PERF – “Not just a performance monitor”

Written and developed by : **Stephen Johnston**
Principal Software Maintenance Engineer
Red Hat North America GSS-SEG
October 2014

WARNING

The information contained in this course may contain proprietary and confidential information from customers, vendors and Red Hat.

UNDER NO CIRCUMSTANCE IS ANY OF THIS INFORMATION TO BE SHARED WITH ANYONE ELSE INCLUDING CUSTOMERS, FRIENDS, OTHER EMPLOYEES. NO EXCEPTIONS.

There is no inferred permission nor authorization given for making any copies of this material. Do not not share it with anyone. If someone is interested in this material/course please have them contact Red Hat North America GSS-SEG management or the author.

Table of Contents

Introduction

This PERF training originally started out as part of the VATP (Vmcore Analysis Training Primer). However, it soon became obvious that PERF along with several other topics in that training really required a second and more in-depth course developing for it. Part of this document (Brendan Gregg's perf) is still included in that original VATP and for ease of not having 2 different versions of Brendan's document floating around with the problems of keeping them up-to-date, that portion of his document is a copy from the VATP into this document. Hence the odd section numbering of section 3.4.x.x.x. The section that I've created is now unique to this PERF training document only (removed completely from the VATP).

Brendan Gregg's perf tool has and continues to undergo major changes including improvements, corrections, new additions. As a result, there are features in RHEL7 and new ones even in RHEL8 which are not in the RHEL6 version. That makes documenting such a tool a lot more complex. However, this course has an underlying goal of making the student comfortable with the tool and so much of this training will focus on the major features including the following:

- perf stat — Getting statistical analysis from the kernel, events, processes
- perf probe — How to probe the kernel by line number and hex instruction addr
- perf record — Recording data
- perf report — Using the TUI (Text User Interface)
- perf script — The usefullness of looking at raw data and the built in scripts
- perf annotate — Displaying assembler code and hex instruction bytes
- perf bench — The built-in benchmark scripts/programs
- perf archive — Getting an archive file from one system to another to examine

At the conclusion of this course the student will be given an optional series of exercises to perform on their own in order to further test their knowledge of perf and help them be motivated to use the tool and its many features. You cannot expect to learn every subcommand of every perf command with every conceivable argument. There are simply too many to focus on each. Therefore this PERF primer is intended to teach the fundamentals and get the student comfortable with perf so that they can then indulge in the man pages themselves for additional nuances that might expand their knowledge of a/each subcommand. Of course as Brendan continues to develop the product, it is already obvious there are changes/enhancements to the RHEL6 vs RHEL7 vs RHEL8 versions.

1.0 – Brendan Gregg's perf

`perf` is an incredibly powerful tool. You can use it to instrument:

- **CPU performance counters**
  CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify hotspots. Among others, it provides per task, per CPU and per-workload counters, sampling on top of these and source code event annotation.

- **Tracepoints**
  Instrumentation points placed at logical locations in code, such as for system calls, TCP/IP events, file system operations, etc. These have negligible overhead when not in use, and can be enabled by the **`perf`** command to collect information including timestamps and stack traces.

- **Kprobes and uprobes (dynamic tracing)**.
  Dynamically created tracepoints using the kprobes and uprobes frameworks, for kernel and userspace dynamic tracing.

Here is a list of commands that you can use:

```
perf stat
perf top
perf record
perf report
perf annotate
perf probe
perf list
perf bench
perf script      (display raw data in perf.data file)
```

Rather than “reinvent the wheel”, let's go through this excellent paper written by Brendan Gregg.

This is a reprint of Brendan Gregg's excellent writeup of perf. The latest and greates version can be found at:

http://www.brendangregg.com/perf.html

perf Examples

These are some examples of using the perf Linux profiler, which has also been called Performance Counters for Linux (PCL), Linux perf events (LPE), or **perf_events**. Like Vince Weaver, I'll call it **perf_events** so that you can search on that term later. Searching for just "**perf**" finds sites on the police, petroleum, weed control, and a T-shirt. This is not an official **perf** page, for either **perf_events** or the T-shirt.

perf_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions. Questions that can be answered include:

- Why is the kernel on-CPU so much? What code-paths?
- Which code-paths are causing CPU level 2 cache misses?
- Are the CPUs stalled on memory I/O?
- Which code-paths are allocating memory, and how much?
- What is triggering TCP retransmits?
- Is a certain kernel function being called, and how often?
- What reasons are threads leaving the CPU?

perf_events is part of the Linux kernel, under **tools/perf**. While it uses many Linux tracing features, some are not yet exposed via the **perf** command, and need to be used via the **ftrace** interface instead. My perf-tools collection (github) uses both **perf_events** and **ftrace** as needed.

This page includes my examples of **perf_events**. The key sections are:

- One-Liners,
- Presentations
- Prerequisites
- CPU statistics
- CPU profiling
- Static Tracing
- Dynamic Tracing

- Flame Graphs
- Heat Maps

Also see my Posts about perf_events, and Links for the main (official) **perf_events** page, awesome tutorial, and other links. The next sections introduce **perf_events** further, starting with a screenshot, one-liners, and then background.

This page is under construction, and there's a lot more to **perf_events** that I'd like to add. Hopefully this is useful so far.

1.1 - Screenshot

I like explaining tools by starting with an actual screenshot showing something useful. Here's **perf** version 3.9.3 tracing disk I/O:

```
# perf record -e block:block_rq_issue -ag
^C
# ls -l perf.data
-rw------- 1 root root 3458162 Jan 26 03:03 perf.data
# perf report
[...]
# Samples: 2K of event 'block:block_rq_issue'
# Event count (approx.): 2216
#
# Overhead       Command      Shared Object                Symbol
# ........  ............  .................  ....................
#
    32.13%            dd  [kernel.kallsyms]  [k] blk_peek_request
                      |
                      --- blk_peek_request
                          virtblk_request
                          __blk_run_queue
                         |
                         |--98.31%-- queue_unplugged
                         |           blk_flush_plug_list
                         |           |
                         |           |--91.00%-- blk_queue_bio
                         |           |           generic_make_request
                         |           |           submit_bio
                         |           |           ext4_io_submit
                         |           |           |
                         |           |           |--58.71%-- ext4_bio_write_page
                         |           |           |           mpage_da_submit_io
                         |           |           |           mpage_da_map_and_submit
                         |           |           |           write_cache_pages_da
                         |           |           |           ext4_da_writepages
                         |           |           |           do_writepages
                         |           |           |           __filemap_fdatawrite_range
```

```
                              |            |            |           filemap_flush
                              |            |            |           ext4_alloc_da_blocks
                              |            |            |           ext4_release_file
                              |            |            |           __fput
                              |            |            |           ____fput
                              |            |            |           task_work_run
                              |            |            |           do_notify_resume
                              |            |            |           int_signal
                              |            |            |           close
                              |            |            |           0x0
                              |            |            |
                              |            |             --41.29%-- mpage_da_submit_io
[...]
```

A perf record command was used to trace the **block:block_rq_issue** probe, which fires when a block device I/O request is issued (disk I/O). Options included **-a** to trace all CPUs, and **-g** to capture call graphs (stack traces). Trace data is written to a **perf.data** file, and tracing ended when **Ctrl-C** was hit. A summary of the **perf.data** file was printed using **perf report**, which builds a tree from the stack traces, coalescing common paths, and showing percentages for each path.

