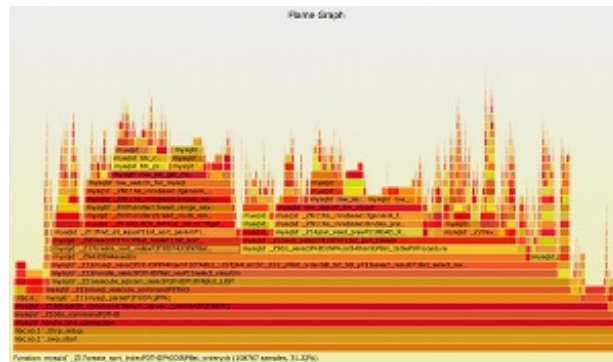


Brendan's blog » Flame Graphs

Flame Graphs

MySQL 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 and accurately.

Problem

There are many tools for profiling applications and the kernel, including *oprofile* and *DTrace*. Here is a profiling example using *DTrace*, where a production MySQL database was busy on-CPU:

```
# 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`__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
1272

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
1643
```

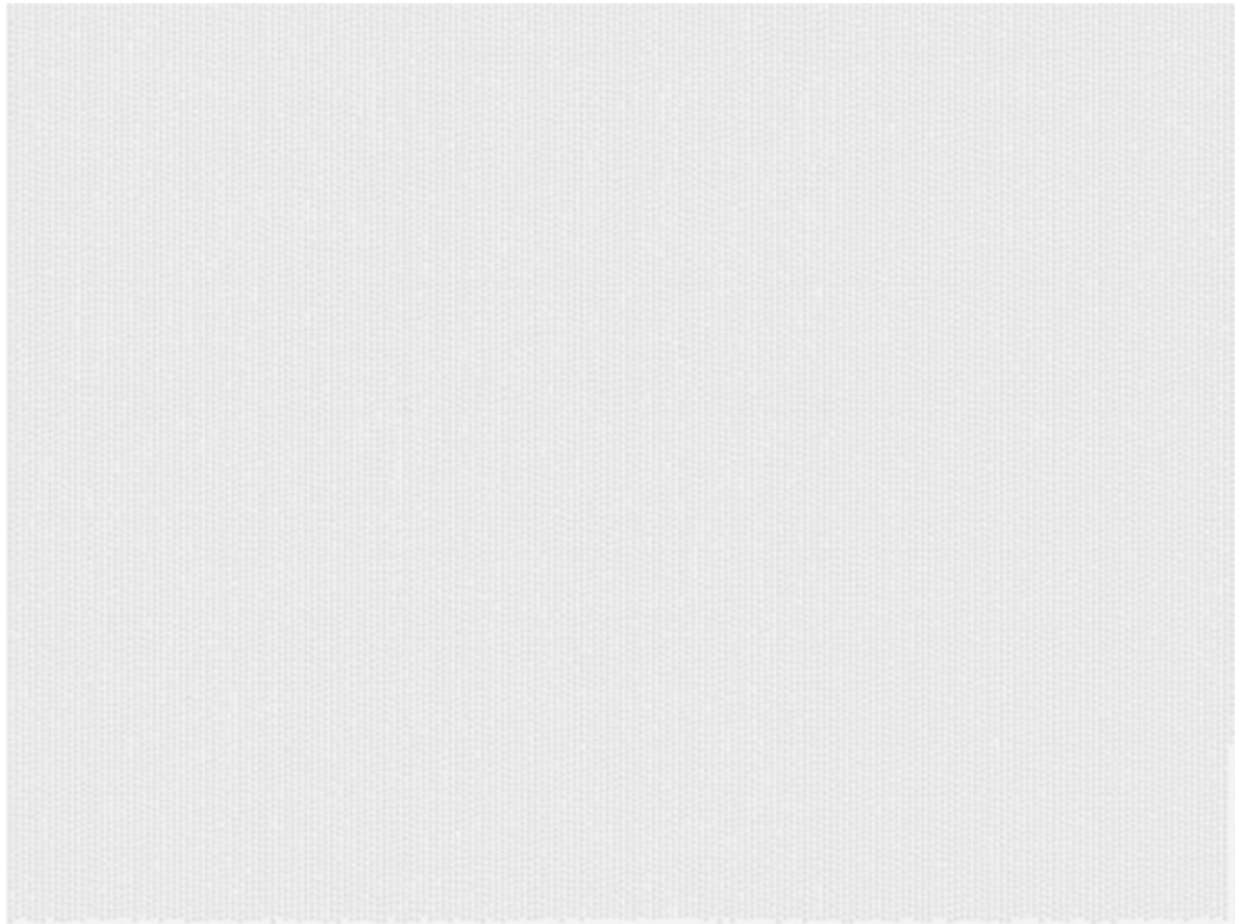
If you haven't seen this output before, it's showing two multi-line stacks with a count at the bottom. Each stack shows the code-path ancestry: the on-CPU function is on top, its parent is below, and so on.

