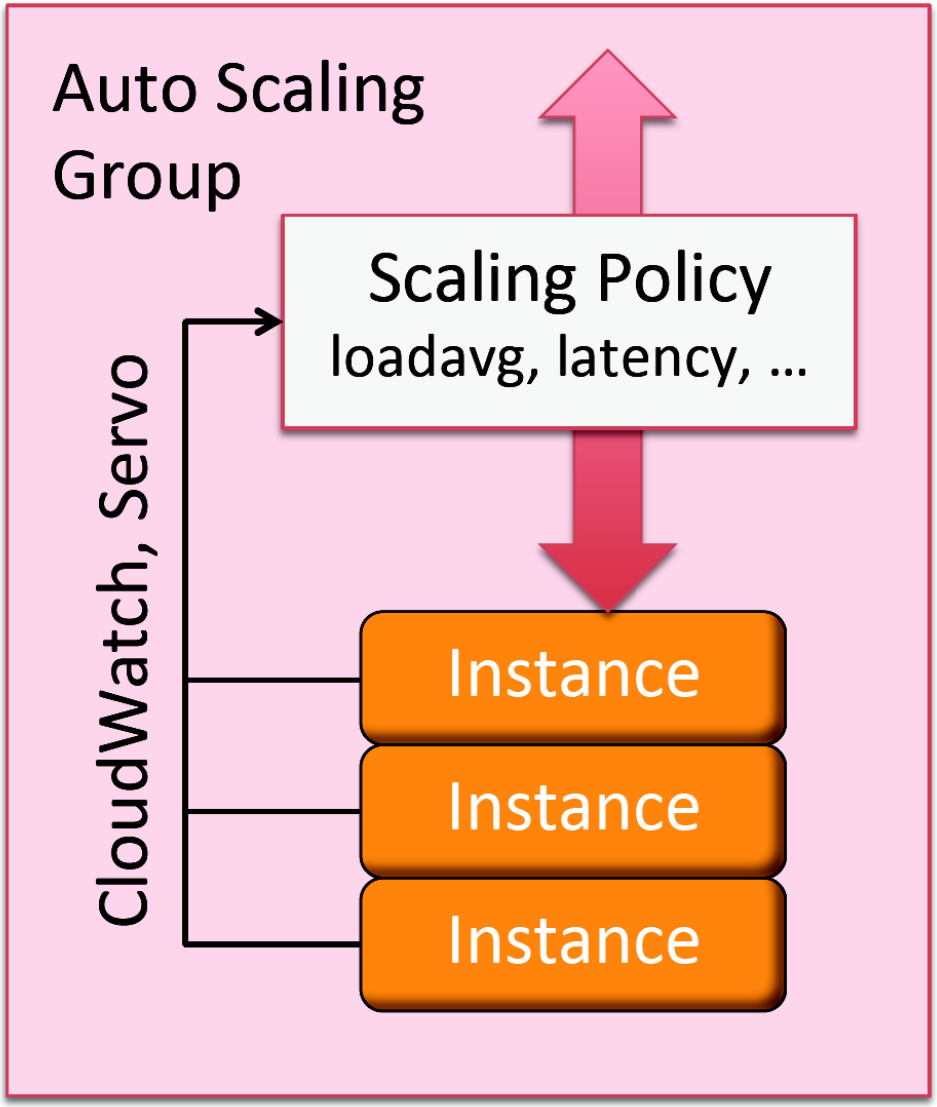
Hardware Performance Counters Interals-Part2

By Mohit Kumar

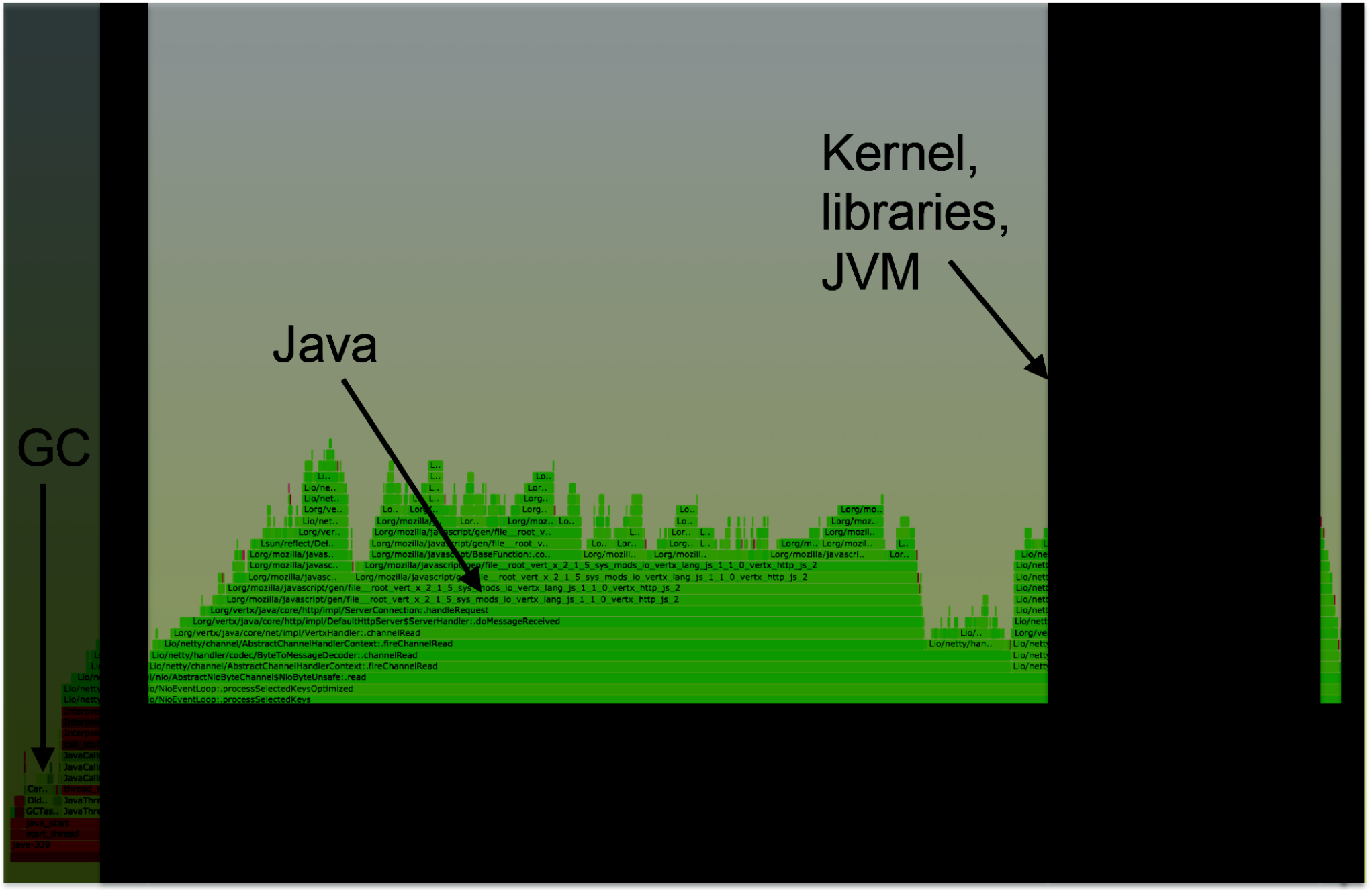
Why we need CPU profiling

- 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



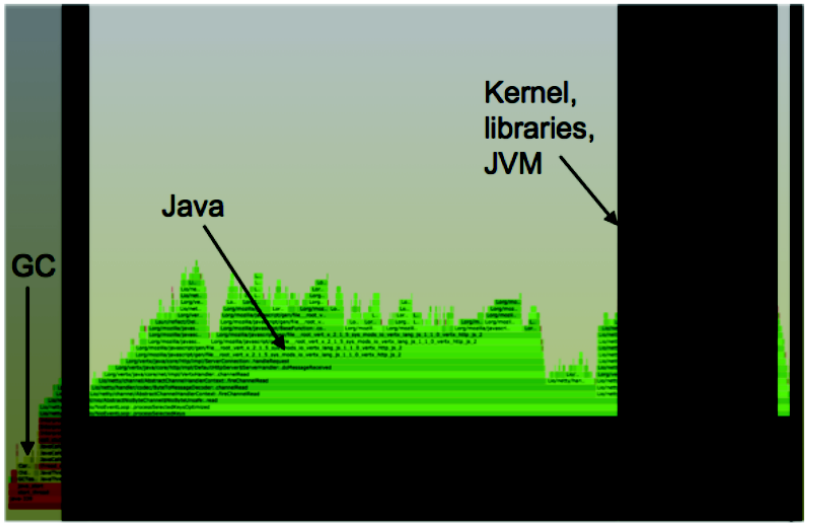
The problem with profilers

Java Profilers



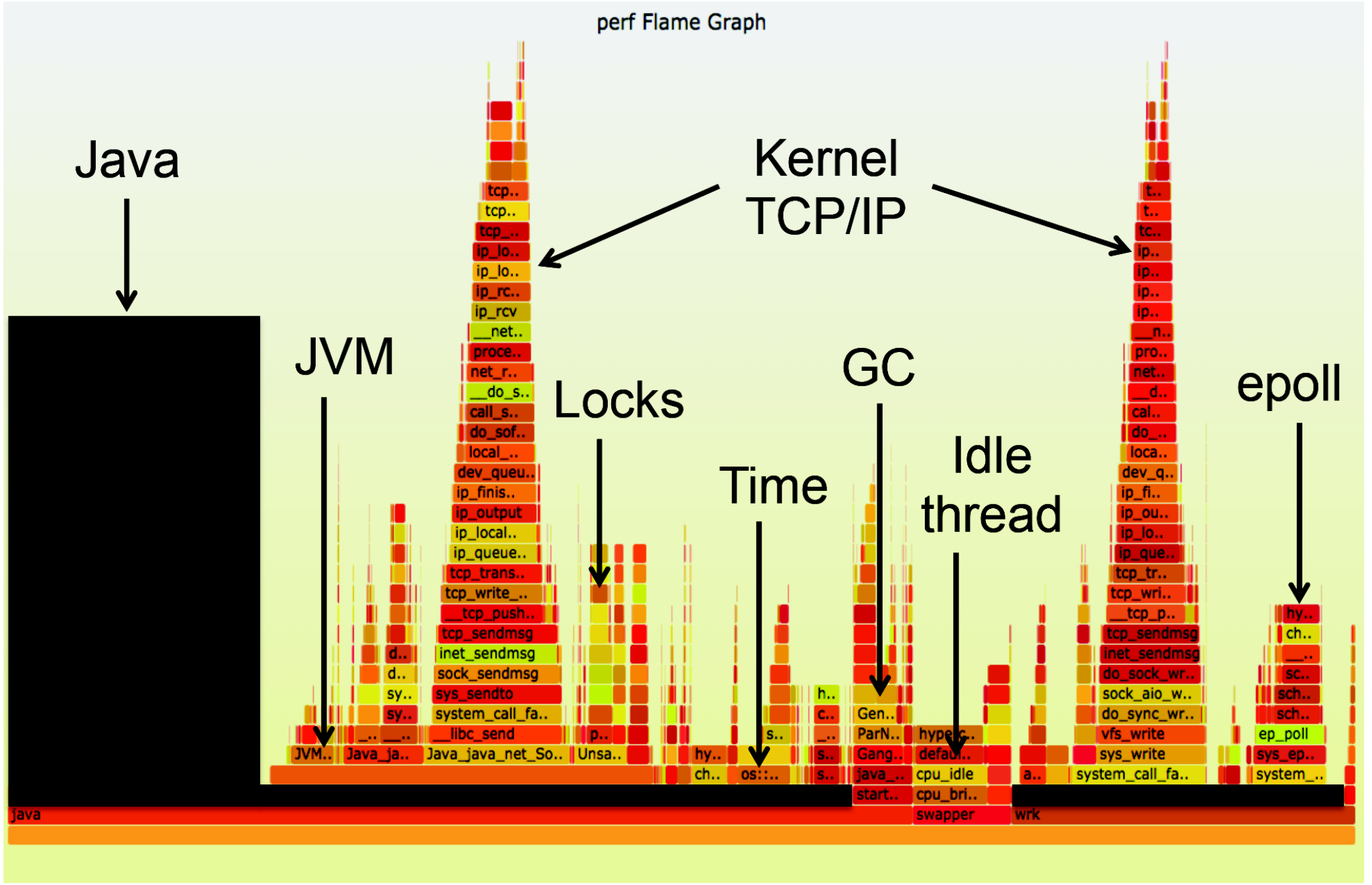
Java Profilers

- Visibility
  - Java method execution
  - Object usage
  - GC logs
  - Custom Java context



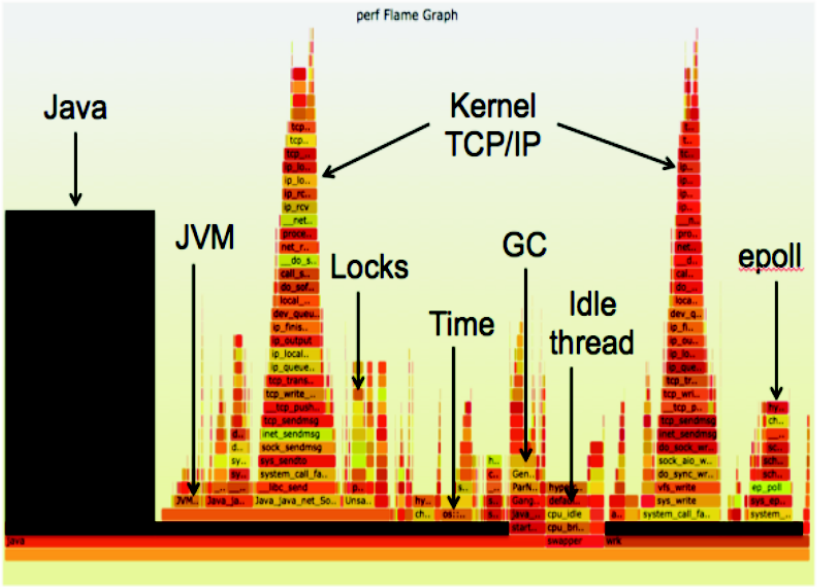
- Typical problems:
  - Sampling often happens at safety/yield points (skew)
  - Method tracing has massive observer effect
  - Misidentifies RUNNING as on-CPU (e.g., epoll)
  - Doesn't include or profile GC or JVM CPU time
  - Tree views not quick (proportional) to comprehend
- **Inaccurate** (skewed)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

System Profilers



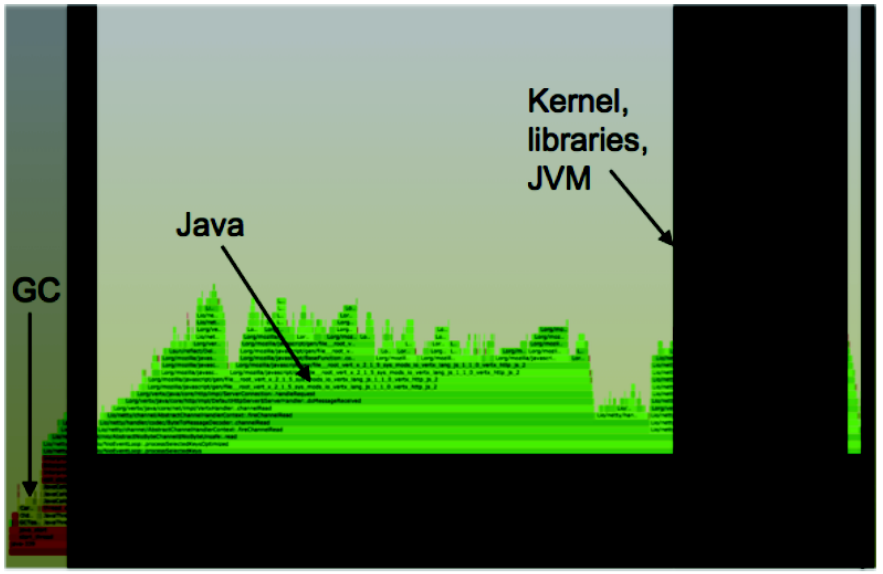
System Profilers

- 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**

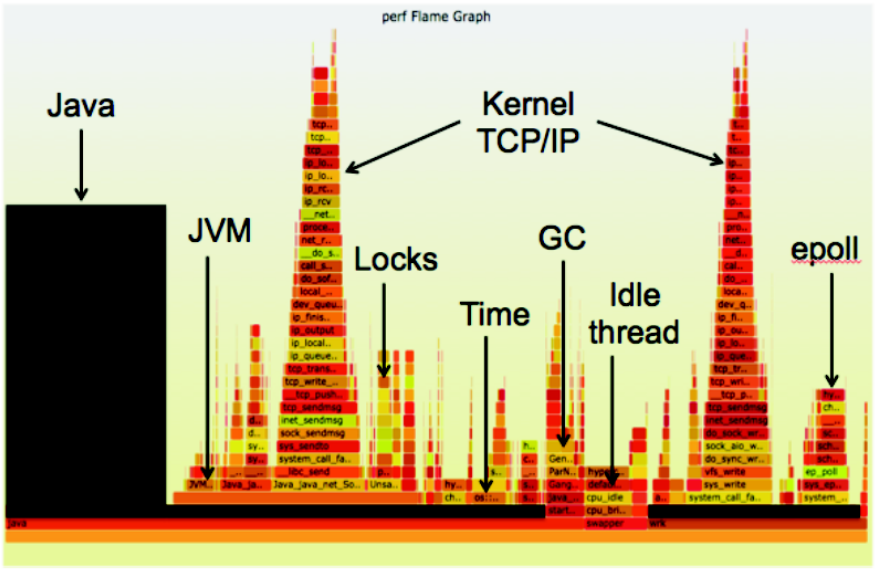


Workaround

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



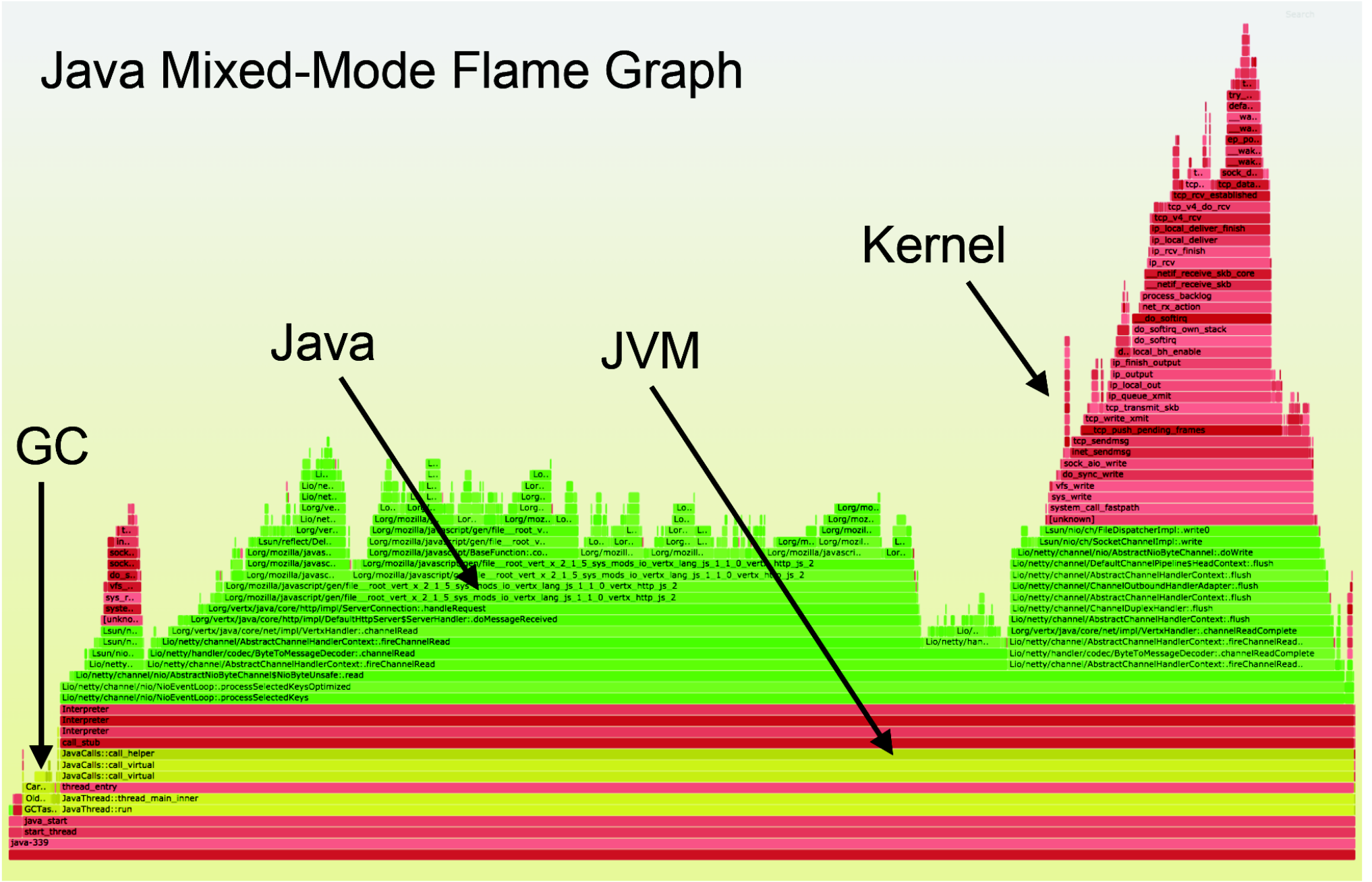
Java



System

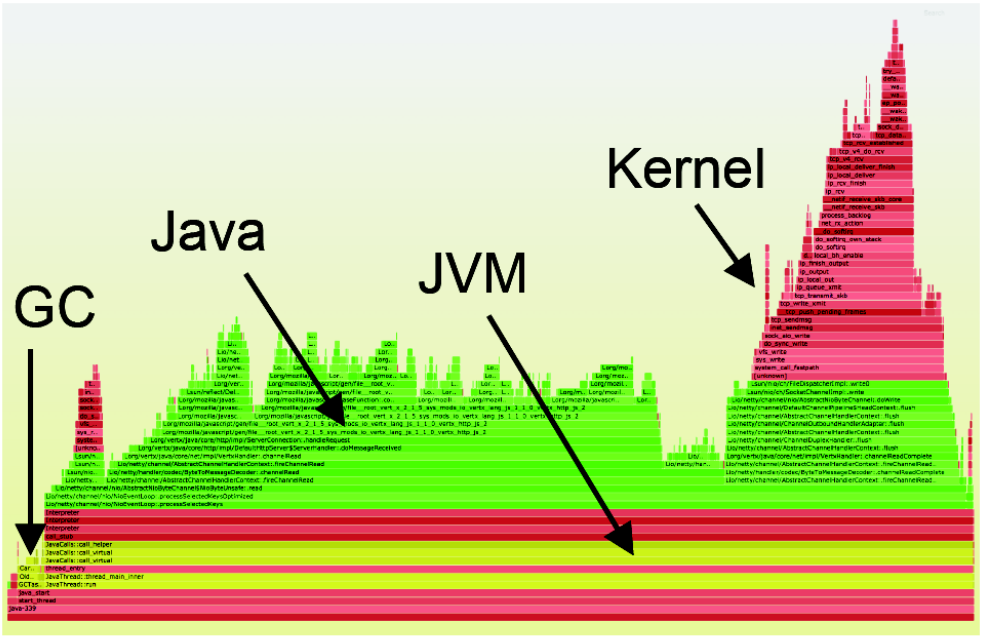
- An improvement, but Java context is often crucial for interpreting system profiles

Solution


Java Mixed-Mode Flame Graph
GC
Java
JVM
Kernel
Interpreter
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-339
Lio/netty/channel/nio/AbstractNioByteChannel$NioByteUnsafe:.read
Lio/netty/channel/nio/NioEventLoop:.processSelectedKeysOptimized
Lio/netty/channel/nio/NioEventLoop:.processSelectedKeys
Lorg/vertx/java/core/http/impl/ServerConnection:.handleRequest
Lorg/vertx/java/core/http/impl/DefaultHttpServer$ServerHandler:.doMessageReceived
Lorg/vertx/java/core/net/impl/VertxHandler:.channelRead
Lio/netty/channel/AbstractChannelHandlerContext:.fireChannelRead
Lio/netty/handler/codec/ByteToMessageDecoder:.channelRead
Lio/netty/channel/AbstractChannelHandlerContext:.fireChannelRead
Lsun/reflect/Del..
Lorg/mozilla/javas..
Lorg/mozilla/javasc..
Lorg/mozilla/javascript/gen/file__root_vert_x_2_1_5_sys_mods_io_vertx_lang_js_1_1_0_vertx_http_js_2
Lorg/mozilla/javascript/BaseFunction:.co..
Lorg/mozill..
Lorg/mozilla/javascri..
Lsun/nio/ch/FileDispatcherImpl:.write0
Lsun/nio/ch/SocketChannelImpl:.write
Lio/netty/channel/nio/AbstractNioByteChannel:.doWrite
Lio/netty/channel/DefaultChannelPipeline$HeadContext:.flush
Lio/netty/channel/AbstractChannelHandlerContext:.flush
Lio/netty/channel/ChannelOutboundHandlerAdapter:.flush
Lio/netty/channel/ChannelDuplexHandler:.flush
Lio/netty/handler/codec/ByteToMessageDecoder:.channelReadComplete
Lorg/vertx/java/core/net/impl/VertxHandler:.channelReadComplete
[unknown]
system_call_fastpath
sys_write
vfs_write
do_sync_write
sock_aio_write
inet_sendmsg
tcp_sendmsg
__tcp_push_pending_frames
tcp_write_xmit
tcp_transmit_skb
ip_queue_xmit
ip_local_out
ip_output
ip_finish_output
local_bh_enable
do_softirq
net_rx_action
process_backlog
__netif_receive_skb
__netif_receive_skb_core
ip_rcv
ip_rcv_finish
ip_local_deliver
ip_local_deliver_finish
tcp_v4_rcv
tcp_v4_do_rcv
tcp_rcv_established
tcp_data..
sock_d..

