The Thirteenth Annual
Southern California Linux Expo

Feb 2015

Linux Profiling at Netflix

using perf_events (aka "perf")

Brendan Gregg

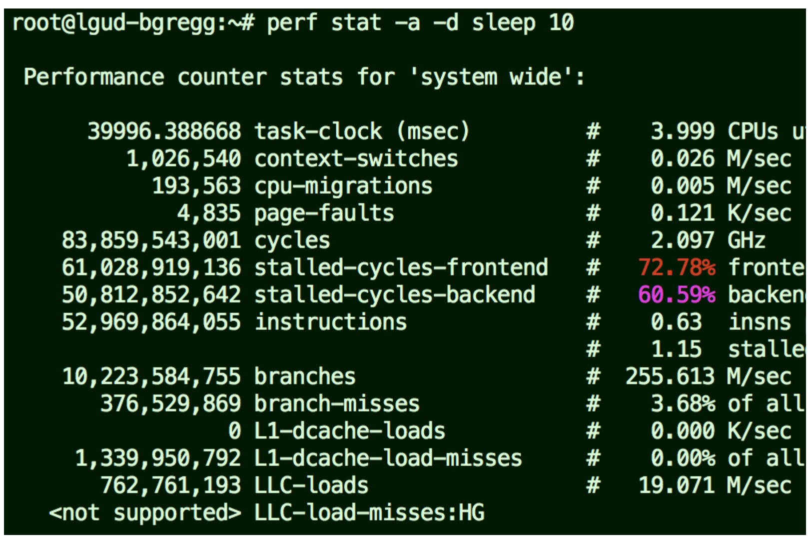
Senior Performance Architect

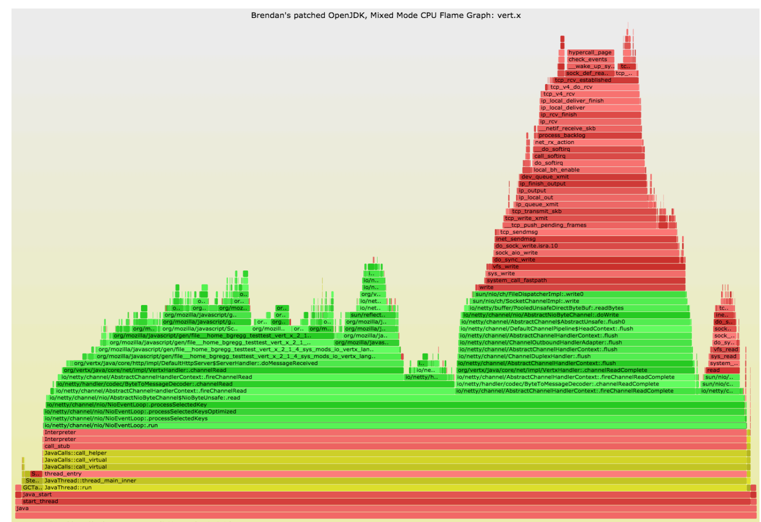
Performance Engineering Team

bgregg@netflix.com @brendangregg

This Talk

- This talk is about Linux profiling using perf_events
  - How to get CPU profiling to work, and overcome gotchas
  - A tour of perf_events and its features
- This is based on our use of perf_events at Netflix





NETFLIX

- Massive Amazon EC2 Linux cloud
  - Tens of thousands of instances
  - Autoscale by ~3k each day
  - CentOS and Ubuntu, Java and Node.js
- FreeBSD for content delivery
  - Approx 33% of the US Internet traffic at night
- Performance is critical
  - Customer satisfaction: >50M subscribers
  - $$$ price/performance
  - Develop tools for cloud-wide analysis, and make them open source: NetflixOSS
  - Use server tools as needed

Agenda

1. Why We Need Linux Profiling
2. Crash Course
3. CPU Profiling
4. Gotchas
   – Stacks (gcc, Java)
   – Symbols (Node.js, Java)
   – Guest PMCs
   – PEBS
   – Overheads
5. Tracing

1. Why We Need Linux Profiling

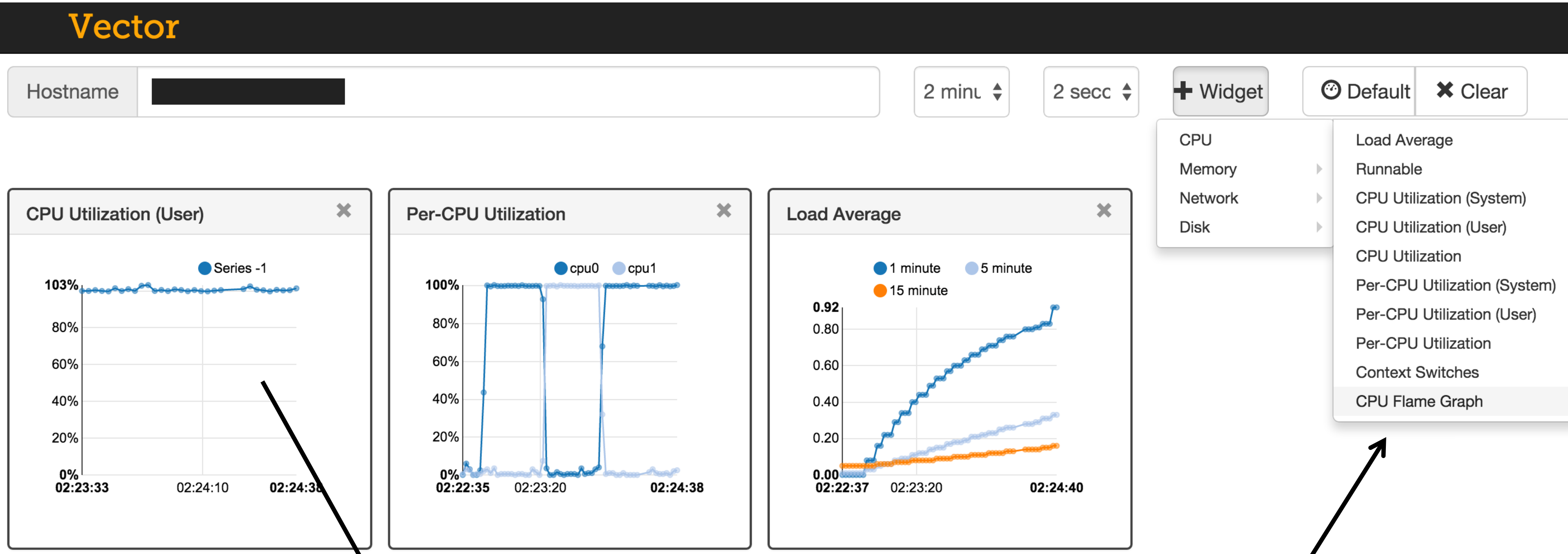
Why We Need Linux Profiling

- Our primary motivation is simple:

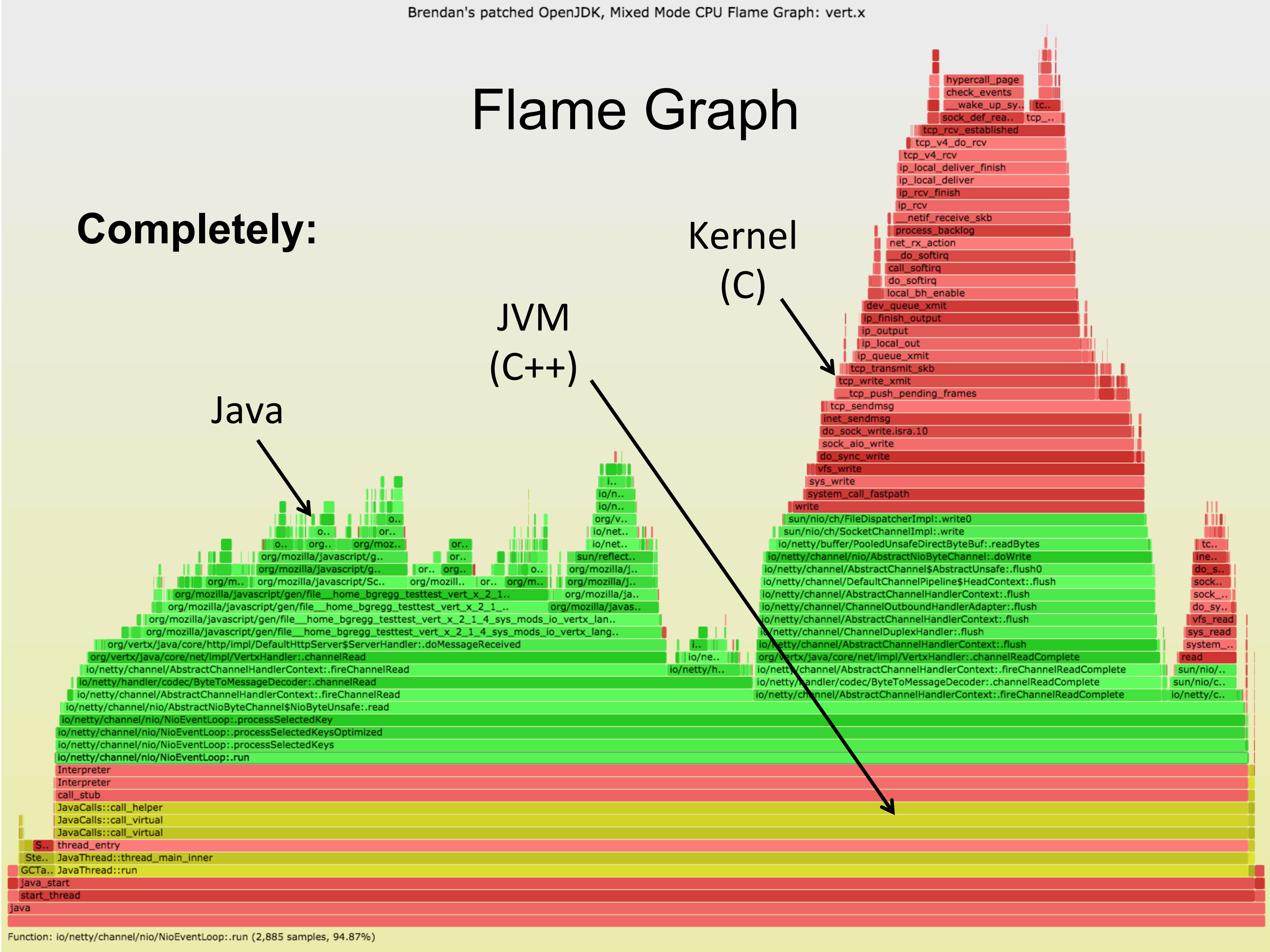
  Understand CPU usage **quickly** and **completely**

Netflix Vector

Quickly:



1. Observe high CPU usage
2. Generate a perf_events-based flame graph


Brendan's patched OpenJDK, Mixed Mode CPU Flame Graph: vert.x
Flame Graph
Completely:
Kernel
(C)
JVM
(C++)
Java
hypercall_page
check_events
__wake_up_sy..
sock_def_rea..
tcp_rcv_established
tcp_v4_do_rcv
tcp_v4_rcv
ip_local_deliver_finish
ip_local_deliver
ip_rcv_finish
ip_rcv
__netif_receive_skb
process_backlog
net_rx_action
__do_softirq
call_softirq
do_softirq
local_bh_enable
dev_queue_xmit
ip_finish_output
ip_output
ip_local_out
ip_queue_xmit
tcp_transmit_skb
tcp_write_xmit
__tcp_push_pending_frames
tcp_sendmsg
inet_sendmsg
do_sock_write.isra.10
sock_aio_write
do_sync_write
vfs_write
sys_write
system_call_fastpath
write
sun/nio/ch/FileDispatcherImpl:.write0
sun/nio/ch/SocketChannelImpl:.write
io/netty/buffer/PooledUnsafeDirectByteBuf:.readBytes
io/netty/channel/nio/AbstractNioByteChannel:.doWrite
io/netty/channel/AbstractChannel$AbstractUnsafe:.flush0
io/netty/channel/DefaultChannelPipeline$HeadContext:.flush
io/netty/channel/AbstractChannelHandlerContext:.flush
io/netty/channel/ChannelOutboundHandlerAdapter:.flush
io/netty/channel/ChannelDuplexHandler:.flush
org/vertx/java/core/net/impl/VertxHandler:.channelReadComplete
io/netty/channel/AbstractChannelHandlerContext:.fireChannelReadComplete
io/netty/handler/codec/ByteToMessageDecoder:.channelReadComplete
org/mozilla/javascript/gen/file__home_bgregg_testtest_vert_x_2_1_4_sys_mods_io_vertx_lang..
org/vertx/java/core/http/impl/DefaultHttpServer$ServerHandler:.doMessageReceived
org/vertx/java/core/net/impl/VertxHandler:.channelRead
io/netty/channel/AbstractChannelHandlerContext:.fireChannelRead
io/netty/handler/codec/ByteToMessageDecoder:.channelRead
io/netty/channel/nio/AbstractNioByteChannel$NioByteUnsafe:.read
io/netty/channel/nio/NioEventLoop:.processSelectedKey
io/netty/channel/nio/NioEventLoop:.processSelectedKeysOptimized
io/netty/channel/nio/NioEventLoop:.processSelectedKeys
io/netty/channel/nio/NioEventLoop:.run
Interpreter
call_stub
JavaCalls::call_helper
JavaCalls::call_virtual
thread_entry
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java
Function: io/netty/channel/nio/NioEventLoop:.run (2,885 samples, 94.87%)

