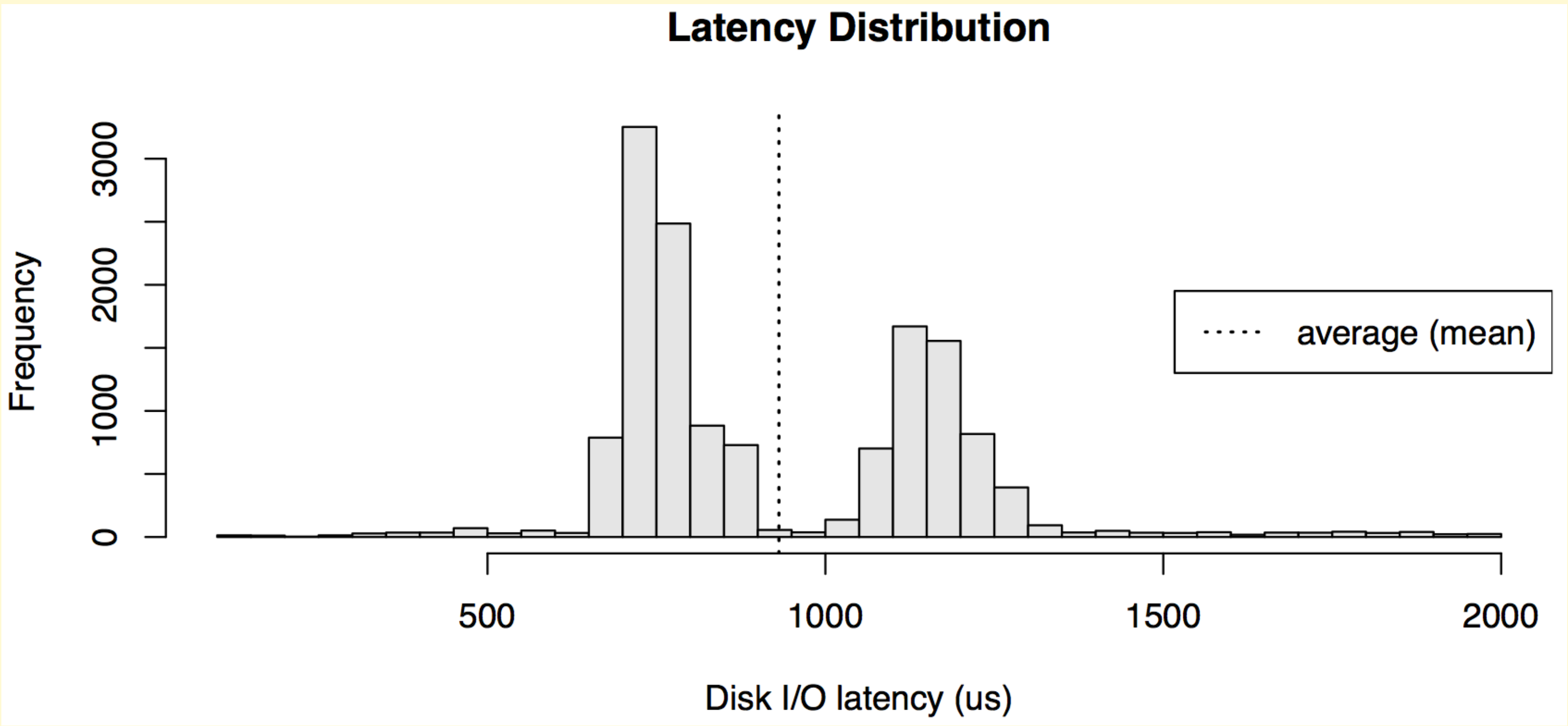
– W.I.E. Gates

Statistics

- Averages can be misleading
  - Hide latency outliers
  - Per-minute averages can hide multi-second issues
- Percentiles can be misleading
  - Probability of hitting 99.9^{th} 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
- See Gil Tene's "How Not to Measure Latency" QCon talk from earlier today

