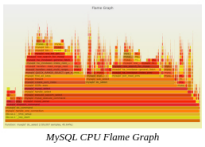
# CPU Flame Graphs



*MySQL CPU Flame Graph*

Determining why CPUs are busy is a routine task for performance analysis, which often involves profiling stack traces. Profiling by sampling at a fixed rate is a coarse but effective way to see which code-paths are hot (busy on-CPU). It usually works by creating a timed interrupt that collects the current program counter, function address, or entire stack back trace, and translates these to something human readable when printing a summary report.

Profiling data can be thousands of lines long, and difficult to comprehend. *Flame graphs* are a visualization for sampled stack traces, which allows hot code-paths to be identified quickly. See the Flame Graphs main page for uses of this visualization other than CPU profiling.

Flame Graphs can work with any CPU profiler on any operating system. My examples here use Linux perf (perf_events), DTrace, SystemTap, and ktap. See the Updates list for other profiler examples, and github for the flame graph software.

On this page I'll introduce and explain CPU flame graphs, list generic instructions for their creation, then discuss generation for specific languages. A full table of contents:



## 1. Problem

Here I'm using Linux perf (aka perf_events) to profile a bash program that is consuming CPU:

```
# perf record -F 99 -p 13204 -g -- sleep 30
# perf report -n --stdio
# ========
# captured on: Tue Dec  9 03:54:11 2014
# hostname : bgregg-test
# os release : 3.13.11.6
[...]
# 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
                    |          |          |          execute_command_internal
                    |          |          |          execute_command
                    |          |          |          execute_while_or_until
                    |          |          |          execute_while_command
                    |          |          |          execute_command_internal
                    |          |          |          execute_command
                    |          |          |          reader_loop
                    |          |          |          main
                    |          |          |          __libc_start_main
                    |          |          |
                    |          |           --34.74%-- do_redirections
                    |          |                      |
                    |          |                      |--54.55%-- execute_builtin_or_function
                    |          |                      |          execute_simple_command
                    |          |                      |          execute_command_internal
                    |          |                      |          execute_command
                    |          |                      |          execute_while_or_until
                    |          |                      |          execute_while_command
                    |          |                      |          execute_command_internal
                    |          |                      |          execute_command
                    |          |                      |          reader_loop
                    |          |                      |          main
                    |          |                      |          __libc_start_main
                    |          |                      |
[...]
```

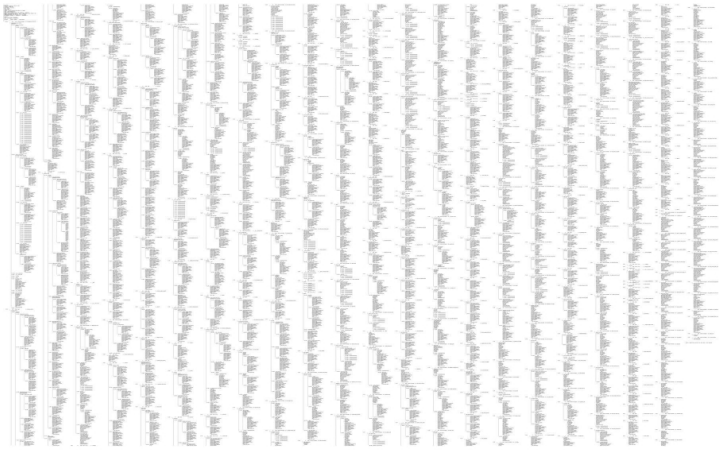
The `perf record` command sampled at 99 Hertz (-F 99), on our target PID (-p 13204), and captured stack traces (-g --) for call graph info.

The `perf report` command does a good job of summarizing the hundreds of stack trace samples as text. Similar code paths are coalesced, and the summary is shown as a tree graph, with percentages on each leaf. Read paths from top left to bottom right, which follows a code path's ancestry (and its stack trace sample). The percentages must be multiplied to determine a full stack trace's absolute frequency.

The first stack trace shown (which includes do_redirection_internal()), accounts for only 2% of the samples. The next stack trace (including execute_builtin_or_function()), 1%. So after reading this screen of text, we can only account for 3% of the samples. In order to understand where the bulk of the CPU time is spent, we'll want an idea of the code path for over 50% of the samples. We might need to do a lot more reading.

### 1.1. Too Much Data

The above output has been truncated, and only shows 45 lines from over 8,000 lines of output. The full output, visualized, looks like this:

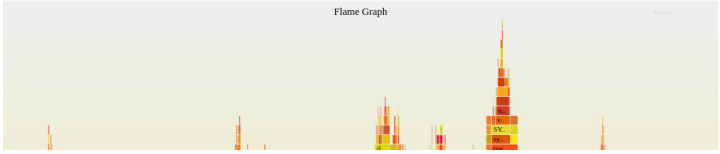


Can you see the earlier two stacks? They are in the top left. (Other versions: text, larger JPG.)

Sometimes, the bulk of the CPU time is in a single code path, and `perf report` summarizes this easily on a single screen. However, often you need to read many screen fulls of text to understand the profile, which is time consuming and tedious.

