# The Flame Graph

**THIS VISUALIZATION OF SOFTWARE EXECUTION IS A NEW NECESSITY FOR PERFORMANCE PROFILING AND DEBUGGING**

BRENDAN GREGG, NETFLIX

An everyday problem in our industry is understanding how software is consuming resources, particularly CPUs. What exactly is consuming how much, and how did this change since the last software version? These questions can be answered using software profilers, tools that help direct developers to optimize their code and operators to tune their environment. The output of profilers can be verbose, however, making it laborious to study and comprehend. The flame graph provides a new visualization for profiler output and can make for much faster comprehension, reducing the time for root cause analysis.

In environments where software changes rapidly, such as the Netflix cloud microservice architecture, it is especially important to understand profiles quickly. Faster comprehension can also make the study of foreign software more successful, where one's skills, appetite, and time are strictly limited.

Flame graphs can be generated from the output of many different software profilers, including profiles for different resources and event types. Starting with CPU profiling, this

article describes how flame graphs work, then looks at the real-world problem that led to their creation.

## CPU PROFILING

A common technique for CPU profiling is the sampling of stack traces, which can be performed using profilers such as Linux perf_events and DTrace. The stack trace is a list of function calls that show the code-path ancestry. For example, the following stack trace shows each function as a line, and the top-down ordering is child to parent:

```
SpinPause
StealTask::do_it
GCTaskThread::run
java_start
start_thread
```

Balancing considerations that include sampling overhead, profile size, and application variation, a typical CPU profile might be collected in the following way: stack traces are sampled at a rate of 99 times per second (not 100, to avoid lock-step sampling) for 30 seconds across all CPUs. For a 16-CPU system, the resulting profile would contain 47,520 stack-trace samples. As text, this would be hundreds of thousands of lines.

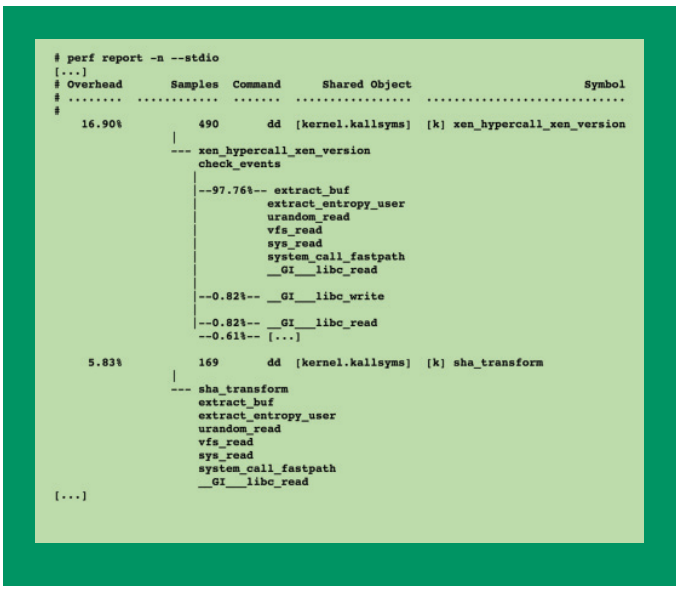
Fortunately, profilers have ways to condense their output. DTrace, for example, can measure and print unique stack traces, along with their occurrence count. This approach is more effective than it might sound: identical stack traces may

be repeated during loops or when CPUs are in the idle state. These are condensed into a single stack trace with a count.

Linux perf_events can condense profiler output even further: not only identical stack trace samples, but also subsets of stack traces can be coalesced. This is presented as a tree view with counts or percentages for each code-path branch, as shown in figure 1.

In practice, the output summary from either DTrace or perf_events is sufficient to solve the problem in many cases, but there are also cases where the output produces a wall of text, making it difficult or impractical to comprehend much of the profile.

FIGURE 1: **SAMPLE LINUX PERF _ EVENTS TREE VIEW**



## THE PROBLEM

The problem that led to the creation of flame graphs was application performance on the Joyent public cloud.[3] The application was a MySQL database that was consuming around 40 percent more CPU resources than expected.

DTrace was used to sample user-mode stack traces for the application at 997 Hz for 60 seconds. Even though DTrace printed only unique stack traces, the output was 591,622 lines long, including 27,053 unique stack traces. Fortunately, the last screenful—which included the most frequently sampled stack traces—looked promising, as shown in figure 2.

The most frequent stack trace included a MySQL `calc_sum_of_all_status()` function, indicating that it was processing a "show status" command. Perhaps the customer had enabled aggressive monitoring, explaining the higher CPU usage?

