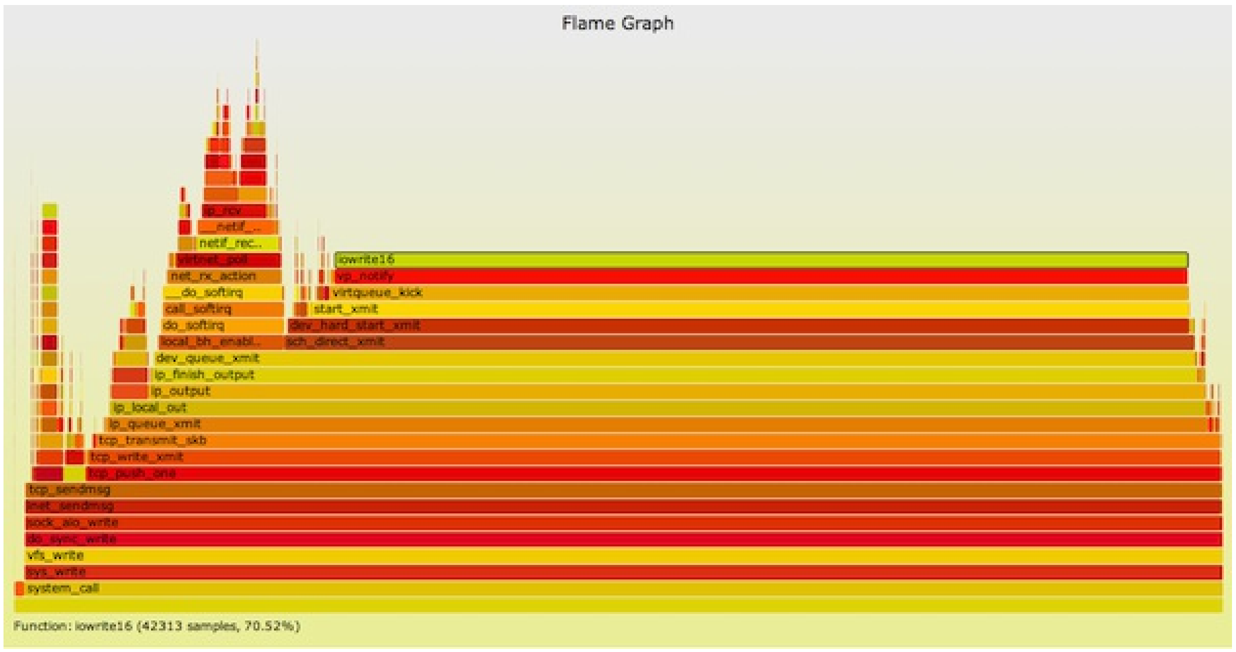
Linux Kernel Performance: Flame Graphs

To get the most out of your systems, you want detailed insight into what the operating system kernel is doing. A typical approach is to sample stack traces; however, the data collected can be time consuming to read or navigate. Flame Graphs are a new way to visualize sampled stack traces, and can be applied to the Linux kernel for some useful (and stunning!) visualizations.

I've used these many times to help solve other kernel and application issues. Since I posted the source to github, others have been using it too (eg, for node.js and IP scaling). I've recently been using them to investigate Linux kernel performance issues (under KVM), which I'll demonstrate in this post using a couple of different profiling tools: perf_events and SystemTap.

Linux Perf Events

This flame graph shows a network workload for the 3.2.9-1 Linux kernel, running as a KVM instance:



click image for interactive SVG; larger PNG here

Flame Graphs show the sample population across the x-axis, and stack depth on the y-axis. Each function (stack frame) is drawn as a rectangle, with the width relative to the number of samples. See my previous post for the full description of how these work.

You can use the mouse to explore where kernel CPU time is spent, quickly quantifying code-paths and determining where performance tuning efforts are best spent. This example shows that most time was spent in the vp_notify() code-path, spending 70.52% of all on-CPU samples performing iowrite16().