Average Latency

- When the index of central tendency isn't…

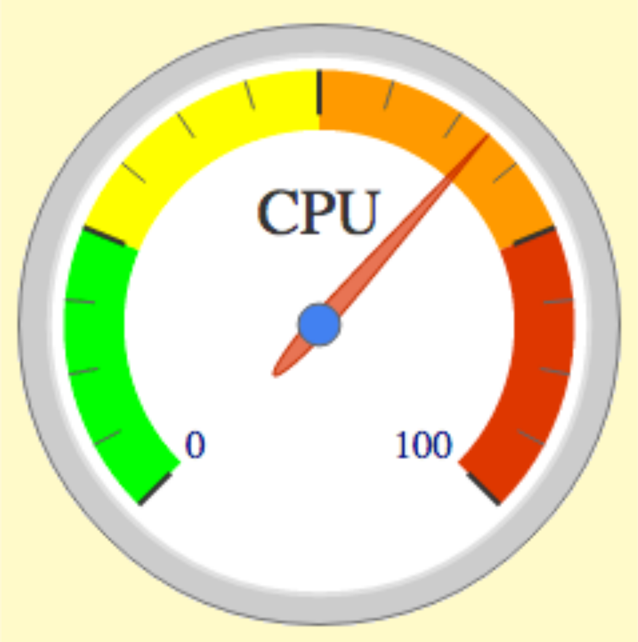


Observability:

