

YOW! 2022

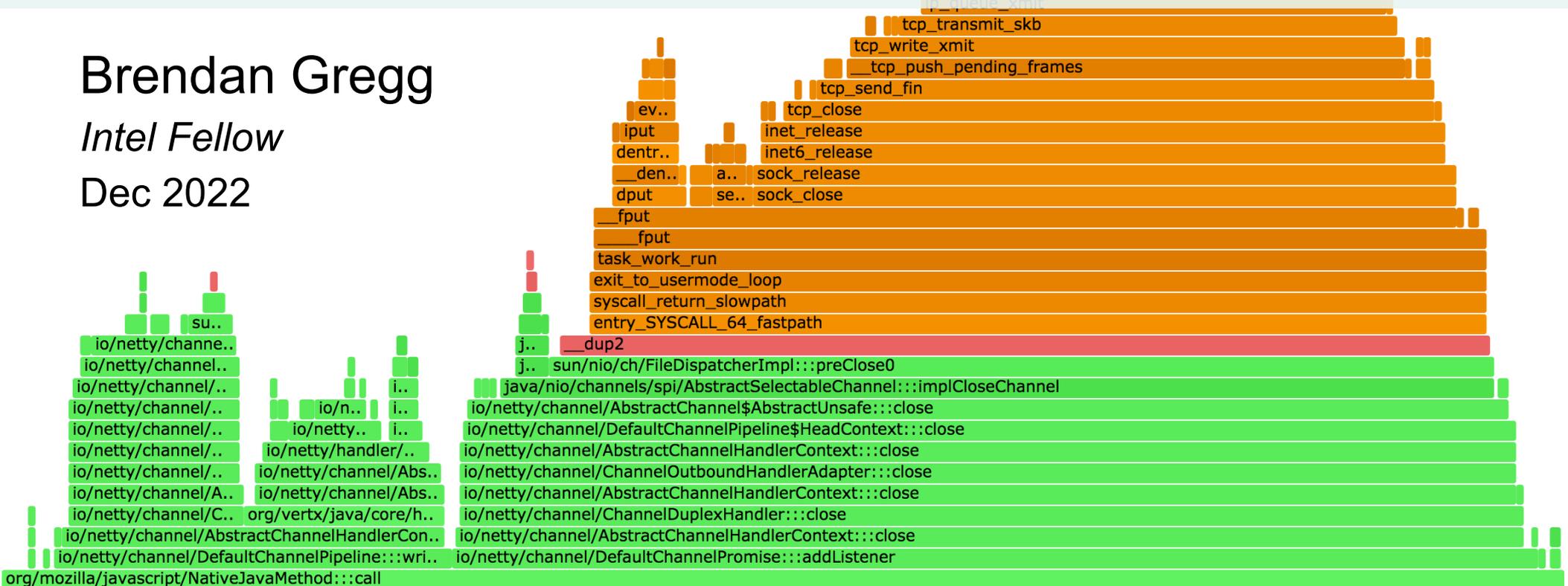
Visualizing Performance

The Developer's Guide to Flame Graphs

Brendan Gregg

Intel Fellow

Dec 2022

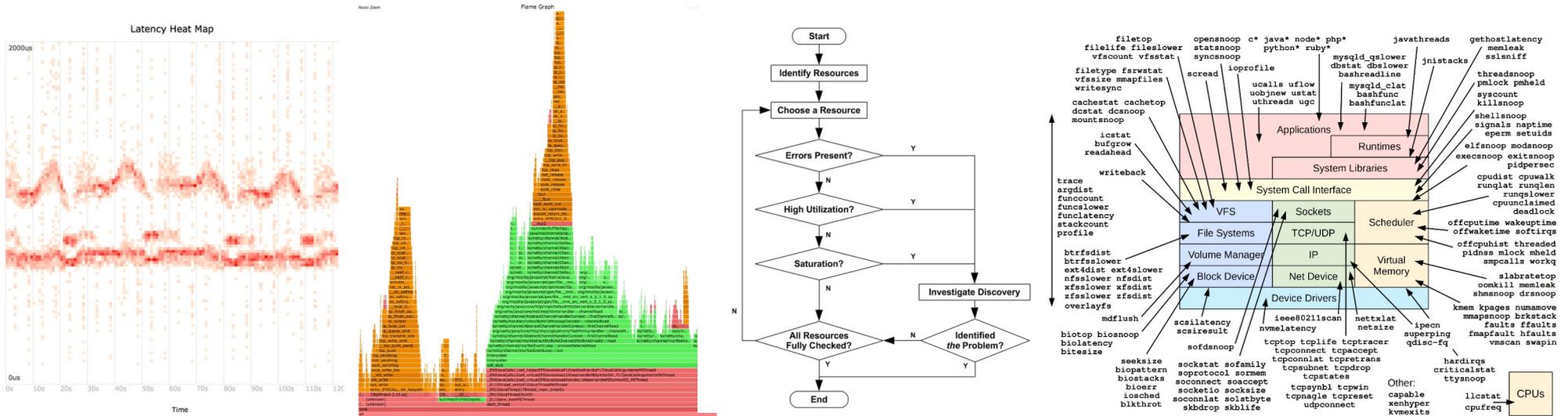


Statement from the heart

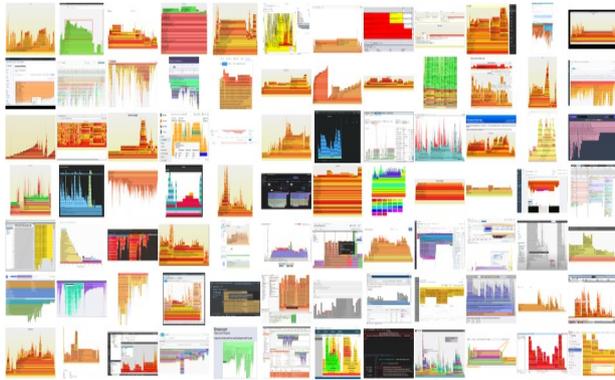
I'd like to begin by acknowledging the Traditional Owners of this land and pay my respects to Elders past and present.

My Dream

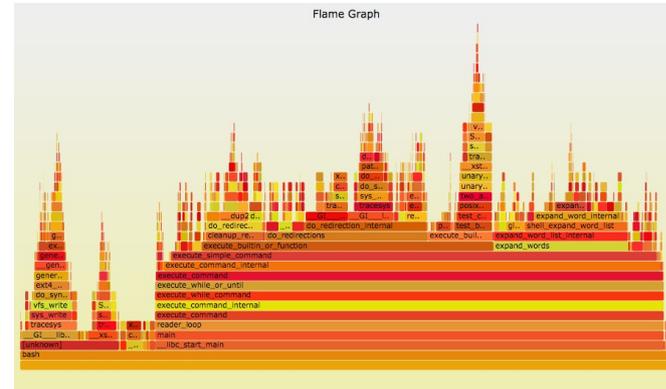
To Completely Understand the Performance of Everything



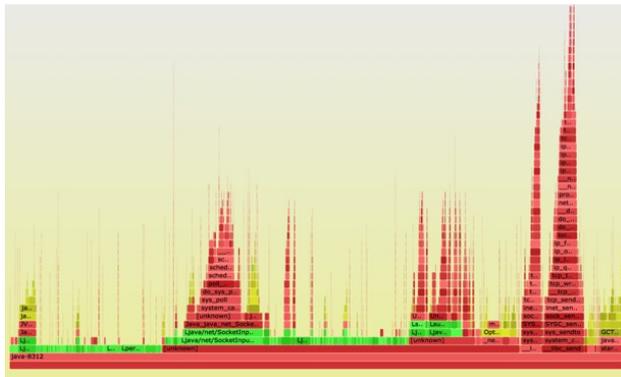
Agenda



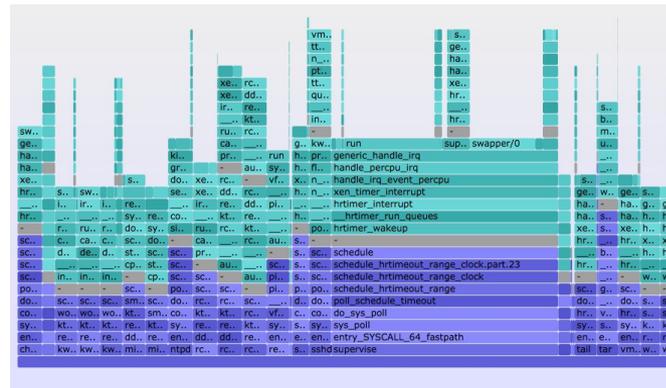
1. Implementations



2. CPU Flame graphs



3. Stacks & Symbols



4. Advanced flame graphs

Take Aways

1. Interpret CPU flame graphs
2. Understand runtime challenges
3. Why eBPF for advanced flame graphs

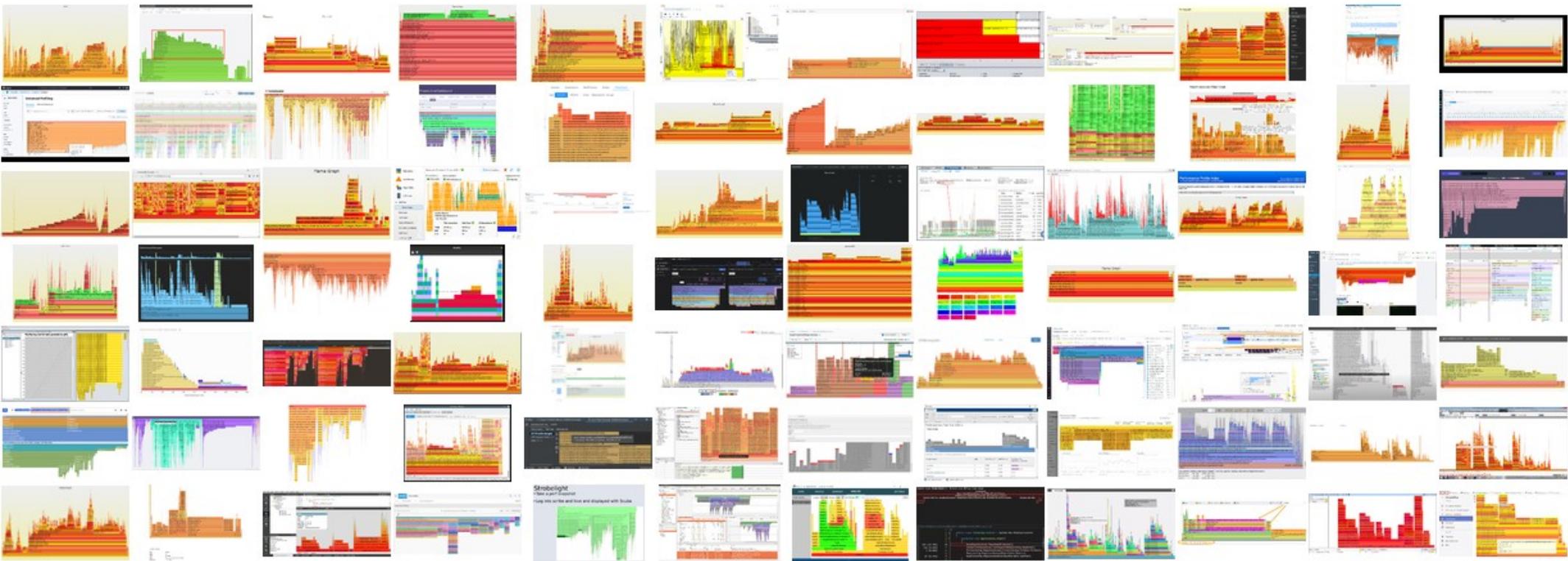
A new tool to lower your cost, latency, and carbon

Slides online:

https://www.brendangregg.com/Slides/YOW2022_flame_graphs.pdf

1. IMPLEMENTATIONS

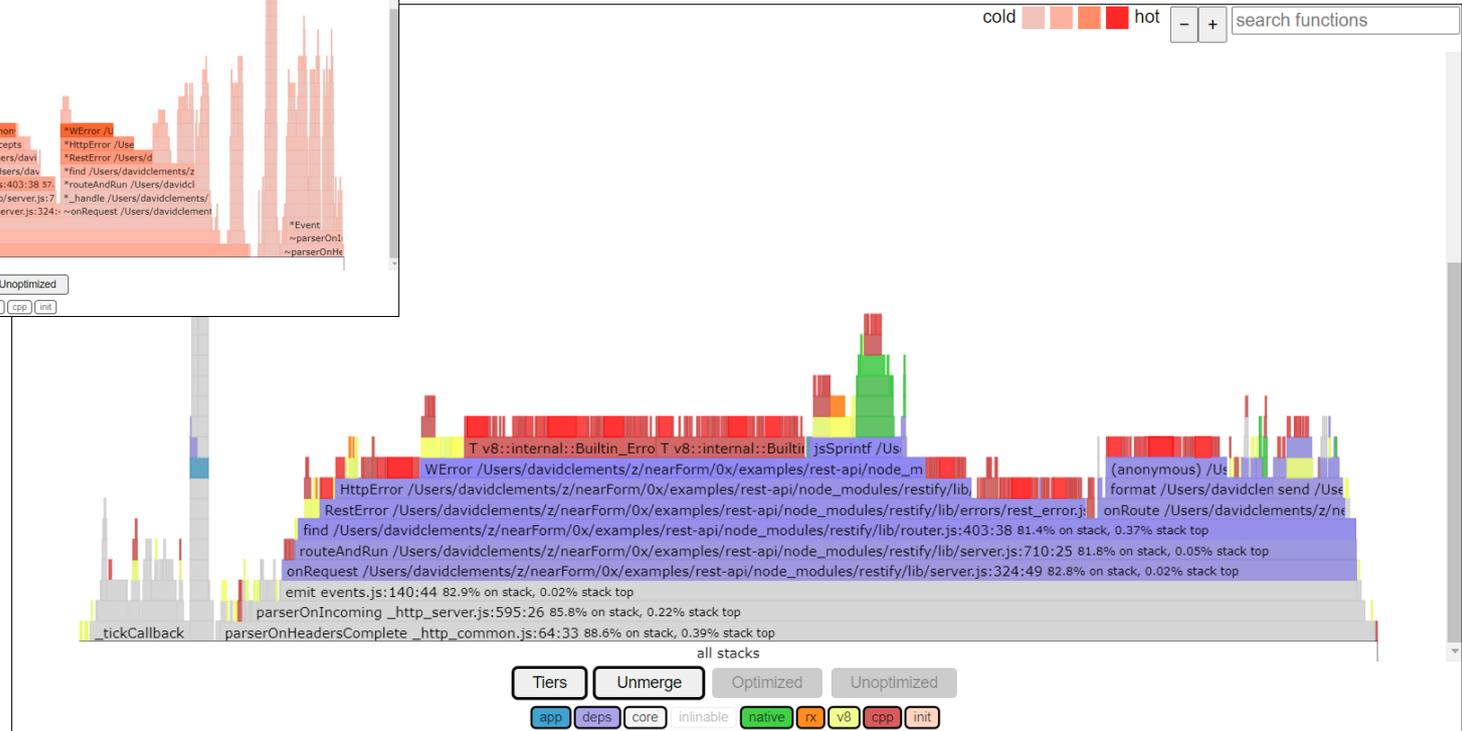
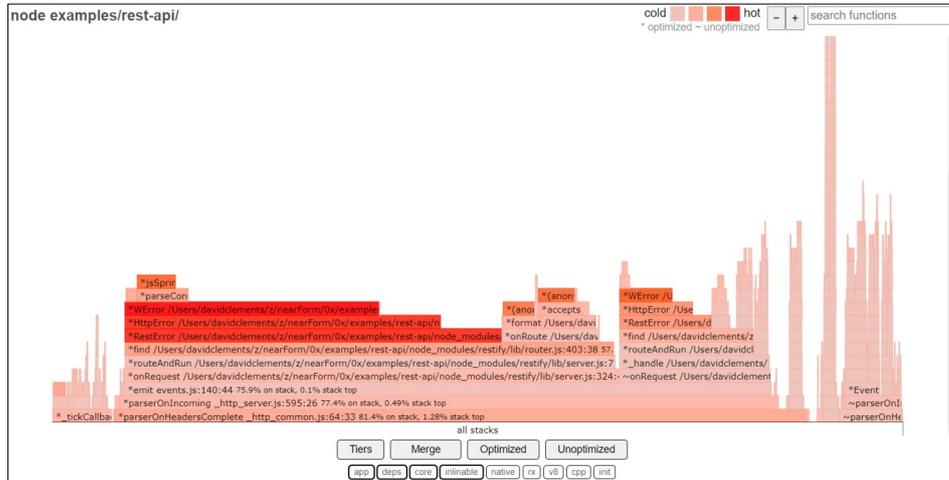
Quick Tour of Some Examples



More examples in later “bonus slides” section.

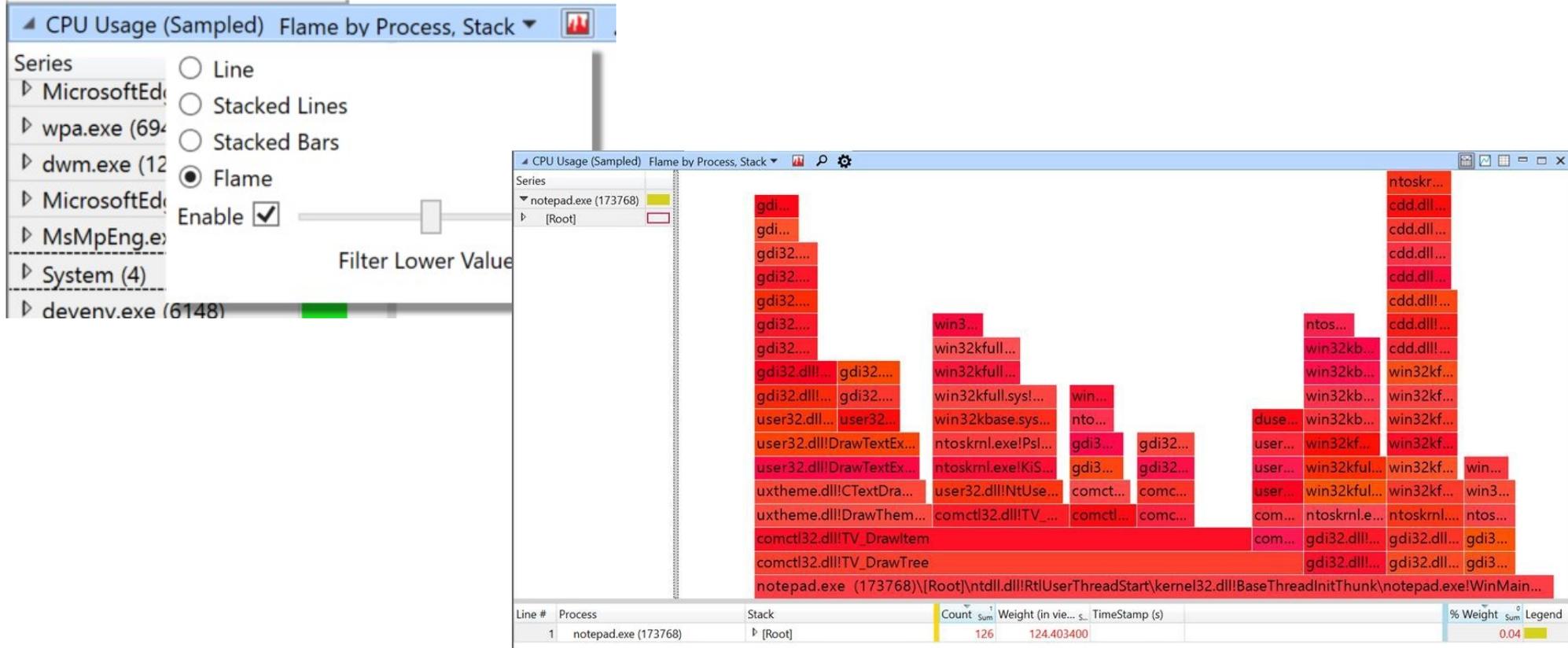
(Note: This is not an an endorsement of any company/product or sponsored by anyone.)

Node.js: 0x (2016)



Source: <https://github.com/davidmarkclements/0x> (David Mark Clements)

Microsoft: WPA / ETW (2016)



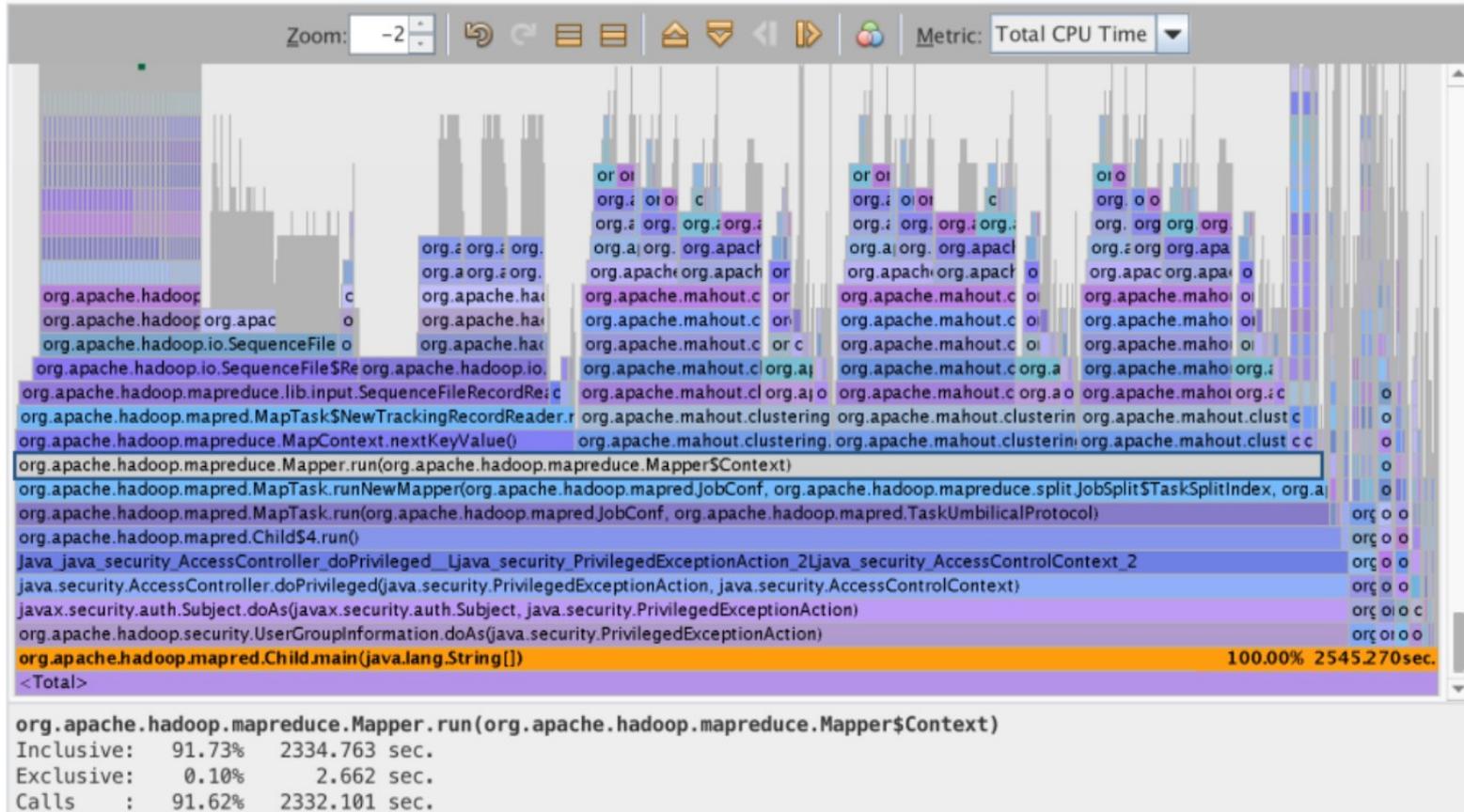
Source: https://learn.microsoft.com/en-us/windows-hardware/test/wpt/graphs#flame_graphs

LinkedIn: ODP (2017)



Source: <https://engineering.linkedin.com/blog/2017/01/odp--an-infrastructure-for-on-demand-service-profiling>

Oracle: Developer Studio Performance Analyzer (2017)



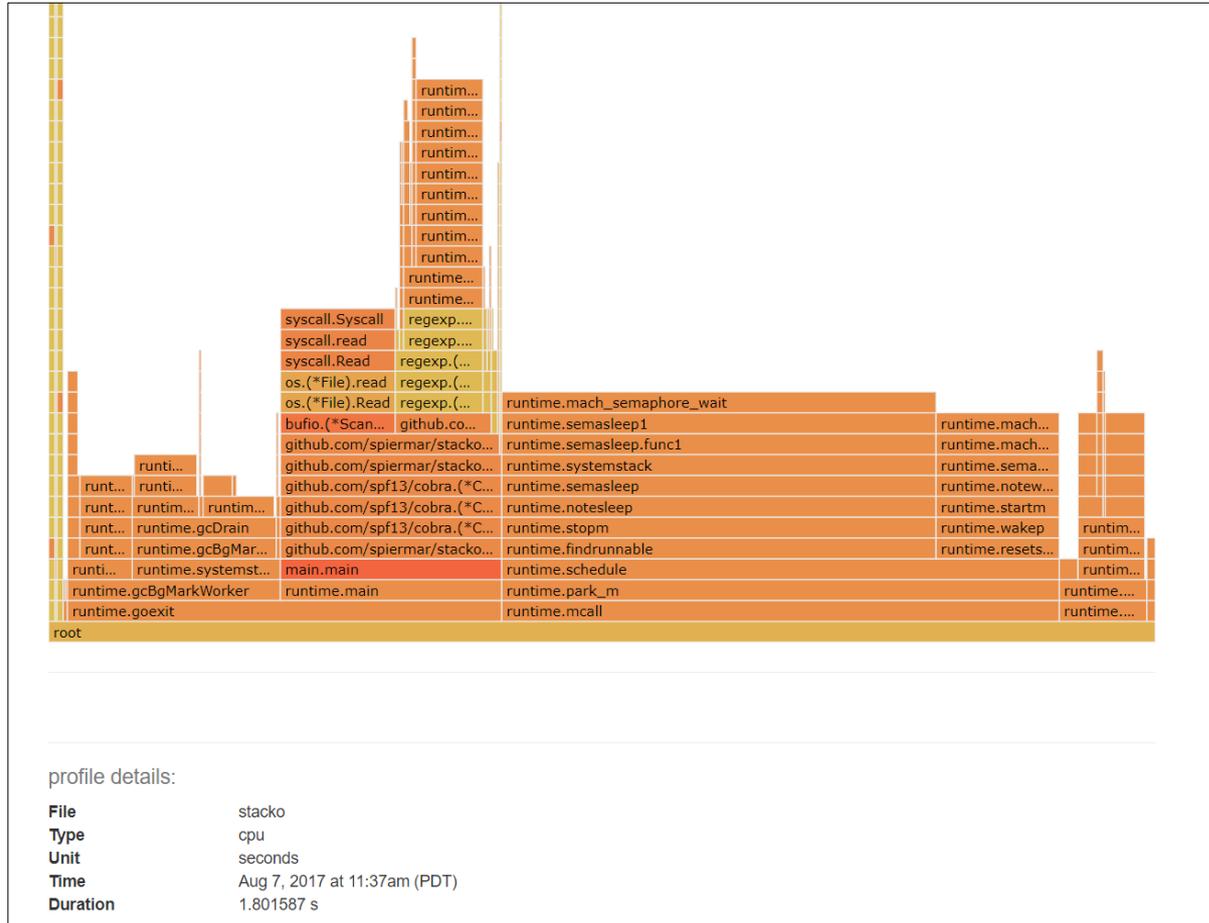
Source: <https://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/o11-151-perf-analyzer-brief-1405338.pdf>

Windows: PerfView (2017)



Source: <https://github.com/Microsoft/perfview/pull/440> (Adam Sitnik)

Google: pprof (2017)



Source: <https://github.com/google/pprof/pull/188> (Martin Spier)

Linux: hotspot (2017)

The screenshot displays the Hotspot performance analysis tool interface, which is used for analyzing performance data. The main window is titled "perf.data - Hotspot" and shows a flame graph of the application's execution. The flame graph is a horizontal bar chart where the length of each bar represents the time spent in a particular function. The bars are color-coded by function, and the total time spent is 6.469E+08 aggregated cycles:u cost in total.

The interface includes several panels:

- Summary:** Shows the current view (Bottom Up, Top Down, Flame Graph) and search options.
- Disassembly:** Shows the assembly code for the symbol "main".
- Caller / Callee:** Shows the caller and callee information for the selected function.
- Time Line:** Shows the time line of the application's execution.
- Processes:** Shows the processes running on the system.
- CPUs:** Shows the CPU usage for the system.