Solution

- Fix system profiling
  - Only way to see it all
- Visibility is everything:
  - Java methods
  - JVM (C++)
  - GC (C++)
  - libraries (C)
  - kernel (C)

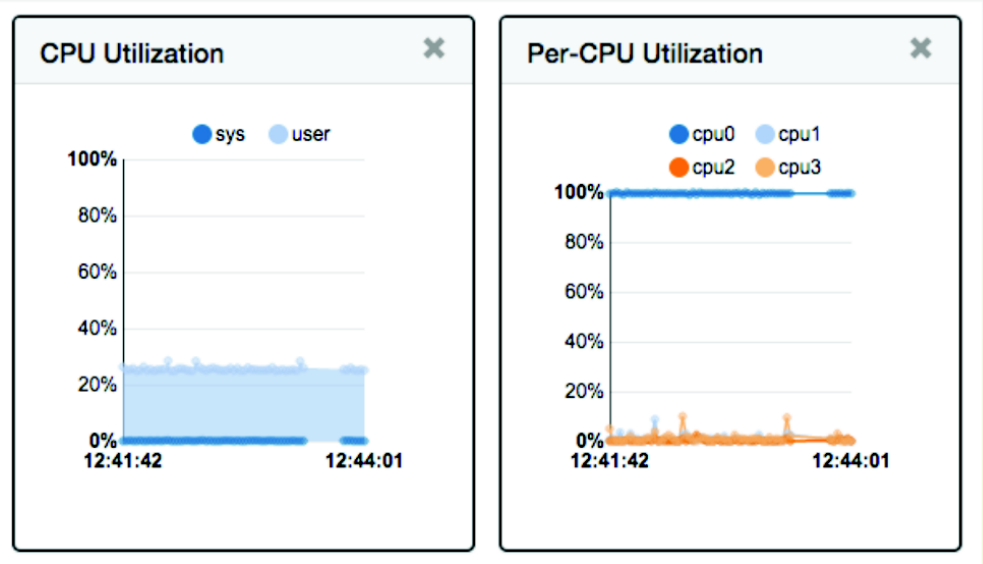


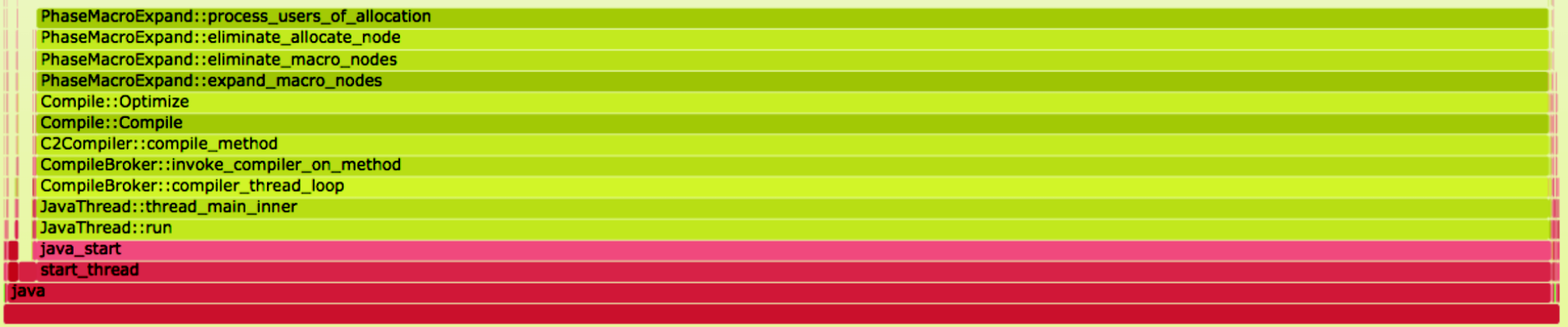
- Minor Problems:
  - 0-3% CPU overhead to enable frame pointers (usually <1%).
  - Symbol dumps can consume a burst of CPU
- **Complete** and **accurate** (asynchronous) profiling

Simple Production Example

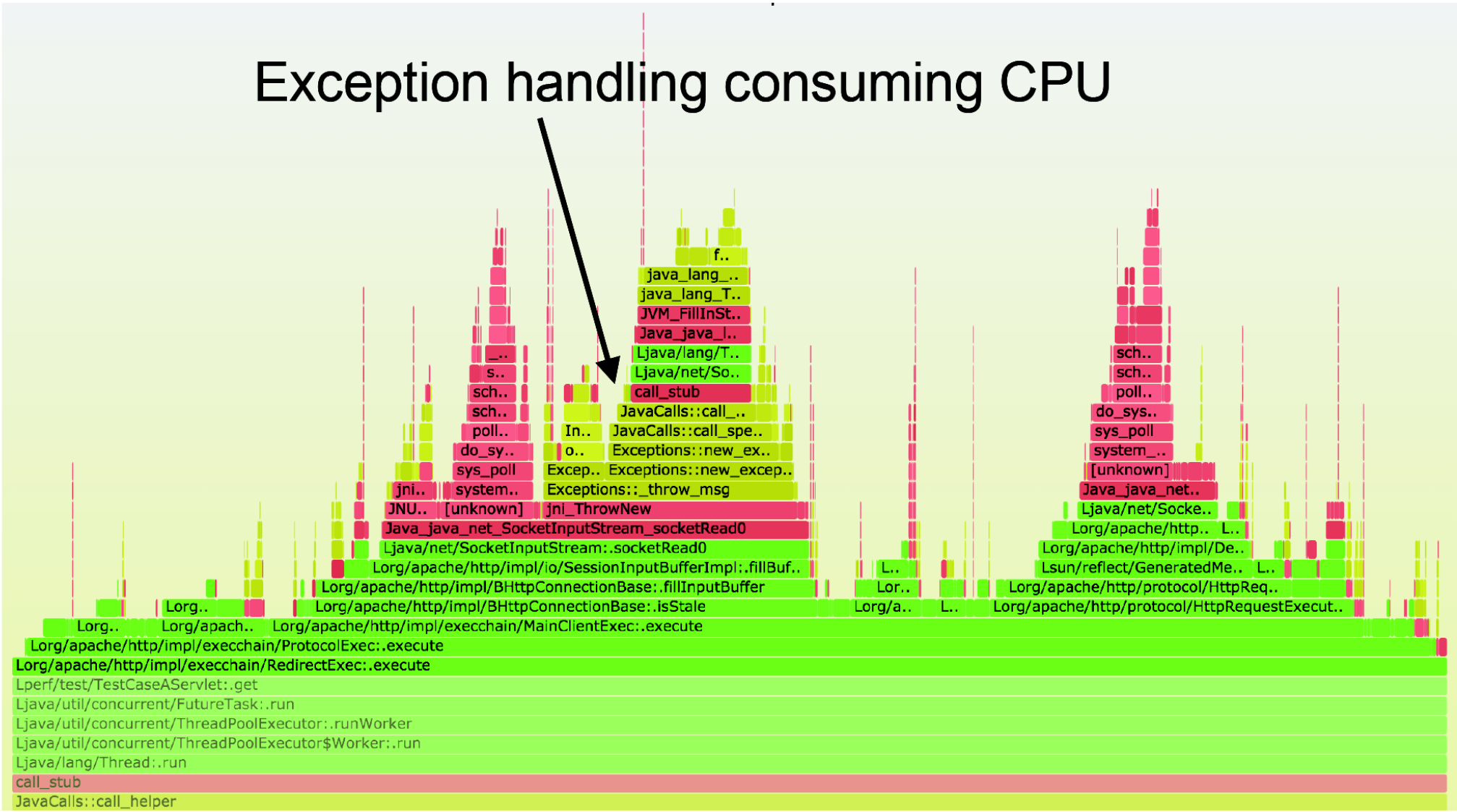
CPU Flame Graph (no idle): , 2015-02-05_20:38:52

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



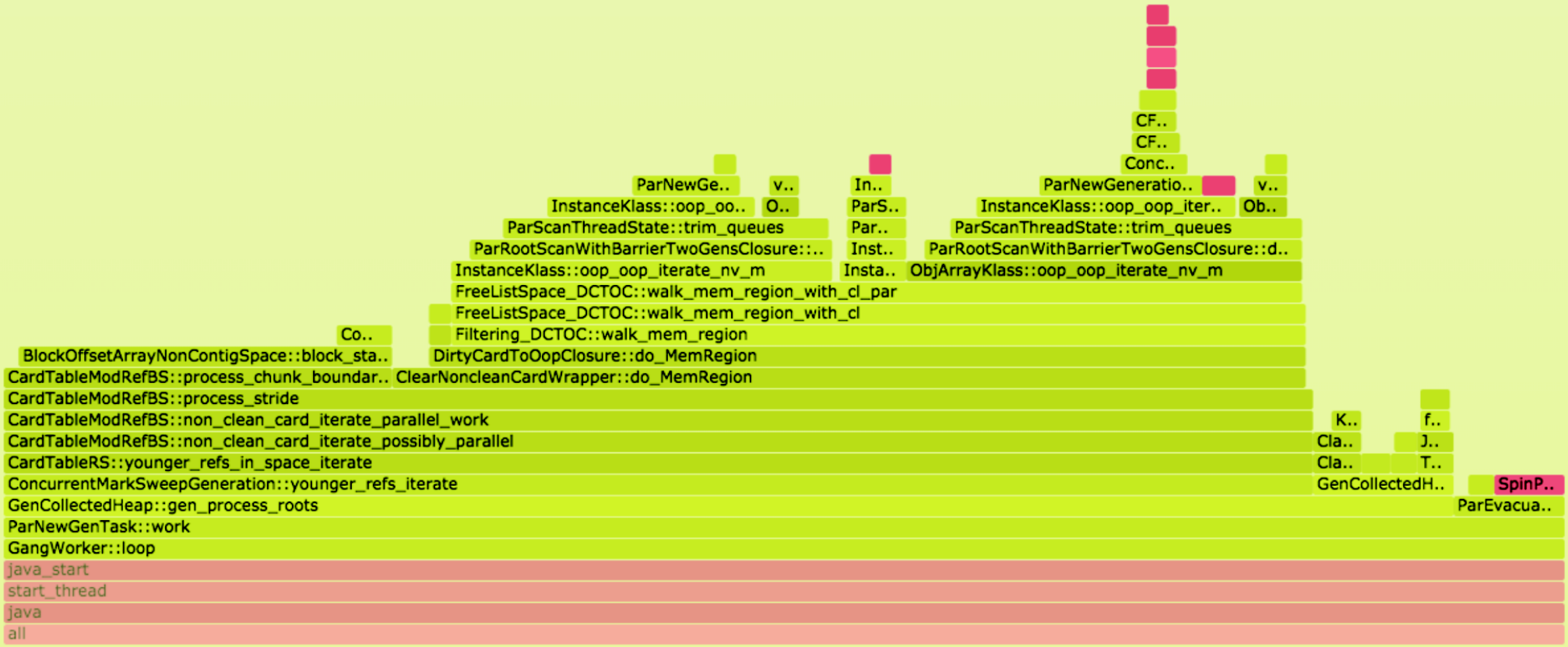


Another System Example



Profiling GC

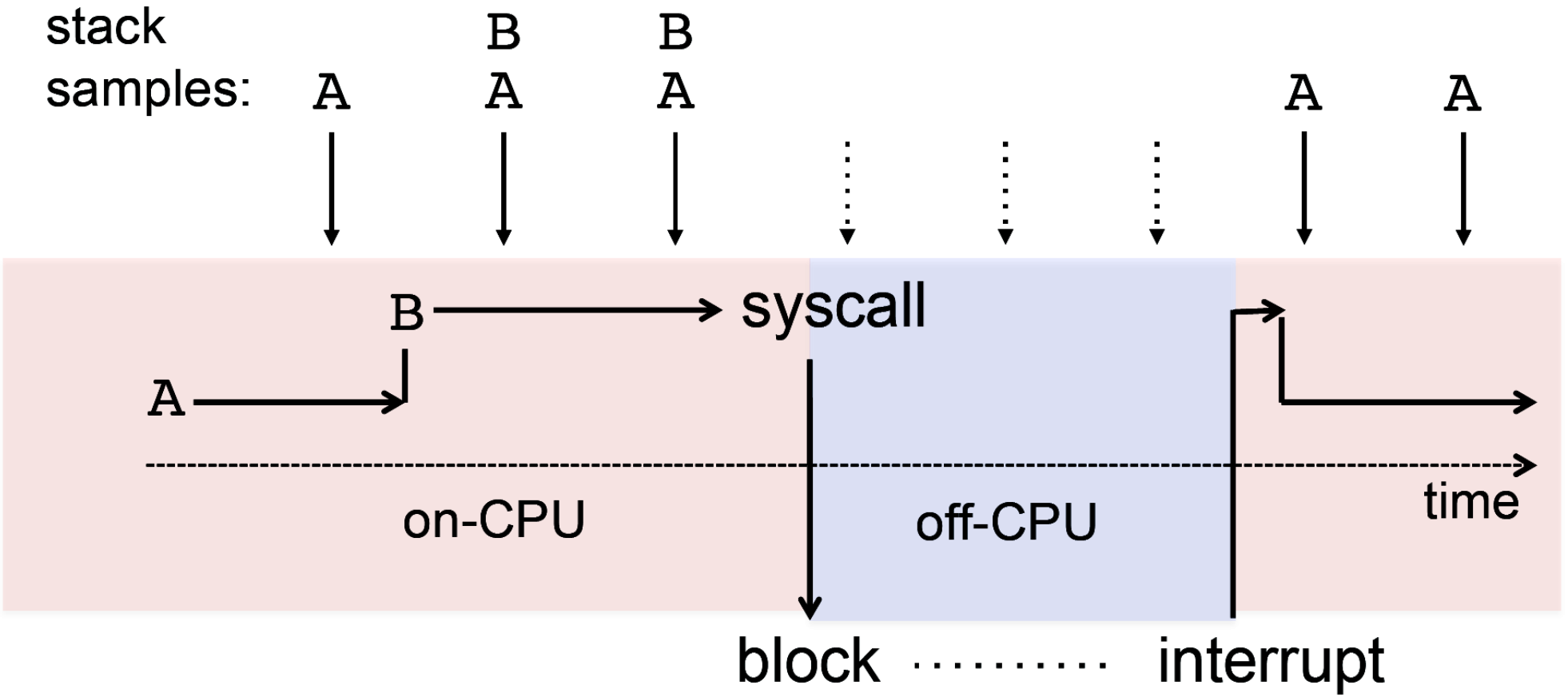
GC internals, visualized:



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)

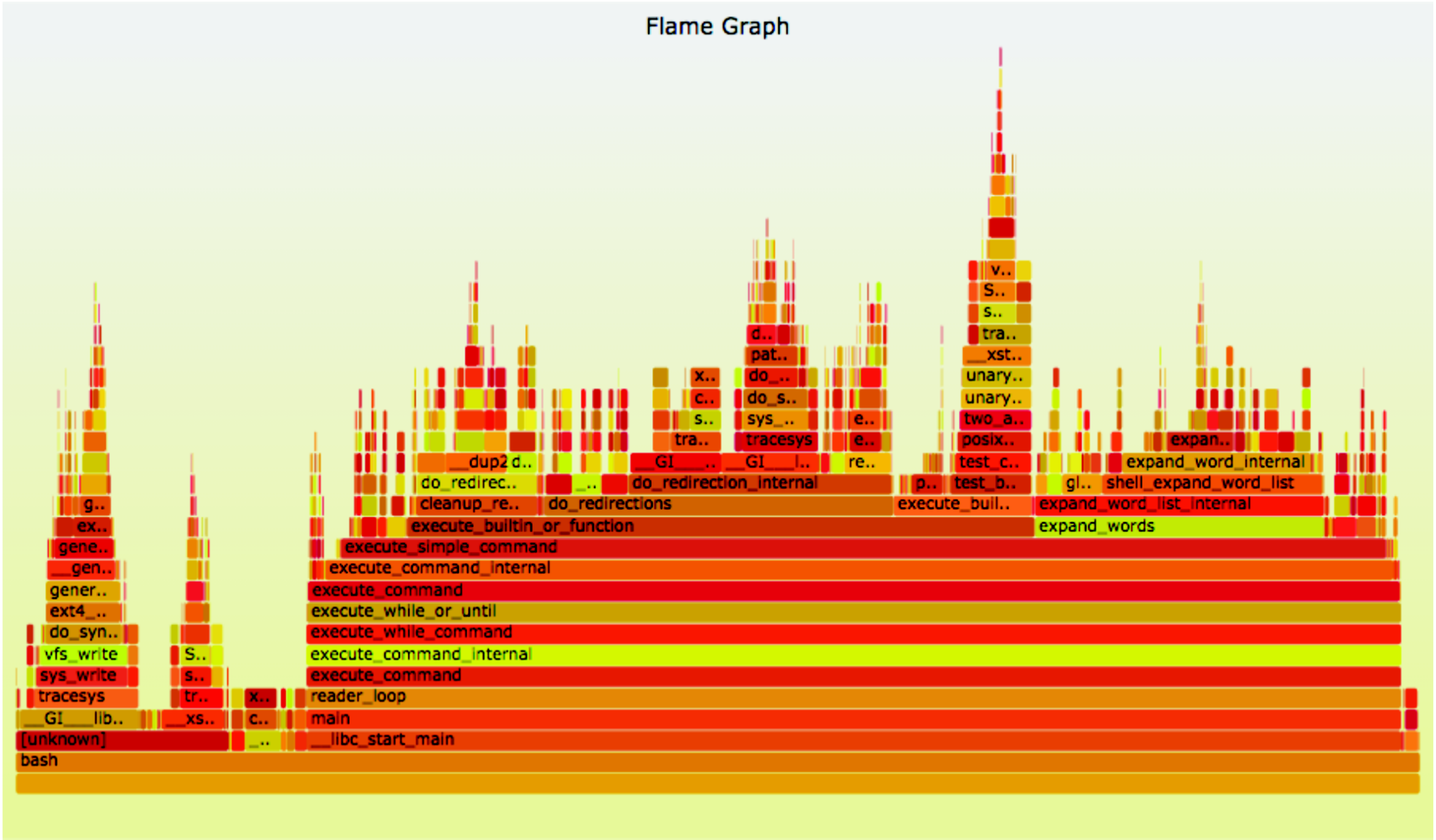
Flame Graphs

`perf report` Verbosity

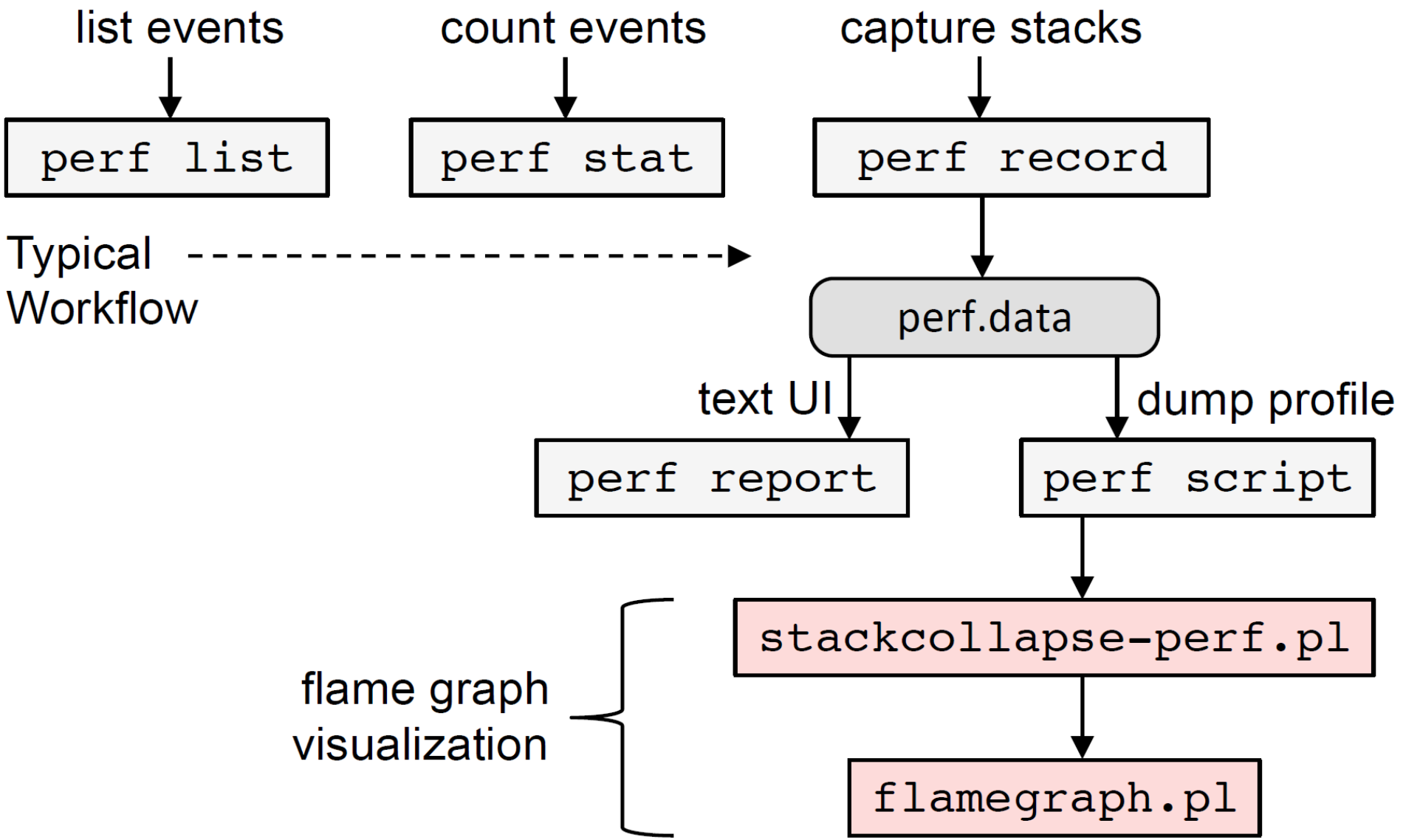
- Despite summarizing, output is still verbose

```
# 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 …]
```