FIGURE 2: **MYSQL DTRACE PROFILE SUBSET**

```
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"/ {
    @[ustack()] = count(); } tick-60s { exit(0); }'
dtrace: description 'profile-997 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  75195                         :tick-60s
[...]
              libc.so.1`__priocntlset+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
```

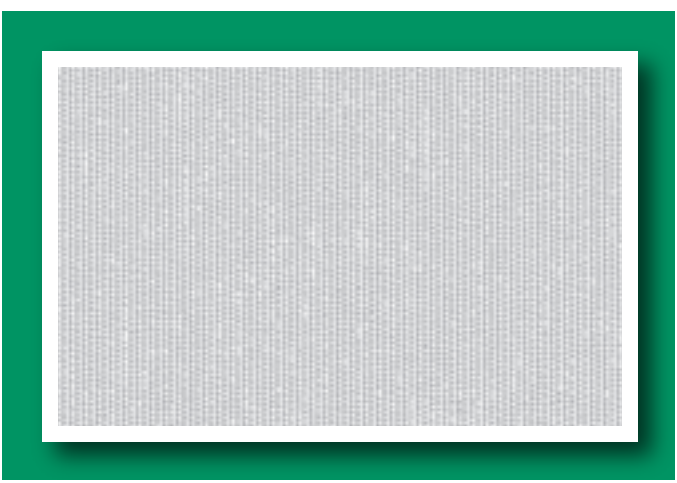
To quantify this theory, the stack-trace count (5,530) was divided into the total samples in the captured profile (348,427), showing that it was responsible for only 1.6 percent of the CPU time. This alone could not explain the higher CPU usage. It was necessary to understand more of the profile.

Browsing more stack traces became an exercise in diminishing returns, as they progressed in order from most to least frequent. The scale of the problem is evident in figure 3, where the entire DTrace output becomes a featureless gray square.

With so much output to study, solving this problem within a reasonable time frame began to feel insurmountable. There had to be a better way.

I created a prototype of a visualization that leveraged the hierarchical nature of stack traces to combine common

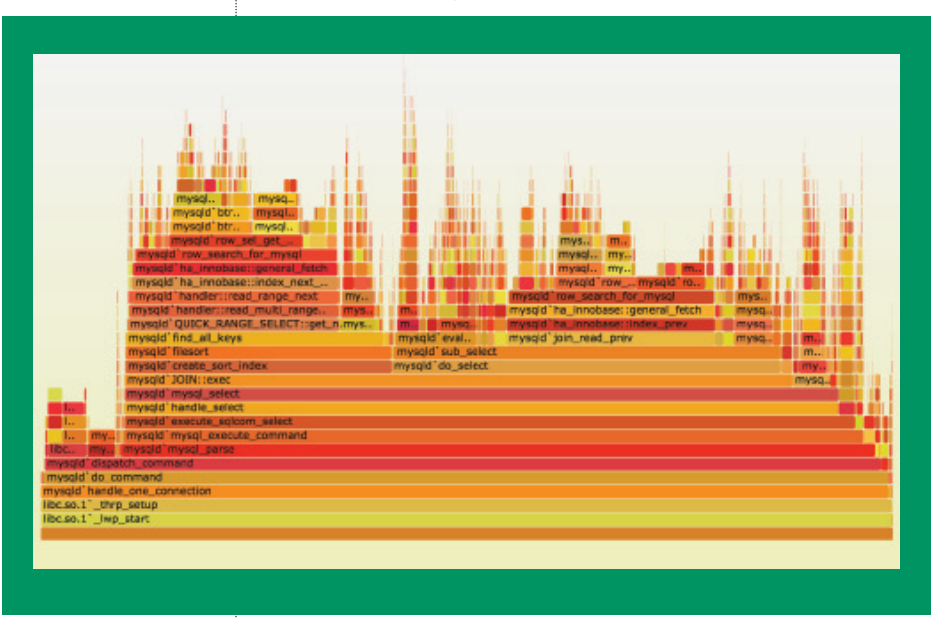
FIGURE 3: **FULL MYSQL DTRACE PROFILE OUTPUT**



paths. The result is shown in figure 4, which visualizes the same output as in figure 3. Since the visualization explained why the CPUs were "hot" (busy), I thought it appropriate to choose a warm palette. With the warm colors and flame-like shapes, these visualizations became known as flame graphs. (An interactive version of figure 4, in SVG [scalable vector graphics] format, is available at http://queue.acm.org/downloads/2016/Gregg4.svg.)

The flame graph allowed the bulk of the profile to be understood very quickly. It showed that the earlier lead, the MySQL `status` command, was responsible for only 3.28 percent of the profile when all stacks were combined. The

FIGURE 4: **FULL MYSQL PROFILER OUTPUT AS A FLAME GRAPH**



bulk of the CPU time was consumed in MySQL **`join`**, which provided a clue to the real problem. The problem was located and fixed, and CPU usage was reduced by 40 percent.

