



# Java Mixed-Mode Flame Graphs

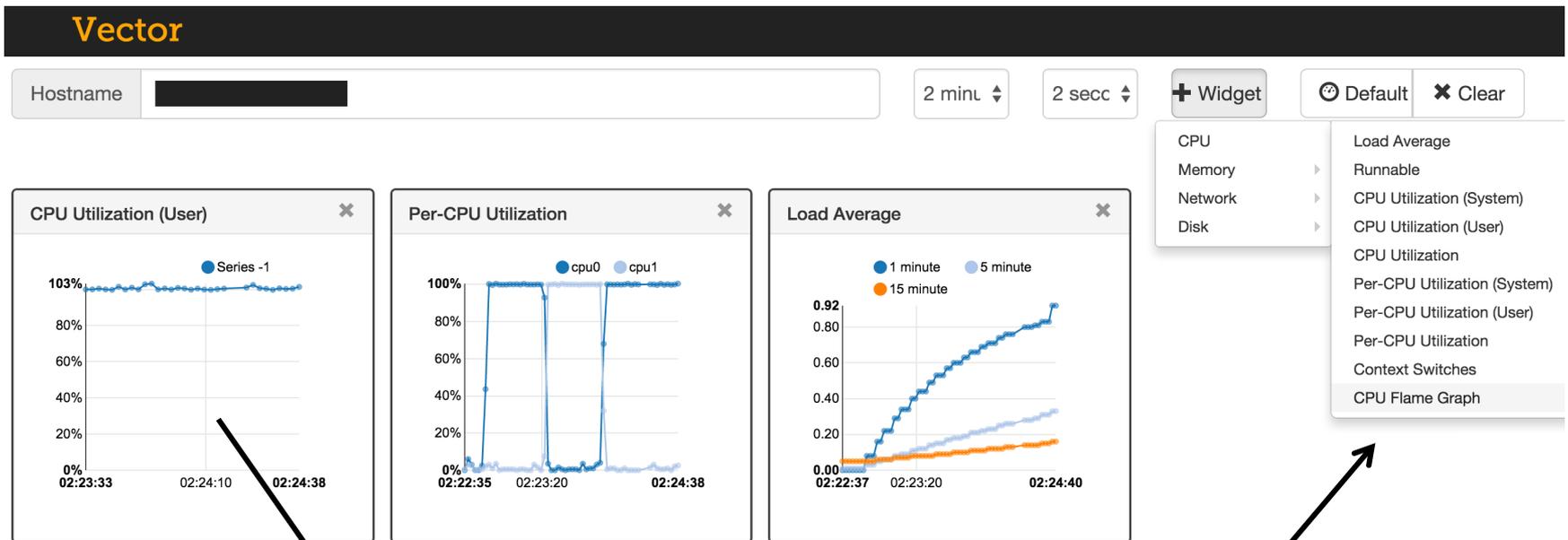
Brendan Gregg Senior Performance Architect



Understanding Java CPU usage  
**quickly and completely**

# Quickly

- Via SSH and open source tools (covered in this talk)
- Or using Netflix Vector GUI (also open source):



1. Observe high CPU usage
2. Generate a flame graph



# 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

- Don't overlook system code: kernel, libraries, etc.

Context

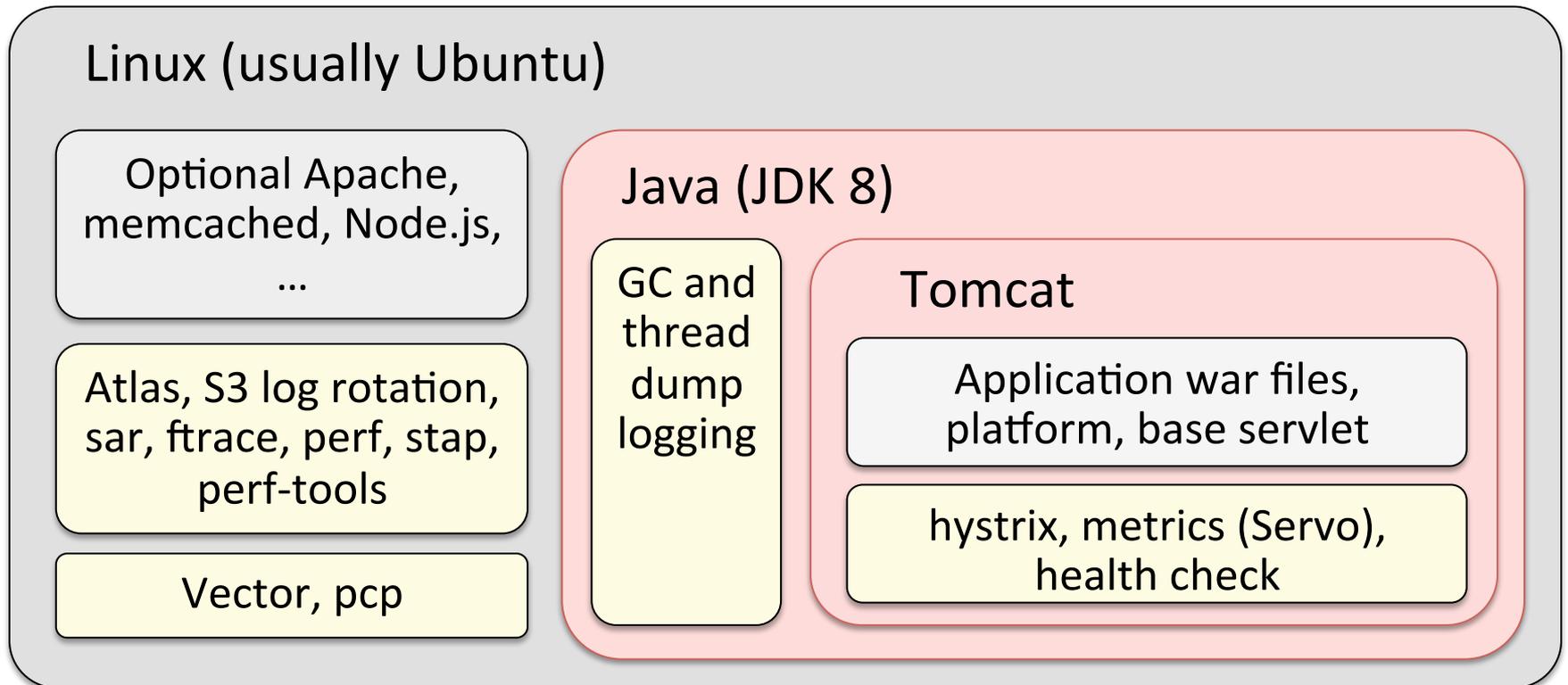
# NETFLIX

- Over 60 million subscribers
  - Just launched in Spain!
- AWS EC2 Linux cloud
- FreeBSD CDN
- Awesome place to work



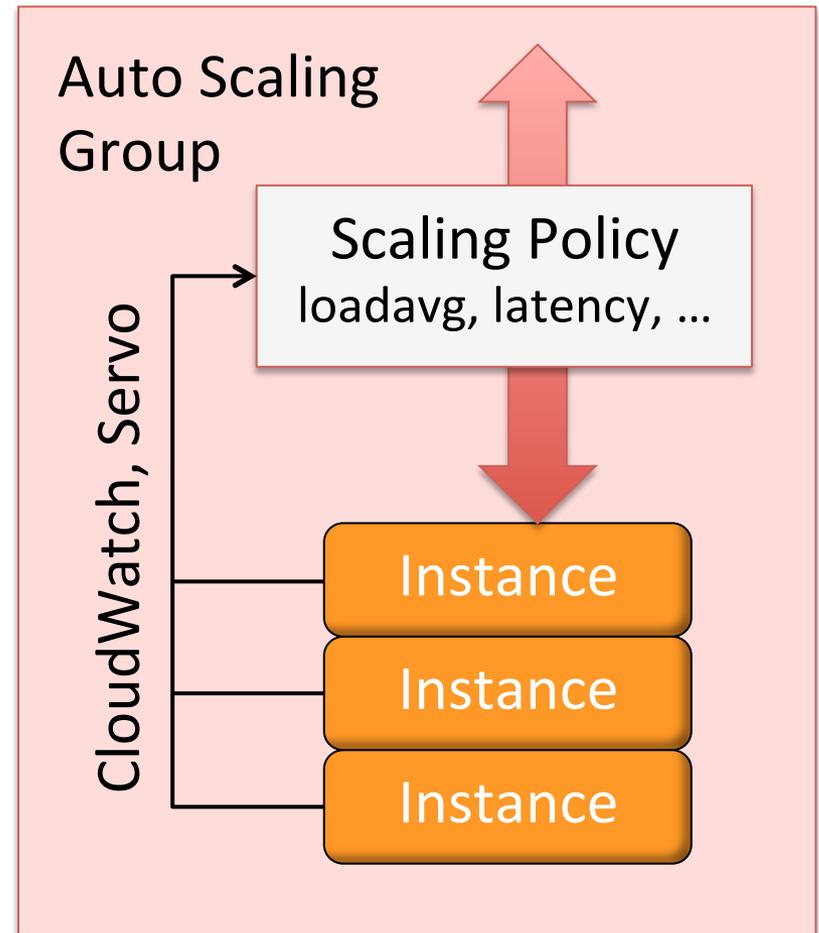
# NETFLIX Cloud

- Tens of thousands of AWS EC2 instances
- Mostly running Java applications (Oracle JVM)



# Why we need CPU profiling

- Improving performance
  - Identify tuning targets
  - Incident response
  - Non-regression testing
  - Software evaluations
  - CPU workload characterization
- Cost savings
  - ASGs often scale on load average (CPU), so CPU usage is proportional to cost

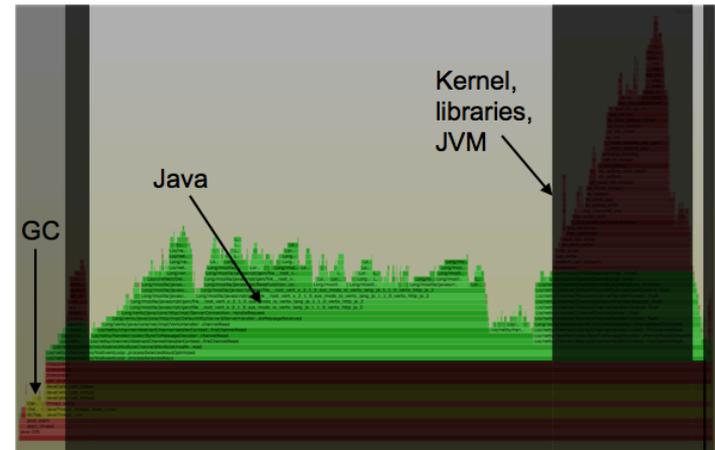


# The Problem with Profilers



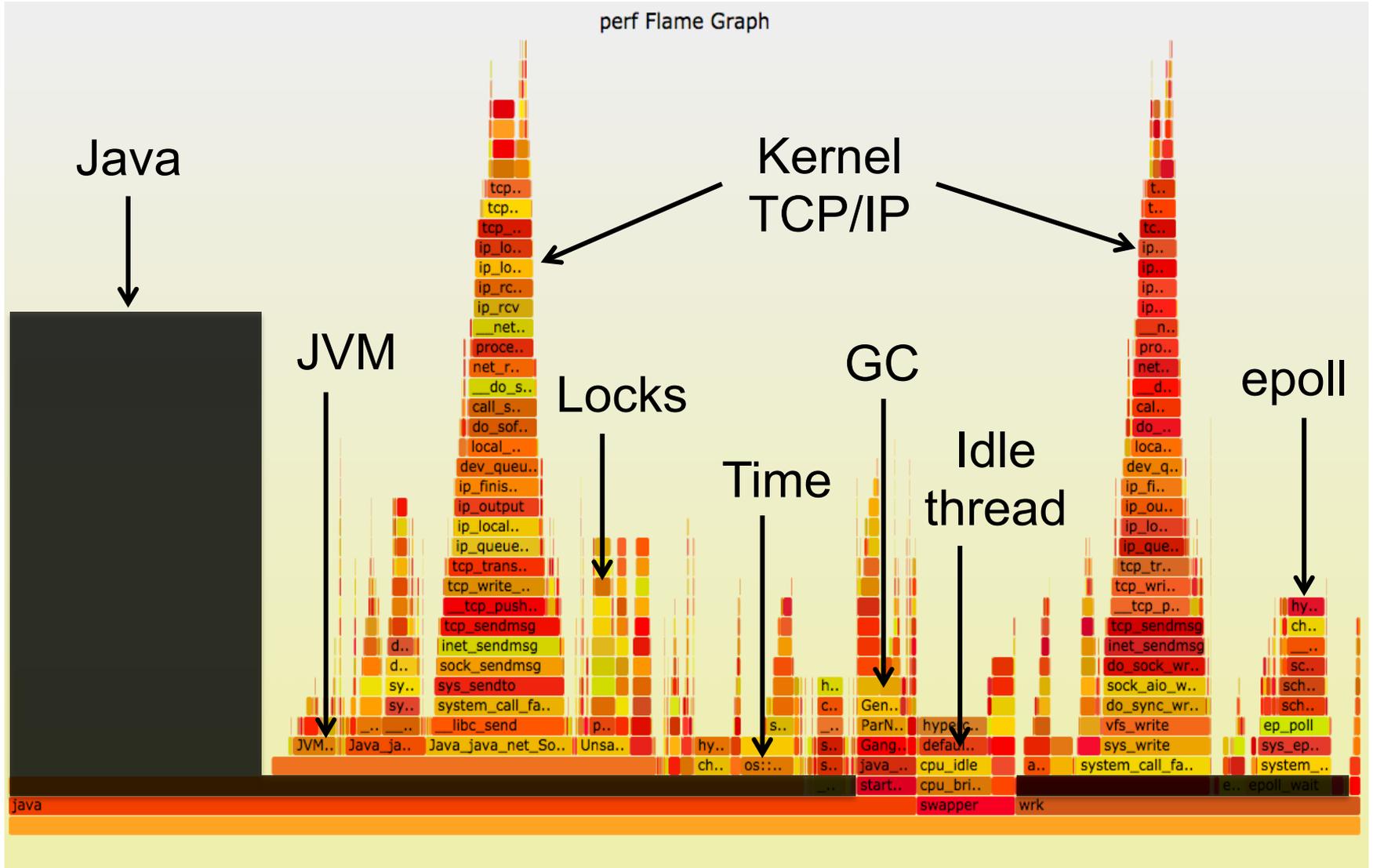
# Java Profilers

- 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



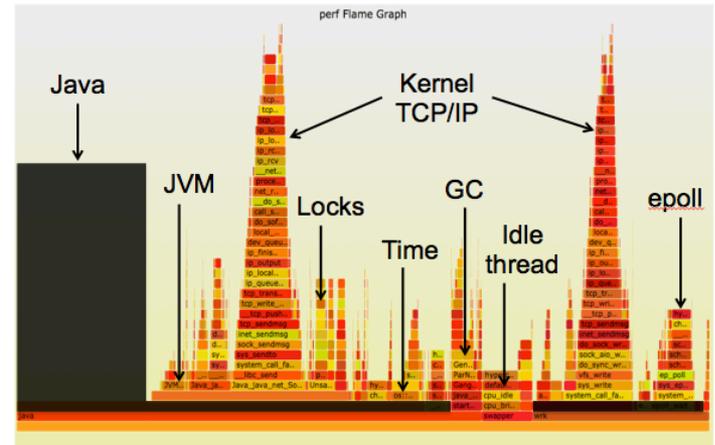
# System Profilers

perf Flame Graph



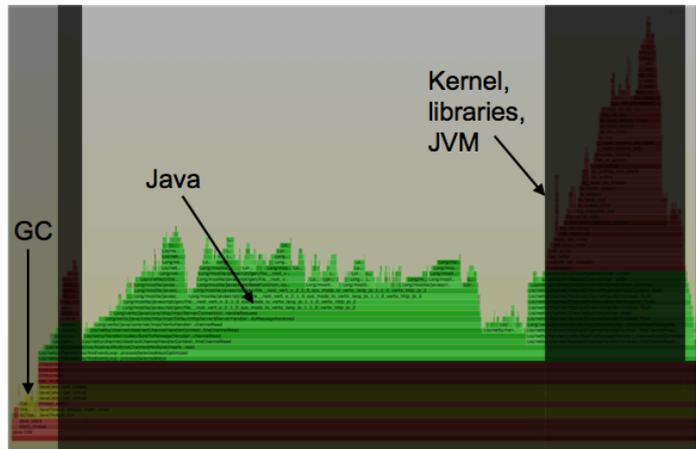
# System Profilers

- Visibility
  - JVM (C++)
  - GC (C++)
  - libraries (C)
  - kernel (C)
- Typical problems (x86):
  - Stacks missing for Java
  - Symbols missing for Java methods
- Other architectures (e.g., SPARC) have fared better
- Profile everything **except Java**