The disassembly view shows the following code:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv)
{
    if (!file) {
        fprintf(stderr, "failed to open file %s\n",
            return 1;
    }
    fseek(file, 0, SEEK_END);
    10b0: 31 f6          xor     %eax, %eax
    10b2: ba 02 00 00 00 mov     %eax, %ebx
    10b7: 48 89 c7      mov     %r9, %r12
    10ba: e8 a1 ffff ff call    %r12, %r12
        fileSize = ftell(file);
    10bf: 48 89 ef      mov     %r12, %r15
    10ca: e8 71 ffff ff call    %r15, %r15
        fclose(file);
    10cf: 48 89 ef      mov     %r12, %r15
    10d2: e8 79 ffff ff call    %r15, %r15
        for (i = 0; i < 5000000; ++i) {
    10d7: 83 eb 01      sub     %ebx, %ebx
    10da: 74 3d        je     %ebx, %ebx
        file = fopen(argv[0], "rb");
    10dc: 49 8b 3c 24   mov     (%r12),%rdi
        }
```

The caller/callee view shows the following data:

Caller	Binary	cycles:u	Callee	Binary	cycles:u	cycles:u (self)	cycles:u (incl.)
main			main				
main	c-syscalls	0.404%	main		0.142%	0.404%	81.1%
dl_main	ld-2.33.so	0%	main...		0.0709%	0%	0.364%
__libc_start_main	libc-2.33.so	0%	main...		0.0705%	0%	81%

The time line view shows the following data:

Source	0.0s	0.2s	0.4s	0.6s	0.8s	1.0s	1.2s	1.4s	1.6s	1.8s	2.0s	2.2s	2.4s
Processes													
CPUs													

Source: <https://github.com/KDAB/hotspot> (Milian Wolff)

Eclipse Foundation: TraceCompass (2018)

The screenshot displays the Eclipse TraceCompass interface. The top panel shows a flame chart for the process 'main' with various function calls like 'print_dir', 'sort_files', and 'mportsort'. The middle panel shows 'Function Duration Statistics' for 'ls-traces/ust/uid/0/64-bit' with a table of metrics. The bottom panel shows a 'Trace' log with columns for Trace, Timestamp, Channel, CPU, Event type, and Contents.

Level	Minimum	Maximum	Average	Standard Deviation	Count	Total
▼ Total	161 ns	51.750 ms	4.799 µs	265.470 µs	111541	535.274 ms
md5_process_block	249 ns	1.442 µs	268 ns	65 ns	5851	1.569 ms
just	165 ns	924 ns	178 ns	44 ns	640	114.039 µs
extract_dirs_from_files	3.619 µs	3.619 µs	3.619 µs	—	1	3.619 µs
close_stdout	71.929 µs	71.929 µs	71.929 µs	—	1	71.929 µs
heap_alloc	1.462 µs	1.462 µs	1.462 µs	—	1	1.462 µs
__stat	3.190 µs	3.190 µs	3.190 µs	—	1	3.190 µs
get_nonce	18.200 µs	18.200 µs	18.200 µs	—	1	18.200 µs
quote_name	853 ns	32.221 µs	1.064 µs	1.849 µs	379	403.253 µs
version_etc_ar	1.970 µs	7.229 µs	4.599 µs	—	2	9.199 µs
hard_locale	198 ns	302 ns	250 ns	—	2	500 ns
gobble_file	1.889 µs	16.743 µs	2.155 µs	—	906	818.734 µs
xrealloc	1.308 µs	13.090 µs	7.199 µs	—	2	14.398 µs
default_key_compare	572 ns	572 ns	572 ns	—	1	572 ns

Trace	Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
ls-traces/kernel	2018-01-29 15:00:45.917 659 277	kernel_2	2	rcu_utilization	s=End context switch, context_perf_cpu_migrations=33
ls-traces/kernel	2018-01-29 15:00:45.917 659 943	kernel_2	2	sched_stat_wait	comm=sort, tid=26070, delay=0, context_perf_cpu_migrations=33
ls-traces/kernel	2018-01-29 15:00:45.917 660 200	kernel_2	2	sched_switch	prev_comm=swapper/2, prev_tid=0, prev_prio=20, prev_state=0, next_comm=sort, next_tid=26070, next...
ls-traces/kernel	2018-01-29 15:00:45.917 661 664	kernel_2	2	syscall_exit_read	ret=326, buf=140720802526912, context_perf_cpu_migrations=33
ls-traces/kernel	2018-01-29 15:00:45.917 662 423	kernel_2	2	syscall_entry_lseek	fd=3, offset=556795, whence=0, context_perf_cpu_migrations=33
ls-traces/kernel	2018-01-29 15:00:45.917 662 698	kernel_2	2	syscall_exit_lseek	ret=556795, context_perf_cpu_migrations=33
ls-traces/kernel	2018-01-29 15:00:45.917 662 936	kernel_2	2	syscall_entry_read	fd=3, count=8, context_perf_cpu_migrations=33

Source: <https://www.eclipse.org/tracecompass/index.html>

Java: Java Mission Control (2018)

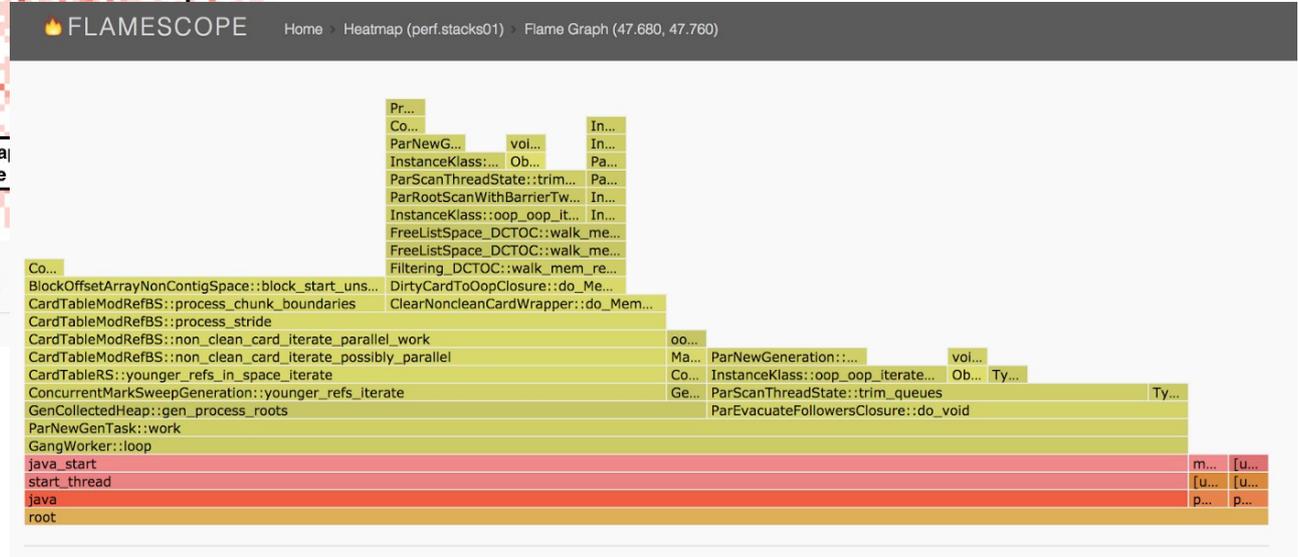
The screenshot displays the Java Mission Control (JMC) 2018 interface. The left sidebar shows the navigation tree with 'Method Profiling' selected. The main window is divided into three panes:

- Method Profiling:** Shows a list of methods being profiled, including `weblogic.xml.bebe.baseparser`, `weblogic.xml.stax`, `java.util.zip`, `com.bea.staxb.runtime.internal`, `weblogic.diagnostics.instrumentation.engine.bas`, `sun.misc`, `java.util.regex`, `com.bea.objectweb.asm`, `weblogic.diagnostics.instrumentation`, `java.nio`, `java.beans`, and `weblogic.xml.bebe.scanner`. The top class is `weblogic.diagnostics.image.descriptor.Instru`.
- Method Profiling View:** A heatmap visualization showing the execution of various methods. A tooltip is visible over a cell, displaying: `name: XMLWriterBase.writeCharacters(String), value: 10`.
- Stack Trace:** Shows the call stack for the selected method, with the following entries and counts:

Stack Trace	Count
<code>void weblogic.diagnostics.image.descriptor.InstrumentationImageSourceBeanImpl.setInstrumentationEvents(InstrumentationEv...</code>	790
<code>↑ void weblogic.diagnostics.image.descriptor.InstrumentationImageSourceBeanImpl.addInstrumentationEvent(InstrumentationEv...</code>	790
<code>↑ InstrumentationEventBean weblogic.diagnostics.image.descriptor.InstrumentationImageSourceBeanImpl.createInstrumentationEvent()</code>	790
<code>↑ void weblogic.diagnostics.instrumentation.InstrumentationImageSource.writeRecentEvents(InstrumentationImageSourceBean)</code>	790
<code>↑ void weblogic.diagnostics.instrumentation.InstrumentationImageSource.createDiagnosticImage(OutputStream)</code>	790
<code>↑ void weblogic.diagnostics.image.ImageSourceWork.run()</code>	790
<code>↑ void weblogic.work.SelfTuningWorkManagerImpl\$WorkAdapterImpl.run()</code>	790
<code>↑ void weblogic.work.ExecuteThread.execute(Runnable)</code>	790
<code>↑ void weblogic.work.ExecuteThread.run()</code>	790

Source: <https://github.com/thegreystone/jmc-flame-view> (Marcus Hirt)

Netflix: FlameScope (2018)



Source: <https://netflixtechblog.com/netflix-flamescope-a57ca19d47bb> (Brendan Gregg, Martin Spier)

Netflix: FlameCommander (2019)

The screenshot displays the FlameCommander web application interface. At the top, there's a navigation bar with 'FLAMECOMMANDER', 'Home', 'Tools', and a search bar. Below this, a 'CPU Profile' section features a 'New CPU Profile' button. A table titled 'All Profiles' lists several profiles with columns for Date, Instance/Container, Account, Region, and Status. An inset window shows a detailed CPU profile for PID #13547 (java) and PID #2001 (java), displaying flame graphs. A second inset window provides a detailed view of a flame graph, showing a stack of Java methods. A 'How to interpret differential flame graphs' section is also visible, explaining the color coding and sampling methods used in the graphs.

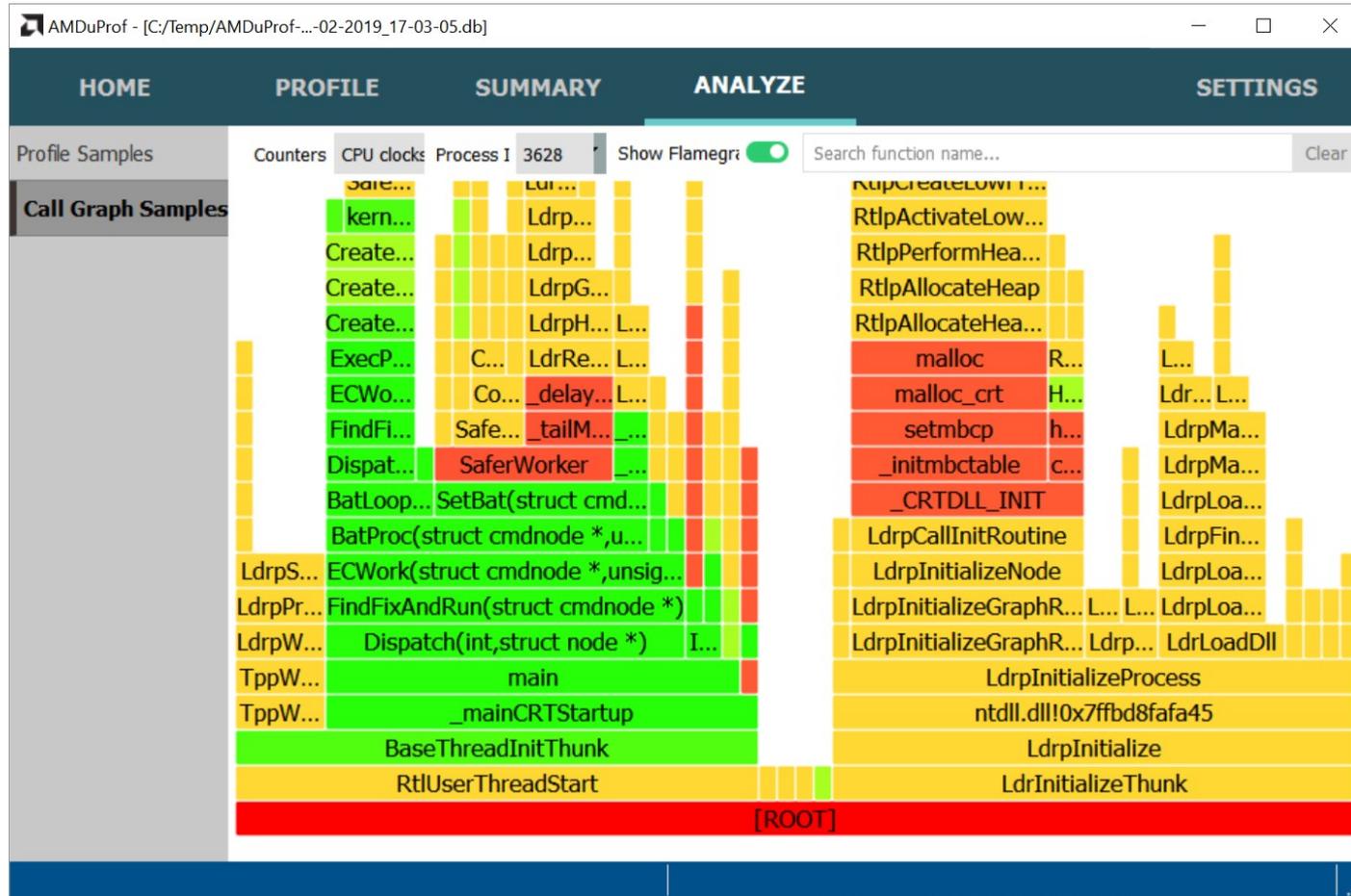
Date	Instance / Container	Account	Region	Status
04/19/2020 → 05/19/2020				
May 19, 2020 11:36 AM	i-Office2f4634a604d		EU	
May 19, 2020 11:33 AM	i-0b817e7970cf26f03		EU	

How to interpret differential flame graphs

- Differential flame graph is identical to the **flame** flame graph (has the same shape and sample count).
- It uses colors to highlight the difference between **flame** and **flame**.
- Blue** frames mean less samples in the **flame** flame graph.
- Red** frames mean more samples in the **flame** flame graph.
- Green** frames mean no difference between **flame** and **flame**.
- Ellided flame graph has only the stacks present in the **flame** flame graph, but not in **flame**. Colors have no meaning in this case.
- Balance Profiles will apply a weight to the comparison so both profiles have the same number of samples.

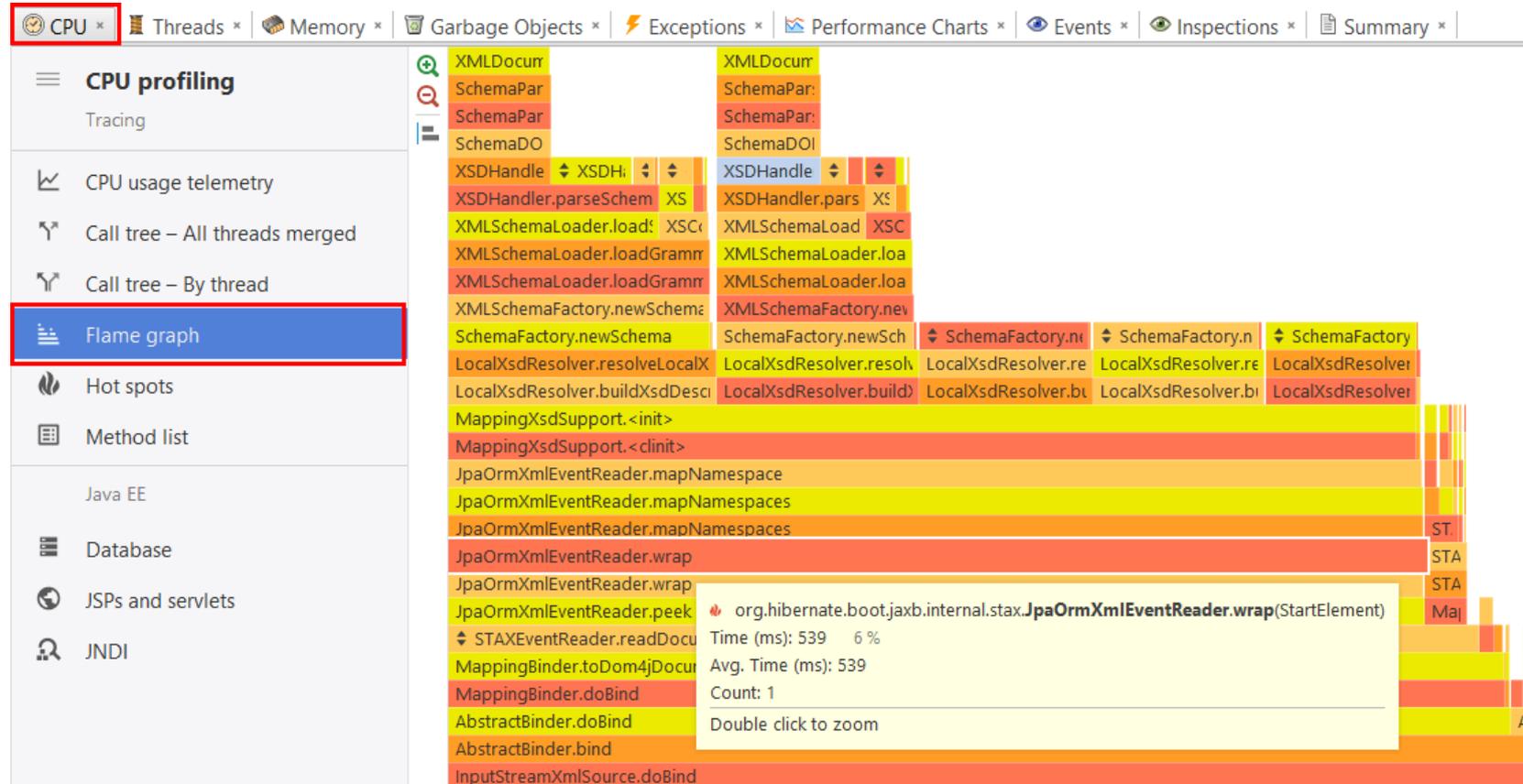
Source: <https://www.youtube.com/watch?v=L58GrWcrD00> (Martin Spier, Jason Koch, Susie Xia, Brendan Gregg) 24

AMD: uProf (2019)



Source: <https://developer.amd.com/amd-uprof/?sf215410082=1>

Java: YourKit (2019)



Source: https://www.yourkit.com/docs/java/help/cpu_flame_graph.jsp

Java: IntelliJ IDEA (2019)

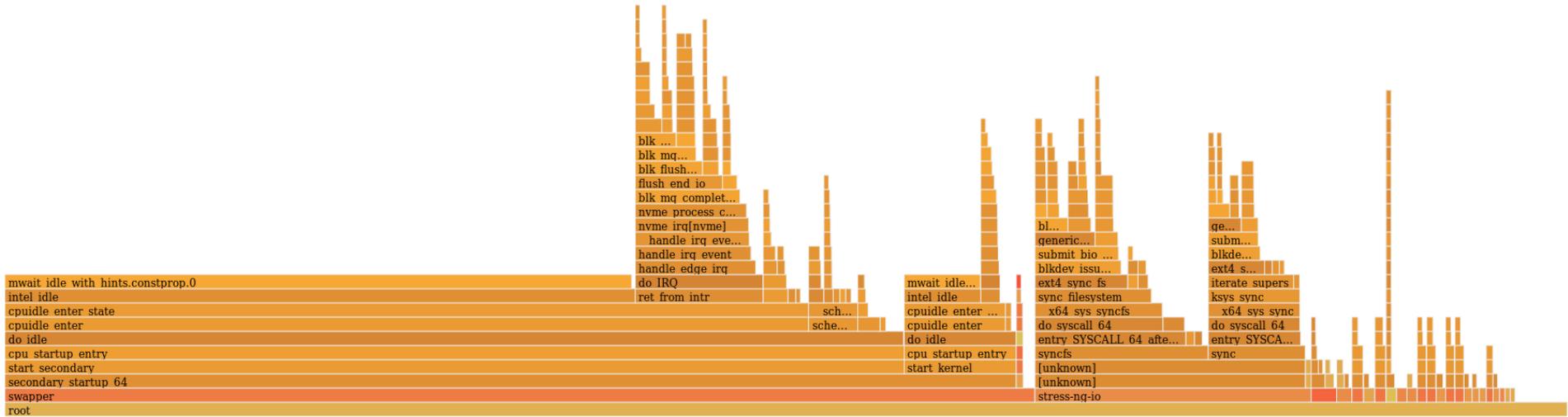
The screenshot displays the IntelliJ IDEA Profiler interface. At the top, there are two tabs: "[Async Profiler] Memory 54646 Scheduler" and "Java Flight Recorder 54666 Scheduler". The "Sampling Call Tree" view is active, showing a list of methods. A tooltip is visible over the method `java.lang.invoke.LambdaForm.createIdentityForms`, indicating it has 1 sample, 100.00% of its parent, and 33.33% of all samples. The interface includes a sidebar with "All threads merged" and "JFR request timer id=6", and a bottom bar with "4: Run", "6: TODO", "Profiler", "Terminal", "Services", and "Event Log".

Method	Parent
<code>java.lang.invoke.DirectMethodHandle.make</code>	
<code>j.l.i.DirectMethodHandle.make</code>	
<code>j.l.i.DirectMethodHandle.make</code>	
<code>j.l.i.DirectMethodHandle.make</code>	
<code>java.lang.invoke.LambdaForm.<clinit></code>	
<code>java.lang.invoke.LambdaForm.createIdentityForms</code>	<code>.Scheduler.main</code>
<code>java.lang.invoke.BoundMethodHandle.<clinit></code>	<code>java.lang.invoke.MethodHandle</code>
<code>j.l.i.BoundMethodHandle\$SpeciesData.<clinit></code>	<code>java.lang.invoke.MethodHandle</code>
<code>j.l.i.BoundMethodHandle\$SpeciesData.initForBootstrap</code>	<code>java.lang.invoke.CallSite.<clinit></code>
<code>j.l.i.BoundMethodHandle\$Factory.makeCtors</code>	<code>java.lang.invoke.MethodHandle</code>
<code>j.l.i.BoundMethodHandle\$Factory.makeCbmhCtor</code>	<code>j.l.i.MethodHandles\$Lookup.g</code>
<code>java.lang.invoke.MethodHandles\$Lookup.findStatic</code>	<code>j.l.i.MethodHandles\$Lookup.g</code>

Source: <https://blog.jetbrains.com/idea/2019/06/intellij-idea-2019-2-eap-4-profiling-tools-structural-search-preview-and-more/>

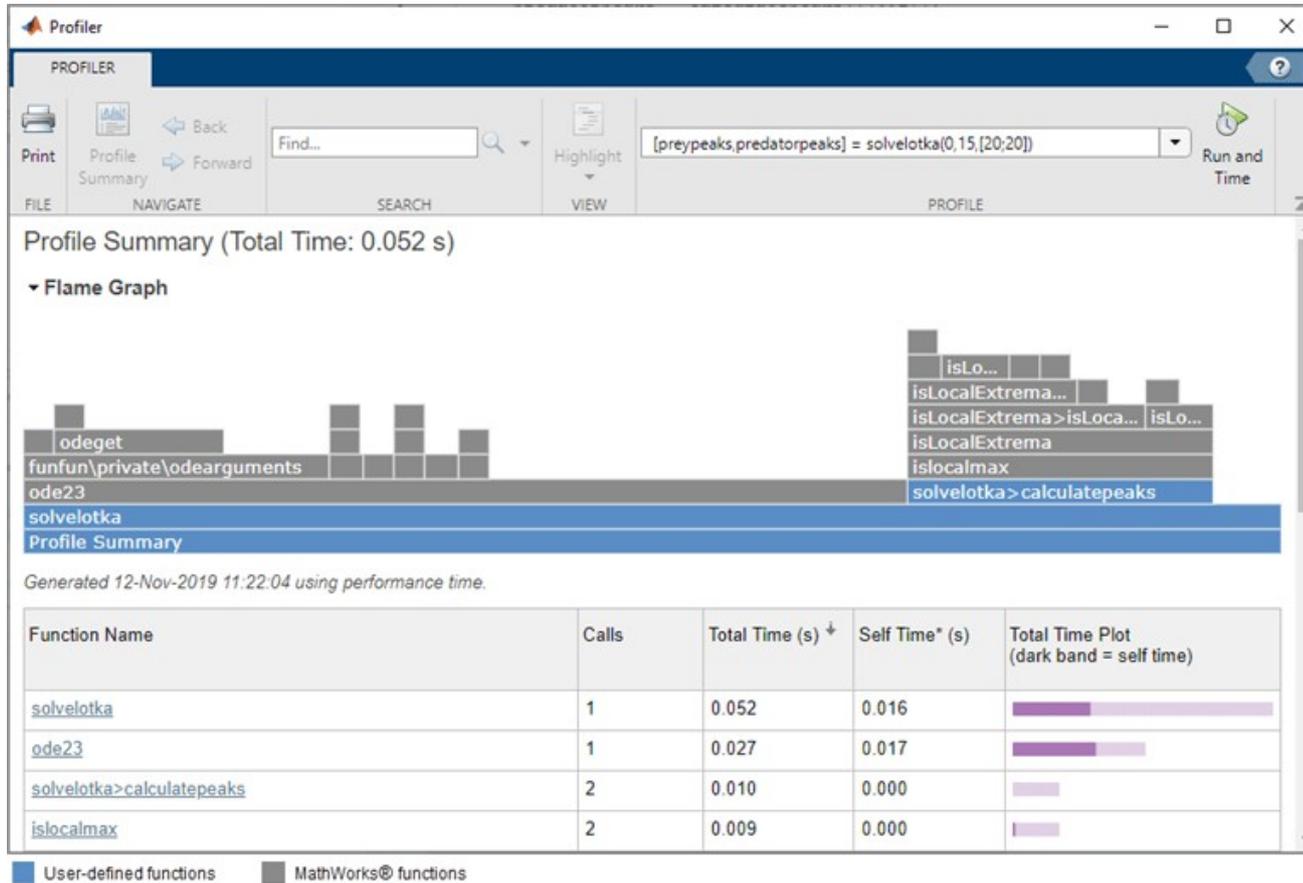
Linux: perf script flamegraph (2020)

Reset Zoom Clear Search

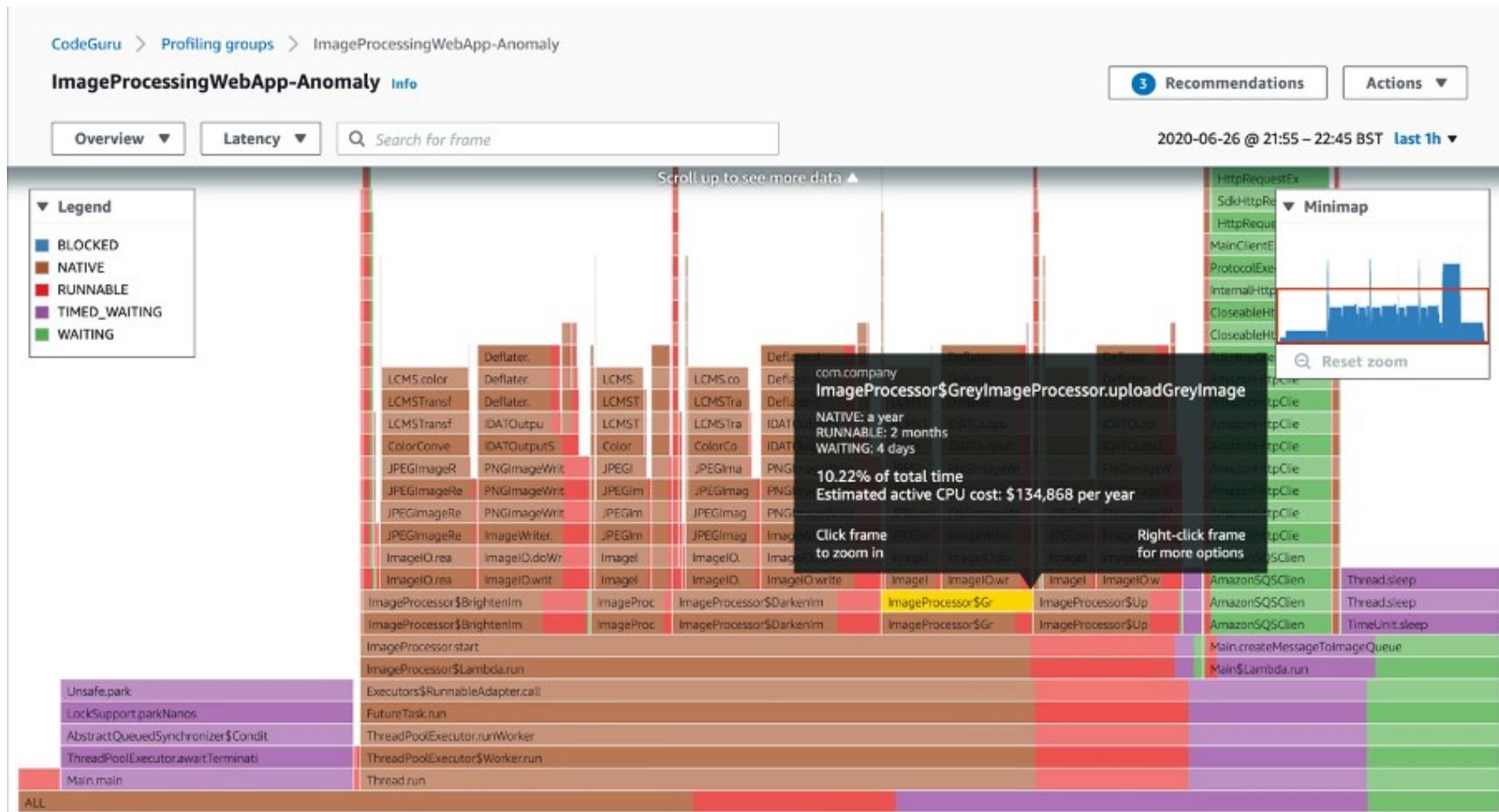


Source: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/getting-started-with-flamegraphs_monitoring-and-managing-system-status-and-performance (Andreas Gerstmayr)

MathWorks: MATLAB Profiler (2020)

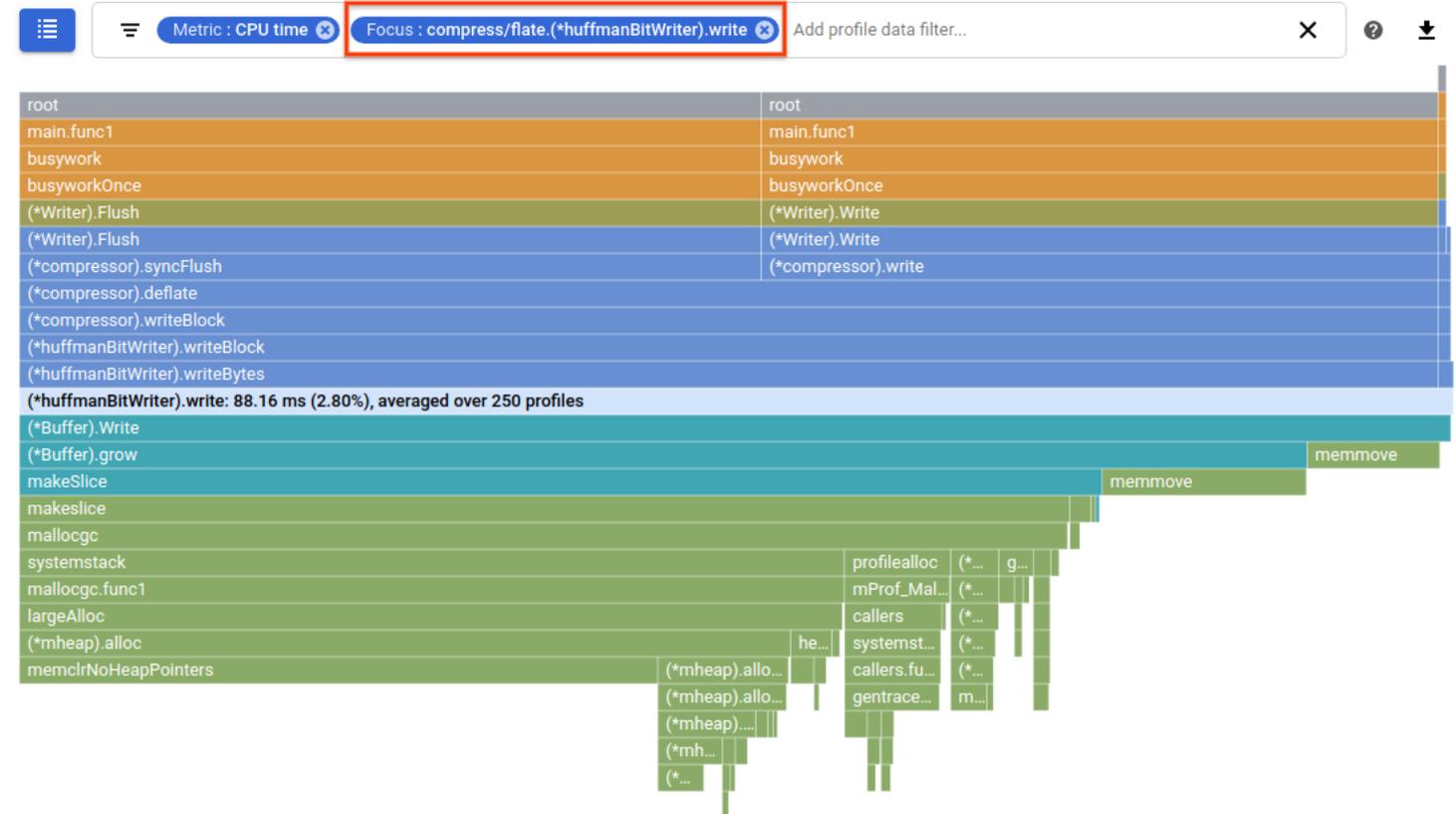


AWS: CodeGuru (2020)



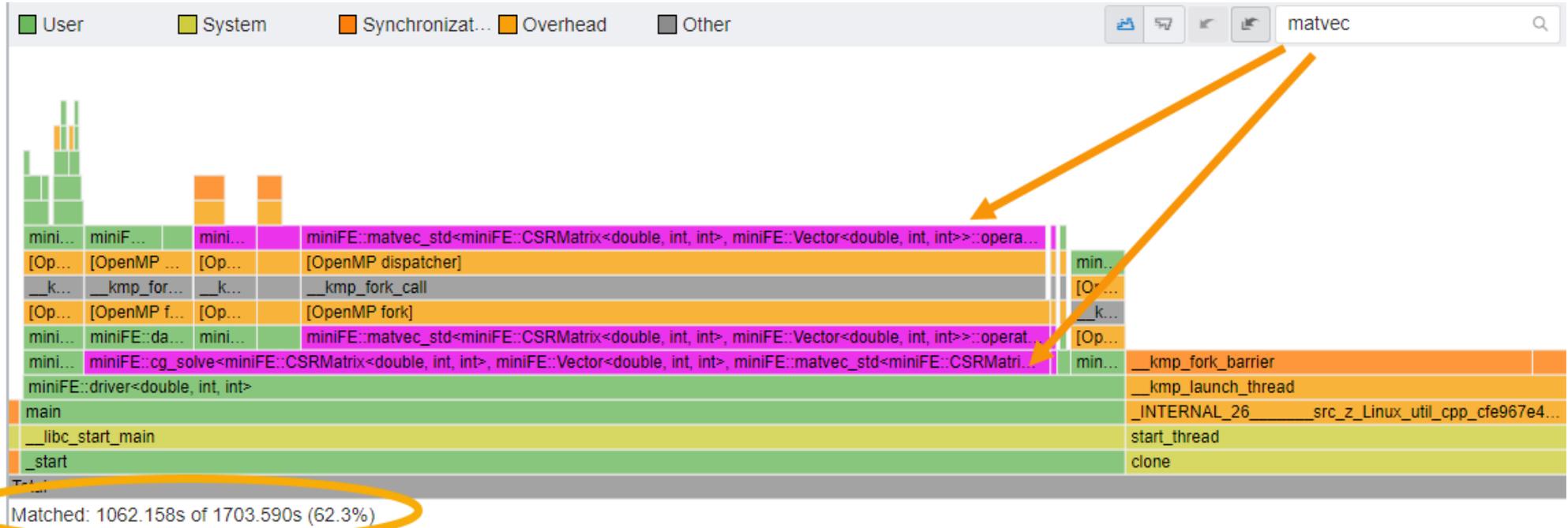
Source: <https://aws.amazon.com/codeguru/features/>

Google: Cloud Profiler (2020)



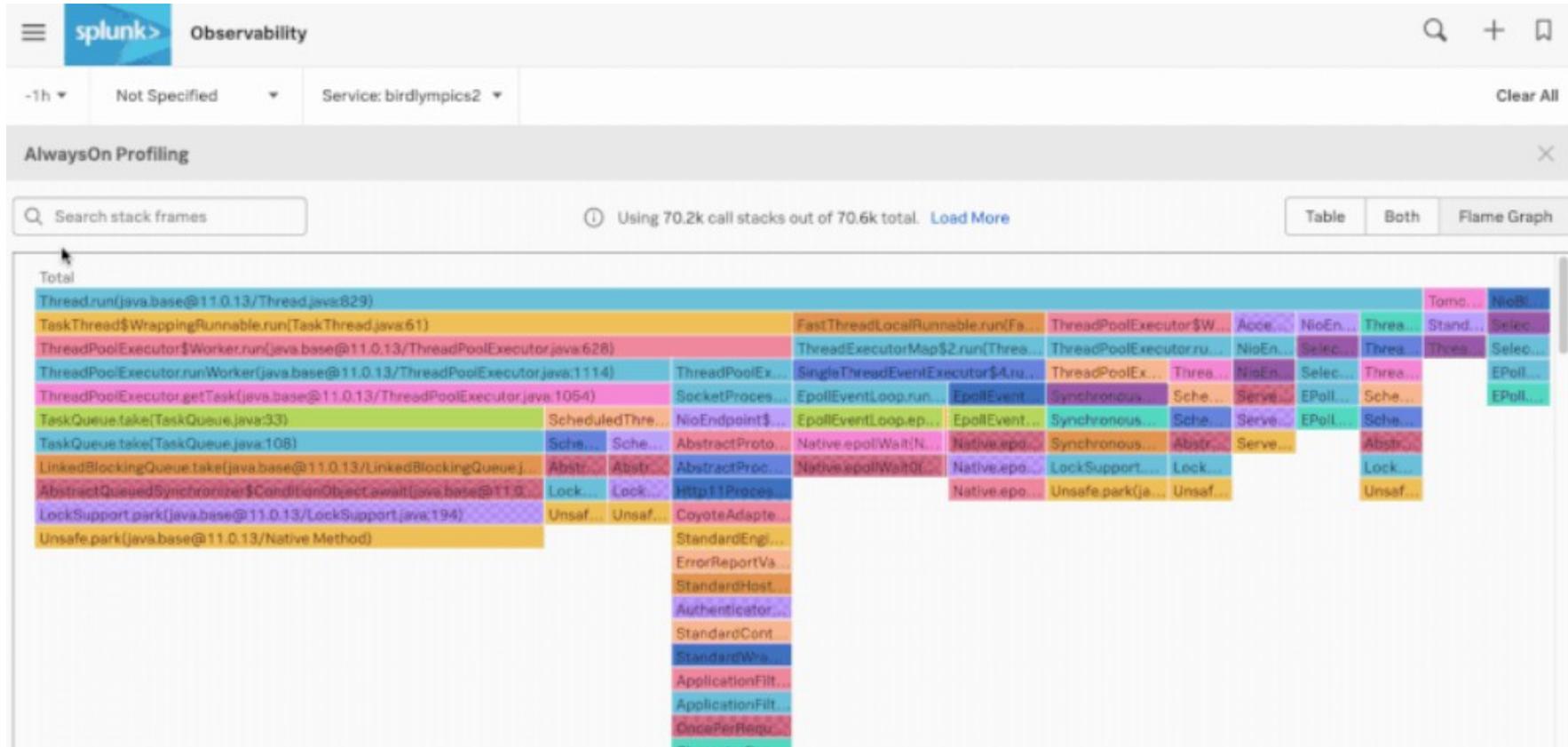
Source: <https://cloud.google.com/profiler/docs/focusing-profiles>

Intel: vTune (2021)



Source: <https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/user-interface-reference/window-flame-graph.html>

Splunk: AlwaysOn Profiling flame graph (2021)



Source: <https://docs.splunk.com/Observability/apm/profiling/using-the-flamegraph.html>

New Relic: flame graphs (2021)

New Relic ONE™

Services

flamegraph-s

Summary

MONITOR

Distributed tracing

Service map

Dependencies

Transactions

Databases

External services

JVMs

Threads

EVENTS

Errors

Violations

Deployments

Thread profiler

REPORTS

SLA

flamegraph-service (staging)

8f974d6c22ff/172.17.0.20

from Nov 2, 02:36 pm to Nov 2, 02:41 pm

No logs found

AVERAGE CPU: 11.2%

AVERAGE HEAP MEMORY: 247.03 MB

TOTAL GC PAUSE TIME: 0.5 sec

HEAP USED: 245.46 MB

HEAP SIZE: 2,048 MB

HEAP COMMITTED: 1,409 MB

Thread	Count
java.lang.Thread.run()V:834	1990
org.eclipse.jetty.util.thread.QueuedThreadPool\$Runner.run()V:938	996
sun.rmi.transport.tcp.TCPTransport\$AcceptLoop.run()V:366	746
org.eclipse.jetty.io.Managed...	...
org.eclipse.jetty.util.thread.st...	...
org.eclipse.jetty.io.Managed...	...
sun.nio.ch.SelectorImpl.sele...	...
sun.nio.ch.SelectorImpl.lock...	...
sun.nio.ch.EPollSelectorImpl...	...
sun.nio.ch.EPoll.wait(JII):-1...	...

User CPU Usage %

Since Nov 2, 02:36 pm until Nov 2, 02:41 pm

Machine CPU Usage %

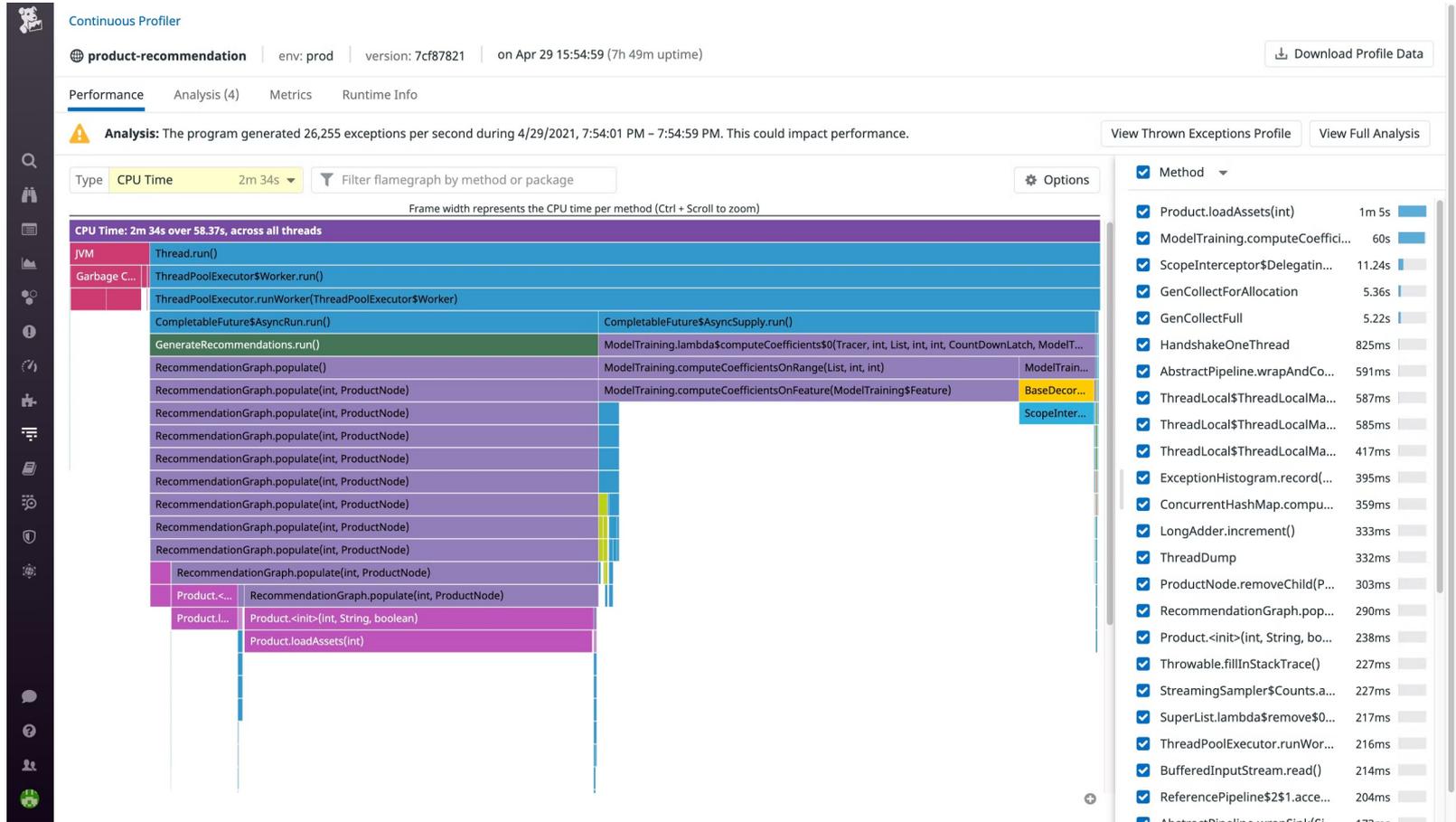
Since Nov 2, 02:36 pm until Nov 2, 02:41 pm

GC Heap Sizes

Since Nov 2, 02:36 pm until Nov 2, 02:41 pm

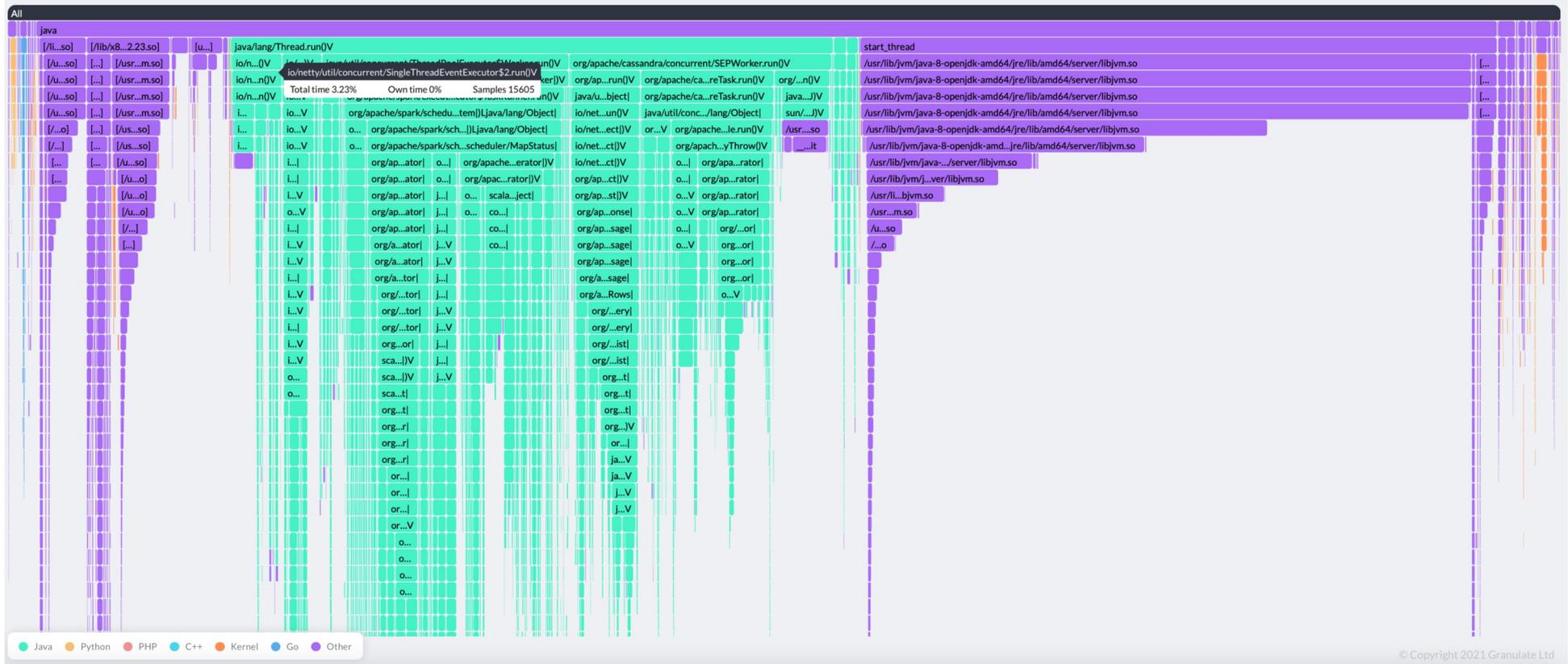
Source: <https://docs.newrelic.com/whats-new/2021/07/whats-new-july-8-realtime-profiling-java/>

DataDog: profiling flame graph (2021)



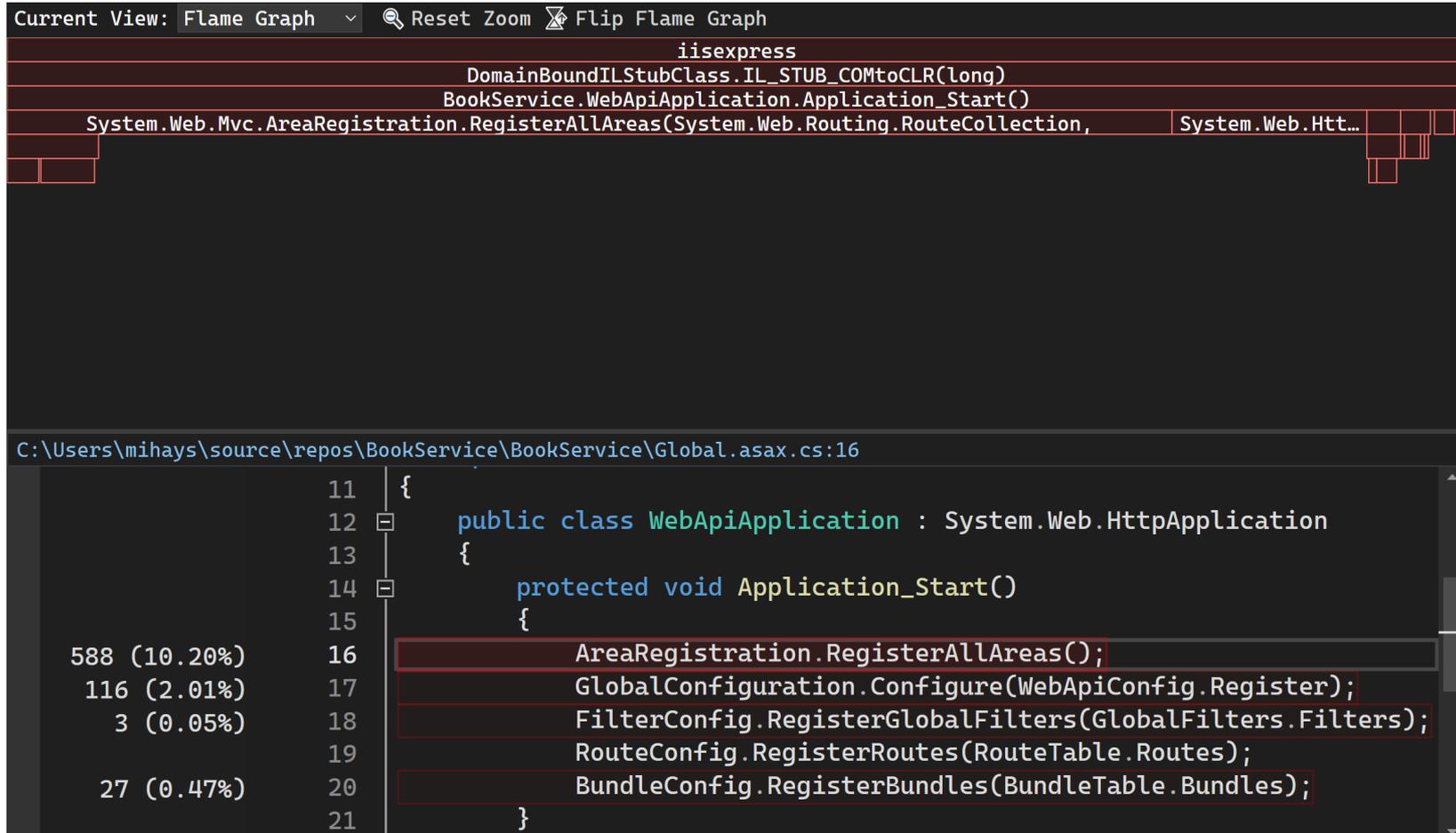
Source: <https://www.datadoghq.com/knowledge-center/distributed-tracing/flame-graph/>

Granulate: gprofiler (2022; now Intel)



Source: <https://docs.gprofiler.io/about-gprofiler/gprofiler-features/views/flame-graph>

Microsoft Visual Studio: Flame Graph (2022)



Source: <https://learn.microsoft.com/en-us/visualstudio/profiling/flame-graph>

GrafanaLabs: Grafana flame graph (2022)



Source: <https://grafana.com/docs/grafana/next/panels-visualizations/visualizations/flame-graph>

Flame Graph Adoption

Implementations: **>80**

Related open source projects: **>400**

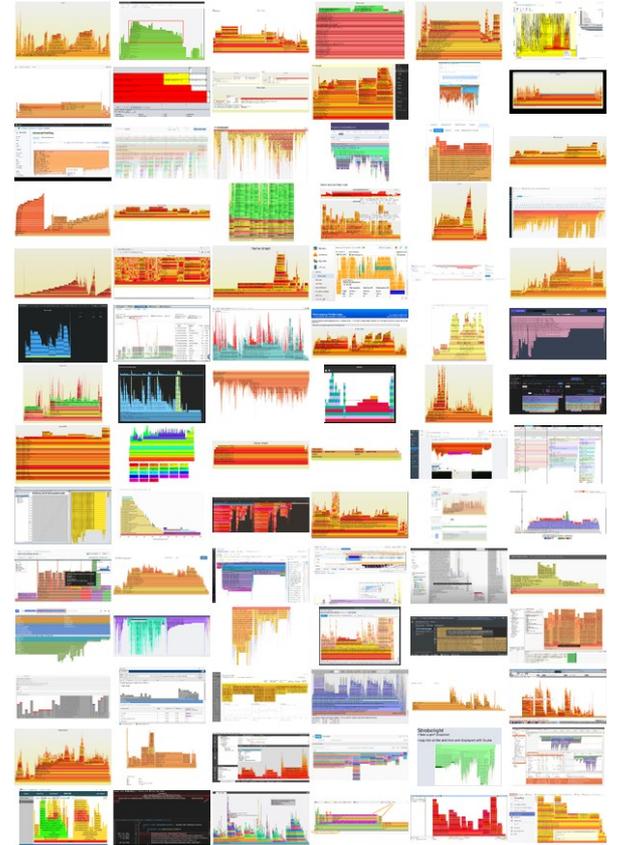
Commercial product adoptions: **>30**

New startups: **4** (so far)

Startup exits: **1** (so far)

Industry investment: **>AUD\$1B**

End users: ? (a lot)



2. CPU PROFILING

An Introduction to Flame Graphs

Stack Traces

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

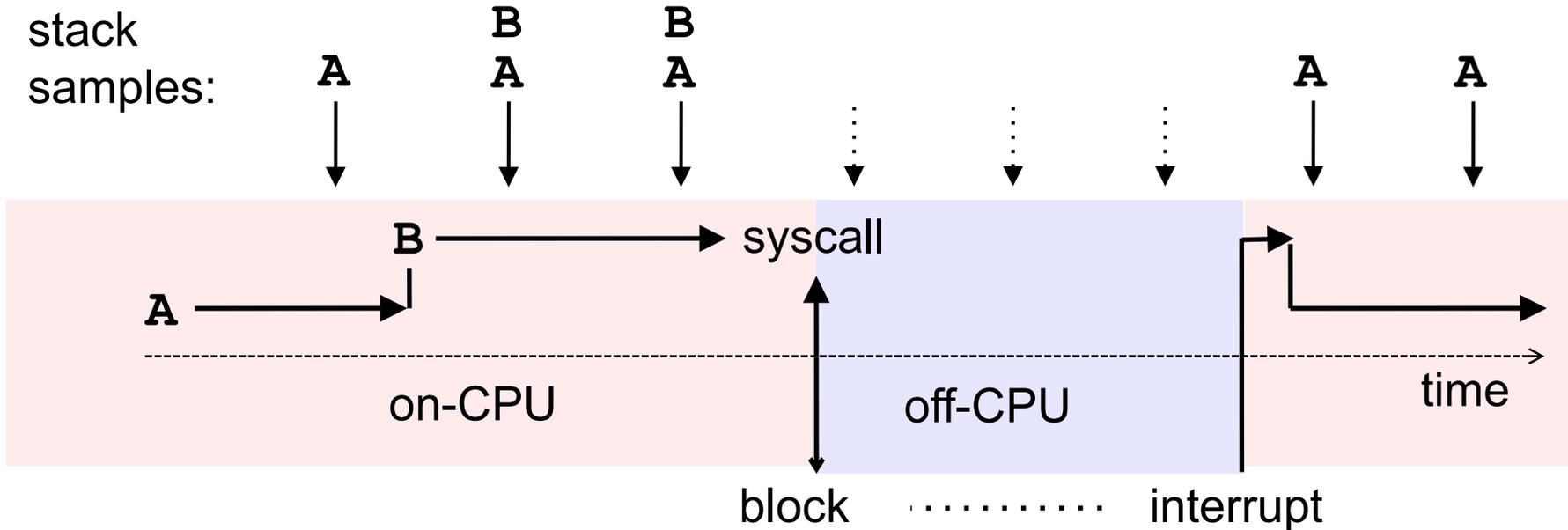
```
$ jstack 1819  
  
[...]  
  