Value for Netflix

- Uses for CPU profiling:
  - Help with incident response
  - Non-regression testing
  - Software evaluations
  - Identify performance tuning targets
  - Part of CPU workload characterization
- Built into Netflix Vector
  - A near real-time instance analysis tool (will be NetflixOSS)

Workload Characterization

- For CPUs:

    1. **Who**
    2. **Why**
    3. **What**
    4. **How**

Workload Characterization

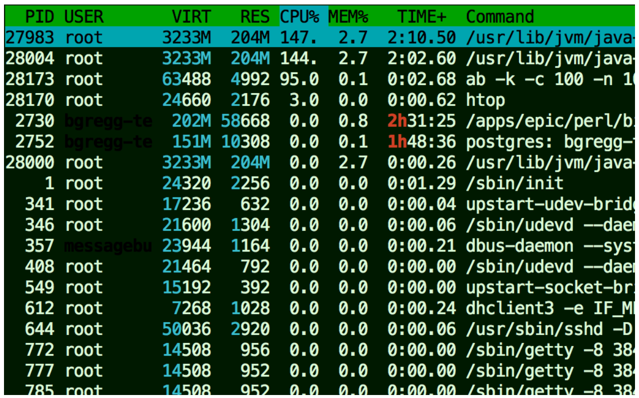
- For CPUs:

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

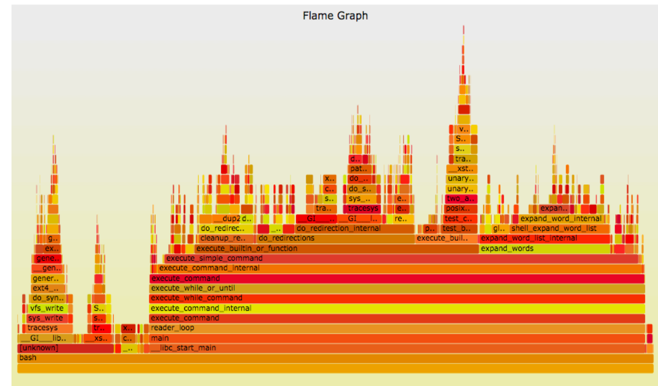
- Can you currently answer them? How?