This was generated using perf_events and the FlameGraph tools:

```
# perf record -a -g -F 1000 sleep 60
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# cat out.perf-folded | ./flamegraph.pl > perf-kernel.svg
```

The first command runs perf in sampling mode (polling) at 1000 Hertz (-F 1000; more on this later) 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:

```
# ls -lh perf.data
-rw-------. 1 root root 15M Mar 12 20:13 perf.data
```

This can be processed in a variety of ways. On recent versions, the `perf report` command launches an ncurses navigator for call graph inspection. Older versions of perf (or if you pipe the new version) print the call graph as a tree, annotated with percentages:

```
# perf report | more
# ========
# captured on: Wed Mar 14 00:09:59 2012
# hostname : fedora1
# os release : 3.2.9-1.fc16.x86_64
# perf version : 3.2.9-1.fc16.x86_64
# arch : x86_64
# nrcpus online : 1
# nrcpus avail : 1
# cpudesc : QEMU Virtual CPU version 0.14.1
# cpuid : GenuineIntel,6,2,3
# total memory : 1020560 kB
# cmdline : /usr/bin/perf record -a -g -F 1000 sleep 60
# event : name = cycles, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0, excl_usr = ...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# ========
#
# Events: 60K cpu-clock
#
# Overhead        Command       Shared Object                     Symbol
# ........  ...............  ....................  ..................................
#
    72.18%          iperf  [kernel.kallsyms]     [k] iowrite16
                    |
                    --- iowrite16
                       |
                       |--99.53%-- vp_notify
                       |          virtqueue_kick
                       |          start_xmit
                       |          dev_hard_start_xmit
                       |          sch_direct_xmit
                       |          dev_queue_xmit
                       |          ip_finish_output
                       |          ip_output
                       |          ip_local_out
                       |          ip_queue_xmit
                       |          tcp_transmit_skb
                       |          tcp_write_xmit
                       |          |
                       |          |--98.16%-- tcp_push_one
                       |          |          tcp_sendmsg
                       |          |          inet_sendmsg
                       |          |          sock_aio_write
                       |          |          do_sync_write
                       |          |          vfs_write
                       |          |          sys_write
                       |          |          system_call
                       |          |          0x369e40e5cd
                       |          |
                       |           --1.84%-- __tcp_push_pending_frames
[...]
```

This tree starts with the on-CPU functions and works back through the ancestry (this is a "callee based call graph"). This follows the flame graph when reading the flame graph top-down. (This behavior can be flipped by using the "caller" option to -g/–call-graph, instead of the "callee" default, generating a tree that follows the flame graph when read bottom-up.) The hottest (most frequent) stack trace in the flame graph (@70.52%) can be seen in this perf call graph as the product of the top three nodes (72.18% x 99.53% x 98.16%, which are relative rates). `perf report` can also be run with “-g graph” to show absolute overhead rates, in which case “70.52%” is directly displayed on the node.

The perf report tree (and the ncurses navigator) do an excellent job at presenting this information as text. However, with text there are limitations. The output often does not fit in one screen (you could say it doesn’t need to, if the bulk of the samples are identified on the first page). Also, identifying the hottest code paths requires reading the percentages. With the flame graph, all the data is on screen at once, and the hottest code-paths are immediately obvious as the widest functions.

For generating the flame graph, the `perf script` command (a newer addition to perf) was used to dump the stack samples, which are then aggregated by stackcollapse-perf.pl and folded into single lines per-stack. That output is then converted by flamegraph.pl into the SVG. I included a gratuitous “cat” command to make it clear that flamegraph.pl can process the output of a pipe, which could include Unix commands to filter or preprocess (grep, sed, awk).