Visualizations

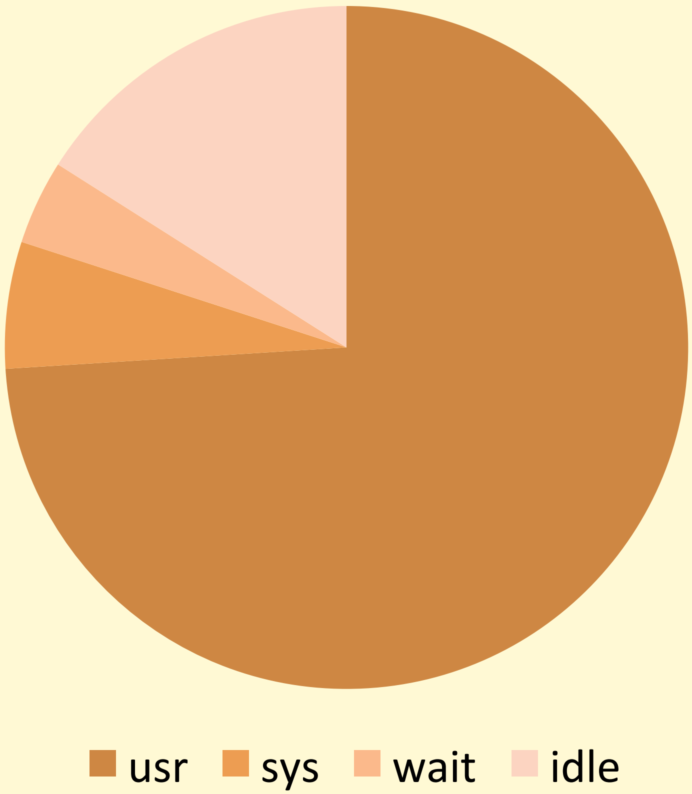
VISUALIZATIONS

Tachometers



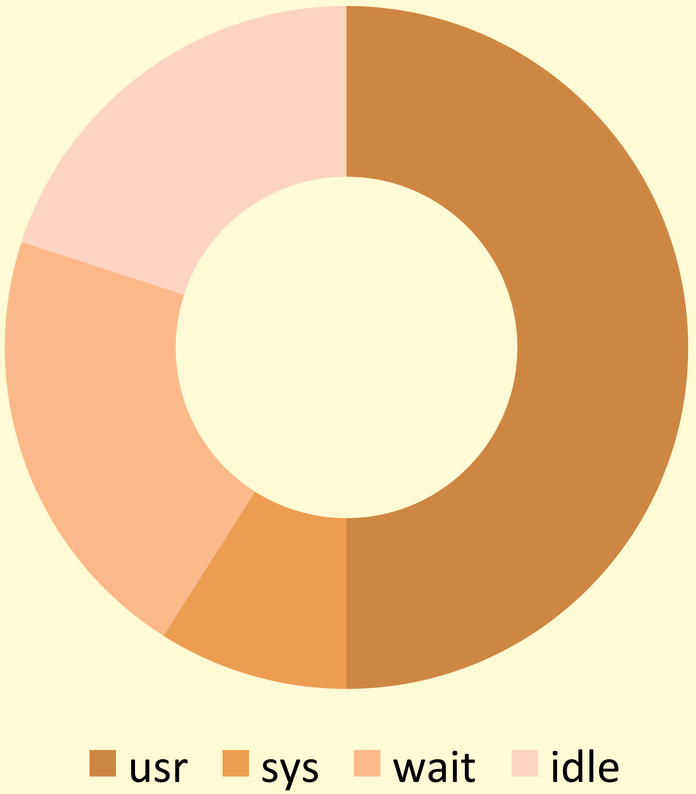
…especially with arbitrary color highlighting

Pie Charts



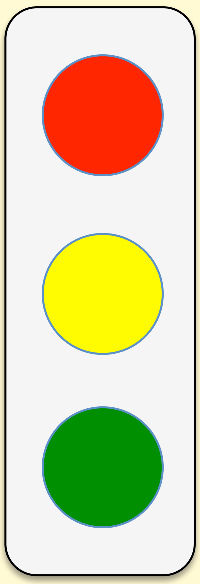
…for real-time metrics

Doughnuts



…like pie charts but worse

Traffic Lights



RED == BAD (usually)

GREEN == GOOD (hopefully)

…when used for *subjective* metrics

These can be used for *objective* metrics

Benchmarking

BENCHMARKING

~100% of benchmarks are wrong

"Most popular benchmarks are flawed"

Source: Traeger, A., E. Zadok, N. Joukov, and C. Wright. "A Nine Year Study of File System and Storage Benchmarking," ACM Transactions on Storage, 2008.

Not only can a popular benchmark be broken, but so can all alternates.

REFUTING BENCHMARKS

The energy needed
to refute benchmarks
is multiple orders of magnitude
bigger than to run them

It can take 1-2 weeks of senior performance engineering time to debug a single benchmark.

Benchmarking

- Benchmarking is a useful form of experimental analysis
  - Try observational first; benchmarks can perturb
- Accurate and realistic benchmarking is vital for technical investments that improve our industry
- However, benchmarking is **error prone**

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
   – you benchmark A, but actually measure B, and conclude you measured C

PRODUCT EVALUATIONS

Product Evaluations

- Benchmarking is used for product evaluations & sales
- The Benchmark Paradox:
  - **If your product's chances of winning a benchmark are 50/50, you'll usually lose**
  - To justify a product switch, a customer may run several benchmarks, and expect you to *win them all*
  - May mean winning a coin toss at least 3 times in a row
  - http://www.brendangregg.com/blog/2014-05-03/the-benchmark-paradox.html
- Solving this seeming paradox (and benchmarking):
  - Confirm benchmark is relevant to intended workload
  - Ask: why isn't it 10x?