## 2. The Flame Graph

Now the same data show previously as a flame graph (click to zoom):

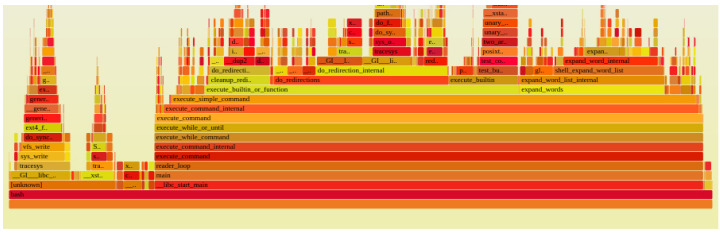





Recent posts:

- 22 Jul 2024 » No More Blue Fridays
- 24 Mar 2024 » Linux Crisis Tools
- 17 Mar 2024 » The Return of the Frame Pointers
- 10 Mar 2024 » eBPF Documentary
- 28 Apr 2023 » eBPF Observability Tools Are Not Security Tools
- 01 Mar 2023 » USENIX SREcon APAC 2022: Computing Performance: What's on the Horizon
- 17 Feb 2023 » USENIX SREcon APAC 2023: CFP
- 02 May 2022 » Brendan@Intel.com
- 15 Apr 2022 » Netflix End of Series 1
- 09 Apr 2022 » TensorFlow Library Performance
- 19 Mar 2022 » Why Don't You Use ...
- 26 Sep 2021 » The Speed of Time
- 06 Sep 2021 » ZFS Is Mysteriously Eating My CPU
- 30 Aug 2021 » Analyzing a High Rate of Paging
- 27 Aug 2021 » Slack's Secret STDERR Messages
- 05 Jul 2021 » USENIX LISA2021 Computing Performance: On the Horizon
- 03 Jul 2021 » How To Add eBPF Observability To Your Product
- 15 Jun 2021 » USENIX LISA2021 BPF Internals (eBPF)
- 04 Jun 2021 » An Unbelievable Demo
- 29 May 2021 » Moving my US tech job to Australia

Blog index
About
RSS



If you have troubles in your browser, try the direct SVG or PNG version.

With the flame graph, all the data is on screen at once, and the hottest code-paths are immediately obvious as the widest functions.

## 3. Description

I'll explain this carefully: it may look similar to other visualizations from profilers, but it is different.

- Each box represents a function in the stack (a "stack frame").
- The **y-axis** shows stack depth (number of frames on the stack). The top box shows the function that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent, just like the stack traces shown earlier. (Some flame graph implementations prefer to invert the order and use an "icicle layout", so flames look upside down.)
- The **x-axis** spans the sample population. It does *not* show the passing of time from left to right, as most graphs do. The left to right ordering has no meaning (it's sorted alphabetically to maximize frame merging).
- The width of the box shows the *total* time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Functions with wide boxes may consume more CPU per execution than those with narrow boxes, or, they may simply be called more often. The call count is not shown (or known via sampling).
- The sample count can exceed elapsed time if multiple threads were running and sampled concurrently.

The colors aren't significant, and are usually picked at random to be warm colors (other meaningful palettes are supported). This visualization was called a "flame graph" as it was first used to show what is hot on-CPU, and, it looked like flames. It is also interactive: mouse over the SVGs to reveal details, and click to zoom.

## 4. Instructions

The code to the FlameGraph tool and instructions are on github. It's a simple Perl program that outputs SVG. They are generated in three steps:

1. Capture stacks
2. Fold stacks
3. flamegraph.pl

The first step is to use the profiler of your choice. See below for some examples using perf, DTrace, SystemTap, and ktap.

The second step generates a line-based output for flamegraph.pl to read, which can also be grep'd to filter for functions of interest. There are a collection of simple Perl programs to do this, named stackcollapse*.pl, to process the output from different profilers. Newer versions of Linux perf and eBPF profile(8) can do steps (1) and (2) at the same time (explained in the following sections).

### 4.1. Linux perf

Linux perf has a variety of capabilities, including CPU sampling. Using it to sample all CPUs and generate a flame graph (this is the old-fashioned way to do it, that works on Linux 2.6.X onwards):

```
# git clone https://github.com/brendangregg/FlameGraph  # or download it from github
# cd FlameGraph
# perf record -F 99 -a -g -- sleep 60
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# ./flamegraph.pl out.perf-folded > perf.svg
# firefox perf.svg  # or chrome, etc.
```

The `perf record` command samples at 99 Hertz (-F 99) across all CPUs (-a), capturing stack traces so that a call graph (-g) of function ancestry can be generated later. The samples are saved in a perf.data file, which are read by `perf script`. This does involve CPU, file system, and disk overheads to save the samples to the file system for later processing. On newer kernels (Linux 4.8+) check out the next section on eBPF profile(8) for a lower-overhead approach.