CPU Tools

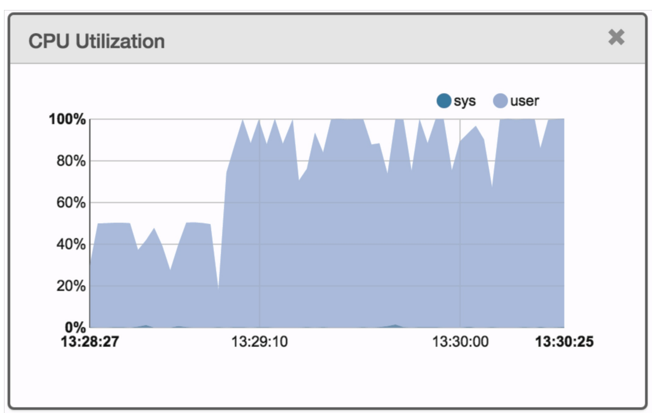
Who



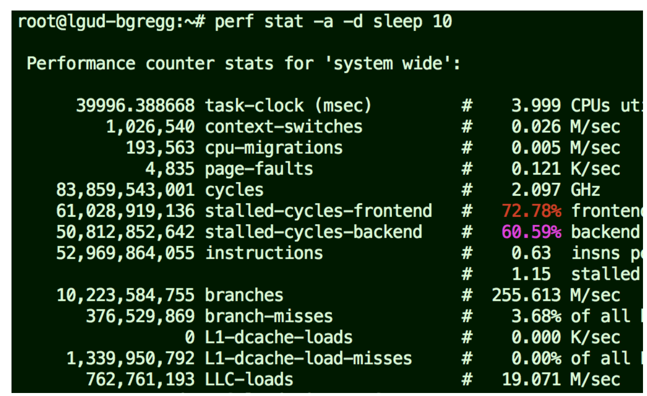
Why



How

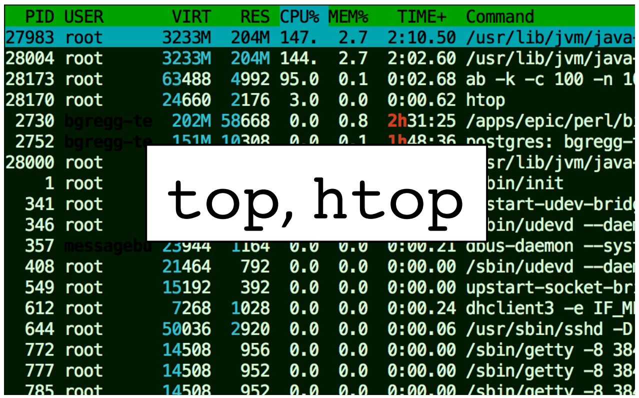


What

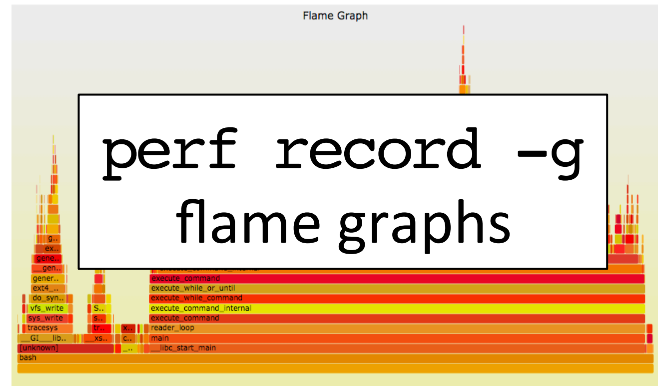


CPU Tools

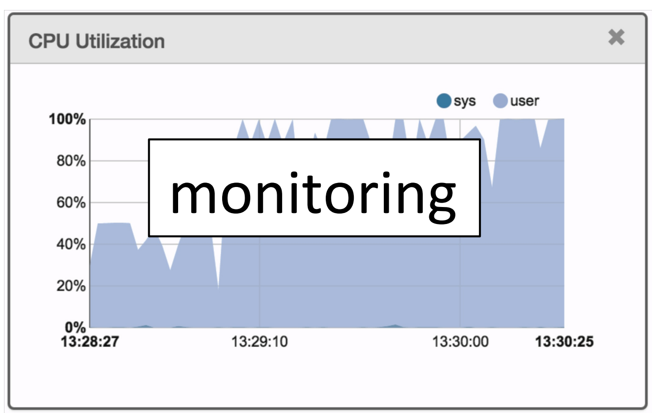
Who



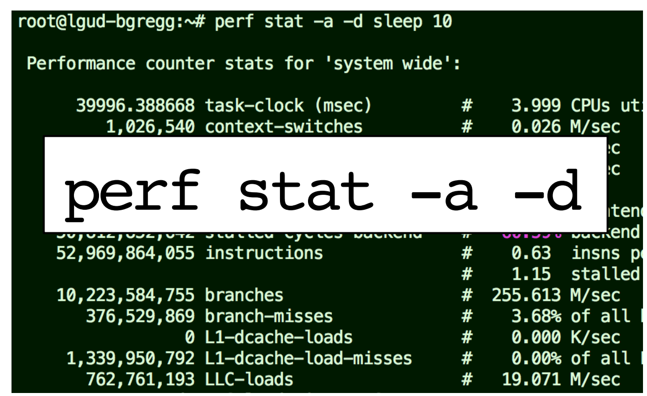
Why



How

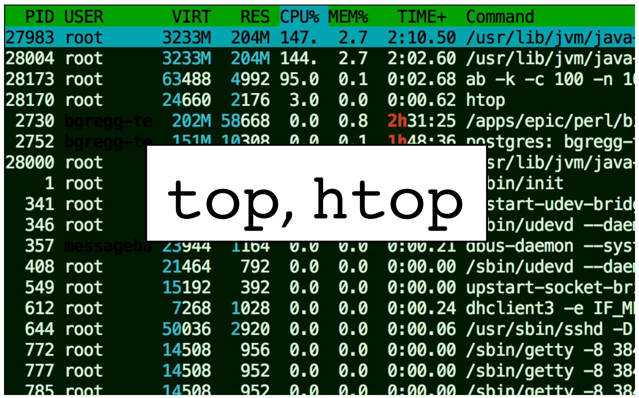


What

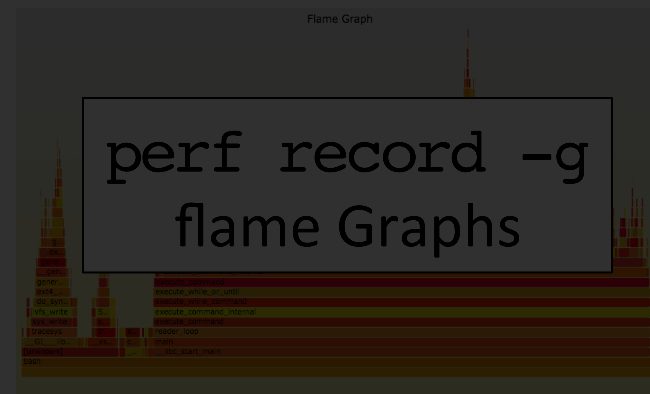


Most companies & monitoring products today

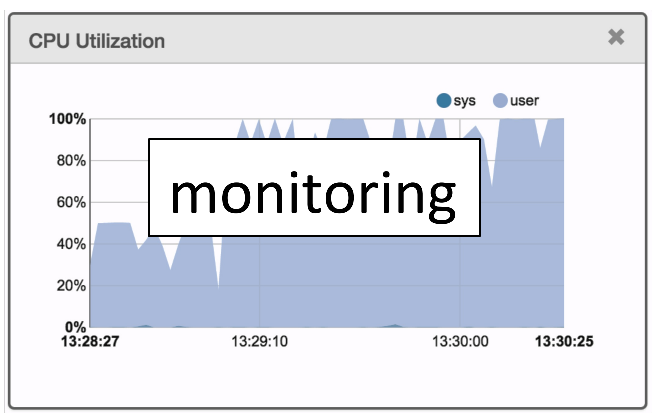
Who



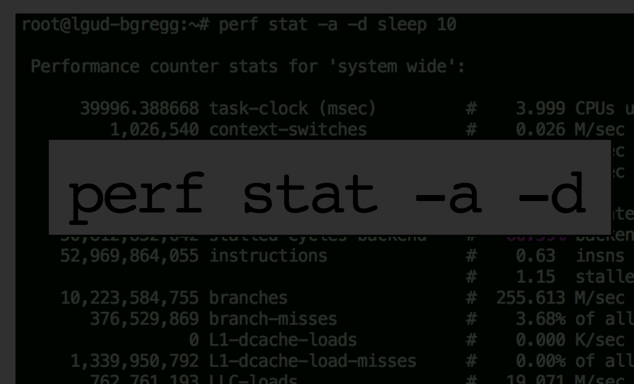
Why



How



What



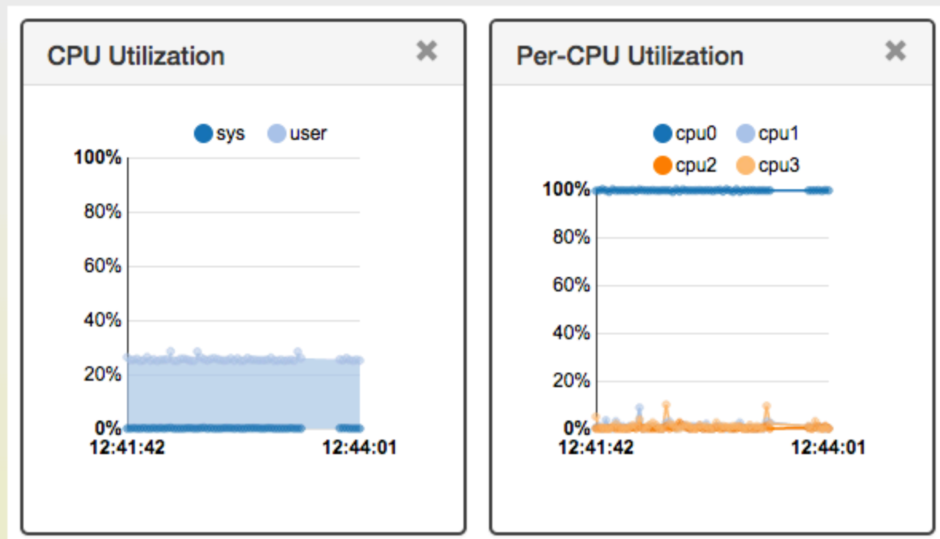
Re-setting Expectations

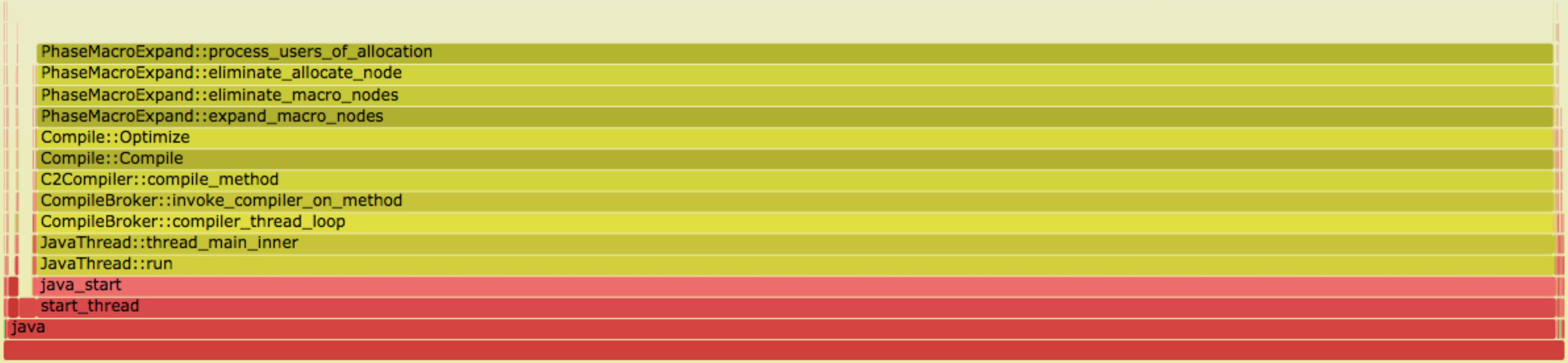
- That was pretty good… 20 years ago.
- Today you should easily understand *why* CPUs are used:
  - A profile of all CPU consumers and code paths
  - Visualized effectively
  - This *should* be easy to do
- Best done using:
  - A perf_events CPU profile of stack traces
  - Visualized as a flame graph
  - This will usually mean some sysadmin/devops work, to get perf_events working, and to automate profiling

Recent Example

CPU Flame Graph (no idle): , 2015-02-05_20:38:52

1. Poor performance, and 1 CPU at 100%
2. perf_events flame graph shows JVM stuck compiling





2. Crash Course

perf_events

- The main Linux profiler, used via the "perf" command
- Add from linux-tools-common, etc.
- Source code & docs in Linux: tools/perf
- Supports many profiling/tracing features:
  - CPU Performance Monitoring Counters (PMCs)
  - Statically defined tracepoints
  - User and kernel dynamic tracing
  - Kernel line and local variable tracing
  - Efficient in-kernel counts and filters
  - Stack tracing, libunwind
  - Code annotation
- Some bugs in the past; has been stable for us

perf_events
ponycorn

A Multitool of Subcommands

```
# perf

 usage: perf [--version] [--help] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   diff            Read two perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   inject          Filter to augment the events stream with additional information
   kmem            Tool to trace/measure kernel memory(slab) properties
   kvm             Tool to trace/measure kvm guest os
   list            List all symbolic event types
   lock            Analyze lock events
   probe           Define new dynamic tracepoints
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   test            Runs sanity tests.
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.

 See 'perf help COMMAND' for more information on a specific command.
```

perf Command Format

- perf instruments using `stat` or `record`. This has three main parts: action, event, scope.
- e.g., profiling on-CPU stack traces:

Action: record stack traces

```
perf record -F 99 -a -g -- sleep 10
```

Scope: all CPUs

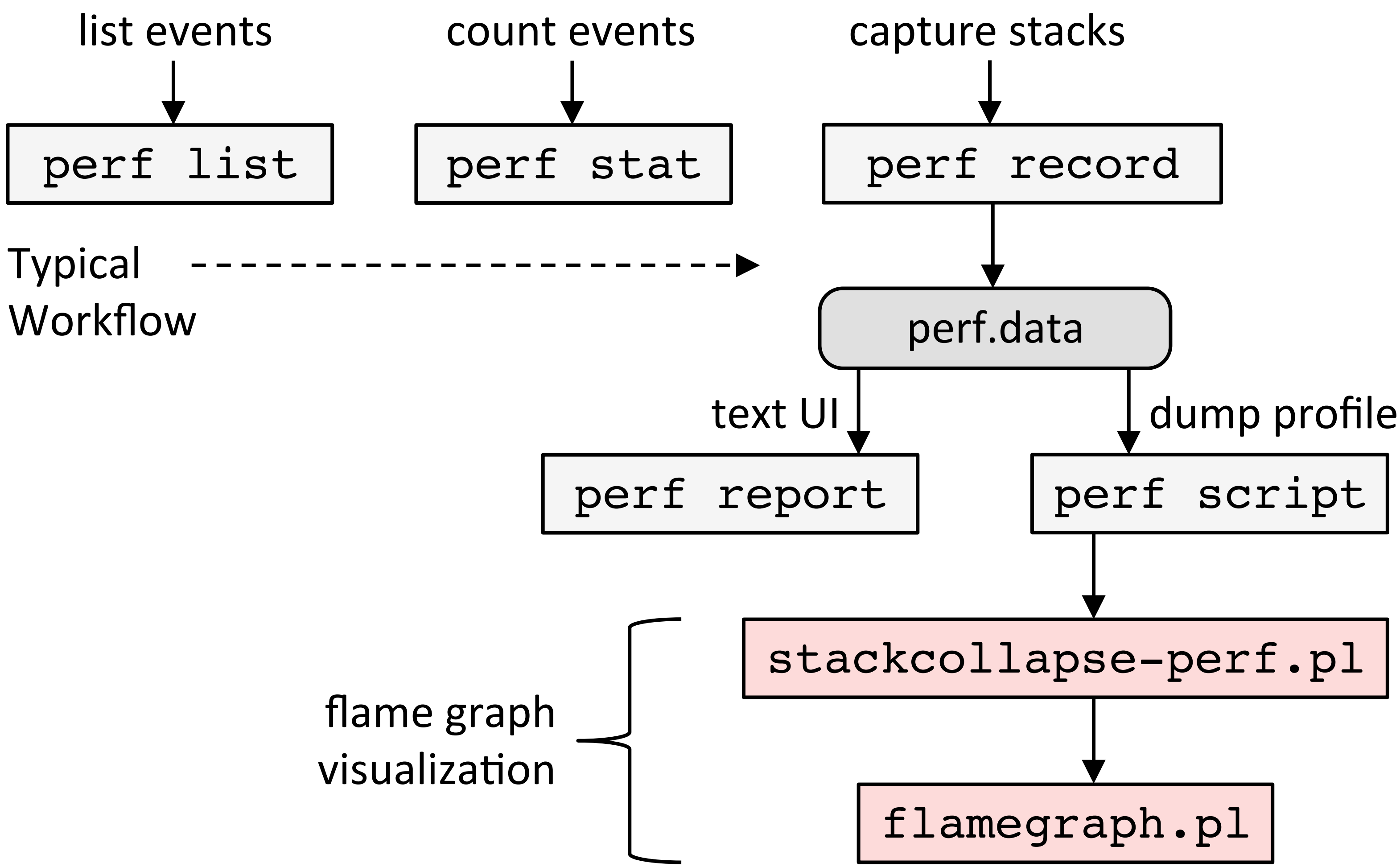
Event: 99 Hertz

Note: sleep 10 is a dummy command to set the duration

perf Actions

- Count events (`perf stat …`)
  - Uses an efficient in-kernel counter, and prints the results
- Sample events (`perf record …`)
  - Records details of every event to a dump file (perf.data)
    - Timestamp, CPU, PID, instruction pointer, …
  - This incurs higher overhead, relative to the rate of events
  - Include the call graph (stack trace) using `-g`
- Other actions include:
  - List events (`perf list`)
  - Report from a perf.data file (`perf report`)
  - Dump a perf.data file as text (`perf script`)
  - top style profiling (`perf top`)

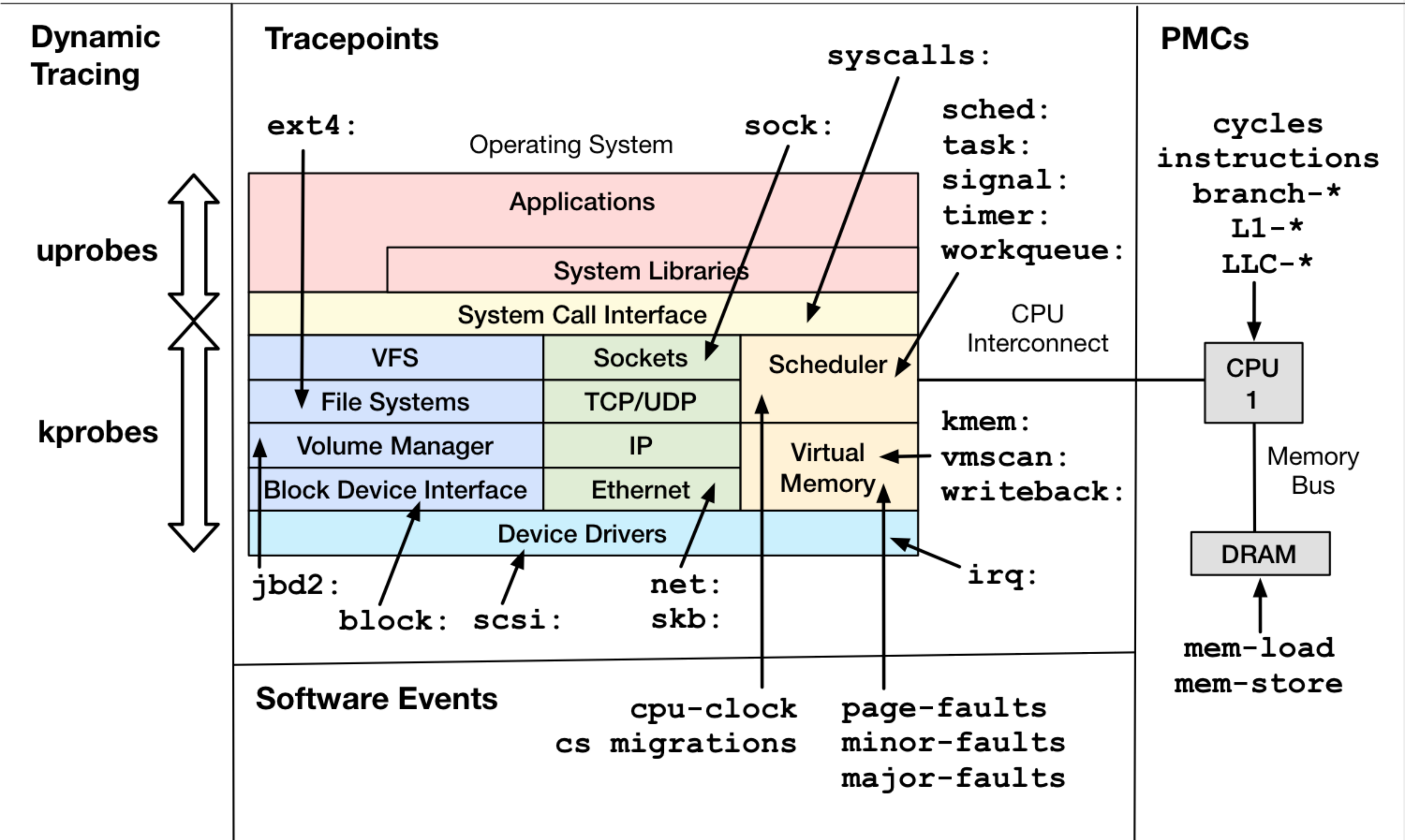
perf Actions: Workflow



perf Events

- Custom timers
  - e.g., 99 Hertz (samples per second)
- Hardware events
  - CPU Performance Monitoring Counters (PMCs)
- Tracepoints
  - Statically defined in software
- Dynamic tracing
  - Created using uprobes (user) or kprobes (kernel)
  - Can do kernel line tracing with local variables (needs kernel debuginfo)

perf Events: Map



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

perf Events: List

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                              [Hardware event]
  instructions                                      [Hardware event]
  cache-references                                  [Hardware event]
  cache-misses                                      [Hardware event]
  branch-instructions OR branches                   [Hardware event]
  branch-misses                                     [Hardware event]
  bus-cycles                                        [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend   [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend     [Hardware event]
[…]
  cpu-clock                                         [Software event]
  task-clock                                        [Software event]
  page-faults OR faults                             [Software event]
  context-switches OR cs                            [Software event]
  cpu-migrations OR migrations                      [Software event]
[…]
  L1-dcache-loads                                   [Hardware cache event]
  L1-dcache-load-misses                             [Hardware cache event]
  L1-dcache-stores                                  [Hardware cache event]
[…]
  skb:kfree_skb                                     [Tracepoint event]
  skb:consume_skb                                   [Tracepoint event]
  skb:skb_copy_datagram_iovec                       [Tracepoint event]
  net:net_dev_xmit                                  [Tracepoint event]
  net:net_dev_queue                                 [Tracepoint event]
  net:netif_receive_skb                             [Tracepoint event]
  net:netif_rx                                      [Tracepoint event]
[…]
```

perf Scope

- System-wide: all CPUs (`-a`)
- Target PID (`-p PID`)
- Target command (…)
- Specific CPUs (`-c …`)
- User-level only (`<event>:u`)
- Kernel-level only (`<event>:k`)
- A custom filter to match variables (`--filter …`)

The following one-liner tour includes some complex action, event, and scope combinations.

One-Liners: Listing Events