The last two most frequent stacks were shown here. The very last was sampled on-CPU 1,643 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 we should go hunting for tunables to disable system stats in MySQL.

The problem is that most of the output was truncated from this screenshot (the ellipsis "[...]"), and what we see here represents less than 1% of the time spent on-CPU. The total sample count in MySQL was 348,427, and the two stacks above are less than 3,000. Given all the output, it's still hard to read through and comprehend it quickly – even if percentages were included instead of sample counts.

Too much data

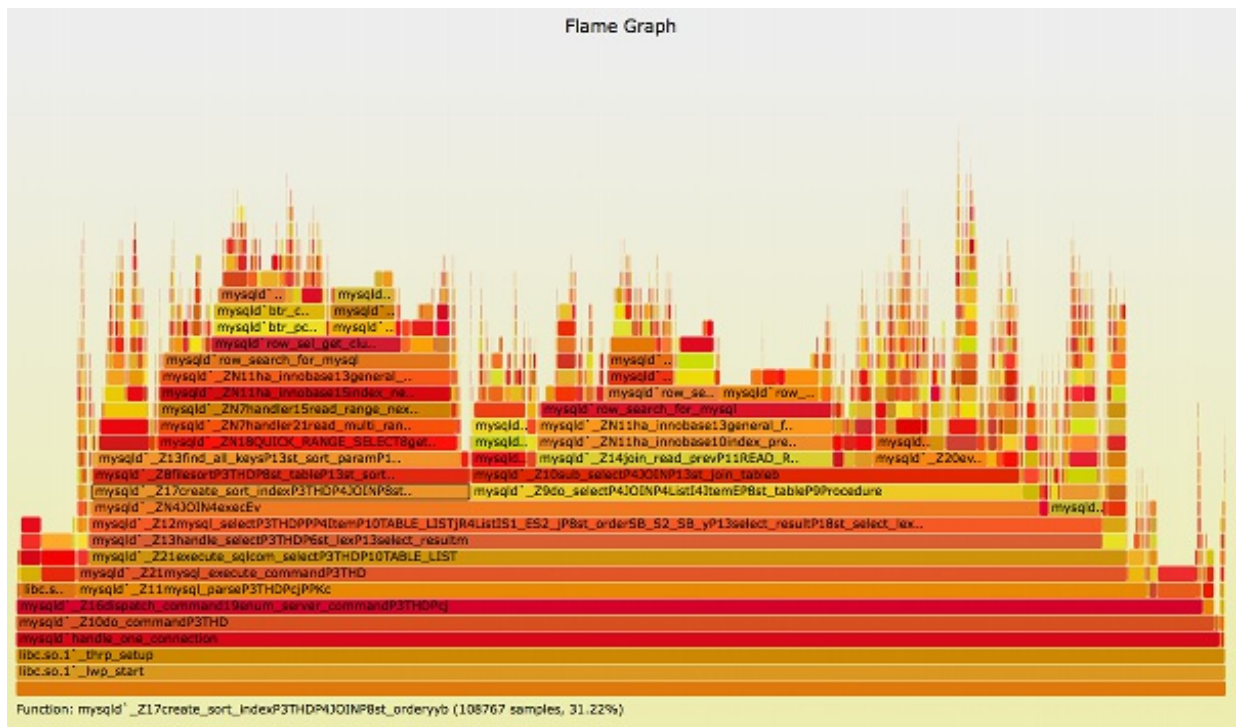
The actual output from the previous command was 591,622 lines long, and included 27,053 stacks like the two pictured above. The entire output looks like this:



Click for a larger image (WARNING: a 7 Mbyte JPEG. Even then, you still can't read the text! I couldn't make the resolution any bigger without breaking the tools I was using to generate it). I've included this to provide a *visual* sense of the amount of data involved.

MySQL Flame Graph

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



Click for the SVG, where you can mouse over elements and see percentages. (If that doesn't work, at least see the high res [PNG](#).)