In the steps above I create the intermediate file, out.perf-folded, to make it a little quicker when creating multiple filtered flame graphs from the same data. E.g.:

```
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# grep -v cpu_idle out.perf-folded | ./flamegraph.pl > nonidle.svg
# grep ext4 out.perf-folded | ./flamegraph.pl > ext4internals.svg
# egrep 'system_call.*sys_(read|write)' out.perf-folded | ./flamegraph.pl > rw.svg
```

Since Linux 4.5, the `perf report` command has included a "folded" mode so that it can emit folded output directly, without using stackcollapse-perf.pl. See my blog post Linux 4.5 perf folded format. Last I tried it needed a touch of awk(1) to put the process name as the bottom frame:

```
# perf report --stdio --no-children -n -g folded,0,caller,count -s comm | \
    awk '/^ / { comm = $3 } /^[0-9]/ { print comm ";" $2, $1 }'
bash;0x436fd 10282
bash;make_child;__libc_fork;return_from_SYSCALL_64;do_syscall_64;sys_clone;_do_fork;copy_process.part.30;copy_page_range 6378
bash;__execve;return_from_SYSCALL_64;do_syscall_6
[...]
```

XXX In more recent Linux (5.something) a flame graph generator was added to perf (I think) so that it can do it all without fetching my Perl code. I was involved earlier on but I lost track of its status. I'll update this when I have a chance.

Note that Netflix still uses perf for per-sample collection for FlameScope, and eBPF for lower-overhead flame graphs.

For more details on perf, see my perf_events Flame Graphs section.

### 4.2. eBPF profile

My profile(8) tool in bcc does in-kernel aggregations of sampled stack traces, so that only the summary is emitted to user space. profile(8) can even print that summary in folded format directly. This can be much more efficient than using perf(1). Here are the commands for Ubuntu Linux (change the package install command and package name to suit your distro; it's bcc that we're installing here):

```
# git clone https://github.com/brendangregg/FlameGraph  # or download it from github
# apt-get install bpfcc-tools  # might be called bcc-tools
# cd FlameGraph
# profile-bpfcc -F 99 -adf 60 > out.profile-folded  # might be called /usr/share/bcc/tools/profile
# ./flamegraph.pl --colors=java out.profile-folded > profile.svg
# firefox profile.svg  # or chrome, etc.
```

This samples at 99 Hertz (-F 99), includes annotations such as "_[k]" for kernel (-a), includes a delimiter between user and kernel stacks (-d), outputs in folded format (-f), and samples for 60 seconds (60). Run `profile-bpfcc -h` to list other options. I added the --colors=java mode to flamegraph.pl so it made use of those annotations, which will color user/kernel/jit code differently.

Of course, you can pipe the output of profile(8) to flamegraph.pl directly to skip the intermediate file.

### 4.3. DTrace

DTrace can be used to profile on-CPU stack traces on systems that support it (Solaris, BSDs). The following example uses DTrace to sample user-level stacks at 99 Hertz for processes named "mysqld", and then generates the flame graph (see the MySQL example later on for example results):

```
# git clone https://github.com/brendangregg/FlameGraph  # or download it from github
# cd FlameGraph
# dtrace -x ustackframes=100 -n 'profile-99 /execname == "mysqld" && arg1/ {
    @[ustack()] = count(); } tick-60s { exit(0); }' -o out.stacks
# ./stackcollapse.pl out.stacks > out.folded
# ./flamegraph.pl out.folded > out.svg
```

The out.folded intermediate file isn't necessary, as stackcollapse.pl could be piped directly to flamegraph.pl. However, I find it handy in case I'd like to edit the profile data a little using vi. For example, when sampling the kernel, to find and delete the idle threads.

To explain the "arg1" check: arg1 is the user-land program counter, so this checks that it is non-zero; arg0 is the kernel. The following is an example DTrace command to sample the kernel:

```
# dtrace -x stackframes=100 -n 'profile-199 /arg0/ {
```

```
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks
```

This time, all threads are sampled, so the output will contain many idle thread samples (you can either use DTrace to filter them, or grep/vi to filter the folded output). The rate is also increased to 199 Hertz, as capturing kernel stacks is much less expensive than user-level stacks. The odd numbered rates, 99 and 199, are used to avoid sampling in lockstep with other activity and producing misleading results.

### 4.4. SystemTap

SystemTap can also sample stack traces via the timer.profile probe, which fires at the system clock rate (CONFIG_HZ). Unlike perf, which dumps samples to a file for later aggregation and reporting, SystemTap can do the aggregation in-kernel and pass a (much smaller) report to user-land. The data collected and output generated can be customized much further via its scripting language.

Using SystemTap v1.7 on Fedora 16 to generate a flame graph:

```
# stap -s 32 -D MAXBACKTRACE=100 -D MAXSTRINGLEN=4096 -D MAXMAPENTRIES=10240 \
    -D MAXACTION=10000 -D STP_OVERLOAD_THRESHOLD=500000000 --all-modules \
    -ve 'global s; probe timer.profile { s[backtrace()] <<< 1; }
    probe end { foreach (i in s+) { print_stack(i);
    printf("\t%d\n", @count(s[i])); } } probe timer.s(60) { exit(); }' \
    > out.stap-stacks
# ./stackcollapse-stap.pl out.stap-stacks > out.stap-folded
# cat out.stap-folded | ./flamegraph.pl > stap-kernel.svg
```

The six options used (-s 32, -D ...) increase various SystemTap limits. The only ones really necessary for flame graphs are "-D MAXBACKTRACE=100 -D MAXSTRINGLEN=4096", so that stack traces aren't truncated; the others became necessary when sampling for long periods (in this case, 60 seconds) on busy workloads, to avoid various threshold and overflow errors.