## FLAME GRAPHS EXPLAINED

A flame graph visualizes a collection of stack traces (a.k.a. call stacks), shown as an adjacency diagram with an inverted icicle layout.[7] Flame graphs are commonly used to visualize CPU profiler output, where stack traces are collected using sampling.

A flame graph has the following characteristics:

- A stack trace is represented as a column of boxes, where each box represents a function (a stack frame).
- The y-axis shows the stack depth, ordered from root at the bottom to leaf at the top. The top box shows the function that was on-CPU when the stack trace was collected, and everything beneath that is its ancestry. The function beneath a function is its parent.
- The x-axis spans the stack trace collection. It does *not* show the passage of time, so the left-to-right ordering has no special meaning. The left-to-right ordering of stack traces is alphabetical on the function names, from the root to the leaf of each stack. This maximizes box merging: when identical function boxes are horizontally adjacent, they are merged.
- The width of each function box shows the frequency at which that function was present in the stack traces, or part of a stack trace ancestry. Functions with wide boxes were more frequent in the stack traces than those with narrow

boxes, in proportion to their widths.

- If the box is wide enough, it displays the full function name. If not, either a truncated function name with an ellipsis is shown, or nothing.
- The background color for each box is not significant and is picked at random to be a warm hue. This randomness helps the eye differentiate boxes, especially for adjacent thin "towers." Other color schemes are discussed later.
- The profile visualized may span a single thread, multiple threads, multiple applications, or multiple hosts. Separate flame graphs can be generated if desired, especially for studying individual threads.
- Stack traces may be collected from different profiler targets, and widths can reflect measures other than sample counts. For example, a profiler (or tracer) could measure the time a thread was blocked, along with its stack trace. This can be visualized as a flame graph, where the x-axis spans the total blocked time, and the flame graph shows the blocking code paths.

As the entire profiler output is visualized at once, the end user can navigate intuitively to areas of interest. The shapes and locations in the flame graphs become visual maps for the execution of software.

While flame graphs use interactivity to provide additional features, these characteristics are fulfilled by a static flame graph, which can be shared as an image (e.g., a PNG file or printed on paper). While only wide boxes have enough room to contain the function label text, they are also usually sufficient to show the bulk of the profile.

## INTERACTIVITY

Flame graphs can support interactive features to reveal more detail, improve navigation, and perform calculations.

The original implementation of flame graphs[4] creates an SVG image with embedded JavaScript for interactivity, which is then loaded in a browser. It supports three interactive features: Mouse-over for information, click to zoom, and search.

### Mouse-over for information

On mouse-over of boxes, an informational line below the flame graph and a tooltip display the full function name, the number of samples present in the profile, and the corresponding percentage for those samples in the profile. For example, `Function: mysqld`JOIN::exec` (272,959 samples, 78.34 percent).

This is useful for revealing the function name from unlabeled boxes. The percentage also quantifies code paths in the profile, which helps the user prioritize leads and estimate improvements from proposed changes.

### Click to zoom

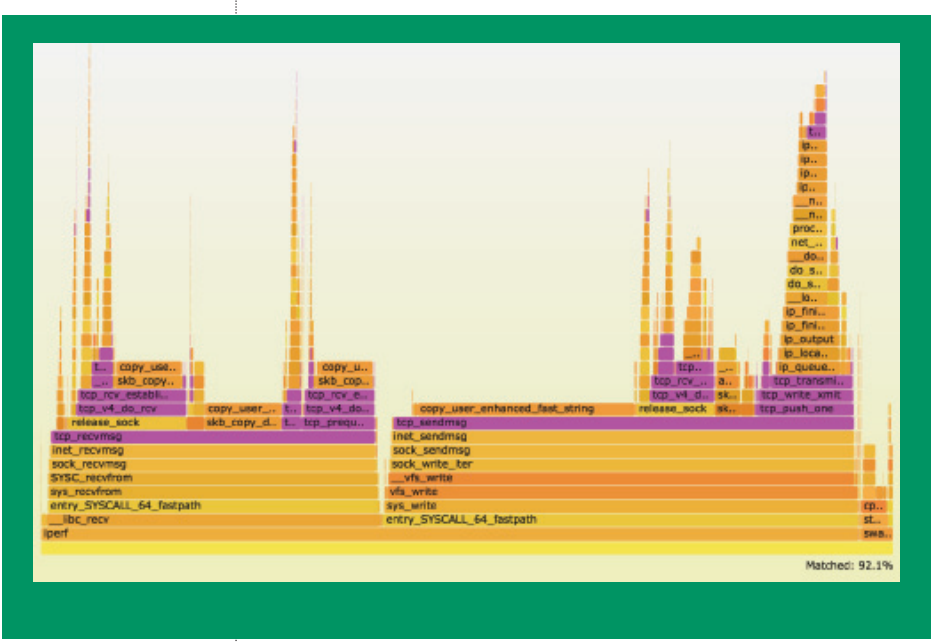
When a box is clicked, the flame graph zooms horizontally. This reveals more detail, often including function names for the child functions. Ancestor frames below the clicked box are shown with a faded background as a visual clue that their widths are now only partially shown. A Reset Zoom button is included to return to the original full profile view. Clicking any box while zoomed will reset the zoom to focus on that new box.