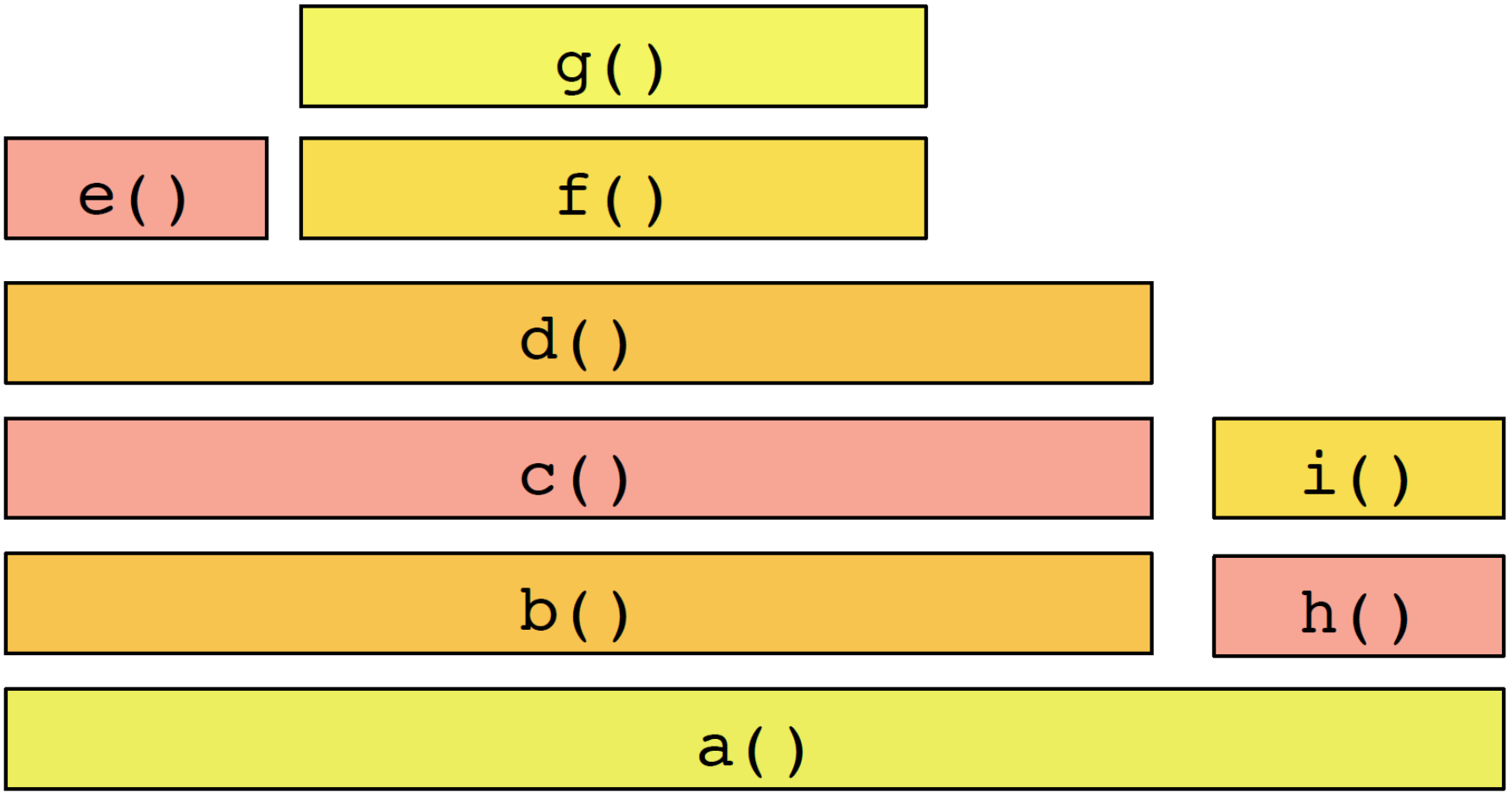
… as a Flame Graph



Linux perf_events Workflow

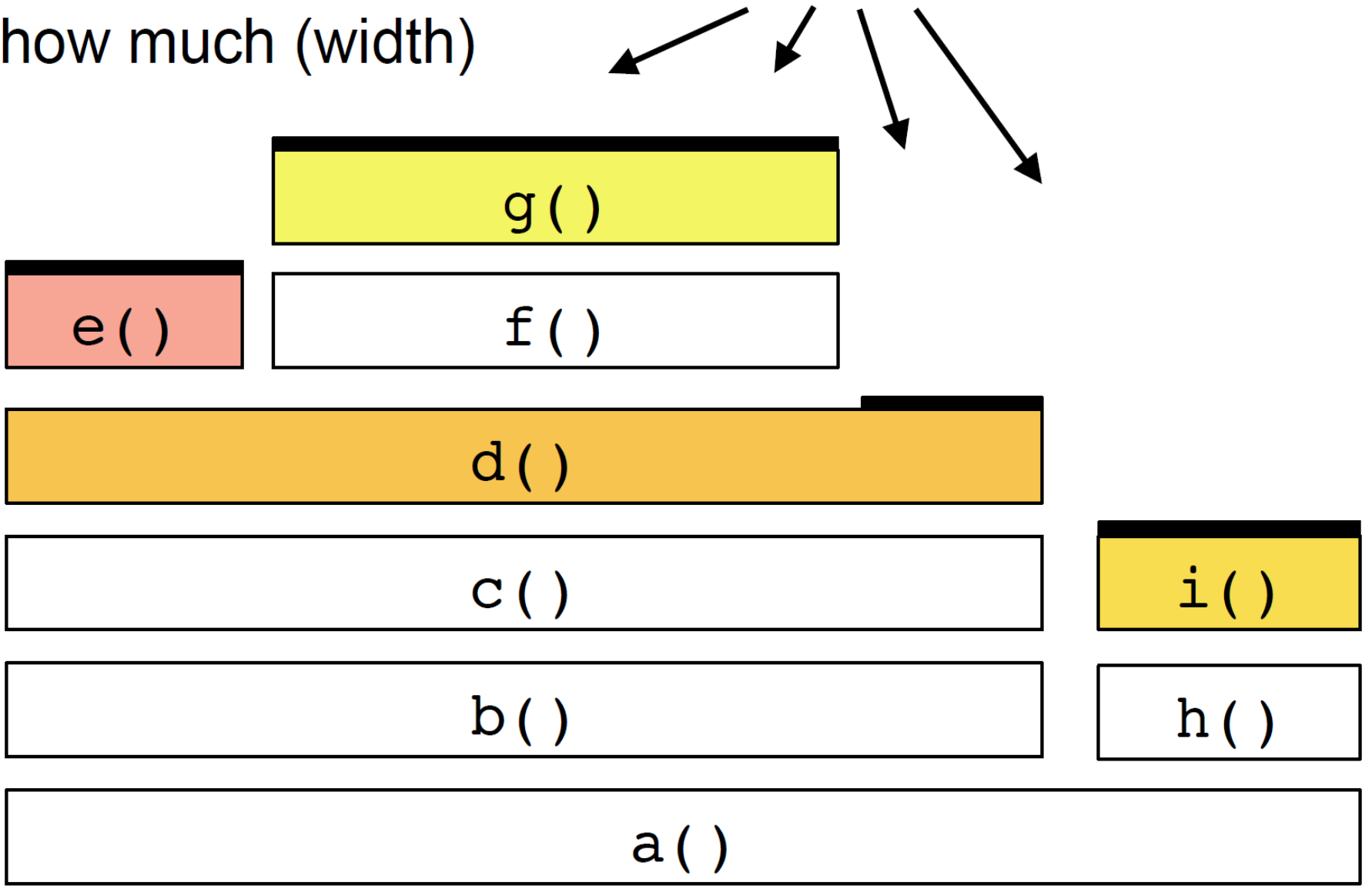


Flame Graph Interpretation



Flame Graph Interpretation (1/3)

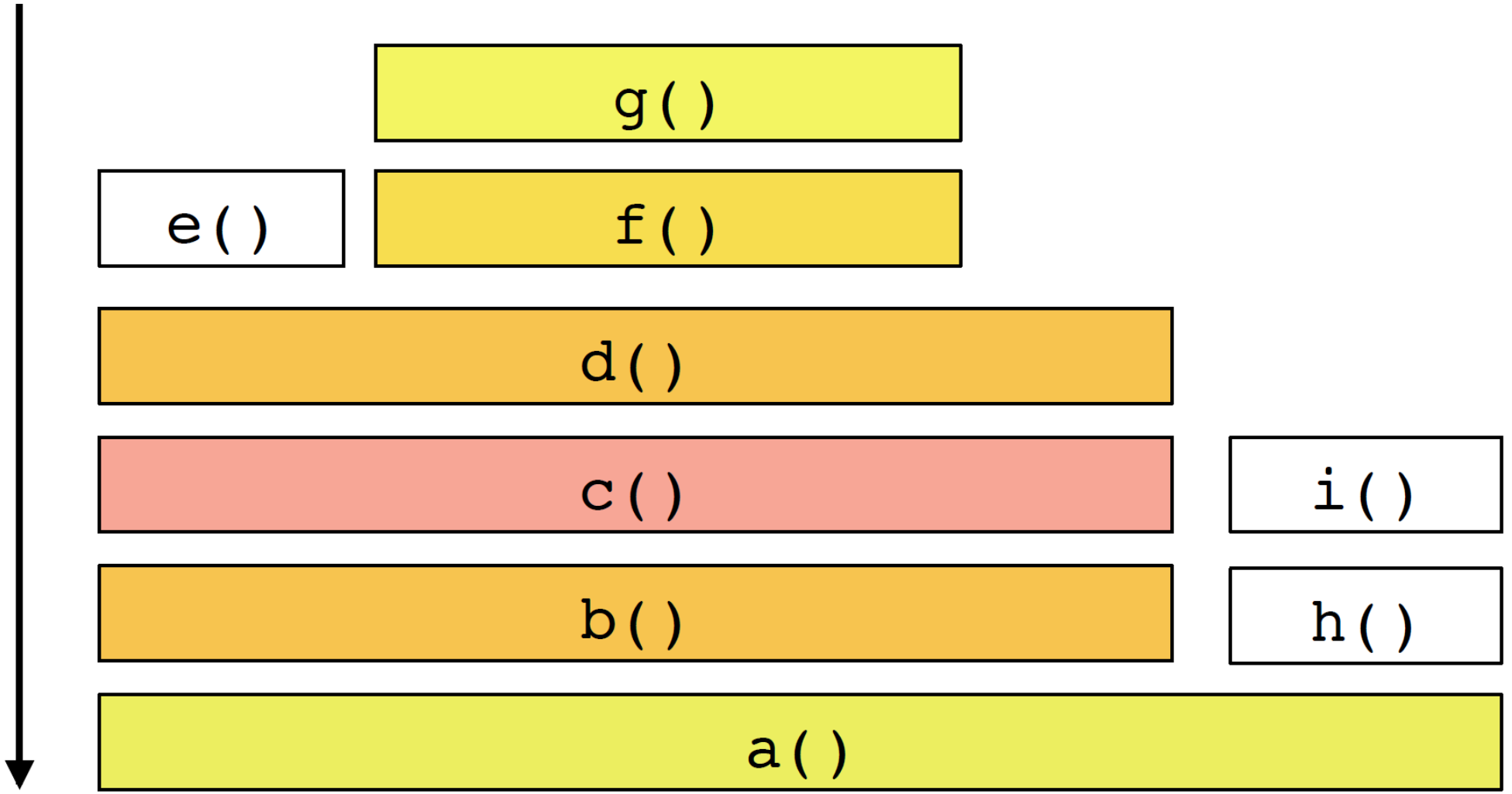
Top edge shows who is running on-CPU,
and how much (width)



Flame Graph Interpretation (2/3)

Top-down shows ancestry

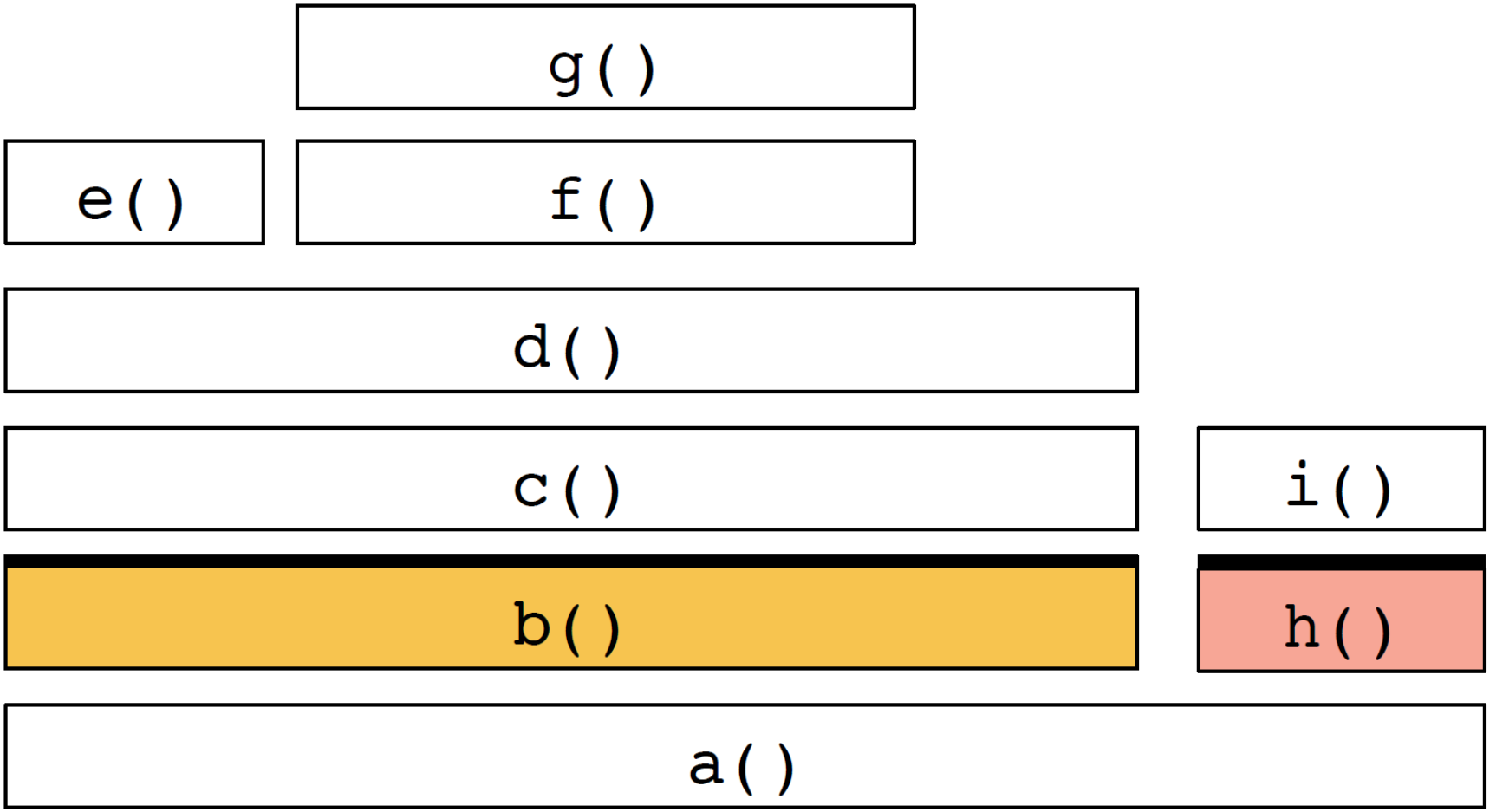
e.g., from g():



Flame Graph Interpretation (3/3)

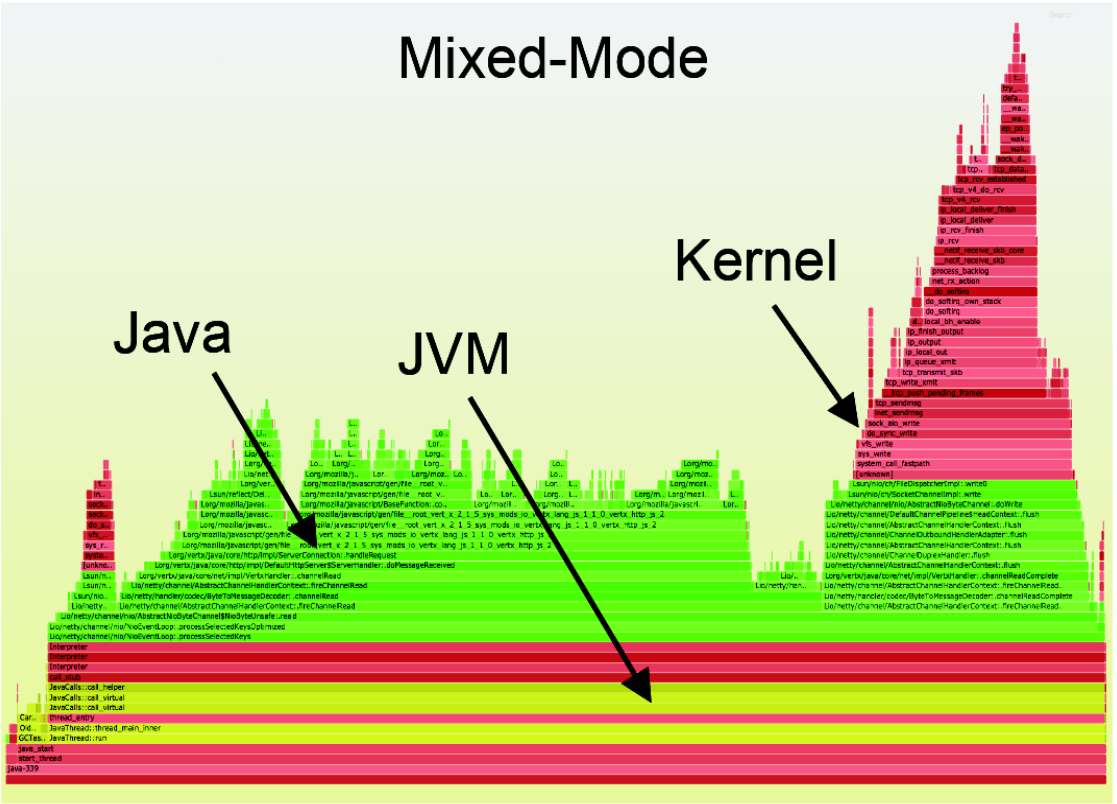
Widths are proportional to presence in samples

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



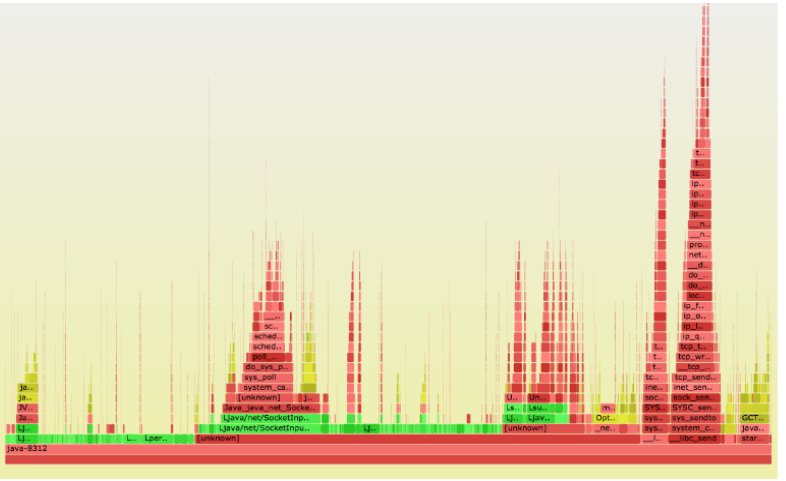
Mixed-Mode Flame Graphs

- Hues:
  - green == Java
  - red == system
  - yellow == C++
- Intensity randomized to differentiate frames
  - Or hashed based on function name



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…

Call Chain Example

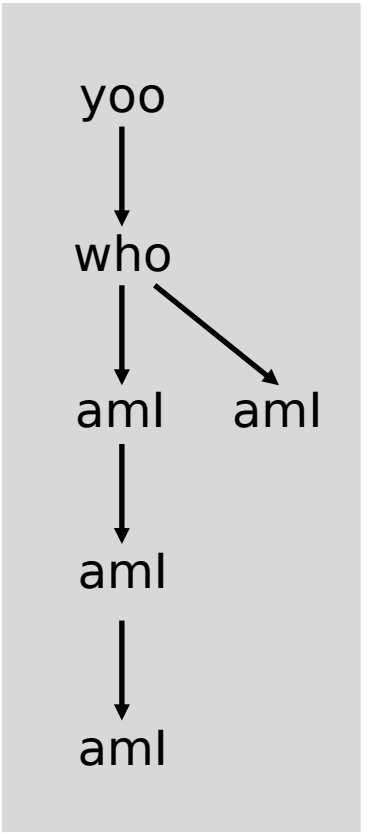
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