The timer.profile probe was used, which samples all CPUs at 100 Hertz: a little coarse, and risking lockstep sampling. Last I checked, the timer.hz(997) probe fired at the correct rate, but couldn't read stack backtraces.

### 4.4. Other Profilers

See the Updates links on the Flame Graphs main page for other profilers. These include Dave Pacheco's flame graphs for node.js functions, Mark Probst's Flame Graphs for Instruments on OS X, and Bruce Dawson's Summarizing Xperf CPU Usage with Flame Graphs on Microsoft Windows.

## 5. Examples

The next example profiles MySQL using DTrace, followed by two CPU flame graphs of the Linux kernel using perf_events. These are old examples, that don't support click-to-zoom (you can still mouse-over for details).

The Linux examples were generated on 3.2.9 (Fedora 16 guest) running under KVM (Ubuntu host), with one virtual CPU. Some code paths and sample ratios will be very different on bare-metal: networking won't be processed via the virtio-net driver, for example.

### 5.1. MySQL

This is the original performance issue that led me to create flame graphs. It was a production MySQL database which was consuming more CPU than hoped. The profiler available was DTrace, and I used it to frequency count on-CPU user-level stacks:

```
# 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
```

The last two most frequent stacks are shown here. The last was sampled on-CPU 5,530 times, and looks like it is MySQL doing some system status housekeeping. If that's the hottest, and we know we have a CPU issue, perhaps I should go hunting for tunables to disable system stats.

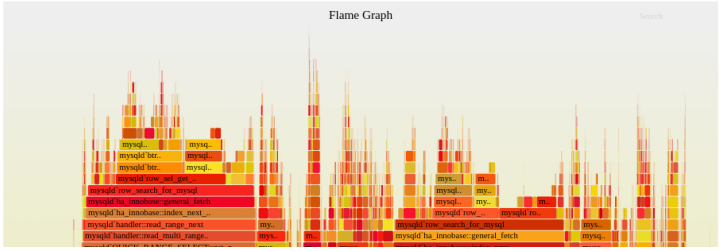
The problem was that most of the output was truncated from this screenshot ("[...]"), and (unlike Linux perf), DTrace doesn't print percentages, so you aren't sure how much these stacks really matter. I manually calculated percentages based on the total sample count (which was 348,427), and found, to my dismay, that these two stacks only represent less than 3% of the CPU samples.

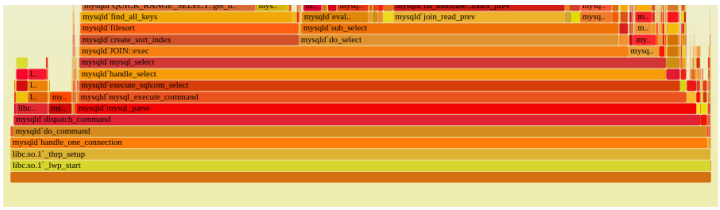
The full output was 591,622 lines long, and included 27,053 stacks. It looks like this:



Click for a larger image (7 Mbyte JPG, although you still can't read the text). DTrace doesn't summarize as well as Linux perf_events (as perf_events can coalesce partial stacks), although I'm not sure that would have helped much here. Production workloads can involve many code paths.

The same MySQL profile data, rendered as a flame graph:



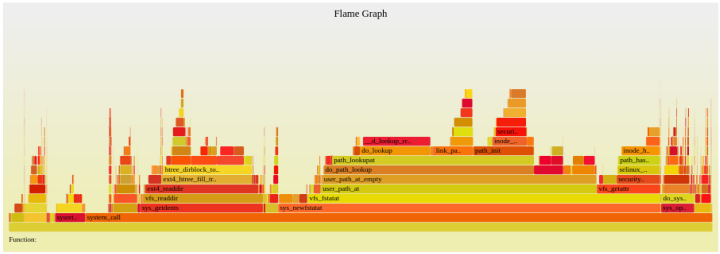



You can mouse over elements to see percentages (but not click-to-zoom, as this is an old version), showing how frequent the element was present in the profiling data. You can also view the SVG separately, or the PNG version.

The earlier truncated text output identified a MySQL status stack as the hottest. The flame graph shows reality: most of the time is really in JOIN::exec. This pointed the investigation in the right direction: JOIN::exec, and the functions above it, and led to the issue being solved.