Description

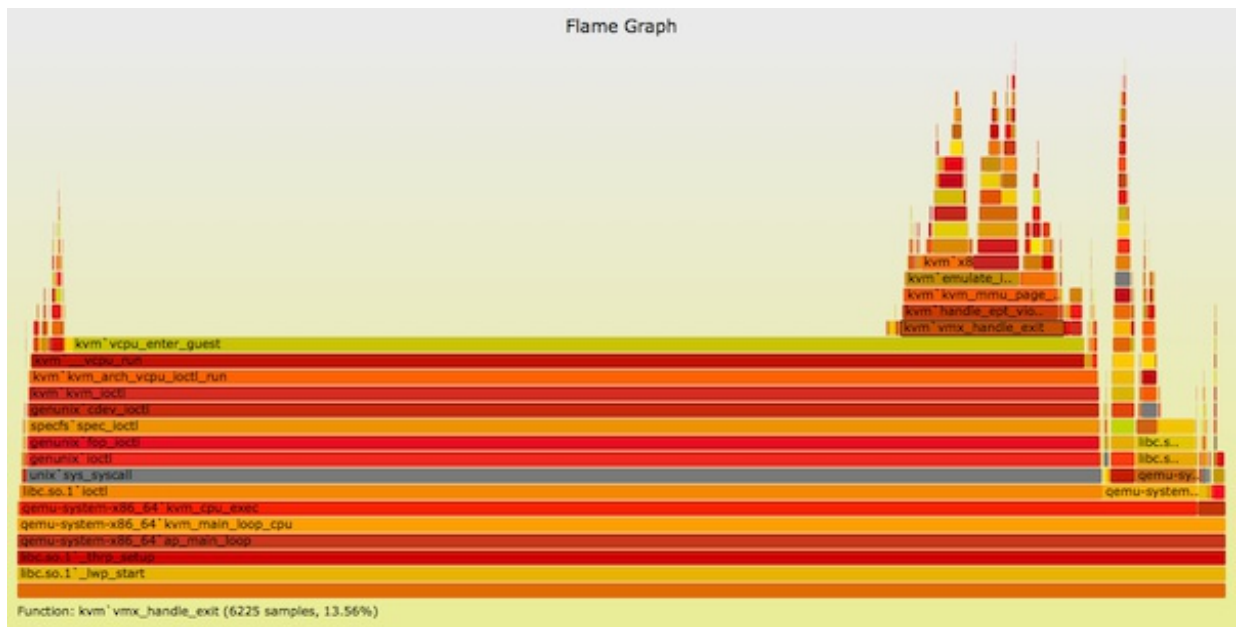
I'll explain this carefully: it may look similar to other visualizations from profilers, but it is different.

- 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.
- 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).
- 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). Wider box functions may be slower than narrow box functions, 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 picked at random to be warm colors. It's called “flame graph” as it's showing what is hot on-CPU. And, it's interactive: mouse over the SVGs to reveal details.

User+Kernel Flame Graph

This example shows both user and kernel stacks (click for SVG version):



This is the CPU usage of qemu thread 3, a [KVM](#) virtual CPU (high res [PNG](#)). Both user and kernel stacks are shown (DTrace can access both at the same time), with the syscall inbetween 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.

... more

[Dave](#) provided a teaser of something he's been working on: [node.js stack translation](#), which he just tried as a flame graph. Here, the user stacks include native JavaScript functions from the V8 engine used by node.js. Check Dave's blog for more information as he posts it.

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 second step generates a line-based output for flamegraph.pl to read, which can also be grep'd to filter for functions of interest. I've currently provided stackcollapse.pl to do this, which processes DTrace output. I suspect it would not be difficult to modify it to process the output from other profilers, to provide input for flamegraph.pl.

An example session:

```
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld" && arg1/ {
  @[ustack()] = count(); } tick-60s { exit(0); }' -o out.stacks
# ./stackcollapse.pl out.stacks > out.folded
# ./flamegraph.pl out.folded > out.svg
```

For that example, only processes called “mysqld” are sampled, and only when they are executing user-land code (the “arg1” check: arg1 is the user-land program counter, so this checks that it is non-zero; arg0 is the kernel). The rate is 997 Hertz for 60 seconds; you may wish to reduce that to lower overhead for busy systems, as needed.

Background

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. Happy hunting!

Update

Not long after this post, Alan Coopersmith generated flame graphs of the [X server](#), and Dave Pacheco created them with [node.js](#) functions. Max Bruning has also shown how he used it to solve an [IP scaling issue](#).

Update 2

More Flame Graph news (updated Apr 2013):

Thanks to everyone who has developed them further and for other communities! I’ll update this page from time to time with more news.

Posted on December 16, 2011 at 10:24 am by Brendan Gregg · [Permalink](#) In: [Performance](#) · Tagged with: [dtrace](#), [performance](#), [profiling](#), [visualizations](#)

[« Previous post](#)

[Next post »](#)