Active Benchmarking

- **Root cause performance analysis** while the benchmark is still running
  - Use observability tools
  - Identify the limiter (or suspected limiter) and include it with the benchmark results
  - Answer: why not 10x?
- This takes time, but uncovers most mistakes

MICRO BENCHMARKS

Micro Benchmarks

- Test a specific function in isolation. e.g.:
  - File system maximum cached read operations/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, e.t.c.
  - 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)

AUTOMATION

Automated Benchmarks

- Completely automated procedure. e.g.:
  - Cloud benchmarks: spin up an instance, benchmark, destroy. Automate.
- Little or no provision for debugging
- Automation is only part of the solution

Benchmarking:

More Examples

BONNIE++

bonnie++

- "simple tests of hard drive and file system performance"
- First metric printed by (thankfully) older versions: **per character sequential output**
- What was 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, ZFS)
  - C++ code, to some extent (bonnie++ 10% slower than Bonnie)
- Actual limiter:
  - Single threaded write_block_putc() and putc() calls

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
- Innovative & useful for the time, but that time has passed
- More problems than I can shake a stick at
- Starting with…

COMPILERS

UnixBench Makefile

- Default (by ./Run) for **Linux**. Would you edit it? Then what?

```
## 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 Makefile

- "Fixing" the Makefile improved the first result, Dhrystone 2, by 64%
- Is everyone "fixing" it the same way, or not? Are they using the same compiler version? Same OS? (No.)

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.**"

SYSTEM MICROBENCHMARKS

UnixBench Tests

- Results summarized as "The BYTE Index". From USAGE:

```
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
   fsdisk-r         File Read 4096 bufsize 8000 maxblocks
   fsdisk           File Copy 4096 bufsize 8000 maxblocks
   shell1           Shell Scripts (1 concurrent) (runs "looper 60 multi.sh 1")
   shell8           Shell Scripts (8 concurrent) (runs "looper 60 multi.sh 8")
   shell16          Shell Scripts (8 concurrent) (runs "looper 60 multi.sh 16")
```

- What can go wrong? Everything.

Anti-Patterns

ANTI-PATTERNS

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

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

Performance Tools Team

- Having a separate performance tools team, who creates tools but doesn't use them (no production exposure)
- At Netflix:
  - The performance engineering team builds tools and uses tools for both service consulting and live production triage
    - Mogul, Vector, …
  - Other teams (CORE, traffic, …) also build performance tools and use them during issues
- Good performance tools are built out of necessity

Messy House Fallacy

- **Fallacy**: my code is a mess, I bet yours is immaculate, therefore the bug must be mine
- **Reality**: everyone's code is terrible and buggy
- When analyzing performance, don't overlook the system: kernel, libraries, etc.

Lessons

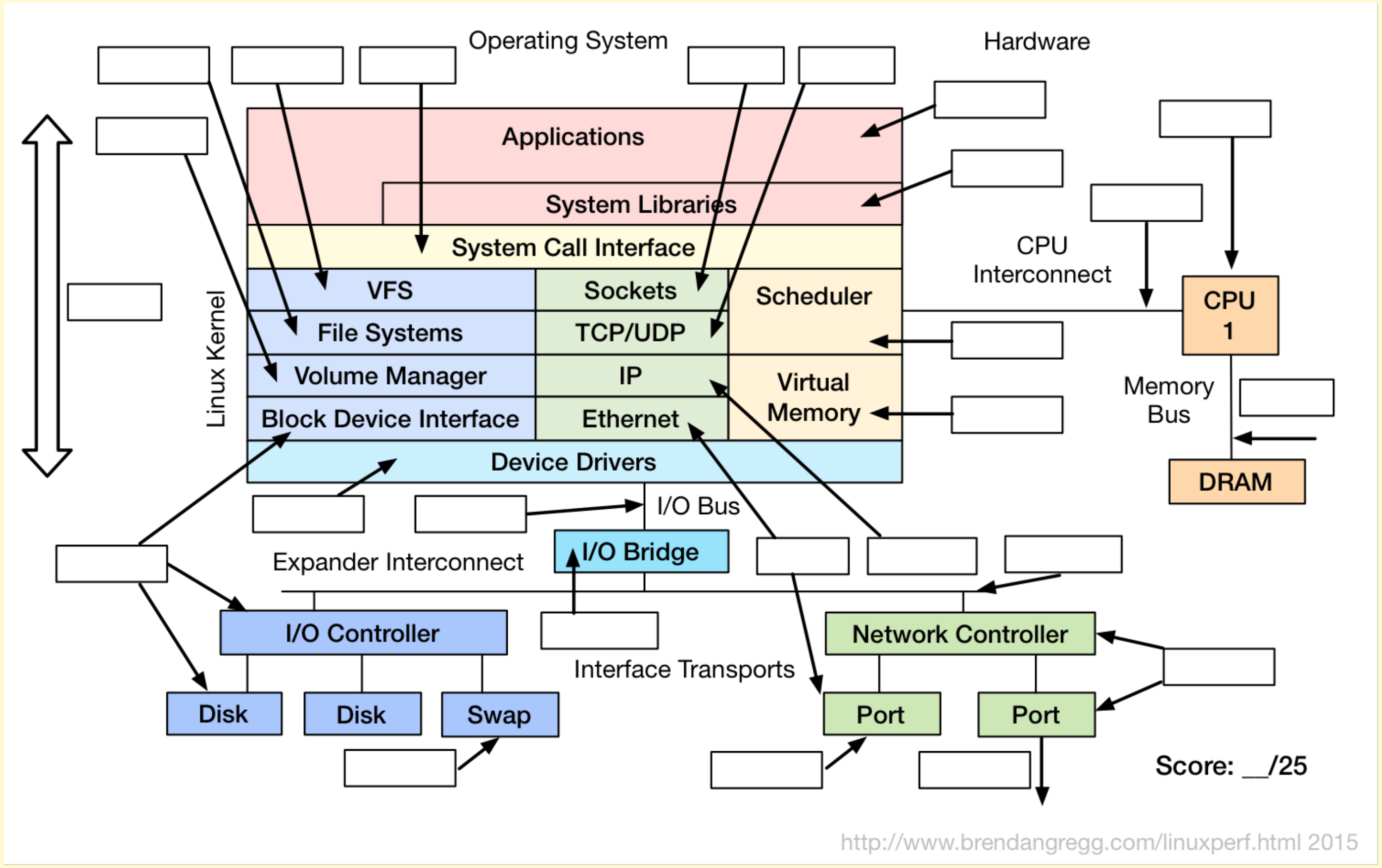
PERFORMANCE TOOLS

Observability

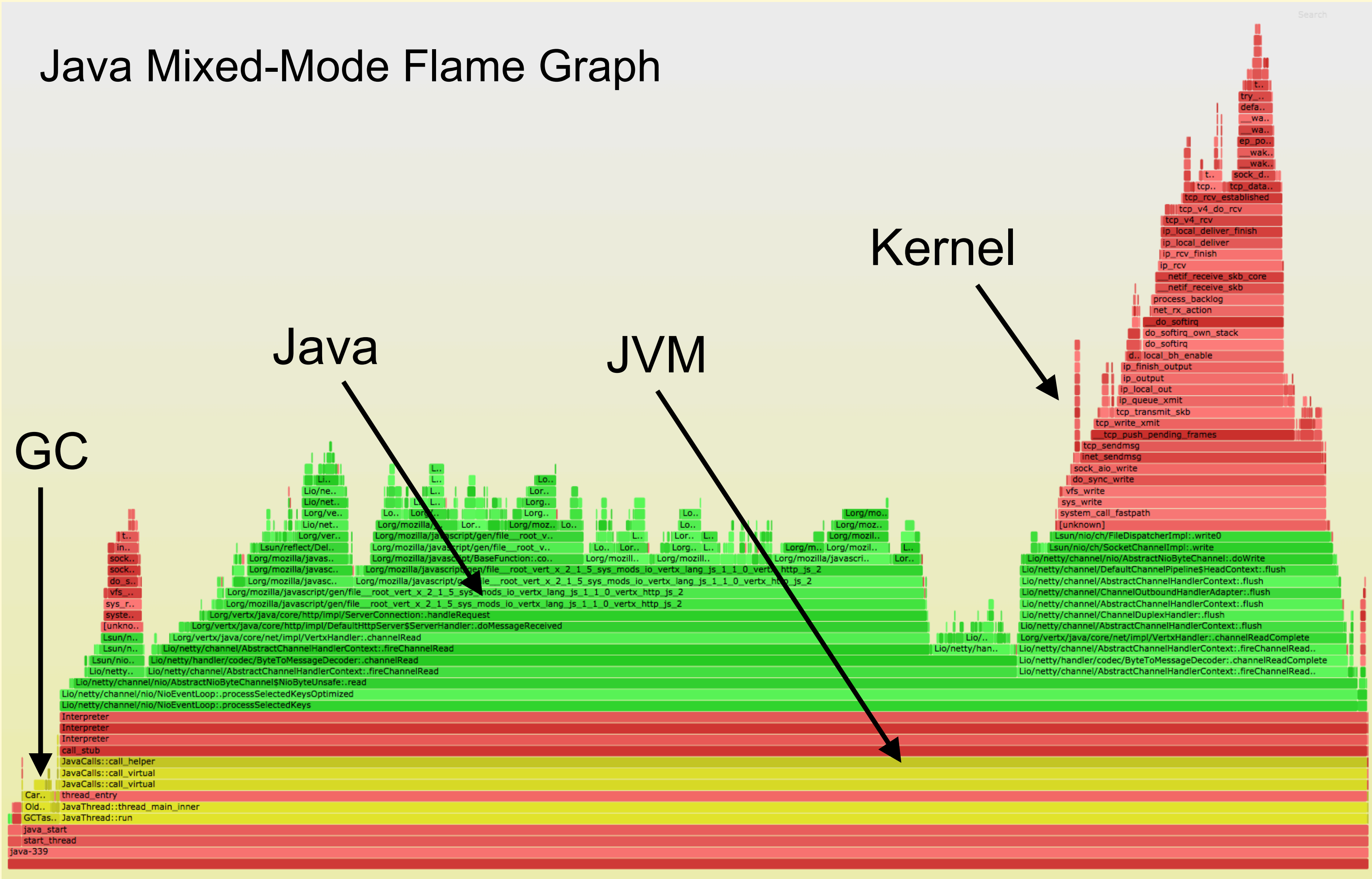
- **Trust nothing**, verify everything
  – Cross-check with other observability tools
  – Write small "known" workloads, and confirm metrics match
  – Find other sanity tests: e.g. check known system limits
  – Determine how metrics are calculated, averaged, updated
- Find metrics to solve problems
  – Instead of understanding hundreds of system metrics
  – What problems do you want to observe? What metrics would be sufficient? Find, verify, and use those. e.g., USE Method.
  – **The metric you want may not yet exist**
- File bugs, get these fixed/improved

Observe Everything