## 5.2. File Systems

As an example of a different workload, this shows the Linux kernel CPU time while an ext4 file system was being archived (SVG, PNG):

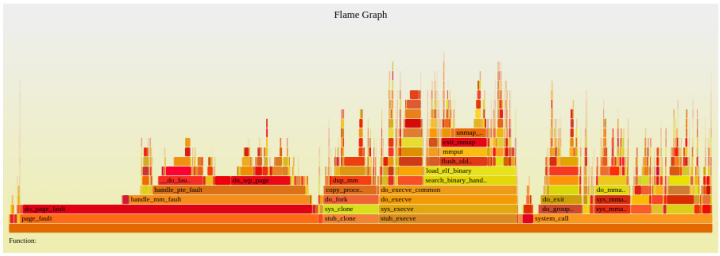



This shows how the file system was being read and where kernel CPU time was spent. Most of the kernel time is in sys_newfstatat() and sys_getdents(): metadata work as the file system is walked. sys_openat() is on the right, as files are opened to be read, which are then mmap()d (look to the right of sys_getdents(), these are in alphabetical order), and finally page faulted into user-space (see the page_fault() mountain on the left).

The actual work of moving bytes is then spent in user-land on the mmap'd segments (and not shown in this kernel flame graph). Had the archiver used the read() syscall instead, this flame graph would look very different, and have a large sys_read() component.

## 5.3. Short Lived Processes

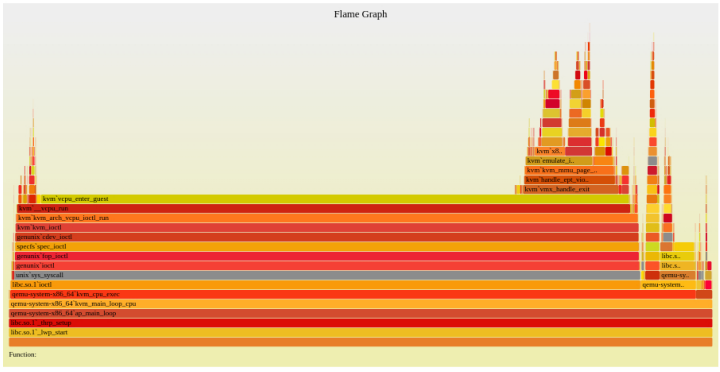
For this flame graph, I executed a workload of short-lived processes to see where kernel time is spent creating them (SVG, PNG):




Apart from performance analysis, this is also a great tool for learning the internals of the Linux kernel.

## 5.4. User+Kernel Flame Graph

This example shows both user and kernel stacks on an illumos kernel hypervisor host (SVG, PNG):



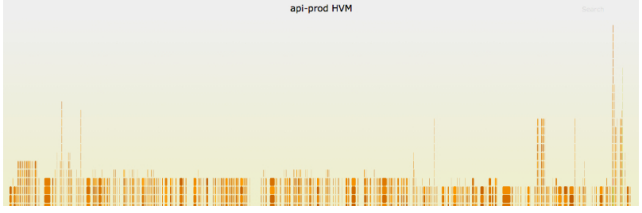

You can also see the standalone SVG and PNG versions. This shows the CPU usage of qemu thread 3, a KVM virtual CPU. Both user and kernel stacks are included (DTrace can access both at the same time), with the syscall in-between colored gray.

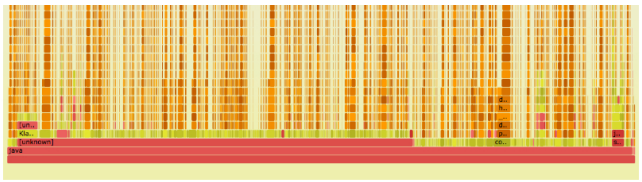
The plateau of vcpu_enter_guest() is where that virtual CPU was executing code inside the virtual machine. I was more interested in the mountains on the right, to examine the performance of the KVM exit code paths.

## 5.5. NUMA Rebalancing

Sometimes flame graphs look busted until you invert the merge order. I included a canonical example in my USENIX ATC'17 Flame Graphs talk and have reproduced it here.

Netflix was rolling out a new Ubuntu version (14.04 Trusty, with its Linux 3.13 kernel) when a service encountered high CPU usage after a few hours. This was after it had "warmed up" with load. A cloud-wide monitoring tool (Atlas) showed it was around 60% system time, which is far higher than is typical (<5%). A system that is CPU-hot in system time is typically easy to solve: Unlike application code, kernel code usually does have working stacks and symbols, so spinning up a flame graph is straightforward. Unfortunately, it looked like this:



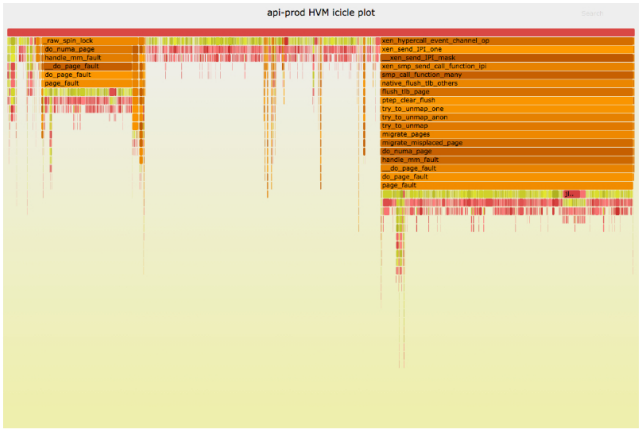


Having "hair" on a flame graph is typically a sign of kernel interrupt load. Interrupts can occur at any time, adding a short burst of CPU cycles, as well as a deep stack trace, on top of any application code. This creates a thin "strand." The more interrupts, the more of them. Now, these strands may be all the same stack frames, but they cannot be merged as they are pasted on top of random application code, so the root stack is different.