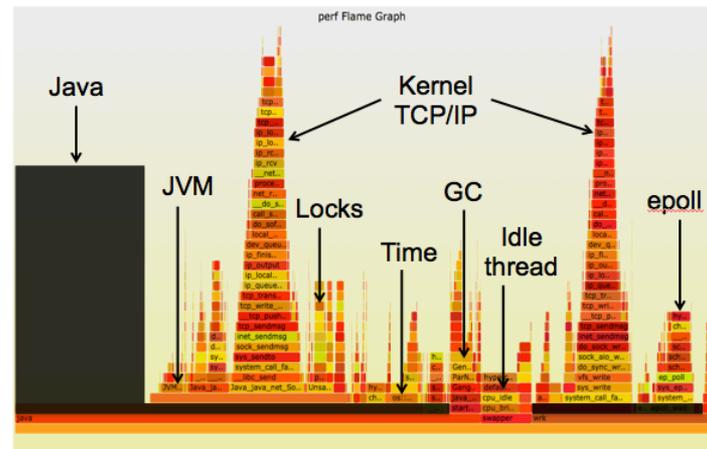


# Workaround

- Capture both Java and system profiles, and examine side by side



Java

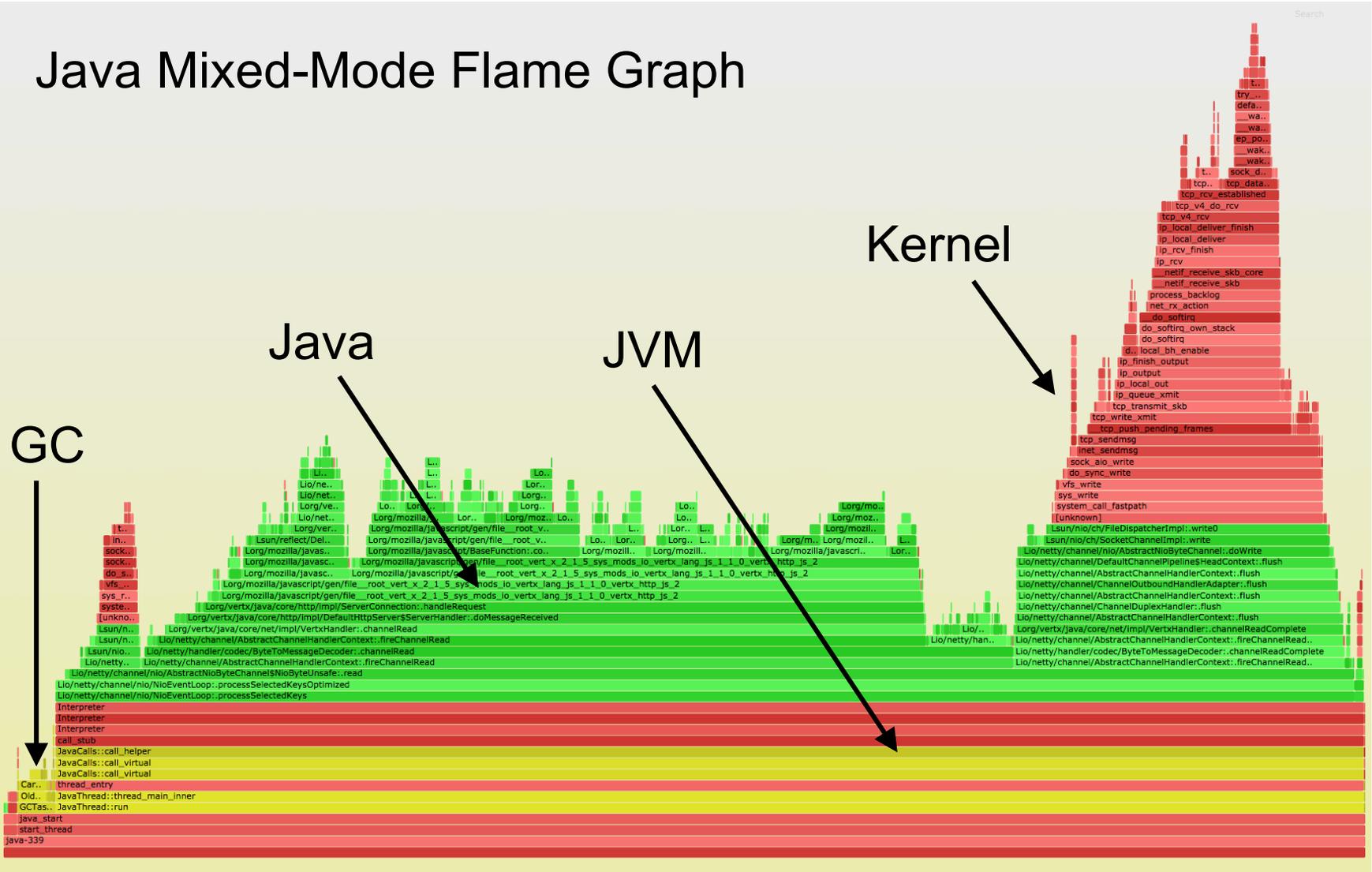


System

- An improvement, but Java context is often crucial for interpreting system profiles

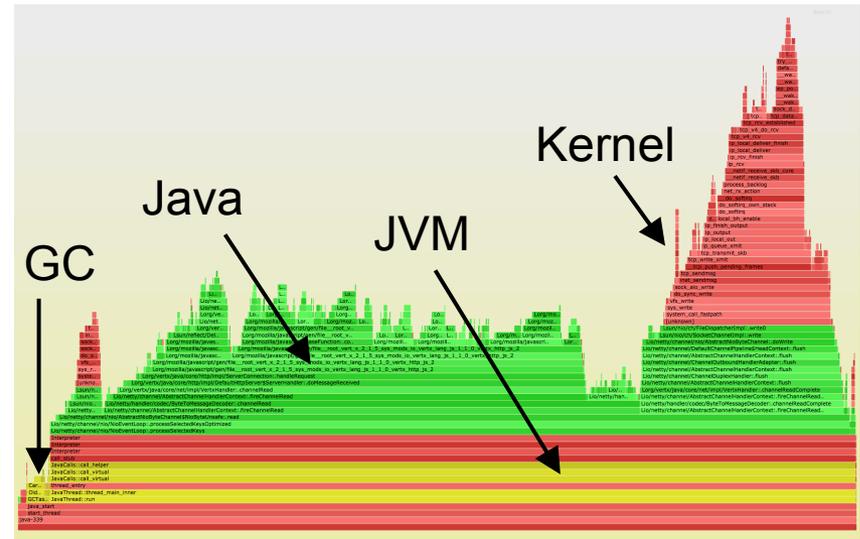
# Solution

## Java Mixed-Mode Flame Graph



# Solution

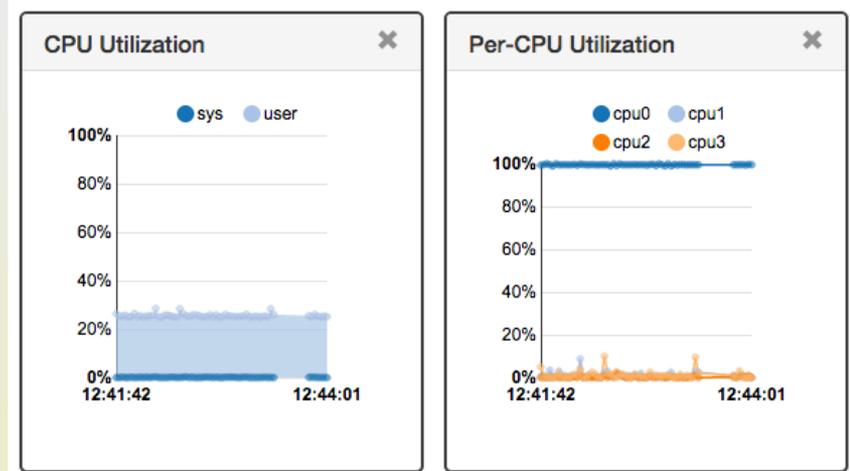
- Fix system profiling
  - Only way to see it all
- Visibility is everything:
  - Java methods
  - JVM (C++)
  - GC (C++)
  - libraries (C)
  - kernel (C)
- Minor Problems:
  - 0-3% CPU overhead to enable frame pointers (usually <1%).
  - Symbol dumps can consume a burst of CPU
- **Complete and accurate** (asynchronous) profiling



# Simple Production Example

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

1. Poor performance, and one CPU at 100%
2. perf\_events flame graph shows JVM stuck compiling



```
PhaseMacroExpand::process_users_of_allocation
PhaseMacroExpand::eliminate_allocate_node
PhaseMacroExpand::eliminate_macro_nodes
PhaseMacroExpand::expand_macro_nodes
Compile::Optimize
Compile::Compile
C2Compiler::compile_method
CompileBroker::invoke_compiler_on_method
CompileBroker::compiler_thread_loop
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java
```



# DEMO

FlameGraph\_tomcat01.svg

# Exonerating The System

- From last week:
  - Frequent thread creation/ destruction assumed to be consuming CPU resources. Recode application?
  - A flame graph quantified this CPU time: near zero
  - Time mostly other Java methods

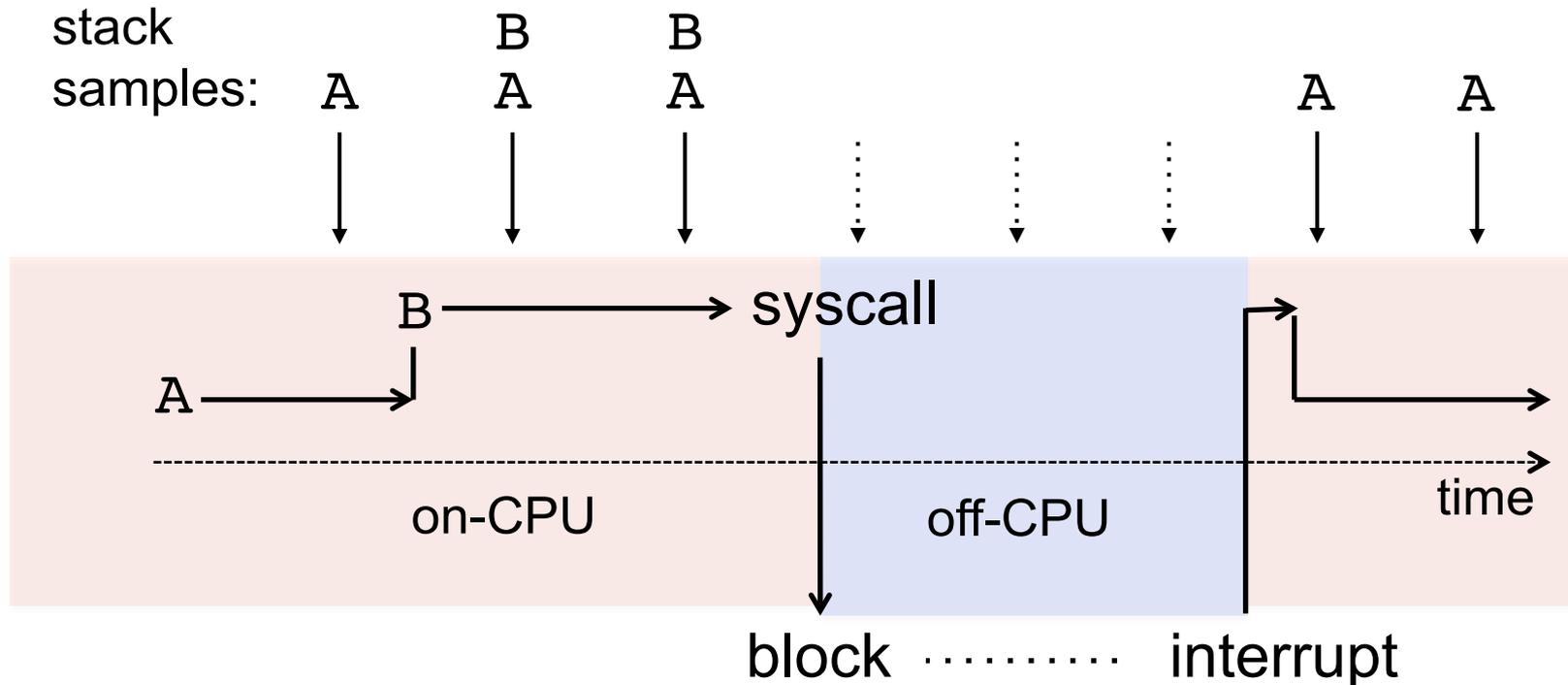




# CPU Profiling

# CPU Profiling

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



# Stack Traces

- A code path snapshot. e.g., from `jstack(1)`:

```
$ jstack 1819
[...]
"main" prio=10 tid=0x00007ff304009000
nid=0x7361 runnable [0x00007ff30d4f9000]
    java.lang.Thread.State: RUNNABLE
        at Func_abc.func_c(Func_abc.java:6)
        at Func_abc.func_b(Func_abc.java:16)
        at Func_abc.func_a(Func_abc.java:23)
        at Func_abc.main(Func_abc.java:27)
```

running  
codepath  
↑  
start

running  
parent  
g.parent  
g.g.paren  
↓

# System Profilers

- Linux
  - perf\_events (aka "perf")
- Oracle Solaris
  - DTrace
- OS X
  - Instruments
- Windows
  - XPerf
- And many others...