The **perf report** output shows that 2,216 events were traced (disk I/O), 32% of which from a **dd** command. These were issued by the kernel function **blk_peek_request()**, and walking down the stacks, about half of these 32% were from the **close()** system call.

1.2 - One-Liners

Some useful one-liners I've gathered or written:

Listing Events

```
# Listing all currently known events:
perf list

# Listing sched tracepoints:
perf list 'sched:*'
```

Counting Events

```
# CPU counter statistics for the specified command:
perf stat command

# Detailed CPU counter statistics (includes extras) for the specified command:
perf stat -d command

# CPU counter statistics for the specified PID, until Ctrl-C:
perf stat -p PID

# CPU counter statistics for the entire system, for 5 seconds:
perf stat -a sleep 5

# Various basic CPU statistics, system wide, for 10 seconds:
perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10

# Various CPU level 1 data cache statistics for the specified command:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores command

# Various CPU data TLB statistics for the specified command:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

# Various CPU last level cache statistics for the specified command:
perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches command

# Using raw PMC counters, eg, unhalted core cycles:
perf stat -e r003c -a sleep 5

# Count system calls for the specified PID, until Ctrl-C:
perf stat -e 'syscalls:sys_enter_*' -p PID

# Count system calls for the entire system, for 5 seconds:
perf stat -e 'syscalls:sys_enter_*' -a sleep 5

# Count scheduler events for the specified PID, until Ctrl-C:
perf stat -e 'sched:*' -p PID

# Count scheduler events for the specified PID, for 10 seconds:
perf stat -e 'sched:*' -p PID sleep 10
```

```
# Count ext4 events for the entire system, for 10 seconds:
perf stat -e 'ext4:*' -a sleep 10

# Count block device I/O events for the entire system, for 10 seconds:
perf stat -e 'block:*' -a sleep 10

# Count all vmscan events, printing a report every second:
perf stat -e 'vmscan:*' -a -I 1000

# Show system calls by process, refreshing every 2 seconds:
perf top -e raw_syscalls:sys_enter -ns comm

# Show sent network packets by on-CPU process, rolling output (no clear):
stdbuf -oL perf top -e net:net_dev_xmit -ns comm | strings
```

Profiling

```
# Sample on-CPU functions for the specified command, at 99 Hertz:
perf record -F 99 command

# Sample on-CPU functions for the specified PID, at 99 Hertz, until Ctrl-C:
perf record -F 99 -p PID

# Sample on-CPU functions for the specified PID, at 99 Hertz, for 10 seconds:
perf record -F 99 -p PID sleep 10

# Sample CPU stack traces for the specified PID, at 99 Hertz, for 10 seconds:
perf record -F 99 -p PID -g -- sleep 10

# Sample CPU stack traces for the PID, using dwarf to unwind stacks, at 99 Hertz,
for 10 seconds:
perf record -F 99 -p PID -g dwarf sleep 10

# Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds:
perf record -F 99 -ag -- sleep 10

# If the previous command didn't work, try forcing perf to use the cpu-clock event:
perf record -F 99 -e cpu-clock -ag -- sleep 10

# Sample CPU stack traces for the entire system, with dwarf stacks, at 99 Hertz,
for 10 seconds:
perf record -F 99 -ag dwarf sleep 10

# Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 seconds:
perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5

# Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds:
perf record -e LLC-load-misses -c 100 -ag -- sleep 5

# Sample on-CPU kernel instructions, for 5 seconds:
perf record -e cycles:k -a -- sleep 5
```

```
# Sample on-CPU user instructions, for 5 seconds:
perf record -e cycles:u -a -- sleep 5

# Sample on-CPU instructions precisely (using PEBS), for 5 seconds:
perf record -e cycles:p -a -- sleep 5

# Perform branch tracing (needs HW support), for 1 second:
perf record -b -a sleep 1

# Sample CPUs at 49 Hertz, and show top addresses and symbols, live (no perf.data file):
perf top -F 49

# Sample CPUs at 49 Hertz, and show top process names and segments, live:
perf top -F 49 -ns comm,dso
```

Static Tracing

```
# Trace new processes, until Ctrl-C:
perf record -e sched:sched_process_exec -a

# Trace all context-switches, until Ctrl-C:
perf record -e context-switches -a

# Trace context-switches via sched tracepoint, until Ctrl-C:
perf record -e sched:sched_switch -a

# Trace all context-switches with stack traces, until Ctrl-C:
perf record -e context-switches -ag

# Trace all context-switches with stack traces, for 10 seconds:
perf record -e context-switches -ag -- sleep 10

# Trace CPU migrations, for 10 seconds:
perf record -e migrations -a -- sleep 10

# Trace all connect()s with stack traces (outbound connections), until Ctrl-C:
perf record -e syscalls:sys_enter_connect -ag

# Trace all accepts()s with stack traces (inbound connections), until Ctrl-C:
perf record -e syscalls:sys_enter_accept* -ag

# Trace all block device (disk I/O) requests with stack traces, until Ctrl-C:
perf record -e block:block_rq_insert -ag

# Trace all block device issues and completions (has timestamps), until Ctrl-C:
perf record -e block:block_rq_issue -e block:block_rq_complete -a

# Trace all block completions, of size at least 100 Kbytes, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'nr_sector > 200'

# Trace all block completions, synchronous writes only, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'rwbs == "WS"'

# Trace all block completions, all types of writes, until Ctrl-C:
perf record -e block:block_rq_complete --filter 'rwbs ~ "*W*"'
```

```
# Trace all minor faults (RSS growth) with stack traces, until Ctrl-C:
perf record -e minor-faults -ag

# Trace all page faults with stack traces, until Ctrl-C:
perf record -e page-faults -ag

# Trace all ext4 calls, and write to a non-ext4 location, until Ctrl-C:
perf record -e 'ext4:*' -o /tmp/perf.data -a

# Trace kswapd wakeup events, until Ctrl-C:
perf record -e vmscan:mm_vmscan_wakeup_kswapd -ag

# Add Node.js USDT probes (Linux 4.10+):
perf buildid-cache --add `which node`

# Trace the node http__server__request USDT event (Linux 4.10+):
perf record -e sdt_node:http__server__request -a
```