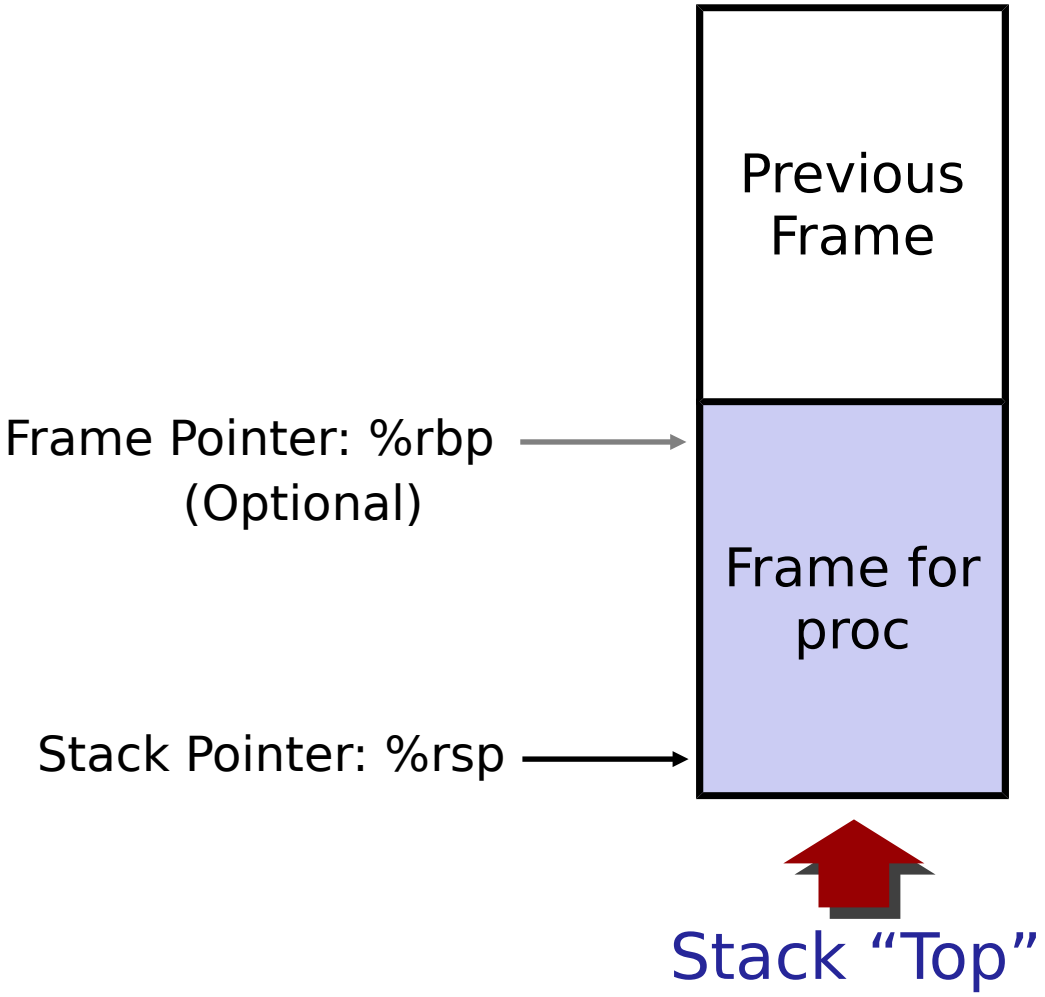
Procedure aml() is recursive

Example
Call Chain

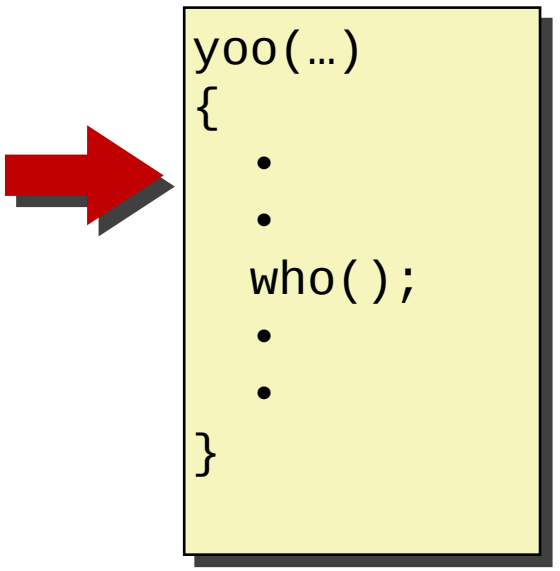


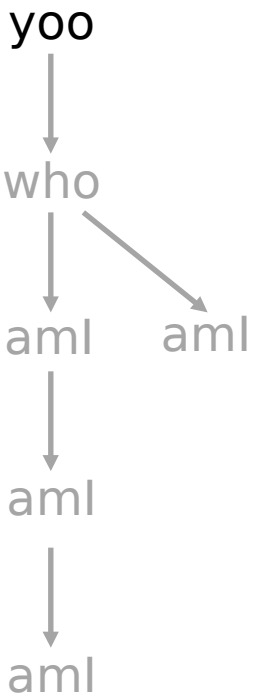
Stack Frames

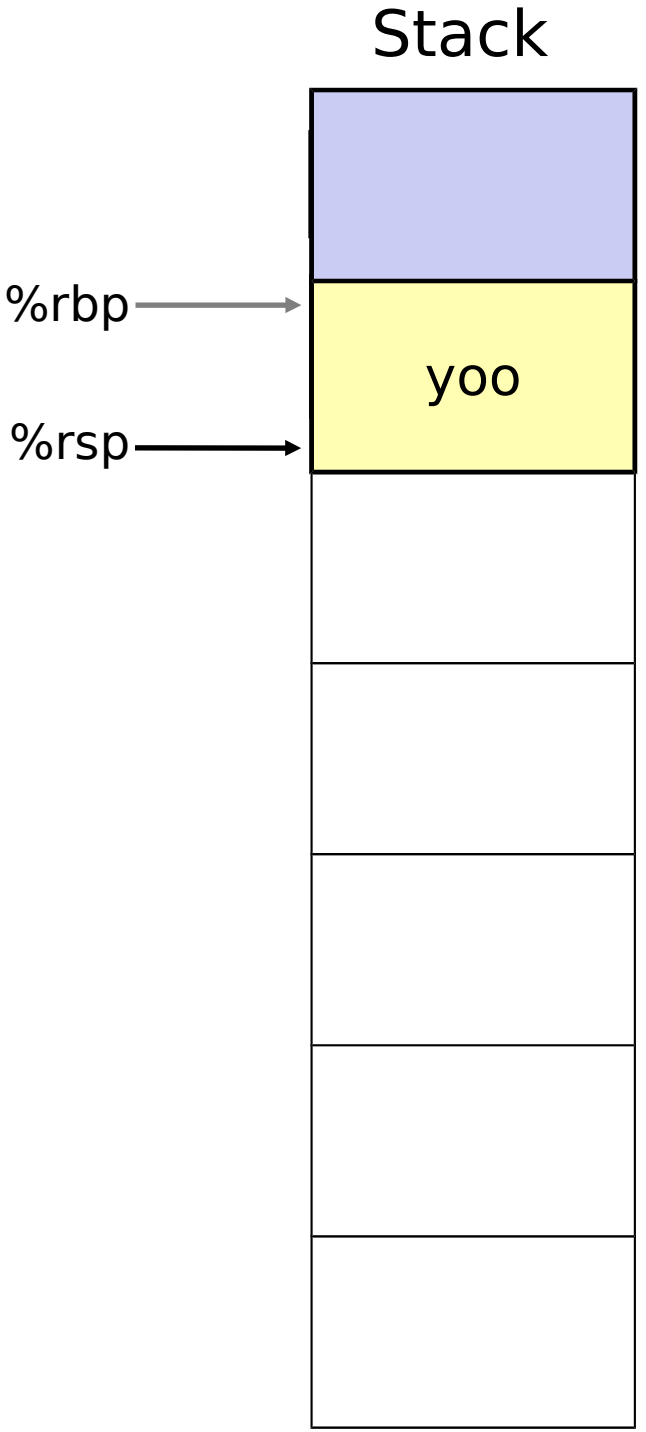
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- **Management**
  - Space allocated when enter procedure
    - “Set-up” code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - “Finish” code
    - Includes pop by **`ret`** instruction