"main" prio=10 tid=0x00007fff304009000  
nid=0x7361 runnable [0x00007fff30d4f9000]  
  
    java.lang.Thread.State: RUNNABLE  
  
        at Func_abc.func_c(Func_abc.java:6)  
        at Func_abc.func_b(Func_abc.java:16)  
        at Func_abc.func_a(Func_abc.java:23)  
        at Func_abc.main(Func_abc.java:27)
```

↓
running
parent
g.parent
g.g.parent

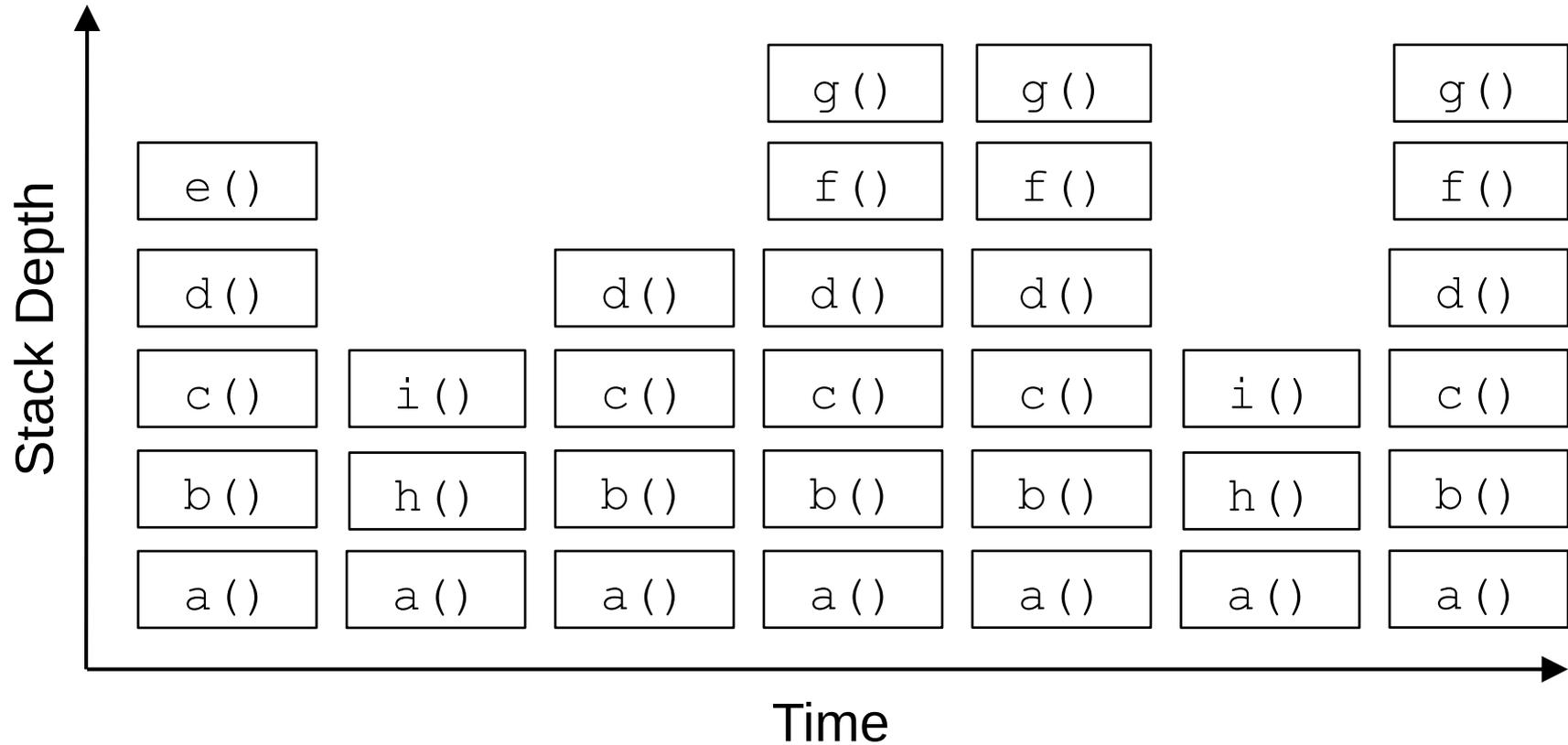
CPU Profiling

Record stacks at a timed interval

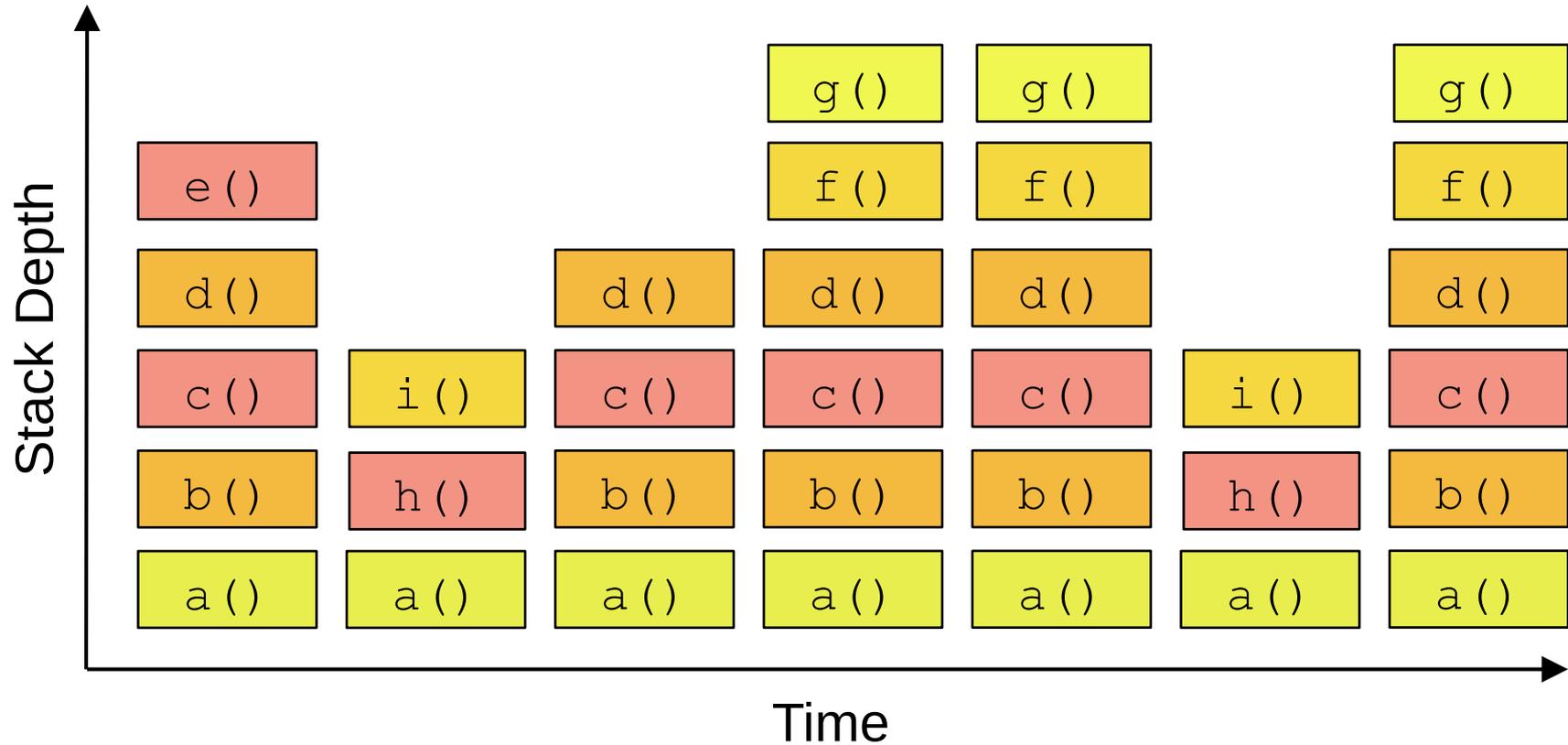
- Pros: Low (deterministic) overhead
- Cons: Coarse accuracy, but usually sufficient



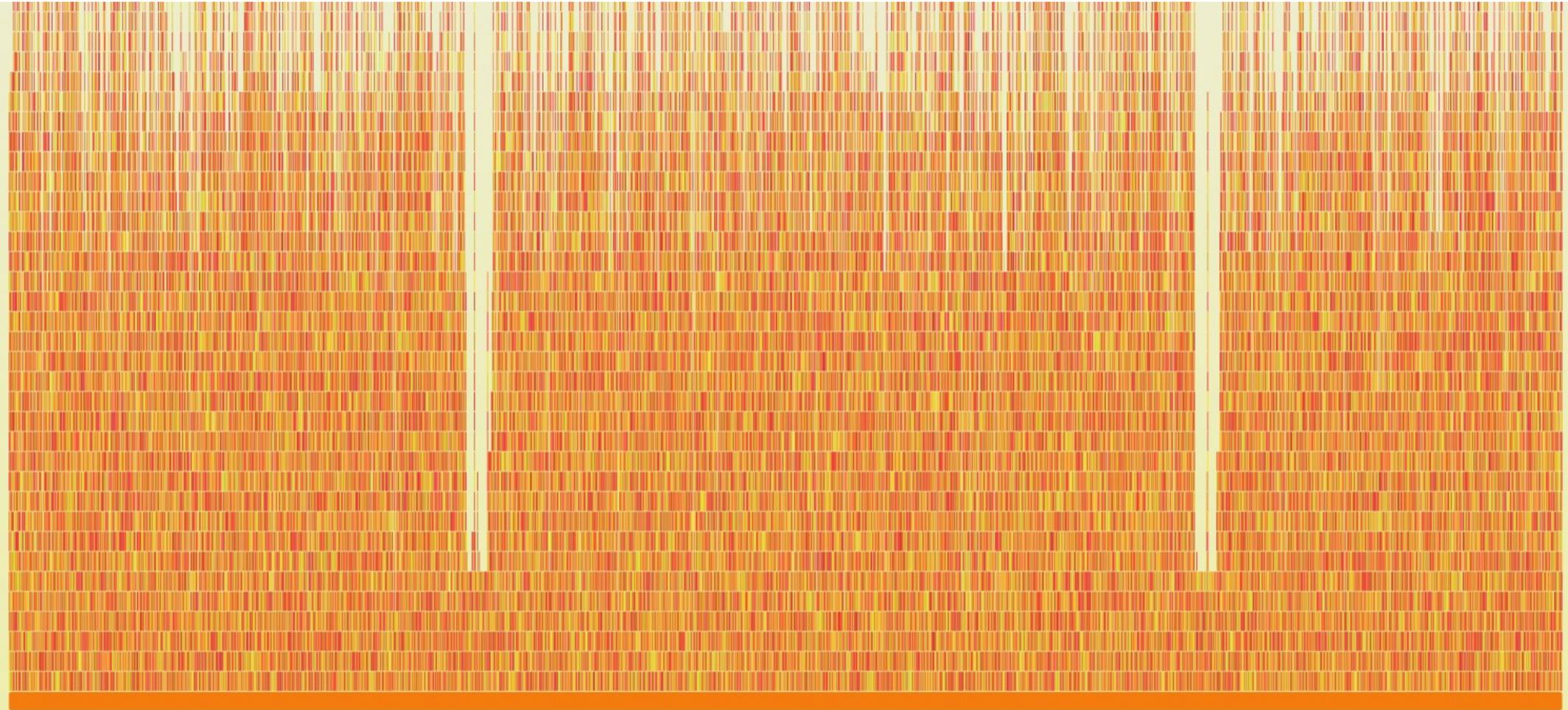
Stack Samples



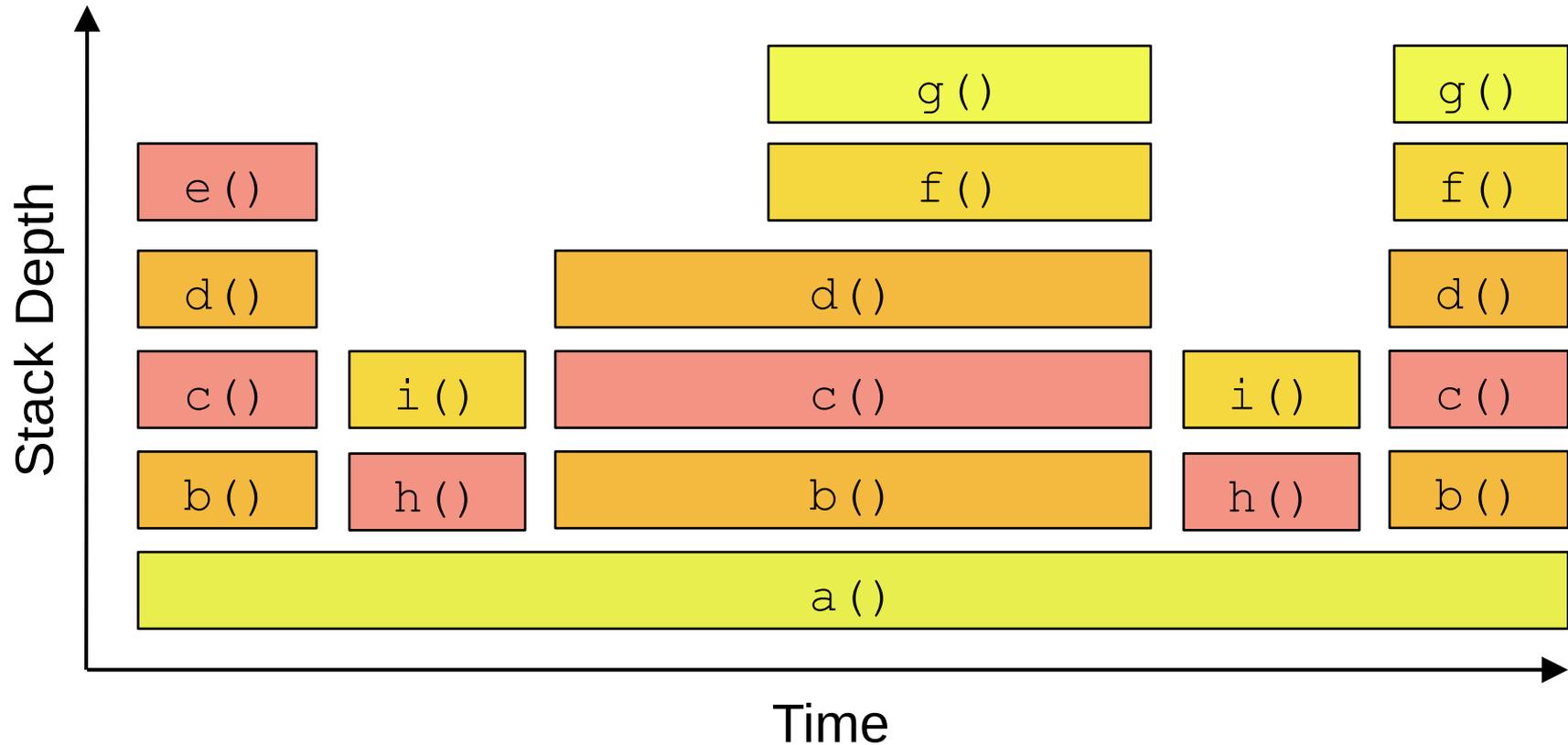
Stack Samples



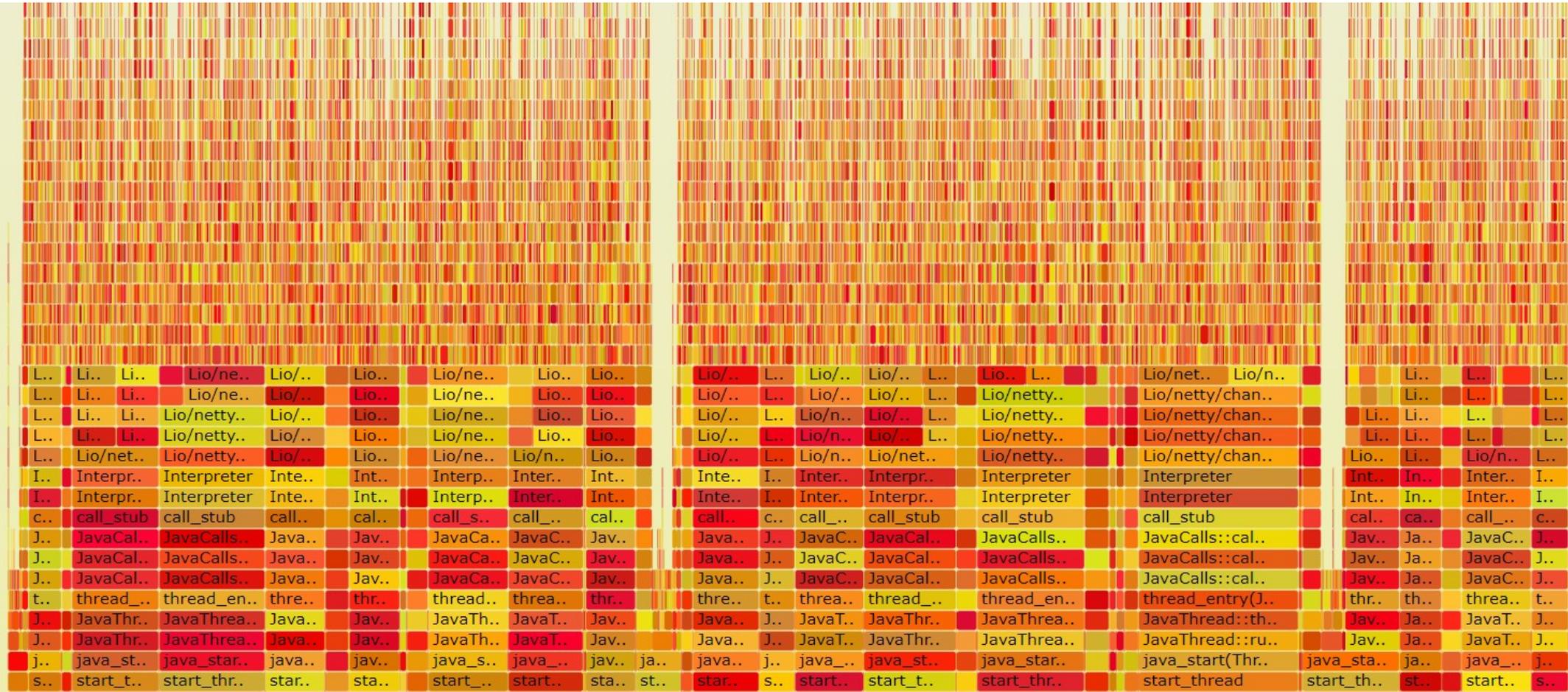
Example Profile (“hair graph”)



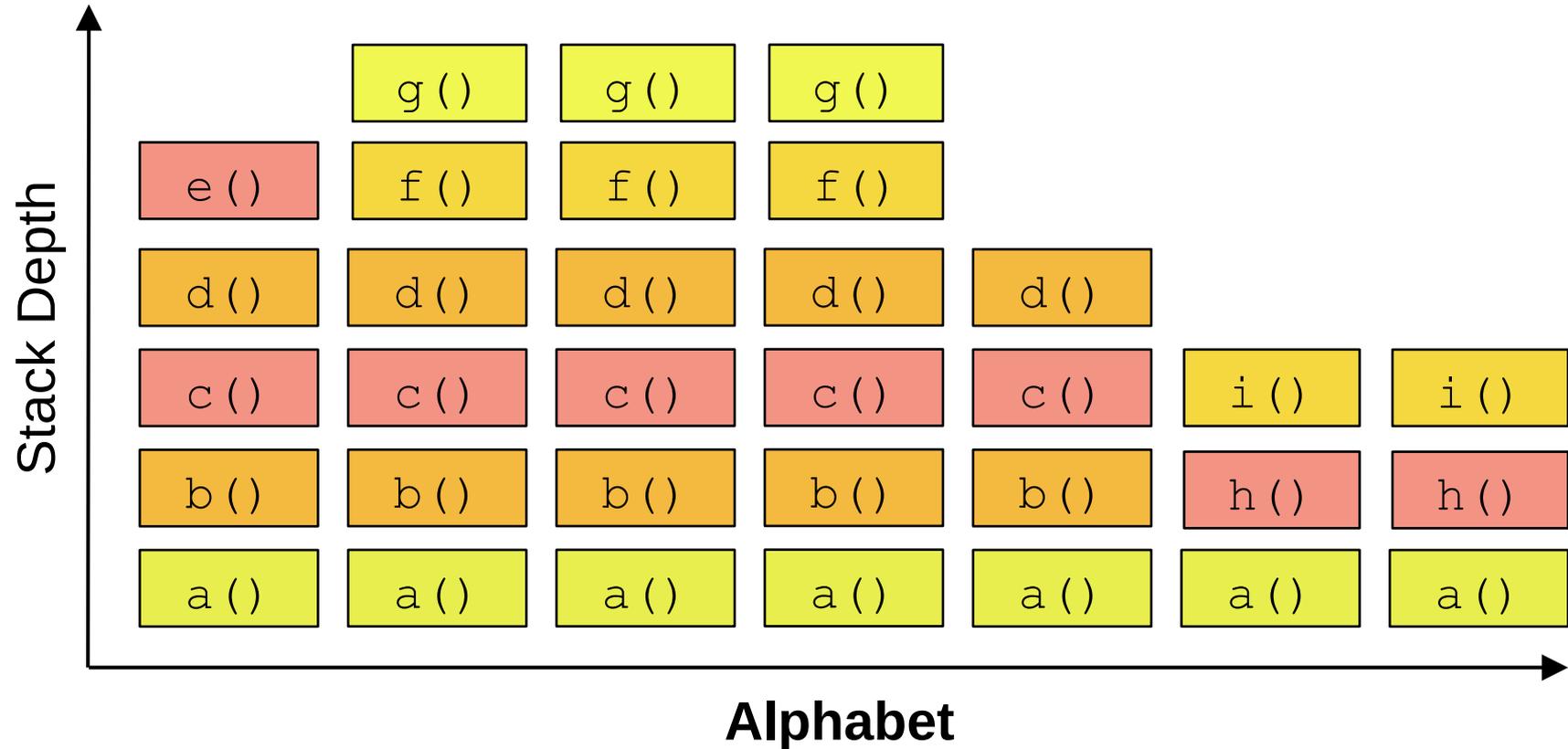
Stack Samples: Merged



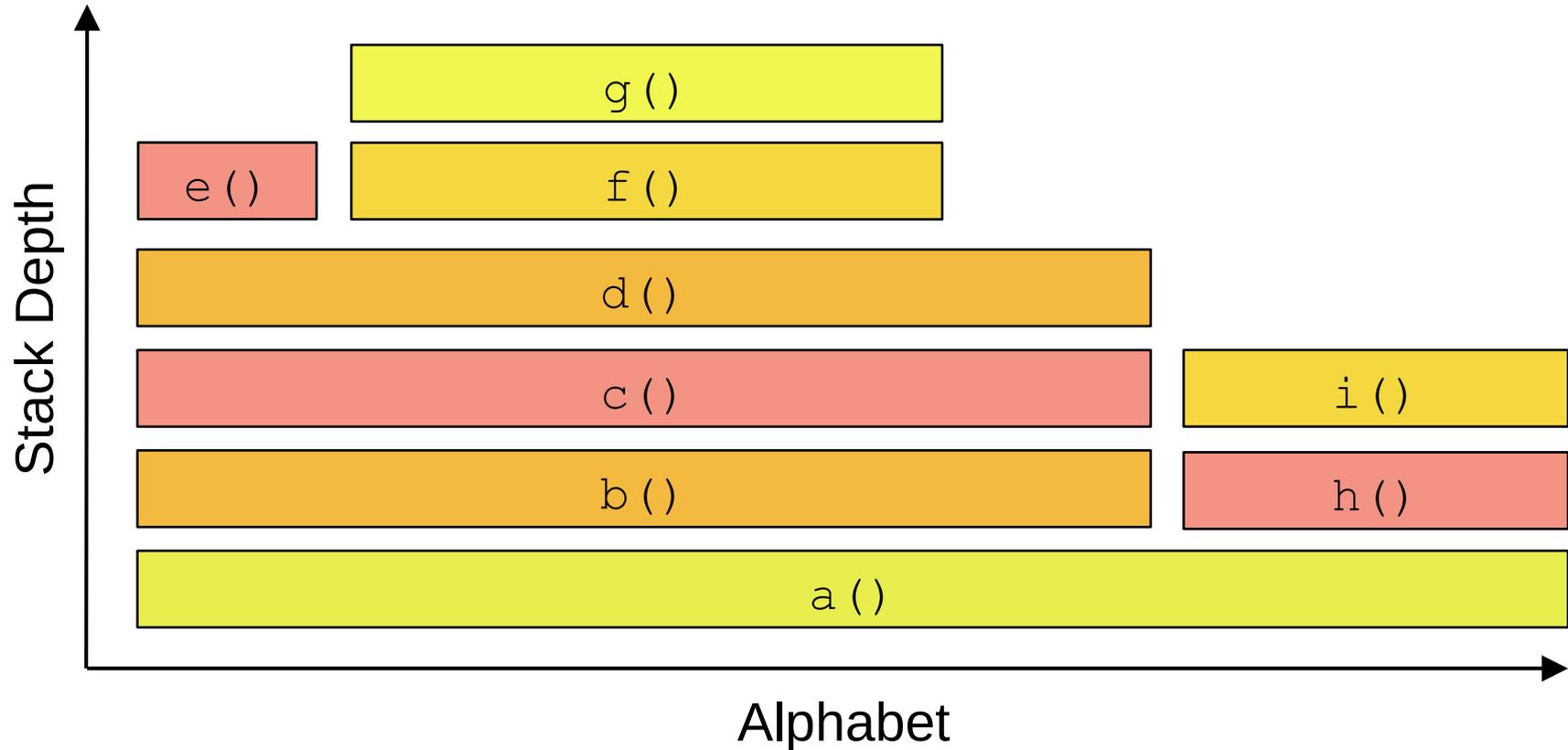
Example Profile: Merged



Alphabet Sort



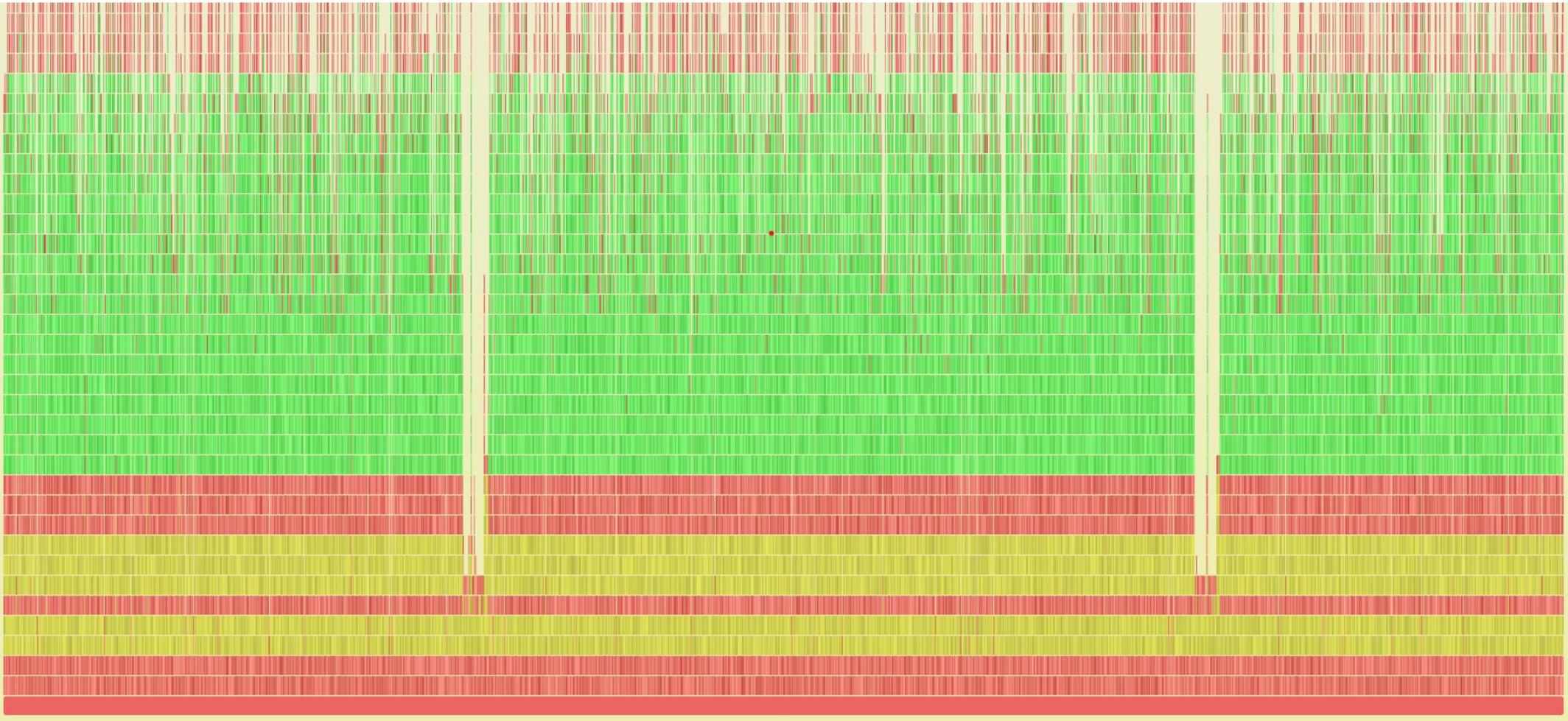
Alphabet Merged (“Flame Graph”)



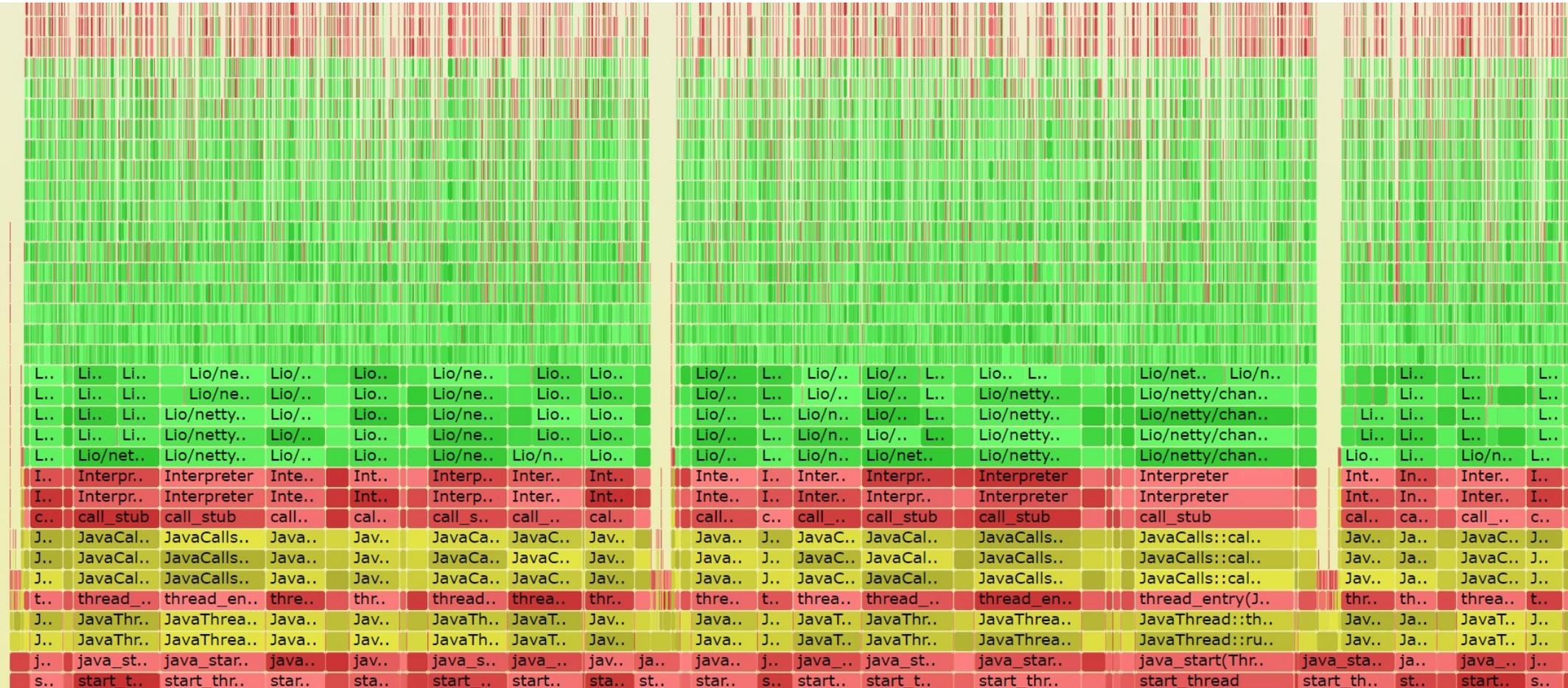
Example Profile: Flame Graph



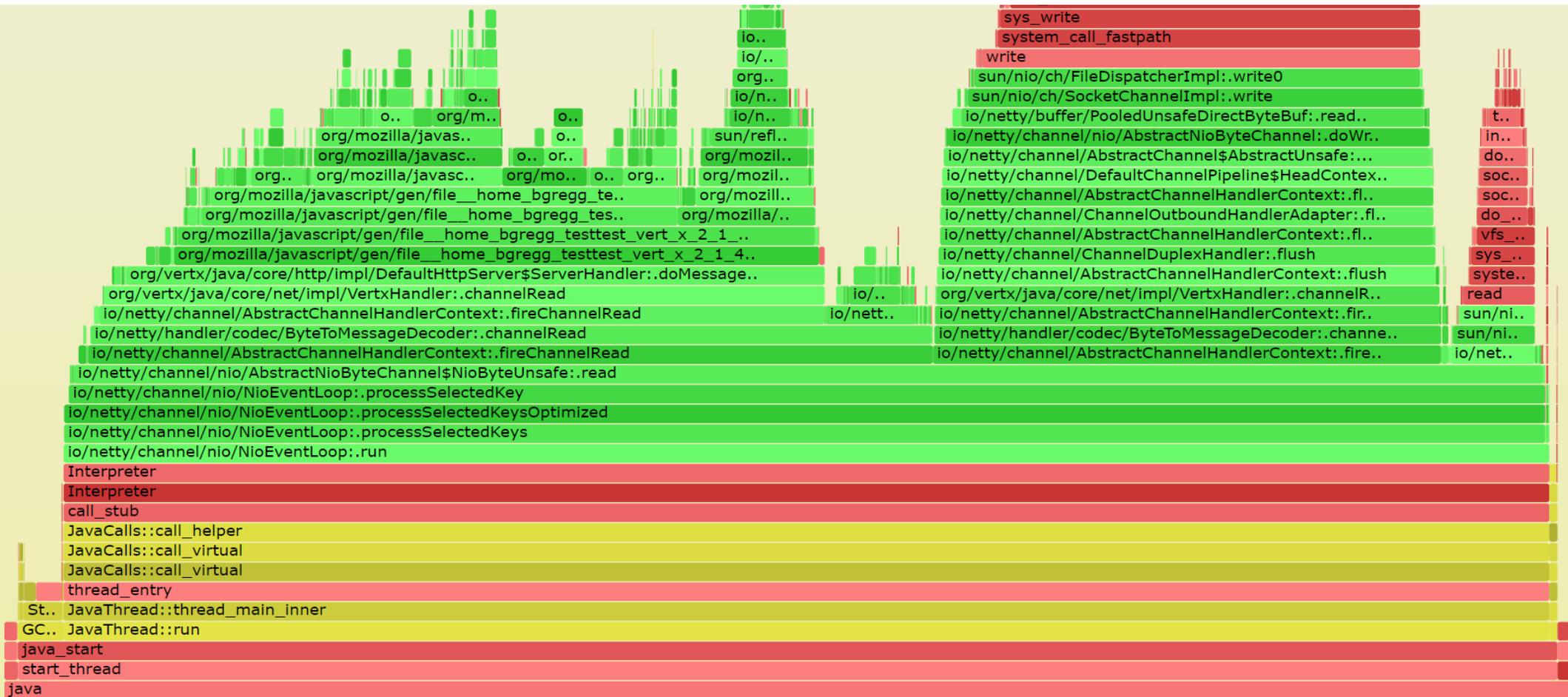
Replay 1/3: Time Columns



Replay 2/3: Time Merged (aka "Flame Chart")



Replay 3/3: Flame Graph



Origin (2011): CPU Profiling