```
# Listing all currently known events:
perf list

# Searching for "sched" tracepoints:
perf list | grep sched

# Listing sched tracepoints:
perf list 'sched:*'
```

One-Liners: Counting Events

```
# CPU counter statistics for the specified command:
perf stat command

# Detailed CPU counter statistics (includes extras) for the specified command:
perf stat -d command

# CPU counter statistics for the specified PID, until Ctrl-C:
perf stat -p PID

# CPU counter statistics for the entire system, for 5 seconds:
perf stat -a sleep 5

# Various CPU last level cache statistics for the specified command:
perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches command

# Count system calls for the specified PID, until Ctrl-C:
perf stat -e 'syscalls:sys_enter_*' -p PID

# Count scheduler events for the specified PID, for 10 seconds:
perf stat -e 'sched:*' -p PID sleep 10

# Count block device I/O events for the entire system, for 10 seconds:
perf stat -e 'block:*' -a sleep 10

# Show system calls by process, refreshing every 2 seconds:
perf top -e raw_syscalls:sys_enter -ns comm
```

One-Liners: Profiling Events

```
# Sample on-CPU functions for the specified command, at 99 Hertz:
perf record -F 99 command

# Sample on-CPU functions for the specified PID, at 99 Hertz, until Ctrl-C:
perf record -F 99 -p PID

# Sample CPU stack traces for the specified PID, at 99 Hertz, for 10 seconds:
perf record -F 99 -p PID -g -- sleep 10

# Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds:
perf record -F 99 -ag -- sleep 10

# Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 s:
perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5

# Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds:
perf record -e LLC-load-misses -c 100 -ag -- sleep 5

# Sample on-CPU kernel instructions, for 5 seconds:
perf record -e cycles:k -a -- sleep 5

# Sample on-CPU user instructions, for 5 seconds:
perf record -e cycles:u -a -- sleep 5
```

One-Liners: Reporting

```
# Show perf.data in an ncurses browser (TUI) if possible:
perf report

# Show perf.data with a column for sample count:
perf report -n

# Show perf.data as a text report, with data coalesced and percentages:
perf report --stdio

# List all raw events from perf.data:
perf script

# List all raw events from perf.data, with customized fields:
perf script -f comm,tid,pid,time,cpu,event,ip,sym,dso

# Dump raw contents from perf.data as hex (for debugging):
perf script -D

# Disassemble and annotate instructions with percentages (needs some debuginfo):
perf annotate --stdio
```

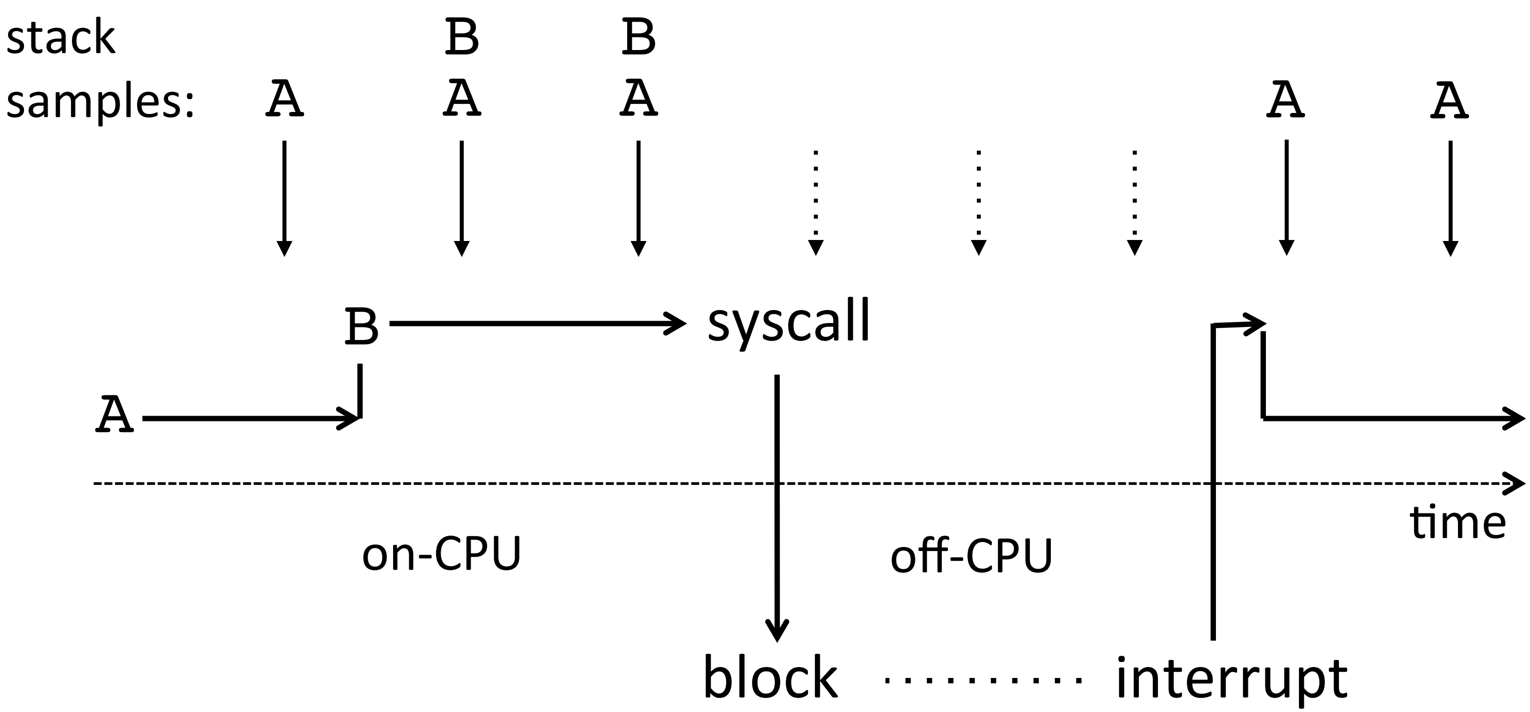
And More

- perf can also probe and record dynamic tracepoints, and custom CPU PMCs
- This can get a little advanced
- I'll start with CPU profiling, then gotchas

3. CPU Profiling

CPU Profiling

- Record stacks at a timed interval: simple and effective
  - Pros: Low (deterministic) overhead
  - Cons: Coarse accuracy, but usually sufficient



perf Screenshot

- Sampling full stack traces at 99 Hertz:

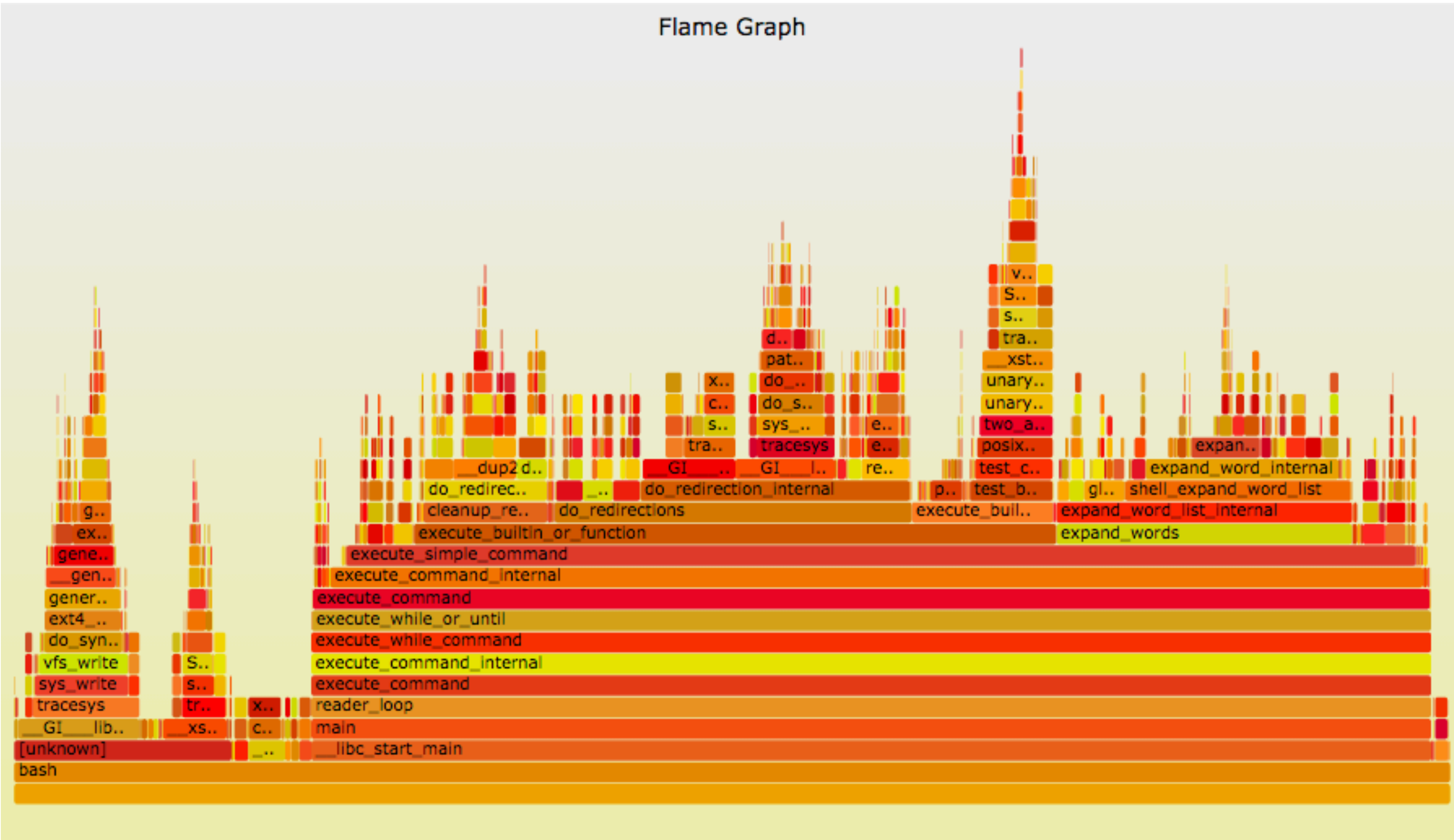
```
# perf record -F 99 -ag -- sleep 30
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.745 MB perf.data (~119930 samples) ]
# perf report -n --stdio
1.40%   162             java  [kernel.kallsyms]       [k] _raw_spin_lock
          |
          --- _raw_spin_lock
             |
             |--63.21%-- try_to_wake_up
             |          |
             |          |--63.91%-- default_wake_function
             |          |          |
             |          |          |--56.11%-- __wake_up_common
             |          |          |          __wake_up_locked
             |          |          |          ep_poll_callback
             |          |          |          __wake_up_common
             |          |          |          __wake_up_sync_key
             |          |          |          |
             |          |          |          |--59.19%-- sock_def_readable
[…78,000 lines truncated…]
```

perf Reporting

- perf report summarizes by combining common paths
- Previous output truncated 78,000 lines of summary
- The following is what a mere 8,000 lines looks like…

perf report

… as a Flame Graph



Flame Graphs

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record -F 99 -a –g -- sleep 30
perf script | ./stackcollapse-perf.pl |./flamegraph.pl > perf.svg
```

- Flame Graphs:
  - **x-axis**: alphabetical stack sort, to maximize merging
  - **y-axis**: stack depth
  - **color**: random, or hue can be a dimension
    - e.g., software type, or difference between two profiles for non-regression testing ("differential flame graphs")
  - interpretation: top edge is on-CPU, beneath it is ancestry
- Just a Perl program to convert perf stacks into SVG
  - Includes JavaScript: open in a browser for interactivity
- Easy to get working
  http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

4. Gotchas

When you try to use perf

- Stacks don't work (missing)
- Symbols don't work (hex numbers)
- Can't profile Java
- Can't profile Node.js/io.js
- PMCs aren't available
- Dynamic tracing function arguments don't work
- perf locks up

How to *really* get started

1. Get "perf" to work
2. Get stack walking to work
3. Fix symbol translation
4. Get IPC to work
5. Test perf under load

This is my actual checklist.