Dynamic Tracing

```
# Add a tracepoint for the kernel tcp_sendmsg() function entry ("--add" is
optional):
perf probe --add tcp_sendmsg

# Remove the tcp_sendmsg() tracepoint (or use "--del"):
perf probe -d tcp_sendmsg

# Add a tracepoint for the kernel tcp_sendmsg() function return:
perf probe 'tcp_sendmsg%return'

# Show available variables for the kernel tcp_sendmsg() function (needs debuginfo):
perf probe -V tcp_sendmsg

# Show available variables for the kernel tcp_sendmsg() function, plus external
vars (needs debuginfo):
perf probe -V tcp_sendmsg --externs

# Show available line probes for tcp_sendmsg() (needs debuginfo):
perf probe -L tcp_sendmsg

# Show available variables for tcp_sendmsg() at line number 81 (needs debuginfo):
perf probe -V tcp_sendmsg:81

# Add a tracepoint for tcp_sendmsg(), with three entry argument registers (platform
specific):
# Note [SJ]. Register names you can use for Intel and AMD are:
#            %di %si %dx %cx %ax %bx %bp %sp %ip %flags
#            %r8 %r9 %r10 %r11 %r12 %r13 %r14 %r15      (Yes I know these are %r's)
perf probe 'tcp_sendmsg %ax %dx %cx'

# Add a tracepoint for tcp_sendmsg(), with an alias ("bytes") for the %cx register
(platform specific):
perf probe 'tcp_sendmsg bytes=%cx'

# Trace previously created probe when the bytes (alias) variable is greater than 100:
perf record -e probe:tcp_sendmsg --filter 'bytes > 100'
```

```
# Add a tracepoint for tcp_sendmsg() return, and capture the return value:
perf probe 'tcp_sendmsg%return $retval'

# Add a tracepoint for tcp_sendmsg(), and "size" entry argument (reliable, but
needs debuginfo):
perf probe 'tcp_sendmsg size'

# Add a tracepoint for tcp_sendmsg(), with size and socket state (needs debuginfo):
perf probe 'tcp_sendmsg size sk->__sk_common.skc_state'

# Tell me how on Earth you would do this, but don't actually do it (needs debuginfo):
perf probe -nv 'tcp_sendmsg size sk->__sk_common.skc_state'

# Trace previous probe when size is non-zero, and state is not TCP_ESTABLISHED(1)
(needs debuginfo):
perf record -e probe:tcp_sendmsg --filter 'size > 0 && skc_state != 1' -a

# Add a tracepoint for tcp_sendmsg() line 81 with local variable seglen (needs
debuginfo):
perf probe 'tcp_sendmsg:81 seglen'

# Add a tracepoint for do_sys_open() with the filename as a string (needs debuginfo):
perf probe 'do_sys_open filename:string'

# Add a tracepoint for myfunc() return, and include the retval as a string:
perf probe 'myfunc%return +0($retval):string'

# Add a tracepoint for the user-level malloc() function from libc:
perf probe -x /lib64/libc.so.6 malloc

# List currently available dynamic probes:
perf probe -l
```

Mixed

```
# Sample stacks at 99 Hertz, and, context switches:
perf record -F99 -e cpu-clock -e cs -a -g

# Sample stacks to 2 levels deep, and, context switch stacks to 5 levels (needs 4.8):
perf record -F99 -e cpu-clock/max-stack=2/ -e cs/max-stack=5/ -a -g
```

Special

```
# Record cacheline events (Linux 4.10+):
perf c2c record -a -- sleep 10

# Report cacheline events from previous recording (Linux 4.10+):
perf c2c report
```

Reporting

```
# Show perf.data in an ncurses browser (TUI) if possible:
perf report

# Show perf.data with a column for sample count:
perf report -n

# Show perf.data as a text report, with data coalesced and percentages:
perf report --stdio

# Report, with stacks in folded format: one line per stack (needs 4.4):
perf report --stdio -n -g folded

# List all events from perf.data:
perf script

# List all perf.data events, with data header (newer kernels; was previously
default):
perf script --header

# List all perf.data events, with customized fields (< Linux 4.1):
perf script -f time,event,trace

# List all perf.data events, with customized fields (>= Linux 4.1):
perf script -F time,event,trace

# List all perf.data events, with my recommended fields (needs record -a; newer
kernels):
perf script --header -F comm,pid,tid,cpu,time,event,ip,sym,dso

# List all perf.data events, with my recommended fields (needs record -a; older
kernels):
perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso

# Dump raw contents from perf.data as hex (for debugging):
perf script -D

# Disassemble and annotate instructions with percentages (needs some debuginfo):
perf annotate --stdio
```

These one-liners serve to illustrate the capabilities of perf_events, and can also be used a bite-sized tutorial: learn perf_events one line at a time. You can also print these out as a perf_events cheatsheet.

1.3 - Presentations

Kernel Recipes (2017)

At Kernel Recipes 2017 I gave an updated talk on Linux perf at Netflix, focusing on getting CPU profiling and flame graphs to work. This talk includes a crash course on perf_events, plus gotchas such as fixing stack traces and symbols when profiling Java, Node.js, VMs, and containers.

A video of the talk is on youtube and the slides are on slideshare:

1.4 - Background

The following sections provide some background for understanding perf_events and how to use it. I'll describe the prerequisites, audience, usage, events, and tracepoints.

1.4.1 - Prerequisites

The **`perf`** tool is in the **linux-tools-common** package. Start by adding that, then running "**`perf`**" to see if you get the USAGE message. It may tell you to install another related package (linux-tools-*kernelversion*).

You can also build and add **`perf`** from the Linux kernel source. See the Building section.

To get the most out **`perf`**, you'll want symbols and stack traces. These may work by default in your Linux distribution, or they may require the addition of packages, or recompilation of the kernel with additional config options.

1.4.2 - Symbols

`perf_events`, like other debug tools, needs symbol information (symbols). These are used to translate memory addresses into function and variable names, so that they can be read by us humans. Without symbols, you'll see hexadecimal numbers representing the memory addresses profiled.