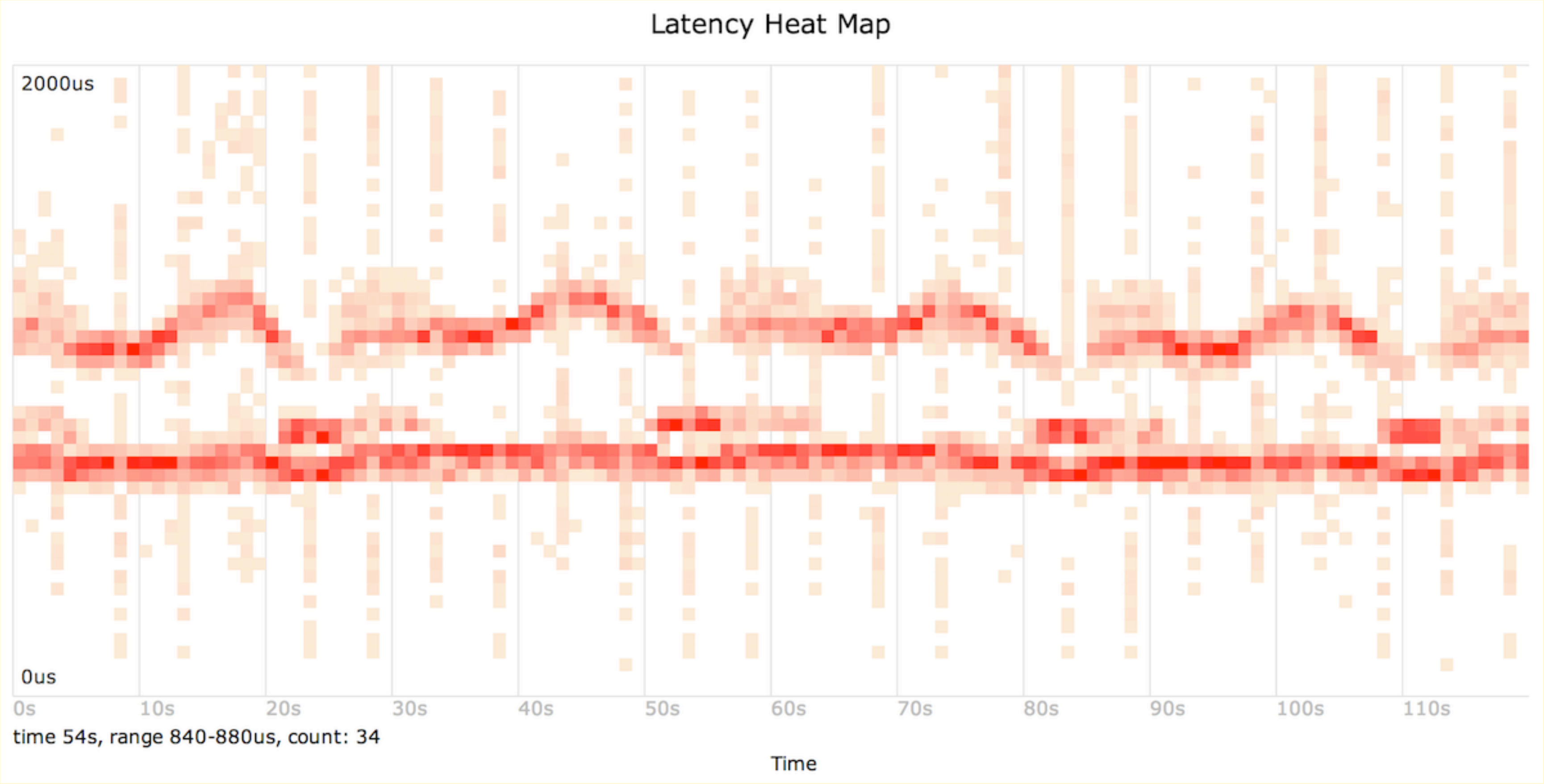
- Use functional diagrams to pose Q's and find missing metrics:



Profile Everything



Visualize Everything



Benchmark Nothing

- **Trust nothing**, verify everything
- Do Active Benchmarking:
  1. Configure the benchmark to run in steady state, 24x7
  2. Do root-cause analysis of benchmark performance
  3. Answer: why is it not 10x?

Links & References

- https://www.rfc-editor.org/rfc/rfc546.pdf
- https://upload.wikimedia.org/wikipedia/commons/6/64/Intel_Nehalem_arch.svg
- http://www.linuxatemyram.com/
- http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/
- 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/blog/2014-05-03/the-benchmark-paradox.html
- http://www.brendangregg.com/ActiveBenchmarking/bonnie++.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/sf2015/presentation/profilers-lying
- Caution signs drawn be me, inspired by real-world signs

THANKS

Thanks

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