How to *really* get started

1. Get "perf" to work
2. Get stack walking to work
3. Fix symbol translation
4. Get IPC to work
5. Test perf under load

Install perf-tools-common and perf-tools-`uname -r` packages; Or compile in the Linux source: tools/perf

The "gotchas"...

This is my actual checklist.

Gotcha #1 Broken Stacks

```
perf record -F 99 -a –g -- sleep 30
perf report -n --stdio
```

Start by testing stacks:

1. Take a CPU profile
2. Run perf report
3. If stacks are often < 3 frames, or don't reach "thread start" or "main", they are probably broken. Fix them.

Identifying Broken Stacks

```
28.10%              146      sed  libc-2.19.so       [.] re_search_internal
                 |
                 --- re_search_internal
                        |
                        |--12.25%-- 0x3
                        |           0x100007
```

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
                        |
                         --3.22%-- rpl_re_search
                                   match_regex
                                   do_subst
                                   execute_program
                                   process_files
                                   main
                                   __libc_start_main
```

not broken

Identifying Broken Stacks

```
 78.50%            409      sed  libc-2.19.so       [.] 0x00000000000dd7d4
                    |
                    |--3.65%-- 0x7f2516d5d10d
                    |
                    |--2.19%-- 0x7f2516d0332f
                    |
                    |--1.22%-- 0x7f2516cffbd2
                    |
                    |--1.22%-- 0x7f2516d5d5ad
```

broken

```
 71.79%            334      sed   sed                [.] 0x000000000001afc1
                    |
                    |--11.65%-- 0x40a447
                    |           0x40659a
                    |           0x408dd8
                    |           0x408ed1
                    |           0x402689
                    |           0x7fa1cd08aec5
                    |
                    |--1.33%-- 0x40a4a1
                    |          |
                    |          |--60.01%-- 0x40659a
                    |          |           0x408dd8
                    |          |           0x408ed1
                    |          |           0x402689
                    |          |           0x7fa1cd08aec5
```

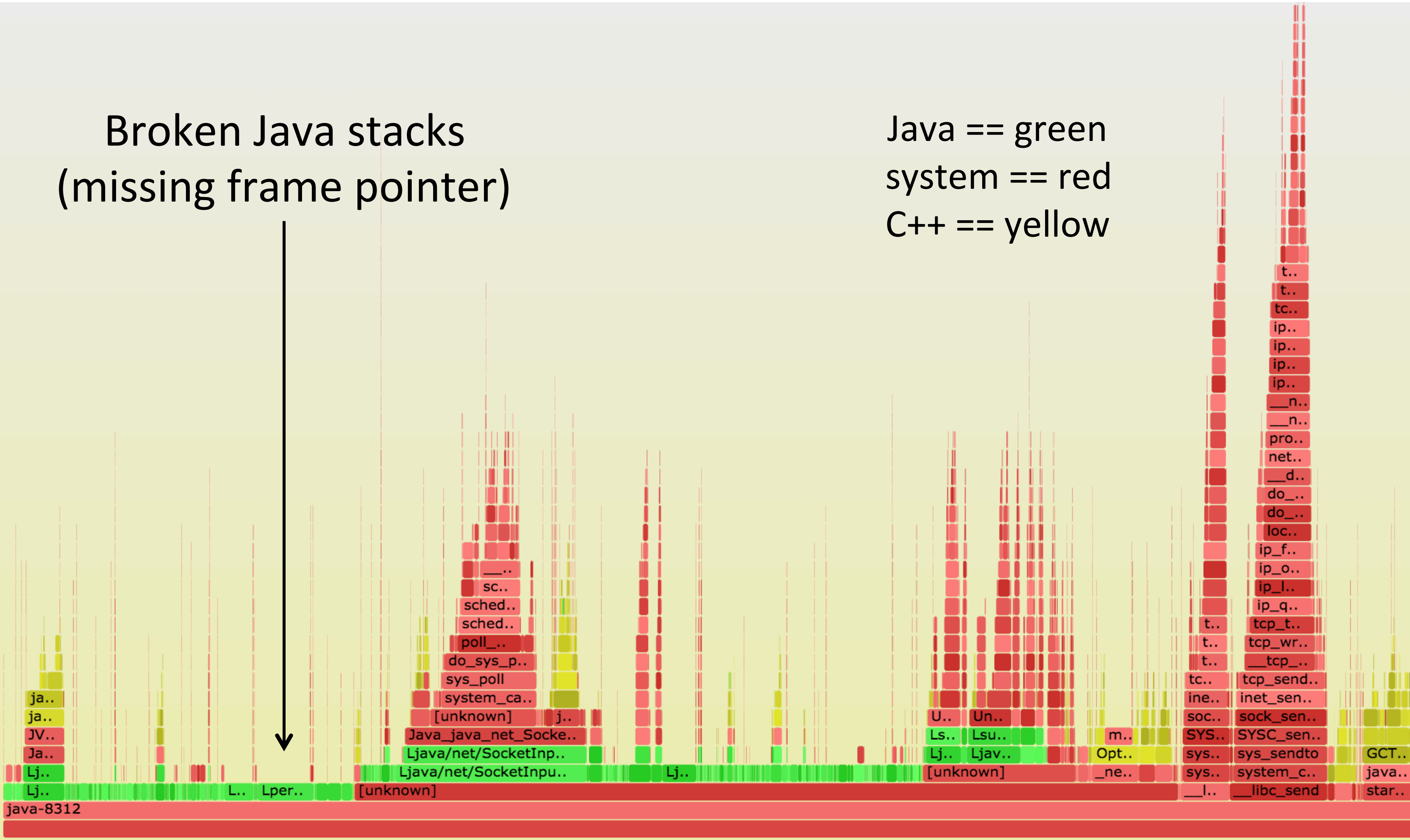
probably not broken

missing symbols, but that's another problem

Broken Stacks Flame Graph

Broken Java stacks
(missing frame pointer)

Java == green
system == red
C++ == yellow



Fixing Broken Stacks

Either:

A. Fix frame pointer-based stack walking (the default)
   - Pros: simple, supports any system stack walker
   - Cons: might cost a little extra CPU to make available

B. Use libunwind and DWARF: `perf record -g dwarf`
   - Pros: more debug info
   - Cons: not on older kernels, and inflates instance size

C. Use a custom walker (probably needs kernel support)
   - Pros: full stack walking (e.g., unwind inlined frames) & args
   - Cons: custom kernel code, can cost more CPU when in use

Our current preference is (A)

- So how do we fix the frame pointer…

gcc -fno-omit-frame-pointer

- *Once upon a time*, x86 had fewer registers, and the frame pointer register was reused for general purpose to improve performance. This breaks system stack walking.
- gcc provides `-fno-omit-frame-pointer` to fix this
  - Please make this the default in gcc!

JDK-8068945

- Java's compilers also reuse the frame pointer, and unfortunately there's no -XX:no-omit-frame-pointer (yet)
- I hacked hotspot to fix the frame pointer, and published the patch as a prototype suggestion (JDK-8068945)

```
--- openjdk8clean/hotspot/src/cpu/x86/vm/x86_64.ad 2014-03-04 …
+++ openjdk8/hotspot/src/cpu/x86/vm/x86_64.ad 2014-11-08 …
@@ -166,10 +166,9 @@
 // 3) reg_class stack_slots( /* one chunk of stack-based "registers" */ )
 //

-// Class for all pointer registers (including RSP)
+// Class for all pointer registers (including RSP, excluding RBP)
 reg_class any_reg(RAX, RAX_H,
                   RDX, RDX_H,
-                  RBP, RBP_H,
                   RDI, RDI_H,
                   RSI, RSI_H,
                   RCX, RCX_H,
[...]
```

Remove RBP from register pools

JDK-8068945

```
--- openjdk8clean/hotspot/src/cpu/x86/vm/macroAssembler_x86.cpp 2014-03-04…
+++ openjdk8/hotspot/src/cpu/x86/vm/macroAssembler_x86.cpp 2014-11-07 …
@@ -5236,6 +5236,7 @@
     // We always push rbp, so that on return to interpreter rbp, will be
     // restored correctly and we can correct the stack.
     push(rbp);
+    mov(rbp, rsp);
     // Remove word for ebp
     framesize -= wordSize;

--- openjdk8clean/hotspot/src/cpu/x86/vm/c1_MacroAssembler_x86.cpp …
+++ openjdk8/hotspot/src/cpu/x86/vm/c1_MacroAssembler_x86.cpp …
[...]
```

Fix x86-64 function prologues

- We've been using our patched OpenJDK for profiling
- To do: make this an option (-XX:MoreFramePointer), and (at some point) fix for invokedynamic
  - See "A hotspot patch for stack profiling (frame pointer)" on the hotspot compiler dev mailing list, being discussed now

Broken Java Stacks

```
# 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)

java  4579 cpu-clock:
      7f417908c194 [unknown] (/tmp/perf-4458.map)
[…]
```

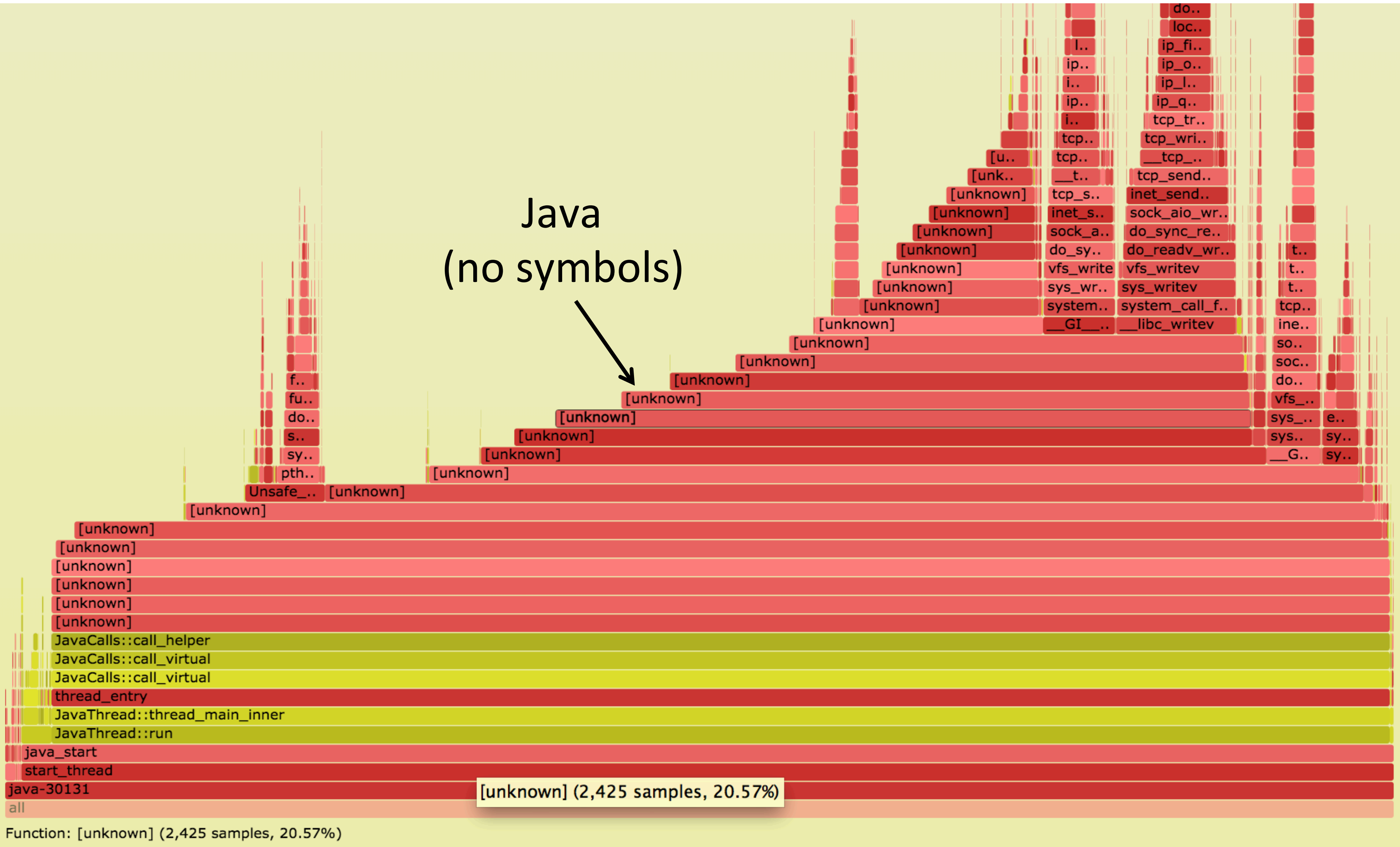
- Check with "perf script" to see stack samples
- These are 1 or 2 levels deep (junk values)

Fixed Java Stacks