The following **`perf`** report output shows stack traces, however, only hexadecimal numbers can be seen:

```
    57.14%     sshd  libc-2.15.so        [.] connect
               |
               --- connect
                  |
                  |--25.00%-- 0x7ff3c1cddf29
                  |
                  |--25.00%-- 0x7ff3bfe82761
                  |          0x7ff3bfe82b7c
                  |
                  |--25.00%-- 0x7ff3bfe82dfc
                   --25.00%-- [...]
```

If the software was added by packages, you may find debug packages (often "**`-dbgsym`**") which provide the symbols. Sometimes **`perf`** report will tell you to install these, eg: "no symbols found in **`/bin/dd`**, maybe install a debug package?".

Here's the same **`perf`** report output seen earlier, after adding **`openssh-server-dbgsym`** and **`libc6-`**

dbgsym (this is on ubuntu 12.04):

```
    57.14%     sshd  libc-2.15.so        [.] __GI___connect_internal
               |
               --- __GI___connect_internal
                  |
                  |--25.00%-- add_one_listen_addr.isra.0
                  |
                  |--25.00%-- __nscd_get_mapping
                  |           __nscd_get_map_ref
                  |
                  |--25.00%-- __nscd_open_socket
                   --25.00%-- [...]
```

I find it useful to add both **libc6-dbgsym** and **coreutils-dbgsym**, to provide some symbol coverage of user-level OS codepaths.

Another way to get symbols is to compile the software yourself. For example, I just compiled node (Node.js):

```
# file node-v0.10.28/out/Release/node
node-v0.10.28/out/Release/node: ELF 64-bit LSB executable, ... not stripped
```

This has not been stripped, so I can profile node and see more than just hex. If the result is stripped, configure your build system not to run **strip**(1) on the output binaries.

Kernel-level symbols are in the kernel debuginfo package, or when the kernel is compiled with **CONFIG_KALLSYMS**.

1.4.3 - JIT Symbols (Java, Node.js)

Programs that have virtual machines (VMs), like Java's JVM and node's v8, execute their own virtual processor, which has its own way of executing functions and managing stacks. If you profile these using perf_events, you'll see symbols for the VM engine, which have some use (eg, to identify if time is spent in GC), but you won't see the language-level context you might be expecting. Eg, you won't see Java classes and methods.

perf_events has JIT support to solve this, which requires the VM to maintain a **/tmp/perf-PID.map** file for symbol translation. Java can do this with perf-map-agent, and Node.js 0.11.13+ with **--perf_basic_prof**. See my blog post Node.js flame graphs on Linux for the steps.

Note that Java may not show full stacks to begin with, due to hotspot on x86 omitting the frame pointer (just like **gcc**). On newer versions (JDK 8u60+), you can use the **-XX:+PreserveFramePointer** option to fix this behavior, and profile fully using **perf**. See my Netflix Tech Blog post, Java in Flames, for a full writeup, and my Java flame graphs section, which links to an older patch and includes an example resulting flame graph.

1.4.4 - Stack Traces

Always compile with frame pointers. Omitting frame pointers is an evil compiler optimization that breaks debuggers, and sadly, is often the default. Without them, you may see incomplete stacks from **`perf_events`**, like seen in the earlier **`sshd`** symbols example. There are two ways to fix this: either using dwarf data to unwind the stack, or returning the frame pointers.

There are other stack walking techniques, like BTS (Branch Trace Store), and the new ORC unwinder. I'll add docs for them at some point (and as perf support arrives).

Frame Pointers

The earlier **`sshd`** example was a default build of OpenSSH, which uses compiler optimizations (-O2), which in this case has omitted the frame pointer. Here's how it looks after recompiling OpenSSH with **-fno-omit-frame-pointer**:

```
    100.00%     sshd  libc-2.15.so   [.] __GI___connect_internal
               |
               --- __GI___connect_internal
                  |
                  |--30.00%-- add_one_listen_addr.isra.0
                  |          add_listen_addr
                  |          fill_default_server_options
                  |          main
                  |          __libc_start_main
                  |
                  |--20.00%-- __nscd_get_mapping
                  |          __nscd_get_map_ref
                  |
                  |--20.00%-- __nscd_open_socket
                   --30.00%-- [...]
```

Now the ancestry from **`add_one_listen_addr()`** can be seen, down to **`main()`** and **`__libc_start_main()`**.

The kernel can suffer the same problem. Here's an example CPU profile collected on an idle server, with stack traces (**`-g`**):

```
    99.97%  swapper  [kernel.kallsyms]  [k] default_idle
            |
            --- default_idle

     0.03%     sshd  [kernel.kallsyms]  [k] iowrite16
               |
               --- iowrite16
                   __write_nocancel
                   (nil)
```

The kernel stack traces are incomplete. Now a similar profile with **CONFIG_FRAME_POINTER=y:**

```
    99.97%  swapper  [kernel.kallsyms]  [k] default_idle
            |
            --- default_idle
                cpu_idle
               |
               |--87.50%-- start_secondary
               |
                --12.50%-- rest_init
                          start_kernel
                          x86_64_start_reservations
                          x86_64_start_kernel

     0.03%     sshd  [kernel.kallsyms]  [k] iowrite16
               |
               --- iowrite16
                   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
                   __tcp_push_pending_frames
                   tcp_sendmsg
                   inet_sendmsg
                   sock_aio_write
                   do_sync_write
                   vfs_write
                   sys_write
                   system_call_fastpath
                   __write_nocancel
```

Much better -- the entire path from the **write()** syscall (**__write_nocancel**) to **iowrite16()** can be seen.

Dwarf

Since about the 3.9 kernel, perf_events has supported a workaround for missing frame pointers in user-level stacks: libunwind, which uses dwarf. This can be enabled using "**--call-graph dwarf**" (or "**-g dwarf**").

Also see the Building section for other notes about building perf_events, as without the right library, it may build itself without dwarf support.