# Linux perf\_events

- Standard Linux profiler
  - Provides the `perf` command (multi-tool)
  - Usually pkg added by `linux-tools-common`, etc.
- Features:
  - Timer-based sampling
  - Hardware events
  - Tracepoints
  - Dynamic tracing
- Can sample stacks of (almost) everything on CPU
  - Can miss hard interrupt ISRs, but these should be near-zero. They can be measured if needed (I wrote my own tools)

# perf record Profiling

- Stack profiling on all CPUs at 99 Hertz, then dump:

```
# 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 script
[...]
bash 13204 cpu-clock:
    459c4c dequote_string (/root/bash-4.3/bash)
    465c80 glob_expand_word_list (/root/bash-4.3/bash)
    466569 expand_word_list_internal (/root/bash-4.3/bash)
    465a13 expand_words (/root/bash-4.3/bash)
    43bbf7 execute_simple_command (/root/bash-4.3/bash)
one  435f16 execute_command_internal (/root/bash-4.3/bash)
stack 435580 execute_command (/root/bash-4.3/bash)
sample 43a771 execute_while_or_until (/root/bash-4.3/bash)
       43a636 execute_while_command (/root/bash-4.3/bash)
       → 436129 execute_command_internal (/root/bash-4.3/bash)
       435580 execute_command (/root/bash-4.3/bash)
       420cd5 reader_loop (/root/bash-4.3/bash)
       41ea58 main (/root/bash-4.3/bash)
       7ff2294edec5 __libc_start_main (/lib/x86_64-linux-gnu/libc-2.19.so)
[... ~47,000 lines truncated ...]
```

# perf report Summary

- Generates a call tree and combines samples:

```
# 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 ...]*

call tree summary

# Flame Graphs

# perf report Verbosity

- Despite summarizing, output is still verbose

```
# perf report -n -stdio
[...]
```

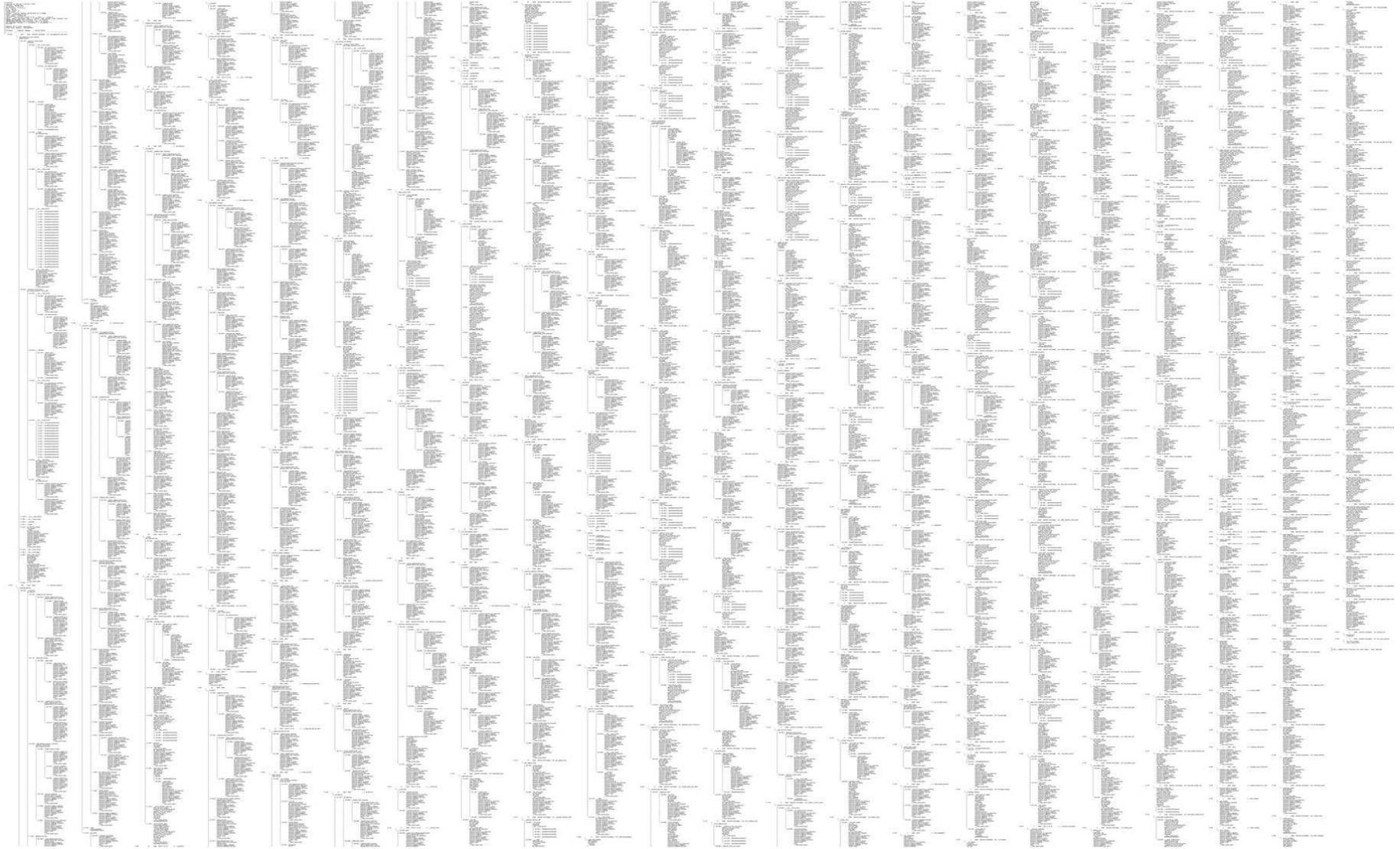
#	Overhead	Samples	Command	Shared Object	Symbol
#	.....	.....	.....	.....	.....
#					
	20.42%	605	bash	[kernel.kallsyms]	[k] xen_hypercall_xen_version

```
|
--- 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
```

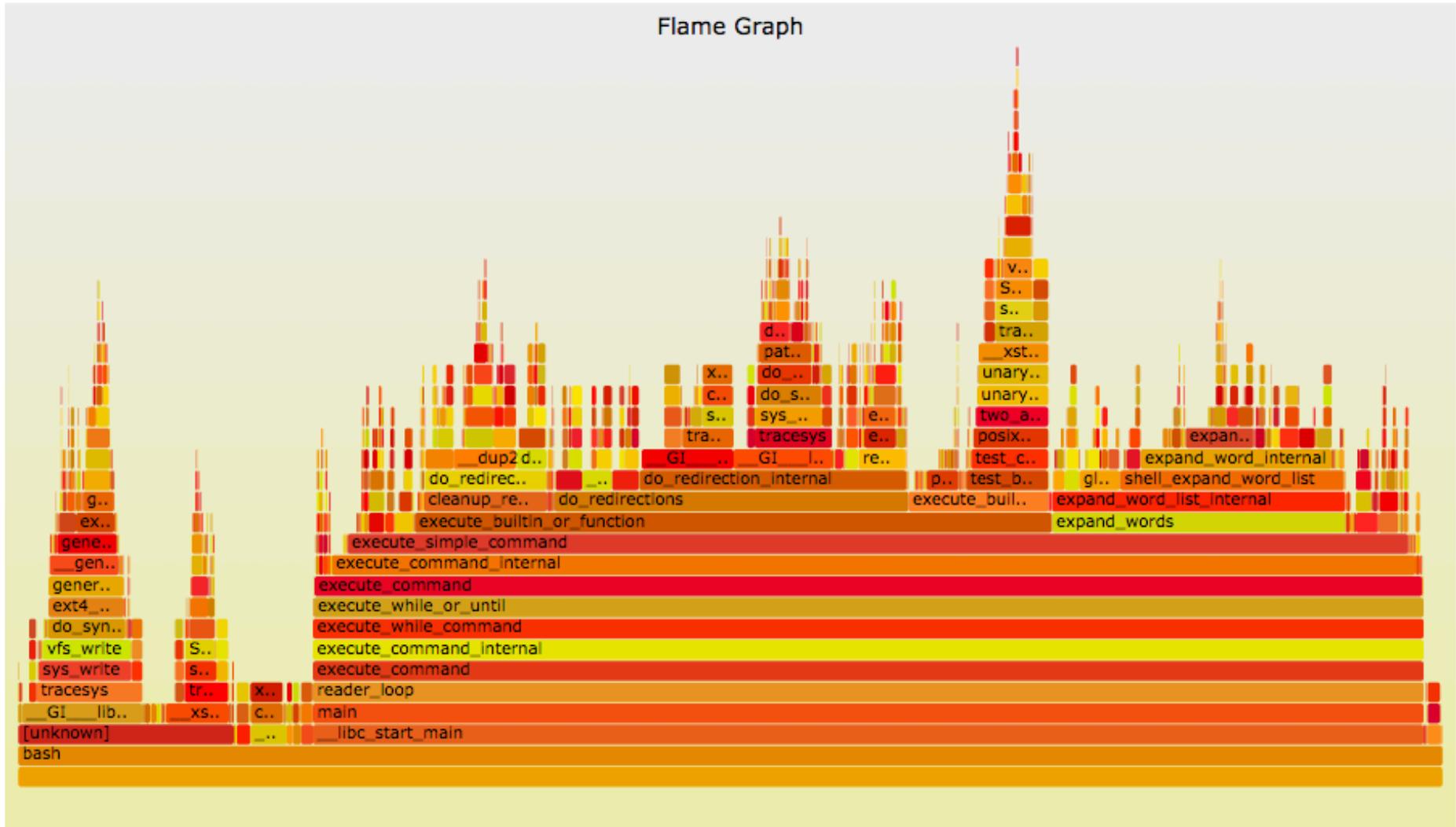
↓