```
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"/ {  
    @[ustack()] = count(); } tick-60s { exit(0); }'  
[... over 500,000 lines truncated ...]
```

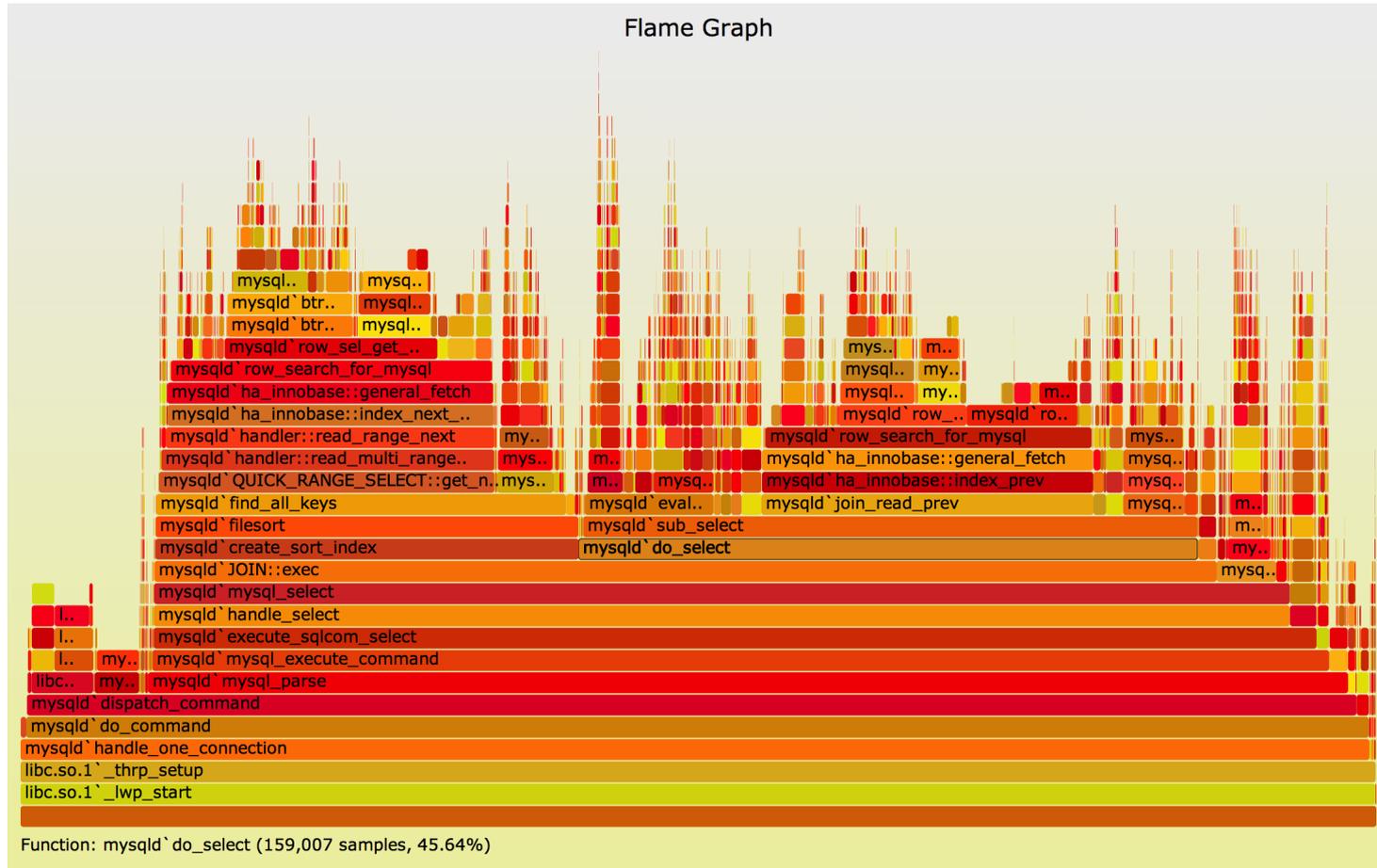
```
    libc.so.1`__prioset+0xa  
    libc.so.1`getparam+0x83  
    libc.so.1`pthread_getschedparam+0x3c  
    libc.so.1`pthread_setschedprio+0x1f  
    mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x9ab  
    mysqld`_Z10do_commandP3THD+0x198  
    mysqld`handle_one_connection+0x1a6  
    libc.so.1`_thrp_setup+0x8d  
    libc.so.1`_lwp_start  
4884
```

```
    mysqld`_Z13add_to_statusP17system_status_varS0_+0x47  
    mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67  
    mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222  
    mysqld`_Z10do_commandP3THD+0x198  
    mysqld`handle_one_connection+0x1a6  
    libc.so.1`_thrp_setup+0x8d  
    libc.so.1`_lwp_start  
5530
```

Full output

[The content of this block is intentionally obscured by a dense gray pattern, representing a full output that is not legible.]

... as a Flame Graph



Linux example: perf Profiling

```
# perf record -F 99 -ag -- sleep 30
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.745 MB perf.data (~119930 samples) ]
# perf report -n -stdio
[...]
```

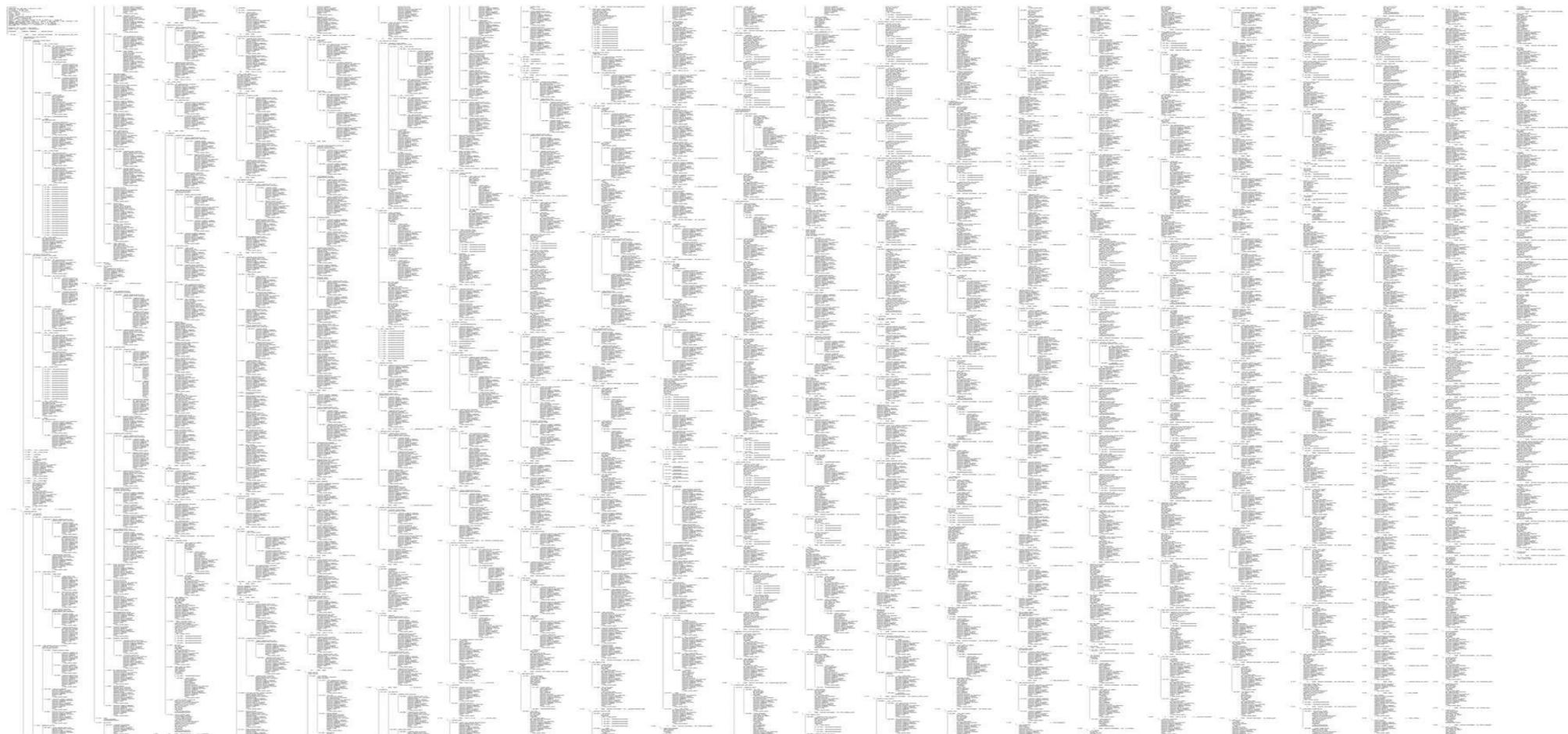
Overhead	Samples	Command	Shared Object	Symbol
20.42%	605	bash	[kernel.kallsyms]	[k] xen_hypercall_xen_version

```
|
--- 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
```

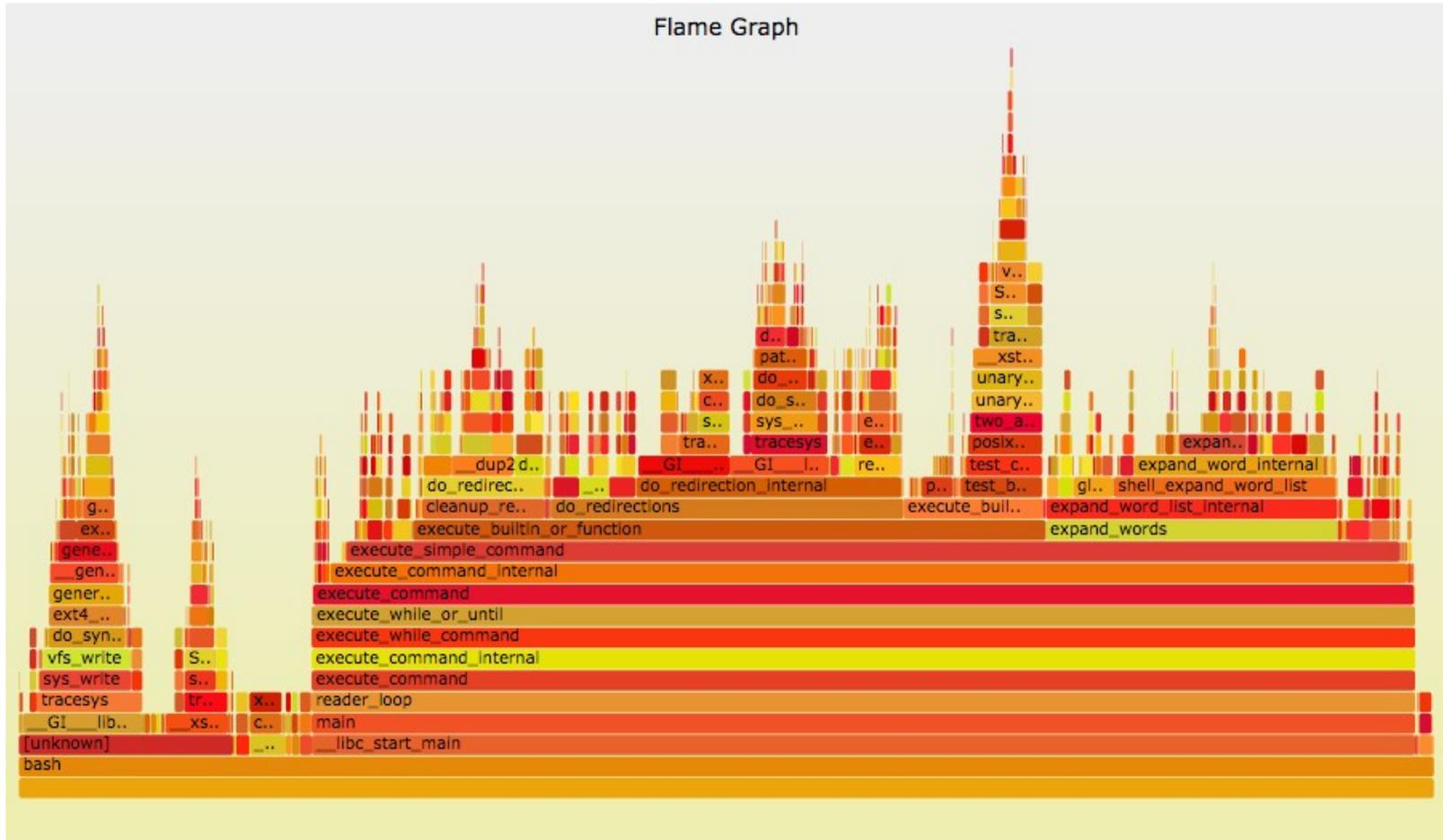
call tree
summary

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

Full perf Output



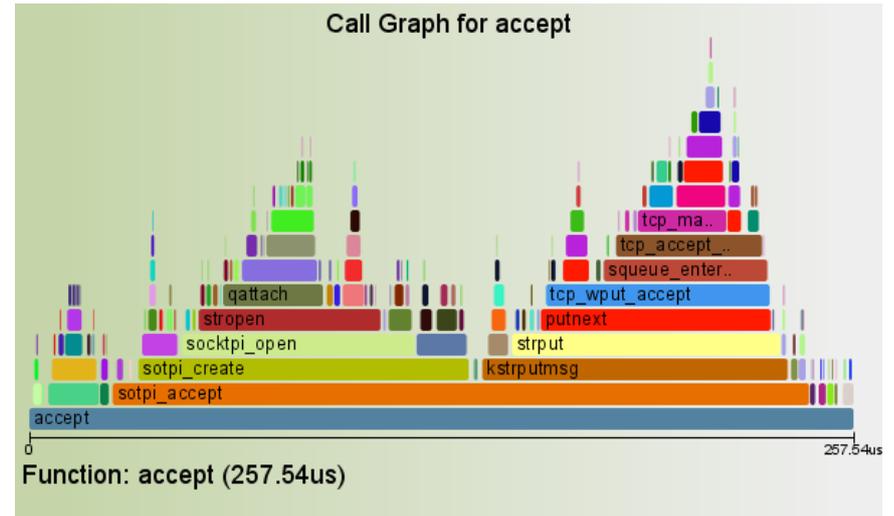
... as a Flame Graph



Inspiration

Neelakanth Nadgir's `function_call_graph.rb` (2007):

- It was inspired by Roch Bourbonnais's `CallStackAnalyzer`, which was inspired by Jan Boerhout's `vftrace`.
- The x-axis is time, and it shows a complete function trace.
- Flame graphs are different: The x-axis is the *population*, and they can show function traces or *stack samples*.



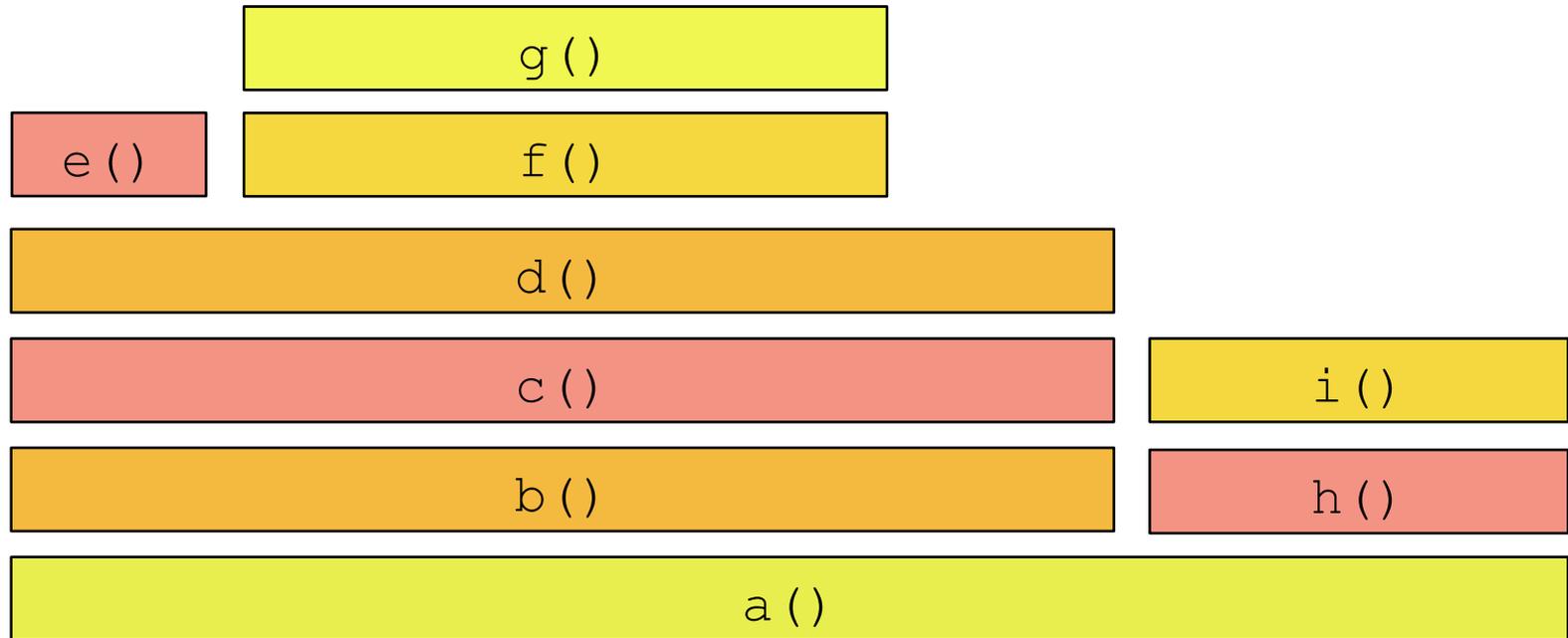
```
# more flamegraph.pl
[...]
```

This was inspired by Neelakanth Nadgir's excellent `function_call_graph.rb` program, which visualized function entry and return trace events. As Neel wrote: "The output displayed is inspired by Roch's `CallStackAnalyzer` which was in turn inspired by the work on `vftrace` by Jan Boerhout". See: https://blogs.oracle.com/realneel/entry/visualizing_callstacks_via_dtrace_and

```
[...]
```

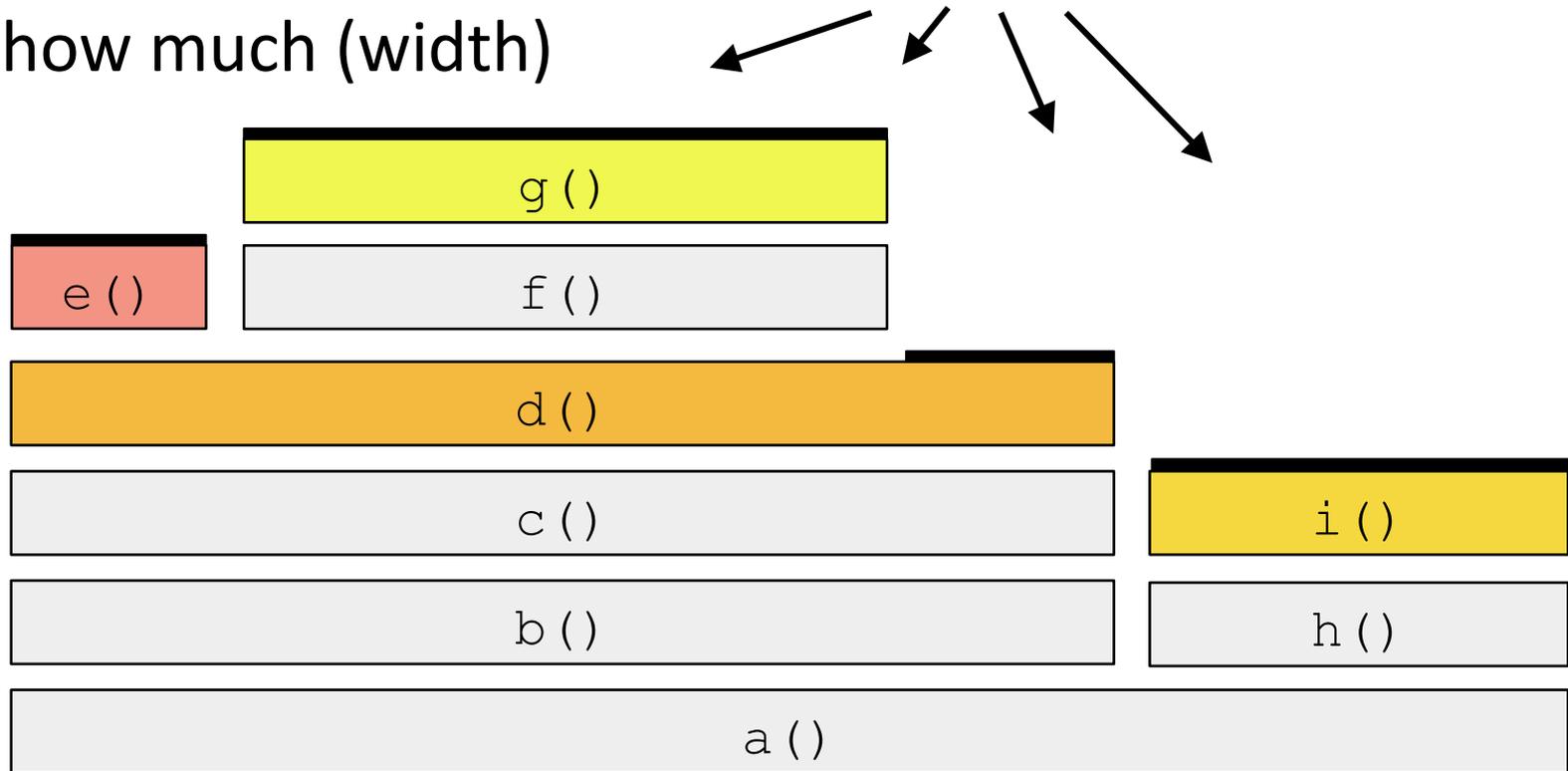
Image source: https://blogs.oracle.com/realneel/entry/visualizing_callstacks_via_dtrace_and

Flame Graph Interpretation



Flame Graph Interpretation (1/4)

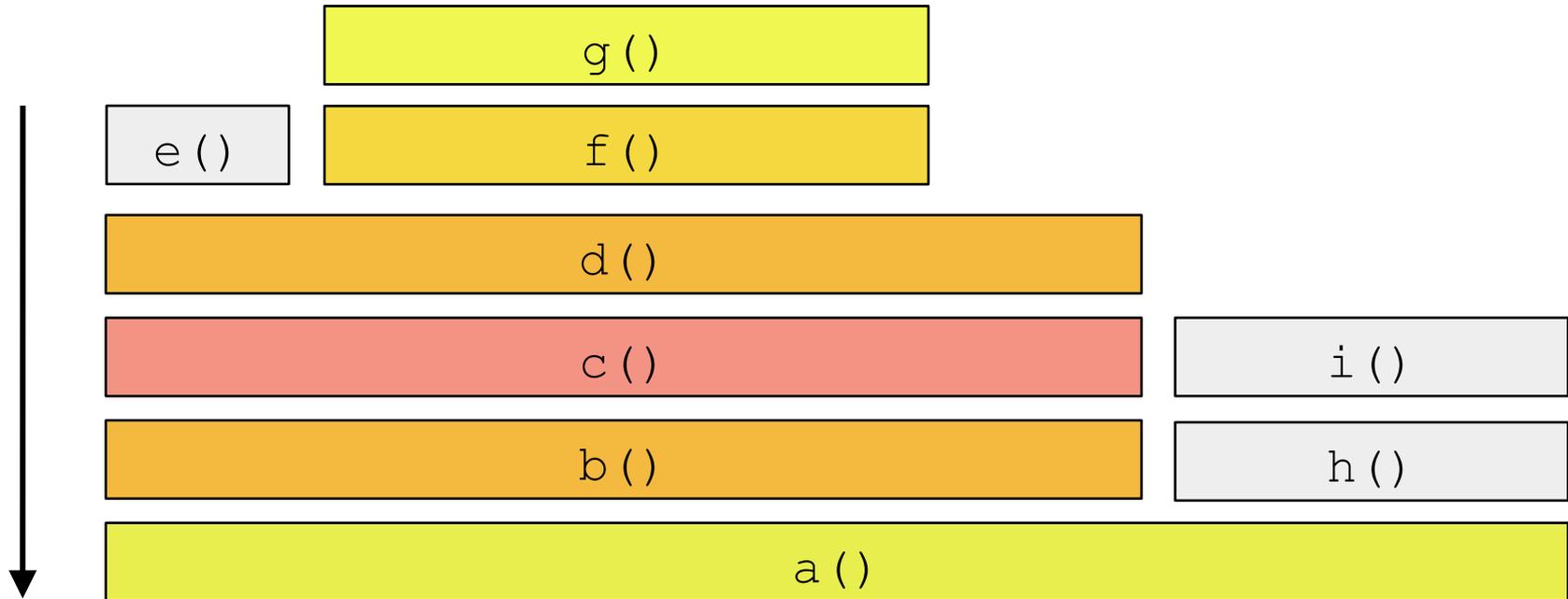
Top edge shows who is running on-CPU,
and how much (width)



Flame Graph Interpretation (2/4)

Top-down shows ancestry

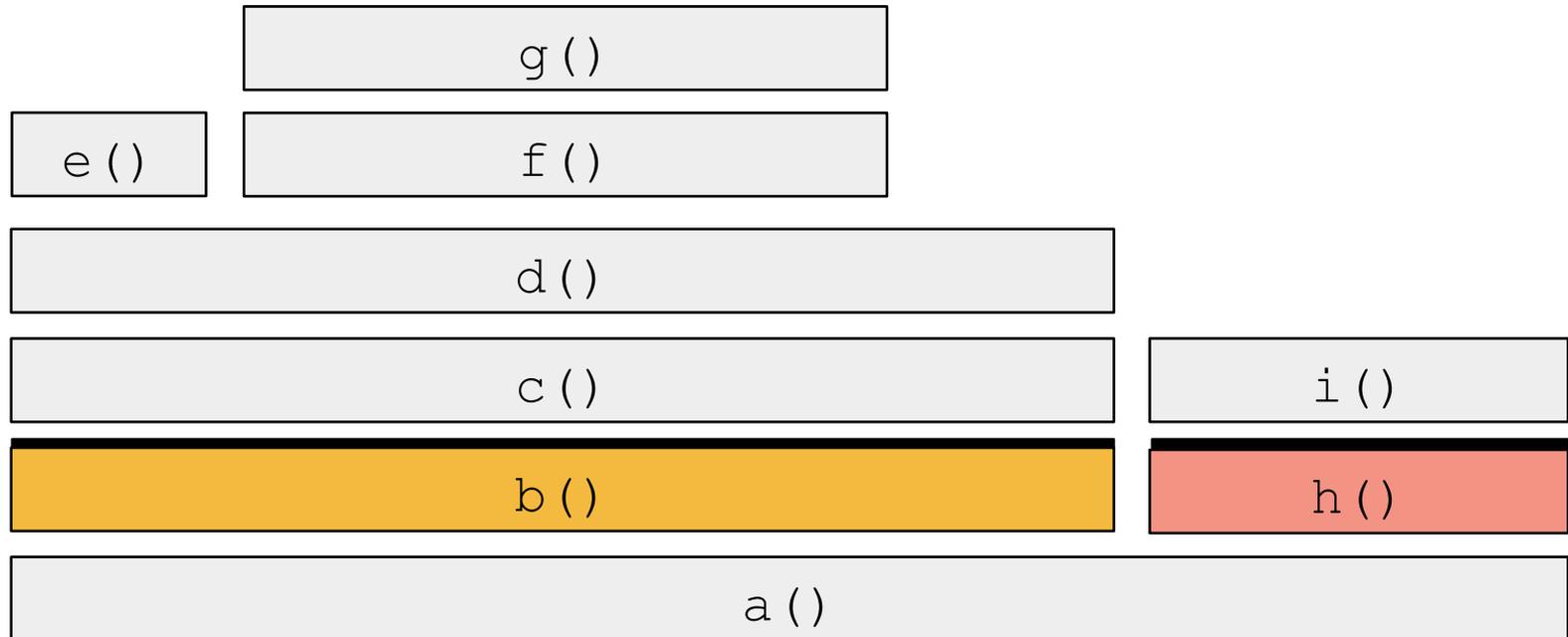
e.g., from g():



Flame Graph Interpretation (3/4)

Widths are proportional to presence in samples

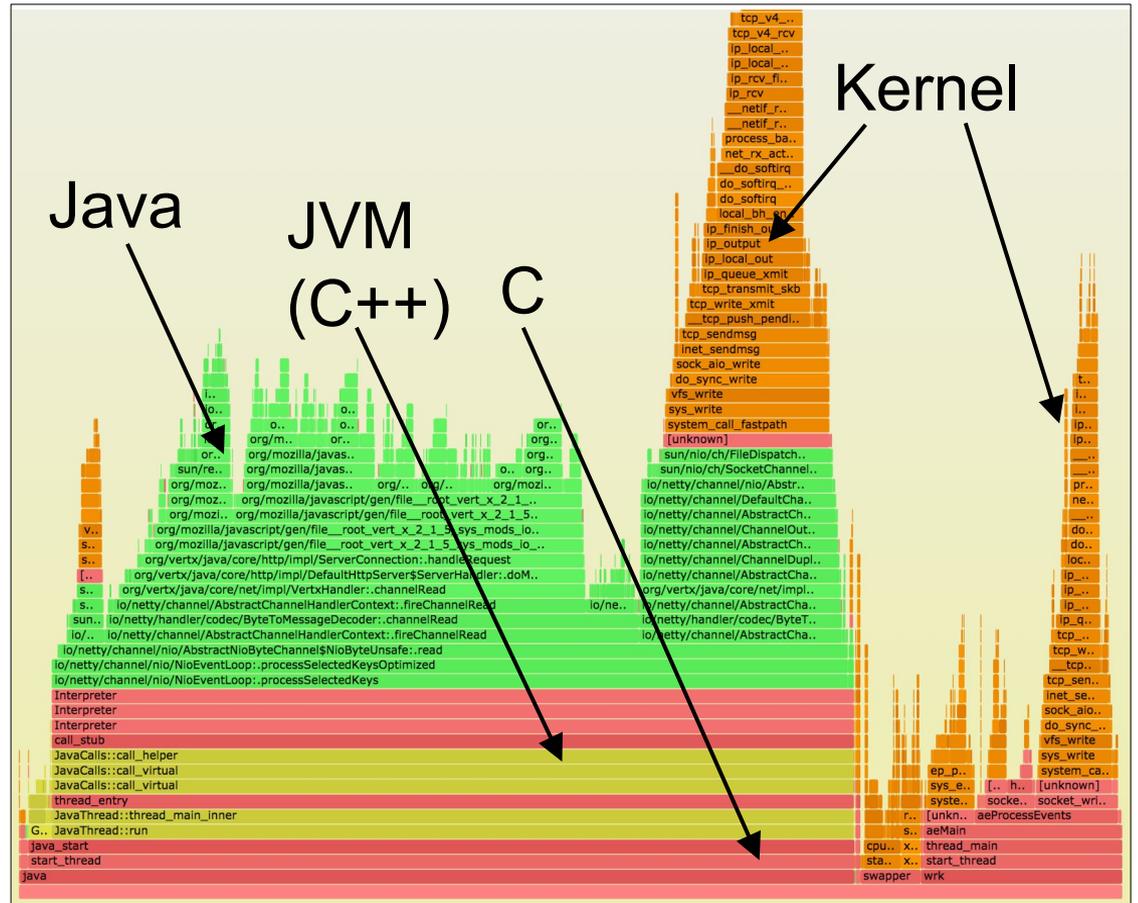
e.g., comparing b() to h() (incl. children)



Flame Graph Interpretation (4/4)

Colors randomized to differentiate frames
Or used for code type;
e.g.:

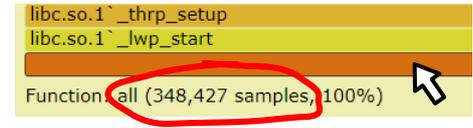
- green == JIT (e.g., Java)
- aqua == inlined
- red == user-level
- orange == kernel
- yellow == C++
- magenta == search term



CPU Flame Graph Tips & Tricks

A) Check sample count (bottom frame): idle system?

- E.g., 49 Hertz x 30 sec x 16 CPUs == 23,520 samples at 100% CPU utilization. <500 samples total would mean <2% busy and probably not interesting!



B) Off-CPU time (I/O, locks) not present

- But their initialization/spin breadcrumbs may be present
- Can use off-CPU flame graphs for this (covered later)

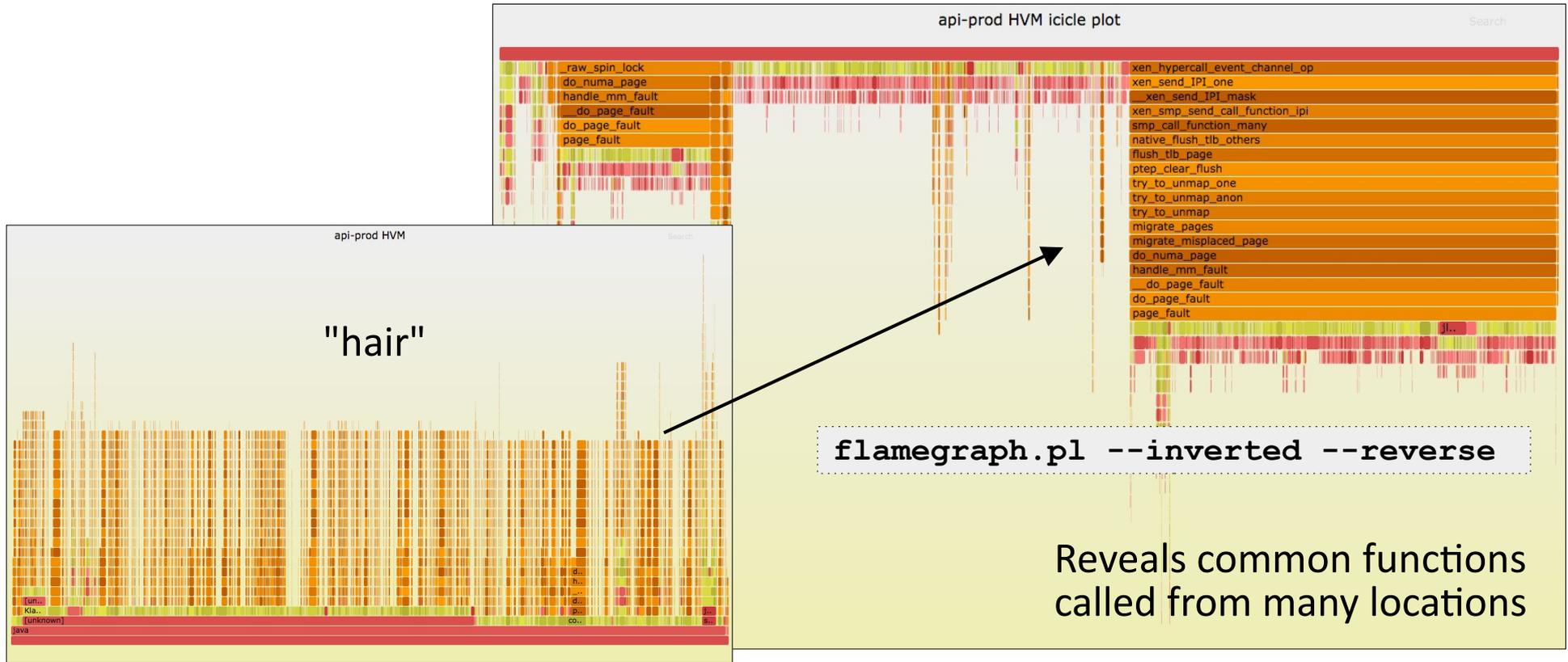
C) Some tiny CPU code paths may be missing

- E.g., some kernel code paths that disable interrupts, and hypervisor time from the guest.
- This is a detail of the profiler and target. Flame graphs just show what the profiler sees.
- Flame graphs may drop tiny frames that are <1 pixel wide unless zoomed in, just to speed up rendering.

D) Too much "hair"? Try a leaf merge...

- Merge stack frames from leaf to root. The default is root to leaf.

Icicle Graph with Leaf Merge



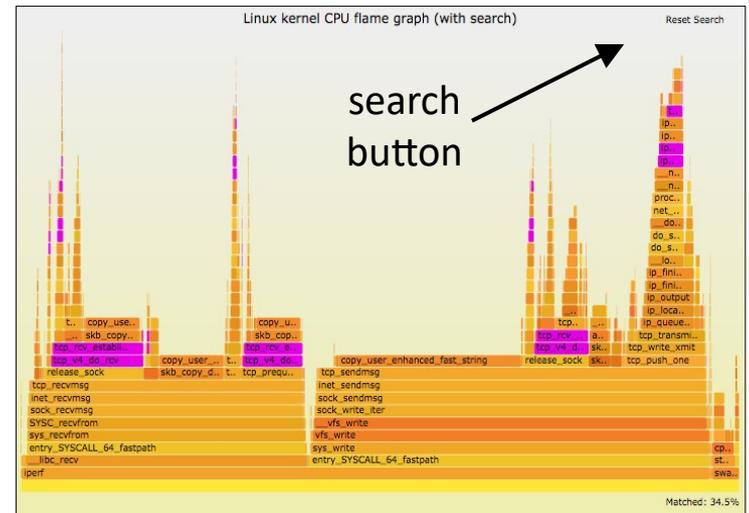
Flame Graph Interactivity

Essentials:

- Mouse-over for frame info (tool tips, status bar)
- Click to zoom
- Search (Ctrl-F or button)

Nice to have:

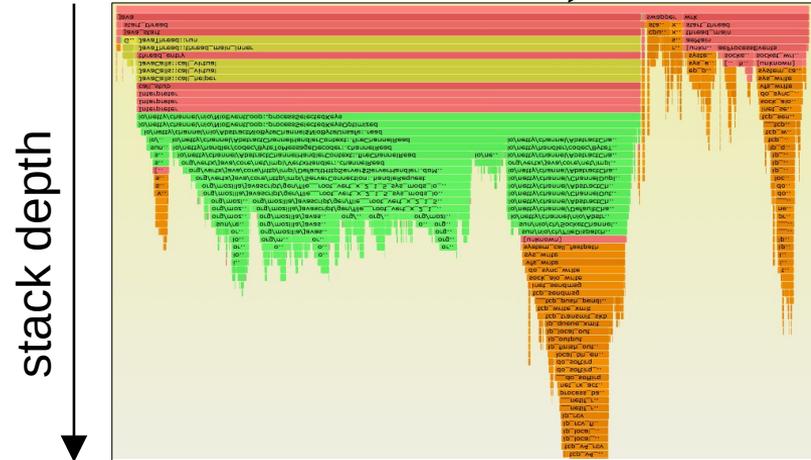
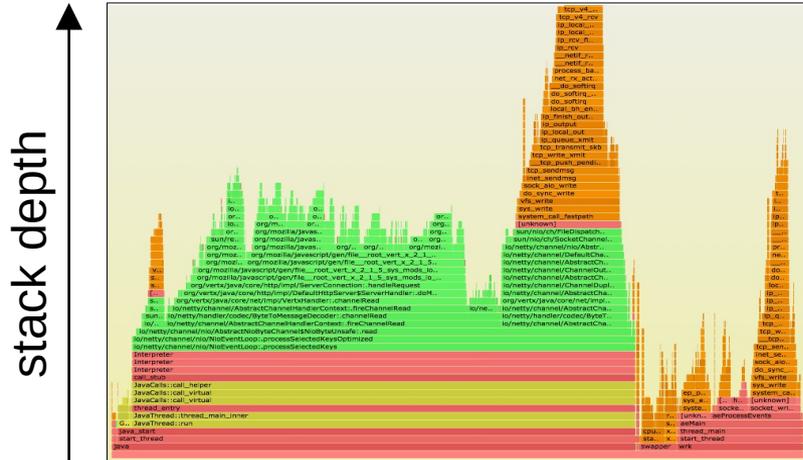
- Merge control: root, leaf, middle
- Y-axis direction: flame or icicle
- Flame chart toggle
- Canned searches
- Collapse filters
- Code links



search matches in magenta

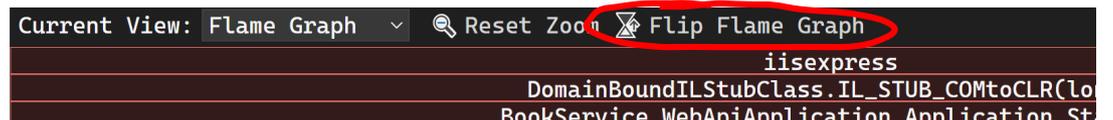
Which way up?

My original flamegraph.pl has --inverted for an “icicle graph”



Either way is fine!

- Icicle layout helps avoid scrolling when starting at the top
- Let the end-user choose



Source: <https://learn.microsoft.com/en-us/visualstudio/profiling/flame-graph>

Differential Flame Graphs

Hues:

- red == more samples
- blue == less samples

Intensity:

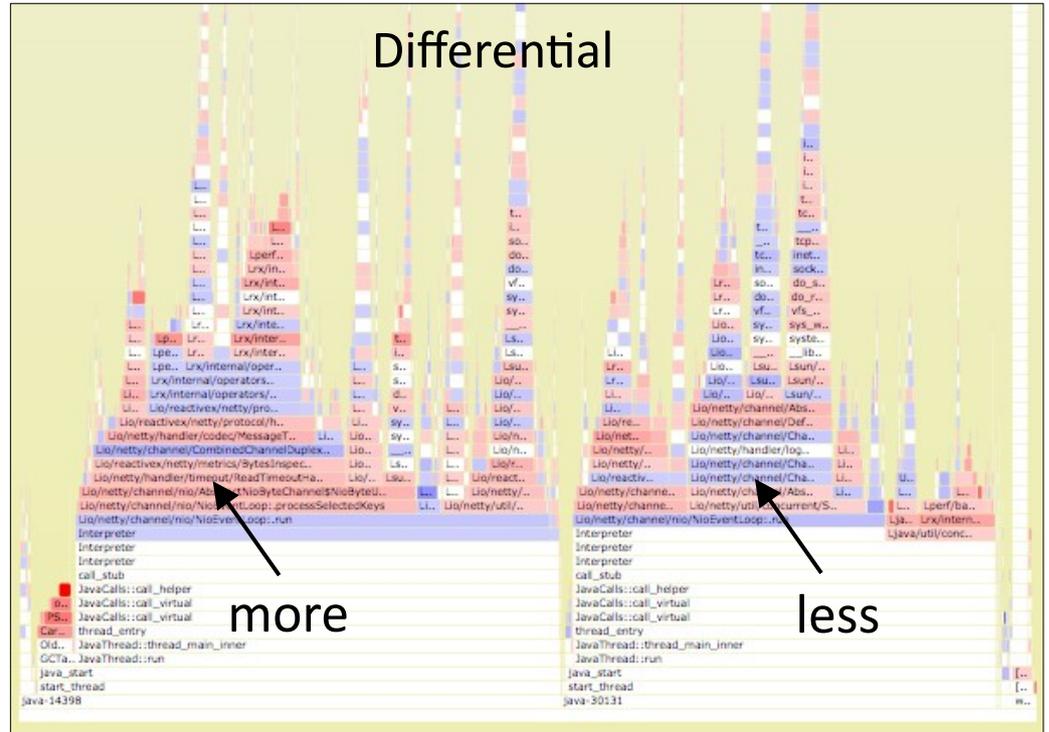
- Degree of difference

Other examples

- flamegraphdiff

This spectrum can show other metrics, like CPI.

Remember to show elided frames!



Poor Man's Differential Flame Graphs

Toggle between tabs in your browser

- Like searching for Pluto!

Or flip between slides

- Cue exciting demo!



Poor Man's Differential Flame Graphs

Toggle between tabs in your browser

- Like searching for Pluto!

Or flip between slides

- Cue exciting demo!



System CPU Profilers

- Linux
 - perf_events (aka "perf")
 - bcc profile (eBPF-based)
- Windows
 - XPerf, WPA (now has flame graphs!)
- OS X
 - Instruments
- And many others...

Tip: use system profilers whenever possible. Runtime profilers (e.g., Java JVMTI-based) are user space and typically don't include kernel CPU time or kernel stacks.

Linux CPU Flame Graphs

Linux 5.8+ via perf for *simplicity* (2020):