## Search

A search button or keystroke (Ctrl-F) prompts the user for a search term, which can include regular expressions. All function names in the profile are searched, and any matched boxes are highlighted with magenta backgrounds. The sum of matched stack traces is also shown on the flame graph as a percentage of the total profile, as in figure 5. (An interactive version of figure 5 in SVG format is available at http://queue.acm.org/downloads/2016/Gregg5.svg.)

This is useful not just for locating functions, but also for highlighting logical groups of functions—for example,

FIGURE 5: **SEARCH HIGHLIGHTING**



searching for **"^ext4_"** to find the Linux **ext4** functions.

For some flame graphs, many different code paths may end with a function of interest—for example, spin-lock functions. If this appeared in 20 or more locations, calculating their combined contribution to the profile would be a tedious task, involving finding then adding each percentage. The search function makes this trivial, as a combined percentage is calculated and shown on screen.

## INSTRUCTIONS

There are several implementations of flame graphs so far.[5] The original implementation, FlameGraph,[4] was written in the Perl programming language and released as open source. It makes the generation of flame graphs a three-step sequence, including the use of a profiler:

1. Use a profiler to gather stack traces (e.g., Linux perf_events, DTrace, Xperf).
2. Convert the profiler output into the "folded" intermediate format. Various programs are included with the FlameGraph software to handle different profilers; the program names begin with "stackcollapse".
3. Generate the flame graph using flamegraph.pl. This reads the previous folded format and converts it to an SVG flame graph with embedded JavaScript.

The folded stack-trace format puts stack traces on a single line, with functions separated by semicolons, followed by a space and then a count. The name of the application, or the name and process ID separated by a dash, can be optionally included at the start of the folded stack trace,

followed by a semicolon. This groups the application's code paths in the resulting flame graph.

For example, a profile containing the following three stack traces:

```
func_c
func_b
func_a
start_thread

func_d
func_a
start_thread

func_d
func_a
start_thread
```

becomes the following in the folded format:

```
start_thread;func_a;func_b;func_c 1
start_thread;func_a;func_d 2
```

If the application name is included—for example, "java"—it would then become:

```
java;start_thread;func_a;func_b;func_c 1
java;start_thread;func_a;func_d 2
```

This intermediate format has allowed others to contribute converters for other profilers. There are now stackcollapse programs for DTrace, Linux perf_events, FreeBSD pmcstat, Xperf, SystemTap, Xcode Instruments, Intel VTune, Lightweight Java Profiler, Java jstack, and gdb.[4]

The final flamegraph.pl program supports many options for customization, including changing the title of the flame graph.

As an example, the following steps fetch the FlameGraph software, gather a profile on Linux (99 Hz, all CPUs, 60 seconds), and then generate a flame graph from the profile:

```
# git clone https://github.com/brendangregg/
FlameGraph
# cd FlameGraph
# perf record -F 99 -a -g -- sleep 60
# perf script | ./stackcollapse-perf.pl | ./
flamegraph.pl > out.svg
```

Since the output of stackcollapse has single lines per record, it can be modified using grep/sed/awk if needed before generating a flame graph.

The online flame graph documentation includes instructions for using other profilers.[4,5]

## FLAME GRAPH INTERPRETATION

Flame graphs can be interpreted as follows:

- The top edge of the flame graph shows the function that was running on the CPU when the stack trace was collected.

For CPU profiles, this is the function that is directly consuming CPU cycles. For other profile types, this is the function that led directly to the instrumented event.

- Look for large plateaus along the top edge, as these show a single stack trace was frequently present in the profile. For CPU profiles, this means a single function was frequently running on-CPU.
- Reading top down shows ancestry. A function was called by its parent, which is shown directly below it; the parent was called by its parent shown below it, and so on. A quick scan downward from a function identifies why it was called.
- Reading bottom up shows code flow and the bigger picture. A function calls any child functions shown above it, which, in turn, call functions shown above them. Reading bottom up also shows the big picture of code flow before various forks split execution into smaller towers.
- The widths of function boxes can be directly compared: wider boxes mean a greater presence in the profile and are the most important to understand first.
- For CPU profiles that employ timed sampling of stack traces, if a function box is wider than another, this may be because it consumes more CPU per function call or because the function was simply called more often. The function-call count is not shown or known via sampling.
- Major forks in the flame graph, spotted as two or more large towers atop a single function, can be useful to study. They can indicate a logical grouping of code, where a function processes work in stages, each with its own function.