The last two commands could be connected via a pipe:

```
# perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf-kernel.svg
```

In practice I don’t do this, as I often re-run flamegraph.pl multiple times, and this one-liner would execute everything multiple times. The output of `perf script` can be many Mbytes, taking many seconds to process. By writing stackcollapse-perf.pl to a file, you’ve cached the slowest step, and can also edit the file (vi) to delete unimportant stacks. The one-line-per-stack output of stackcollapse-perf.pl is also great food for grep(1). Eg:

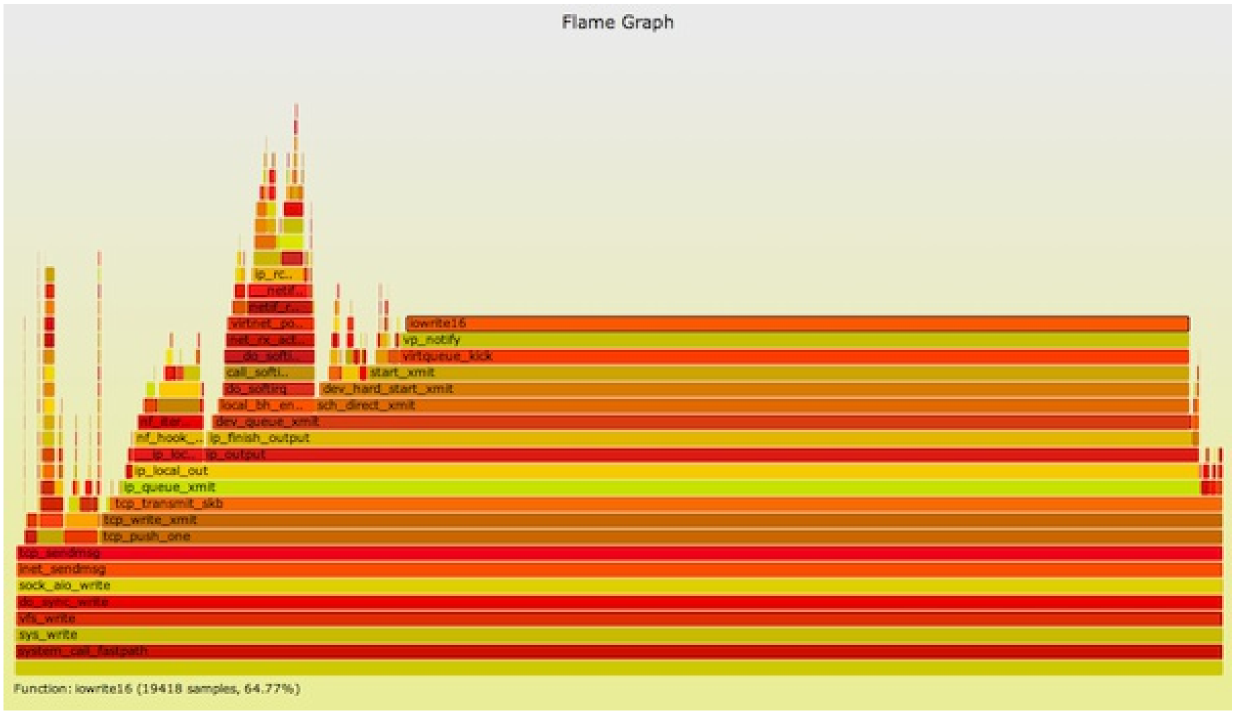
```
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# grep -v cpu_idle out.perf-folded | ./flamegraph.pl > nonidle.svg
# grep ext4 out.perf-folded | ./flamegraph.pl > ext4internals.svg
# egrep 'system_call.*sys_(read|write)' out.perf-folded | ./flamegraph.pl > rw.svg
```

Note that it would be a little more efficient to process the output of `perf report` instead of `perf script`; better still, `perf report` could have a report style (eg, “-g folded”) that output folded stacks directly, obviating the need for stackcollapse-perf.pl. There could even be a perf mode that output the SVG directly (which wouldn’t be the first one; see perf-timechart), although, that would miss the value of being able to grep the folded stacks (which I use frequently).

If you’ve never used perf_events before, you may want to test before production use (it has had kernel panic bugs in the past). My experience has been a good one (no panics).

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. The examples here were generated on Fedora 16 (where it works much better than Ubuntu/CentOS).



The commands for SystemTap version 1.6 are:

```
# stap -s 32 -D MAXTRACE=100 -D MAXSTRINGLEN=4096 -D MAXMAPENTRIES=10240 \
    -D MAXACTION=10000 -D STP_OVERLOAD_THRESHOLD=5000000000 --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 MAXTRACE=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, since you can get errors such as:

```
WARNING: There were 233 transport failures.