```
perf script flamegraph -F 49 -a -- sleep 30
```

- Generates flamegraph.html. One command! Thanks Andreas Gerstmayr.

Linux 4.9+ via eBPF for *efficiency* (2016):

```
apt-get install bpfcc-tools  
git clone https://github.com/brendangregg/FlameGraph  
profile-bcc.py -dF 49 30 | ./FlameGraph/flamegraph.pl > perf.svg
```

- eBPF (no longer an acronym) is the name of an in-kernel execution environment, used in this case for aggregating stack samples in kernel context
- Most efficient: no perf.data file, summarizes in-kernel

Some runtimes (e.g., JVM) require extra steps for stacks & symbols (next section)

Older Linux CPU Flame Graphs

Linux 2.6+ via perf.data and perf script (2009):

```
git clone https://github.com/brendangregg/FlameGraph; cd FlameGraph
perf record -F 49 -a -g -- sleep 30
perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg
```

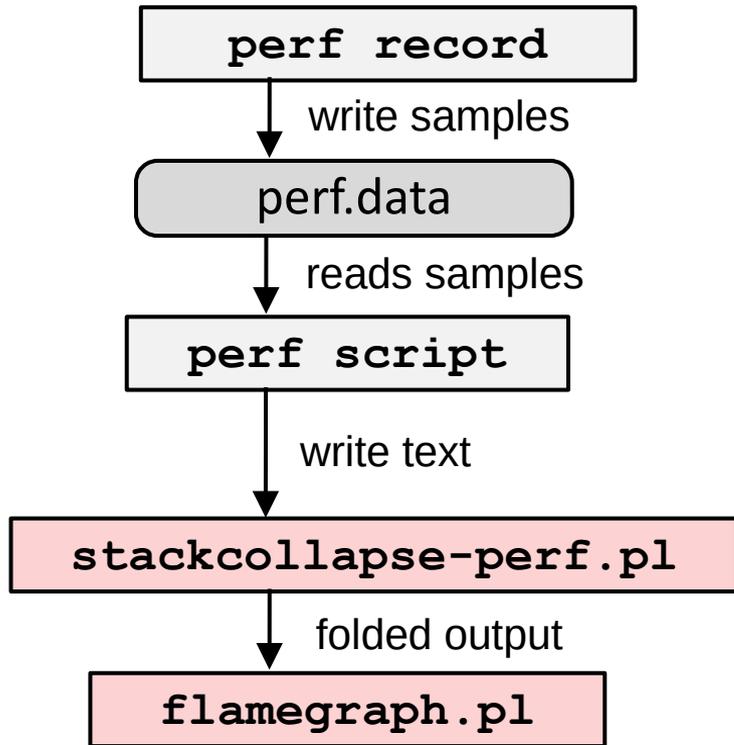
Linux 4.5 can use folded output (2016):

- Skips the CPU-costly stackcollapse-perf.pl step; see:
<http://www.brendangregg.com/blog/2016-04-30/linux-perf-folded.html>

Linux Profiling Optimizations

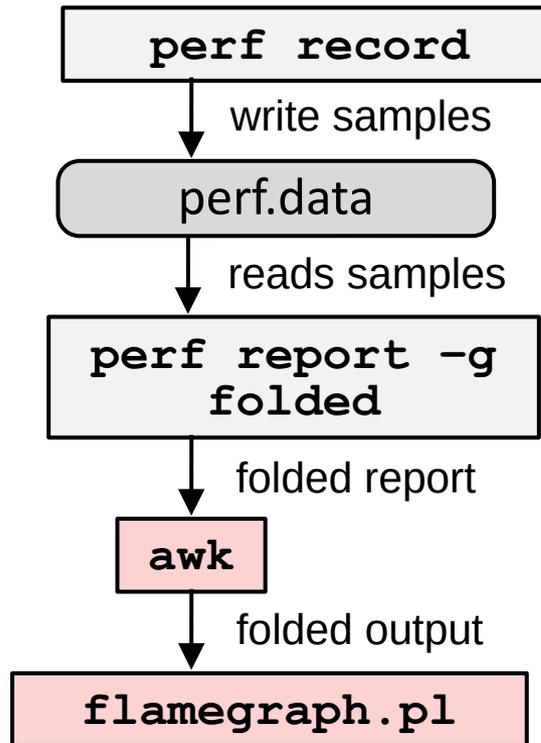
Linux 2.6

capture stacks



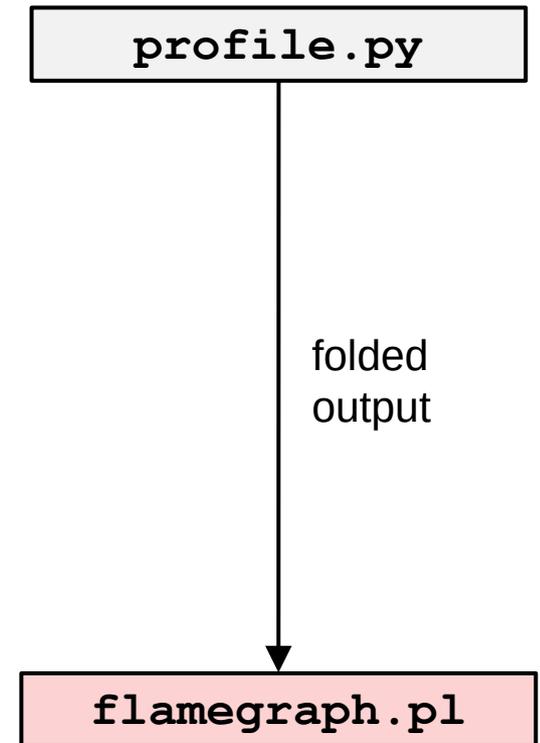
Linux 4.5

capture stacks



Linux 4.9

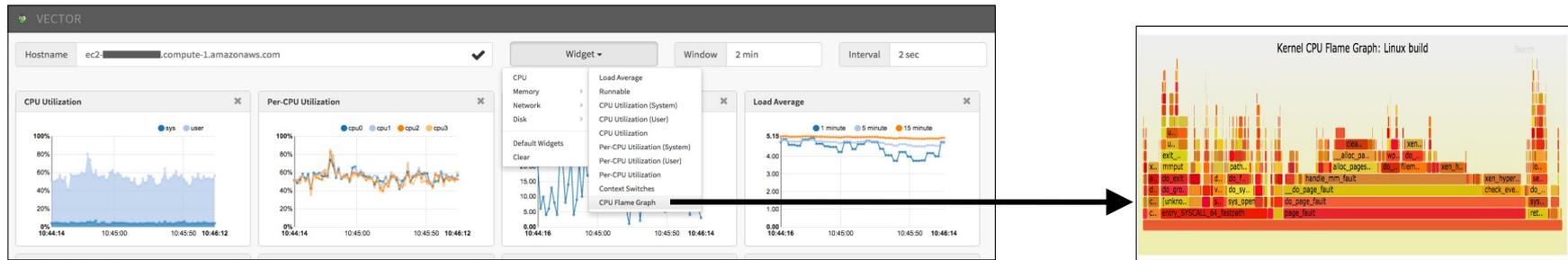
count stacks (BPF)



GUI Automation

There are many options nowadays. I've worked on five:

- **Netflix Vector** (now retired!):



- **Netflix FlameScope** (covered later)
- **Netflix FlameCommander** (continuous profiling; not open source yet)
- I'm now helping with **Intel vTune** and **Intel gProfiler**

Open source examples include **Granulate gProfiler**, **Eclipse TraceCompass**, **Grafana flame graphs**, **Firefox profiler**, and more (see implementation slides). Build your own!

Flame Charts (2013)

Inspired by flame graphs: https://bugs.webkit.org/show_bug.cgi?id=111162

Ilya Tikhonovsky 2013-03-01 04:33:26 PST

[Description](#)

Flame Chart may give to the developer a better clue what is going on with the performance without expanding the entire tree.

<http://dtrace.org/blogs/brendan/2011/12/16/flame-graphs/>

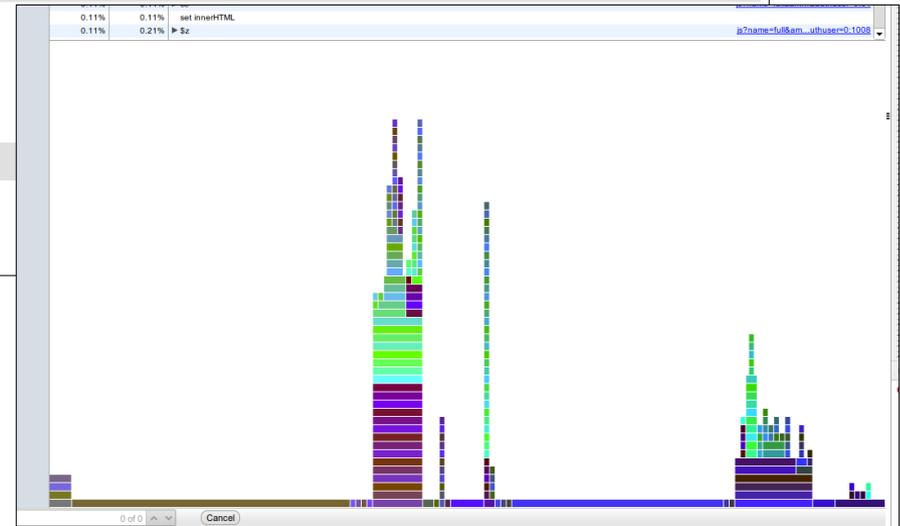
Ilya Tikhonovsky 2013-03-01 04:35:00 PST

[Comment 1](#)

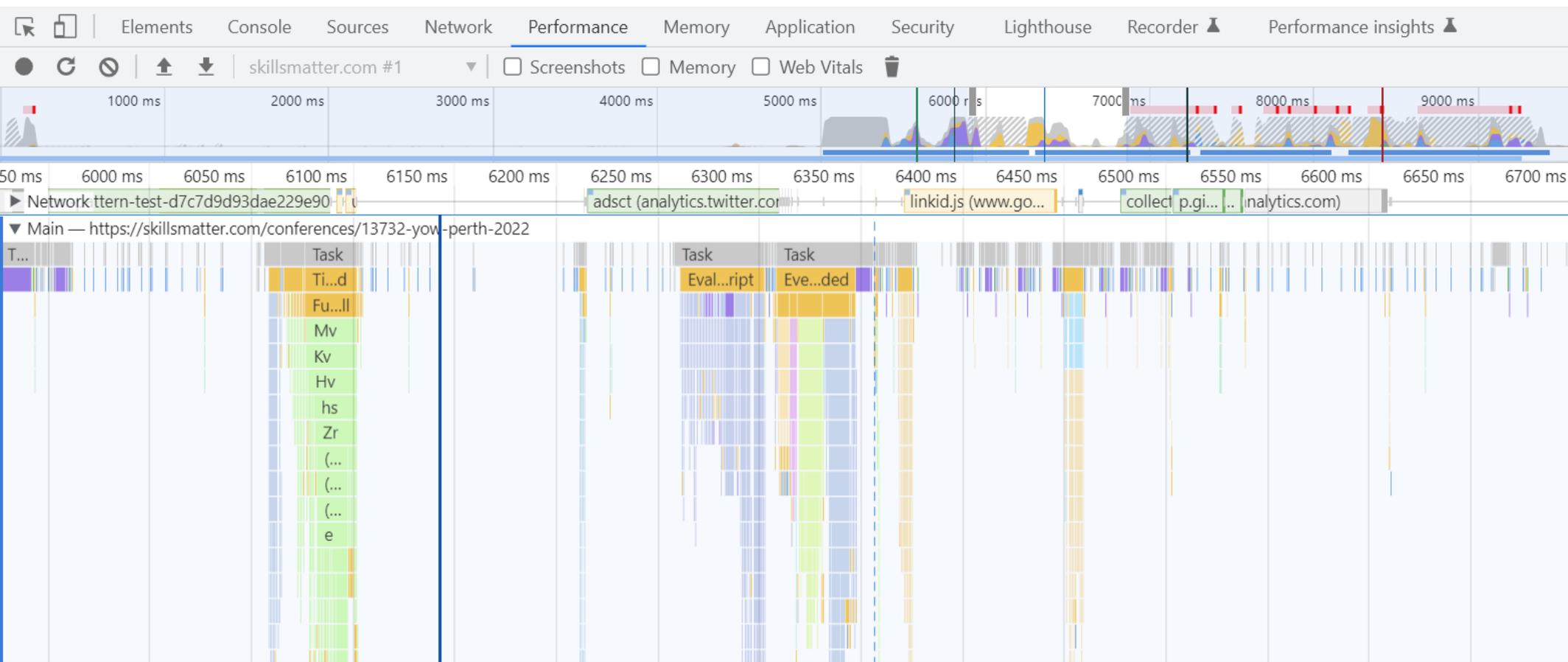
Created [attachment 190933](#) [\[details\]](#)
screenshot

Ilya Tikhonovsky 2013-03-01 04:37:34 PST

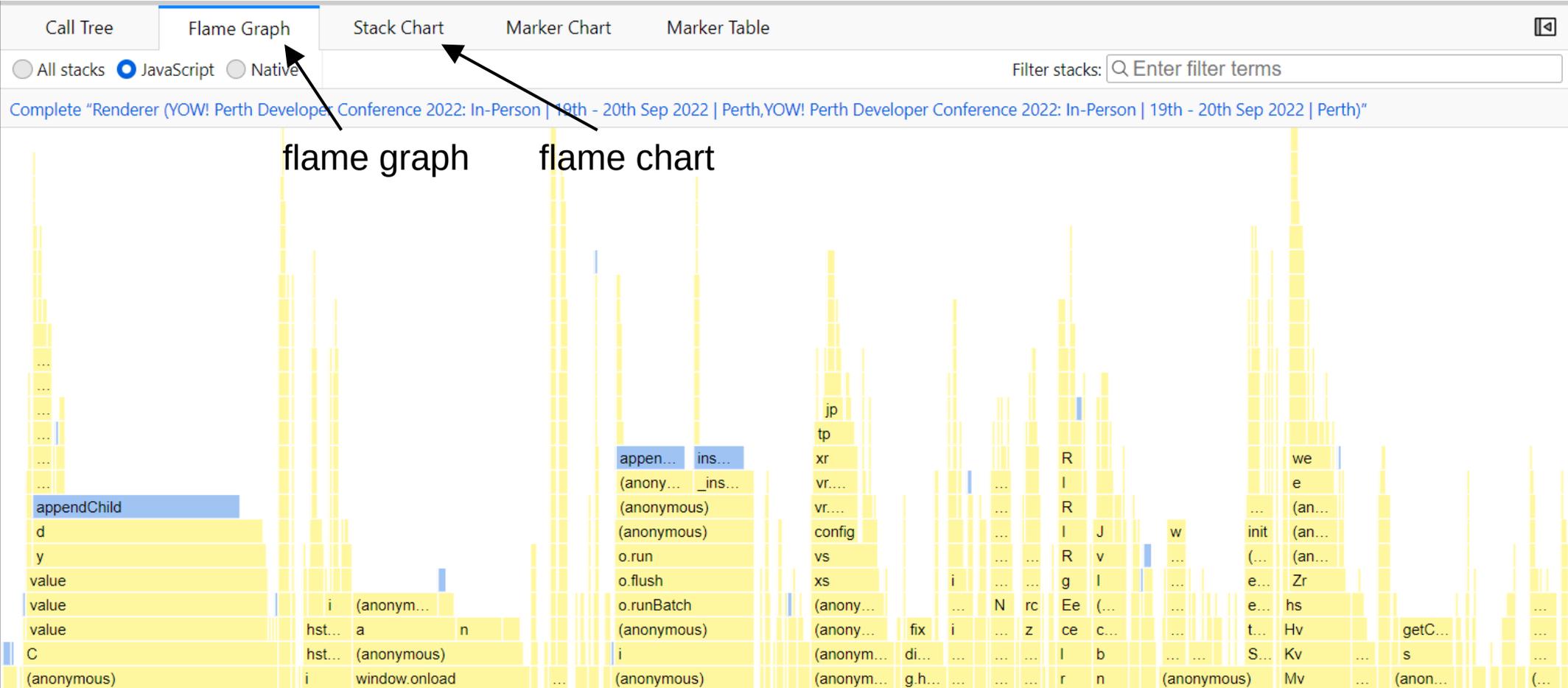
Created [attachment 190934](#) [\[details\]](#)
Patch



Chrome DevTools Flame Charts (2022)



Firefox Profiler Flame Graph (2022)



Flame Charts

x-axis: time

Flame Graphs

x-axis: population

alphabet sort or another frame merging algorithm

3. STACKS AND SYMBOLS

And Other Issues

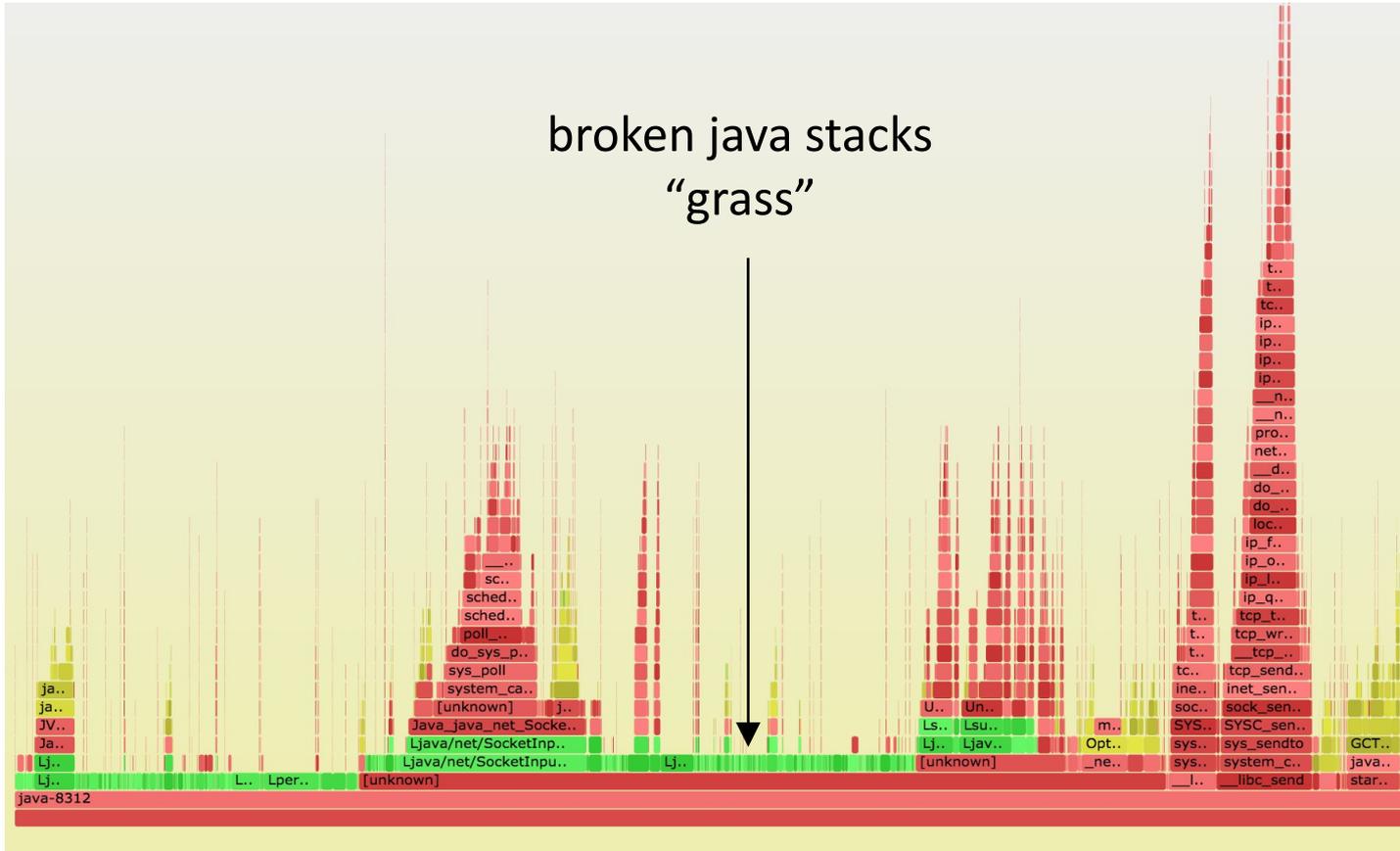
Broken Stack Traces are Common

```
# perf record -F 99 -a -g - sleep 30
# perf script
[...]
java 4579 cpu-clock:
    7f417908c10b [unknown] (/tmp/perf-4458.map)

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

should probably have more frames

... as a Flame Graph



Fixing Stack Walking

A. Frame pointer-based

- Fix by disabling that compiler optimization: gcc's `-fno-omit-frame-pointer`
- Pros: simple, supported by many tools
- Cons: might cost a little extra CPU (usually <1%)

B. Debug info (DWARF) walking

- Cons: costs disk space, and not supported by all profilers, expensive for real-time tracing

C. JIT-provided runtime walkers

- Pros: include more internals, such as inlined frames (e.g., JVM TI stacks)
- Cons: limited to application internals: no kernel

D. Last branch record (LBR)

E. Add-on walkers (eBPF)

Fixing Java Stack Traces

```
# perf script
[...]  
java 4579 cpu-clock:  
    7f417908c10b [unknown] (/tmp/...  
  
java 4579 cpu-clock:  
    7f41792fc65f [unknown] (/tmp/...  
    a2d53351ff7da603 [unknown] ([unkn...  
[...]
```



I prototyped JVM frame pointers. Oracle rewrote it and included it in Java as **-XX:+PreserveFramePointer** (JDK 8 u60b19)

```
# 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...
```


Fixing Native Symbols

- A. Add a -dbgsym package, if available
- B. Recompile from source

Fixing JIT Symbols (Java, Node.js, ...)

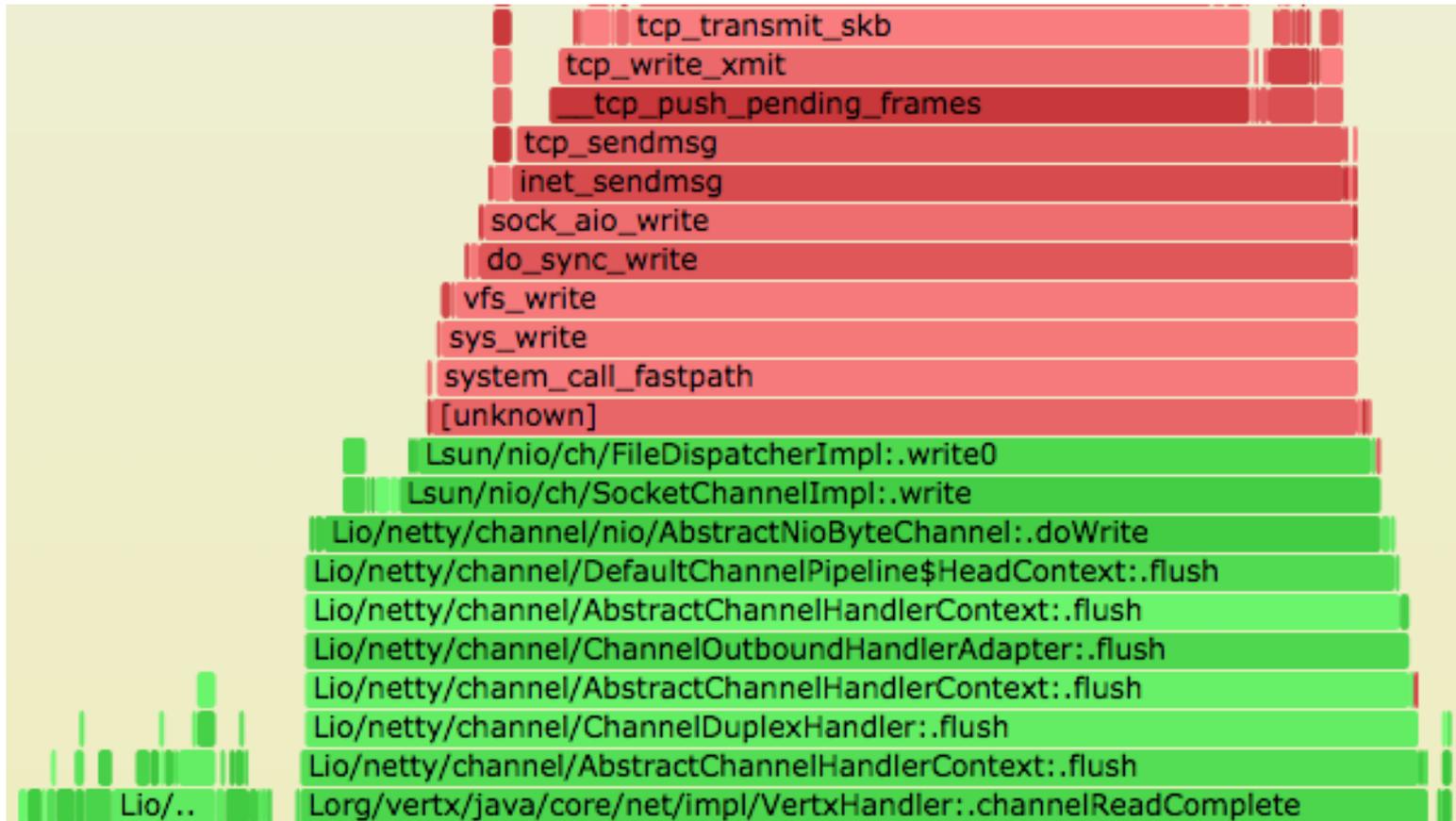
Just-in-time runtimes don't have a pre-compiled symbol table
So Linux perf looks for an externally provided symbol file

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

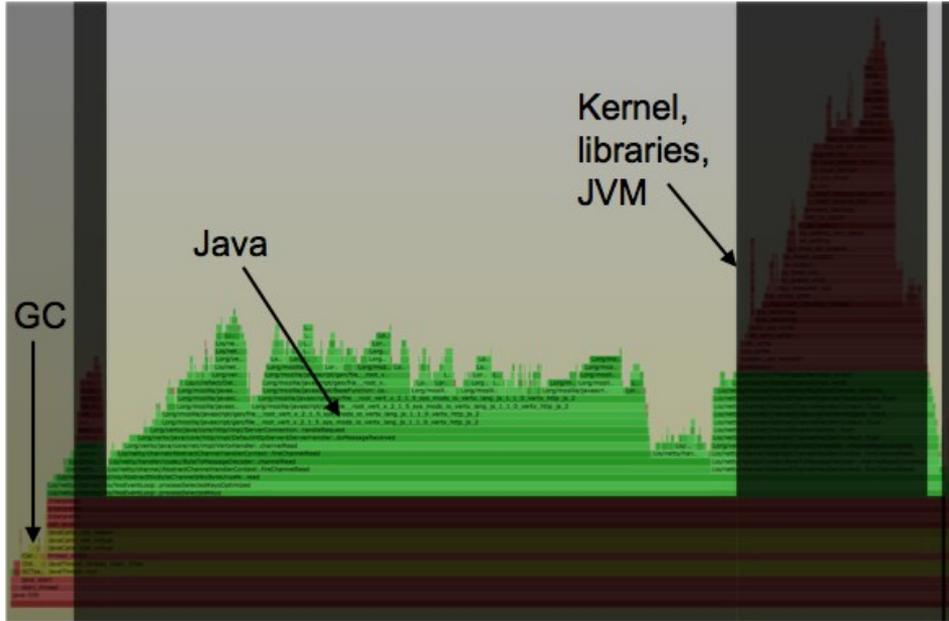


- This can be created by runtimes; e.g., Java's perf-map-agent
- Not the only solution; can also integrate with JIT-based walkers, or have an external symbol translator (perf script, or eBPF-based).

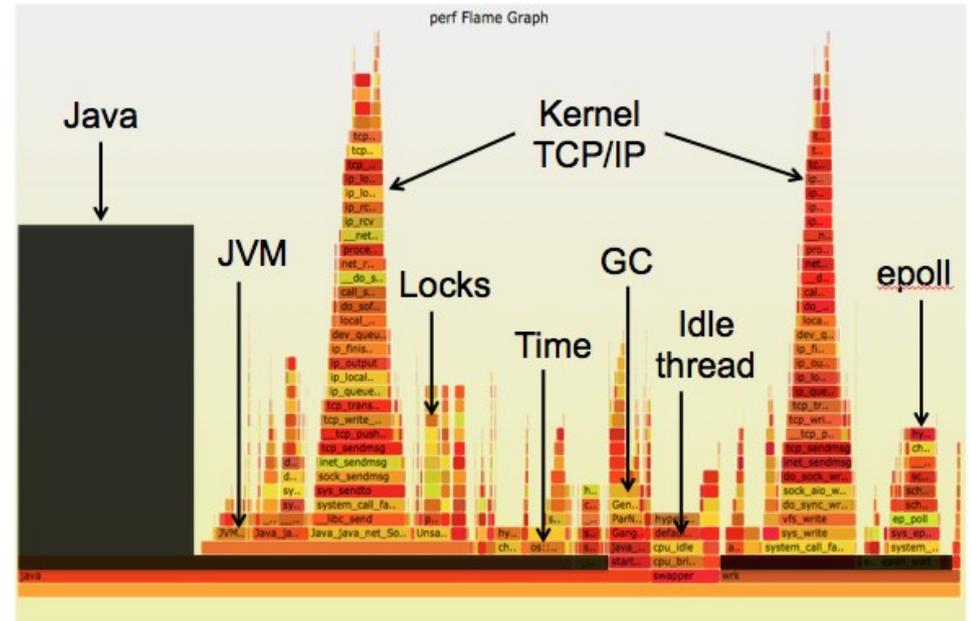
Fixed Symbols (zoom)



2014: Java Profiling (broken stacks)

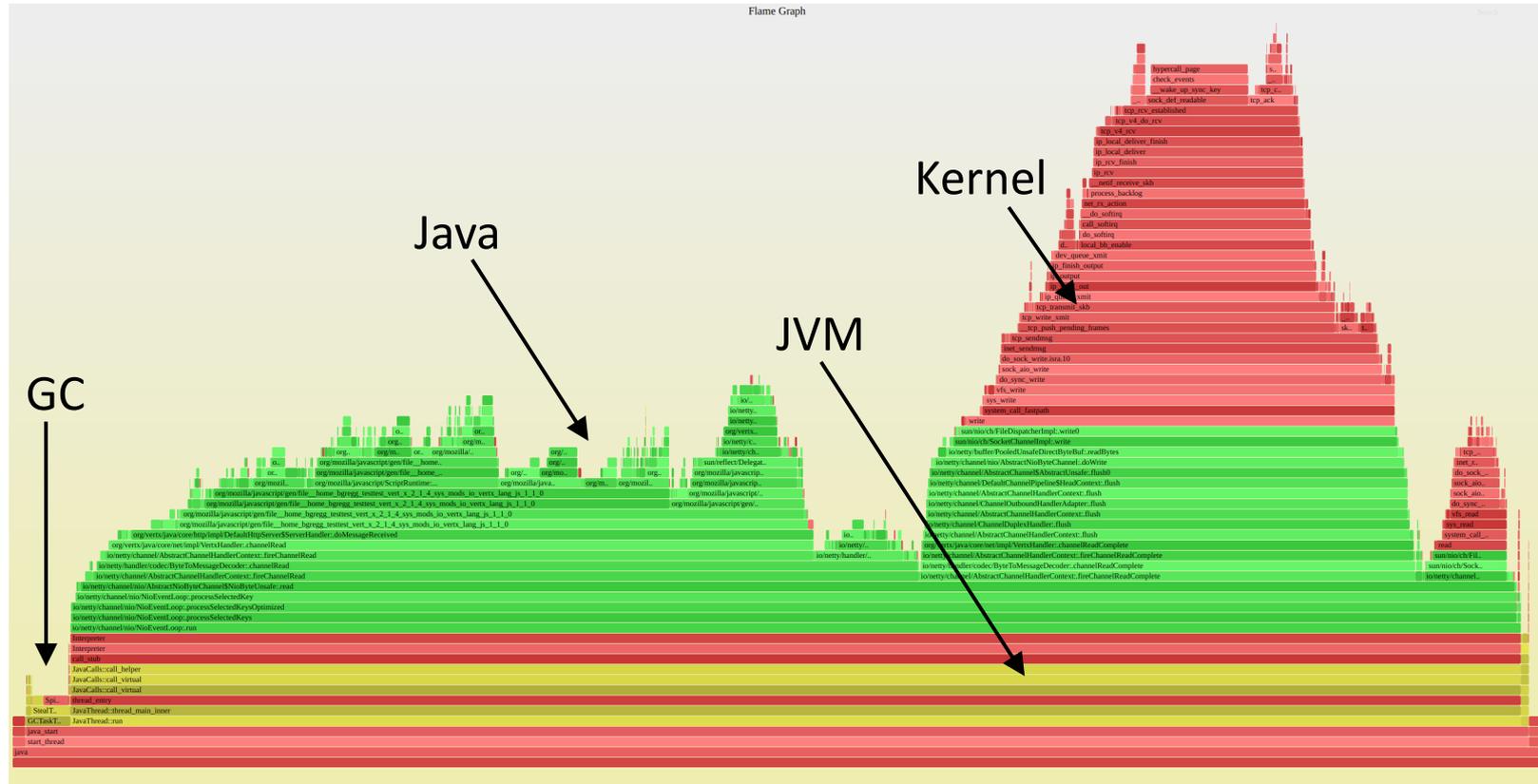


Java Profilers



System Profilers

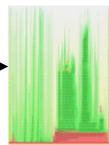
2018: Java Profiling (fixed stacks)



CPU Mixed-mode Flame Graph

Other Issues

- JIT Symbol churn
 - Take before and after snapshots, or use perf's timestamped symbol logs.
- Containers
 - Are symbol files read from the right namespace? Should now work.
- Stack Depth limits
 - Linux perf had a 127 frame limit, now tunable. Thanks Arnaldo Carvalho de Melo!



perf_event_max_stack=1024

A Java microservice
with a stack depth
of > 900



Language/Runtime Issues

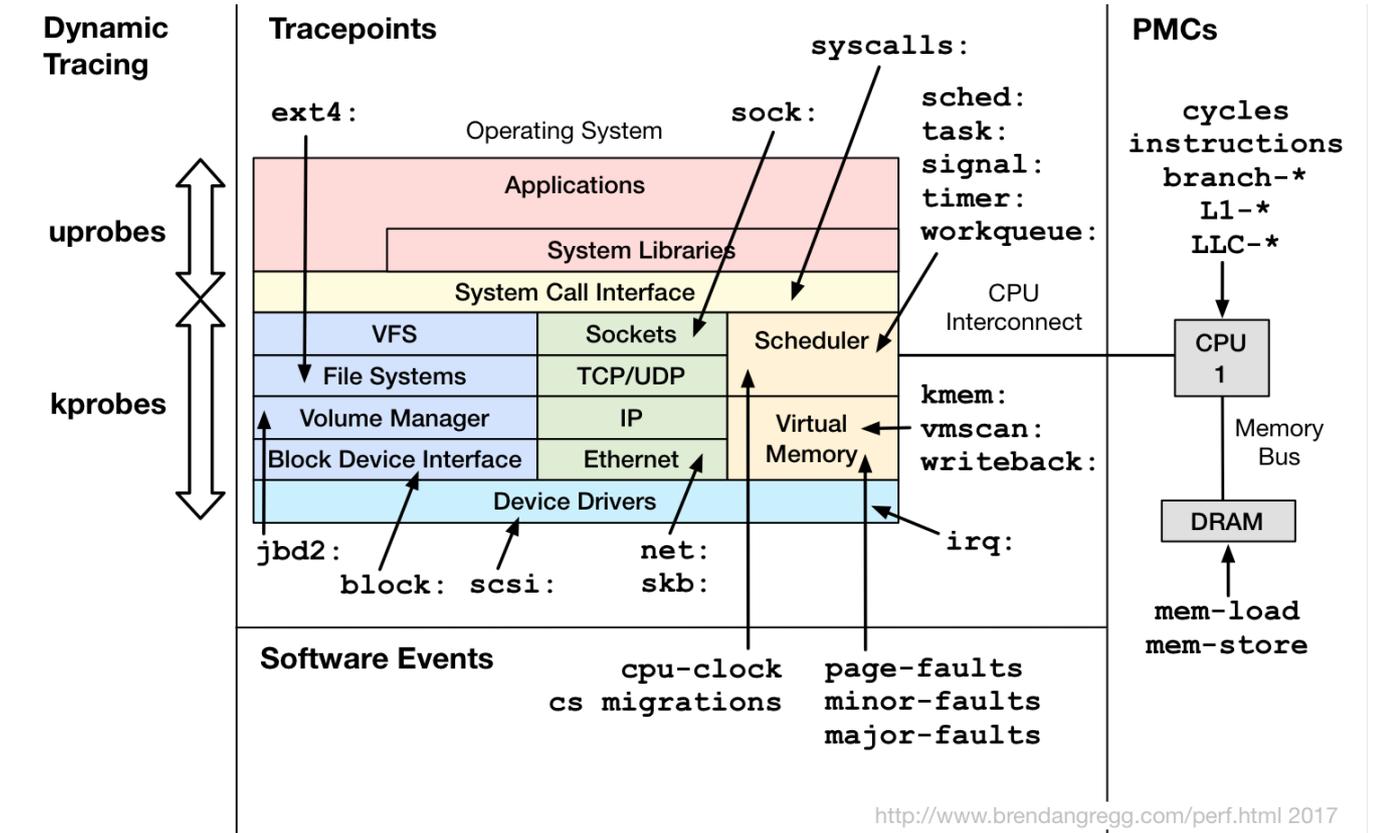
Each may have special stack/symbol instructions

- **Java, Node.js, Python, Ruby, C++, Go, ...**
- See: <https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>
- Check if flame graphs are already in the “official” profiler
- Try an Internet search

4. ADVANCED FLAME GRAPHS

Flame graphs can visualize any stack trace collection

On Linux, stacks from any of these events:



Page Faults

Show what triggered main memory (resident) to grow:

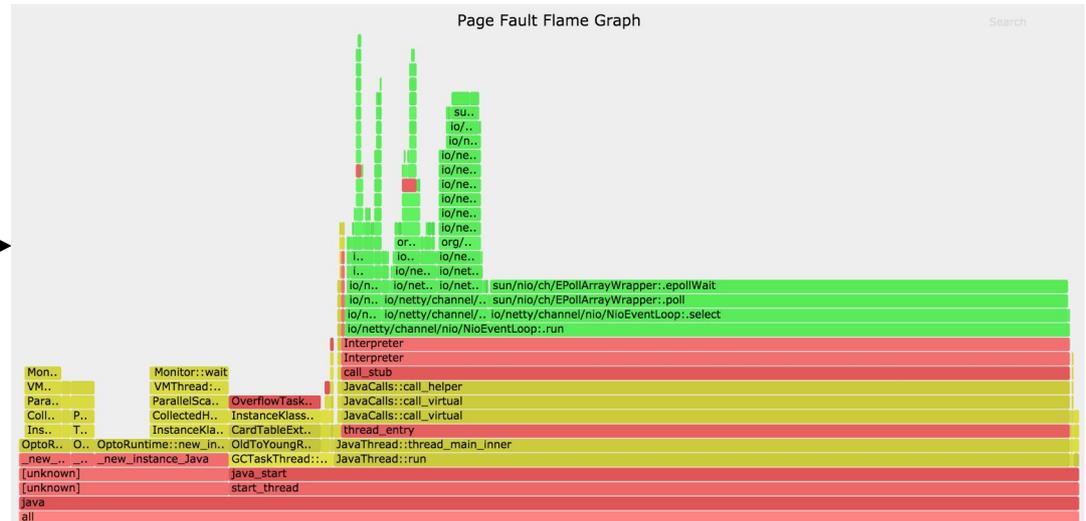
```
# perf record -e page-faults -p PID -g -- sleep 120
```