They can also be caused by a conditional statement, which chooses which function to call.

INTERPRETATION EXAMPLE

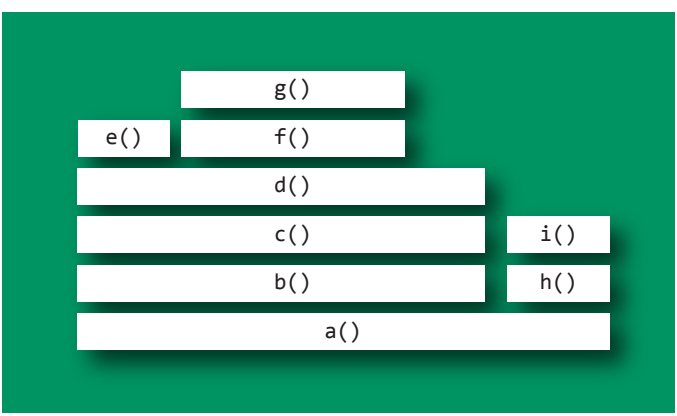
As an example of interpreting a flame graph, consider the mock one shown in figure 6. Imagine this is visualizing a CPU profile, collected using timed samples of stack traces (as is typical).

The top edge shows that function `g()` is on-CPU the most; `d()` is wider, but its exposed top edge is on-CPU the least. Functions including `b()` and `c()` do not appear to have been sampled on-CPU directly; rather, their child functions were running.

Functions beneath `g()` show its ancestry: `g()` was called by `f()`, which was called by `d()`, and so on.

Visually comparing the widths of functions `b()` and `h()` shows that the `b()` code path was on-CPU about four times

FIGURE 6: **EXAMPLE FOR INTERPRETATION**



more than **h()**. The actual functions on-CPU in each case were their children.

A major fork in the code paths is visible where **a()** calls **b()** and **h()**. Understanding why the code does this may be a major clue to its logical organization. This may be the result of a conditional (if conditional, call **b()**, else call **h()**) or a logical grouping of stages (where **a()** is processed in two parts: **b()** and **h()**).

## OTHER CODE-PATH VISUALIZATIONS

As was shown in figure 1, Linux perf_events prints a tree of code paths with percentage annotations. This is another type of hierarchy visualization: an indented tree layout.[7] Depending on the profile, this can sometimes sufficiently summarize the output, but not always. Unlike flame graphs, one cannot zoom out to see the entire profile and still make sense of this text-based visualization, especially after the percentages can no longer be read.

KCacheGrind[14] visualizes code paths from profile data using a directed acyclic graph. This involves representing functions as labeled boxes (where the width is scaled to fit the function name), parent-to-child relationships as arrows, and then profile data is annotated on the boxes and arrows as percentages with bar chart-like icons. Similar to the problem with perf_events, if the visualization is zoomed out to fit a complex profile, then the annotations may no longer be legible.

The sunburst layout is equivalent to the icicle layout used by flame graphs, but it uses polar coordinates.[7] While this

can generate interesting shapes, there are some difficulties: function names are harder to draw and read from sunburst slices than they are in the rectangular flame-graph boxes. Also, comparing two functions becomes a matter of comparing two angles rather than two line lengths, which has been evaluated as a more difficult perceptual task.[10]

Flame charts are a similar code-path visualization to flame graphs (and were inspired by flame graphs[13]). On the x-axis, however, they show the passage of time instead of an alphabetical sort. This has its advantages: time-ordered issues can be identified. It can greatly reduce merging, however, a problem exacerbated when profiling multiple threads. It could be a useful option for understanding time order sequences when used with flame graphs for the bigger picture.

## CHALLENGES

Challenges with flame graphs mostly involve system profilers and not flame graphs themselves. There are two typical problems with profilers:

- **Stack traces are incomplete.** Some system profilers truncate to a fixed stack depth (e.g., 10 frames), which must be increased to capture the full stack traces, or else frame merging can fail. A worse problem is when the software compiler reuses the frame pointer register as a compiler optimization, breaking the typical method of stack-trace collection. The fix requires either a different compiled binary (e.g., using gcc's -fno-omit-frame-pointer) or a different stack-walking technique.
- **Function names are missing.** In this case, the stack trace

is complete, but many function names are missing and may be represented as hexadecimal addresses. This commonly happens with JIT (just-in-time) compiled code, which may not create a standard symbol table for profilers. Depending on the profiler and runtime, there are different fixes. For example, Linux perf_events supports supplemental symbol files, which the application can create.