```
[... ~13,000 lines truncated ...]
```

# Full perf report Output



# ... as a Flame Graph

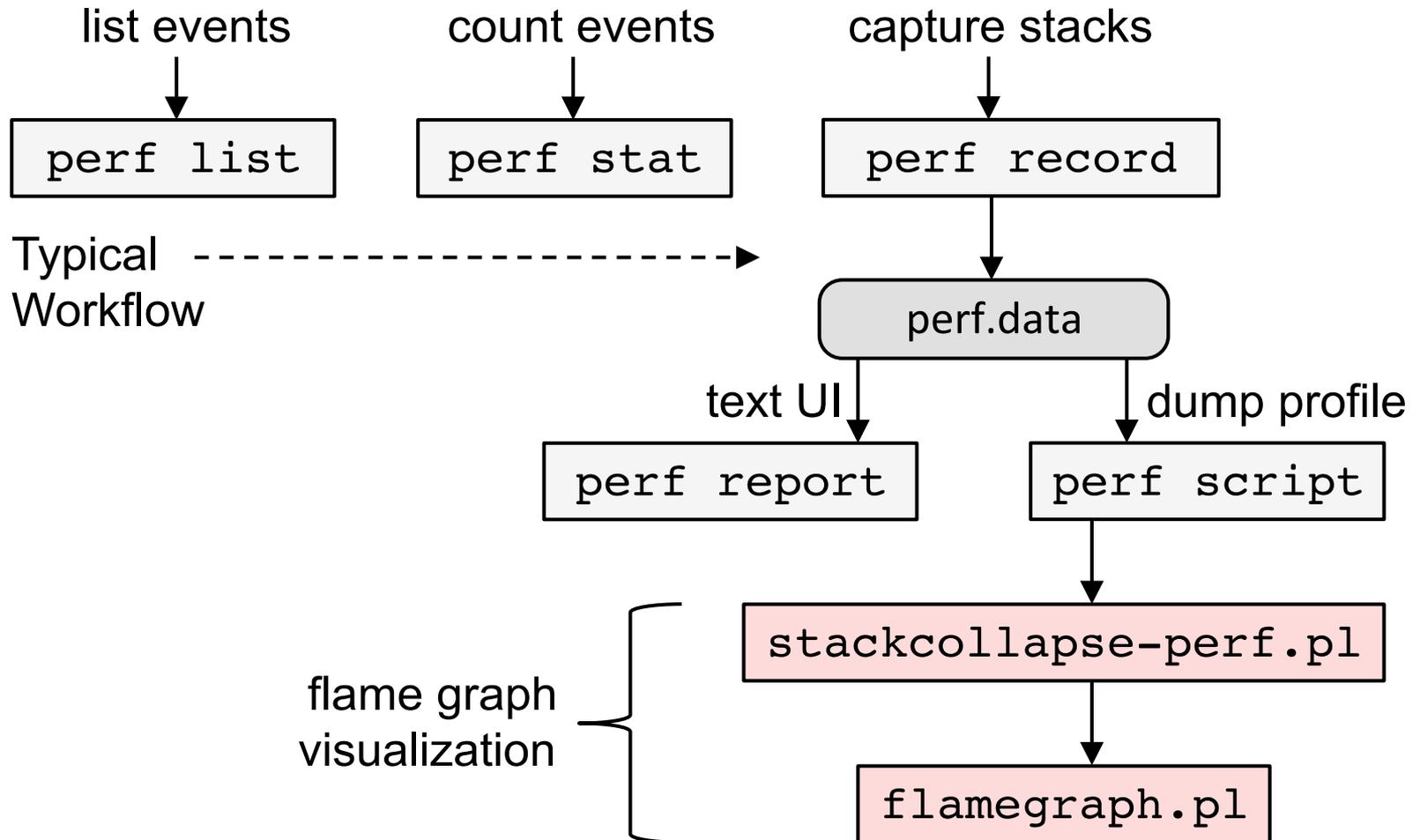


# Flame Graphs

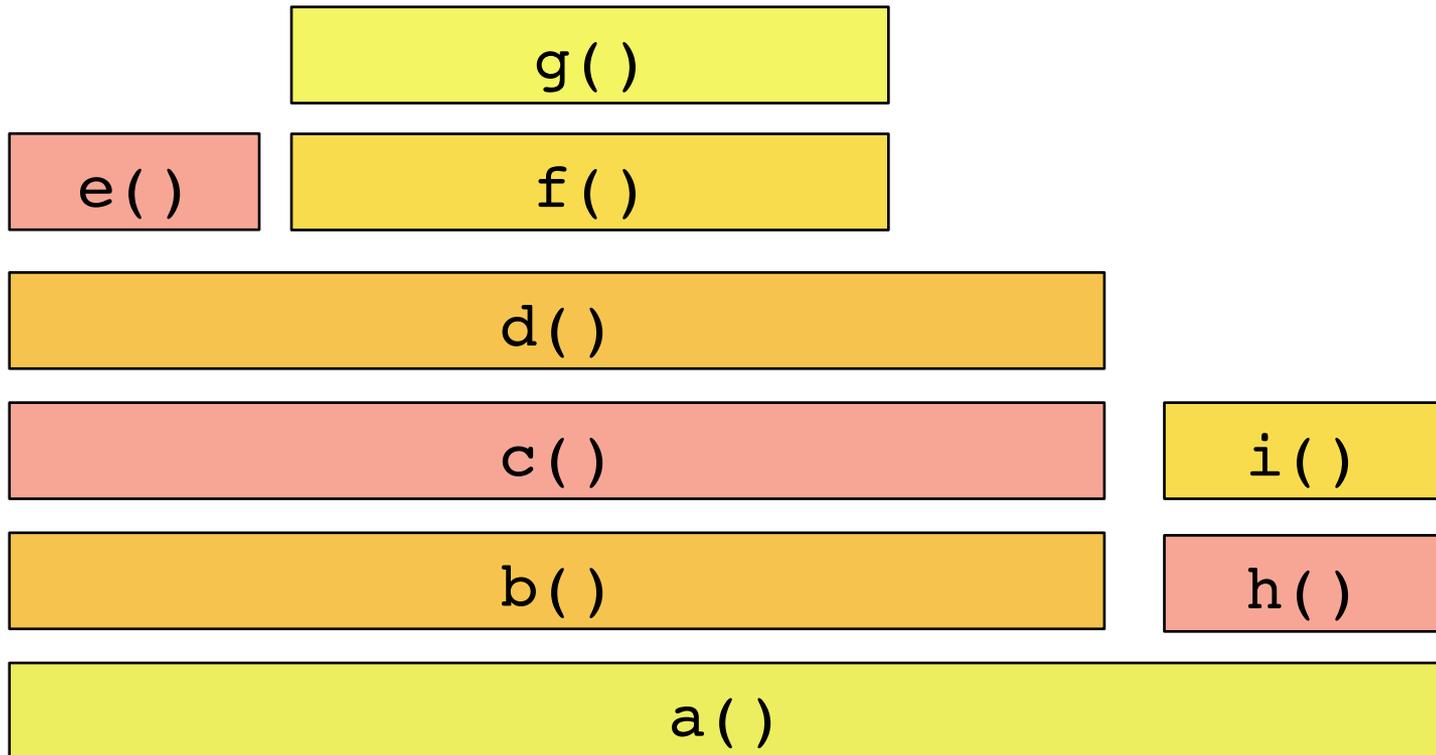
```
git clone --depth 1 https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record -F 99 -a -g -- sleep 30
perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg
```

- Flame Graphs:
  - **x-axis**: alphabetical stack sort, to maximize merging
  - **y-axis**: stack depth
  - **color**: random (default), or a dimension
- Currently made from Perl + SVG + JavaScript
  - Multiple d3 versions are being developed
- Easy to get working
  - <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>
  - Above commands are Linux; see URL for other OSes

# Linux perf\_events Workflow

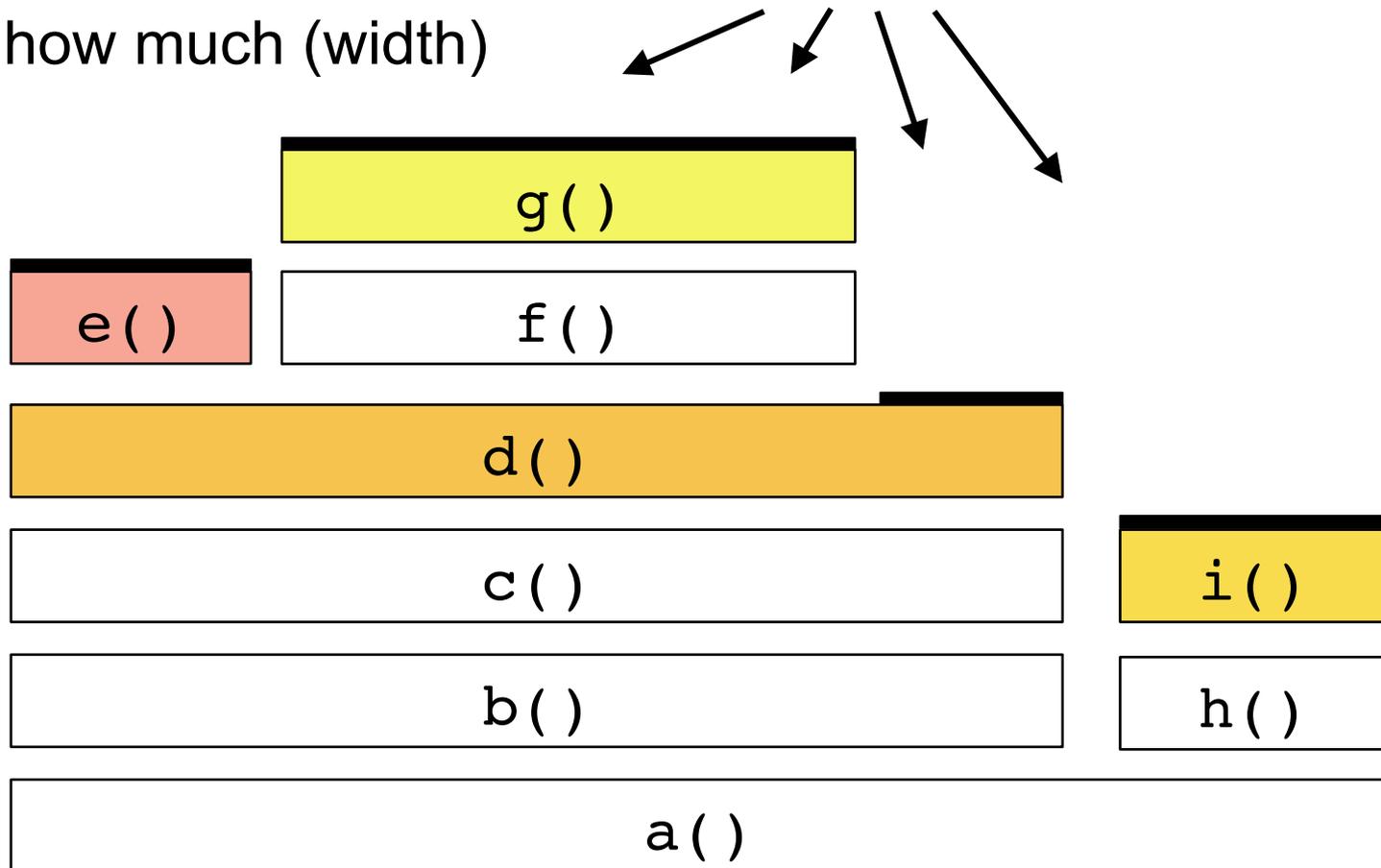


# Flame Graph Interpretation



# Flame Graph Interpretation (1/3)

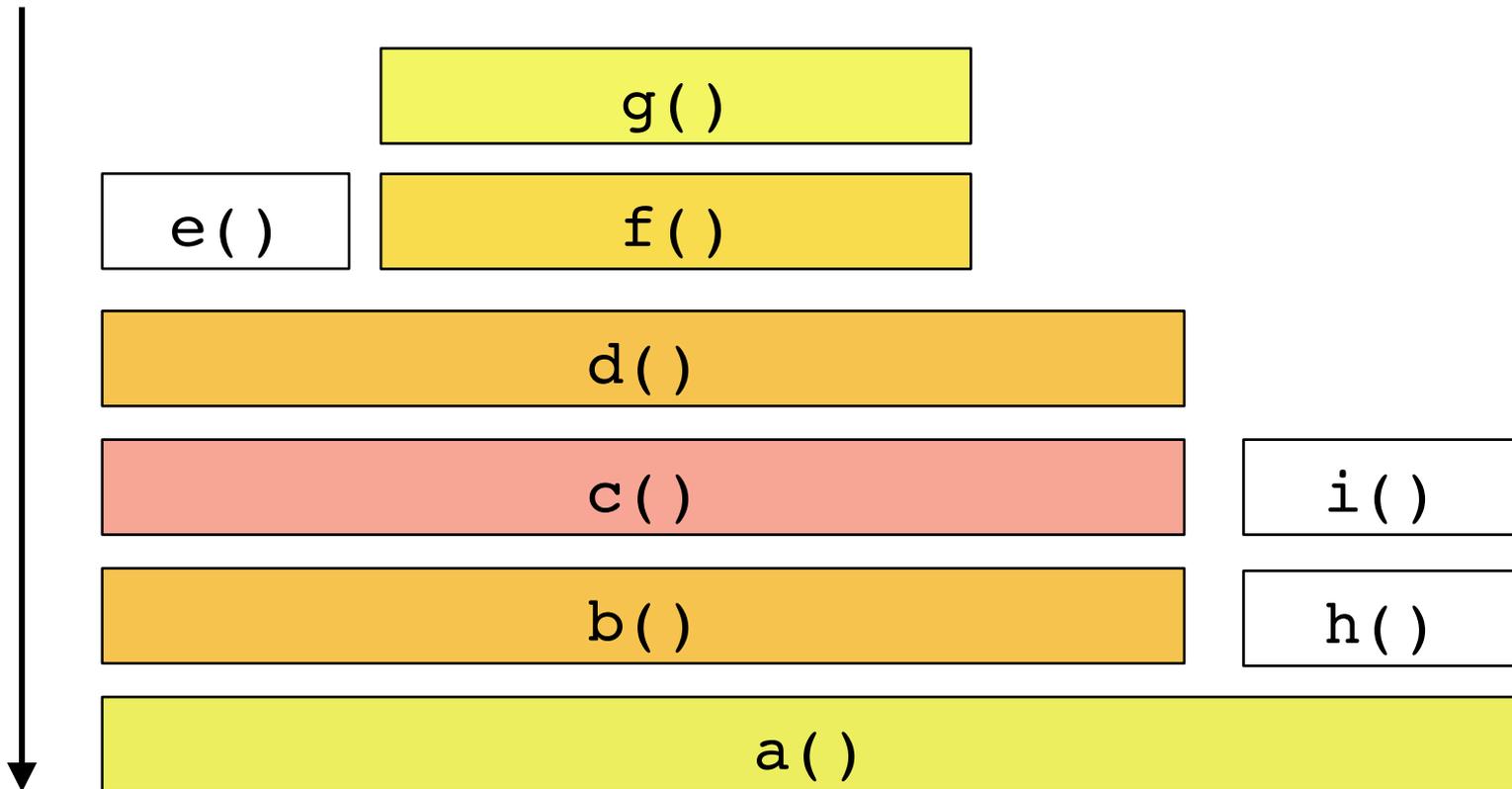
Top edge shows who is running on-CPU,  
and how much (width)



# Flame Graph Interpretation (2/3)

Top-down shows ancestry

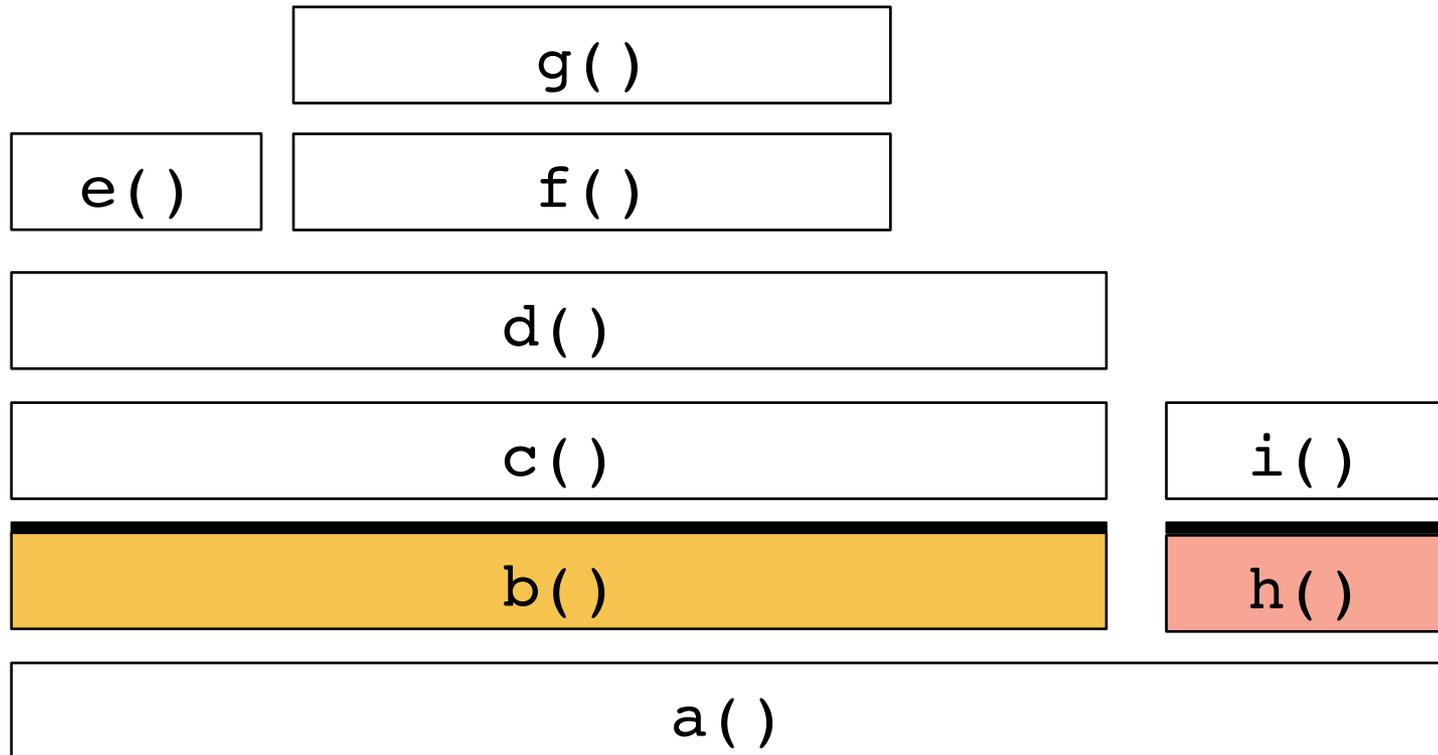
e.g., from g():



# Flame Graph Interpretation (3/3)

Widths are proportional to presence in samples

e.g., comparing b() to h() (incl. children)



# Flame Graph Colors

- Randomized by default
- Can be used as a dimension. e.g.:
  - Mixed-mode flame graphs
  - Differential flame graphs
  - Search

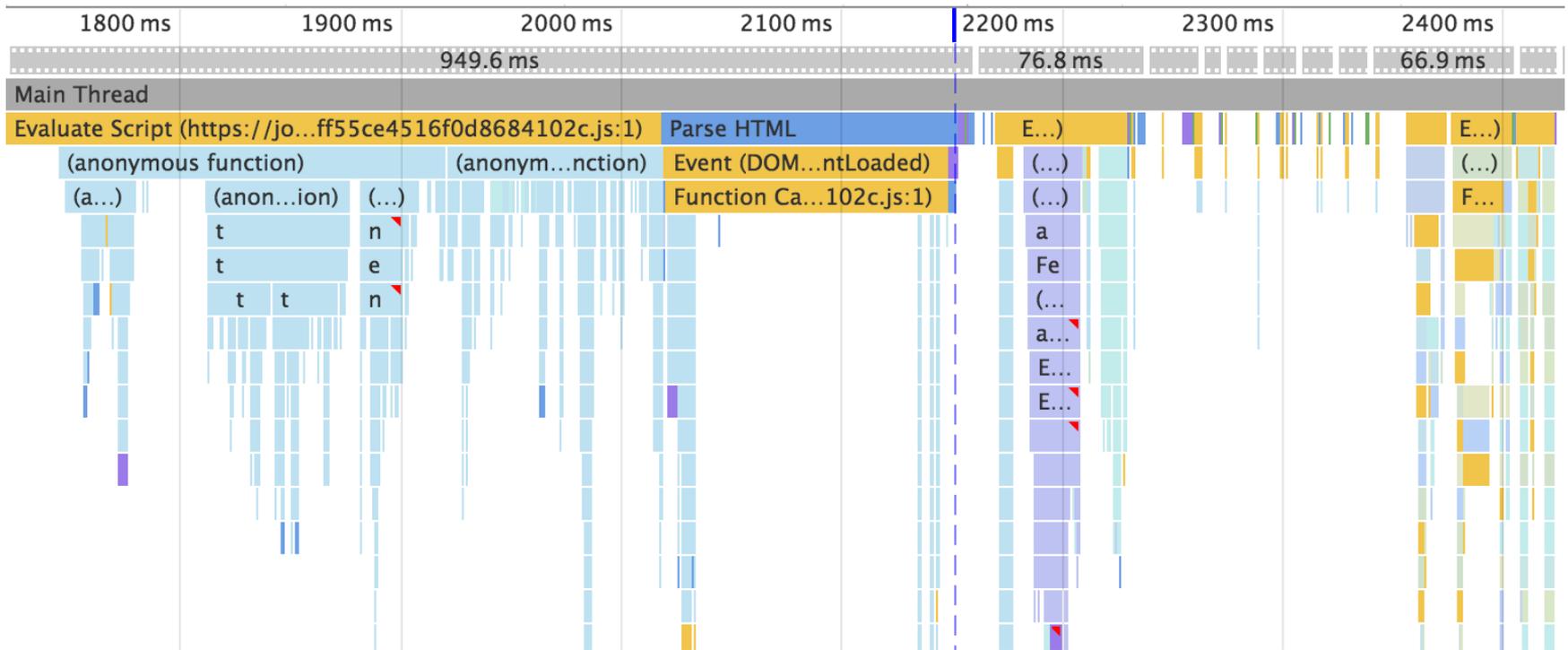






# Flame Charts

- Final note: these are useful, but are not flame *graphs*



- Flame **charts**: x-axis is time
- Flame **graphs**: x-axis is population (maximize merging)

# Stack Tracing

# System Profiling Java on x86

- For example, using Linux 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)

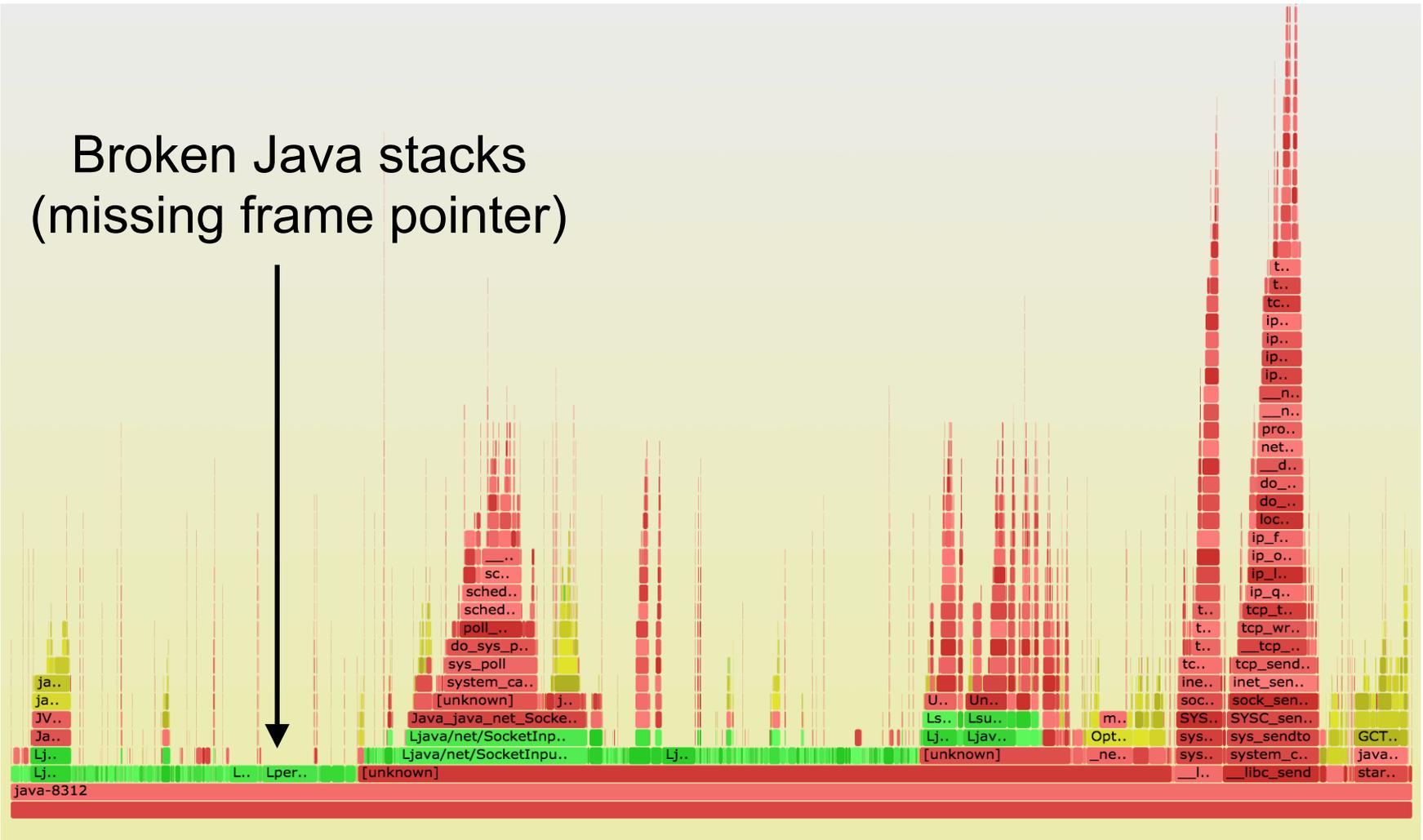
java 4579 cpu-clock:
    7f41792fc65f [unknown] (/tmp/perf-4458.map)
    a2d53351ff7da603 [unknown] ([unknown])

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

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

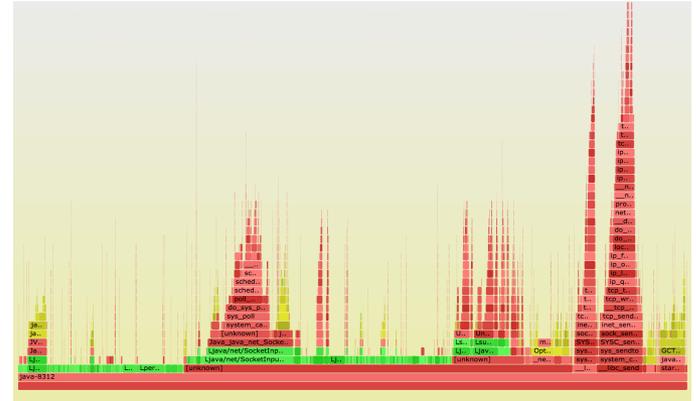
# ... as a Flame Graph

Broken Java stacks  
(missing frame pointer)



# Why Stacks are Broken

- On x86 (x86\_64), hotspot uses the frame pointer register (RBP) as general purpose
- This "compiler optimization" breaks (simple) 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, but the JVM had no such option...



# Fixing Stack Walking

Possibilities:

A. Fix frame pointer-based stack walking (the default)

- Pros: simple, supported by many tools
- Cons: might cost a little extra CPU

B. Use a custom walker (likely needing kernel support)

- Pros: full stack walking (incl. inlining) & arguments
- Cons: custom kernel code, can cost more CPU when in use

C. Try libunwind and DWARF

- Even feasible with JIT?

Our current preference is (A)

# Hacking OpenJDK (1/2)

- As a proof of concept, I hacked hotspot to support an x86\_64 frame pointer