Using both of the following options switches the merge order and visualization order:

```
# flamegraph.pl -h
[..]
        --reverse       # generate stack-reversed flame graph
        --inverted      # icicle graph
[..]
```

Which produces:




Ah-ha! Now I can see that this is indeed the same interrupt: These two "towers" alone add up to 55% of all samples. The hair has been merged.

The problem was NUMA rebalancing in this Ubuntu release goes haywire, spending more than half the CPU cycles to better balance memory between nodes for performance. The problem only appeared on multi-NUMA-node systems, and this system had two NUMA nodes. While trying to help application performance, NUAM rebalancing was greatly harming it! My workaround was to disable NUMA rebalancing for this kernel version (there are sysctls to disable/tune it). The issue was fixed in later kernel versions and I enabled it again (although it can still need some tuning to keep it in check).

Note that many people prefer the inverted layout (the "icicle graph") and use it by default. If you are one of them, I'd suggest switching to the bottom-up flame graph layout when reversing the stack order, as a visual clue to the end-user that the profile is now different than usual. It also preserves the ordering of the frames on the y-axis.

I've been supporting a rewrite of flame graphs in d3, as it can support changing merge orders live without having to regenerate the flame graph as with my Perl program. See d3 flame graph.

## 6. C

Easy to profile, see Instructions for steps. A catch is that many compilers, including gcc, don't honour the frame pointer register as a compiler optimization, which breaks frame pointer-based stack walking. This commonly happens for software installed via Linux repositories, and your flame graph will be missing towers. The fixes are either:

- Recompile with -fno-omit-frame-pointer.
- On Linux: install debuginfo for the software with DWARF data and use perf's DWARF stack walker.

## 7. C++

Easy to profile, but can suffer broken stack traces for the same reason as C: see the profiling C section.

## 8. Java

My JavaOne 2016 talk Java Performance Analysis on Linux with Flame Graphs summarizes the latest technique to use Linux perf to generate mixed-mode flame graphs.

### 8.1. Background

In order to generate flame graphs, you need a profiler that can sample stack traces. There has historically been two types of profilers:

1. **System profilers**: like Linux perf, which shows system code paths (eg, JVM GC, syscalls, TCP), but not Java methods.
2. **JVM profilers**: like hprof, LJP, and commercial profilers. These show Java methods, but usually not system code paths.

A flame graph using (1) can be performed as previously described. (2) depends on the profiler you want to use. The flame graph software includes stackcollapse-ljp.pl, for processing the output of the Lightweight Java Profiler (LJP). My blog post Java Flame Graphs summarizes how to use LJP. If you create a system flame graph (eg, using perf on Linux), as well as an LJP flame graph, you can generally solve all issues by examining both.

Ideally, we have one flame graph that does it all: system and Java code paths. Apart from convenience, it also shows system code-paths in Java context, which can be crucial for understanding a profile properly.

The problem is getting a system profiler to understand Java methods and stack traces. If you try Linux perf_events, for example, you'll see hexadecimal numbers and broken stack traces, as it can't convert addresses into Java symbols, and can't walk the JVM stack. DTrace has long supported a jstack() action, however that also has issues, described later.

There are two specific problems:

1. The JVM compiles methods on the fly (just-in-time: JIT), and doesn't expose a traditional symbol table for system profilers to read.
2. The JVM also uses the frame pointer register (RBP on x86-64) as a general purpose register, breaking traditional stack walking.

### 8.2. Linux perf_events

One approach to solve the above two problems involves:

1. A JVMTI agent, perf-map-agent (previously here), which can provide a Java symbol table for perf to read (/tmp/perf-PID.map).
2. The -XX:+PreserveFramePointer JVM option, so that perf can walk frame pointer-based stacks.

The +PreserveFramePointer was added to JDK8u60 to facilitate perf and flame graph generation (I'd emailed a prototype to the hotspot compiler devs mailing list, A hotspot patch for stack profiling (frame pointer), which became JDK-8068945: Use RBP register as proper frame pointer in JIT compiled code on x64). I summarized the final steps on the Netflix Tech blog: Java in Flames. Here are updated steps:

1. Install perf-map-agent:

```
sudo bash
apt-get install cmake
export JAVA_HOME=/path-to-your-new-jdk8
cd /destination-for-perf-map-agent      # I use /usr/lib/jvm
git clone --depth=1 https://github.com/jvm-profiling-tools/perf-map-agent
cd perf-map-agent
cmake .
make
```