At Netflix we encountered both problems when attempting to create flame graphs for Java.[6] The first has been fixed by the addition of a JVM (Java Virtual Machine) option—XX:+PreserveFramePointer, which allows Linux perf_events to capture full stack traces. The second has been fixed using a Java agent, perf-map-agent,[11] which creates a symbol table for Java methods.

One challenge with the Perl flame-graph implementation has been the resulting SVG file size. For a large profile with many thousands of unique code paths, the SVG file can be tens of megabytes in size, which becomes sluggish to load in a browser. The fix has been to elide code paths that are so thin they are normally invisible in the flame graph. This does not affect the big-picture view and has kept the SVG file smaller.

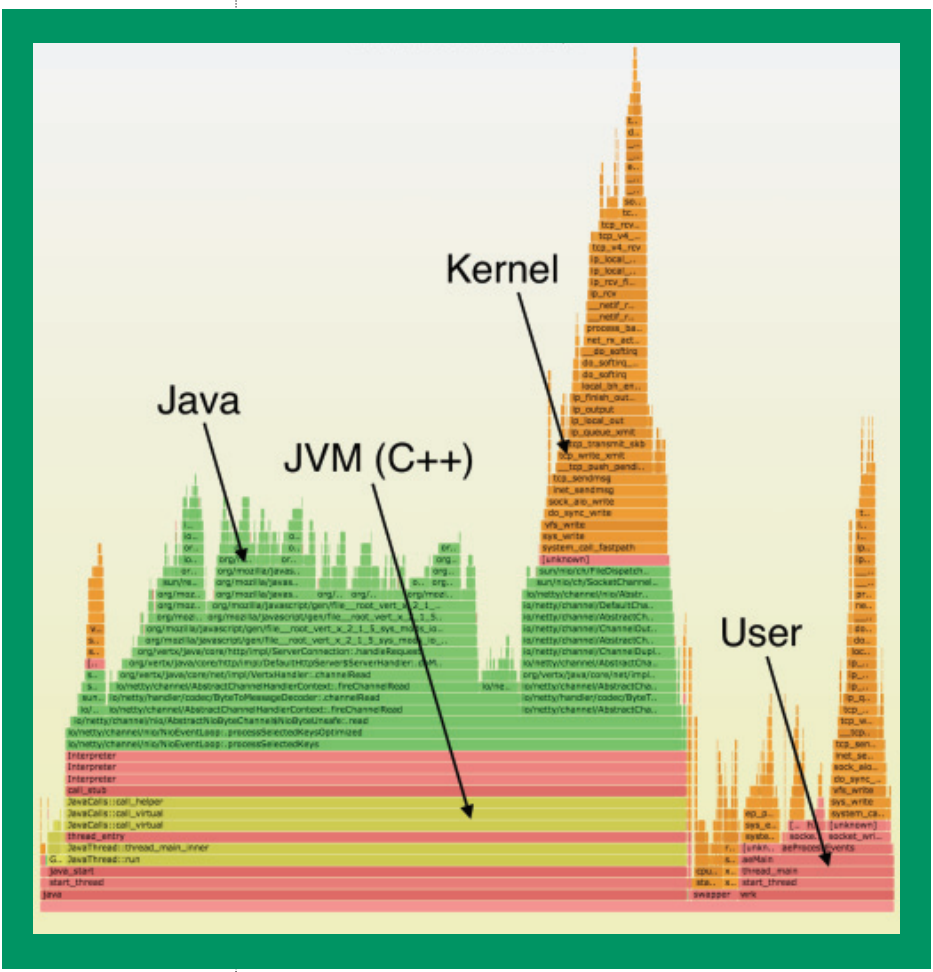
## OTHER COLOR SCHEMES

Apart from a random warm palette, other flame-graph color schemes can be used, such as for differentiating code or including an extra dimension of data.

Various palettes can be selected in the Perl flame-graph version, including "java," which uses different hues to highlight a Java mixed-mode flame graph: green for

Java methods, yellow for C++, red for all other user-mode functions, and orange for kernel-mode functions. An example is shown in figure 7. (An interactive version of figure 7 in SVG

FIGURE 7: **JAVA MIXED-MODE CPU FLAME GRAPH**



format is available at http://queue.acm.org/downloads/2016/Gregg7.svg.)

Another option is a hashing color scheme, which picks a color based on a hash of the function name. This keeps colors consistent, which is helpful when comparing multiple flame graphs from the same system.

Color can also be used for differential flame graphs, described in the next section.

## DIFFERENTIAL FLAME GRAPHS

A differential flame graph shows the difference between two profiles, A and B. The Perl flame-graph software currently supports one method, where the B profile is displayed and then colored using the delta from A to B. Red shades indicate functions that increased, and blue shades indicate those that decreased. A problem with this approach is that some code paths present in the A profile may be missing entirely in the B profile, and so will be missing from the final visualization. This could be misleading.