```
# perf script
[…]
java 8131 cpu-clock:
        7fff76f2dce1 [unknown] ([vdso])
        7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm…
        7fd301861e46 [unknown] (/tmp/perf-8131.map)
        7fd30184def8 [unknown] (/tmp/perf-8131.map)
        7fd30174f544 [unknown] (/tmp/perf-8131.map)
        7fd30175d3a8 [unknown] (/tmp/perf-8131.map)
        7fd30166d51c [unknown] (/tmp/perf-8131.map)
        7fd301750f34 [unknown] (/tmp/perf-8131.map)
        7fd3016c2280 [unknown] (/tmp/perf-8131.map)
        7fd301b02ec0 [unknown] (/tmp/perf-8131.map)
        7fd3016f9888 [unknown] (/tmp/perf-8131.map)
        7fd3016ece04 [unknown] (/tmp/perf-8131.map)
        7fd30177783c [unknown] (/tmp/perf-8131.map)
        7fd301600aa8 [unknown] (/tmp/perf-8131.map)
        7fd301a4484c [unknown] (/tmp/perf-8131.map)
        7fd3010072e0 [unknown] (/tmp/perf-8131.map)
        7fd301007325 [unknown] (/tmp/perf-8131.map)
        7fd301007325 [unknown] (/tmp/perf-8131.map)
        7fd3010004e7 [unknown] (/tmp/perf-8131.map)
        7fd3171df76a JavaCalls::call_helper(JavaValue*,…
        7fd3171dce44 JavaCalls::call_virtual(JavaValue*…
        7fd3171dd43a JavaCalls::call_virtual(JavaValue*…
        7fd31721b6ce thread_entry(JavaThread*, Thread*)…
        7fd3175389e0 JavaThread::thread_main_inner() (/…
        7fd317538cb2 JavaThread::run() (/usr/lib/jvm/nf…
        7fd3173f6f52 java_start(Thread*) (/usr/lib/jvm/…
        7fd317a7e182 start_thread (/lib/x86_64-linux-gn…
```

- With JDK-8068945 stacks are full, and go all the way to start_thread()
- This is what the CPUs are really running: inlined frames are not present

Fixed Stacks Flame Graph



Gotcha #2 Missing Symbols

- Missing symbols should be obvious in perf report/script:

```
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 installed packages:
  A. Add a -dbgsym package, if available
  B. Recompile from source
- For JIT (Java, Node.js, …):
  A. Create a /tmp/perf-PID.map file. perf already looks for this
  B. Or use one of the new symbol loggers (see lkml)

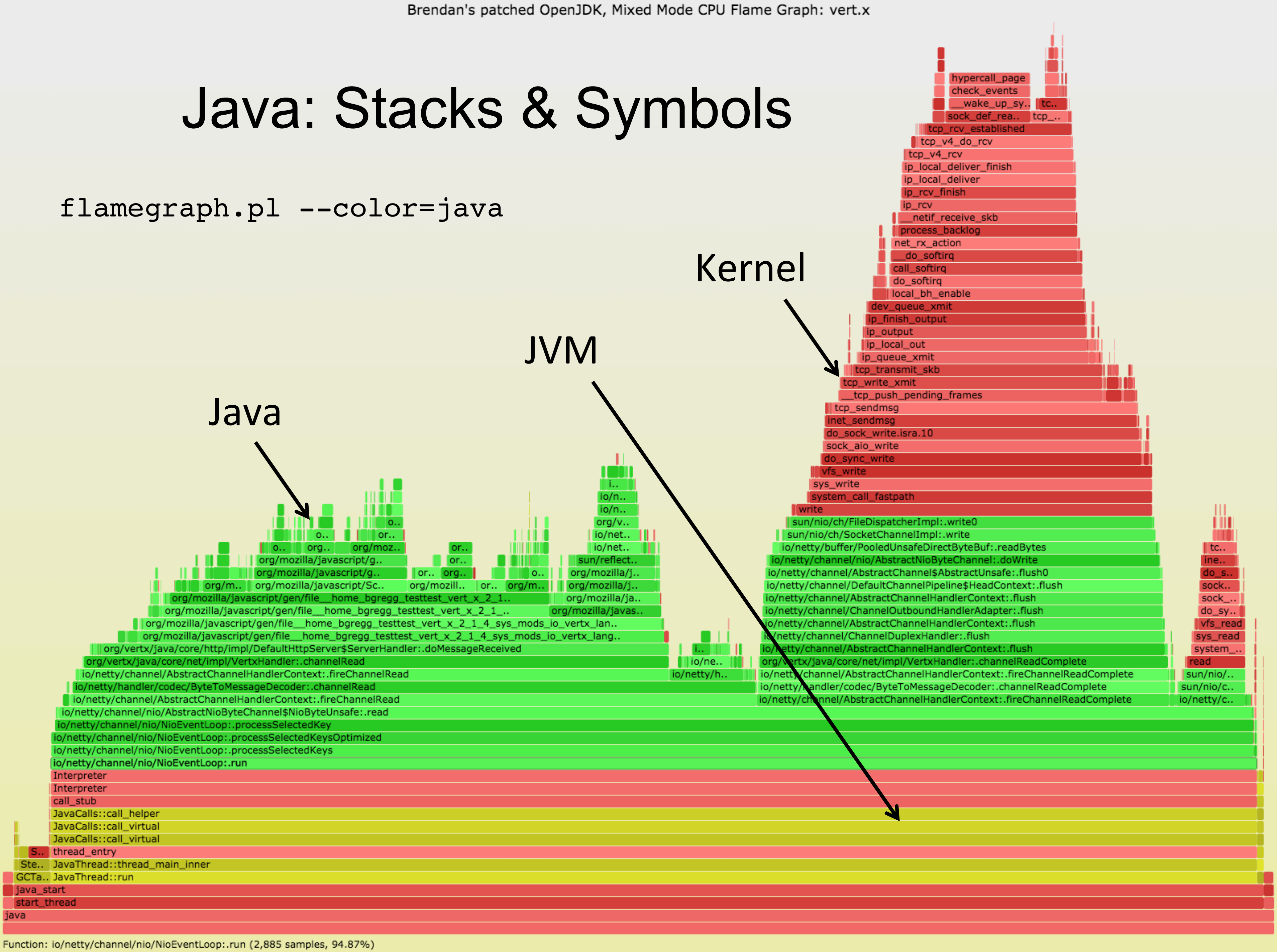
```
# perf script
Failed to open /tmp/perf-8131.map, continuing without symbols
[…]
java 8131 cpu-clock:
    7fff76f2dce1 [unknown] ([vdso])
    7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm…
    7fd301861e46 [unknown] (/tmp/perf-8131.map)
[…]
```

perf JIT symbols: Java, Node.js

- Using the /tmp map file for symbol translation:
  - Pros: simple, can be low overhead (snapshot on demand)
  - Cons: stale symbols
  - Map format is "START SIZE symbolname"
  - Another gotcha: If perf is run as root, chown root <mapfile>
- Java
  - https://github.com/jrudolph/perf-map-agent
  - Agent attaches and writes the map file on demand (previous versions attached on Java start, and wrote continually)
- Node.js
  - `node --perf_basic_prof` writes the map file continually
  - Available from 0.11.13+
  - Currently has a file growth issue; see my patch in https://code.google.com/p/v8/issues/detail?id=3453

Java: Stacks & Symbols

```
flamegraph.pl --color=java
```

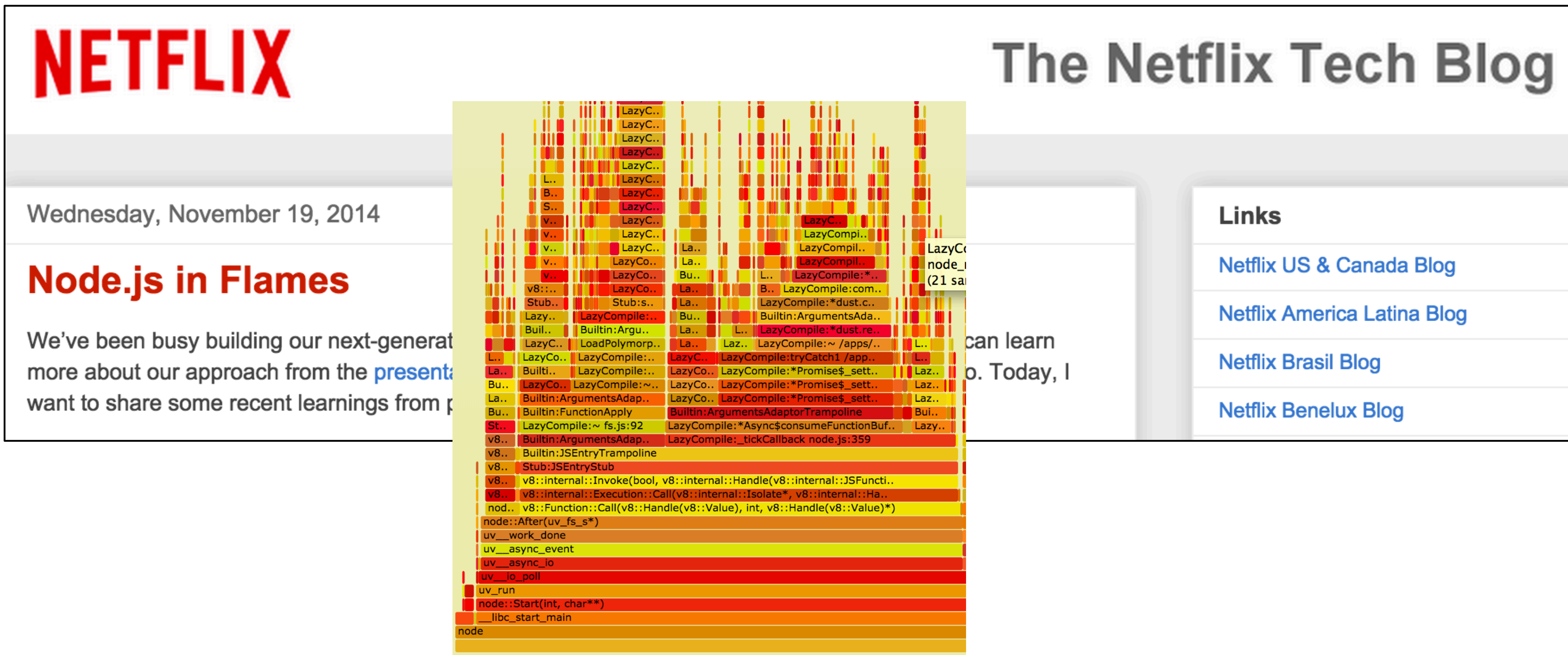


Java: Inlining

- Disabling inlining:
  - -XX:-Inline
  - Many more Java frames
  - 80% slower (in this case)
- Not really necessary
  - Inlined flame graphs often make enough sense
  - Or tune -XX:MaxInlineSize and -XX:InlineSmallCode a little to reveal more frames, without costing much perf
  - Can even go faster!