- "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

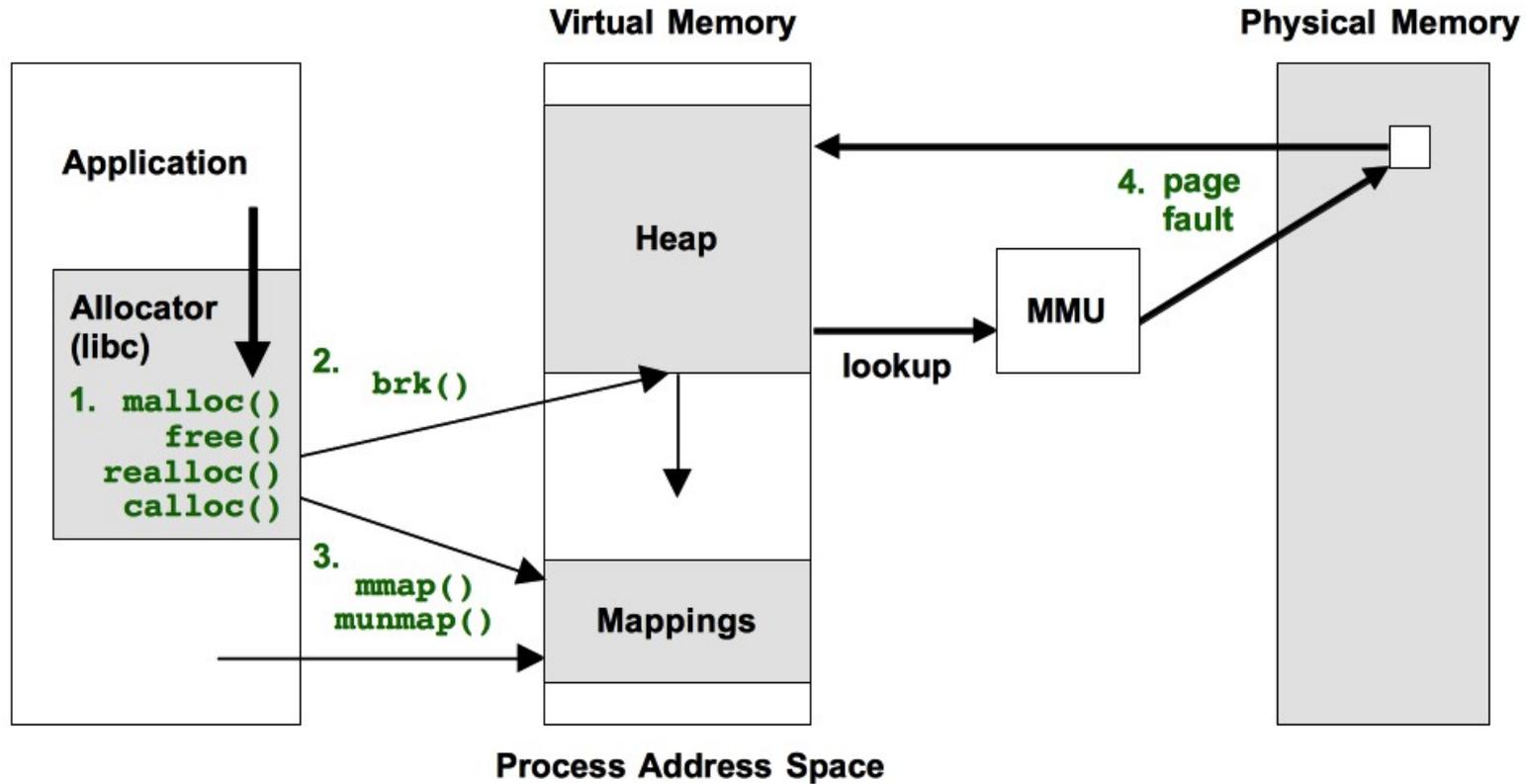
RES column in top(1)

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

grows
because



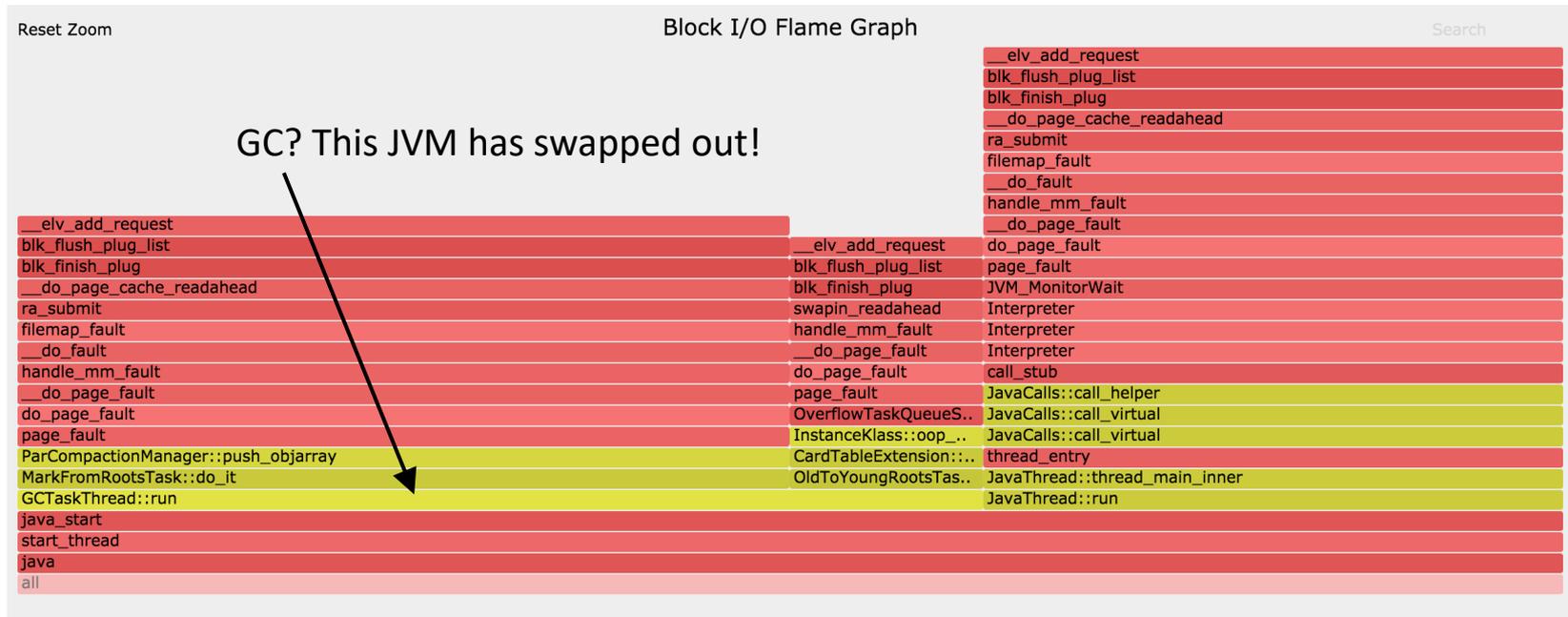
Other Memory Sources



Disk I/O Requests

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

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



TCP Events

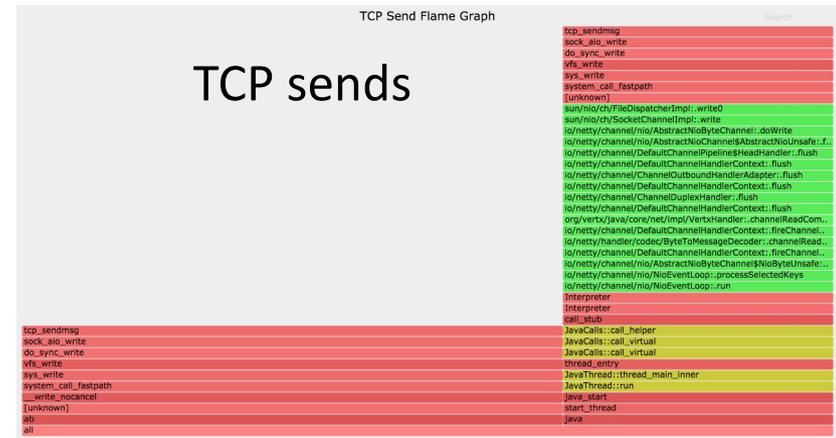
TCP transmit, using eBPF:

```
# bpftrace -e 'kprobe:tcp_sendmsg { @[kstack, ustack] = count(); }'
```

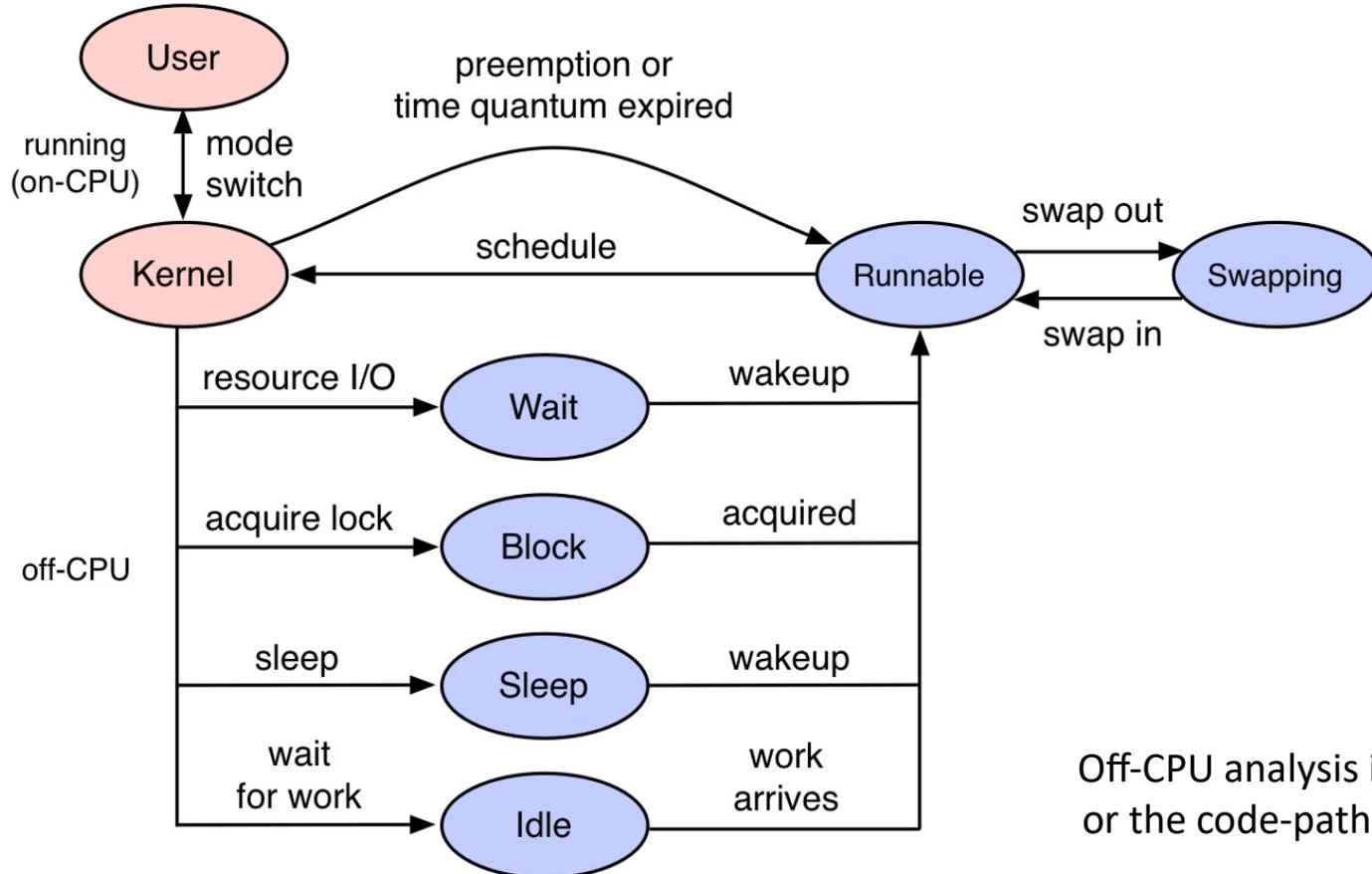
- For eBPF, can cost **noticeable overhead** for high packet rates (test and measure)
- For perf, can have *prohibitive overhead* due to the trace, dump, post-process cycle
- Note that **TCP receive is async**, so stack traces are meaningless. Trace socket read instead.

Can also trace TCP connect, accept

- Lower frequency, therefore lower overhead

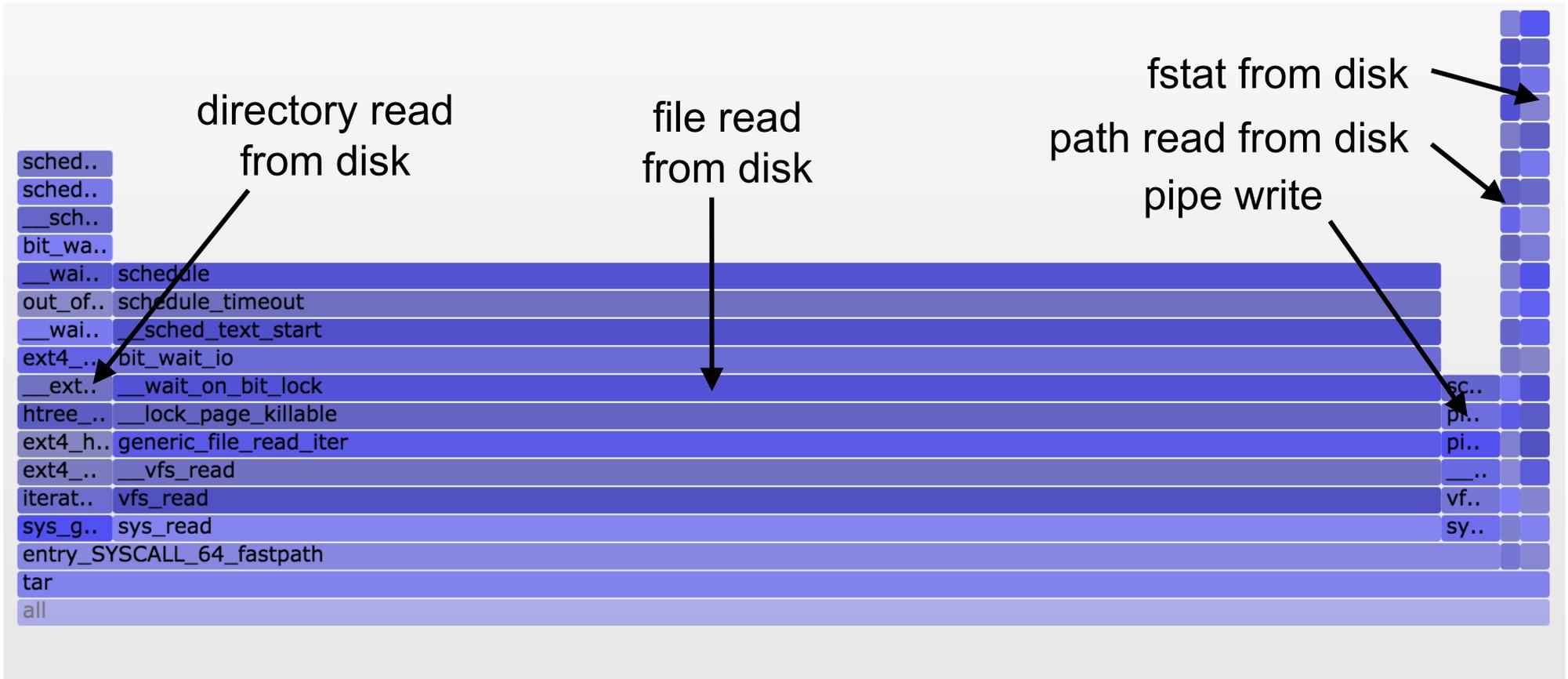


Off-CPU Analysis



Off-CPU analysis is the study of blocking states, or the code-path (stack trace) that led to them

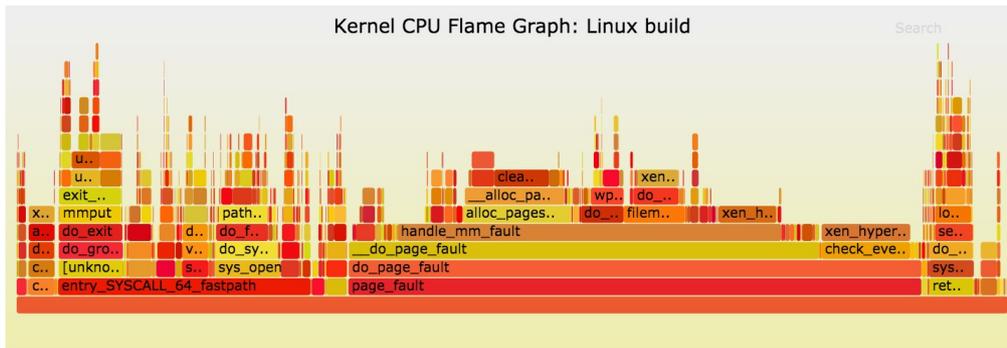
Off-CPU Time (zoomed): tar(1)



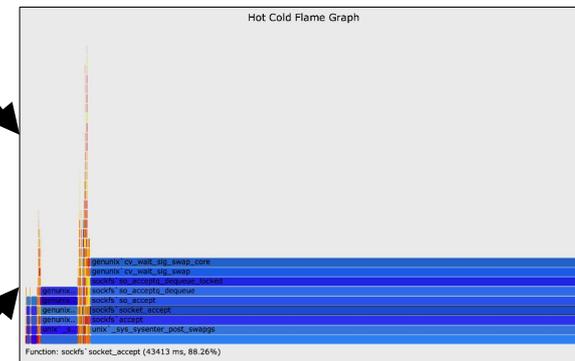
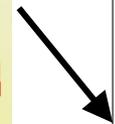
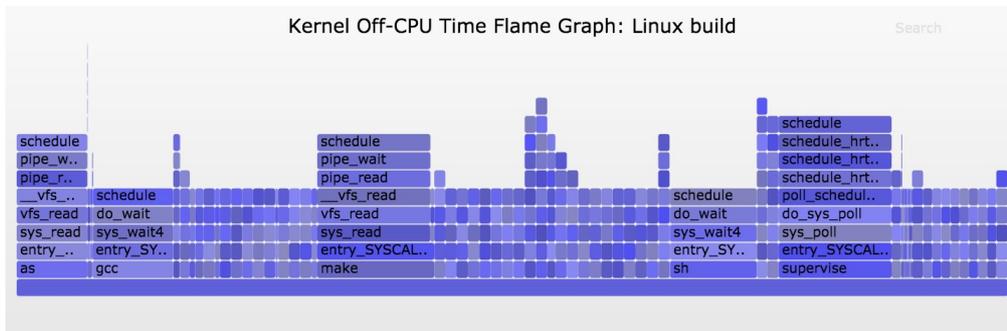
Only showing kernel stacks in this example

CPU + Off-CPU Flame Graphs: See Everything

CPU

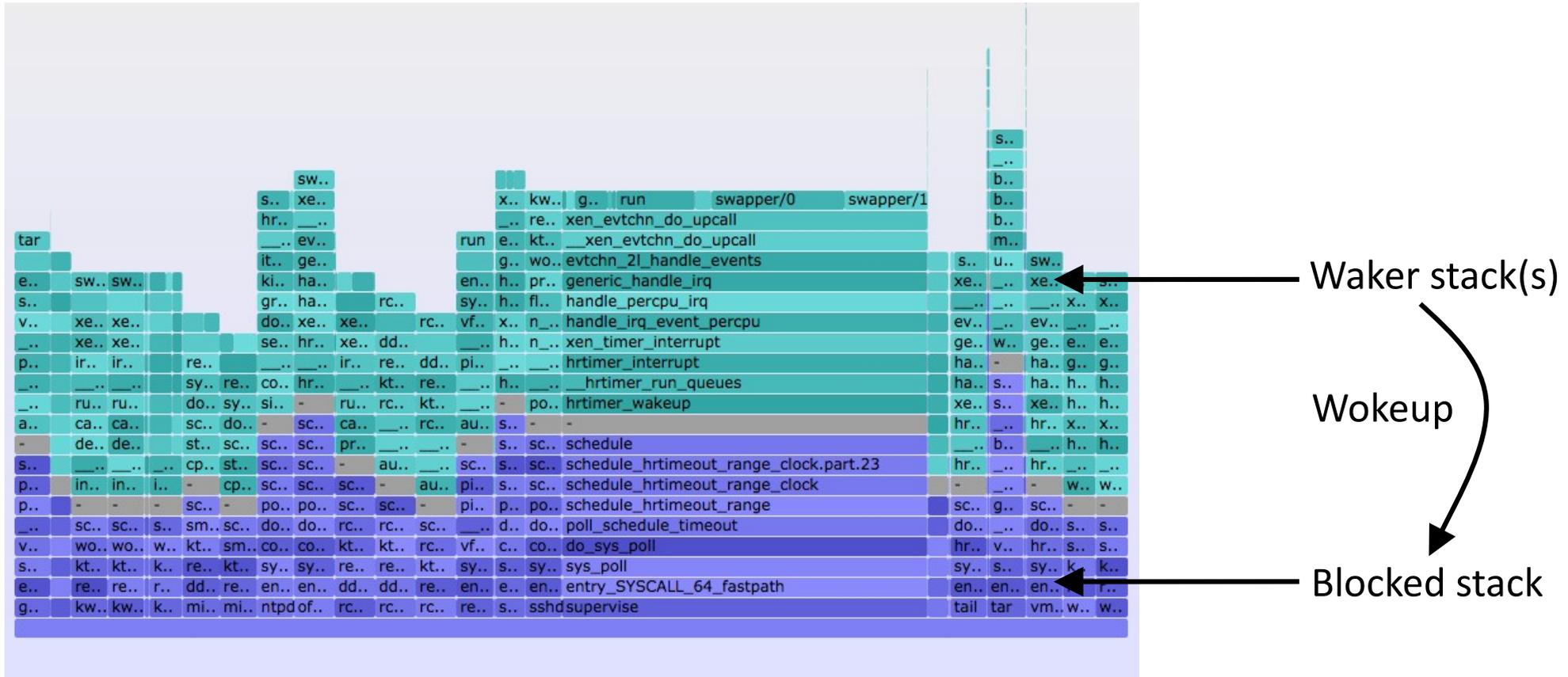


Off-CPU



Everything
(All thread time)

Off-Wake Time Flame Graph

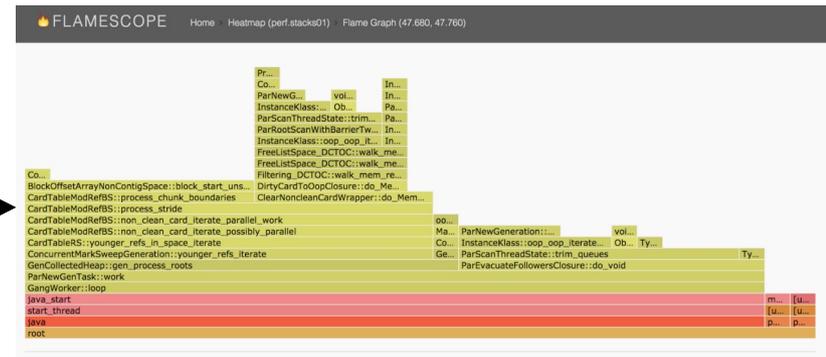
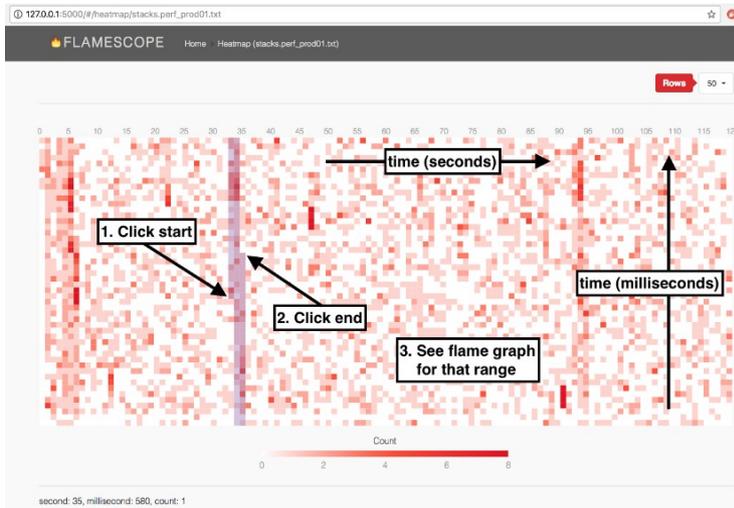


Uses Linux enhanced BPF to merge off-CPU and waker stack in kernel context

FlameScope

Flame graphs can hide time-based issues of **variation** and **perturbations**.

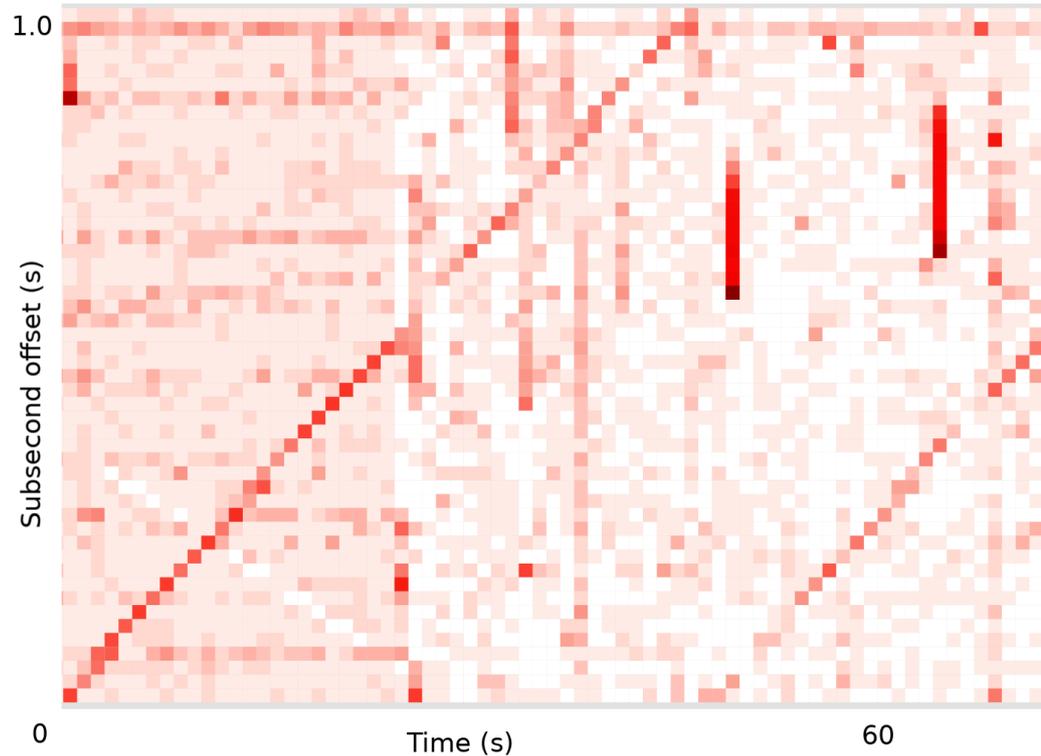
FlameScope uses **subsecond-offset heat maps** to show these issues. They can then be selected for the corresponding flame graph.



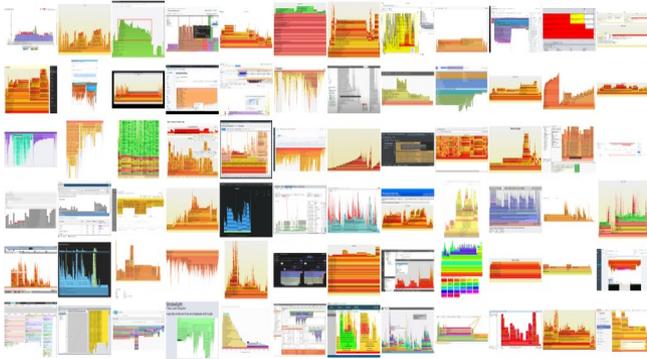
<https://brendangregg.com/blog/2018-12-15/flamescope-origin.html>
<https://www.brendangregg.com/HeatMaps/subsecondoffset.html>

FlameScope Example

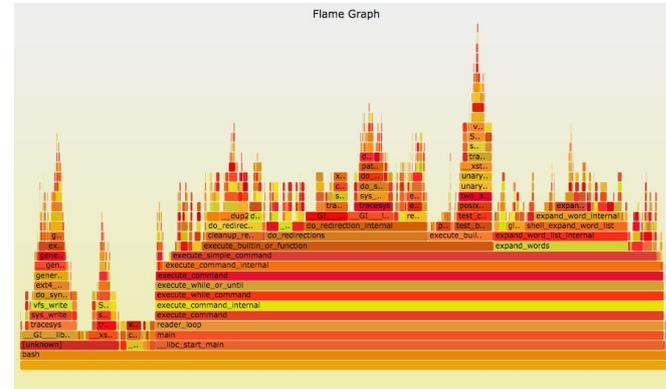
How many patterns can you see?



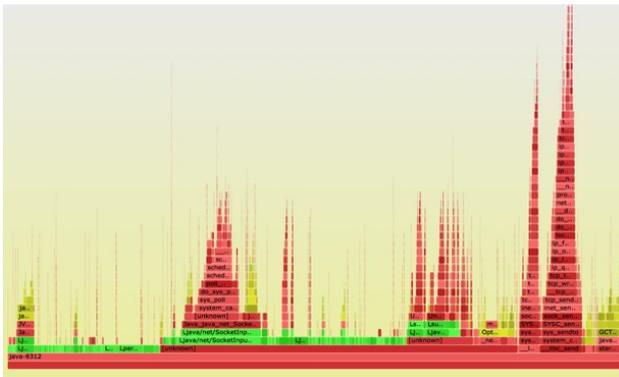
Agenda Recap



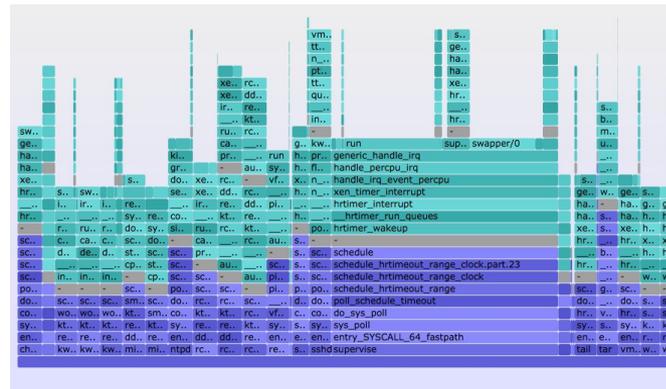
1. Implementations



2. CPU Flame graphs



3. Stacks & Symbols



4. Advanced flame graphs

Take Aways

1. Interpret CPU flame graphs
2. Understand runtime challenges
3. Why eBPF for advanced flame graphs

A new tool to lower your cost, latency, and carbon

Links & References

Flame Graphs

- "The Flame Graph" Communications of the ACM, Vol. 56, No. 6 (June 2016)
- <http://queue.acm.org/detail.cfm?id=2927301>
- <http://www.brendangregg.com/flamegraphs.html>
- <http://www.brendangregg.com/flamegraphs.html#Updates>
- <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>
- <http://www.brendangregg.com/FlameGraphs/memoryflamegraphs.html>
- <http://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html>
- <http://techblog.netflix.com/2015/07/java-in-flames.html>
- <http://techblog.netflix.com/2016/04/saving-13-million-computational-minutes.html>
- <http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html>
- <http://www.brendangregg.com/blog/2016-01-20/ebpf-offcpu-flame-graph.html>
- <http://www.brendangregg.com/blog/2016-02-01/linux-wakeup-offwake-profiling.html>
- <http://www.brendangregg.com/blog/2016-02-05/ebpf-chaingraph-prototype.html>
- <https://brendangregg.com/blog/2018-12-15/flamescope-origin.html>
- <https://github.com/brendangregg/FlameGraph>
- <https://github.com/spiermar/d3-flame-graph>
- <https://github.com/Netflix/flamescope>
- <http://corpaul.github.io/flamegraphdiff/>
- <https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/user-interface-reference/window-flame-graph.html>
- <https://gprofiler.io/>

Linux perf

- https://perf.wiki.kernel.org/index.php/Main_Page
- <http://www.brendangregg.com/perf.html>

Linux eBPF

- <https://ebpf.io/> <https://www.brendangregg.com/ebpf.html>

These slides: https://www.brendangregg.com/Slides/YOW2022_flame_graphs.pdf

YOW! 2022

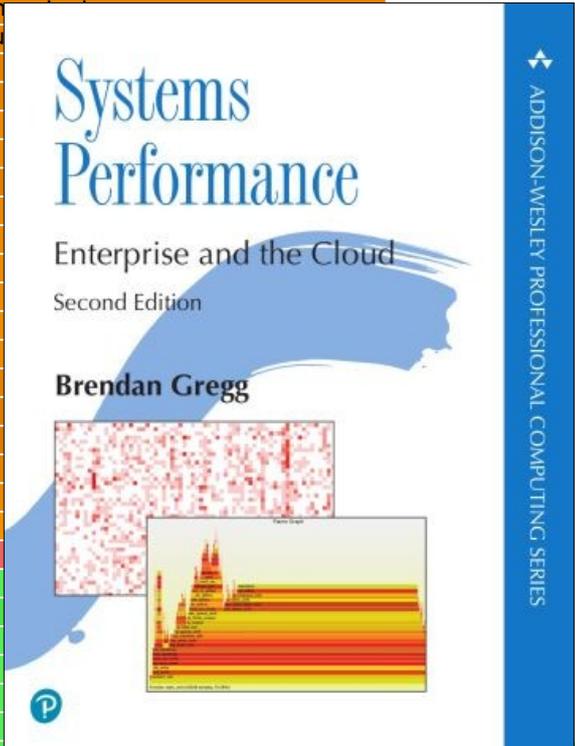
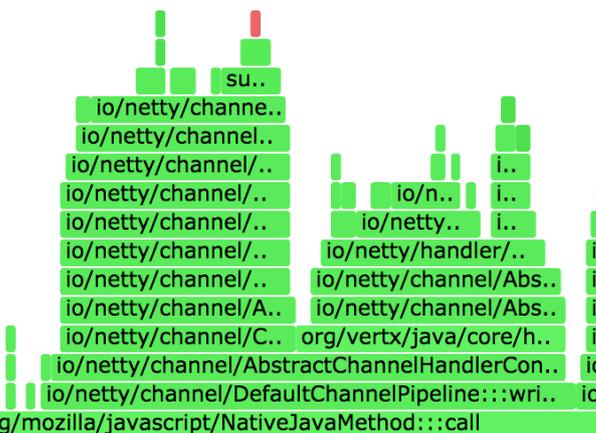
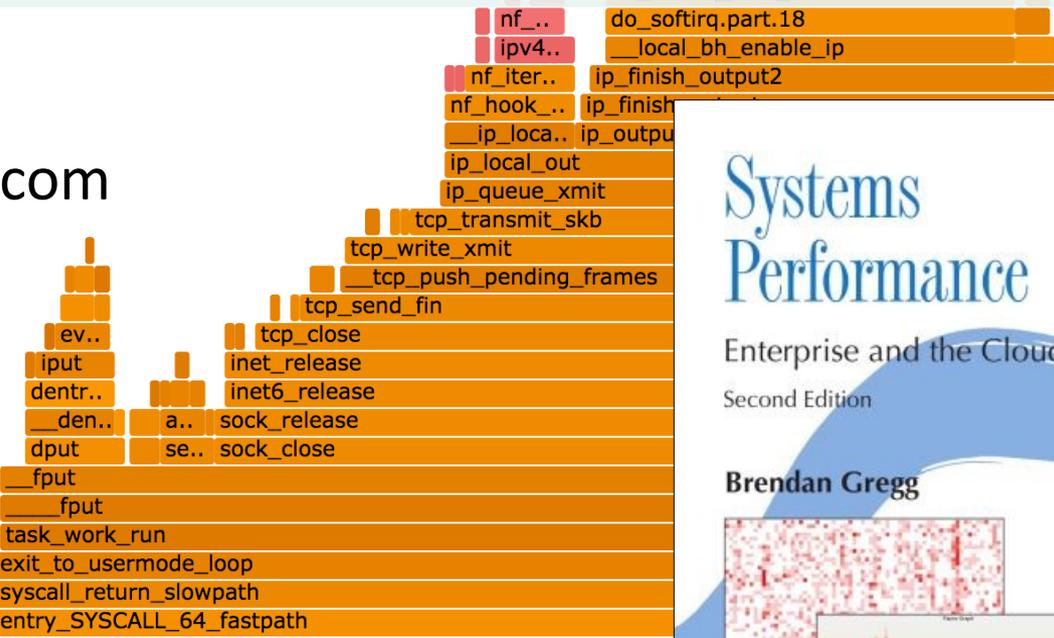
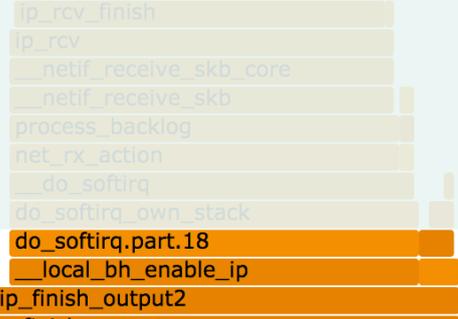
Thank you!

<http://www.brendangregg.com>

brendan@intel.com

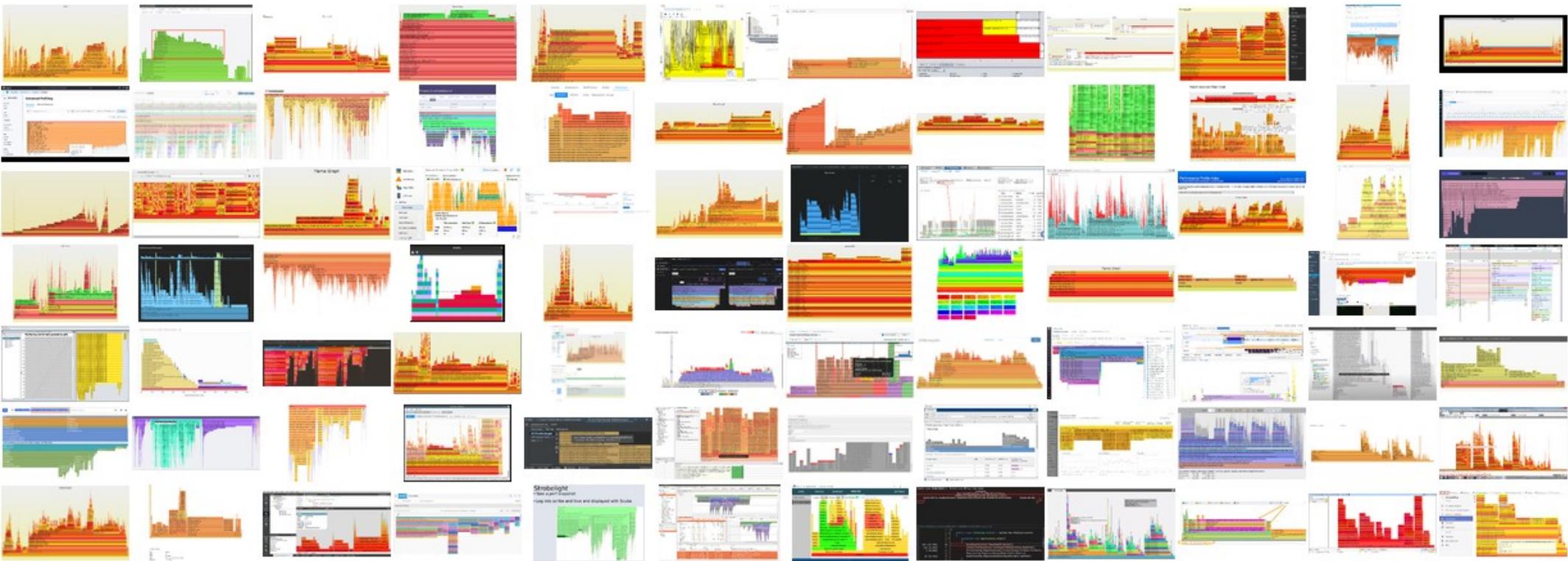
@brendangregg

Questions?