Another implementation, flamegraphdiff,[2] solves this problem by using three flame graphs. The first shows the A profile, the second shows the B profile, and the third shows only the delta between them. A mouse-over of one function in any flame graph also highlights the others to help navigation. Optionally, the flame graphs can also be colored using a red/blue scheme to indicate which code paths increased or decreased.

## OTHER TARGETS

As previously mentioned, flame graphs can visualize any profiler output. This includes stack traces collected on CPU PMC (performance monitoring counter) overflow events, static tracing events, and dynamic tracing events. Following are some specific examples.

### Stall Cycles

A stall-cycle flame graph shows code paths that commonly block on processor or hardware resources—typically memory I/O. The input stack traces can be collected using a PMC profiler, such as Linux perf_events. This can direct the developer to employ a different optimization technique for the identified code paths, one that aims to reduce memory I/O rather than reducing instructions.

### CPI

CPI (cycles per instruction), or its inverse, IPC (instructions per cycle), is a measure that also helps explain the types of CPU cycles and can direct tuning effort. A CPI flame graph shows a CPU sample flame graph where widths correspond to CPU cycles, but it uses a color scale from red to blue to indicate each function's CPI: red for a high CPI and blue for a low CPI. This can be accomplished by capturing two profiles—a CPU sample profile and an instruction count profile—and then using a differential flame graph to color the difference between them.

### Memory

Flame graphs can shed light on memory growth by visualizing a number of different memory events.

A `malloc()` flame graph, created by tracing the `malloc()` function, visualizes code paths that allocated memory. This can be difficult in practice, as allocator functions can be called frequently, making the cost to trace them prohibitive in some scenarios.

Tracing the `brk()` and `mmap()` syscalls can show code paths that caused an expansion in virtual memory for a process, typically related to the allocation path, although this could also be an asynchronous expansion of the application's memory. These are typically lower frequency, making them more suitable for tracing.

Tracing memory page faults shows code paths that caused an expansion in physical memory for a process. Unlike allocator code paths, this shows the code that populated the allocated memory. Page faults are also typically a lower-frequency activity.

### I/O

The issuing of I/O, such as file system, storage device, and network, can usually be traced using system tracers. A flame graph of these profiles illustrates different application paths that synchronously issued I/O.

In practice, this has revealed types of I/O that were otherwise not known. For example, disk I/O may be issued: synchronously by the application, by a file system read-ahead routine, by an asynchronous flush of dirty data, or by

a kernel background scrub of disk blocks. An I/O flame graph identifies each of these types by illustrating the code paths that led to issuing disk I/O.

### Off-CPU

Many performance issues are not visible using CPU flame graphs, as they involve time spent while the threads are blocked, not running on a CPU (off-CPU). Reasons for a thread to block include waiting on I/O, locks, timers, a turn on-CPU, and waiting for paging or swapping. These scenarios can be identified by the stack trace when the thread was descheduled. The time spent off-CPU can also be measured by tracing the time from when a thread left the CPU to when it returned. System profilers commonly use static trace points in the kernel to trace these events.

An off-CPU time flame graph can illustrate this off-CPU time by showing the blocked stack traces, where the width of a box is proportional to the time spent blocked.

### Wakeups

A problem found in practice with off-CPU time flame graphs is that they are inconclusive when a thread blocks waiting for a signal from another thread. We needed information on why the other thread took so long.

A wakeup time flame graph can be generated by tracing thread wakeup events. This includes stack traces from the waker threads, and so they shed light on why they were blocked. This flame-graph type can be studied along with an off-CPU time flame graph for more information on blocked threads.

### Chain Graphs

One wakeup flame graph may not be enough. The thread that woke up a blocked thread may itself have been blocked by another thread. In practice, one thread may have been blocked on a second, which was blocked on a third, and a fourth.

A chain flame graph is an experimental visualization[3] that begins with an off-CPU flame graph and then adds all wakeup stack traces to the top of each blocked stack. By reading bottom-up, you see the blocked off-CPU stack trace, and then the first stack trace that woke it, then the next stack trace that woke it, and so on. Widths correspond to the time that threads were off-CPU and the time taken for wakeups.

This can be accomplished by tracing all off-CPU and wakeup events with time stamps and stack traces, and post-processing. These events can be extremely frequent, however, and impractical to instrument in production using current tools.

## FUTURE WORK

Much of the work related to flame graphs has involved getting different profilers to work with different runtimes so that the input for flame graphs can be captured correctly (e.g., for Node.js, Ruby, Perl, Lua, Erlang, Python, Java, golang, and with DTrace, perf_events, pmcstat, Xperf, Instruments, etc.). There is likely to be more of this type of work in the future.