LBR

You must have Last Branch Record access to be able to use this. It is disabled in most cloud environments, where you'll get this error:

```
# perf record -F 99 -a --call-graph lbr
Error:
PMU Hardware doesn't support sampling/overflow-interrupts.
```

Here's an example of it working:

```
# perf record -F 99 -a --call-graph lbr
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.903 MB perf.data (163 samples) ]
# perf script
[...]
stackcollapse-p 23867 [007] 4762187.971824:   29003297 cycles:ppp:
                  1430c0 Perl_re_intuit_start (/usr/bin/perl)
                  144118 Perl_regexec_flags (/usr/bin/perl)
                   cfcc9 Perl_pp_match (/usr/bin/perl)
                   cbee3 Perl_runops_standard (/usr/bin/perl)
                   51fb3 perl_run (/usr/bin/perl)
                   2b168 main (/usr/bin/perl)

stackcollapse-p 23867 [007] 4762187.980184:   31532281 cycles:ppp:
                   e3660 Perl_sv_force_normal_flags (/usr/bin/perl)
                  109b86 Perl_leave_scope (/usr/bin/perl)
                  1139db Perl_pp_leave (/usr/bin/perl)
                   cbee3 Perl_runops_standard (/usr/bin/perl)
                   51fb3 perl_run (/usr/bin/perl)
                   2b168 main (/usr/bin/perl)

stackcollapse-p 23867 [007] 4762187.989283:   32341031 cycles:ppp:
                   cfae0 Perl_pp_match (/usr/bin/perl)
                   cbee3 Perl_runops_standard (/usr/bin/perl)
                   51fb3 perl_run (/usr/bin/perl)
                   2b168 main (/usr/bin/perl)
```

Nice! Note that LBR is usually limited in stack depth (either 8, 16, or 32 frames), so it may not be suitable for deep stacks or flame graph generation, as flame graphs need to walk to the common root for merging.

Here's that same program sampled using the by-default frame pointer walk:

```
# perf record -F 99 -a -g
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.882 MB perf.data (81 samples) ]
# perf script
[...]
stackcollapse-p 23883 [005] 4762405.747834:   35044916 cycles:ppp:
                  135b83 [unknown] (/usr/bin/perl)
```

```
stackcollapse-p 23883 [005] 4762405.757935:   35036297 cycles:ppp:
                    ee67d Perl_sv_gets (/usr/bin/perl)

stackcollapse-p 23883 [005] 4762405.768038:   35045174 cycles:ppp:
                   137334 [unknown] (/usr/bin/perl)
```

You can recompile Perl with frame pointer support (in its ./Configure, it asks what compiler options: add -fno-omit-frame-pointer). Or you can use LBR if it's available, and you don't need very long stacks.

1.4.5 - Audience

To use **perf_events**, you'll either:

- Develop your own commands
- Run example commands

Developing new invocations of **perf_events** requires the study of kernel and application code, which isn't for everyone. Many more people will use **perf_events** by running commands developed by other people, like the examples on this page. This can work out fine: your organization may only need one or two people who can develop **perf_events** commands or source them, and then share them for use by the entire operation and support groups.

Either way, you need to know the capabilities of perf_events so you know when to reach for it, whether that means searching for an example command or writing your own. One goal of the examples that follow is just to show you what can be done, to help you learn these capabilities. You should also browse examples on other sites (Links).

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).

1.4.6 - Usage

perf_events provides a command line tool, **perf**, and subcommands for various profiling activities. This is a single interface for the different instrumentation frameworks that provide the various events.

The **perf** command alone will list the subcommands; here is perf version 3.9.3 (for the Linux 3.9.3 kernel):

```
# perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
```

```
The most commonly used perf commands are:
  annotate        Read perf.data (created by perf record) and display annotated code
  archive         Create archive with object files with build-ids found in perf.data file
  bench           General framework for benchmark suites
  buildid-cache   Manage build-id cache.
  buildid-list    List the buildids in a perf.data file
  config          Get and set variables in a configuration file.
  data            Data file related processing
  diff            Read perf.data files and display the differential profile
  evlist          List the event names in a perf.data file
  inject          Filter to augment the events stream with additional information
  kmem            Tool to trace/measure kernel memory properties
  kvm             Tool to trace/measure kvm guest os
  list            List all symbolic event types
  lock            Analyze lock events
  mem             Profile memory accesses
  record          Run a command and record its profile into perf.data
  report          Read perf.data (created by perf record) and display the profile
  sched           Tool to trace/measure scheduler properties (latencies)
  script          Read perf.data (created by perf record) and display trace output
  stat            Run a command and gather performance counter statistics
  test            Runs sanity tests.
  timechart       Tool to visualize total system behavior during a workload
  top             System profiling tool.
  probe           Define new dynamic tracepoints
  trace           strace inspired tool

See 'perf help COMMAND' for more information on a specific command.
```

Apart from separate help for each subcommand, there is also documentation in the kernel source under **tools/perf/Documentation**. Note that **perf** has evolved, with different functionality added over time, and so its usage may not feel consistent as you switch between activities. It's best to think of it as a multi-tool.

perf_events can instrument in three ways (using the **perf_events** terminology):

- **counting** events in-kernel context, where a summary of counts is printed by perf. This mode does not generate a **perf.data** file.

- **sampling** events, which writes event data to a kernel buffer, which is read at a gentle asynchronous rate by the **perf** command to write to the **perf.data** file. This file is then read by the **perf report** or **perf script** commands.

- **bpf** programs on events, a new feature in Linux 4.4+ kernels that can execute custom user-defined programs in kernel space, which can perform efficient filters and summaries of the data. Eg, efficiently-measured latency histograms.

Try starting by counting events using the **perf stat** command, to see if this is sufficient. This subcommand costs the least overhead.

When using the sampling mode with **perf record**, you'll need to be a little careful about the overheads, as the capture files can quickly become hundreds of Mbytes. It depends on the rate of the event you are tracing: the more frequent, the higher the overhead and larger the **perf.data** size.

To really cut down overhead and generate more advanced summaries, write BPF programs executed by perf. See the eBPF section.

1.4.7 - Usage Examples

These example sequences have been chosen to illustrate some different ways that **perf** is used, from gathering to reporting.

Performance counter summaries, including IPC, for the **gzip** command:

```
# perf stat gzip largefile
```

Count all scheduler process events for 5 seconds, and count by tracepoint:

```
# perf stat -e 'sched:sched_process_*' -a sleep 5
```

Trace all scheduler process events for 5 seconds, and count by both tracepoint and process name:

```
# perf record -e 'sched:sched_process_*' -a sleep 5
# perf report
```

Trace all scheduler process events for 5 seconds, and dump per-event details:

```
# perf record -e 'sched:sched_process_*' -a sleep 5
# perf script
```

Trace read() syscalls, when requested bytes is less than 10:

```
# perf record -e 'syscalls:sys_enter_read' --filter 'count < 10' -a
```

Sample CPU stacks at 99 Hertz, for 5 seconds:

```
# perf record -F 99 -ag -- sleep 5
# perf report
```

Dynamically instrument the kernel tcp_sendmsg() function, and trace it for 5 seconds, with stack traces:

```
# perf probe --add tcp_sendmsg
# perf record -e probe:tcp_sendmsg -ag -- sleep 5
# perf probe --del tcp_sendmsg
# perf report
```

Deleting the tracepoint (--del) wasn't necessary; I included it to show how to return the system to its original state.