```
--- 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

# Hacking OpenJDK (2/2)

```
--- 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); ← Fix x86_64 function
    // Remove word for ebp                                prologues
    framesize -= wordSize;

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

- We used this patched version successfully for some limited (and urgent) performance analysis

# -XX:+PreserveFramePointer

- We shared our patch publicly
  - See "A hotspot patch for stack profiling (frame pointer)" on the hotspot compiler dev mailing list
  - It became **JDK-8068945** for JDK 9 and **JDK-8072465** for JDK 8, and the -XX:+PreserveFramePointer option
- Zoltán Majó (Oracle) took this on, rewrote it, and it is now:
  - In **JDK 9**
  - In **JDK 8 update 60** build 19
  - Thanks to Zoltán, Oracle, and the other hotspot engineers for helping get this done!
- It might cost 0 – 3% CPU, depending on workload

# Broken Java Stacks (before)

```
# 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

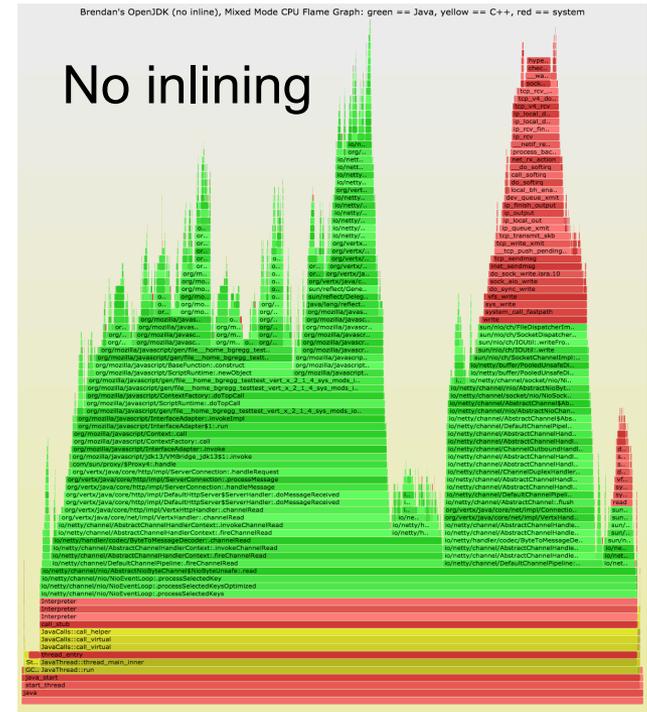
```
# perf script
[...]
java 8131 cpu-clock:
 7fff76f2dce1 [unknown] ([vdso])
 7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm...
 7fd301861e46 [unknown] (/tmp/perf-8131.map)
 7fd30184def8 [unknown] (/tmp/perf-8131.map)
 7fd30174f544 [unknown] (/tmp/perf-8131.map)
 7fd30175d3a8 [unknown] (/tmp/perf-8131.map)
 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 `-XX:+PreserveFramePointer` stacks are full, and go all the way to `start_thread()`
- This is what the CPUs are really running: inlined frames are not present



# Stacks & Inlining

- Frames may be missing (inlined)
- Disabling inlining:
  - `-XX:-Inline`
  - Many more Java frames
  - Can be 80% slower!
- May not be necessary
  - Inlined flame graphs often make enough sense
  - Or tune `-XX:MaxInlineSize` and `-XX:InlineSmallCode` a little to reveal more frames
    - Can even improve performance!
- `perf-map-agent` (next) has experimental un-inline support



**Symbols**

# Missing Symbols

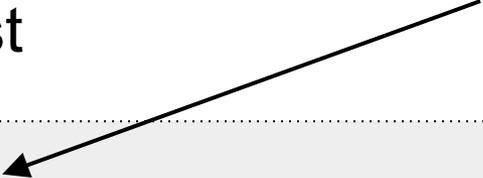
- Missing symbols may show up as hex; e.g., Linux perf:

```
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 JIT'd code, Linux perf already looks for an externally provided symbol file: /tmp/perf-PID.map, and warns if it doesn't exist



```
# 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)
[...]
```

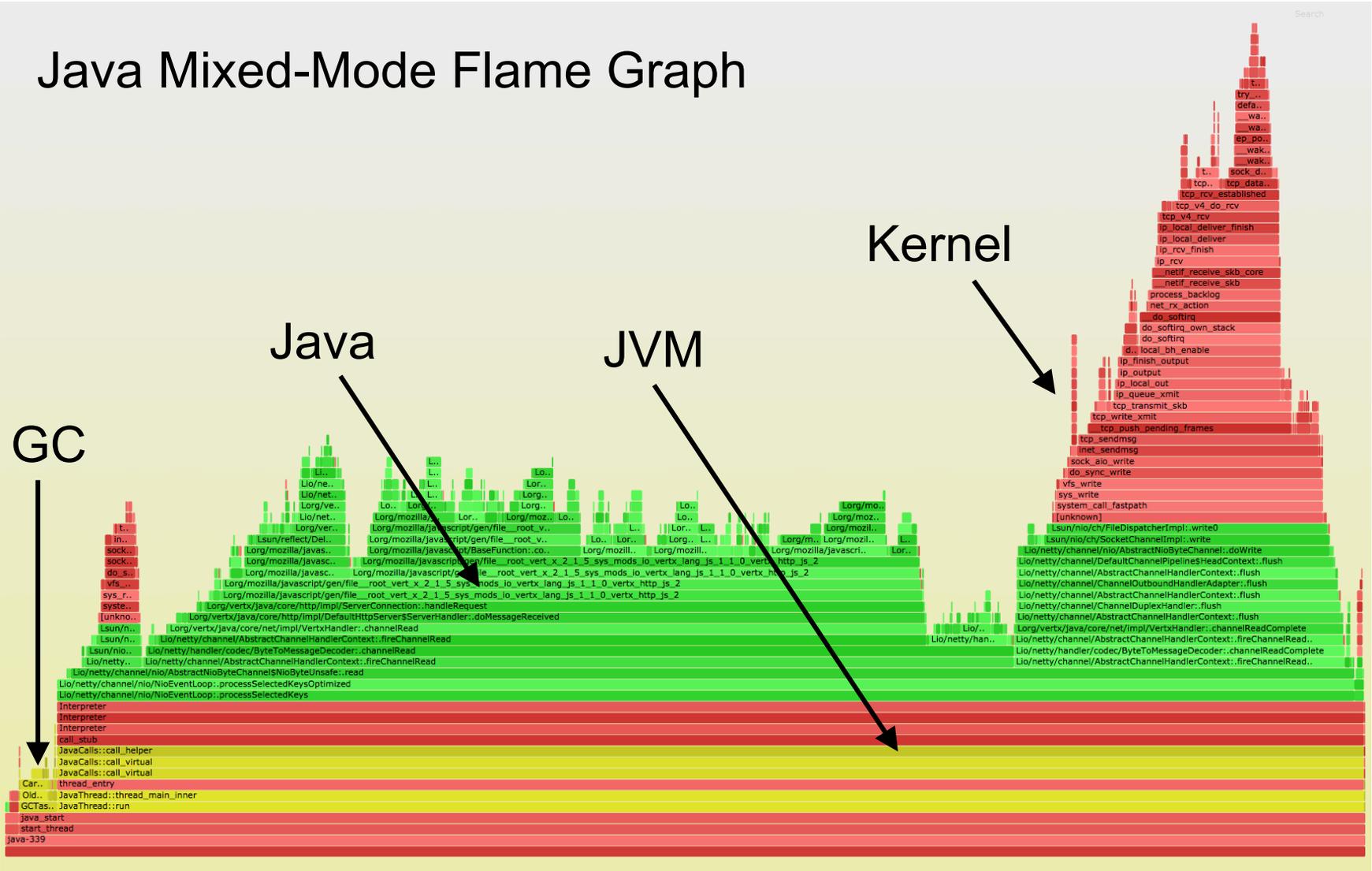
- This file can be created by a Java agent