Another in-progress differential flame graph, called a

white/black differential, uses the single flame-graph scheme described earlier plus an extra region on the right to show only the missing code paths. Differential flame graphs (of any type) should also see more adoption in the future; at Netflix, we are working to have these generated nightly for microservices to identify regressions and aid with performance-issue analysis.

Several other flame-graph implementations are in development, exploring different features. Netflix has been developing d3-flame-graph,[12] which includes transitions when zooming. The hope is that this can provide new interactivity features, including a way to toggle the merge order from bottom-up to top-down, and also to merge around a given function. Changing the merge order has already proven useful for the original flamegraph.pl, which can optionally merge top-down and then show this as an icicle plot. A top-down merge groups together leaf paths, such as spin locks.

## CONCLUSION

The flame graph is an effective visualization for collected stack traces and is suitable for CPU profiling, as well as many other profile types. It creates a visual map for the execution of software and allows the user to navigate to areas of interest. Unlike other code-path visualizations, flame graphs convey information intuitively using line lengths and can handle large-scale profiles, while usually remaining readable on one screen. The flame graph has become an essential tool for understanding profiles quickly and has been instrumental in countless performance wins.

## Acknowledgments

Inspiration for the general layout, SVG output, and JavaScript interactivity came from Neelakanth Nadgir's function_call_graph.rb time-ordered visualization for callstacks,[9] which itself was inspired by Roch Bourbonnais's CallStackAnalyzer and Jan Boerhout's vftrace. Adrien Mahieux developed the horizontal zoom feature for flame graphs, and Thorsten Lorenz added a search feature to his implementation.[8] Cor-Paul Bezemer researched differential flame graphs and developed the first solution.[1] Off-CPU time flame graphs were first discussed and documented by Yichun Zhang.[15]

Thanks to the many others who have documented case studies, contributed ideas and code, given talks, created new implementations, and fixed profilers to make this possible. See the updates section for a list of this work.[5] Finally, thanks to Deirdré Straughan for editing and feedback.

## References

1. Bezemer, C.-P. Flamegraphdiff. GitHub; http://corpaul.github.io/flamegraphdiff/.
2. Bezemer, C.-P., Pouwelse, J., Gregg, B. 2015. Understanding software performance regressions using differential flame graphs. Published in *Software Analysis, Evolution and Reengineering* (SANER), 2015 IEEE 22nd International Conference; http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=7081872&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D7081872.

3. Gregg, B. 2013. Blazing performance with flame graphs. 27th Large Installation System Administration Conference; https://www.usenix.org/conference/lisa13/technical-sessions/plenary/gregg.
4. Gregg, B. FlameGraph. GitHub; https://github.com/brendangregg/FlameGraph.
5. Gregg, B. Flame graphs; http://www.brendangregg.com/flamegraphs.html.
6. Gregg, B., Spier, M. 2015. Java in flames. The Netflix Tech Blog; http://techblog.netflix.com/2015/07/java-in-flames.html.
7. Heer, J., Bostock, M., Ogievetsky, V. 2010. A tour through the visualization zoo. *acmqueue* 8(5); http://queue.acm.org/detail.cfm?id=1805128.
8. Lorenz, T. Flamegraph. GitHub; https://github.com/thlorenz/flamegraph.
9. Nadgir, N. 2007. Visualizing callgraphs via dtrace and ruby. Oracle Blogs; https://blogs.oracle.com/realneel/entry/visualizing_callstacks_via_dtrace_and.
10. Odds, G. 2013. The science behind data visualisation. Creative Bloq; http://www.creativebloq.com/design/science-behind-data-visualisation-8135496.
11. Rudolph, J. perf-map-agent. GitHub; https://github.com/jrudolph/perf-map-agent.
12. Spier, M. 2015. d3-flame-graph. GitHub; https://github.com/spiermar/d3-flame-graph.
13. Tikhonovsky, I. 2013. Web Inspector: implement flame chart for CPU profiler. Webkit Bugzilla; https://bugs.webkit.org/show_bug.cgi?id=111162.

14. Weidendorfer, J. KCachegrind; https://kcachegrind.github.io/html/Home.html.
15. Zhang, Y. 2013. Introduction to off-CPU time flame graphs; http://agentzh.org/misc/slides/off-cpu-flame-graphs.pdf.

**LOVE IT, HATE IT? LET US KNOW** feedback@queue.acm.org

Brendan Gregg *is a senior performance architect at Netflix, where he does large-scale computer performance design, analysis, and tuning. He is the author of multiple technical books including Systems Performance (Prentice Hall, 2013). He received the Usenix LISA Award for Outstanding Achievement in System Administration. He was previously a performance lead and kernel engineer at Sun Microsystems, where he developed the ZFS L2ARC and worked on performance. He has also created numerous performance-analysis tools, which have been included in multiple operating systems. His recent work includes developing methodologies and visualizations for performance analysis.*