ERROR: Array overflow, check MAXMAPENTRIES near identifier 's' at <input>:1:33

MAXACTION:
ERROR: MAXACTION exceeded near operator '{' at <input>:1:87

STP_OVERLOAD_THRESHOLD:
ERROR: probe overhead exceeded threshold
```

The "transport failures" is fixed by increasing the buffer size (-s); the other messages include the names of the tunables that need to be increased.

Also, be sure you have the fix for the #13714 kernel panic (which led to CVE-2012-0875), or the latest version of SystemTap.

On SystemTap v1.7 (latest):

```
# stap -s 32 -D MAXBACKTRACE=100 -D MAXSTRINGLEN=4096 -D
MAXMAPENTRIES=10240 \
   -D MAXACTION=10000 -D STP_OVERLOAD_THRESHOLD=5000000000 --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 only difference is that MAXTRACE became MAXBACKTRACE.

The "-v" option is used to provide details on what SystemTap is doing. When running this one-liner for the first time, it printed:

```
Pass 1: parsed user script and 82 library script(s) using 200364virt/23076res/2996shr kb, in
100usr/10sys/260real ms.
Pass 2: analyzed script: 3 probe(s), 3 function(s), 0 embed(s), 1 global(s) using
200892virt/23868res/3228shr kb, in 0usr/0sys/9real ms.
Pass 3: translated to C into "/tmp/stapllG8kv/stap_778fac70871457bfb540977b1ef376d3_2113_src.c"
using 361936virt/48824res/16640shr kb, in 710usr/90sys/1843real ms.
Pass 4: compiled C into "stap_778fac70871457bfb540977b1ef376d3_2113.ko" in
7630usr/560sys/19155real ms.
Pass 5: starting run.
```

This provides timing details for each initialization stage. Compilation took over 18 seconds, during which the performance of the system dropped by 45%. Fortunately, this only occurs on the first invocation. SystemTap caches the compiled objects under ~/.systemtap, which subsequent executions use. I haven't tried, but I suspect it's possible to compile on one machine (eg, lab, to test for panics), then transfer the cached objects to the target for execution – avoiding the compilation step.

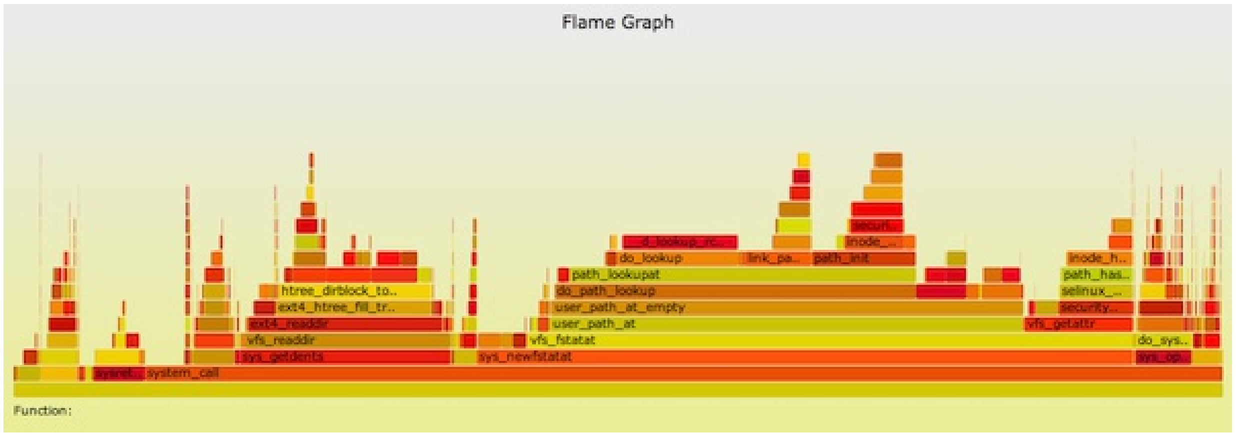
1000 Hertz

The above examples both used 1000 Hertz, so that I could show them both doing the same thing. Ideally, I'd sample at 997 Hertz (or something similar) to avoid sampling in lock-step with timed tasks (which can lead to over-sampling or under-sampling, misrepresenting what is actually happening). With perf_events, the frequency can be set with -F; for example, "-F 997".