Node.js: Stacks & Symbols

- Covered previously on the Netflix Tech Blog
  http://techblog.netflix.com/2014/11/nodejs-in-flames.html



Gotcha #3 Guest PMCs

- Using PMCs from a Xen guest (currently):

```
# perf stat -a -d sleep 5

 Performance counter stats for 'system wide':

      10003.718595 task-clock (msec)         #    2.000 CPUs utilized          [100.00%]
               323 context-switches          #    0.032 K/sec                  [100.00%]
                17 cpu-migrations            #    0.002 K/sec                  [100.00%]
               233 page-faults               #    0.023 K/sec
   <not supported> cycles
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
   <not supported> instructions
   <not supported> branches
   <not supported> branch-misses
   <not supported> L1-dcache-loads
   <not supported> L1-dcache-load-misses
   <not supported> LLC-loads
   <not supported> LLC-load-misses

       5.001607197 seconds time elapsed
```

Guest PMCs

- Without PMCs, %CPU is ambiguous. We can't measure:
  - **Instructions Per Cycle** (IPC)
  - CPU cache hits/misses
  - MMU TLB hits/misses
  - Branch misprediction
  - Stall cycles
- Should be fixable: hypervisors can expose PMCs
  - At the very least, enough PMCs for IPC to work: INST_RETIRED.ANY_P & CPU_CLK_UNHALTED.THREAD_P
- In the meantime
  - I'm using a physical server for PMC analysis
  - Also some MSRs on the cloud

MSRs

- Model Specific Registers (MSRs) may be exposed when PMCs are not
- Better than nothing. Can solve some issues.

```
# ./showboost
CPU MHz      : 2500
Turbo MHz    : 2900 (10 active)
Turbo Ratio  : 116% (10 active)
CPU 0 summary every 5 seconds...

TIME       C0_MCYC      C0_ACYC        UTIL  RATIO    MHz
17:28:03   4226511637   4902783333      33%   116%   2900
17:28:08   4397892841   5101713941      35%   116%   2900
17:28:13   4550831380   5279462058      36%   116%   2900
17:28:18   4680962051   5429605341      37%   115%   2899
17:28:23   4782942155   5547813280      38%   115%   2899
[...]
```

  – showboost is from my msr-cloud-tools collection (on github)

Gotcha #4 Instruction Profiling

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

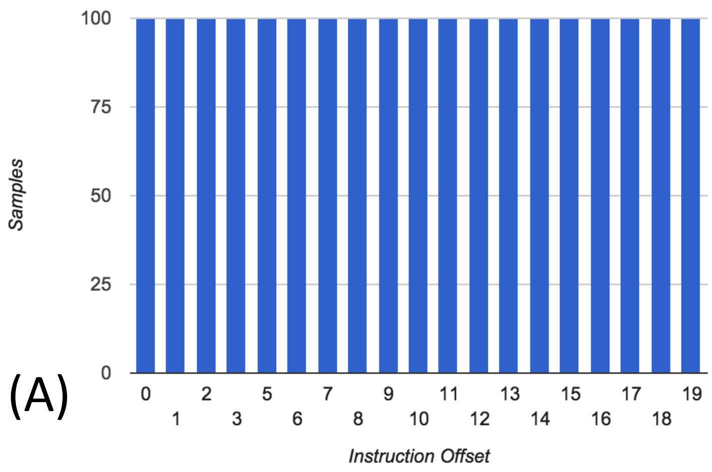
16 NOPs in a loop

Let's profile instructions to see which are hot!

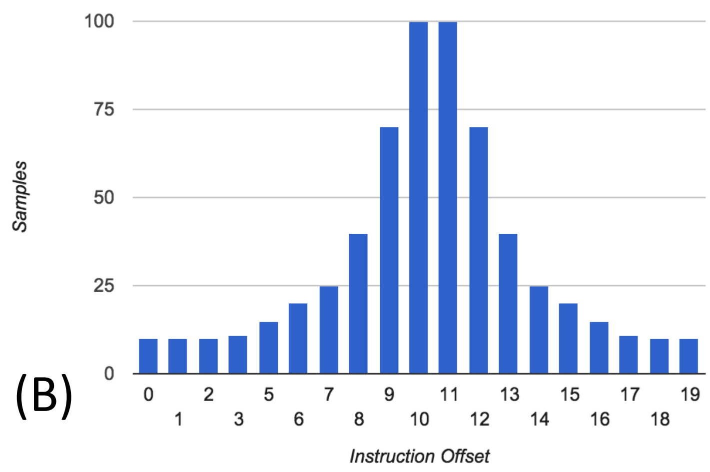
(have I lost my mind?)

Instruction Profiling

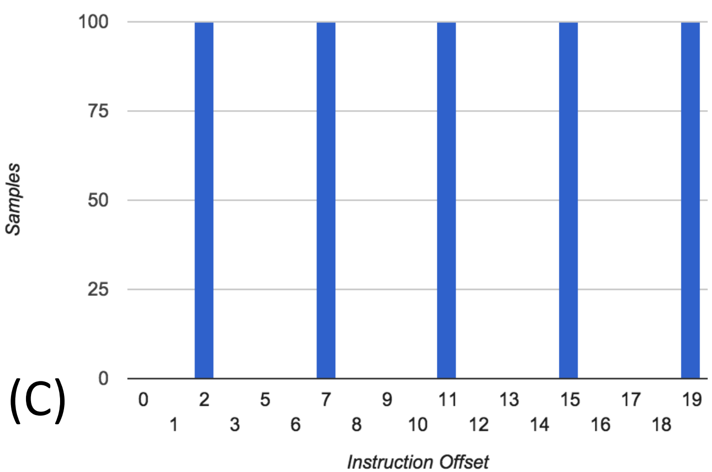
- Even distribution (A)? Or something else?



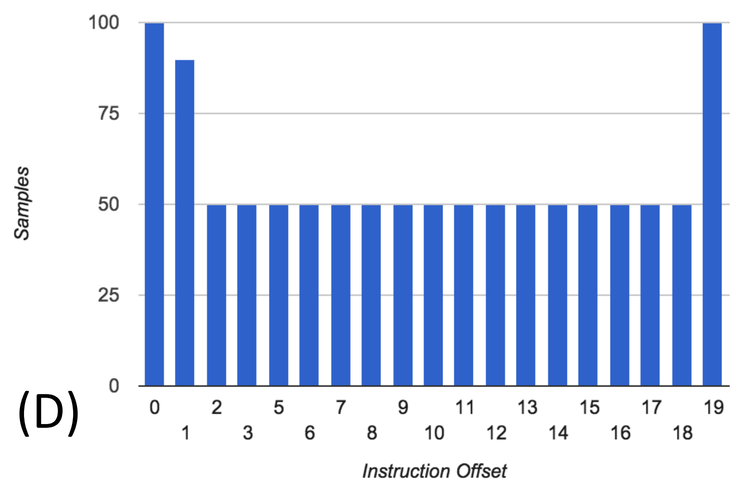
(A)



(B)



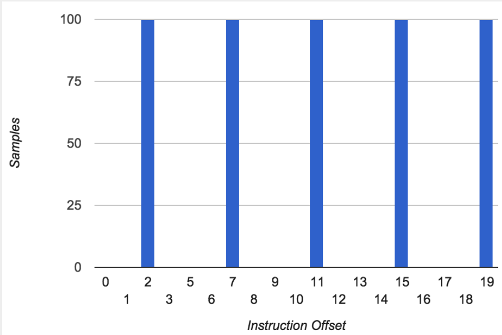
(C)



(D)

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>
```



Go home instruction pointer, you're drunk

PEBS

- I believe this is sample "skid", plus parallel and out-of-order execution of micro-ops: the sampled IP is the resumption instruction, not what is currently executing.
- PEBS may help: Intel's Precise Event Based Sampling
- perf_events has support:
  - `perf record -e cycles:pp`
  - The 'p' can be specified multiple times:
    - 0 - SAMPLE_IP can have arbitrary skid
    - 1 - SAMPLE_IP must have constant skid
    - 2 - SAMPLE_IP requested to have 0 skid
    - 3 - SAMPLE_IP must have 0 skid
  - … from tools/perf/Documentation/perf-list.txt

Gotcha #5 Overhead

- Overhead is relative to the rate of events instrumented
- `perf stat` does in-kernel counts, with relatively low CPU overhead
- `perf record` writes perf.data, which has slightly higher CPU overhead, plus file system and disk I/O
- Test before use
  - In the lab
  - Run perf stat to understand rate, before perf record
- Also consider `--filter`, to filter events in-kernel

5. Tracing

Profiling vs Tracing

- Profiling takes samples. Tracing records every event.
- There are many tracers for Linux (SystemTap, ktap, etc), but only two in mainline: perf_events and ftrace

Linux Tracing Stack

one-liners:	*many*
front-end tools:	`perf`, perf-tools
tracing frameworks:	perf_events, ftrace, eBPF, …
tracing instrumentation:	tracepoints, kprobes, uprobes

Tracing Example

```
# perf record -e block:block_rq_insert -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.172 MB perf.data (~7527 samples) ]

# perf script
[…]
  java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 () 4783360 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783448 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783536 + 24 [java]
  java 9940 [000] 1199510.065195: block_rq_insert: 202,1 R 0 () 4864088 + 88 [java]
[…]
```

Tracing Example

```
# perf record -e block:block_rq_insert -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.172 MB perf.data (~7527 samples) ]

# perf script
[…]
  java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 () 4783360 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783448 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783536 + 24 [java]
  java 9940 [000] 1199510.065195: block_rq_insert: 202,1 R 0 () 4864088 + 88 [java]
[…]
```

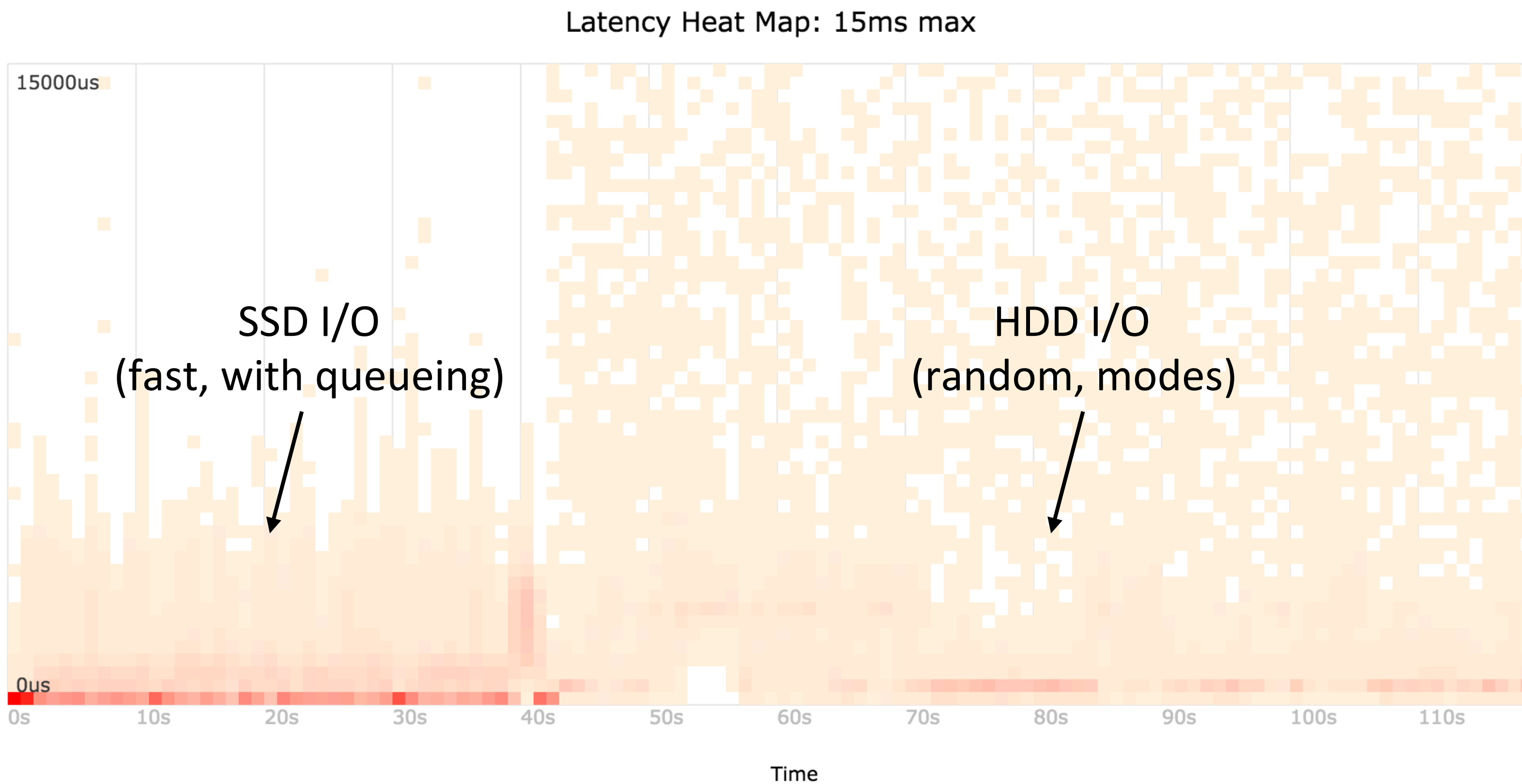
process PID [CPU] timestamp: eventname: format string

```
include/trace/events/block.h: java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 ()
4783360 + 88 [java]
DECLARE_EVENT_CLASS(block_rq,
[...]
 TP_printk("%d,%d %s %u (%s) %llu + %u [%s]",
                 MAJOR(__entry->dev), MINOR(__entry->dev),
                 __entry->rwbs, __entry->bytes, __get_str(cmd),
                 (unsigned long long)__entry->sector,
                 __entry->nr_sector, __entry->comm)
```

kernel source may be the only docs

perf Block I/O Latency Heat Map

- Trace data may only make sense when visualized
- e.g., block I/O latency from perf_events trace data:



http://www.brendangregg.com/blog/2014-07-01/perf-heat-maps.html

One-Liners: Static Tracing

```
# Trace new processes, until Ctrl-C:
perf record -e sched:sched_process_exec -a