BONUS SLIDES

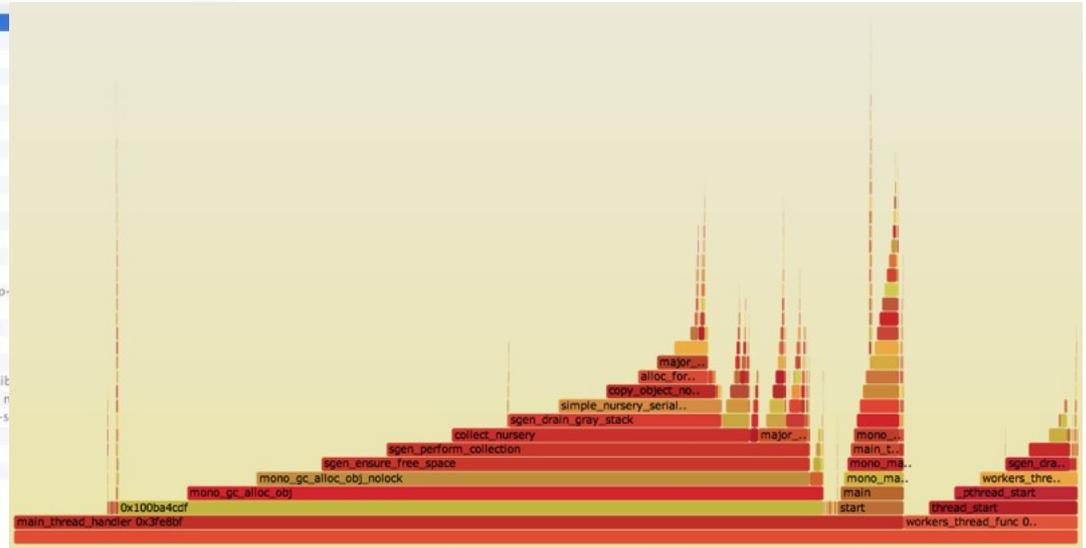
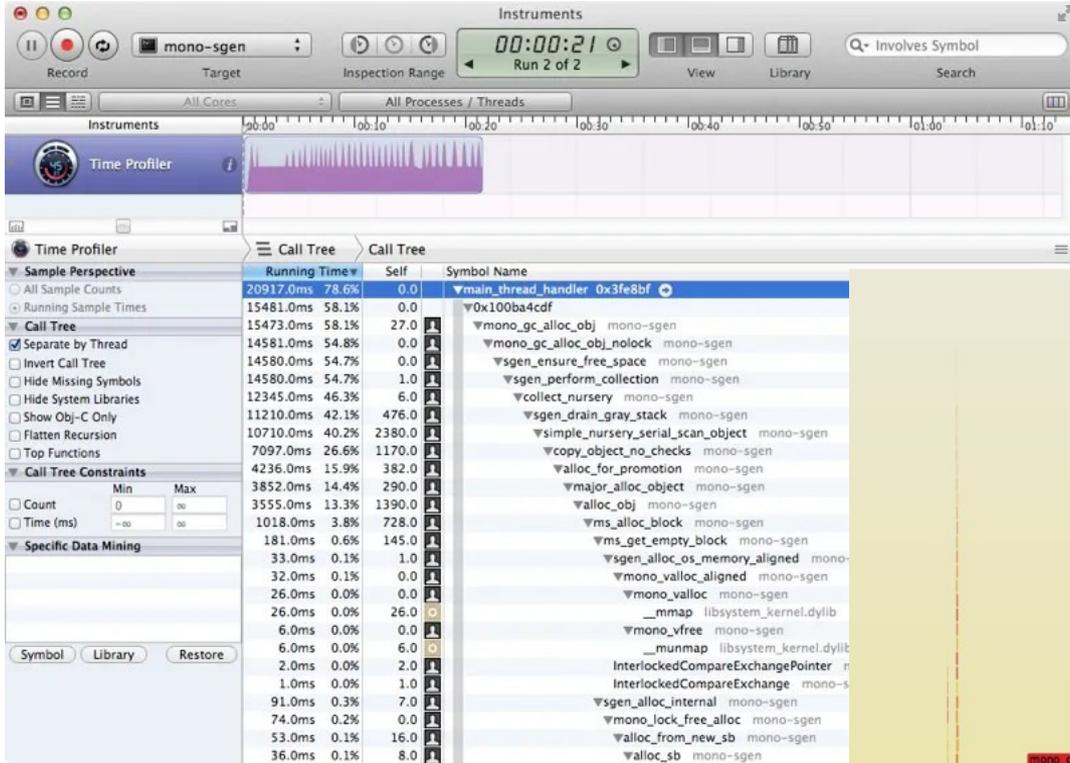
More Implementations



These are in addition to the earlier examples.

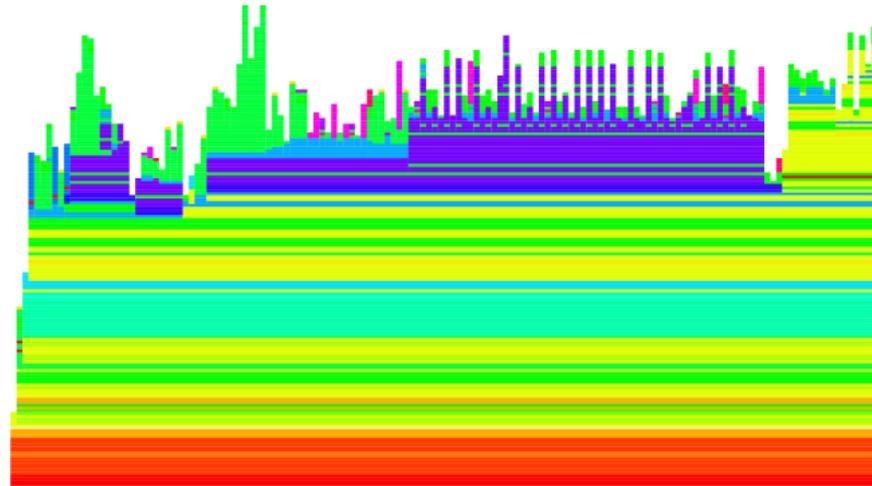
(Note: This is not an endorsement of any company/product or sponsored by anyone.)

OSX: Instruments (2012; converter)



Source: <https://schani.wordpress.com/2012/11/16/flame-graphs-for-instruments/>

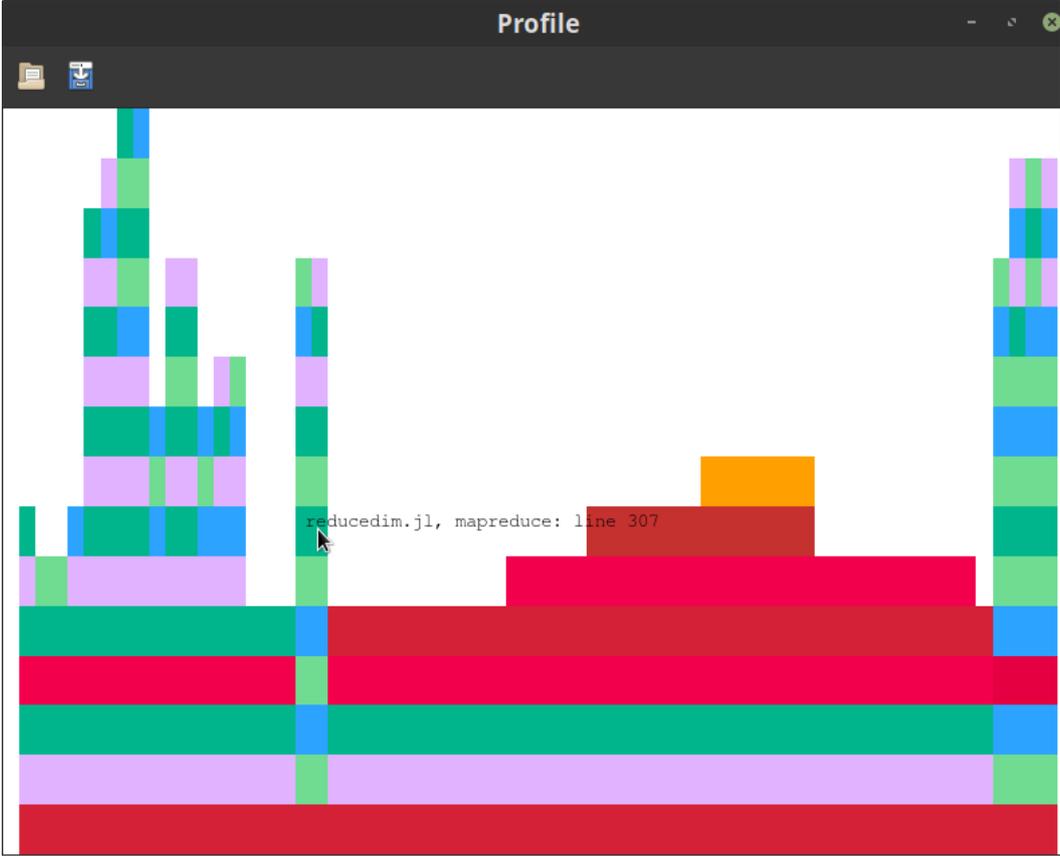
Ruby: mini-profiler (2013)



home/sam/Source/MiniProfiler/Ruby/lib/mini_profiler/profiler.rb:277:in `call' (148 samples - 99.33%)

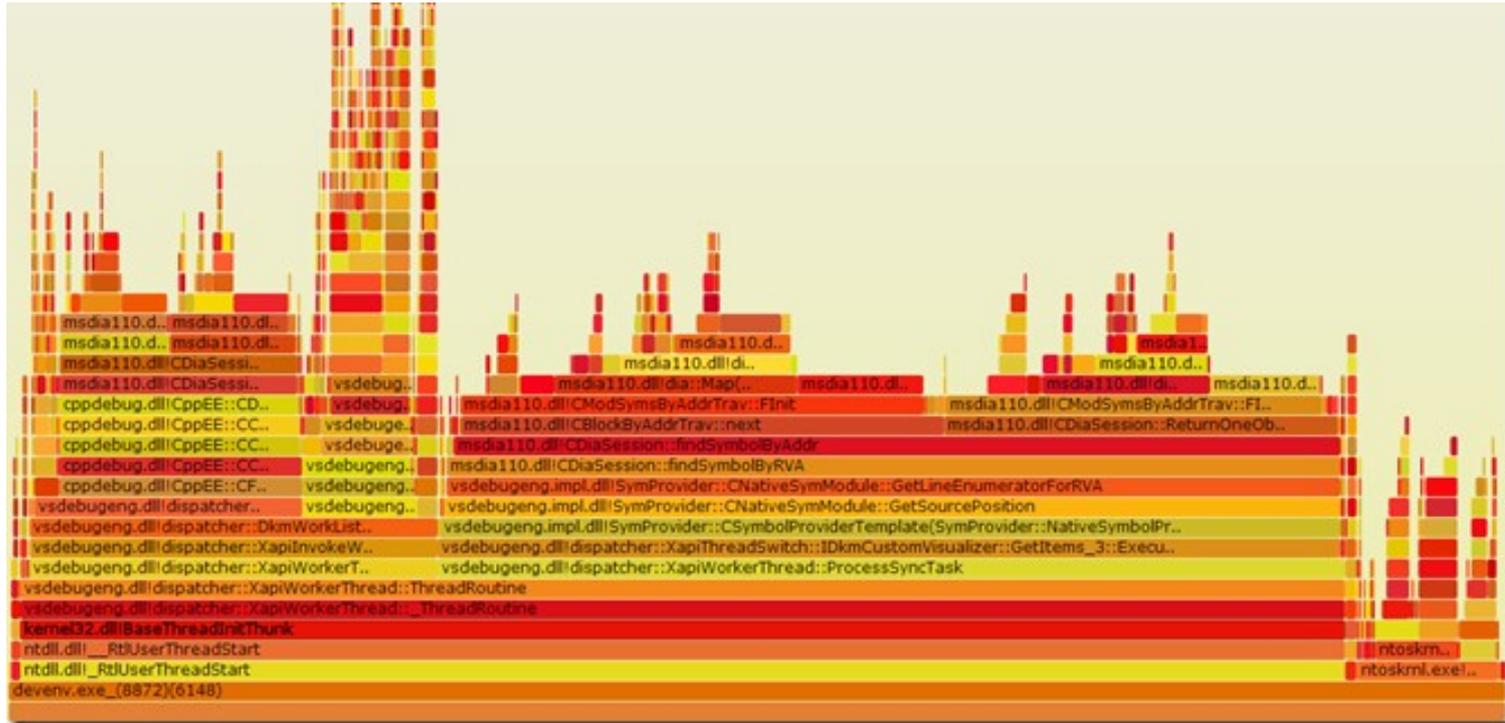
ruby 2.0.0-p0 turbo (148 samples - 99.33%)	bin 1.5.0 (148 samples - 99.33%)	eventmachine-1.0.1 (148 samples - 99.33%)	rails-3.2.12 (148 samples - 99.33%)	Ruby (148 samples - 99.33%)	actionpack-3.2.12 (148 samples - 99.33%)
rack-1.4.5 (147 samples - 98.66%)	silence_logger.rb:19:in `call' (147 samples - 98.66%)	quiet_logger.rb:10:in `call_with_quiet_assets' (147 samples - 98.66%)	activerecord-3.2.12 (147 samples - 98.66%)	activerecord-3.2.12 (147 samples - 98.66%)	message_bus (145 samples - 97.32%)
omniauth 1.1.1 (145 samples - 97.32%)	omniauth_browsesid af924667626c (145 samples - 97.32%)	journey-1.0.4 (145 samples - 97.32%)	discourse (144 samples - 96.64%)	redis-3.0.2 (3 samples - 2.01%)	redis-activerecord-3.2.3 (1 sample - 0.67%)
redis-stove-1.1.3 (1 sample - 0.67%)	multi_json-1.6.1 (116 samples - 77.85%)	active_model_serializers-0114e492388f (112 samples - 75.17%)	activemodel-3.2.12 (9 samples - 6.04%)	arel-3.0.2 (4 samples - 2.68%)	libas-0.6.1 (9 samples - 6.04%)
mail-2.4.4 (5 samples - 3.36%)	indefo-0.3.37 (3 samples - 2.01%)	redcarpet-2.2.2 (1 sample - 0.67%)			

Julia: ProfileView.jl (2013)



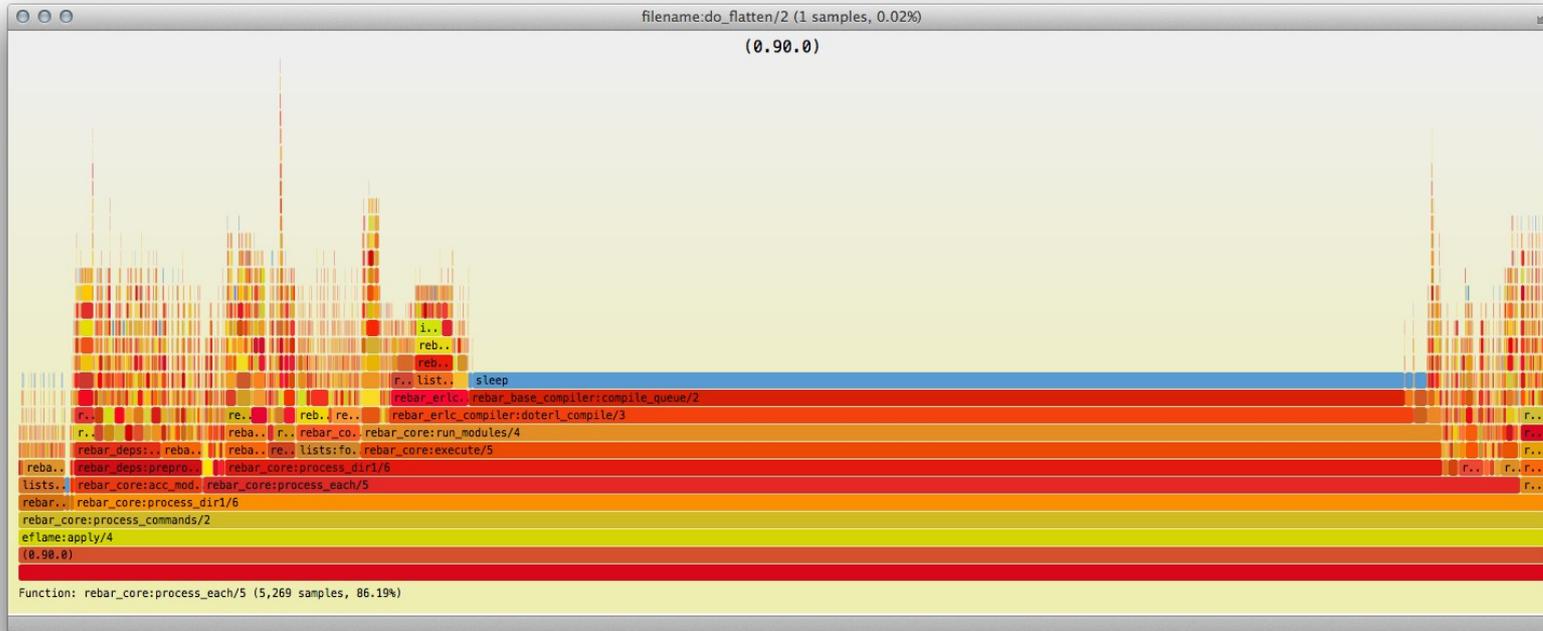
Source: <https://github.com/timholy/ProfileView.jl> (Tim Holy)

Windows: Xperf (2013; converter)



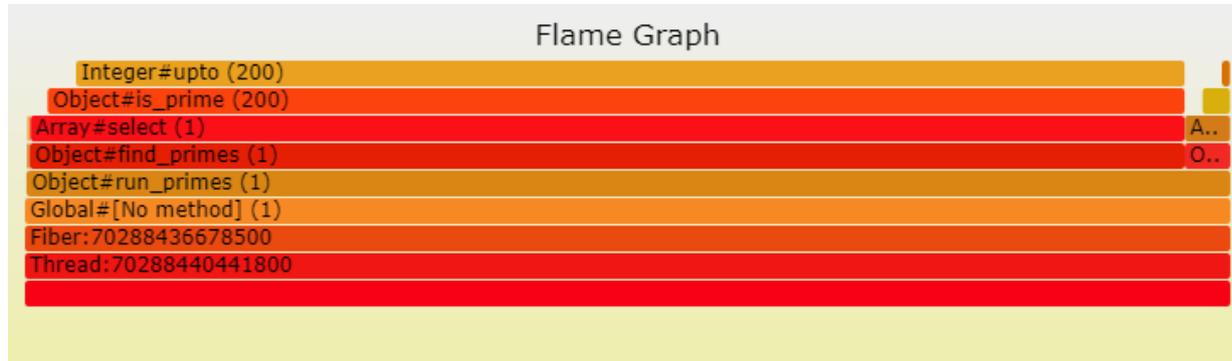
Source: <https://randomascii.wordpress.com/2013/03/26/summarizing-xperf-cpu-usage-with-flame-graphs/>
(Bruce Dawson)

Erlang: Eflame (2013)

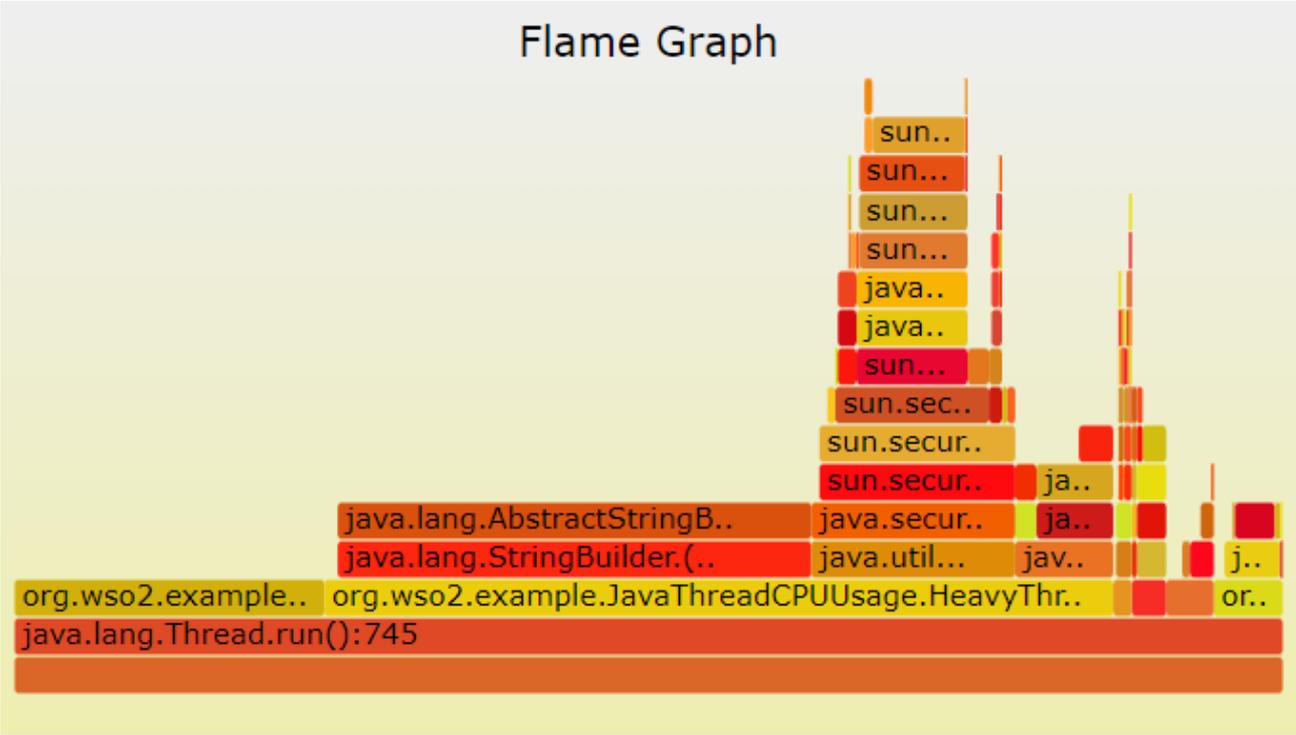


Source: <https://github.com/proger/eflame> (Volodymyr Ky)

Ruby: ruby-prof-flamegraph (2014)

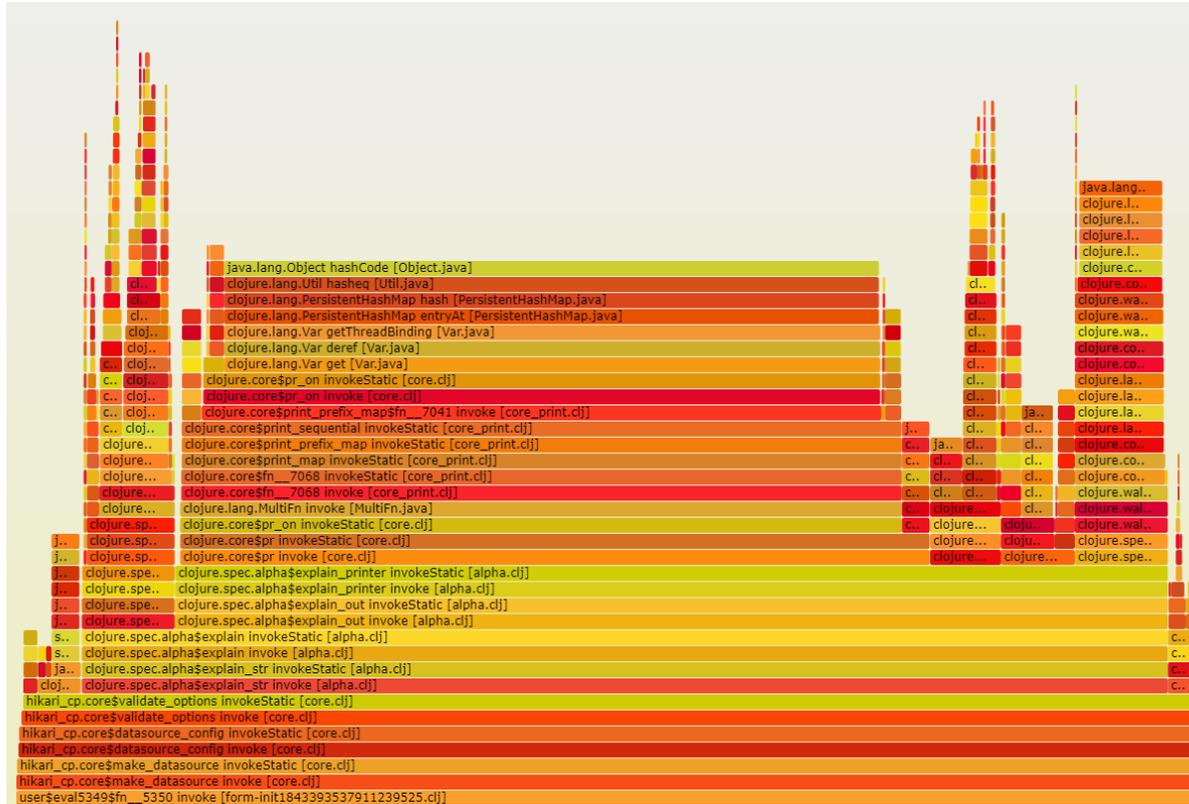


Java: jfr-flame-graph (2015)



Source: <http://isuru-perera.blogspot.com/2015/05/flame-graphs-with-java-flight-recordings.html>
(M. Isuru Tharanga Crishantha Perera)

Clojure: Flames (2015)



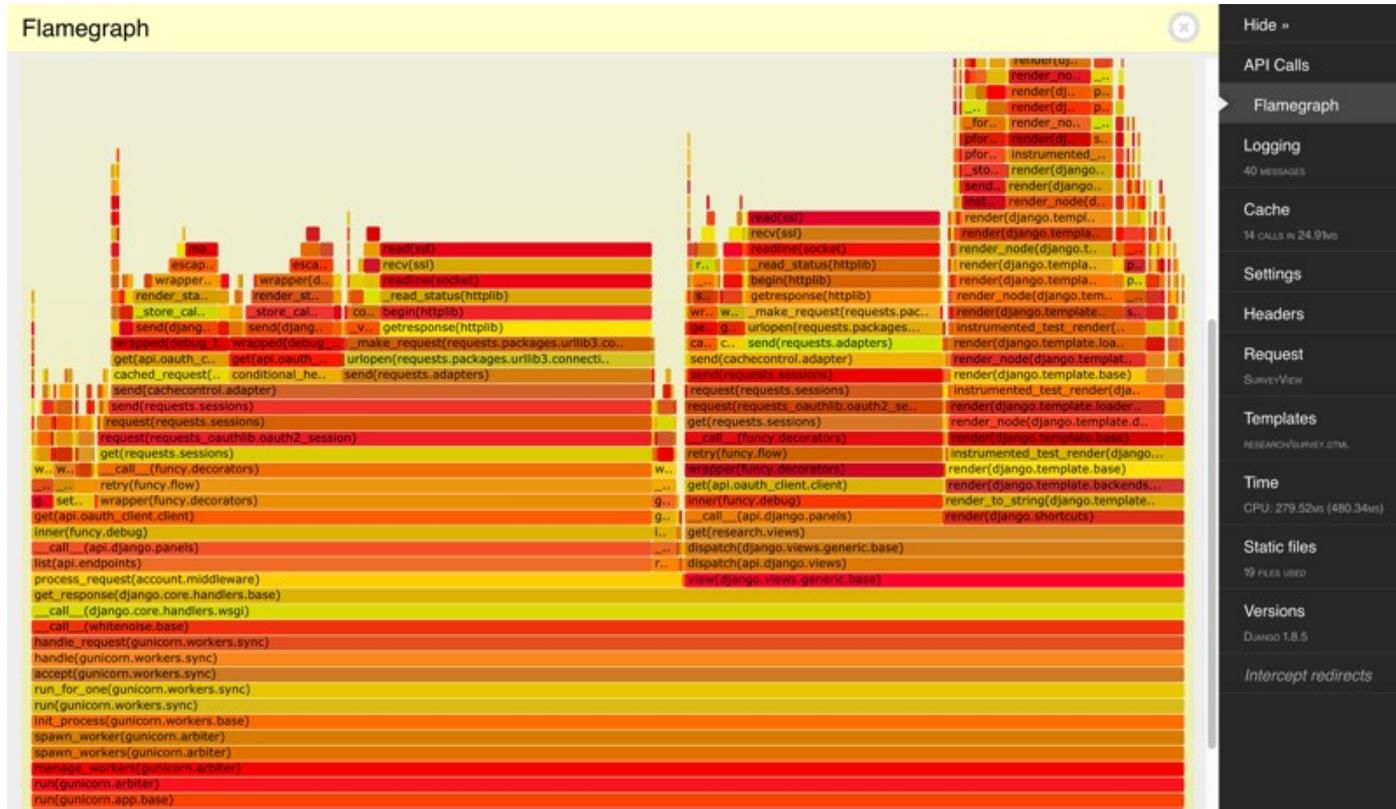
Source: <https://github.com/jstepien/flames/> (Jan Stępień)

Python: python-flamegraph (2015)



Source: <https://github.com/evanhempel/python-flamegraph> (Evan Hempel)

Django: djdt-flamegraph (2015)



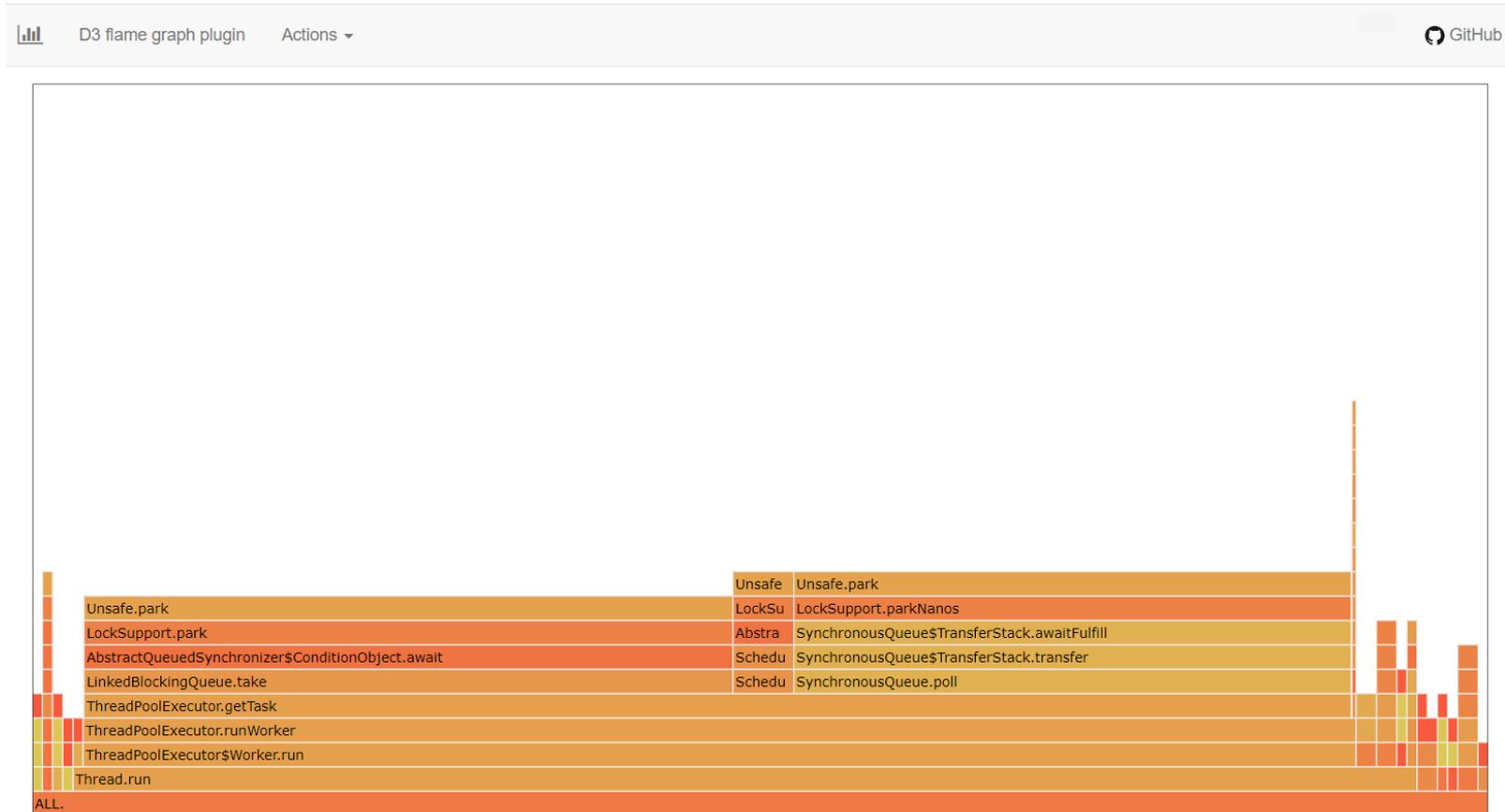
Source: <https://github.com/blopper/djdt-flamegraph> (Bo Lopker)

NodeSource: Nsolid (Node.js; 2015)



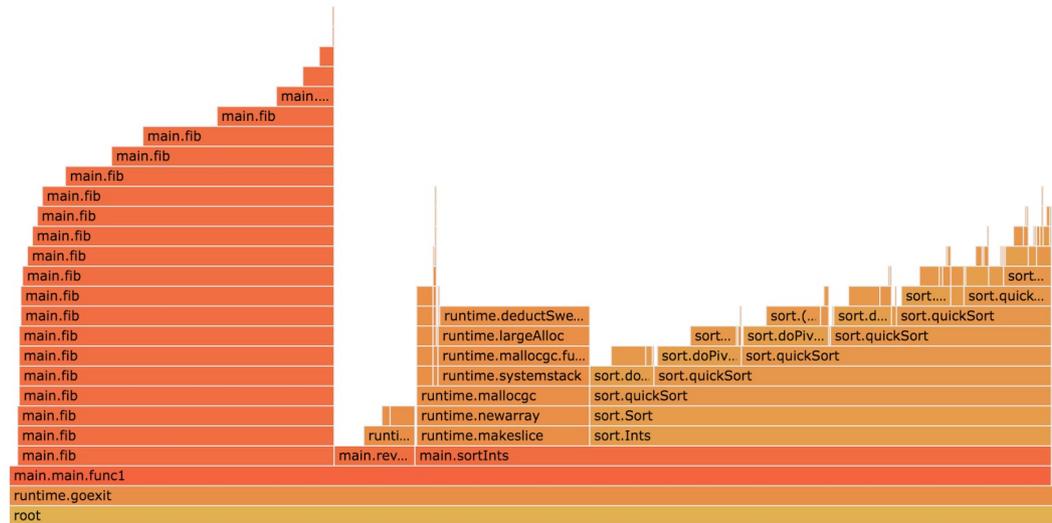
Source: <https://nodesource.com/blog/understanding-cpu-flame-graphs>

D3: d3-flame-graphs (2015)



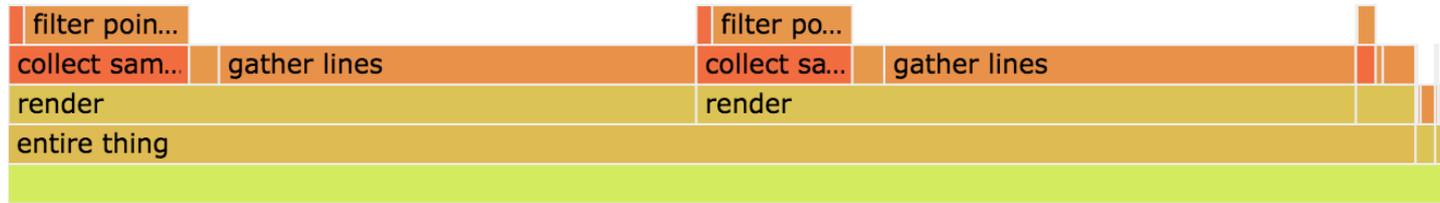
Source: <https://cimi.io/d3-flame-graphs/> (Alex Ciminian)

Golang: Goprofui (2015)

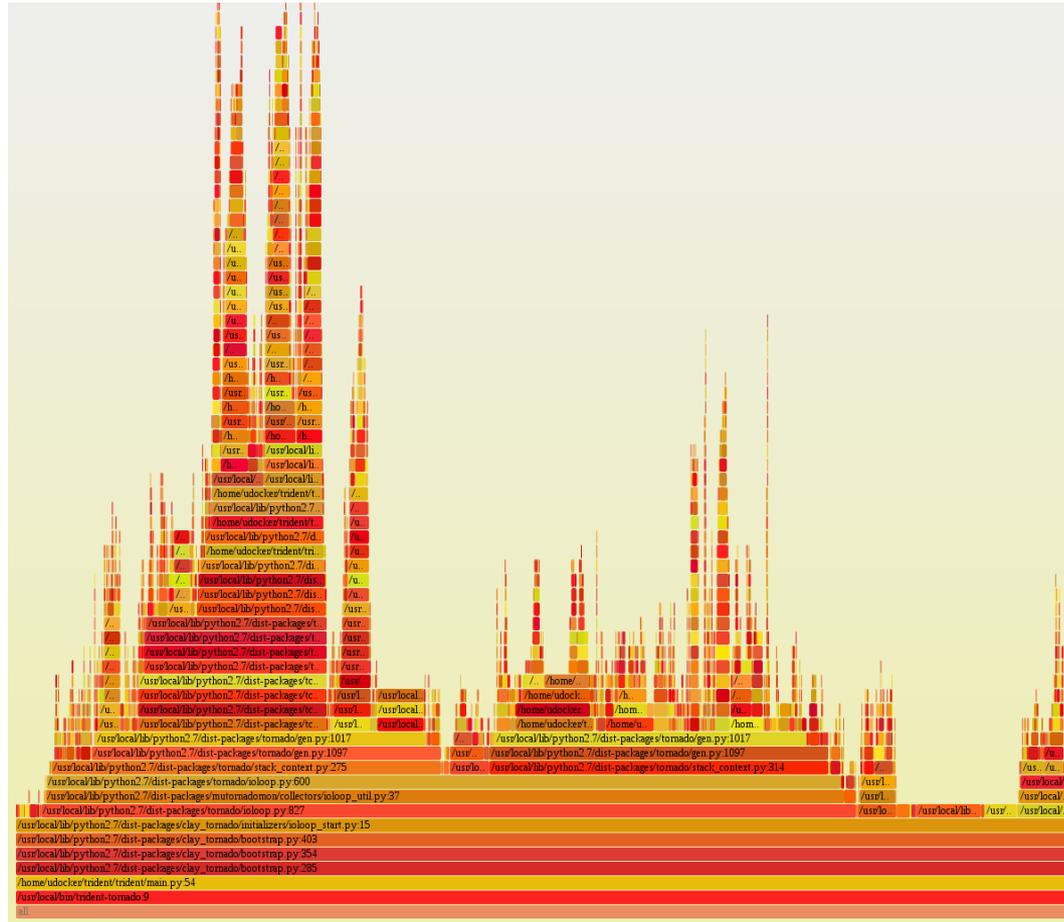


Source: <https://github.com/wirelessregistry/goprofui> (Srdjan Marinovic, Julia Allyce)

Rust: flame (2016)

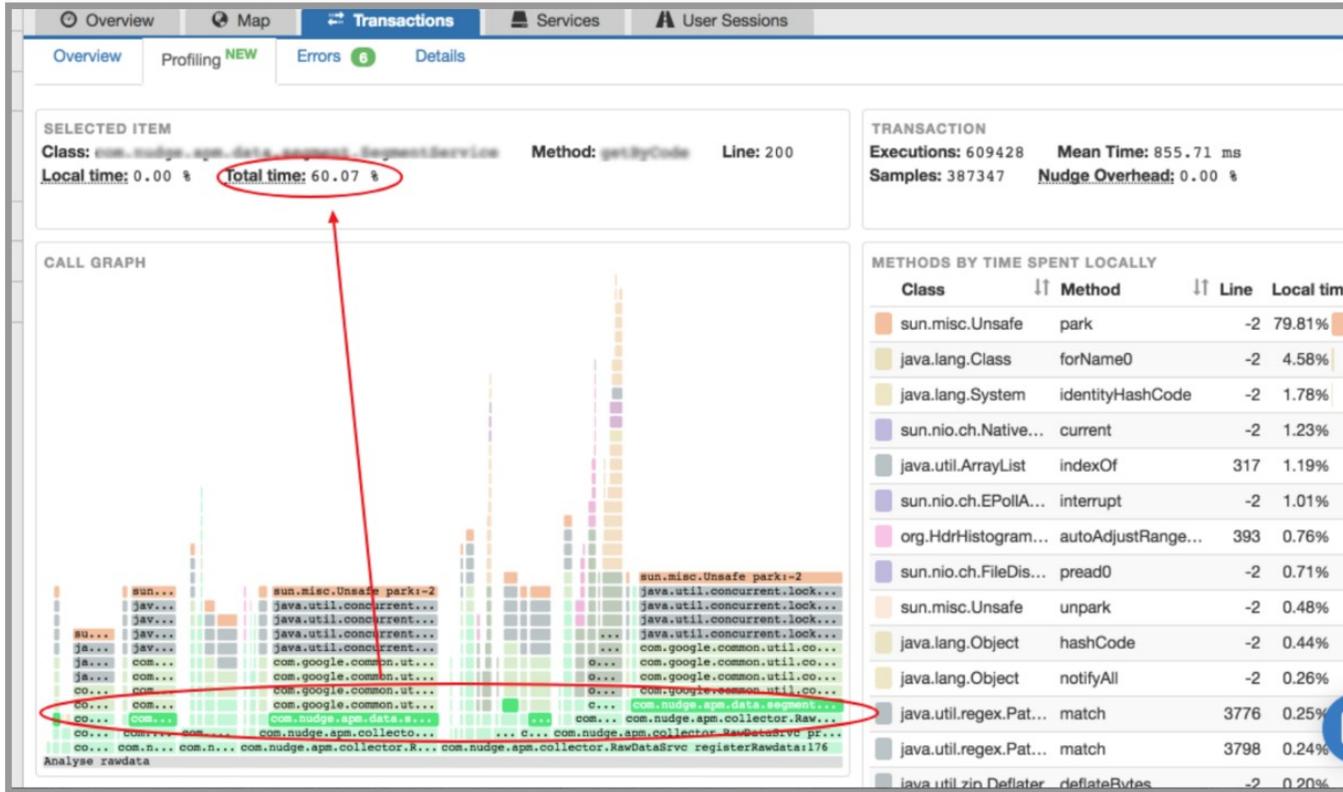


Uber: pyflame (Python; 2016)



Source: <https://www.uber.com/en-AU/blog/pyflame-python-profiler/>

Nudge: APM (for Java; 2017)