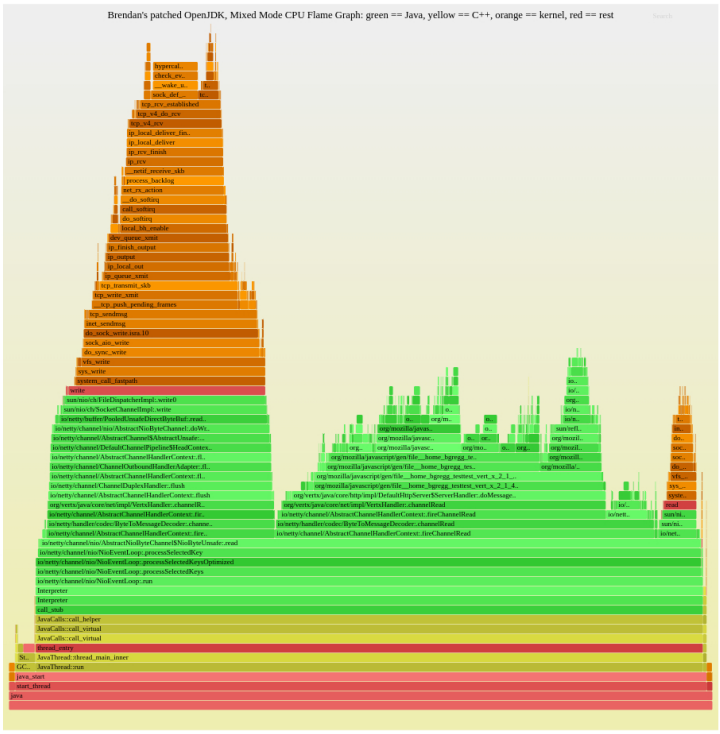
2. Ensure java is running with -XX:+PreserveFramePointer. This may cost a little overhead (usually <1%, but test and quantify; in one rare case it was 10%).

3. Profile and flame graph generation:

```
git clone --depth=1 https://github.com/brendangregg/FlameGraph
sudo bash
perf record -F 49 -a -g -- sleep 30; ./FlameGraph/jmaps
perf script > out.stacks01
cat out.stacks01 | ./FlameGraph/stackcollapse-perf.pl --all | grep -v cpu_idle | \
    ./FlameGraph/flamegraph.pl --color=java --hash > out.stacks01.svg
```

Note that jmaps (a helper script that calls perf-map-agent to do symbol dumps) runs immediately after `perf record`, to minimize symbol churn.

An example resulting flame graph is (SVG, supports click-to-zoom):




When running flamegraph.pl I used `--color=java`, which uses different hues for different types of frames (the stackcollaps-perf.pl option --all includes extra annotations to help with this). Green is Java, yellow is C++, orange is kernel, and red is the remainder (native user-level, or kernel modules).

## 8.3. Java async-profiler

The Java async-profiler is a newer method that uses Linux perf and matches frames with JVMTI calls to AsyncGetCallTrace. An advantage is that it does not require `-XX:+PreserveFramePointer`, which can cost some overhead (usually < 1%, but measure and find out in case it's more).

XXX add more details when I get a chance.

## 8.4. DTrace

DTrace, using its jstack() action, can profile user-level stacks along with Java methods and classes. (In theory: see the bugs listed below.) The capability it uses is called a "DTrace ustack helper" (search for that term to learn more about them). So, to generate a CPU flame graph for a Java program, the stacks can be collected using:

```
# dtrace -n 'profile-97 /execname == "java"/ { @[jstack(100, 8000)] = count(); }
    tick-30s { exit(0); }' -o out.javastacks_01
```

The output file can then be fed to stackcollapse.pl and flamegraph.pl as shown in earlier sections.

The ustack helper action is jstack(), and if it worked you'll have stacks that look like this:

```
              libjvm.so`jni_GetObjectField+0x10f
              libnet.so`Java_java_net_PlainSocketImpl_socketAvailable+0x39
              java/net/PlainSocketImpl.socketAvailable()I*
              java/net/PlainSocketImpl.available()I*
              java/lang/Thread.run()V
              0xfb60035e
              libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_pnMmethodHa...
              libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_pnMmeth...
              libjvm.so`__1cJJavaCallsMcall_virtual6FpnJJavaValue_nGHandle_n...
              libjvm.so`__1cMthread_entry6FpnKJavaThread_pnGThread__v_+0x113
              libjvm.so`__1cKJavaThreadDrun6M_v_+0x2c6
              libjvm.so`java_start+0x1f2
              libc.so.1`_thrp_setup+0x88
              libc.so.1`_lwp_start
                9
```

Note that it includes both libjvm frames and classes/methods: java/net/PlainSocketImpl.available() etc. If there are no classes/methods visible, only hexadecimal numbers, you'll need to get jstack() to work first.

Unfortunately, there are multiple issues. First, jstack() doesn't work at all in many JVM versions. See bug JDK-7187999 for the list. Fortunately, there is a workaround for this first issue, described in Adam's email and illumos issue 3123, and the final usage (which was changed since his original proposal) involves setting an environment variable when launching your Java program, which ensures that the ustack helper is loaded. Eg:

```
illumos# LD_AUDIT_32=/usr/lib/dtrace/libdtrace_forceload.so java myprog
```

Note that this may add over 60 seconds of startup time. (This, itself, needs performance analysis.)