Caveats

The use of **`-p PID`** as a filter doesn't work properly on some kernel versions: **`perf`** hits 100% CPU and needs to be killed. It's annoying. The workaround is to profile all CPUs (**`-a`**), and filter PIDs later.

1.4.8 - Special Usage

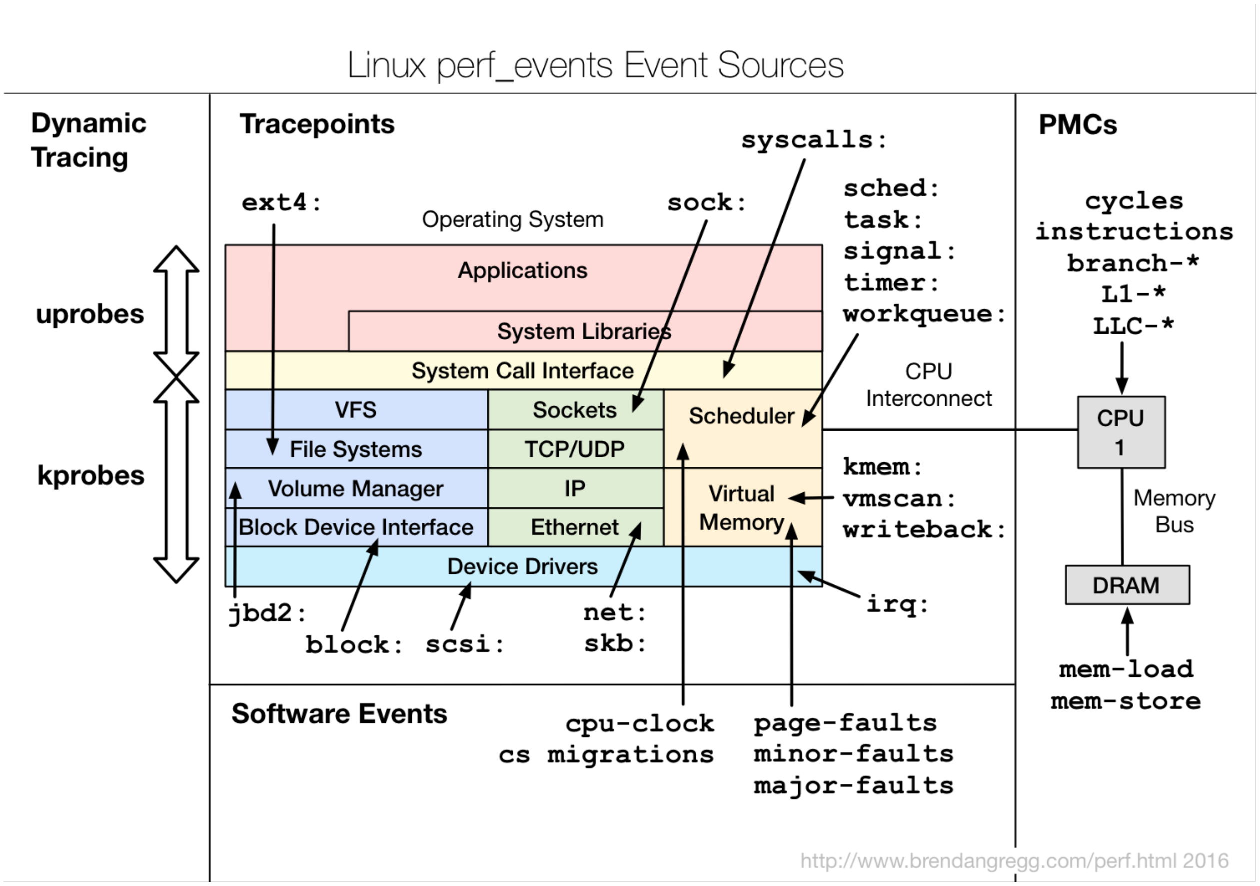
There's a number of subcommands that provide special purpose functionality. These include:

- **perf c2c** (Linux 4.10+): cache-2-cache and cacheline false sharing analysis.
- **perf kmem**: kernel memory allocation analysis.
- **perf kvm**: KVM virtual guest analysis.
- **perf lock**: lock analysis.
- **perf mem**: memory access analysis.
- **perf sched**: kernel scheduler statistics. Examples.

These make use of perf's existing instrumentation capabilities, recording selected events and reporting them in custom ways.

1.5 - Events

perf_events instruments "events", which are a unified interface for different kernel instrumentation frameworks. The following map (from my [SCaLE13x talk]) illustrates the event sources:



The types of events are:

- **Hardware Events**: These instrument low-level processor activity based on CPU performance counters. For example, CPU cycles, instructions retired, memory stall cycles, level 2 cache misses, etc. Some will be listed as Hardware Cache Events.

- **Software Events**: These are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, etc.

- **Kernel Tracepoint Events**: This are static kernel-level instrumentation points that are hardcoded in interesting and logical places in the kernel.

- **User Statically-Defined Tracing (USDT)**: These are static tracepoints for user-level programs and applications.

- **Dynamic Tracing**: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the kprobes framework. For user-level software, uprobes.

- **Timed Profiling**: Snapshots can be collected at an arbitrary frequency, using perf record -F*Hz*. This is commonly used for CPU usage profiling, and works by creating custom timed interrupt events.

Details about the events can be collected, including timestamps, the code path that led to it, and other specific details. The capabilities of **`perf_events`** are enormous, and you're likely to only ever use a fraction.

Currently available events can be listed using the **`list`** subcommand:

```
# perf list

List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                              [Hardware event]
  instructions                                      [Hardware event]
  cache-references                                  [Hardware event]
  cache-misses                                      [Hardware event]
  branch-instructions OR branches                   [Hardware event]
  branch-misses                                     [Hardware event]
  bus-cycles                                        [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend   [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend     [Hardware event]
  ref-cycles                                        [Hardware event]
  cpu-clock                                         [Software event]
  task-clock                                        [Software event]
  page-faults OR faults                             [Software event]
  context-switches OR cs                            [Software event]
  cpu-migrations OR migrations                      [Software event]
  minor-faults                                      [Software event]
  major-faults                                      [Software event]
  alignment-faults                                  [Software event]
  emulation-faults                                  [Software event]
  L1-dcache-loads                                   [Hardware cache event]
  L1-dcache-load-misses                             [Hardware cache event]
  L1-dcache-stores                                  [Hardware cache event]
[...]
  rNNN                                              [Raw hardware event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier                 [Raw hardware event descriptor]
   (see 'man perf-list' on how to encode it)
  mem:<addr>[:access]                               [Hardware breakpoint]
  probe:tcp_sendmsg                                 [Tracepoint event]
[...]
  sched:sched_process_exec                          [Tracepoint event]
  sched:sched_process_fork                          [Tracepoint event]
```

```
  sched:sched_process_wait                           [Tracepoint event]
  sched:sched_wait_task                              [Tracepoint event]
  sched:sched_process_exit                           [Tracepoint event]
[...]
# perf list | wc -l
     657
```

When you use dynamic tracing, you are extending this list. The **probe:tcp_sendmsg** tracepoint in this list is an example, which I added by instrumenting **tcp_sendmsg()**. Profiling (sampling) events are not listed.