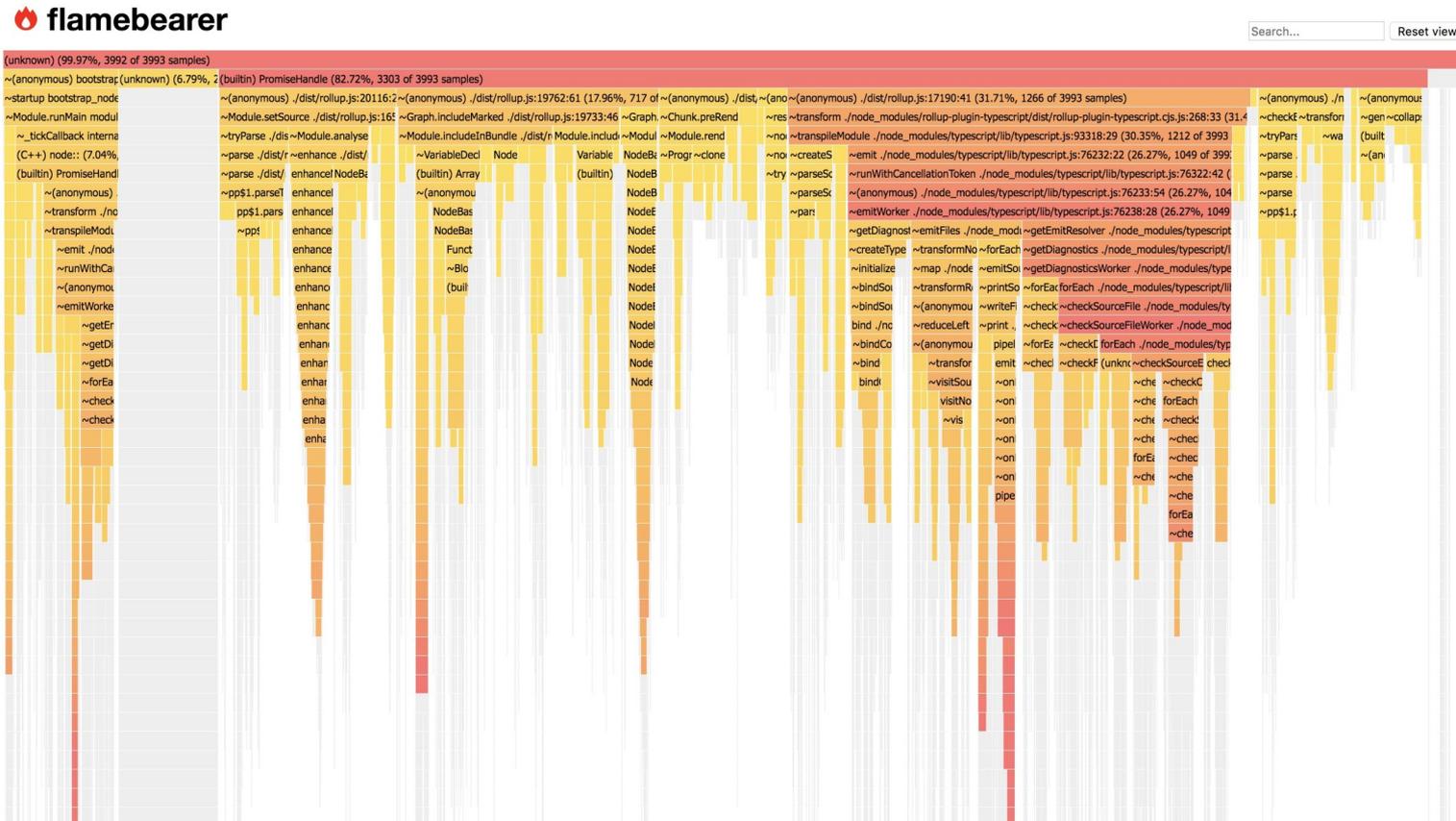
Source: <https://nudge-apm.com/features/#profiling>

Java: clj-async-profiler (2017)



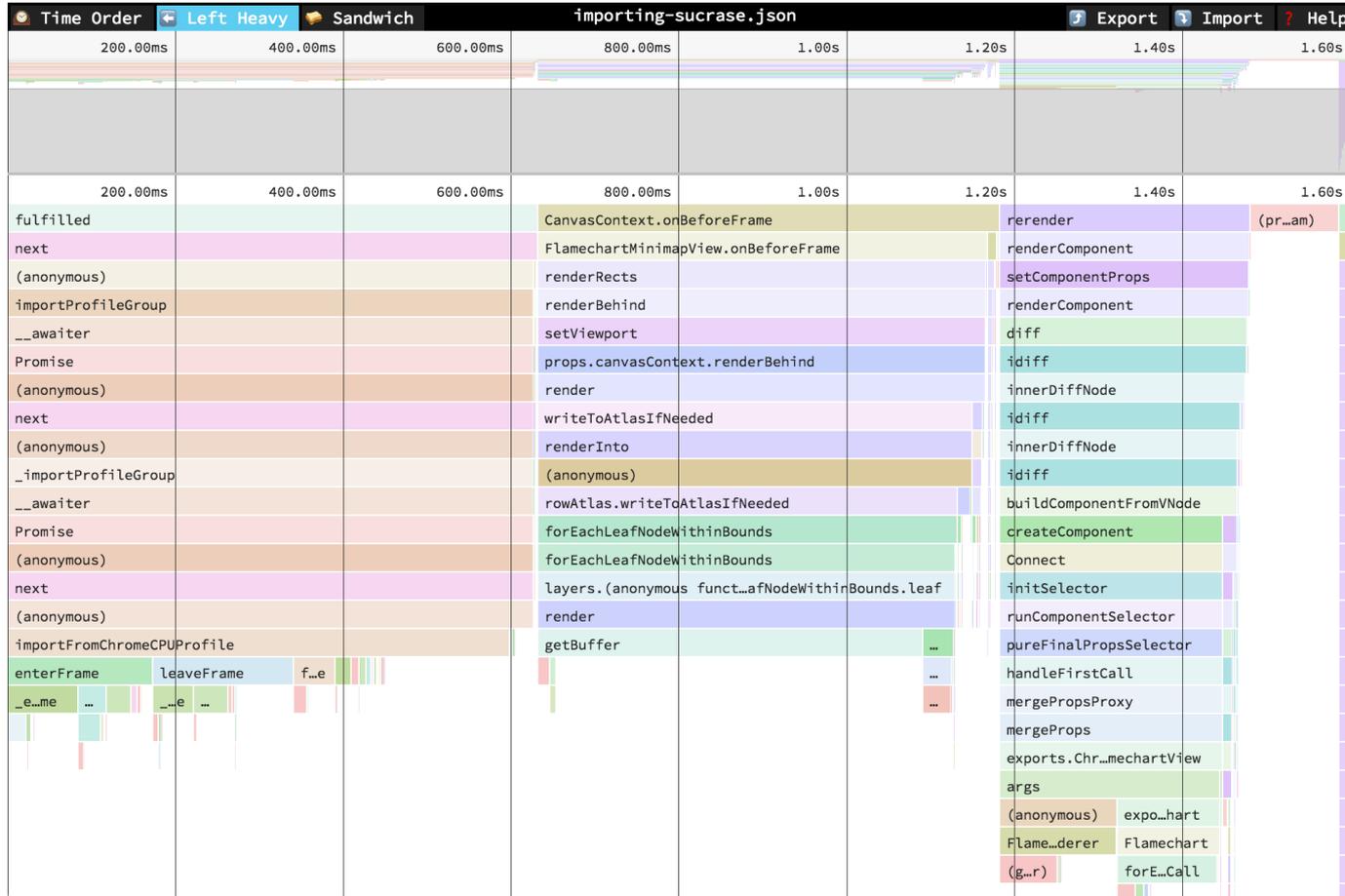
Source: <http://clojure-goes-fast.com/blog/profiling-tool-async-profiler/> (Alexander Yakushev)

Node.js: Flamebearer (2018)



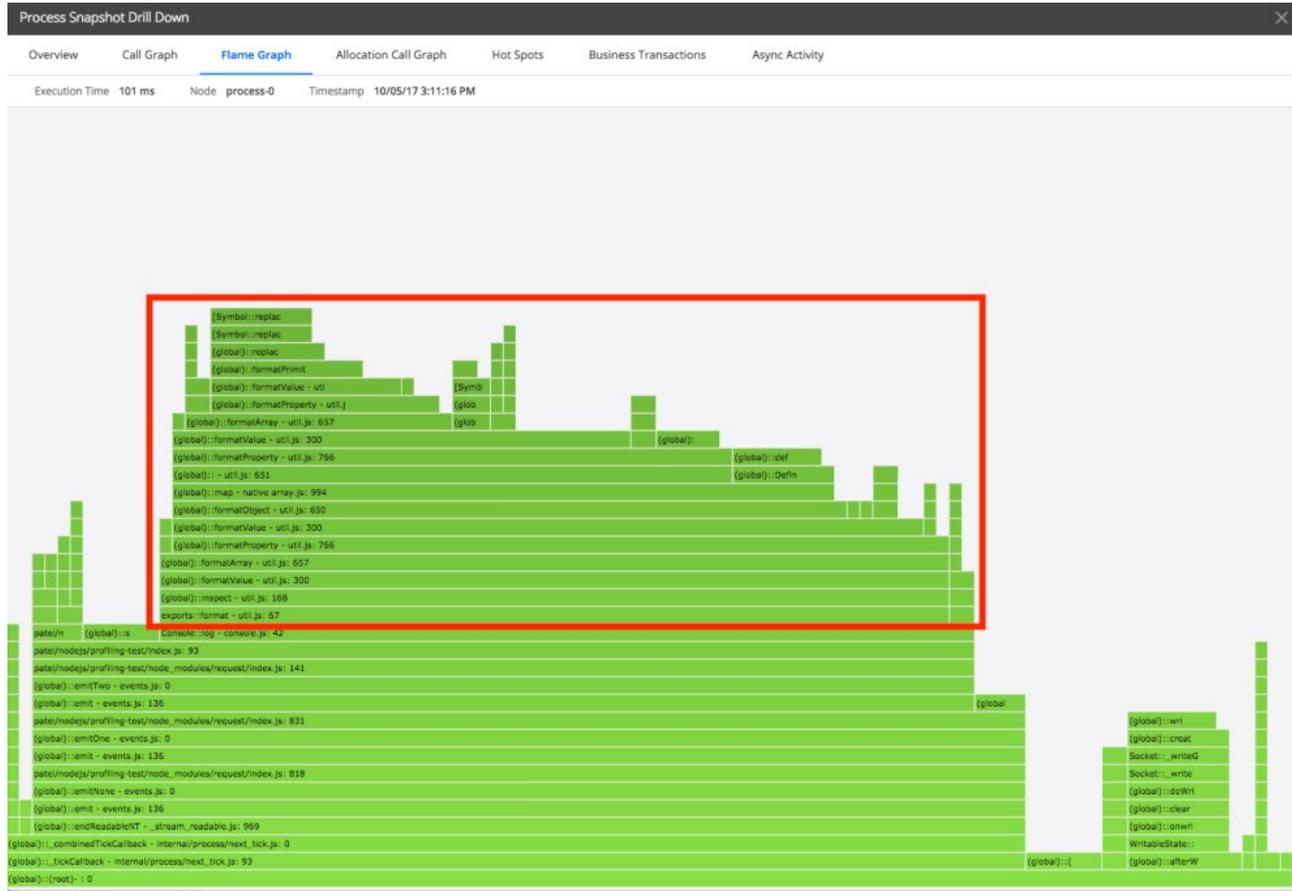
Source: <https://github.com/mapbox/flamebearer> (Volodymyr Agafonkin)

Speedscope: left heavy view (2018)



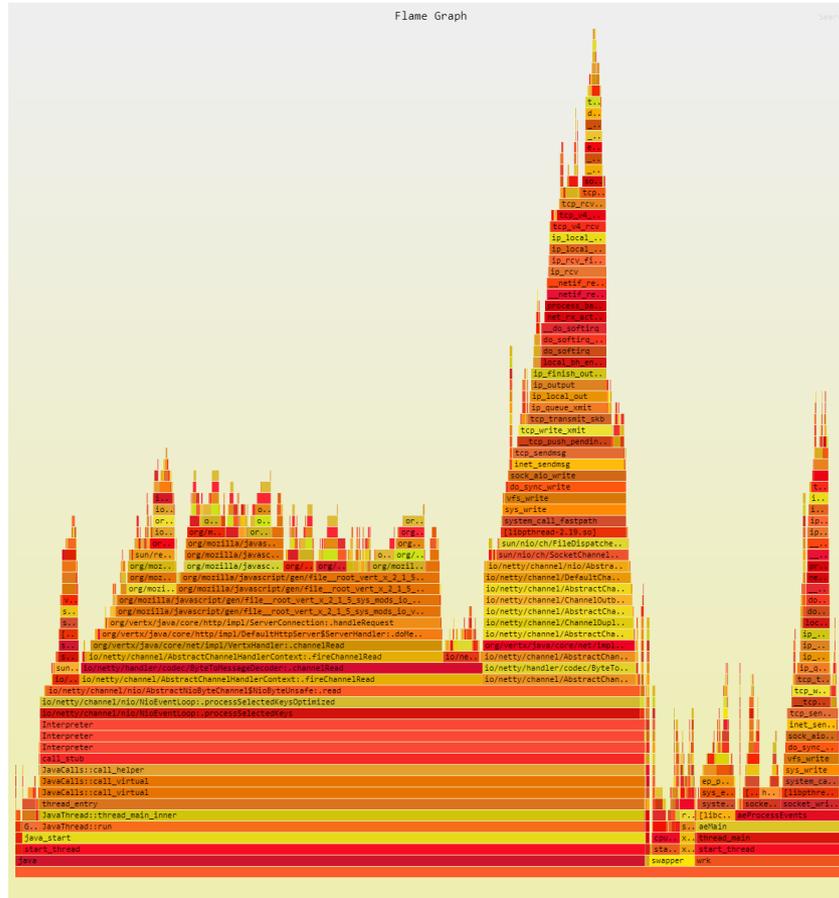
Source: <https://jamie-wong.com/post/speedscope/> (Jamie Wong)

AppDynamics: flame graph (2018; now Cisco)



Source: <https://docs.appdynamics.com/appd/20.x/en/application-monitoring/troubleshooting-applications/event-loop-blocking-in-node-js#EventLoopBlockinginNode.js-FlameGraph>

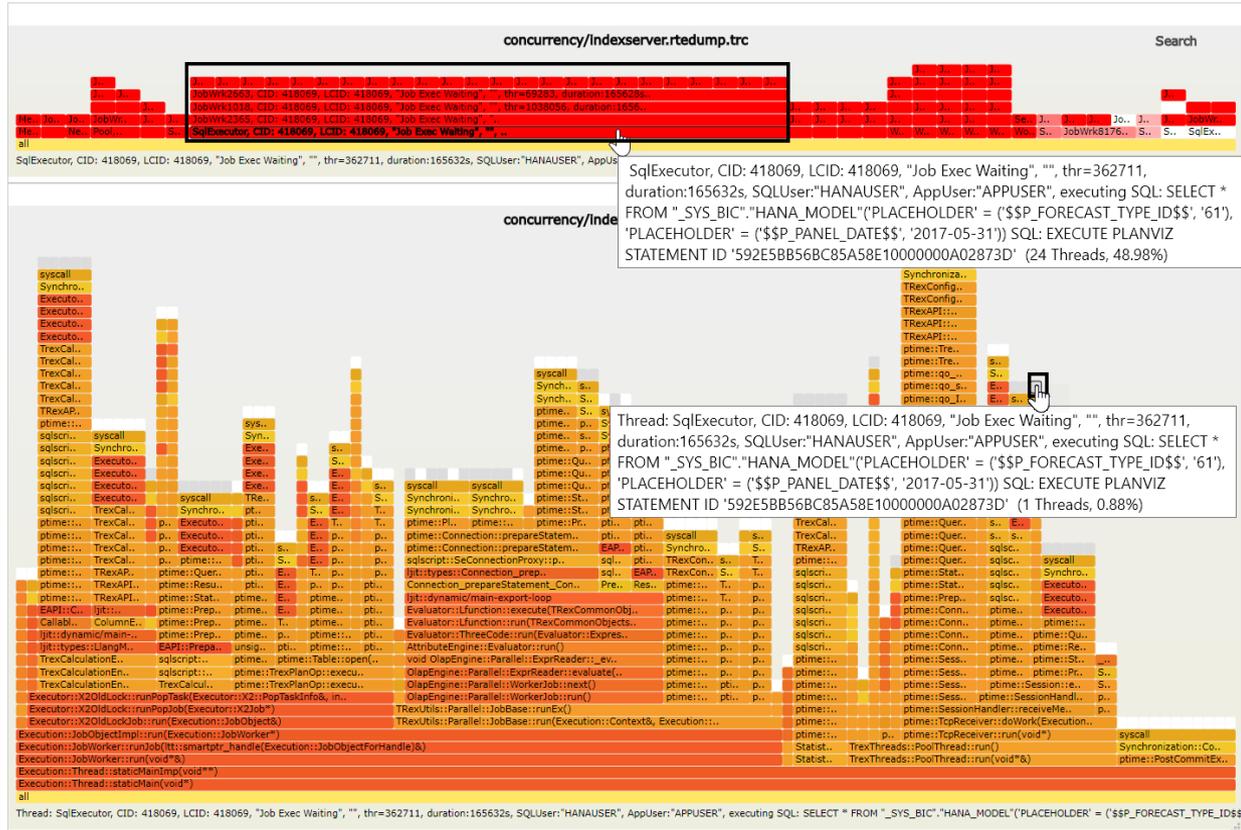
Inferno: flame graph (Rust port; 2019)



Source: <https://github.com/jonhoo/inferno> (Jon Gjengset)

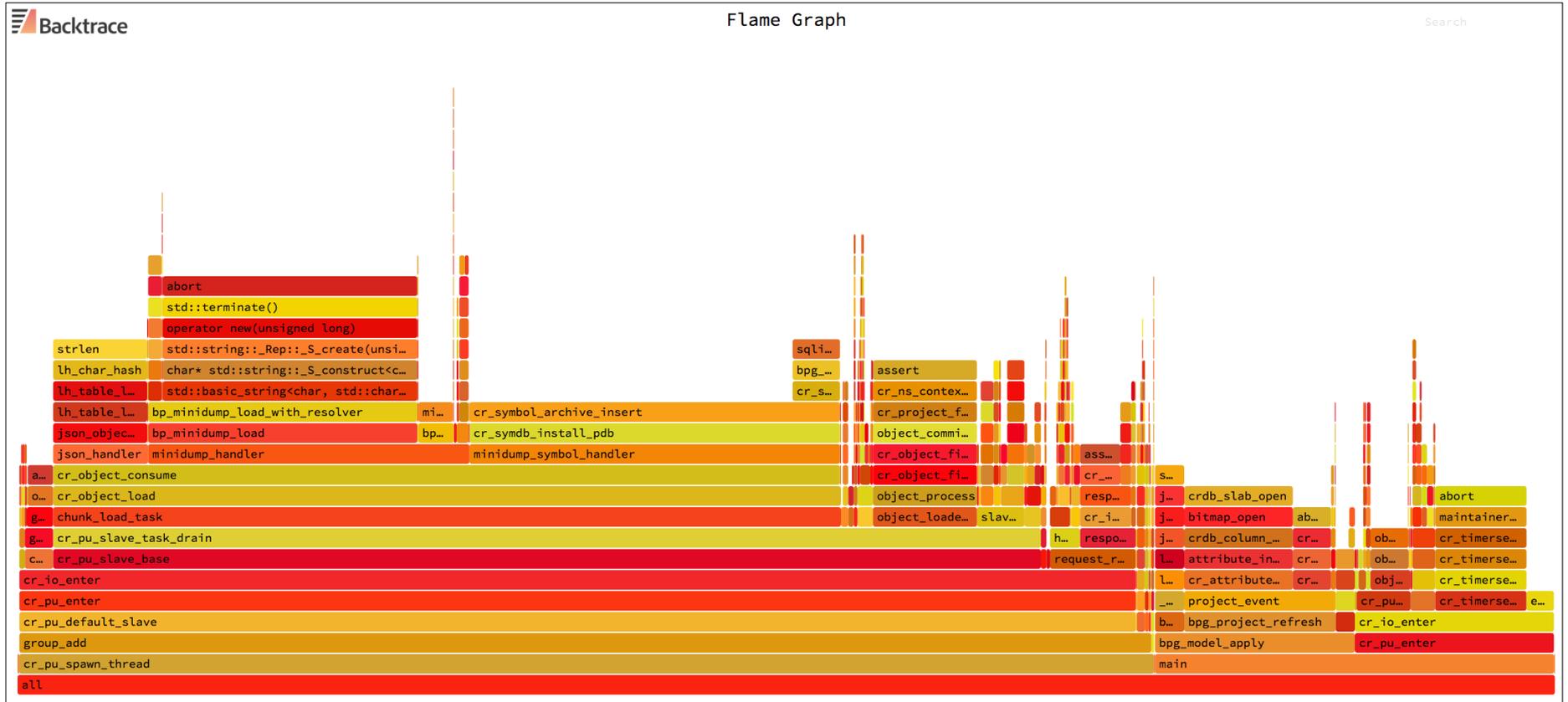
SAP: HANA Dump Analyzer (2019)

Mixed Concurrency Flame Graph



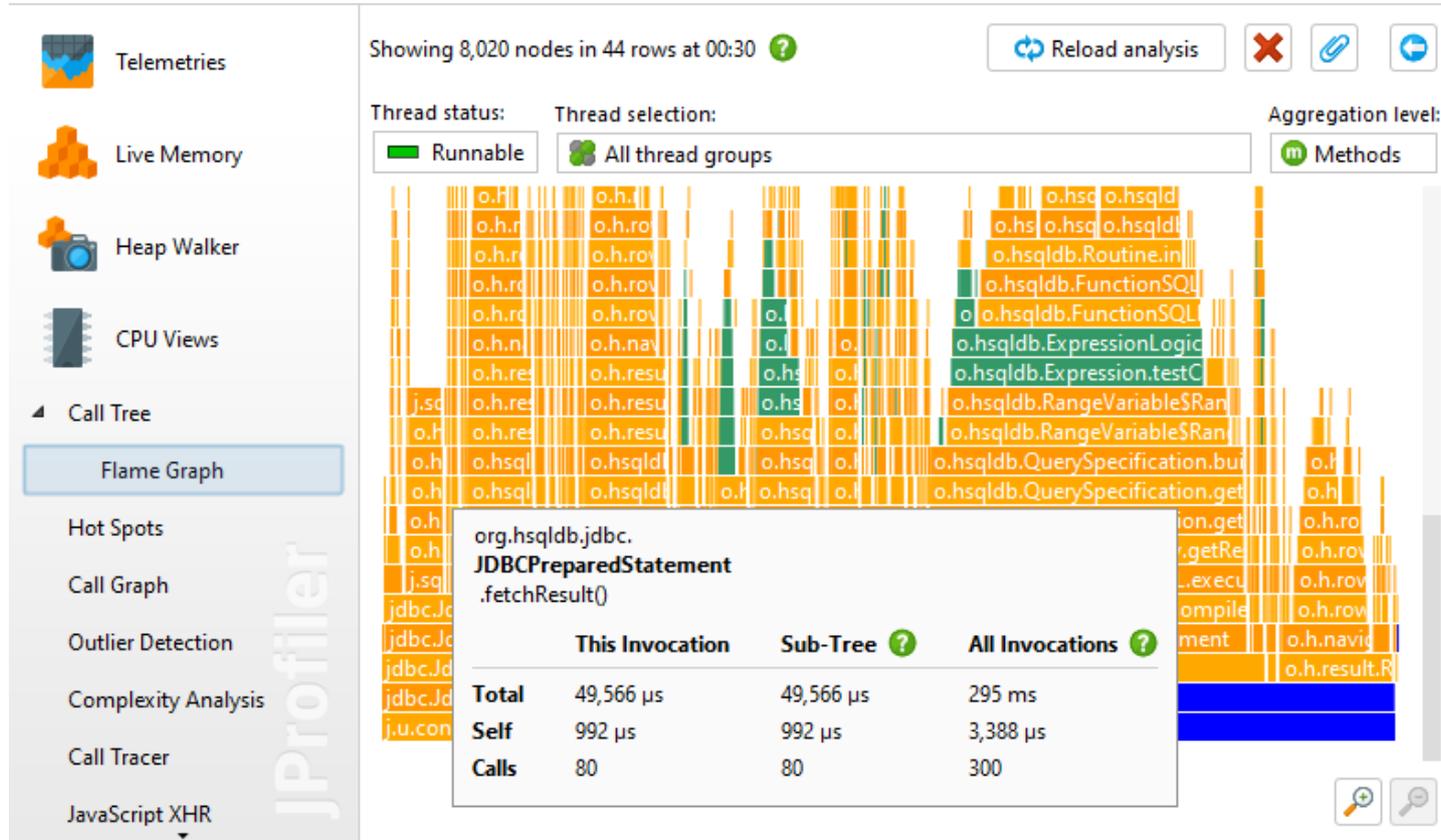
Source: <https://blogs.sap.com/2019/04/22/visualizing-olap-requests-on-sap-hana-system-with-concurrency-flame-graph-using-sap-hana-dump-analyzer/>

Backtrace: flame graph (2019)

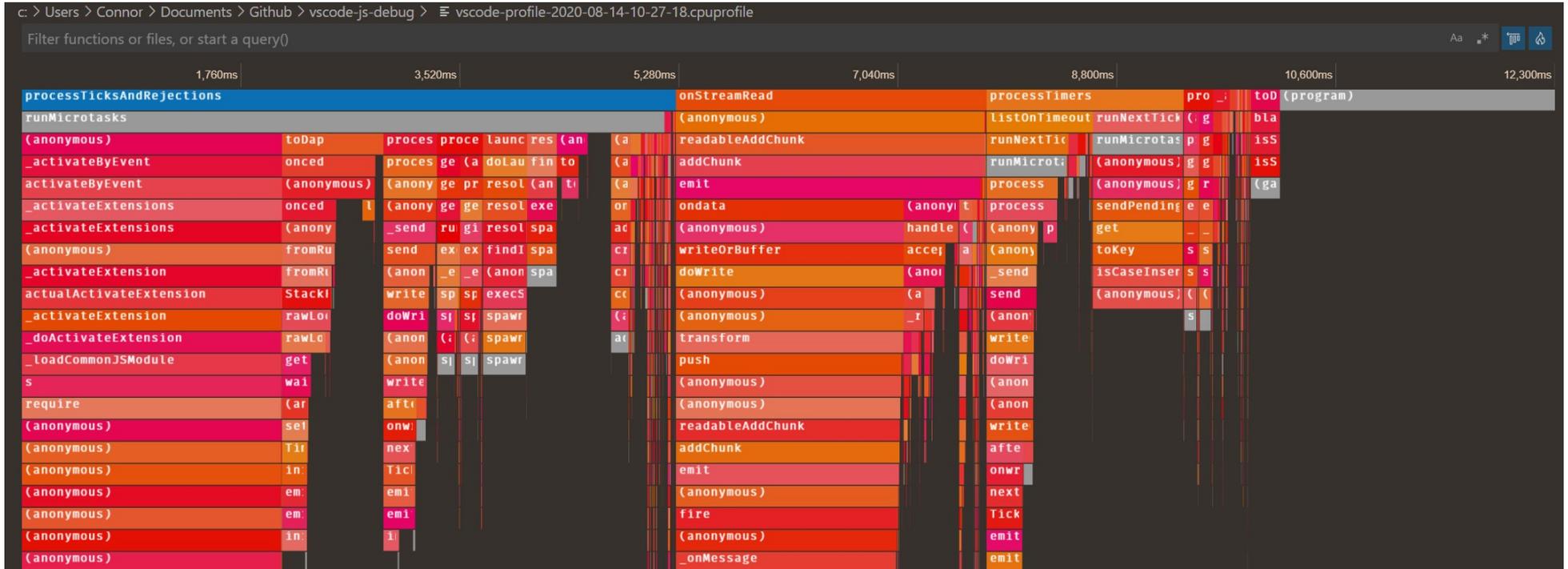


Source: <https://support.backtrace.io/hc/en-us/articles/360040515971-Flame-graphs>

ej-technologies: JProfiler Flame Graph (for Java; 2020)



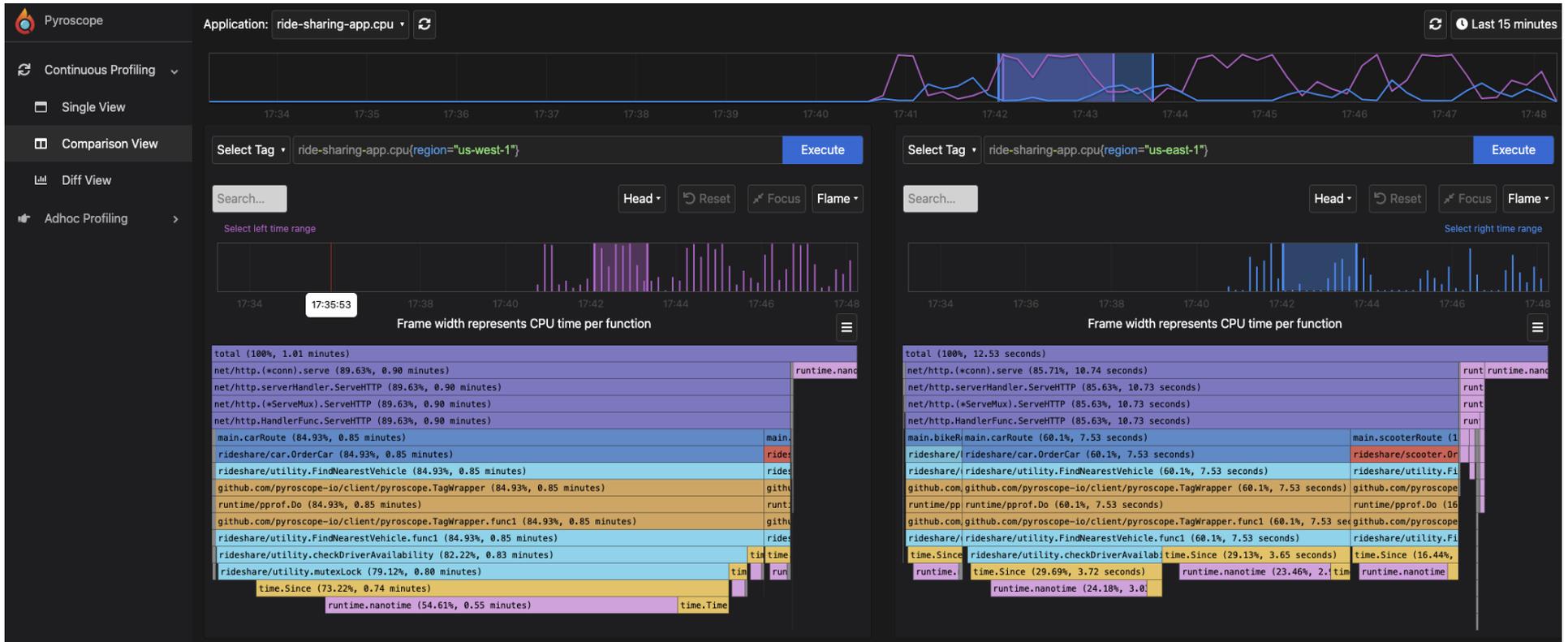
Microsoft Visual Studio: vscode-js-profile-flame (for JavaScript; 2020)



Left Heavy view

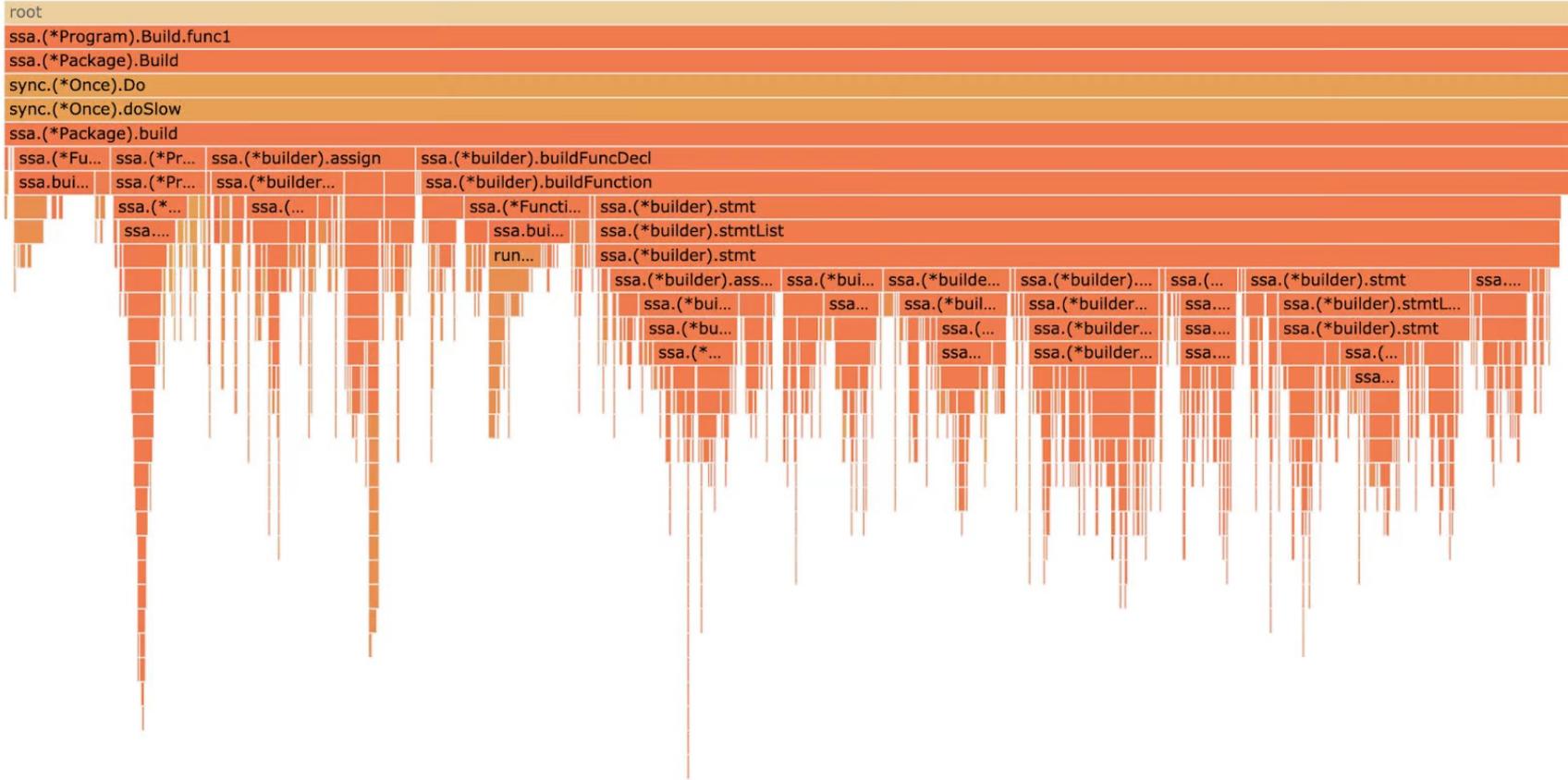
Source: <https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-js-profile-flame>

Pyroscope: flame graph (2020)



Source: <https://pyroscope.io/blog/what-is-a-flamegraph/>

Uber: pprof++ (2021; for Golang)



Source: <https://www.uber.com/en-AU/blog/pprof-go-profiler/> (Pengfei Su)

Lightstep: flame graph (2021)



Source: <https://www.instana.com/blog/instana-announces-the-industrys-first-commercial-continuous-production-profiler/>

Dynatrace: allocation flame graph (2021)



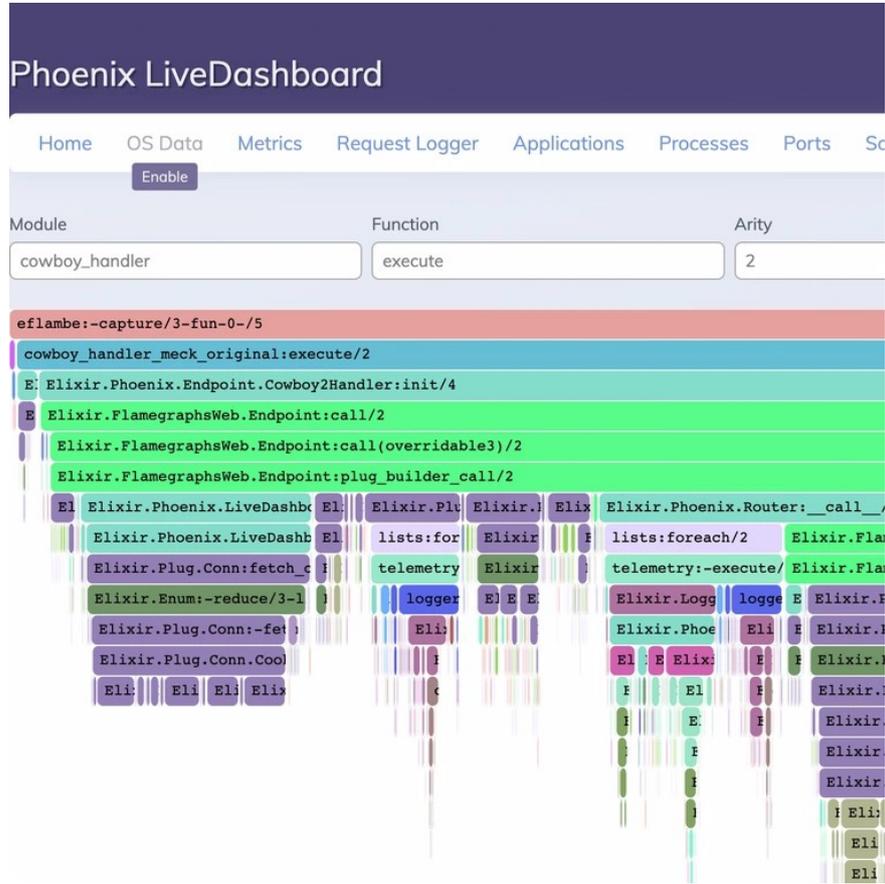
Source: <https://www.dynatrace.com/support/help/how-to-use-dynatrace/diagnostics/memory-profiling>

Pixie Labs: pod performance flamegraph (2021)



Source: <https://docs.pixielabs.ai/tutorials/pixie-101/profiler/>

Dockyard: Flame On (for Elixir apps; 2022)



Source: <https://dockyard.com/blog/2022/02/22/profiling-elixir-applications-with-flame-graphs-and-flame-on>
(Mike Binns)

Elastic: universal profiling (2022)

The screenshot displays the Elastic Universal Profiling interface. The top navigation bar includes the Elastic logo, a search bar with the text "Find apps, content, and more. Ex: Discover", and user profile icons. The breadcrumb trail shows "Observability" > "Universal Profiling" > "Flamegraphs" > "Flamegraph".

The left sidebar contains a navigation menu with sections: "Observability" (Overview, Alerts, Cases), "Logs" (Stream, Anomalies, Categories), "Infrastructure" (Inventory, Metrics Explorer), "APM" (Services, Traces, Dependencies, Service Map), and "Uptime" (Monitors, TLS Certificates).

The main content area is titled "Universal Profiling" and has two tabs: "Flamegraph" (selected) and "Differential flamegraph". Below the tabs is a search bar containing "container.name:'vmedia'", a date range selector for "Sep 20, 2022 @ 04:00:00.000" to "Sep 20, 2022 @ 06:00:00.000", and a "Refresh" button. A "Show information window" toggle is also present.

The central visualization is a flamegraph showing CPU time distribution. A tooltip is displayed over a specific function call, providing the following details:

- Python: put_media_key() in video_key_manager.py #46
- Inclusive CPU: 11.65%
- Exclusive CPU: 0.00%
- Samples: 12370

The tooltip also shows the function call stack for the selected point, including:

- root: Represents 100% of CPU time.
- Python: _handle_and_close_when_done() in baseserver.py #34
- Python: handle() in gevent.py #119
- Python: handle() in base_async.py #55
- Python: handle_request() in gevent.py #127
- Python: handle_request() in base_async.py #108
- Python: sentry_patched_wsgi_app() in flask.py #80
- Python: __call__() in wsgi.py #122
- Python: <lambda>() in flask.py #85
- Python: __call__() in app.py #2086
- Python: wrapper() in helpers.py #20
- Python: traced_wsgi_app() in patch.py #255
- Python: wsgi_app() in app.py #2073
- Python: full_dispatch_request() in app.py #1516
- Python: wrapper() in helpers.py #20
- Python: _traced_request() in patch.py #458
- Python: dispatch_request() in app.py #1480
- Python: trace_func() in wrappers.py #19
- Python: _wrapper() in metric_collector.py #403
- Python: wrapper() in flask_vauth_client.py #271
- Python: _wrapper() in decorators.py #61
- Python: submit_segments() in segment_retention.py #93
- Python: get_latest_video() in video_manager.py #578
- Python: query() in video_manager.py #77

... and more

The screenshot shows the GitHub search interface. The search bar contains the query "flamegraph OR flame graph". On the left, a sidebar lists various categories with their respective counts: Repositories (407), Code (29K), Commits (72K+), Issues (19K), Discussions (158), Packages (5), Marketplace (0), Topics (6), and Wikis (413). The main content area displays three security-related messages and a search result for "brendangregg/FlameGraph". The text "407 repository results" is circled in red. A "Sort: Best match" dropdown is visible on the right. The date "(Dec 2022)" is noted at the bottom right of the screenshot.

Category	Count
Repositories	407
Code	29K
Commits	72K+
Issues	19K
Discussions	158
Packages	5
Marketplace	0
Topics	6
Wikis	413

(Dec 2022)

Thanks for all the open source contributions!