See the resulting flame graph as a SVG or PNG. The workload is ttcp (test TCP), a TCP benchmark written in Java. The flame graph shows that most CPU time is spent in socketWrite(), with 9.2% in releaseFD().

The second issue is that jstack() may not be able to walk stacks properly due to bug JDK-6276264, filed in 2005, and as of 2014 not fixed. For my production workloads, almost all sampled stacks are broken due to this bug. jstack() is unusable.

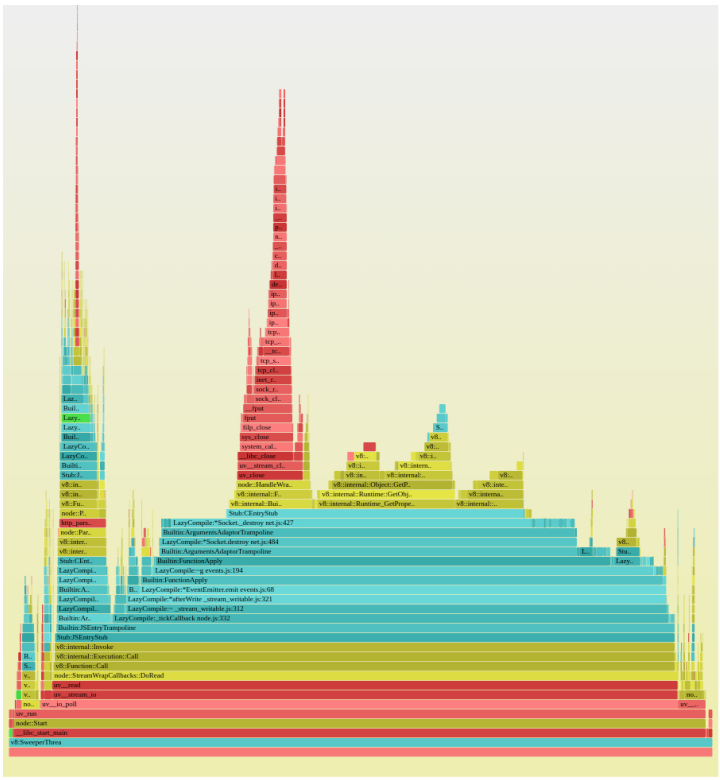
# 9. Node.js

Profiling v8 has similar issues to Java. We ideally would like some profiler that can sample both system code-paths and v8 JavaScript code-paths.

## 9.1. Linux perf

Linux perf_events can profile JavaScript stacks, when using the v8 option --perf_basic_prof or --perf_basic_prof_only_functions. See my blog post Node Flame Graphs on Linux. There is also the original instructions by Trevor Norris, and his example.

Here is an example Linux perf CPU flame graph (SVG):




This uses `--color=js` to use different hues: green == JS, aqua == built-ins, yellow == C++, red == system (native user-level, and kernel).

My example here is little more than hello world, so there's very little JavaScript (green frames) to be seen. Here's a more interesting example.

### 9.2. DTrace

Like with Java, DTrace employs a different approach, using a "ustack helper". See Dave Pacheco's post, where he also demonstrates flame graphs.

## 10. Other Languages

Until I add sections here, please see the Flame Graphs Updates section, and search for the language you are interested in.

## 11. Other Uses

The flame graph visualization works for any stack trace plus value combination, not just stack traces with CPU sample counts like those above. For example, you could trace device I/O, syscalls, off-CPU events, specific function calls, and memory allocation. For any of these, the stack trace can be gathered, along with a relevant value: counts, bytes, or latency. See the other types on the Flame Graphs page.

## 12. Background & Acknowledgements

I created this visualization out of necessity: I had huge amounts of stack sample data from a variety of different performance issues, and needed to quickly dig through it. I first tried creating some text-based tools to summarize the data, with limited success. Then I remembered a time-ordered visualization created by Neelakanth Nadgir (and another Roch Bourbonnais had created and shown me), and thought stack samples could be presented in a similar way. Neel's visualization looked great, but the process of tracing every function entry and return for my workloads altered performance too much. In my case, what mattered more was to have accurate percentages for quantifying code-paths, not the time ordering. This meant I could sample stacks (low overhead) instead of tracing functions (high overhead).

The very first visualization worked, and immediately identified a performance improvement to our KVM code (some added functions were more costly than expected). I've since used it many times for both kernel and application analysis, and others have been using it on a wide range of targets, including node.js profiling. Happy hunting!

## 13. References

- The Flame Graph page, and github repo
- Linux perf_events wiki
- The Unofficial Linux Perf Events Web-Page
- The SystemTap homepage
- My Linux Performance Checklist, which includes perf_events, SystemTap and other tools

See the Flame Graph: Updates section for other articles, examples, and instructions for using flame graphs on many other targets, including Ruby, OS X, Lua, Erlang, node.js, and Oracle.