1.5.1 - Hardware Events (PMCs)

perf_events began life as a tool for instrumenting the processor's performance monitoring unit (PMU) hardware counters, also called performance monitoring counters (PMCs), or performance instrumentation counters (PICs). These instrument low-level processor activity, for example, CPU cycles, instructions retired, memory stall cycles, level 2 cache misses, etc. Some will be listed as Hardware Cache Events.

PMCs are documented in the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2* and the *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. There are thousands of different PMCs available.

A typical processor will implement PMCs in the following way: only a few or several can be recorded at the same time, from the many thousands that are available. This is because they are a fixed hardware resource on the processor (a limited number of registers), and are programmed to begin counting the selected events.

For examples of using PMCs, see 6.1. CPU Statistics.

1.5.2 - Tracepoints

Summarizing the tracepoint library names and numbers of tracepoints, on my system:

```
# perf list | awk -F: '/Tracepoint event/ { lib[$1]++ } END { for (l in lib) { printf "  %-16.16s
%d\n", l, lib[l] } }' | sort | column
    alarmtimer      4         i2c             8         page_isolation 1          swiotlb         1
    block           19        iommu           7         pagemap         2         syscalls        614
    btrfs           51        irq             5         power           22        task            2
    cgroup          9         irq_vectors     22        printk          1         thermal         7
    clk             14        jbd2            16        random          15        thermal_power_  2
    cma             2         kmem            12        ras             4         timer           13
    compaction      14        libata          6         raw_syscalls    2         tlb             1
    cpuhp           3         mce             1         rcu             1         udp             1
    dma_fence       8         mdio            1         regmap          15        vmscan          15
```

```
exceptions      2       migrate      2       regulator    7       vsyscall     1
ext4            95      mmc          2       rpm          4       workqueue    4
fib             3       module       5       sched        24      writeback    30
fib6            1       mpx          5       scsi         5       x86_fpu      14
filelock        10      msr          3       sdt_node     1       xen          35
filemap         2       napi         1       signal       2       xfs          495
ftrace          1       net          10      skb          3       xhci-hcd     9
gpio            2       nmi          1       sock         2
huge_memory     4       oom          1       spi          7
```

These include:

- **block**: block device I/O
- **ext3**, **ext4**: file system operations
- **kmem**: kernel memory allocation events
- **random**: kernel random number generator events
- **sched**: CPU scheduler events
- **syscalls**: system call enter and exits
- **task**: task events

It's worth checking the list of tracepoints after every kernel upgrade, to see if any are new. The value of adding them has been debated from time to time, with it wondered if anyone will use them (I will!). There is a balance to aim for: I'd include the smallest number of probes that sufficiently covers common needs, and anything unusual or uncommon can be left to dynamic tracing.

For examples of using tracepoints, see Static Kernel Tracing.

1.5.3 - User-Level Statically Defined Tracing (USDT)

Similar to kernel tracepoints, these are hardcoded (usually by placing macros) in the application source at logical and interesting locations, and presented (event name and arguments) as a stable API. Many applications already include tracepoints, added to support DTrace. However, many of these applications do not compile them in by default on Linux. Often you need to compile the application yourself using a --with-dtrace flag.

For example, compiling USDT events with this version of Node.js:

```
$ sudo apt-get install systemtap-sdt-dev       # adds "dtrace", used by node build
$ wget https://nodejs.org/dist/v4.4.1/node-v4.4.1.tar.gz
$ tar xvf node-v4.4.1.tar.gz
$ cd node-v4.4.1
$ ./configure --with-dtrace
$ make -j 8
```

To check that the resulting node binary has probes included:

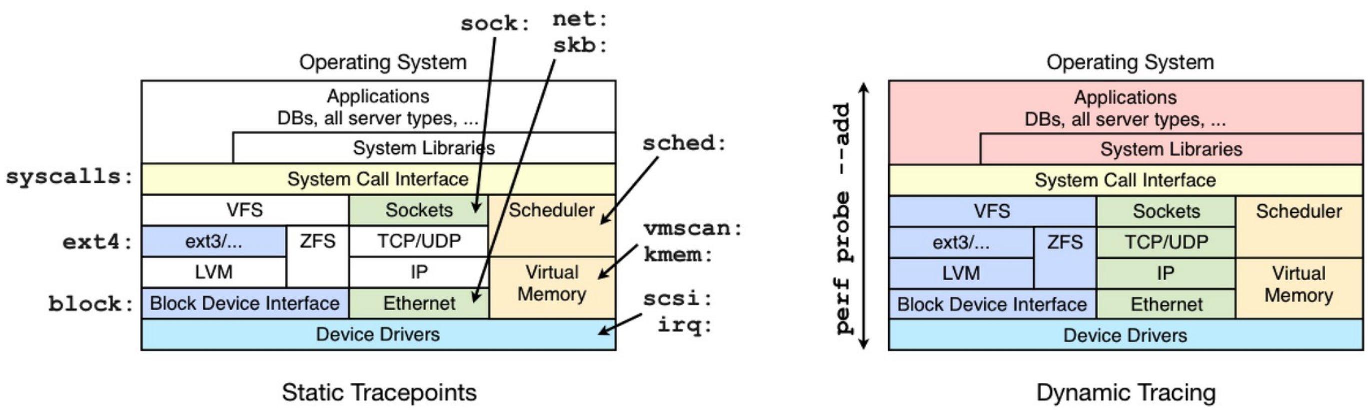
```
$ readelf -n node
```

```
Displaying notes found at file offset 0x00000254 with length 0x00000020:
  Owner                 Data size       Description
  GNU                  0x00000010       NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 2.6.32

Displaying notes found at file offset 0x00000274 with length 0x00000024:
  Owner                 Data size       Description
  GNU                  0x00000014       NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: 1e01659b0aecedadf297b2c56c4a2b536ae2308a

Displaying notes found at file offset 0x00e70994 with length 0x000003c4:
  Owner                 Data size       Description
  stapsdt              0x0000003c       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__start
    Location: 0x0000000000dc14e4, Base: 0x000000000112e064, Semaphore: 0x000000000147095c
    Arguments: 4@%esi 4@%edx 8@%rdi
  stapsdt              0x0000003b       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__done
    Location: 0x0000000000dc14f4, Base: 0x000000000112e064, Semaphore: 0x000000000147095e
    Arguments: 4@%esi 4@%edx 8@%rdi
  stapsdt              0x00000067       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__server__response
    Location: 0x0000000000dc1894, Base: 0x000000000112e064, Semaphore: 0x0000000001470956
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000061       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: net__stream__end
    Location: 0x0000000000dc1c44, Base: 0x000000000112e064, Semaphore: 0x0000000001470952
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000068       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: net__server__connection
    Location: 0x0000000000dc1ff4, Base: 0x000000000112e064, Semaphore: 0x0000000001470950
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000060       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__client__response
    Location: 0x0000000000dc23c5, Base: 0x000000000112e064, Semaphore: 0x000000000147095a
    Arguments: 8@%rdx 8@-1144(%rbp) -4@%eax -4@-1152(%rbp)
  stapsdt              0x00000089       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__client__request
    Location: 0x0000000000dc285e, Base: 0x000000000112e064, Semaphore: 0x0000000001470958
    Arguments: 8@%rax 8@%rdx 8@-2184(%rbp) -4@-2188(%rbp) 8@-2232(%rbp) 8@-2240(%rbp) -4@-2192(%rbp)
  stapsdt              0x00000089       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__server__request
    Location: 0x0000000000dc2e69, Base: 0x000000000112e064, Semaphore: 0x0000000001470954
    Arguments: 8@%r14 8@%rax 8@-4344(%rbp) -4@-4348(%rbp) 8@-4304(%rbp) 8@-4312(%rbp) -4@-4352(%rbp)
```