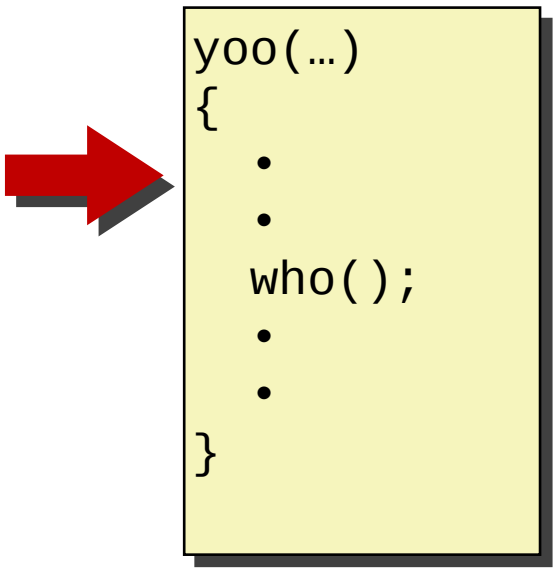
Example

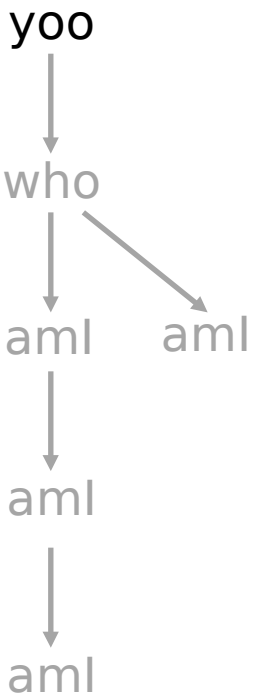


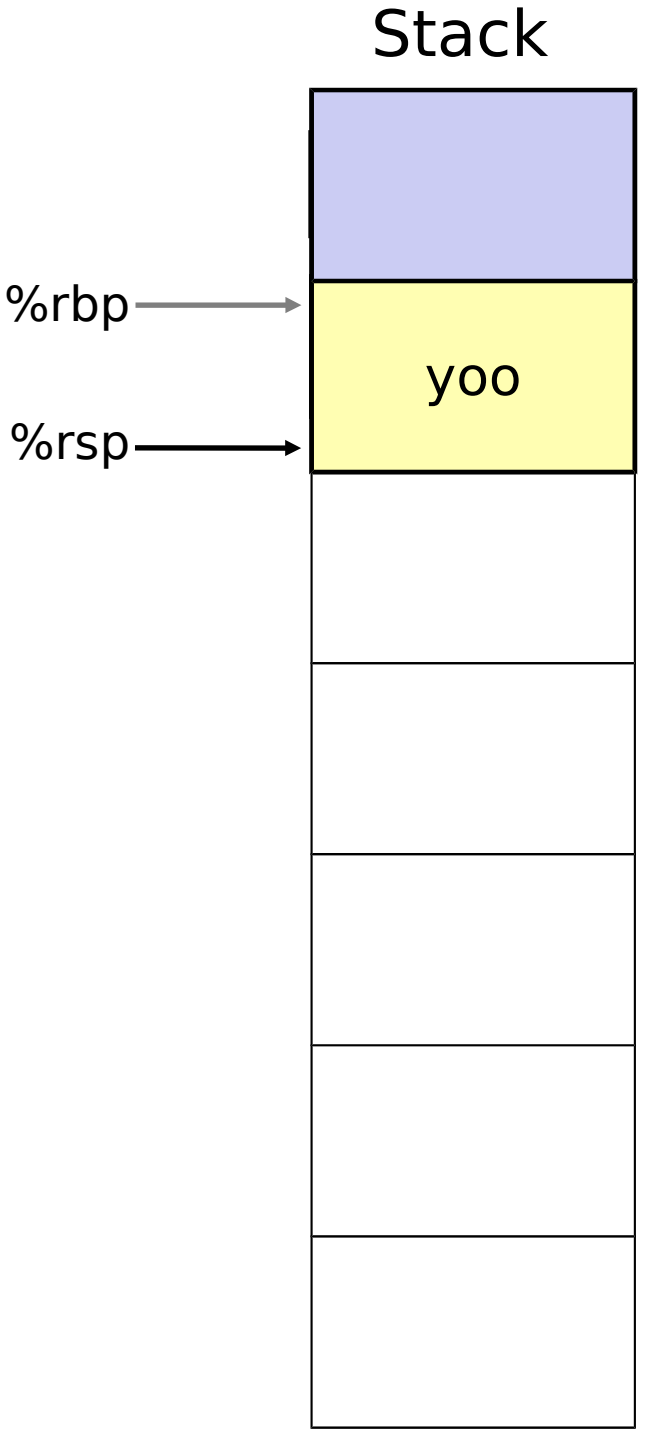




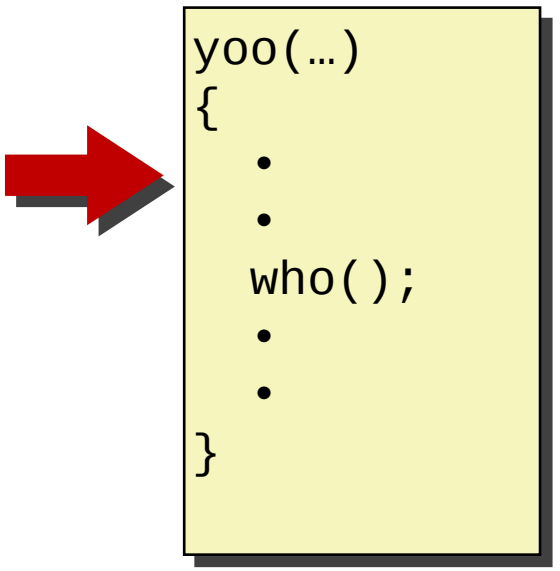
Example

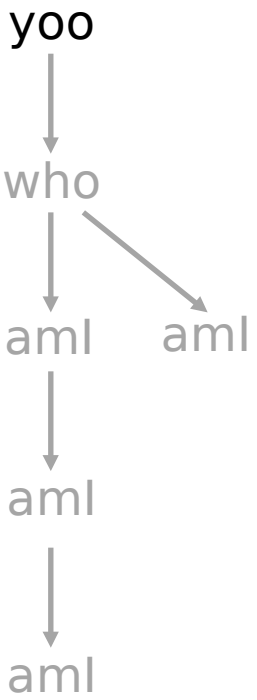


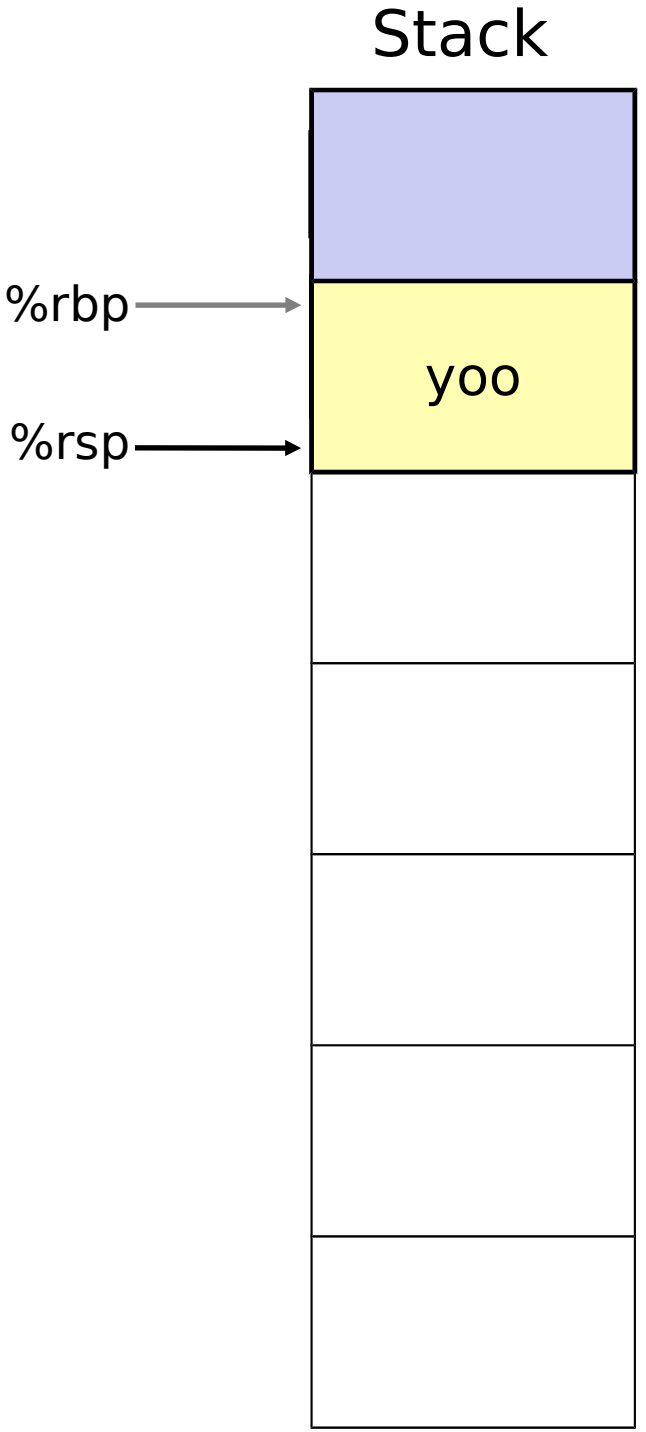




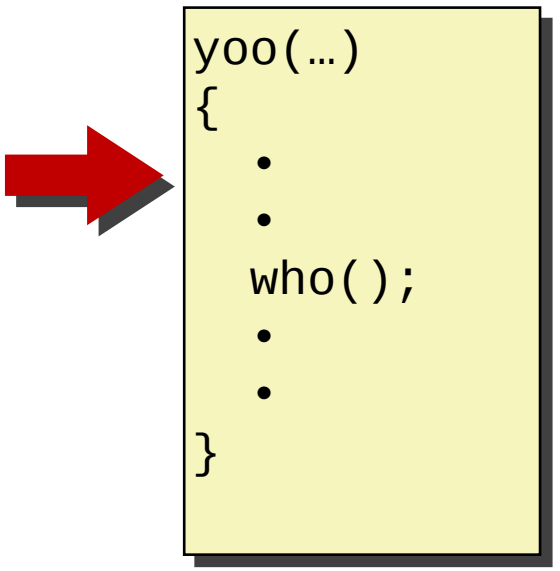
Example

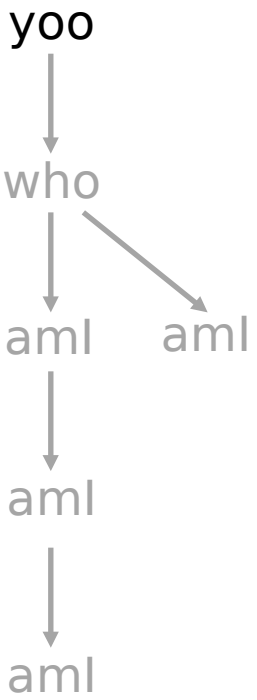


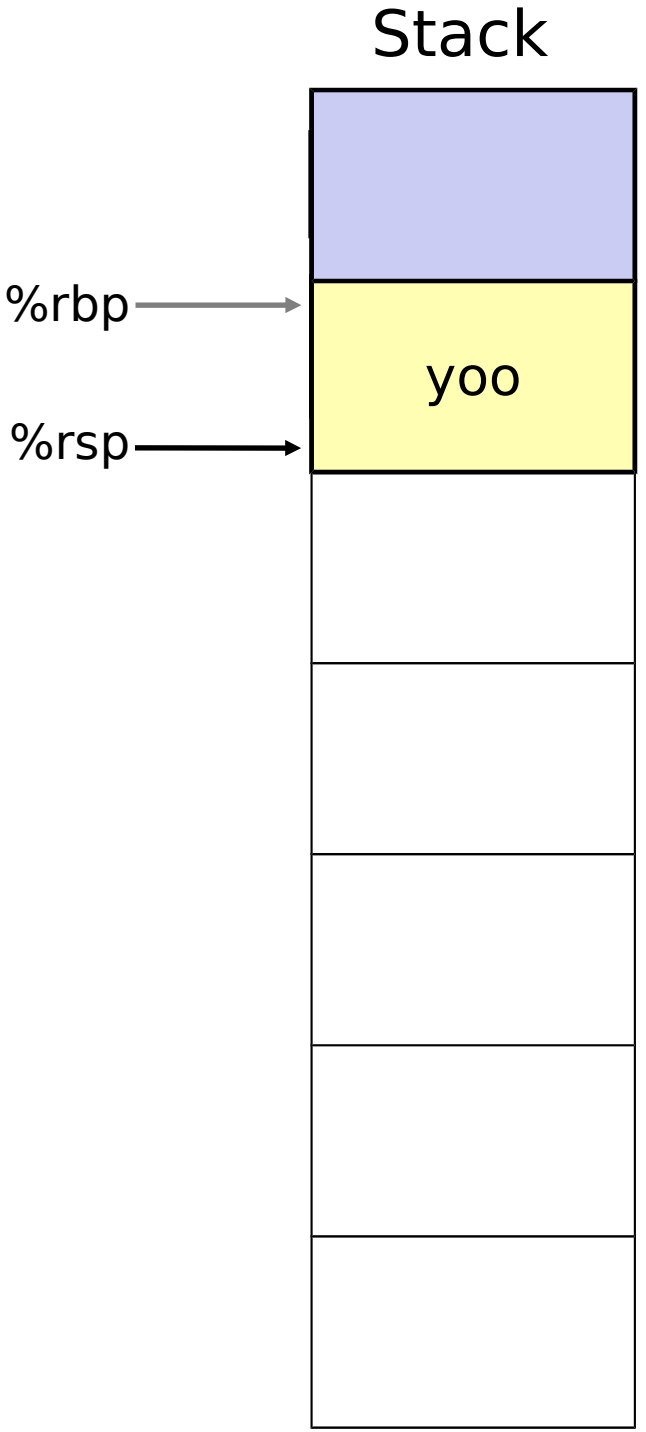




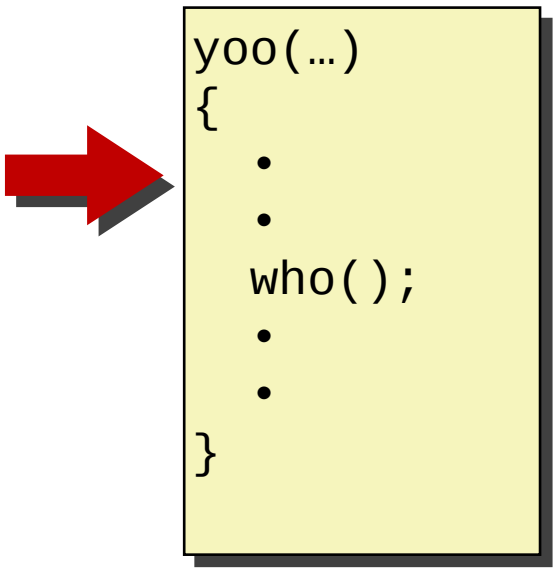
Example

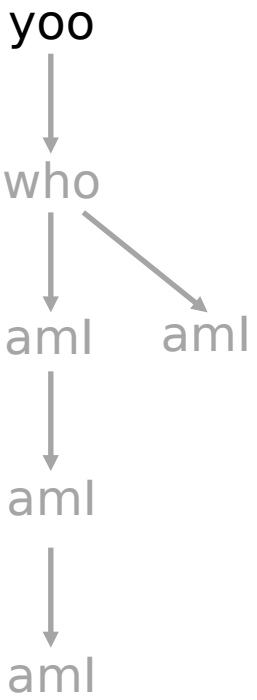


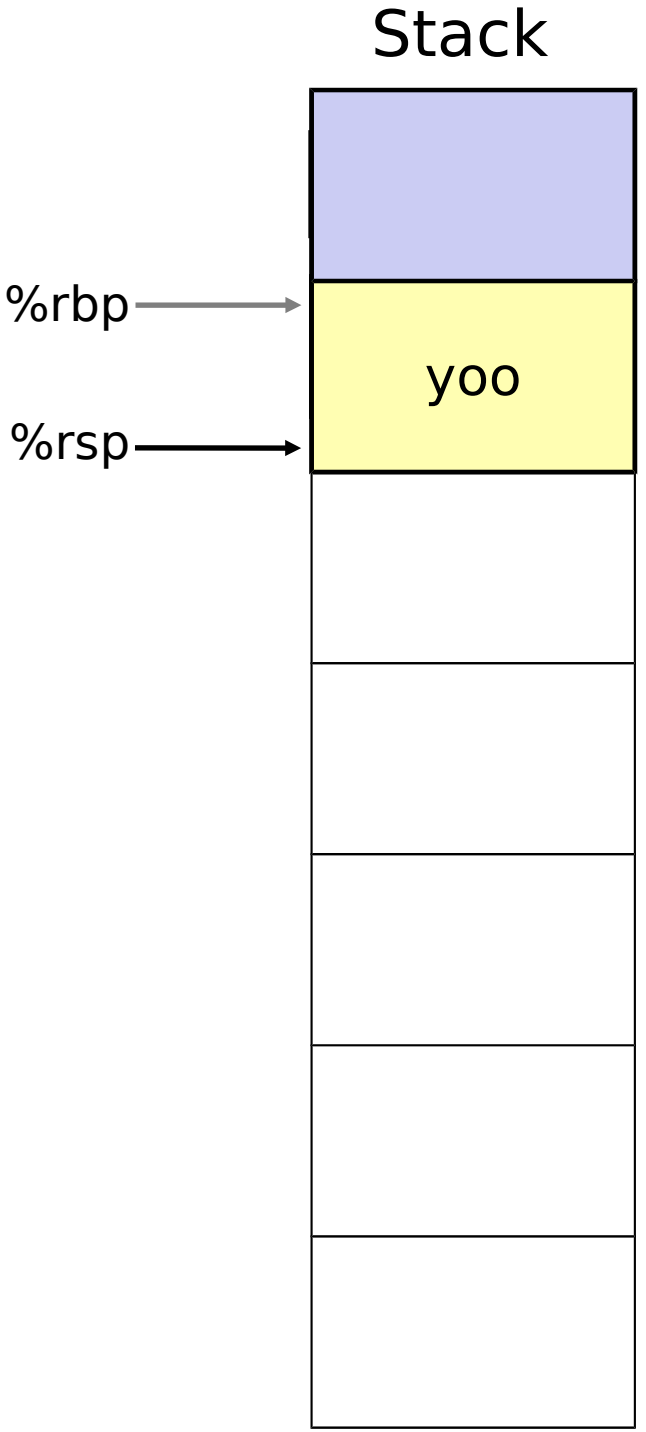




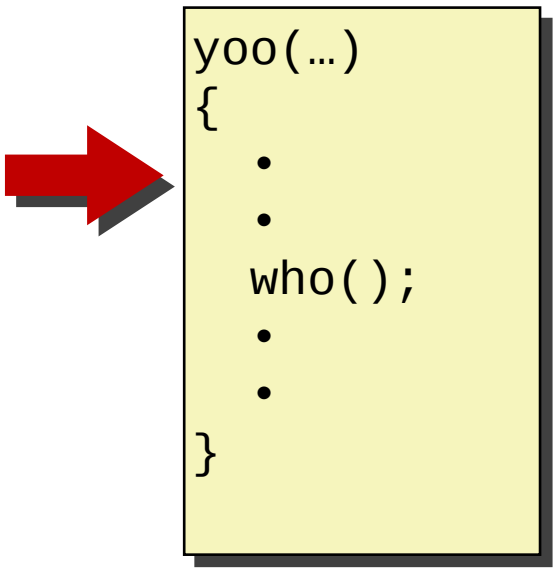
Example

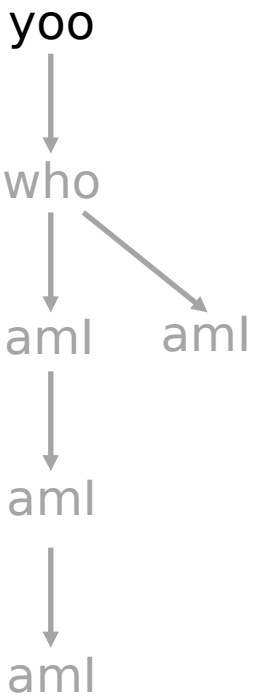


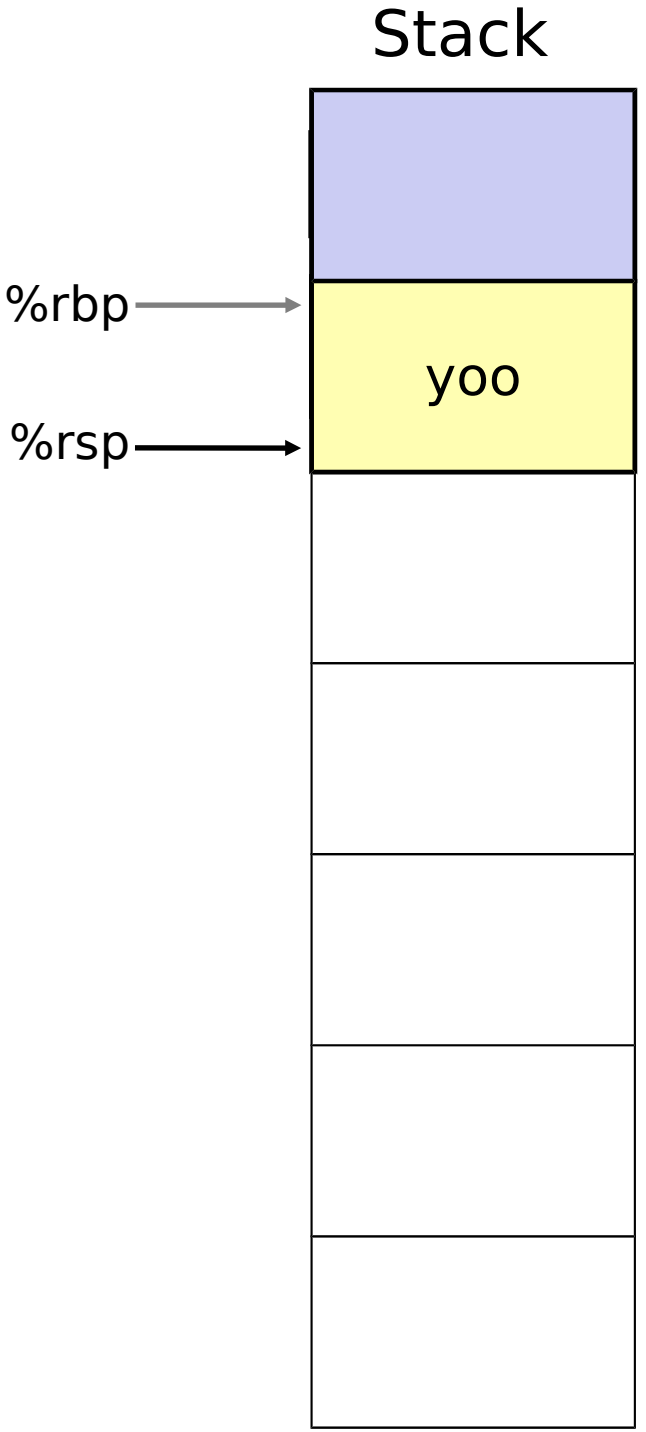




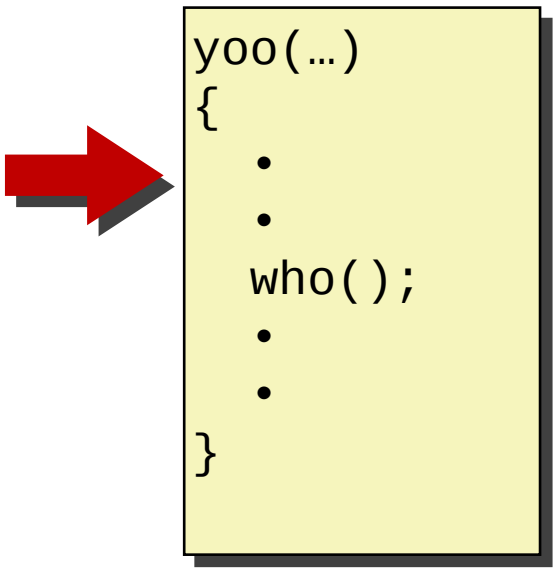
Example

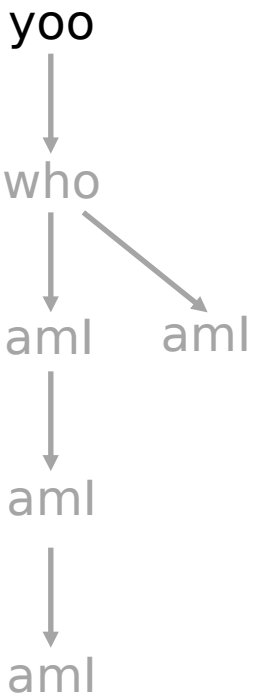


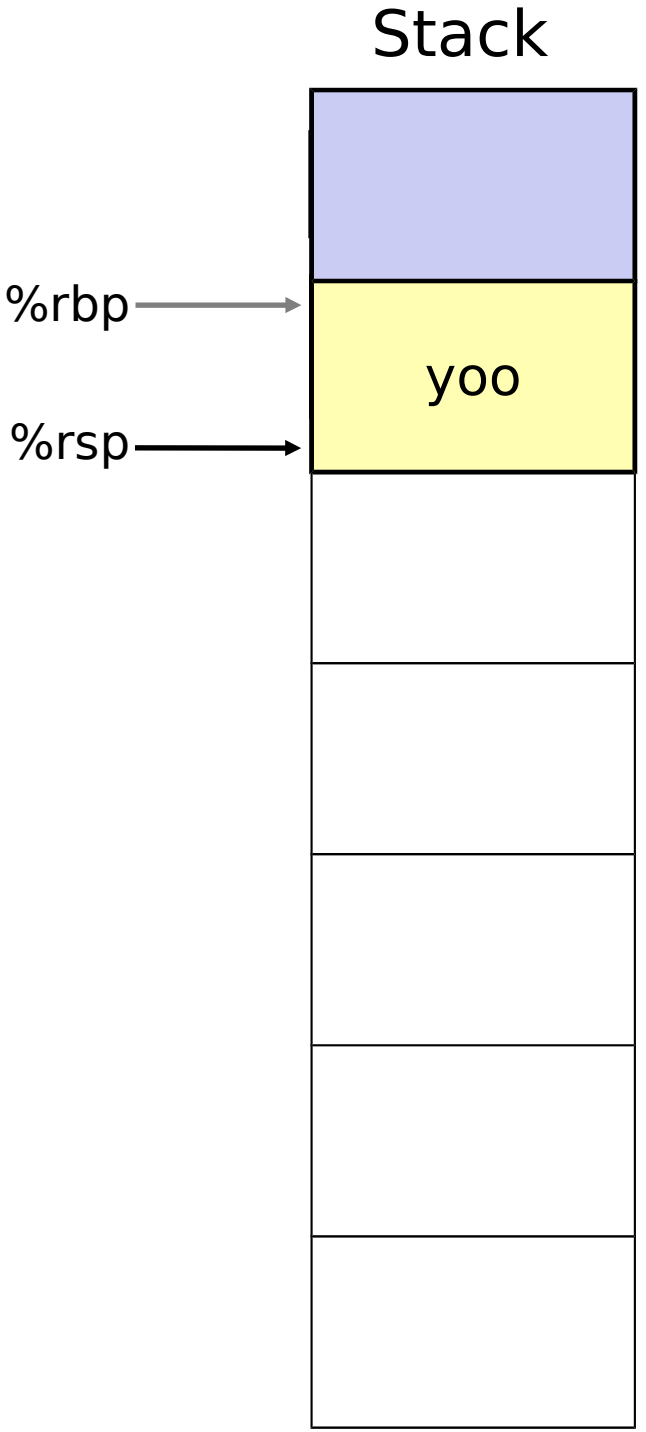




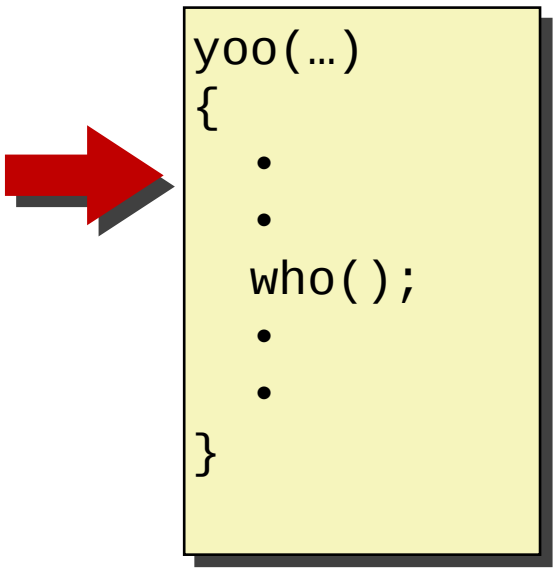
Example

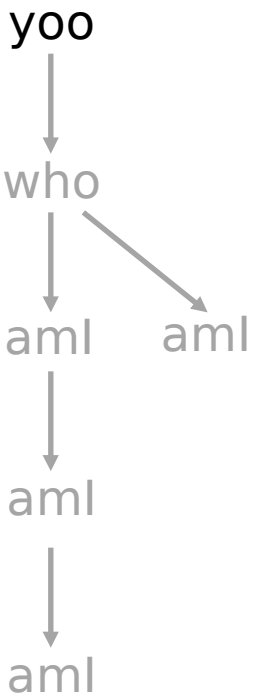


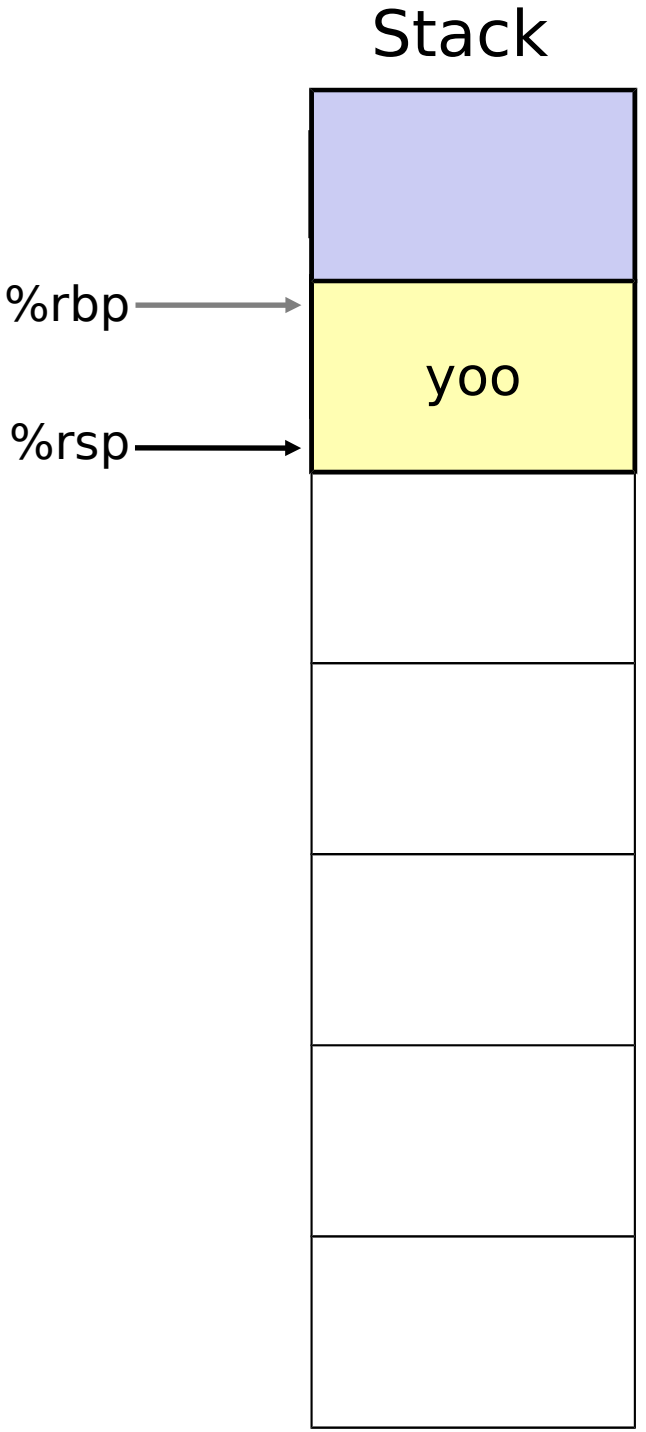




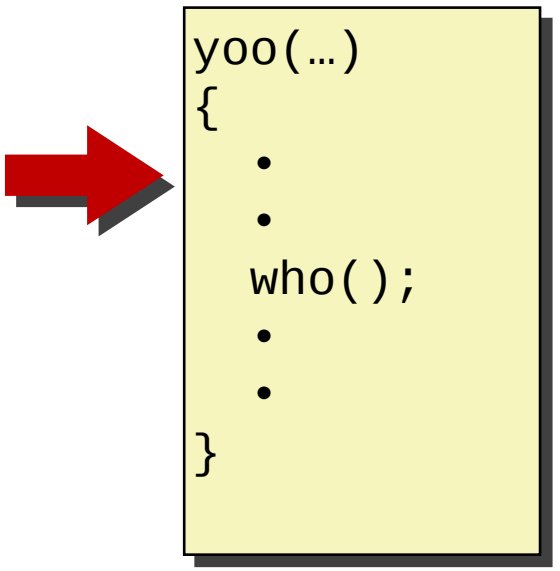
Example

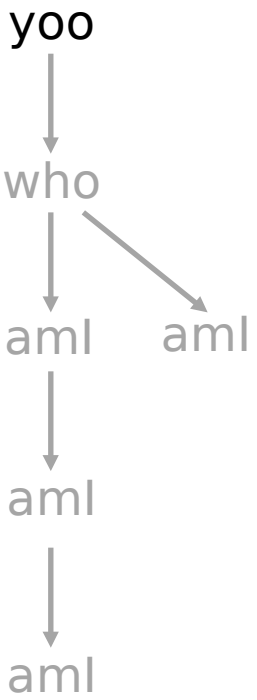


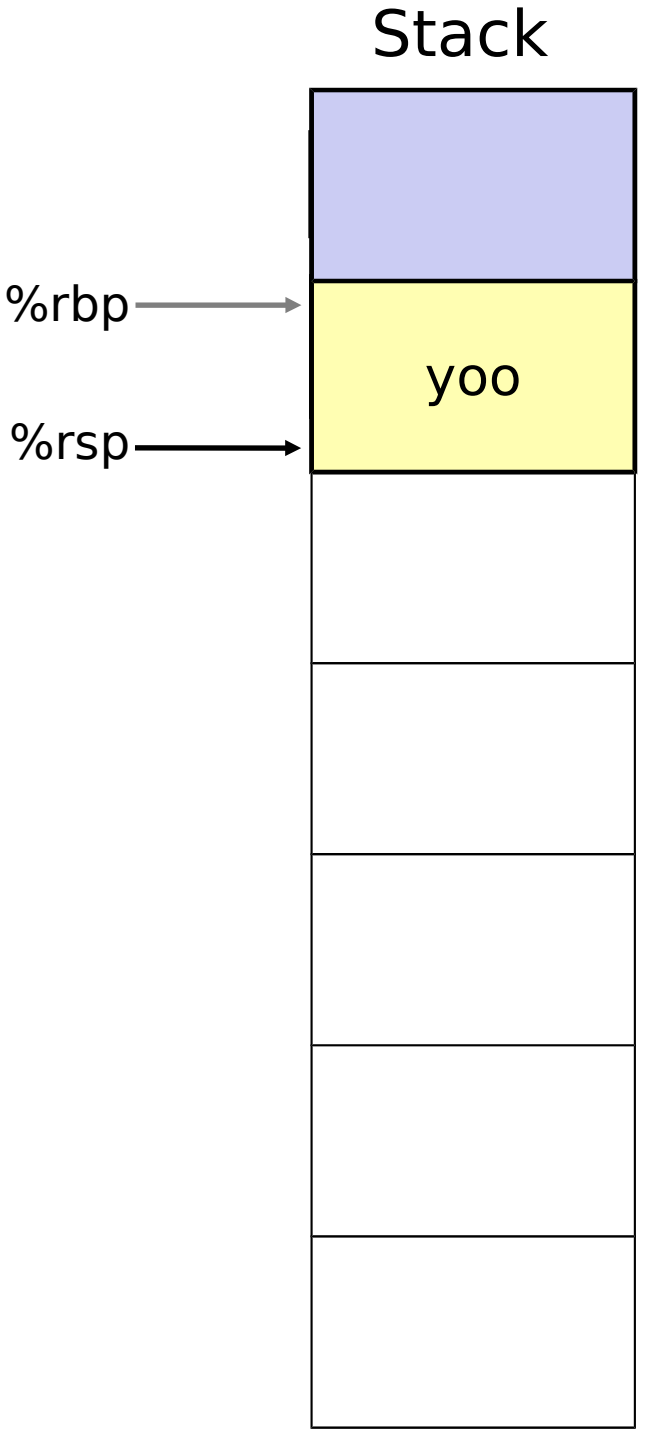




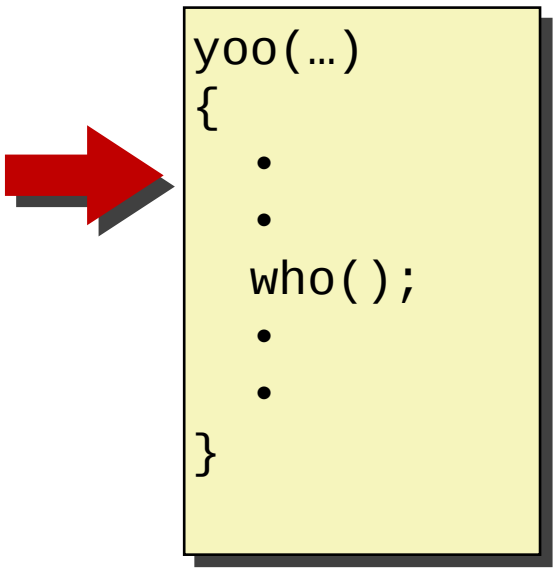
Example

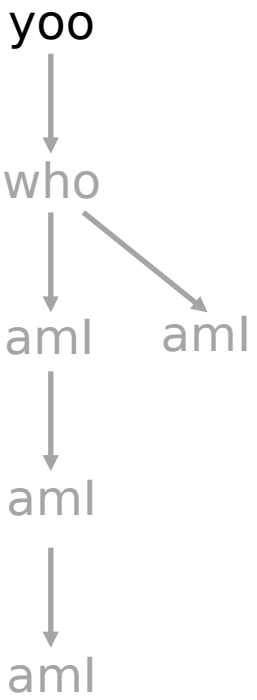


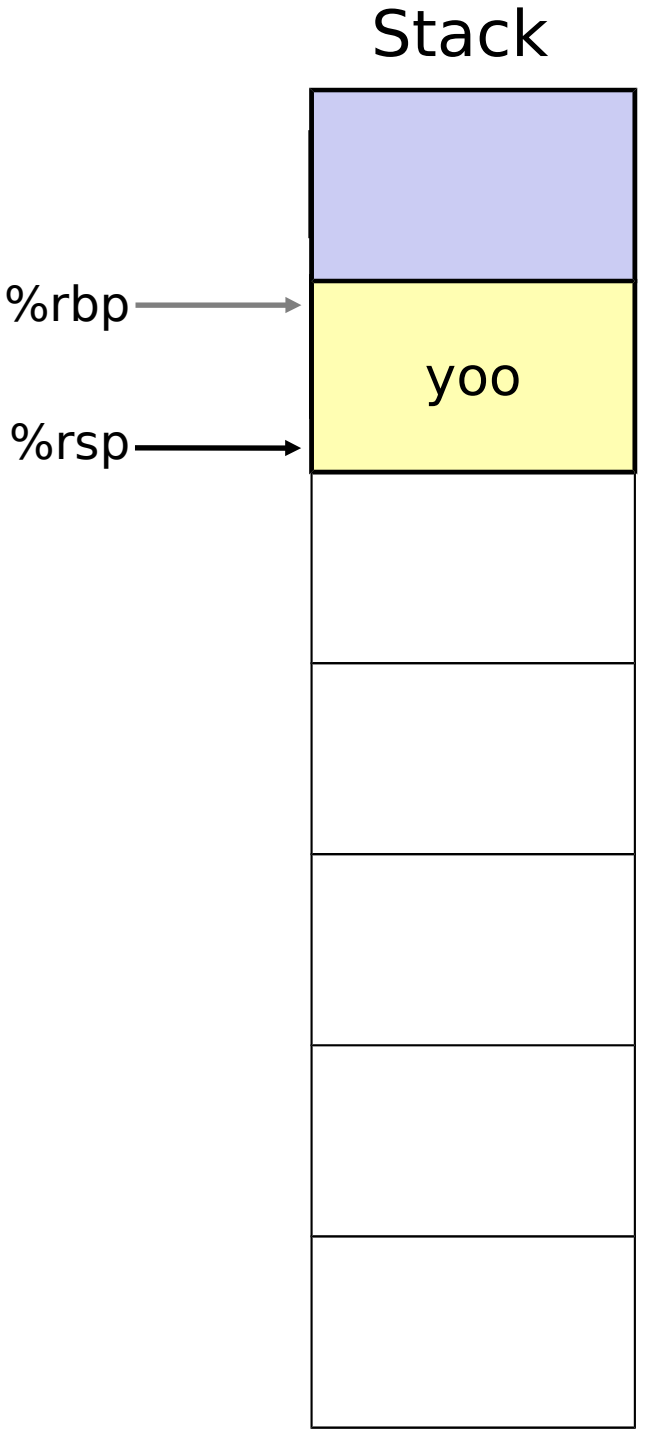




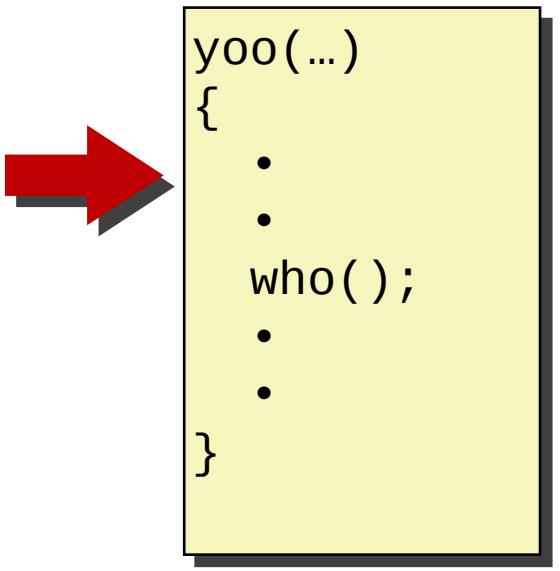
Example

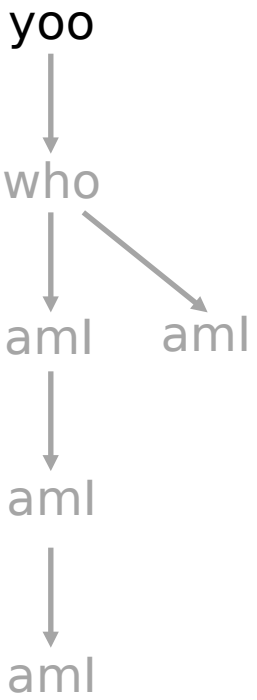


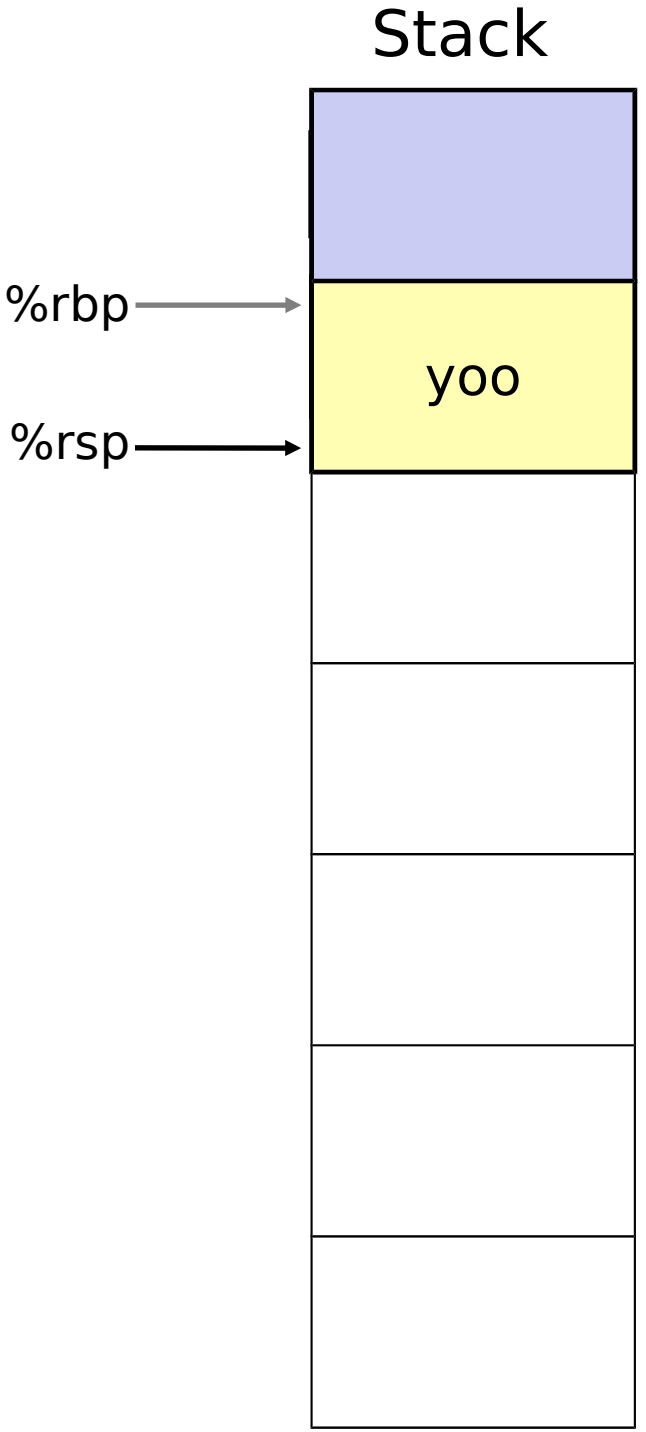




Example

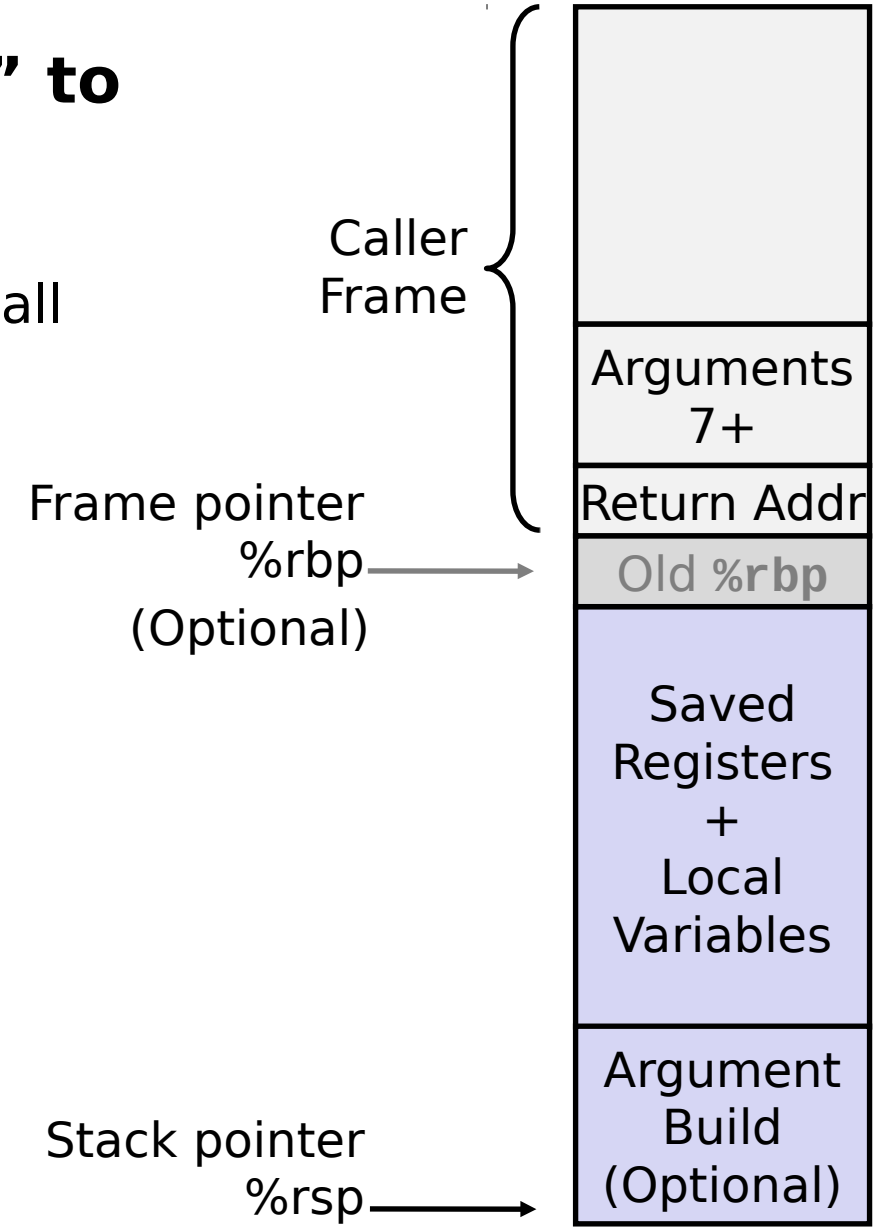