# Trace all context-switches with stack traces, for 1 second:
perf record -e context-switches –ag -- sleep 1

# Trace CPU migrations, for 10 seconds:
perf record -e migrations -a -- sleep 10

# Trace all connect()s with stack traces (outbound connections), until Ctrl-C:
perf record -e syscalls:sys_enter_connect –ag

# Trace all block device (disk I/O) requests with stack traces, until Ctrl-C:
perf record -e block:block_rq_insert -ag

# Trace all block device issues and completions (has timestamps), until Ctrl-C:
perf record -e block:block_rq_issue -e block:block_rq_complete -a

# Trace all block completions, of size at least 100 Kbytes, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'nr_sector > 200'

# Trace all block completions, synchronous writes only, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'rwbs == "WS"'

# Trace all block completions, all types of writes, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'rwbs ~ "*W*"'

# Trace all ext4 calls, and write to a non-ext4 location, until Ctrl-C:
perf record -e 'ext4:*' -o /tmp/perf.data -a
```

Tracepoint Variables

- Some previous one-liners used variables with --filter
- The ftrace interface has a way to print them:

```
# cat /sys/kernel/debug/tracing/events/block/block_rq_insert/format
name: block_rq_insert
ID: 884
format:
	field:unsigned short common_type;	offset:0; size:2;	signed:0;
	field:unsigned char common_flags;	offset:2; size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3; size:1;	signed:0;
	field:int common_pid;	offset:4; size:4;	signed:1;

	field:dev_t dev;	offset:8; size:4;	signed:0;
	field:sector_t sector;	offset:16;size:8;	signed:0;
	field:unsigned int nr_sector;	offset:24;size:4;	signed:0;
	field:unsigned int bytes;offset:28;size:4;	signed:0;
	field:char rwbs[8];	offset:32;size:8;	signed:1;
	field:char comm[16];offset:40;size:16;	signed:1;
	field:__data_loc char[] cmd;	offset:56;size:4;	signed:1;

print fmt: "%d,%d %s %u (%s) %llu + %u [%s]", ((unsigned int) ((REC->dev) >> 20)),
((unsigned int) ((REC->dev) & ((1U << 20) - 1))), REC->rwbs, REC->bytes, __get_str(cmd),
(unsigned long long)REC->sector, REC->nr_sector, REC->comm
```

variables

(format string internals)

One-Liners: Dynamic Tracing

```
# Add a tracepoint for the kernel tcp_sendmsg() function entry (--add optional):
perf probe --add tcp_sendmsg

# Remove the tcp_sendmsg() tracepoint (or use --del):
perf probe -d tcp_sendmsg

# Add a tracepoint for the kernel tcp_sendmsg() function return:
perf probe 'tcp_sendmsg%return'

# Show avail vars for the tcp_sendmsg(), plus external vars (needs debuginfo):
perf probe -V tcp_sendmsg --externs

# Show available line probes for tcp_sendmsg() (needs debuginfo):
perf probe -L tcp_sendmsg

# Add a tracepoint for tcp_sendmsg() line 81 with local var seglen (debuginfo):
perf probe 'tcp_sendmsg:81 seglen'

# Add a tracepoint for do_sys_open() with the filename as a string (debuginfo):
perf probe 'do_sys_open filename:string'

# Add a tracepoint for myfunc() return, and include the retval as a string:
perf probe 'myfunc%return +0($retval):string'

# Add a tracepoint for the user-level malloc() function from libc:
perf probe -x /lib64/libc.so.6 malloc

# List currently available dynamic probes:
perf probe -l
```

One-Liners: Advanced Dynamic Tracing

```
# Add a tracepoint for tcp_sendmsg(), with three entry regs (platform specific):
perf probe 'tcp_sendmsg %ax %dx %cx'

# Add a tracepoint for tcp_sendmsg(), with an alias ("bytes") for %cx register:
perf probe 'tcp_sendmsg bytes=%cx'

# Trace previously created probe when the bytes (alias) var is greater than 100:
perf record -e probe:tcp_sendmsg --filter 'bytes > 100'

# Add a tracepoint for tcp_sendmsg() return, and capture the return value:
perf probe 'tcp_sendmsg%return $retval'

# Add a tracepoint for tcp_sendmsg(), and "size" entry argument (debuginfo):
perf probe 'tcp_sendmsg size'

# Add a tracepoint for tcp_sendmsg(), with size and socket state (debuginfo):
perf probe 'tcp_sendmsg size sk->__sk_common.skc_state'

# Trace previous probe when size > 0, and state != TCP_ESTABLISHED(1) (debuginfo):
perf record -e probe:tcp_sendmsg --filter 'size > 0 && skc_state != 1' -a
```

- Kernel debuginfo is an onerous requirement for the Netflix cloud
- We can use registers instead (as above). But which registers?

The Rosetta Stone of Registers

One server instance with kernel debuginfo, and -nv (dry run, verbose):

```
# perf probe -nv 'tcp_sendmsg size sk->__sk_common.skc_state'
[…]
Added new event:
Writing event: p:probe/tcp_sendmsg tcp_sendmsg+0 size=%cx:u64 skc_state=+18(%si):u8
  probe:tcp_sendmsg (on tcp_sendmsg with size skc_state=sk->__sk_common.skc_state)

You can now use it in all perf tools, such as:

     perf record -e probe:tcp_sendmsg -aR sleep 1
```

Copy-n-paste!

All other instances (of the same kernel version):

```
# perf probe 'tcp_sendmsg+0 size=%cx:u64 skc_state=+18(%si):u8'
Failed to find path of kernel module.
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with size=%cx:u64 skc_state=+18(%si):u8)

You can now use it in all perf tools, such as:

     perf record -e probe:tcp_sendmsg -aR sleep 1
```

Masami Hiramatsu was investigating a way to better automate this

perf_events Scripting

- perf also has a scripting interface (Perl or Python)
  - These run perf and post-process trace data in user-space
  - Data path has some optimizations
    - Kernel buffering, and dynamic (optimal) number of buffer reads
    - But may still not be enough for high volume events
- Andi Kleen has scripted perf for his PMC tools
  - https://github.com/andikleen/pmu-tools
  - Includes toplev for applying "Top Down" methodology
- I've developed my own tools for perf & ftrace
  - https://github.com/brendangregg/perf-tools
  - Each tool has a man page and examples file
  - These are unsupported temporary hacks; their internals should be rewritten when kernel features arrive (e.g., eBPF)

perf-tools: bitesize

- Block (disk) I/O size distributions:

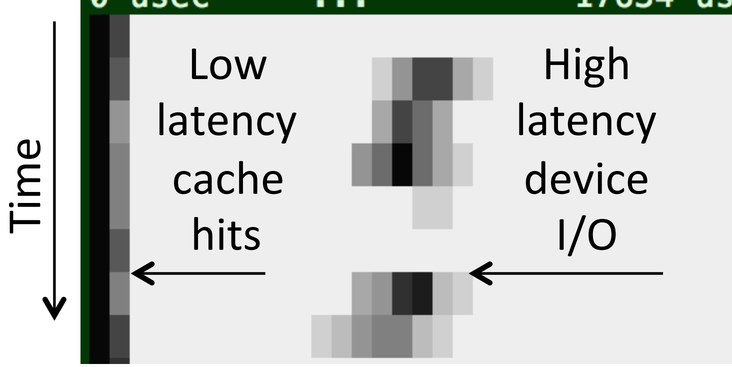
```
# ./bitesize
Tracing block I/O size (bytes), until Ctrl-C...
^C
        Kbytes          : I/O       Distribution
          -> 0.9        : 0         |                                      |
      1.0 -> 7.9        : 38        |#                                     |
      8.0 -> 63.9       : 10108     |######################################|
     64.0 -> 127.9      : 13        |#                                     |
    128.0 ->            : 1         |#                                     |
[…]
```

- This automates perf with a set of in-kernel filters and counts for each bucket, to reduce overhead
- Will be much easier with eBPF

eBPF

- Extended BPF: programs on tracepoints
  - High-performance filtering: JIT
  - In-kernel summaries: maps
- e.g., in-kernel latency heat map (showing bimodal):

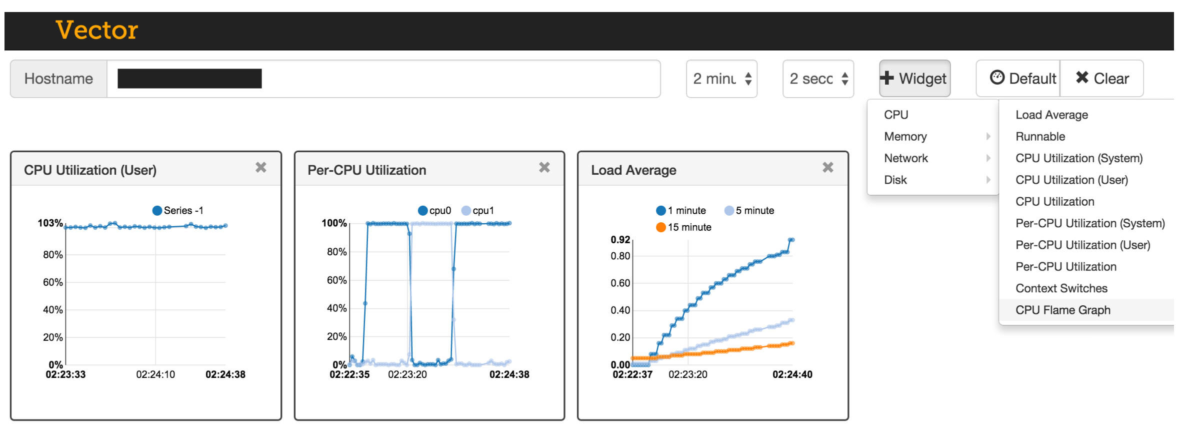
```
root@bgregg-test-i-b7874e9d:/mnt/src/linux-3.16bpf2/samples/bpf# ./ex3
writing bpf-7 -> /sys/kernel/debug/tracing/events/block/block_rq_issue/filter
writing bpf-9 -> /sys/kernel/debug/tracing/events/block/block_rq_complete/filter
waiting for events to determine average latency...
  IO latency in usec
    - many events with this latency
    - few events
0 usec     ...          17634 usec
```




```
captured=270 missed=0 max_lat=0 usec
captured=3694 missed=0 max_lat=0 usec
captured=3485 missed=12 max_lat=18902 usec
captured=3541 missed=19 max_lat=82377 usec
captured=1945 missed=33 max_lat=24441 usec
captured=1636 missed=0 max_lat=0 usec
captured=3441 missed=18 max_lat=51263 usec
captured=2864 missed=71 max_lat=60497 usec
```

Linux Profiling Future

- eBPF is integrating, and provides the final missing piece of tracing infrastructure: efficient kernel programming
  - perf_events + eBPF?
  - ftrace + eBPF?
  - Other tracers + eBPF?
- At Netflix, the future is Vector, and more self-service automation of perf_events



Summary & Your Action Items

- Short term: **get full CPU profiling to work**
  A. Automate perf CPU profiles with flame graphs. See this talk!
  B. … or use Netflix Vector when it is open sourced
  C. … or ask performance monitoring vendors for this
  – Most importantly, you should *expect* that full CPU profiles are available at your company. The ROI is worth it.
- Long term: **PMCs & tracing**
  – Use perf_events to profile other targets: CPU cycles, file system I/O, disk I/O, memory usage, …
- Go forth and profile!

The "real" checklist reminder:

1. Get "perf" to work
2. Get stack walking to work
3. Fix symbol translation
4. Get IPC to work
5. Test perf under load

Links & References

- perf_events
  - Kernel source: **tools/perf/Documentation**
  - https://perf.wiki.kernel.org/index.php/Main_Page
  - http://www.brendangregg.com/perf.html
  - http://web.eece.maine.edu/~vweaver/projects/perf_events/
  - **Mailing list** http://vger.kernel.org/vger-lists.html#linux-perf-users
- perf-tools: https://github.com/brendangregg/perf-tools
- PMU tools: https://github.com/andikleen/pmu-tools
- perf, ftrace, and more: http://www.brendangregg.com/linuxperf.html
- Java frame pointer patch
  - http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2014-December/016477.html
  - https://bugs.openjdk.java.net/browse/JDK-8068945
- Node.js: http://techblog.netflix.com/2014/11/nodejs-in-flames.html
- Methodology: http://www.brendangregg.com/methodology.html
- Flame graphs: http://www.brendangregg.com/flamegraphs.html
- Heat maps: http://www.brendangregg.com/heatmaps.html
- eBPF: http://lwn.net/Articles/603983/

Thanks

- Questions?
- http://slideshare.net/brendangregg
- http://www.brendangregg.com
- bgregg@netflix.com
  - Performance and Reliability Engineering
- @brendangregg

The Thirteenth Annual
Southern California Linux Expo