# Java Symbols for perf

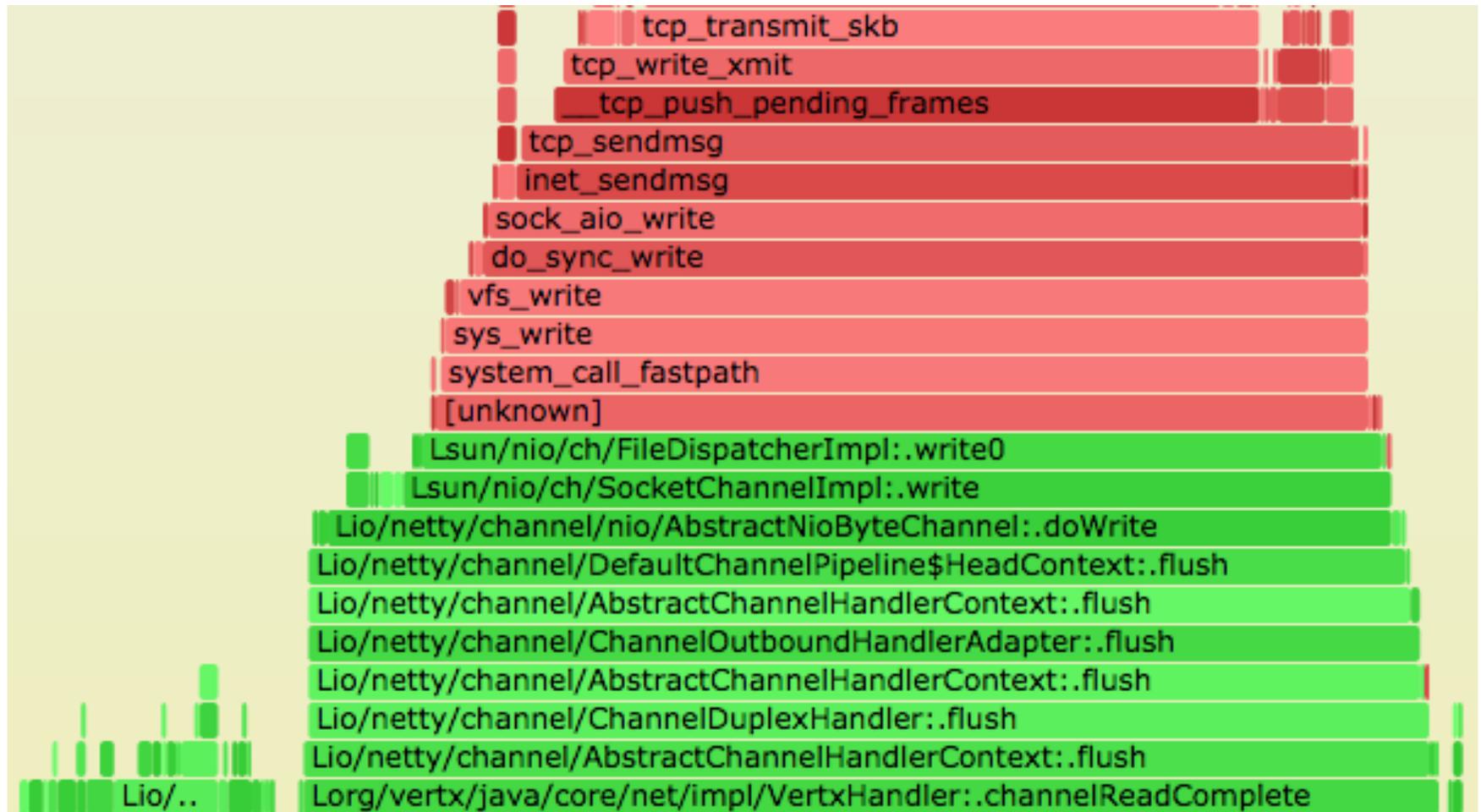
- perf-map-agent
  - <https://github.com/jrudolph/perf-map-agent>
  - Agent attaches and writes the /tmp file on demand (previous versions attached on Java start, wrote continually)
  - Thanks Johannes Rudolph!
- Use of a /tmp symbol file
  - Pros: simple, can be low overhead (snapshot on demand)
  - Cons: stale symbols
- Using a symbol logger with perf instead
  - Patch by Stephane Eranian currently being discussed on lkml; see "perf: add support for profiling jitted code"

# Stacks & Symbols

## Java Mixed-Mode Flame Graph



# Stacks & Symbols (zoom)



# Instructions

# Instructions

1. Check Java version
2. Install perf-map-agent
3. Set `-XX:+PreserveFramePointer`
4. Profile Java
5. Dump symbols
6. Generate Mixed-Mode Flame Graph

Note these are unsupported: use at your own risk

Reference: <http://techblog.netflix.com/2015/07/java-in-flames.html>

# 1. Check Java Version

- Need JDK8u60 or better
  - for -XX:+PreserveFramePointer

```
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

- Upgrade Java if necessary

## 2. Install perf-map-agent

- Check <https://github.com/jrudolph/perf-map-agent> for the latest instructions. e.g.:

```
$ sudo bash
# apt-get install -y cmake
# export JAVA_HOME=/usr/lib/jvm/java-8-oracle
# cd /usr/lib/jvm
# git clone --depth=1 https://github.com/jrudolph/perf-map-agent
# cd perf-map-agent
# cmake .
# make
```

# 3. Set -XX:+PreserveFramePointer

- Needs to be set on Java startup
- Check it is enabled (on Linux):

```
$ ps wwp `pgrep -n java` | grep PreserveFramePointer
```

# 4. Profile Java

- Using Linux `perf_events` to profile all processes, at 99 Hertz, for 30 seconds (as root):

```
# perf record -F 99 -a -g -- sleep 30
```

- Just profile one `PID` (broken on some older kernels):

```
# perf record -F 99 -p PID -g -- sleep 30
```

- These create a `perf.data` file

# 5. Dump Symbols

- See perf-map-agent docs for updated usage
- e.g., as the same user as java:

```
$ cd /usr/lib/jvm/perf-map-agent/out
$ java -cp attach-main.jar:$JAVA_HOME/lib/tools.jar \
  net.virtualvoid.perf.AttachOnce PID
```

- perf-map-agent contains helper scripts. I wrote my own:
  - <https://github.com/brendangregg/Misc/blob/master/java/jmaps>
- Dump symbols quickly after perf record to minimize stale symbols. How I do it:

```
# perf record -F 99 -a -g -- sleep 30; jmaps
```

## 6. Generate a Mixed-Mode Flame Graph

- Using my FlameGraph software:

```
# perf script > out.stacks01
# git clone --depth=1 https://github.com/brendangregg/FlameGraph
# cat out.stacks01 | ./FlameGraph/stackcollapse-perf.pl | \
  ./FlameGraph/flamegraph.pl --color=java --hash > flame01.svg
```

- perf script reads perf.data with /tmp/\*.map
- out.stacks01 is an intermediate file; can be handy to keep
- Finally open flame01.svg in a browser
- Check for newer flame graph implementations (e.g., d3)

# Automation

# Netflix Vector



VECTOR

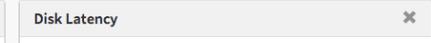
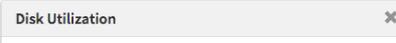
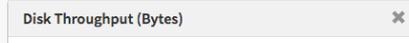
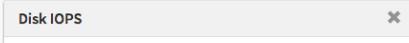
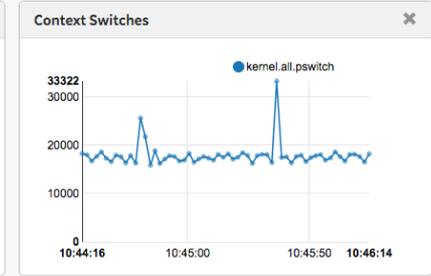
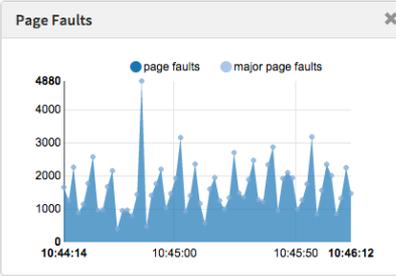
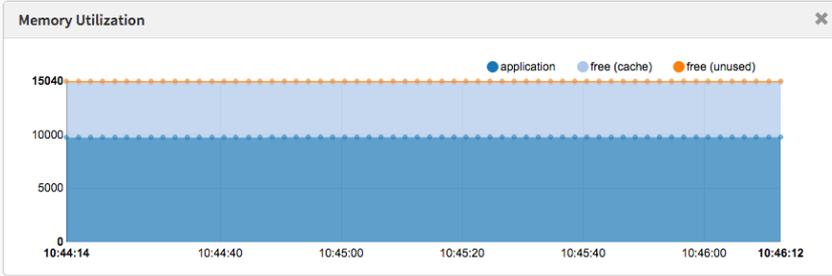
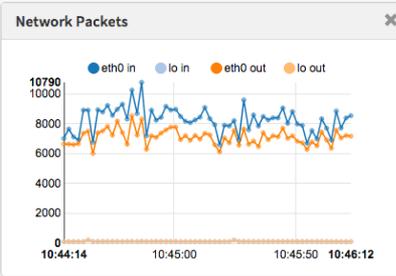
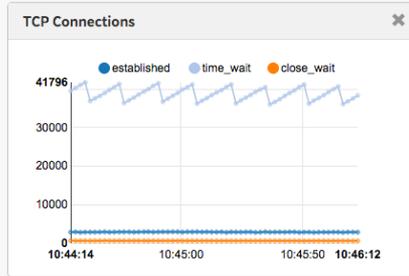
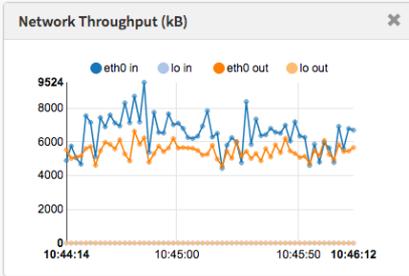
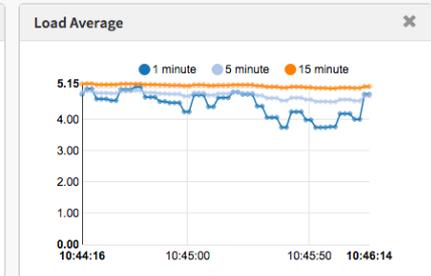
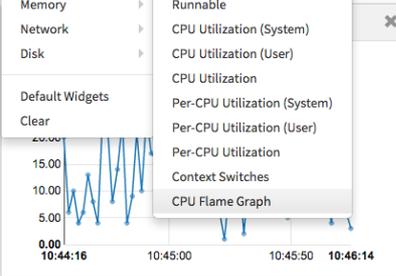
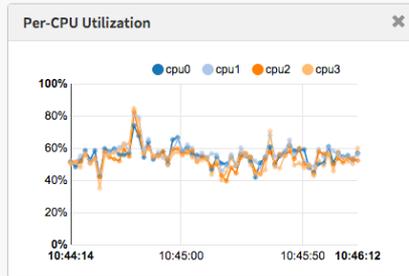
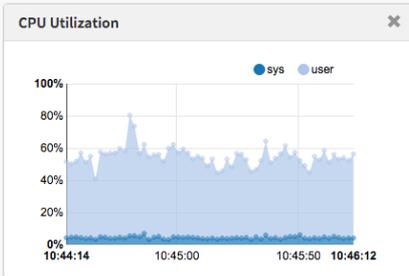
Hostname ec2-██████████.compute-1.amazonaws.com ✓

Widget ▾

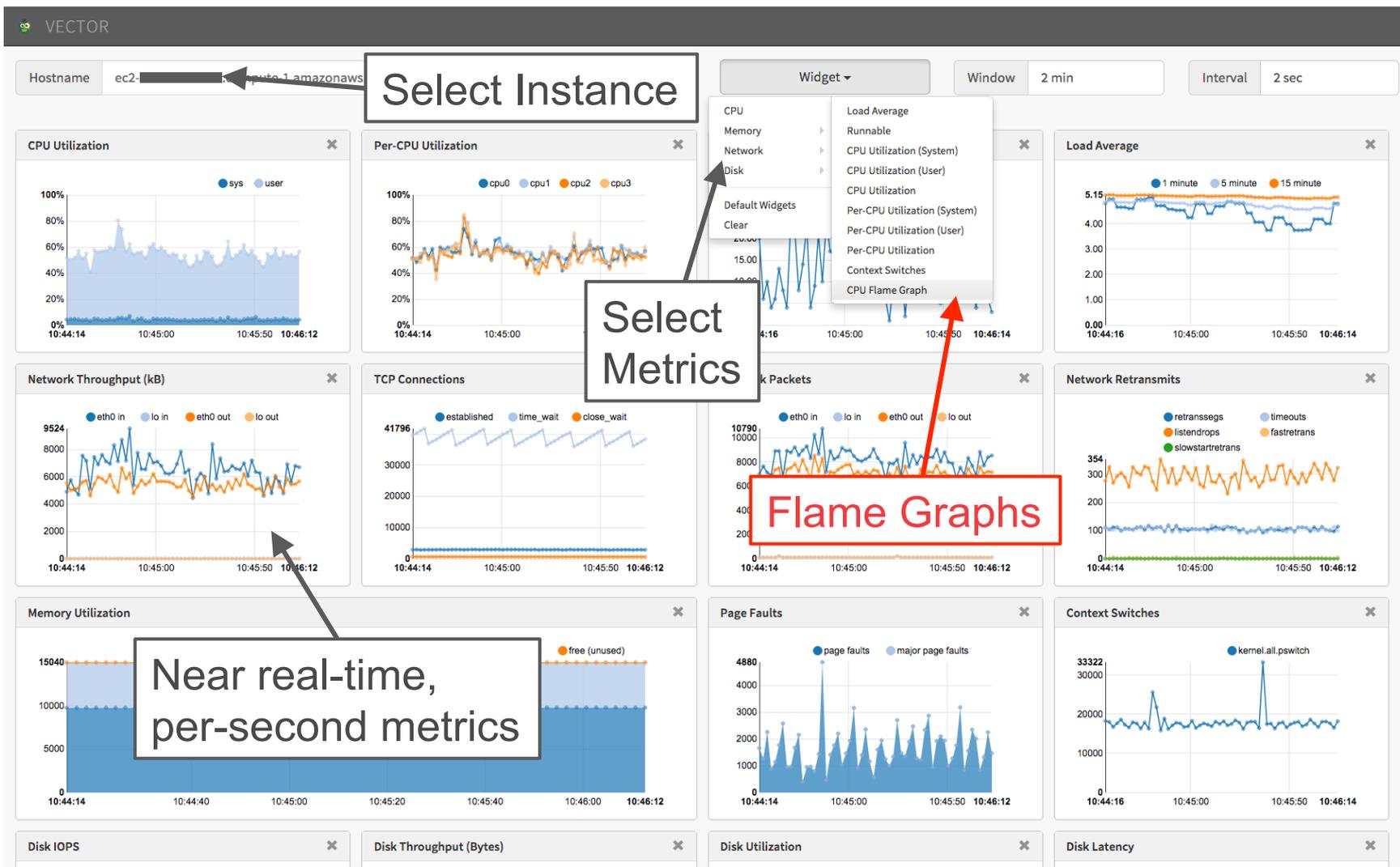
Window 2 min

Interval 2 sec

- CPU
- Memory
- Network
- Disk
- Default Widgets
- Clear
- Load Average
- Runnable
- CPU Utilization (System)
- CPU Utilization (User)
- CPU Utilization
- Per-CPU Utilization (System)
- Per-CPU Utilization (User)
- Per-CPU Utilization
- Context Switches
- CPU Flame Graph



# Netflix Vector



# Netflix Vector

- Open source, on-demand, instance analysis tool
  - <https://github.com/netflix/vector>
- Shows various real-time metrics
- Flame graph support currently in development
  - Automating previous steps
  - Using it internally already
  - Also developing a new d3 front end

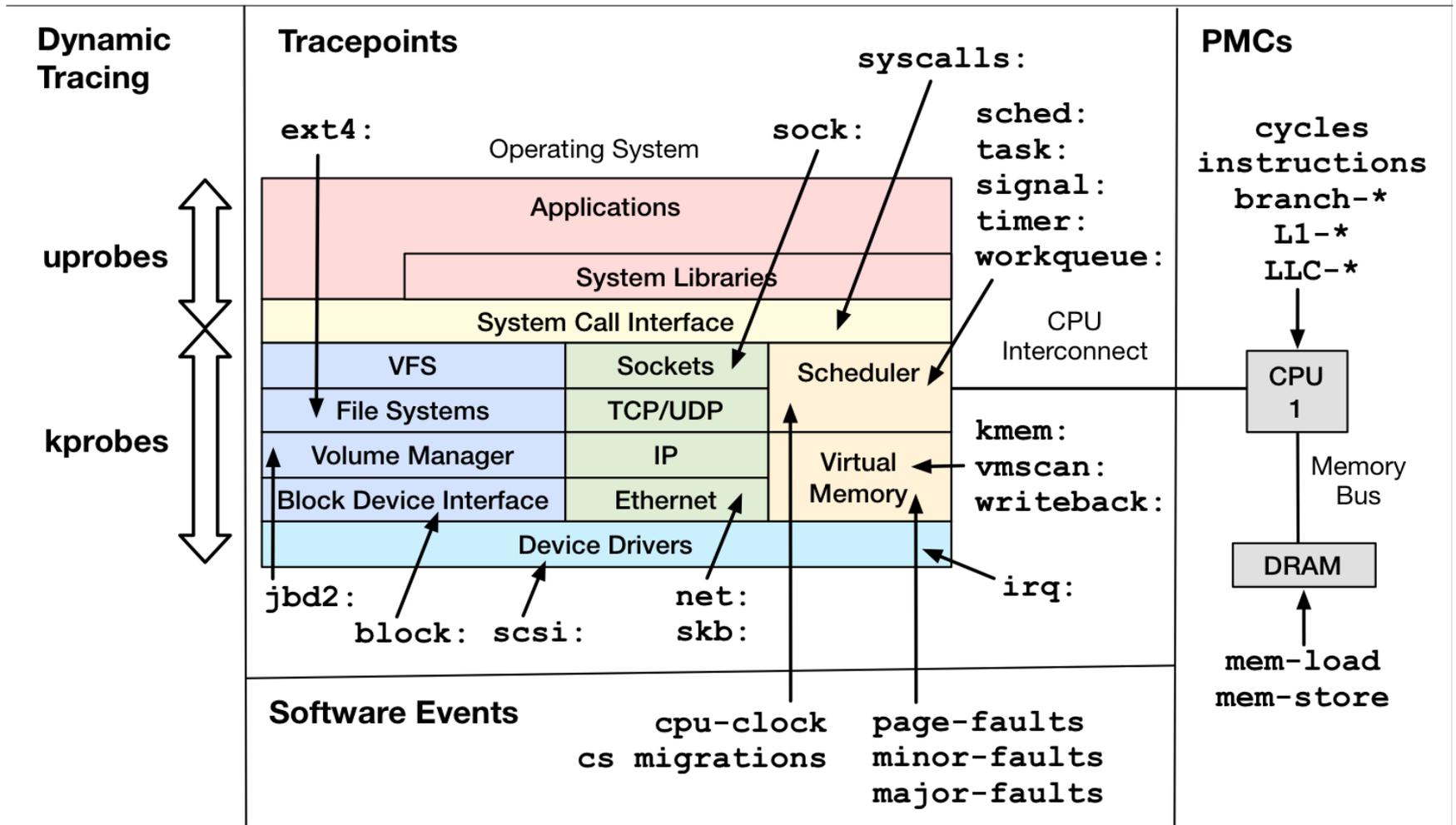


# DEMO

d3-flame-graph

# Advanced Analysis

# Linux perf\_events Coverage



... all possible with Java stacks

# Advanced Flame Graphs

- Examples:
  - Page faults
  - Context switches
  - Disk I/O requests
  - TCP events
  - CPU cache misses
  - CPI
- Any event issued in *synchronous* Java context

# Synchronous Java Context

- Java thread still on-CPU, and event is directly triggered
  - Examples:
    - Disk I/O requests issued directly by Java → yes
      - direct reads, sync writes, page faults
    - Disk I/O completion interrupts → no\*
    - Disk I/O requests triggered async, e.g., readahead → no\*
- \* can be made yes by tracing and associating context

# Page Faults

- Show what triggered main memory (resident) to grow:

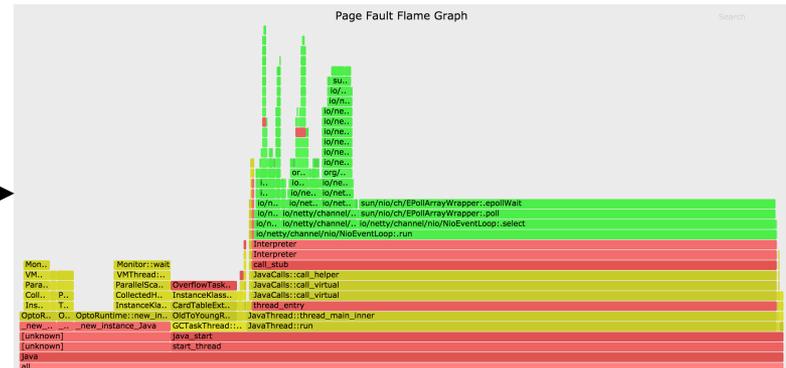
```
# perf record -e page-faults -p PID -g -- sleep 120
```

- "fault" as (physical) main memory is allocated on-demand, when a virtual page is first populated
- Low overhead tool to solve some types of memory leak

RES column in top(1)

VIRT	RES	COMMAND
3972756	376876	java
344752	231344	ab
0	0	kworker/1:1
1069716	44032	evolution-calen
0	0	ksoftirqd/2

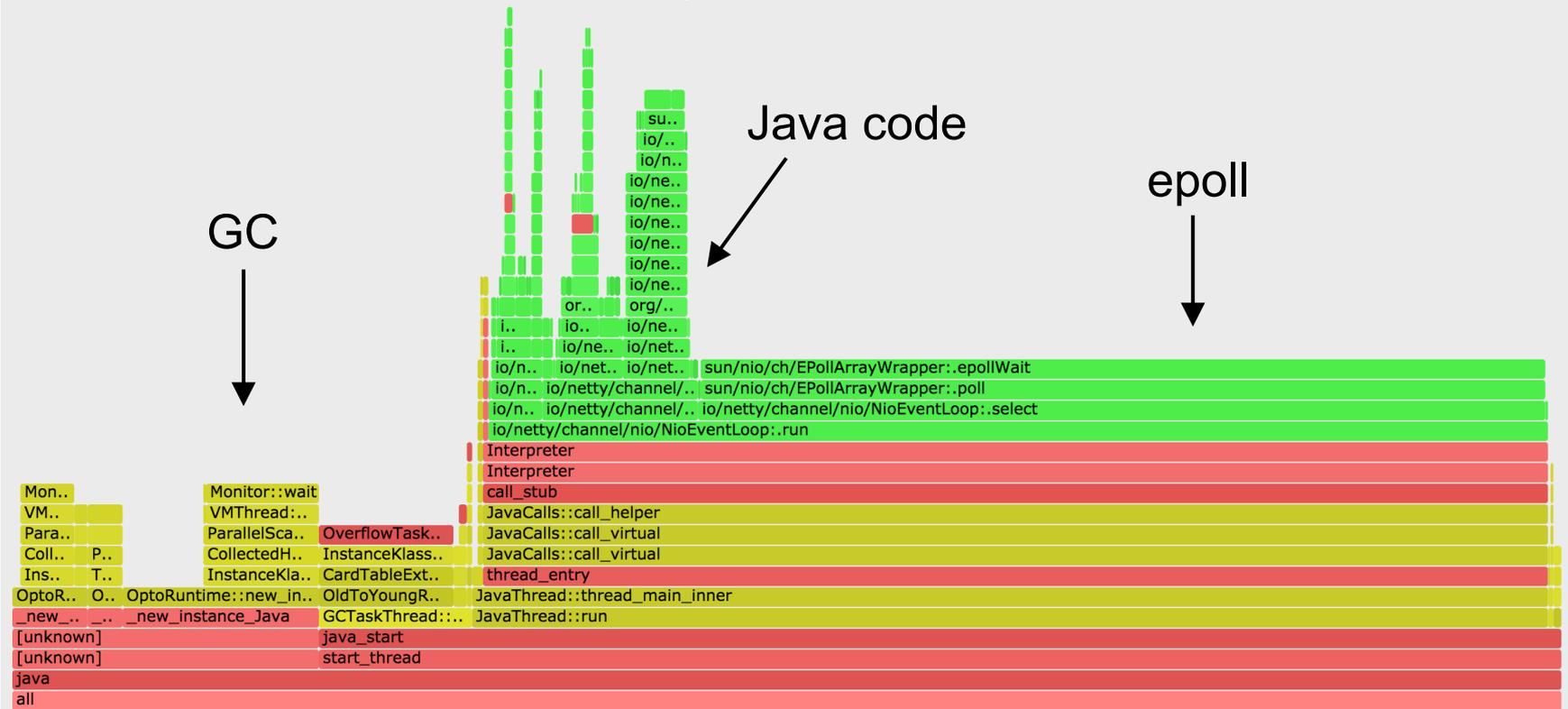
grows  
because



# Page Fault Flame Graph

Page Fault Flame Graph

Search



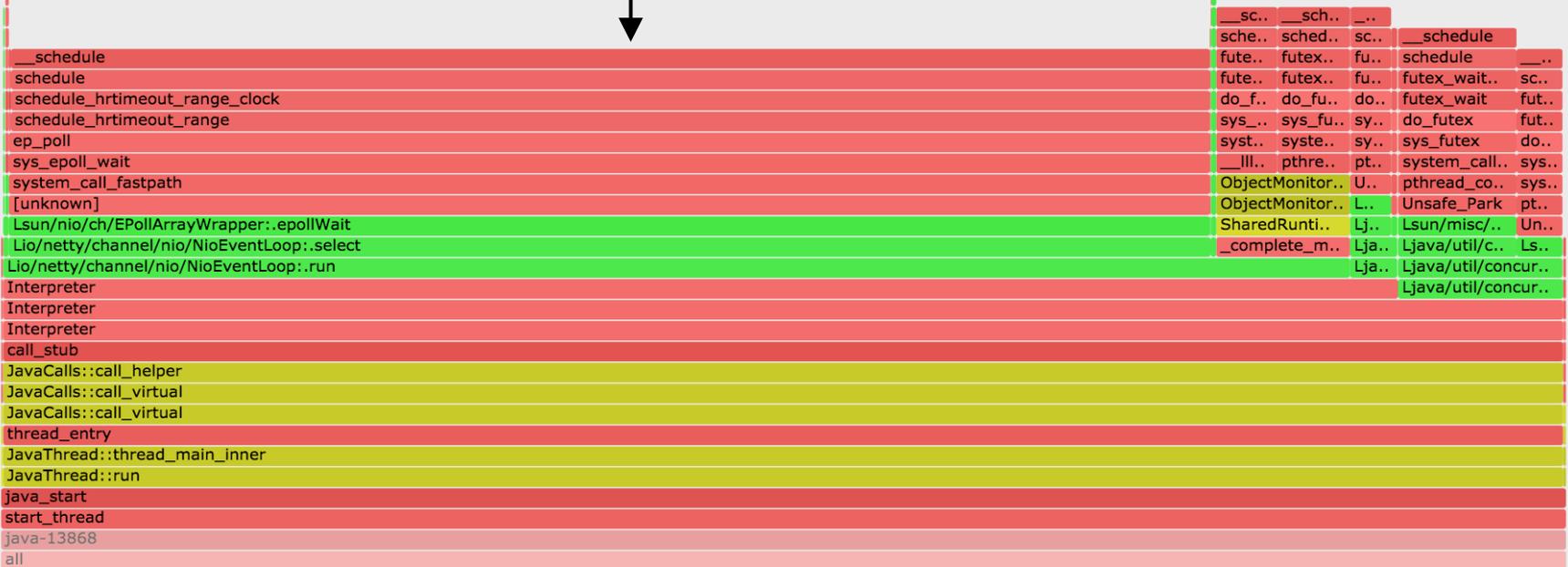


# Context Switch Flame Graph (1/2)

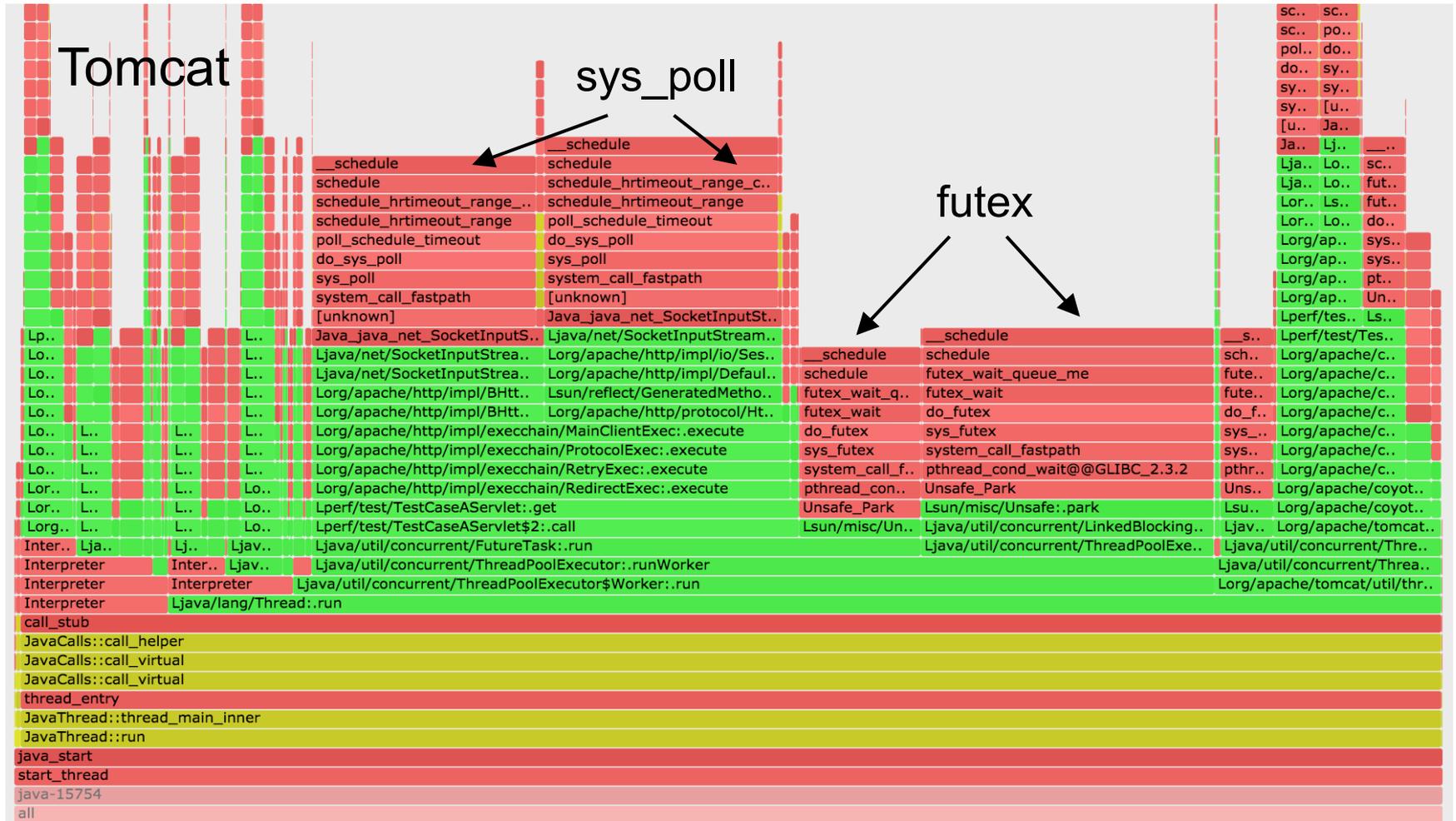
rxNetty

epoll

futex



# Context Switch Flame Graph (2/2)

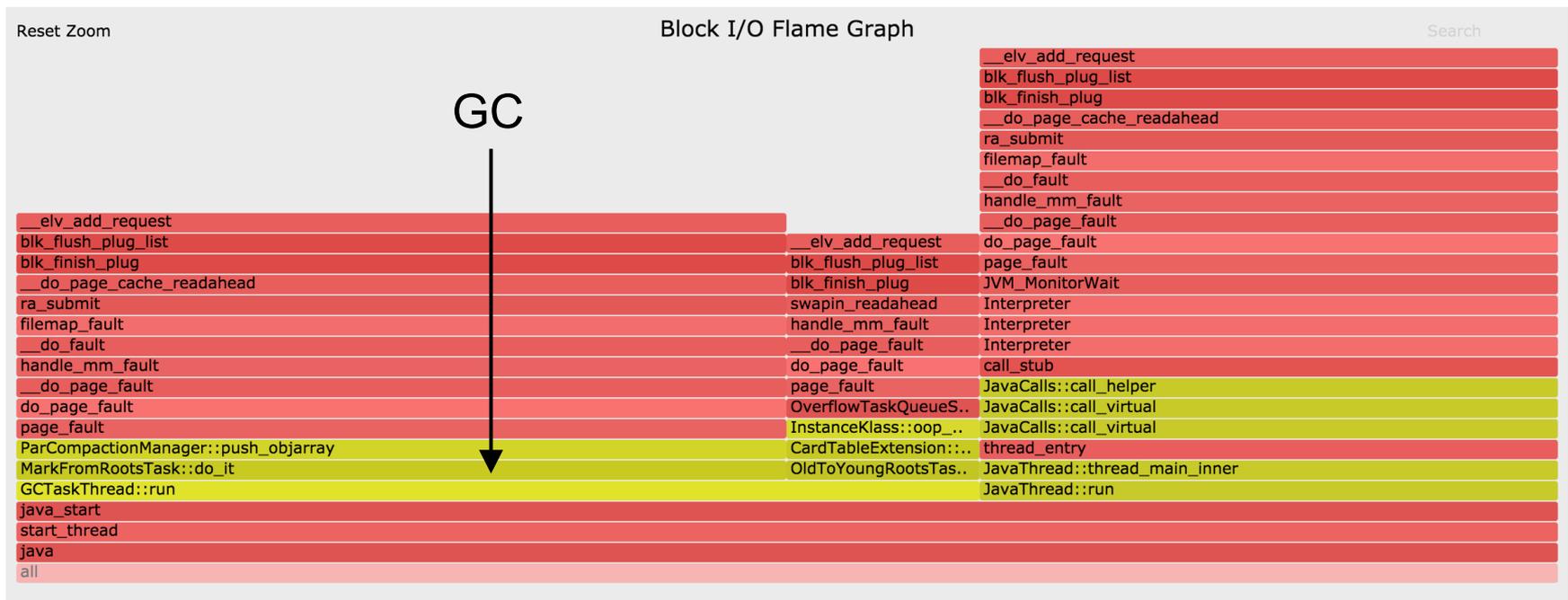


# Disk I/O Requests

- Shows who issued disk I/O (sync reads & writes):