x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- **Caller Stack Frame**
  - Return address
    - Pushed by call instruction
  - Arguments for this call



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)

-XX:+PreserveFramePointer

- We shared our patch publicly
  - See "A hotspot patch for stack profiling (frame pointer)" on the hotspot complier 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)
[…]
```

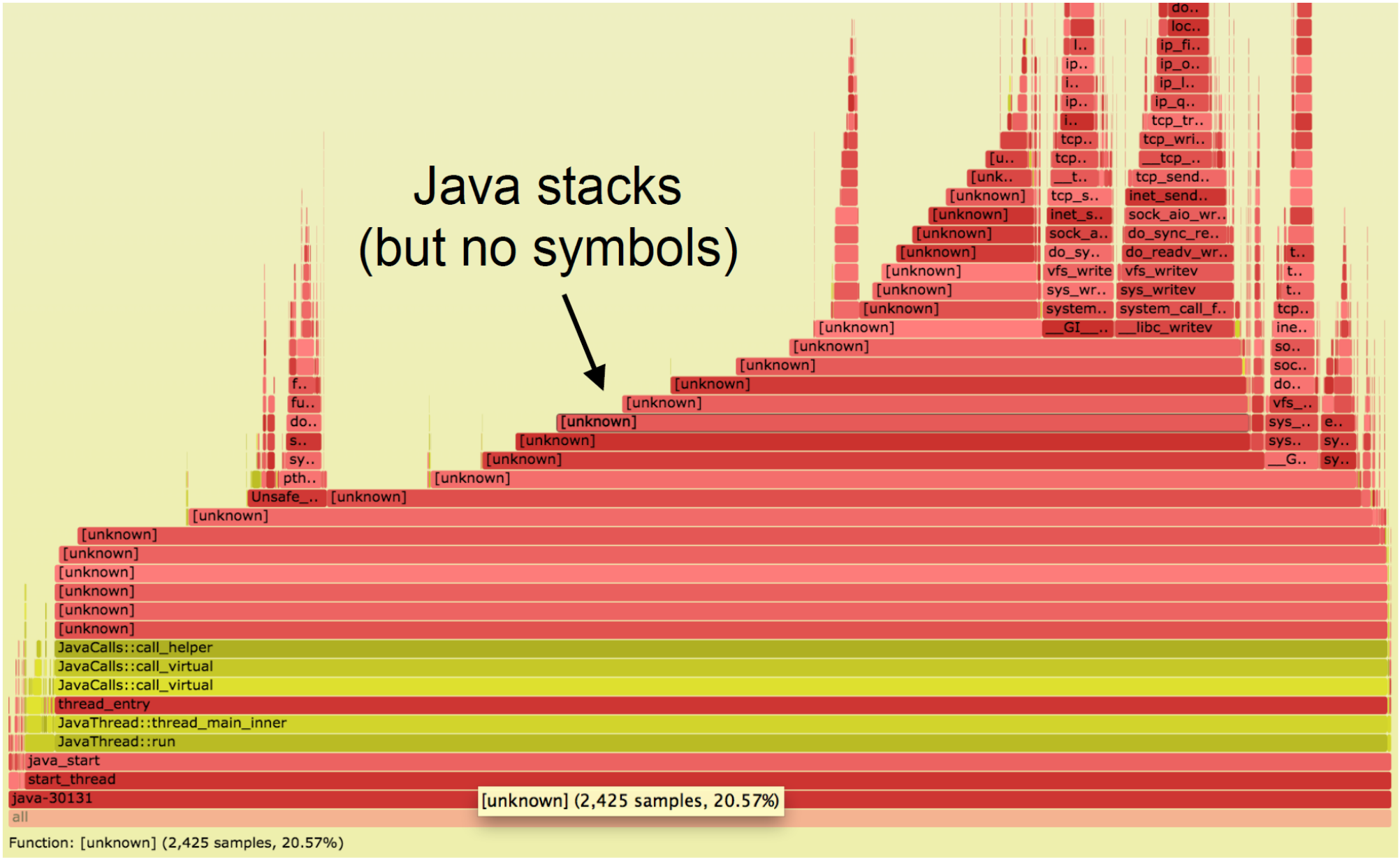
- Check with "perf script" to see stack samples
- These are 1 or 2 levels deep (junk values)

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…
```

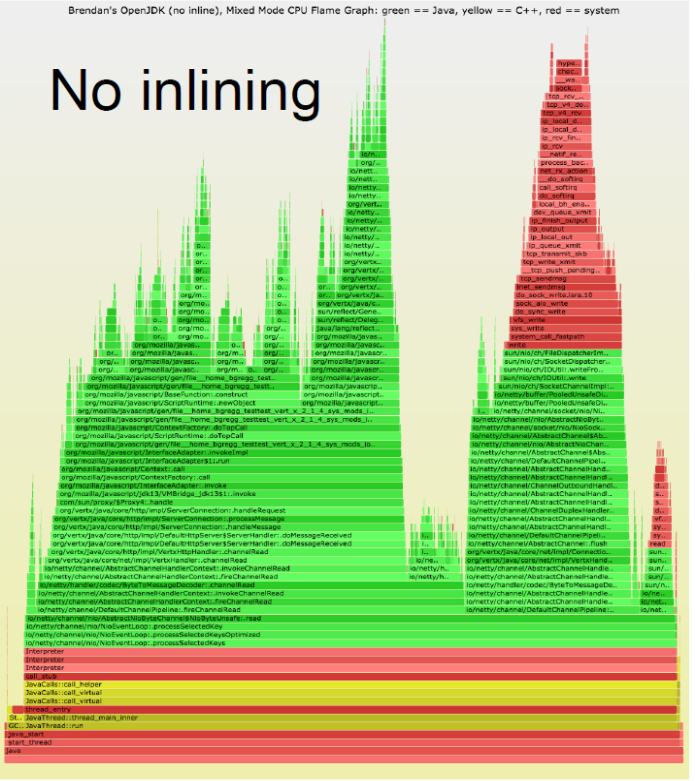
- With -XX: +PreserveFramePointer stacks are full, and go all the way to start_thread()
- This is what the CPUs are really running: inlined frames are not present

Fixed Stacks Flame Graph



Stacks & Inlining

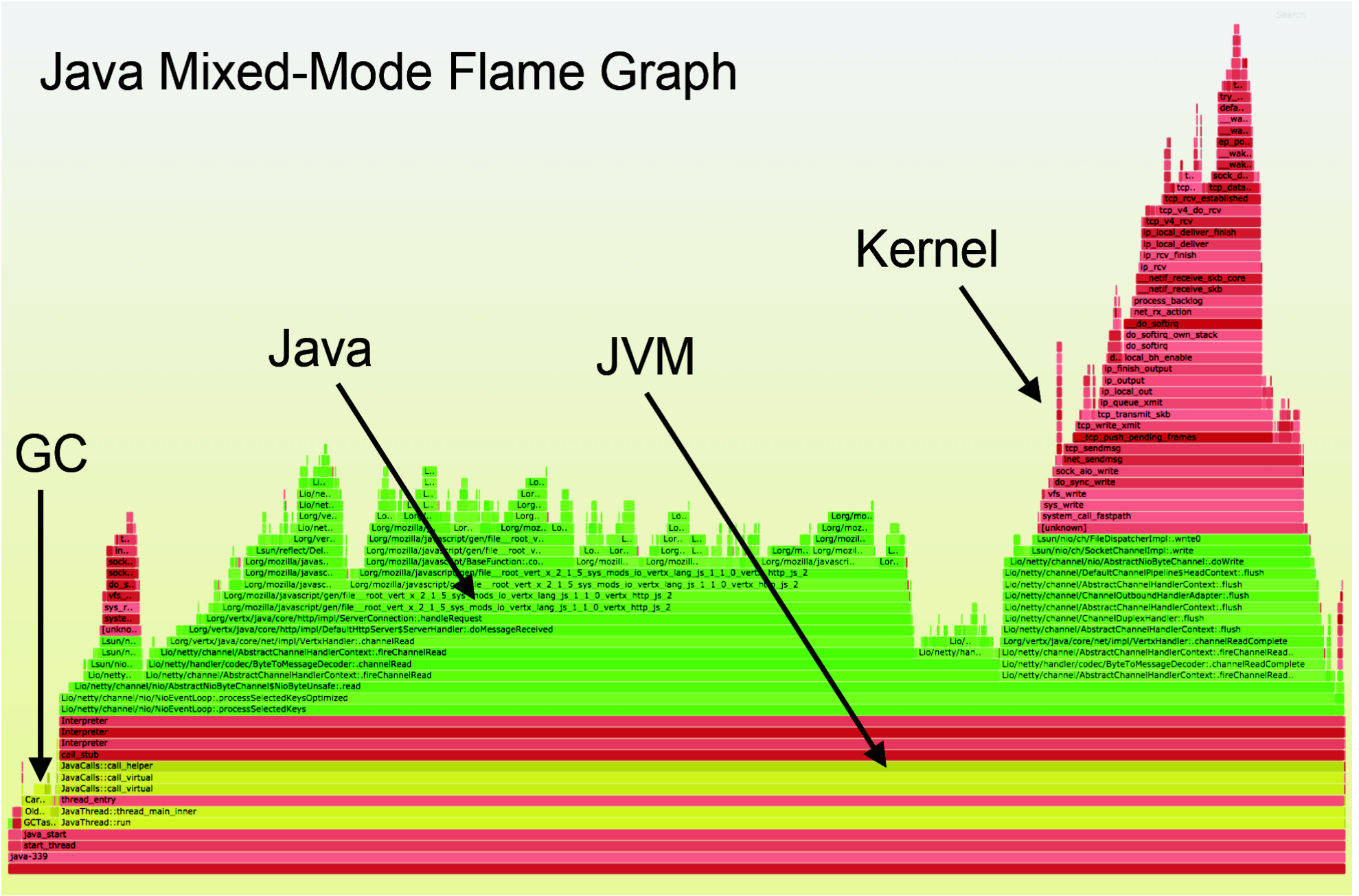
- Frames may be missing (inlined)
- Disabling inlining:
  - -XX:-Inline
  - Many more Java frames
  - Can be 80% slower!
- May not be necessary
  - Inlined flame graphs often make enough sense
  - Or tune -XX:MaxInlineSize and -XX:InlineSmallCode a little to reveal more frames
    - Can even improve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



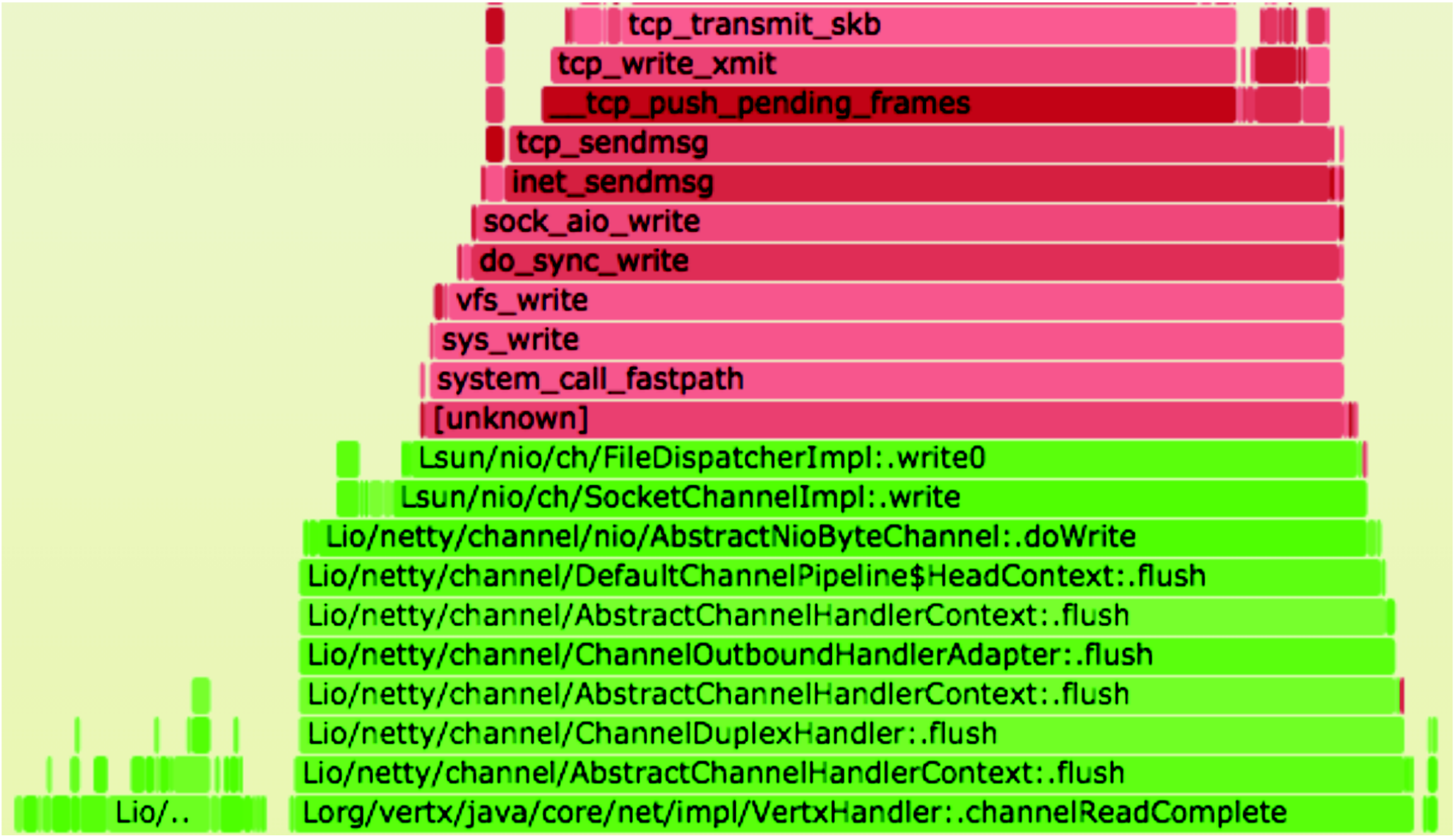
Java Symbols for perf

- 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"