For SystemTap, sampling at 997 Hertz (or anything other than CONFIG_HZ) is currently difficult: the timer.hz(997) probe fires at the correct rate, but can't read stack backtraces. It's possible that it can be done via the perf probes based on CPU reference cycle counts (eg, "probe perf.type(0).config(0). sample(N)", where N = CPU_MHz * 1000000 / Sample_Hz). See #13820 for the status on this.

File System

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

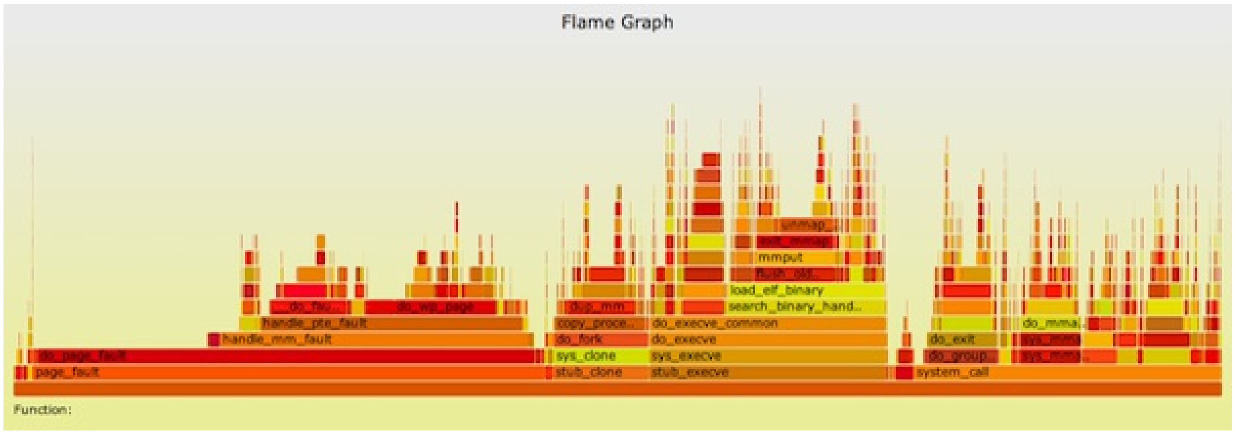


This used perf_events (PNG version); the SystemTap version looks almost identical (SVG, PNG).

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.

Short Lived Processes

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



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

oprofile

Before anyone asks, oprofile could also be used for stack sampling. I haven't written a stackcollapse.pl version for oprofile yet.

Notes

All of the above flame graphs were generated on the Linux 3.2.9 kernel (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 a start. On systems with a high degree of idle time, the flame graph can be dominated by the idle task, which can be filtered using "grep -v cpu_idle" of the folded stacks. Note that by default the flame graph aggregates samples from multiple CPUs; with some shell scripting, you could aggregate samples from multiple hosts as well. Although, it's sometimes useful to generate separate flame graphs for individual CPUs: I've done this for mapped hardware interrupts, for example.

Conclusion

With the Flame Graph visualization, CPU time in the Linux kernel can be quickly understood and inspected. In this post, I showed Flame Graphs for different workloads: networking, file system I/O, and process execution. As a SVG in the browser, they can be navigated with the mouse to inspect element details, revealing percentages so that performance issues or tuning efforts can be quantified.

I used perf_events and SystemTap to sample stack traces, one task out of many that these powerful tools can do. It shouldn't be too hard to use oprofile to provide the data for Flame Graphs as well.

References

Thanks to those using flame graphs and putting it to use in other new areas, and to the SystemTap engineers for answering questions and fixing bugs.

Posted on March 17, 2012 at 9:24 am by Brendan Gregg · Permalink In: Performance · Tagged with: flamegraphs, linux, performance, perf_events, systemtap, visualizations

« Previous post
Next post »