```
# perf record -e block:block_rq_insert -a -g -- sleep 60
```

- e.g.: page faults in GC? This JVM has swapped out!:

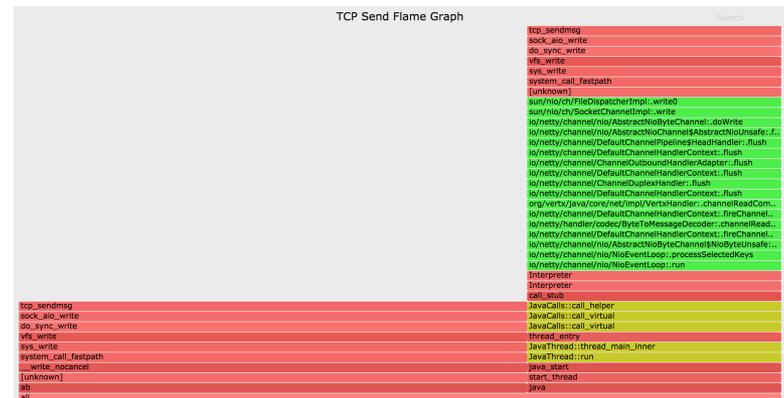


# TCP Events

- TCP transmit, using dynamic tracing:

```
# perf probe tcp_sendmsg
# perf record -e context-switches -a -g -- sleep 1; jmaps
# perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso,trace > out.stacks
# perf probe --del tcp_sendmsg
```

- Note: can be high overhead for high packet rates
  - For the current perf trace, dump, post-process cycle
- Can also trace TCP connect & accept (lower overhead)
- TCP receive is async
  - Could trace via socket read



# TCP Send Flame Graph

Only one code-path  
taken in this example

TCP Send Flame Graph

Search

kernel

Java

JVM

ab (client process)

```
tcp_sendmsg
sock_aio_write
do_sync_write
vfs_write
sys_write
system_call_fastpath
__write_nocancel
[unknown]
ab
all
```

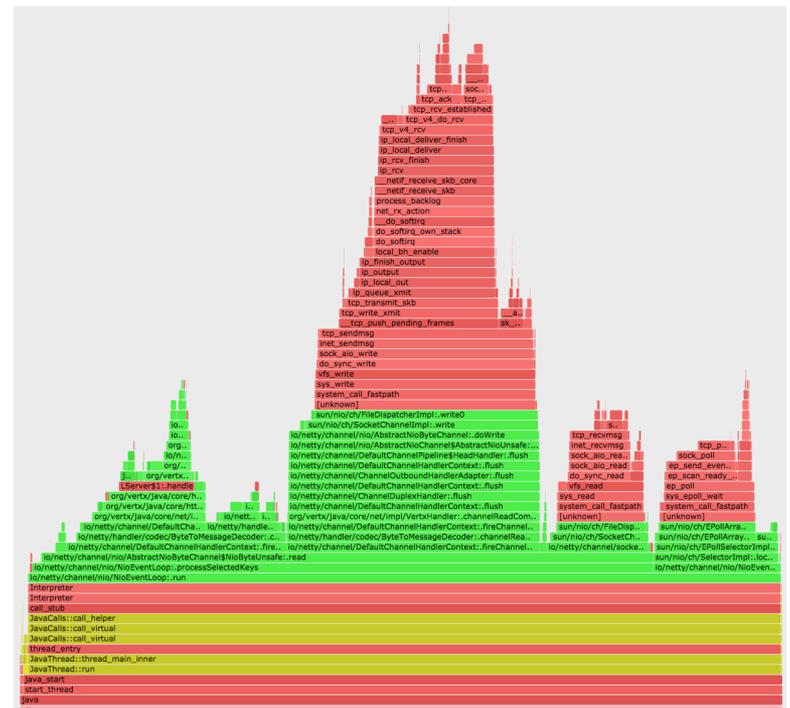
```
tcp_sendmsg
sock_aio_write
do_sync_write
vfs_write
sys_write
system_call_fastpath
[unknown]
sun/nio/ch/FileDispatcherImpl:.write0
sun/nio/ch/SocketChannelImpl:.write
io/netty/channel/nio/AbstractNioByteChannel:.doWrite
io/netty/channel/nio/AbstractNioChannel$AbstractNioUnsafe:.f..
io/netty/channel/DefaultChannelPipeline$HeadHandler:.flush
io/netty/channel/DefaultChannelHandlerContext:.flush
io/netty/channel/ChannelOutboundHandlerAdapter:.flush
io/netty/channel/DefaultChannelHandlerContext:.flush
io/netty/channel/ChannelDuplexHandler:.flush
io/netty/channel/DefaultChannelHandlerContext:.flush
org.vertx.java.core.net.impl.VertxHandler:.channelReadCom..
io/netty/channel/DefaultChannelHandlerContext:.fireChannel..
io/netty/handler/codec/ByteToMessageDecoder:.channelRead..
io/netty/channel/DefaultChannelHandlerContext:.fireChannel..
io/netty/channel/nio/AbstractNioByteChannel$NioByteUnsafe:..
io/netty/channel/nio/NioEventLoop:.processSelectedKeys
io/netty/channel/nio/NioEventLoop:.run
Interpreter
Interpreter
call_stub
JavaCalls::call_helper
JavaCalls::call_virtual
JavaCalls::call_virtual
thread_entry
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java
```

# CPU Cache Misses

- In this example, sampling via Last Level Cache loads:

```
# perf record -e LLC-loads -c 10000 -a -g -- sleep 5; jmaps  
# perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso > out.stacks
```

- c is the count (samples once per count)
- Use other CPU counters to sample hits, misses, stalls

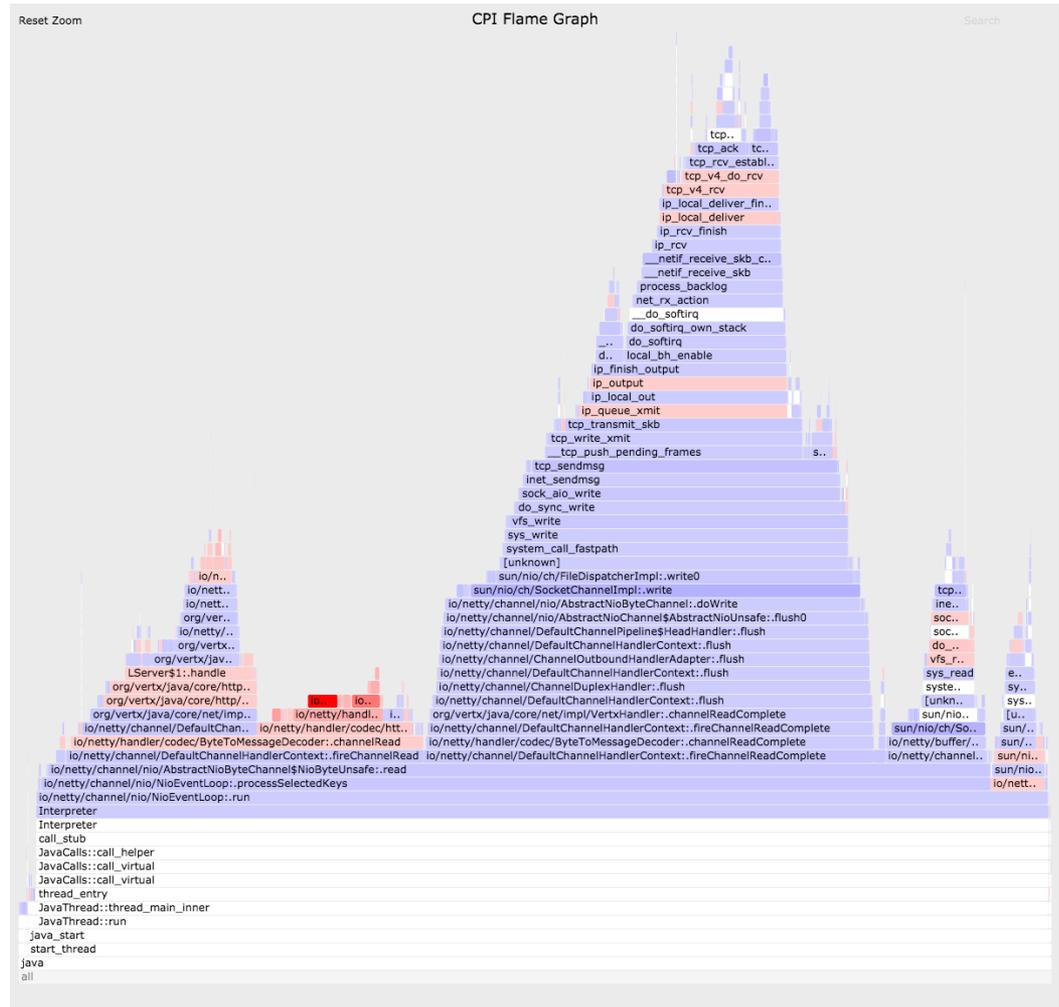
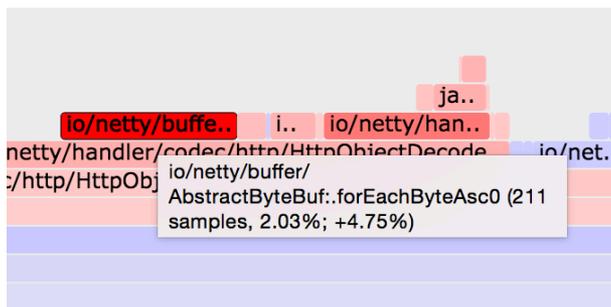




# CPI Flame Graph

- Cycles Per Instruction!
  - red == instruction heavy
  - blue == cycle heavy (likely mem stall cycles)

zoomed:



# Links & References

- Flame Graphs
  - <http://www.brendangregg.com/flamegraphs.html>
  - <http://techblog.netflix.com/2015/07/java-in-flames.html>
  - <http://techblog.netflix.com/2014/11/nodejs-in-flames.html>
  - <http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html>
- Linux perf\_events
  - [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page)
  - <http://www.brendangregg.com/perf.html>
  - <http://www.brendangregg.com/blog/2015-02-27/linux-profiling-at-netflix.html>
- Netflix Vector
  - <https://github.com/netflix/vector>
  - <http://techblog.netflix.com/2015/04/introducing-vector-netflixs-on-host.html>
- JDK tickets
  - JDK8: <https://bugs.openjdk.java.net/browse/JDK-8072465>
  - JDK9: <https://bugs.openjdk.java.net/browse/JDK-8068945>
- hprof: <http://www.brendangregg.com/blog/2014-06-09/java-cpu-sampling-using-hprof.html>



# Thanks

- Questions?
- <http://techblog.netflix.com>
- <http://slideshare.net/brendangregg>
- <http://www.brendangregg.com>
- [bgregg@netflix.com](mailto:bgregg@netflix.com)
- @brendangregg