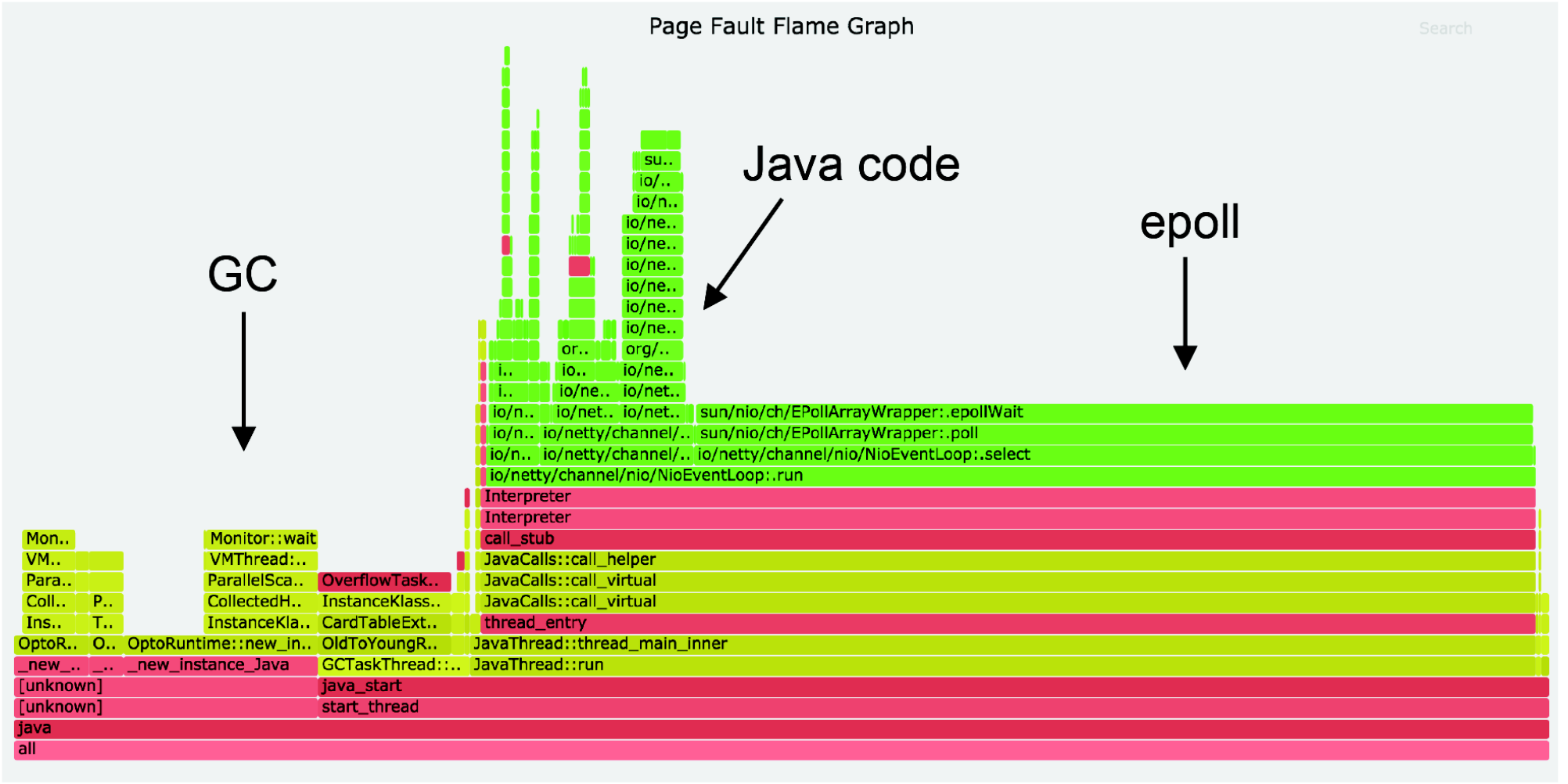
Stacks & Symbols



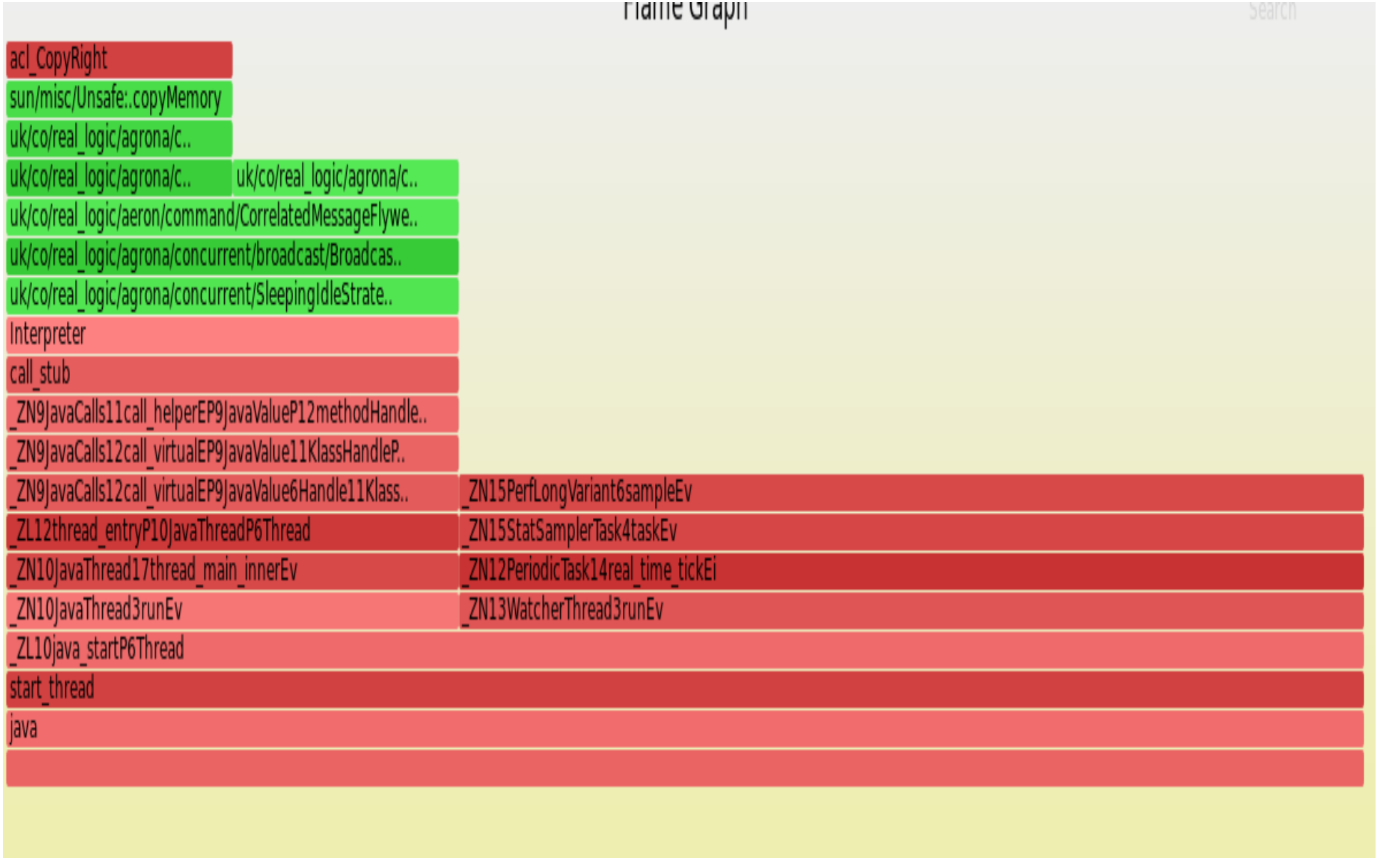
Stacks & Symbols (zoom)



Page Fault Flame Graph



Lets Look Inside:CPU:No Page Faults after start up

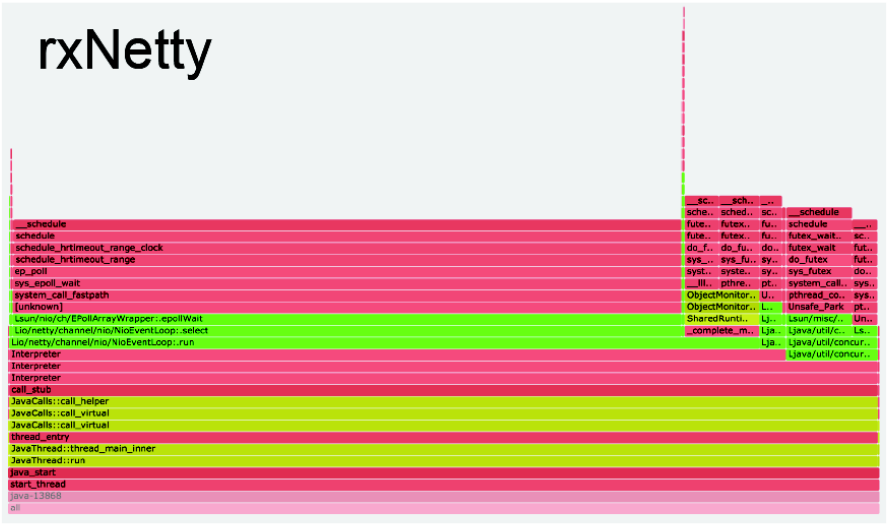


Context Switches

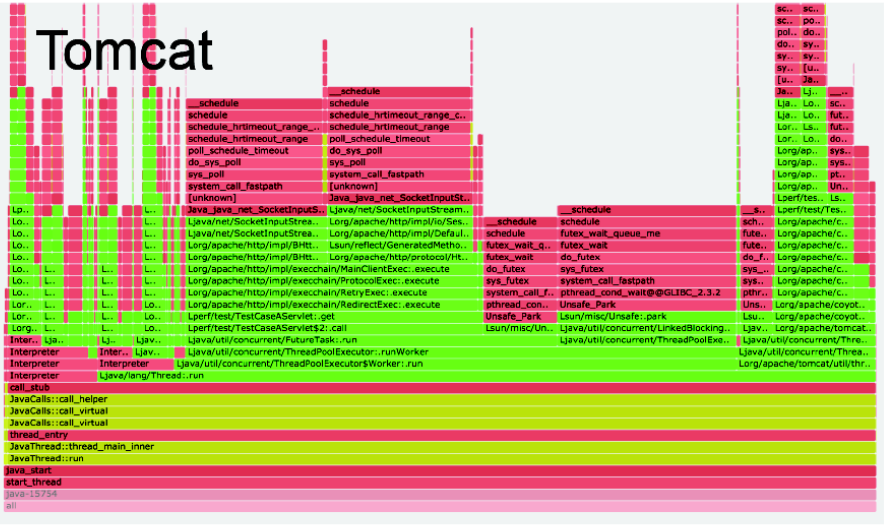
- Show why Java blocked and stopped running on-CPU:

```
# perf record -e context-switches -p PID -g -- sleep 5
```

- Identifies locks, I/O, sleeps
  - If code path shouldn't block and looks random, it's an involuntary context switch. I could filter these, but you should have solved them beforehand (CPU load).
- e.g., was used to understand framework differences:

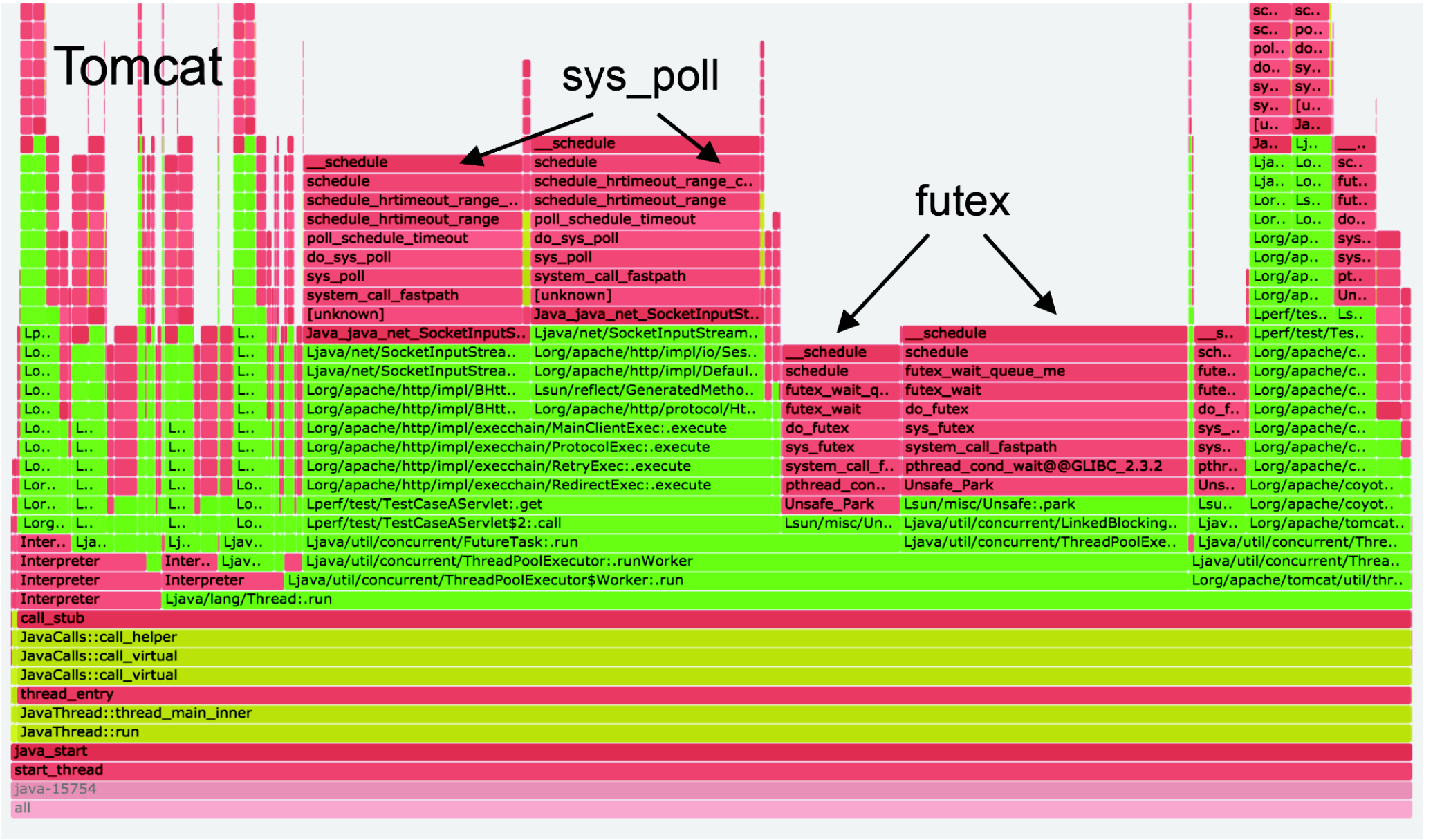


vs

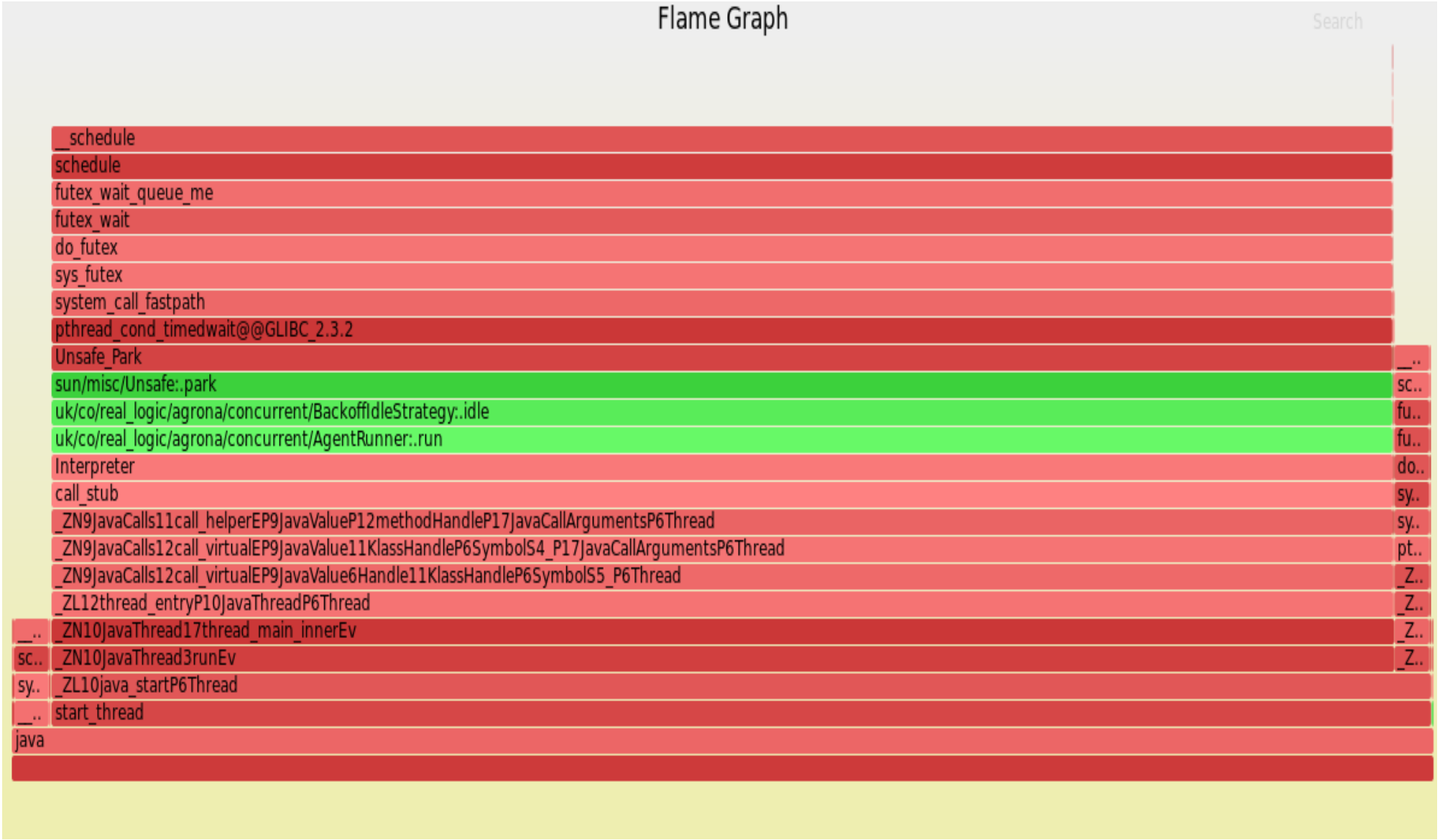


Context Switch Flame Graph (1/2)

Context Switch Flame Graph (2/2)



Lets Look Inside:CPU:almost Nil Context switches(Look closely)

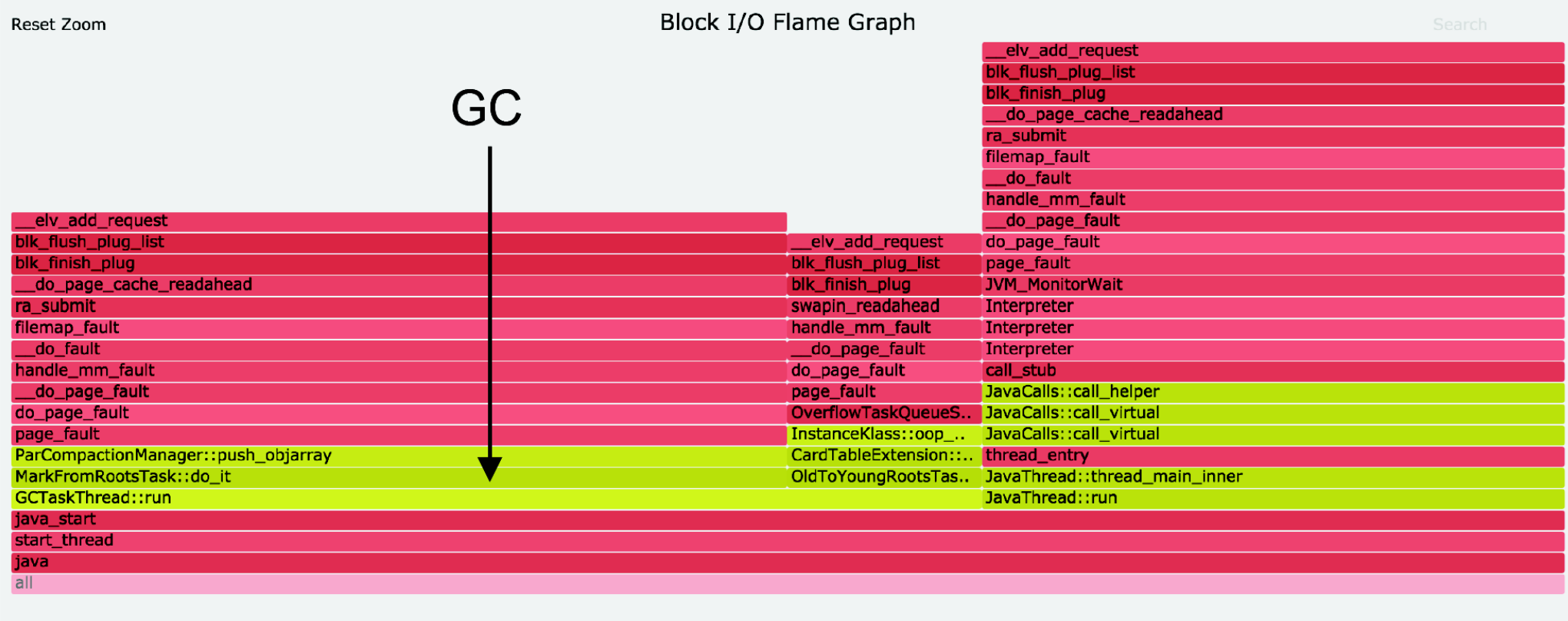


Disk I/O Requests

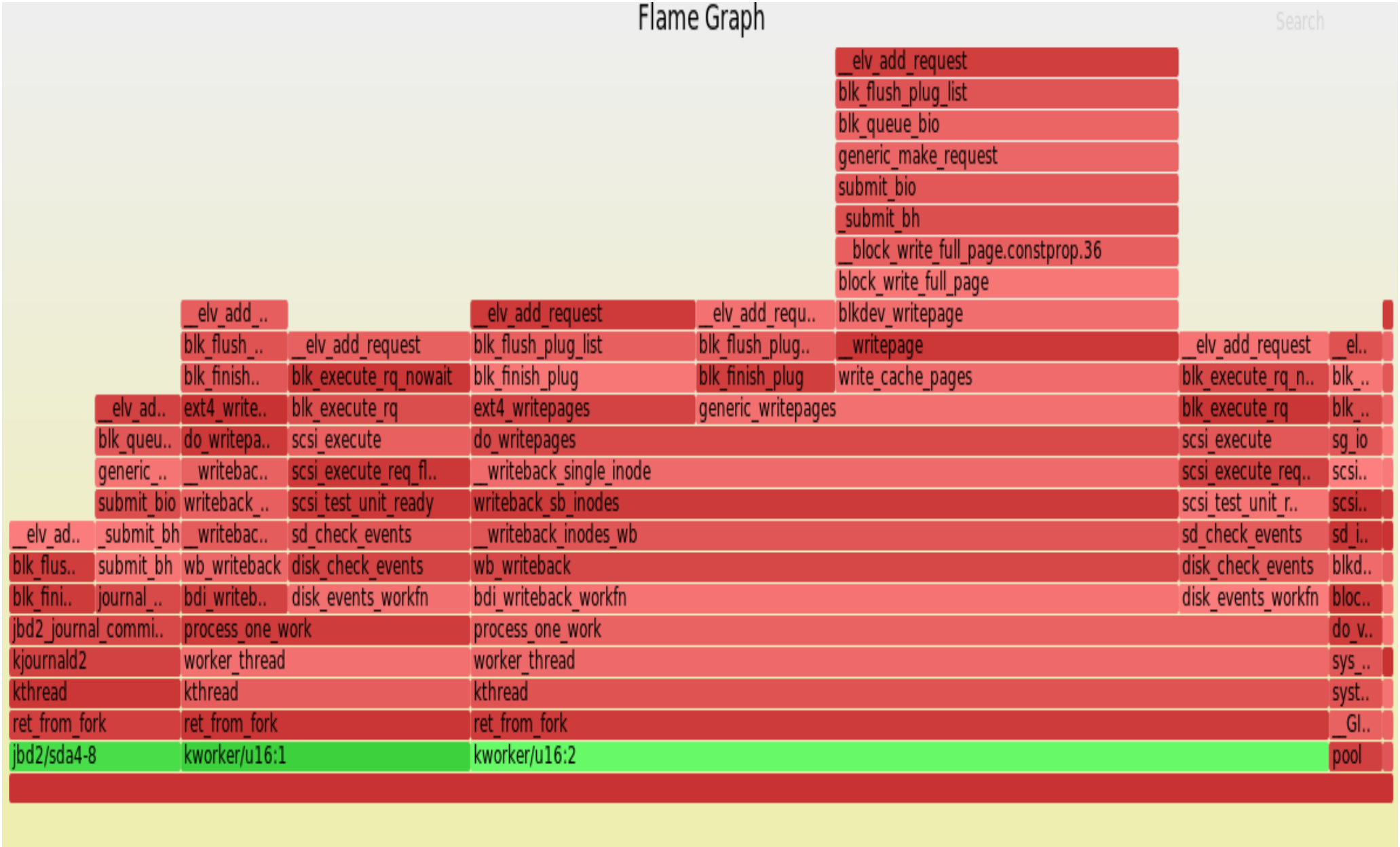
- Shows who issued disk I/O (sync reads & writes):

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

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



Lets Look Inside:CPU:Nil DiskIO

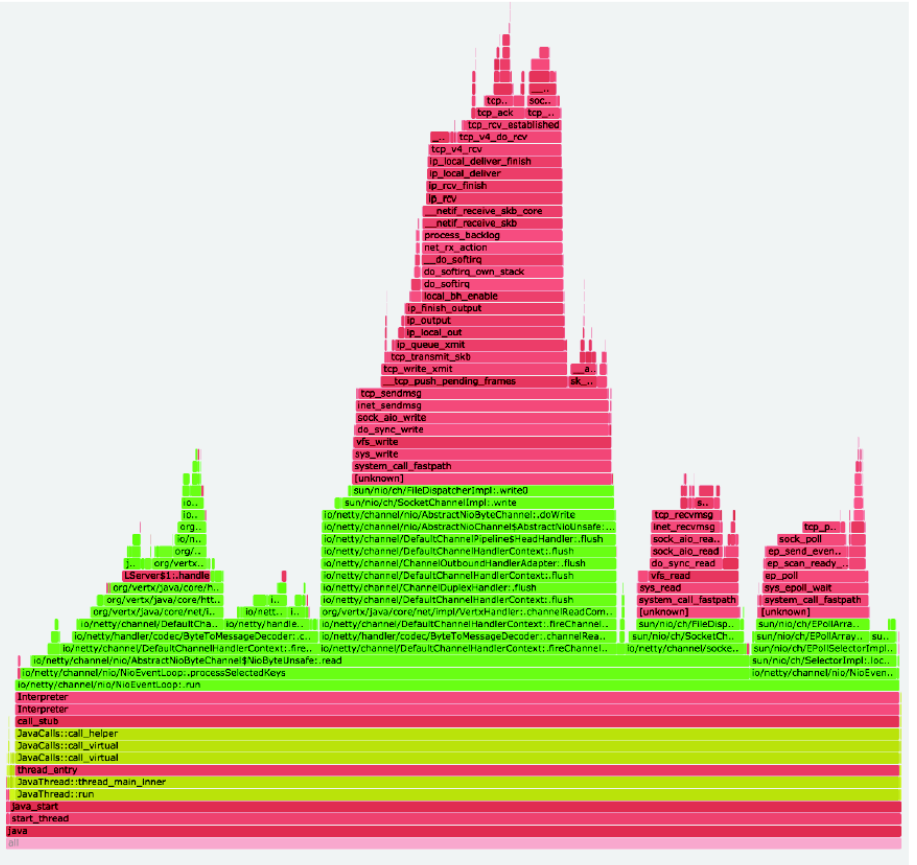


CPU Cache Misses

- 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



Lets Look Inside:CPU:Minimal cache misses considering amount od data being shovelled

perf annotate - examples -

perf record ./a.out
perf annotate –s main

```
ain
         Disassembly of section .text:

         0000000000400474 <main>:
         int main(void){
           push   %rbp
           mov    %rsp,%rbp
           int i;
           for(i=0; i<1000000000; i++)[]
           movl   $0x0,-0x4(%rbp)
         ↓ jmp    11
14.78     d:┌─→addl   $0x1,-0x4(%rbp)
 3.78    11:│  cmpl   $0x3b9ac9ff,-0x4(%rbp)
81.45       └─jle    d
           return 0;
           mov    $0x0,%eax
         }
           leaveq
         ← retq
```

```
int main(void){
        int i;
        for( i=0; i<1000000000; i++ );
        return 0;}
```

Perf:Listing Events

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

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

# Listing sched tracepoints (older syntax):
perf list -e 'sched:*'
```

Perf: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 basic CPU statistics, system wide, for 10 seconds:
perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10

# Various CPU level 1 data cache statistics for the specified command:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores command

# Various CPU data TLB statistics for the specified command:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

# 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 system calls for the entire system, for 5 seconds:
perf stat -e 'syscalls:sys_enter_*' -a sleep 5

# Count scheduler events for the specified PID, until Ctrl-C:
perf stat -e 'sched:*' -p PID

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

# Count ext4 events for the entire system, for 10 seconds:
perf stat -e 'ext4:*' -a 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
```

Perf:Profiling

```
# 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 on-CPU functions for the specified PID, at 99 Hertz, for 10 seconds:
perf record -F 99 -p PID sleep 10

# 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 PID, using dwarf to unwind stacks, at 99 Hertz, for 10 seconds:
perf record -F 99 -p PID -g dwarf 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 for the entire system, with dwarf stacks, at 99 Hertz, for 10 seconds:
perf record -F 99 -ag dwarf sleep 10

# Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 seconds:
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
```

Perf:Static Tracing

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

# Trace all context-switches, until Ctrl-C:
perf record -e context-switches -a

# Trace all context-switches with stack traces, until Ctrl-C:
perf record -e context-switches -ag

# Trace all context-switches with stack traces, for 10 seconds:
perf record -e context-switches -ag -- sleep 10

# 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 accepts()s with stack traces (inbound connections), until Ctrl-C:
perf record -e syscalls:sys_enter_accept* -ag

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

# Trace all page faults with stack traces, until Ctrl-C:
perf record -e page-faults -ag

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

# Trace kswapd wakeup events, until Ctrl-C:
perf record -e vmscan:mm_vmscan_wakeup_kswapd -ag
```

Perf:Dynamic Tracing-1

```
# Add a tracepoint for the kernel tcp_sendmsg() function entry ("--add" is 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 available variables for the kernel tcp_sendmsg() function (needs debuginfo):
perf probe -V tcp_sendmsg

# Show available variables for the kernel tcp_sendmsg() function, 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

# Show available variables for tcp_sendmsg() at line number 81 (needs debuginfo):
perf probe -V tcp_sendmsg:81

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

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

# Trace previously created probe when the bytes (alias) variable 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 (reliable, but needs debuginfo):
perf probe 'tcp_sendmsg size'

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

# Tell me how on Earth you would do this, but don't actually do it (needs debuginfo):
perf probe -nv 'tcp_sendmsg size sk->__sk_common.skc_state'

# Trace previous probe when size is non-zero, and state is not TCP_ESTABLISHED(1) (needs debuginfo):
perf record -e probe:tcp_sendmsg --filter 'size > 0 && skc_state != 1' -a

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

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

Perf:Dynamic Tracing-2

```
# 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
```

Perf: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 time,event,trace

# 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
```

Intel Performance Counter Monitor

Provided by Intel as source code

- Driver; GUI/command line tools; C++ library

Linux

- build upon MSR kernel module
- KDE: ksysguard plug-in

Windows

- compile/modify sample driver
- Perfmon plug-in

Linux: libpfm4

Using Linux perf_events interface

libpfm4

- Retrieve supported events per source
- Translating event IDs and names
- Program events

Architecture support

- Intel x86: since Pentium P6, Core Duo/Solo, Atom, Nehalem
- AMD64 x86: K7, K8 and newer; uncore since Bulldozer
- Some ARM, SPARC, IBM Power, MIPS models

Libpfm4 on ubuntu-numa0101.fsoc

Source code: examples and tools

showevtinfo (example tool)

- Lists supported and detected PMUs
- Xeon E5-2620 (Sandy Bridge EP):
- Lists 4196 available events, 634 supported
- Per event: PMU, index, parameters, description

Monitoring Hardware Counters

- libpfm – running examples/showevtinfo

```
#-----------------------------
IDX      : 37748738
PMU name : ix86arch (Intel X86 architectural PMU)
Name     : UNHALTED_REFERENCE_CYCLES
Equiv    : None
Flags    : None
Desc     : count reference clock cycles while the clock signal on the specific core is running. The reference clock operates a
t a fixed frequency, irrespective of core frequency changes due to performance state transitions
Code     : 0x13c
Modif-00 : 0x00 : PMU : [k] : monitor at priv level 0 (boolean)
Modif-01 : 0x01 : PMU : [u] : monitor at priv level 1, 2, 3 (boolean)
Modif-02 : 0x02 : PMU : [e] : edge (may require counter-mask >= 1) (boolean)
Modif-03 : 0x03 : PMU : [i] : invert (bool
Modif-04 : 0x04 : PMU : [c] : counter-mask in ra      255] (integer)
Modif-05 : 0x05 : PMU : [t] : measure any thread (boo
#-----------------------------
```

UMask = 01H
Event Select = 3CH

- sudo perf stat -e r13c -a sleep 1

```
Performance counter stats for 'system wide':

        1,848,495      r13c

      1.000977098 seconds time elapsed
```

Monitoring Hardware Counters

- The system has limited number of hardware performance counters.
- If you exceed them, perf would arbitrate

```
./perf stat -e cache-misses
-e cache-references -e cpu-cycles -e dTLB-
loads -e iTLB-loads -a -- sleep 1
```

```
Performance counter stats for 'system wide':

       1,113,116      cache-misses              #   27.387 % of all cache refs     [80.14%]
       4,064,355      cache-references                                             [80.15%]
     327,853,786      cpu-cycles                [80.16%]
      13,941,380      dTLB-loads                                                   [80.15%]
          22,766      iTLB-loads                                                   [79.73%]

    1.000972757 seconds time elapsed
```

Monitoring Hardware Counters

- Perdefined software events can be monitored
- `perf stat -e minor-faults -- ls`

```
Performance counter stats for 'ls':

               254      minor-faults

      0.001568812 seconds time elapsed
```

Monitoring Hardware Counters

- Perdefined software events can be monitored
- `perf stat -e minor-faults -- ls`

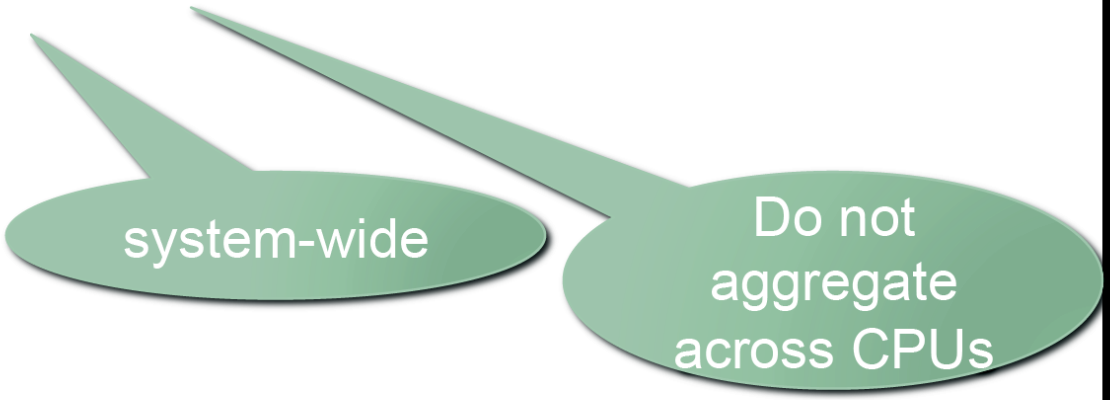
```
Performance counter stats for 'ls':

               254      minor-faults

      0.001568812 seconds time elapsed
```

Monitoring Hardware Counters

- `perf stat -e minor-faults -a -A -- ls`




```
 Performance counter stats for 'system wide':

CPU0                    0      minor-faults
CPU1                    0      minor-faults
CPU2                    0      minor-faults
CPU3                    0      minor-faults
CPU4                    0      minor-faults
CPU5                    0      minor-faults
CPU6                    0      minor-faults
CPU7                    0      minor-faults
CPU8                    0      minor-faults
CPU9                    0      minor-faults
CPU10                   0      minor-faults
CPU11                   0      minor-faults
CPU12                   0      minor-faults
CPU13                   0      minor-faults
CPU14                 256      minor-faults
CPU15                  26      minor-faults
CPU16                   0      minor-faults
CPU17                   7      minor-faults
CPU18                   0      minor-faults
CPU19                   0      minor-faults
CPU20                   0      minor-faults
CPU21                   0      minor-faults
CPU22                   0      minor-faults
CPU23                   0      minor-faults

       0.001905321 seconds time elapsed
```

- In many cases you would just run `sleep [X]` as your process for the duration you want to sample
- Use -x, for comma delimited file

Monitoring Hardware Counters

- You can tell when the event counter should take place

```
u - user-space counting
k - kernel counting
h - hypervisor counting
G - guest counting (in KVM guests)
H - host counting (not in KVM guests)
p - precise level
S - read sample value (PERF_SAMPLE_READ)
D - pin the event to the PMU
```

- `perf stat -e minor-faults:u -e minor-faults:k -- ls`

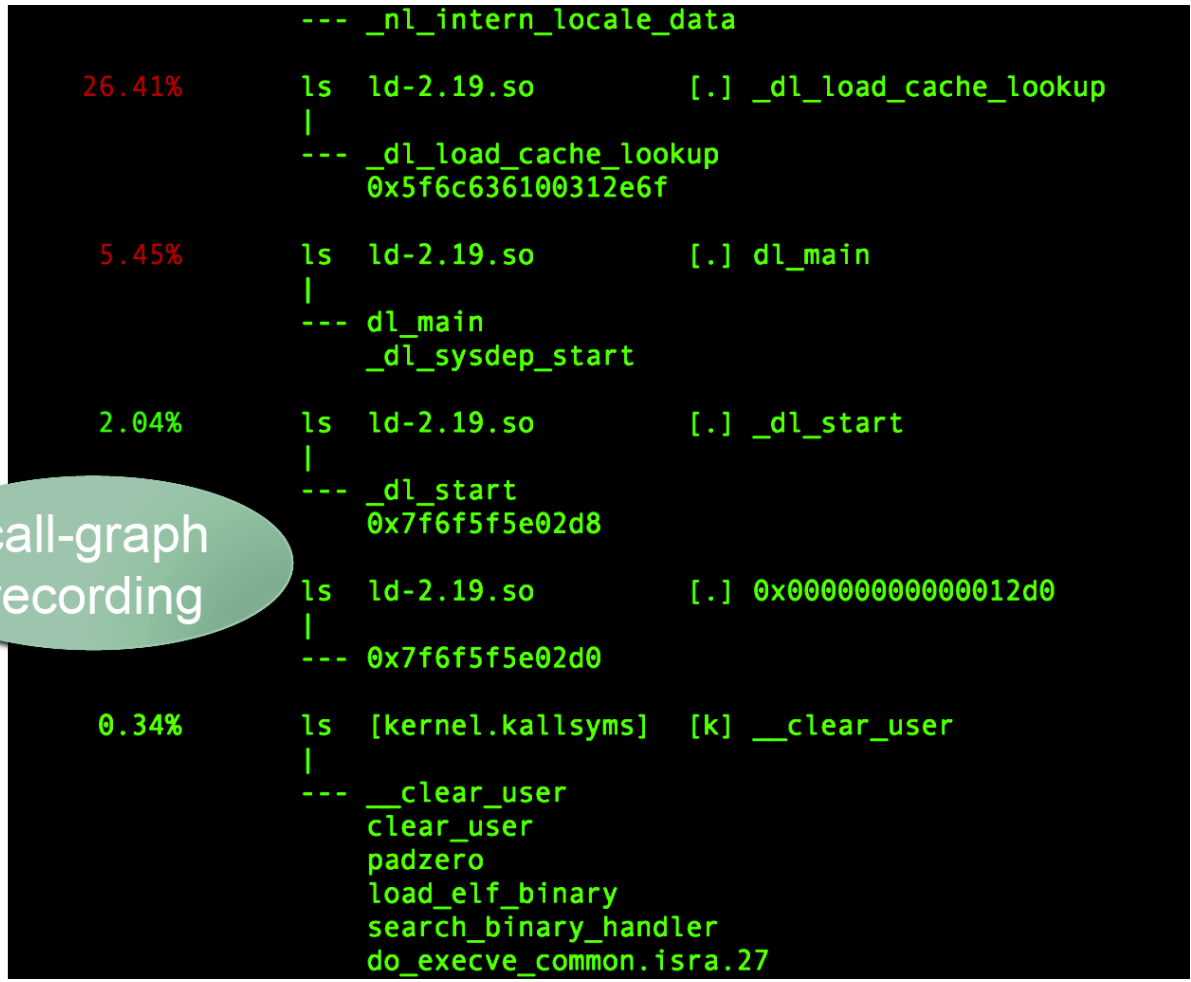
```
Performance counter stats for 'ls':

             247      minor-faults:u
               8      minor-faults:k

       0.001965714 seconds time elapsed
```

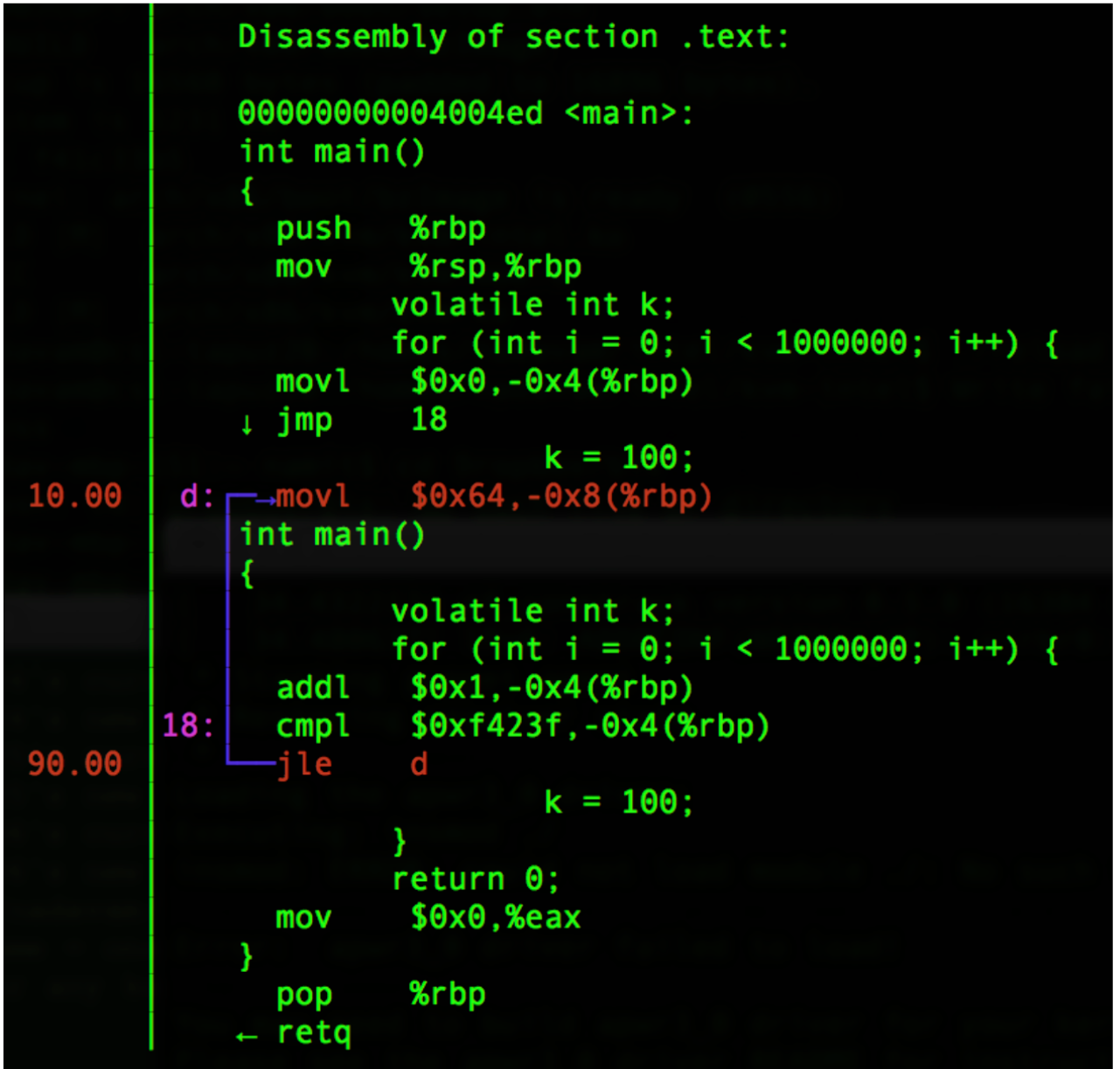
Monitoring Hardware Counters

- Recording and reporting is possible
- `perf record -e minor-faults -g -- ls`
- `perf report`



Annotating the Source

- You can use `perf annotate [func]` or `perf report` to use annotation facilities
- You can extract vmlinux and use -k [vmlinux]



Monitoring Hardware Counters

- You can create your own trace-points (but not likely get them upstream)
- See and include linux/tracepoint.h

```
TRACE_EVENT(kvm_userspace_exit,
            TP_PROTO(__u32 reason, int errno),
            TP_ARGS(reason, errno),

        TP_STRUCT__entry(
                __field(        __u32,          reason          )
                __field(        int,            errno           )
        ),

        TP_fast_assign(
                __entry->reason         = reason;
                __entry->errno          = errno;
        ),

        TP_printk("reason %s (%d)",
                  __entry->errno < 0 ?
                  (__entry->errno == -EINTR ? "restart" : "error") :
                  __print_symbolic(__entry->reason, kvm_trace_exit_reason),
                  __entry->errno < 0 ? -__entry->errno : __entry->reason)
);
```

```
                goto out;
        r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
        trace_kvm_userspace_exit(vcpu->run->exit_reason, r);
        break;
se KVM_GET_REGS: {
```

Usage

Monitoring Hardware Counters

- Memory access overhead
- `sudo ./perf mem record`
- `sudo ./perf mem report`
- Use -g to generate call-graph

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 16  of event 'cpu/mem-loads/pp'
# Total weight : 149
# Sort order   : local_weight,mem,sym,dso,symbol_daddr,dso_daddr,snoop,tlb,locked
#
# Overhead       Samples  Local Weight            Memory access                 Symbol      Shared Object                        Data Symbo
# ........  ............  ............  ........................  ....................  ..............  ...................................
#
    14.77%             1  22            L1 hit                    [k] handle_mm_fault       [kernel.kallsyms]  [k] 0xffff88041947b0b0
     9.40%             1  14            L1 hit                    [k] acpi_map_lookup       [kernel.kallsyms]  [k] acpi_ioremaps+0x0
     9.40%             1  14            L2 hit                    [k] perf_event_aux        [kernel.kallsyms]  [k] 0xffff88081934be28
     8.72%             1  13            L1 hit                    [k] prepare_creds         [kernel.kallsyms]  [k] 0xffff880419ba96f8
     8.05%             1  12            L1 hit                    [.] get_next_seq          libc-2.19.so       [.] 0x00007fff956615a0
     5.37%             1  8             L1 hit                    [k] perf_event_aux_ctx    [kernel.kallsyms]  [k] 0xffff8800c8c31dc0
     4.70%             1  7             L1 hit                    [.] get_next_seq          libc-2.19.so       [.] 0x00007fff956615e8
     4.70%             1  7             L1 hit                    [.] __strcoll_l           libc-2.19.so       [.] 0x00007fff956616e8
     4.70%             1  7             L1 hit                    [k] perf_event_aux_ctx    [kernel.kallsyms]  [k] 0xffff8800b1ec3878
     4.70%             1  7             L1 hit                    [k] vunmap_page_range     [kernel.kallsyms]  [k] 0xffff88041fce6d78
     4.70%             1  7             L1 hit                    [k] memcpy                [kernel.kallsyms]  [k] 0xffff8800c8c31bb8
     4.70%             1  7             L1 hit                    [k] acpi_map_lookup       [kernel.kallsyms]  [k] 0xffff88081923c680
     4.03%             1  6             L1 hit                    [.] _dl_catch_error       ld-2.19.so         [.] 0x00007fff95661838
     4.03%             1  6             L1 hit                    [.] _IO_getdelim          libc-2.19.so       [.] 0x00007fff95661cac
     4.03%             1  6             L1 hit                    [k] syscall_trace_leave   [kernel.kallsyms]  [k] 0xffff8800c8c31f38
     4.03%             1  6             L1 hit                    [k] put_prev_task_fair    [kernel.kallsyms]  [k] 0xffff88041688db0c
```

Monitoring Hardware Counters:PAPI

```
mohit@nomind:~/Work/UltraLowLatency$ papi_avail
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.5.1.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz (70)
CPU Revision             : 1.000000
CPUID Info               : Family: 6  Model: 70  Stepping: 1
CPU Max Megahertz        : 3700
CPU Min Megahertz        : 800
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 1
NUMA Nodes               : 1
CPUs per Node            : 8
Total CPUs               : 8
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 384
--------------------------------------------------------------------------------


================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
```

Monitoring Hardware Counters:PAPI

```
float t0, t1;
long_long values[2];
float matrixa[INDEX][INDEX], matrixb[INDEX][INDEX], mresult[INDEX][INDEX];
int events[2] = {PAPI_TOT_CYC, PAPI_L3_TCM}, ret;
int retval;
int i, j, k;

if (PAPI_num_counters() < 2) {
    fprintf(stderr, "No hardware counters here, or PAPI not supported.\n");
    exit(1);
}

/* Check to see if the preset, PAPI_TOT_INS, exists */
retval = PAPI_query_event( PAPI_TOT_CYC );
if (retval != PAPI_OK) {
    fprintf (stderr,"No instruction counter? How lame.\n");
    exit(1);
}
/* Initialize the Matrix arrays */
for (i = 0; i < INDEX * INDEX; i++) {
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float) 1.1;
}

/* Setup PAPI library and begin collecting data from the counters */
t0 = gettime();
if ((ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(ret))
    exit(1);
}
/* Matrix-Matrix multiply */
for (i = 0; i < INDEX; i++)
    for (j = 0; j < INDEX; j++)
        for (k = 0; k < INDEX; k++)
            mresult[i][j] = mresult[i][j] + matrixa[i][k] * matrixb[k][j];

/* Collect the data into the variables passed in */
if ((ret = PAPI_read_counters(values, 2)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(ret));
    exit(1);
}
t1 = gettime();
```