For examples of using USDT events, see Static User Tracing.

1.5.4 - Dynamic Tracing

The difference between tracepoints and dynamic tracing is shown in the following figure, which illustrates the coverage of common tracepoint libraries:



While dynamic tracing can see everything, it's also an unstable interface since it is instrumenting raw code. That means that any dynamic tracing tools you develop may break after a kernel patch or update. Try to use the static tracepoints first, since their interface should be much more stable. They can also be easier to use and understand, since they have been designed with a tracing end-user in mind.

One benefit of dynamic tracing is that it can be enabled on a live system without restarting anything. You can take an already-running kernel or application and then begin dynamic instrumentation, which (safely) patches instructions in memory to add instrumentation. That means there is zero overhead or tax for this feature until you begin using it. One moment your binary is running unmodified and at full speed, and the next, it's running some extra instrumentation instructions that you dynamically added. Those instructions should eventually be removed once you've finished using your session of dynamic tracing.

The overhead while dynamic tracing is in use, and extra instructions are being executed, is relative to the frequency of instrumented events multiplied by the work done on each instrumentation.

For examples of using dynamic tracing, see [6.5. Dynamic Tracing].

1.6 - Examples

These are some examples of **perf_events**, collected from a variety of 3.x Linux systems.

1.6.1 - CPU Statistics

The **perf stat** command instruments and summarizes key counters. This is from **perf** version 3.5.7.2:

```
# perf stat gzip file1

 Performance counter stats for 'gzip file1':

       1920.159821 task-clock                #    0.991 CPUs utilized
                13 context-switches          #    0.007 K/sec
                 0 CPU-migrations            #    0.000 K/sec
               258 page-faults               #    0.134 K/sec
     5,649,595,479 cycles                    #    2.942 GHz                     [83.43%]
     1,808,339,931 stalled-cycles-frontend   #   32.01% frontend cycles idle    [83.54%]
     1,171,884,577 stalled-cycles-backend    #   20.74% backend  cycles idle    [66.77%]
     8,625,207,199 instructions              #    1.53  insns per cycle
                                             #    0.21  stalled cycles per insn [83.51%]
     1,488,797,176 branches                  #  775.351 M/sec                   [82.58%]
        53,395,139 branch-misses             #    3.59% of all branches         [83.78%]

       1.936842598 seconds time elapsed
```

This includes instructions per cycle (IPC), labled "insns per cycle", or in earlier versions, "IPC". This is a commonly examined metric, either IPC or its invert, CPI. Higher IPC values mean higher instruction throughput, and lower values indicate more stall cycles. I'd generally interpret high IPC values (eg, over 1.0) as good, indicating optimal processing of work. However, I'd want to double check what the instructions are, in case this is due to a spin loop: a high rate of instructions, but a low rate of actual work completed.

There are some advanced metrics now included in perf stat: frontend cycles idle, backend cycles idle, and stalled cycles per insn. To really understand these, you'll need some knowledge of CPU microarchitecture.

CPU Microarchitecture

The frontend and backend metrics refer to the CPU pipeline, and are also based on stall counts. The frontend processes CPU instructions, in order. It involves instruction fetch, along with branch prediction, and decode. The decoded instructions become micro-operations (uops) which the backend processes, and it may do so out of order. For a longer summery of these components, see Shannon Cepeda's great posts on frontend and backend.

The backend can also process multiple uops in parallel; for modern processors, three or four. Along with pipelining, this is how IPC can become greater than one, as more than one instruction can be completed ("retired") per CPU cycle.

Stalled cycles per instruction is similar to IPC (inverted), however, only counting stalled cycles, which will be for memory or resource bus access. This makes it easy to interpret: stalls are latency, reduce stalls. I really like it as a metric, and hope it becomes as commonplace as IPC/CPI. Lets call it SCPI.

Detailed Mode

There is a "detailed" mode for **perf stat**:

```
# perf stat -d gzip file1

 Performance counter stats for 'gzip file1':

       1610.719530 task-clock                #    0.998 CPUs utilized
                20 context-switches          #    0.012 K/sec
                 0 CPU-migrations            #    0.000 K/sec
               258 page-faults               #    0.160 K/sec
     5,491,605,997 cycles                    #    3.409 GHz                     [40.18%]
     1,654,551,151 stalled-cycles-frontend   #   30.13% frontend cycles idle    [40.80%]
     1,025,280,350 stalled-cycles-backend    #   18.67% backend  cycles idle    [40.34%]
     8,644,643,951 instructions              #    1.57  insns per cycle
                                             #    0.19  stalled cycles per insn [50.89%]
     1,492,911,665 branches                  #  926.860 M/sec                   [50.69%]
        53,471,580 branch-misses             #    3.58% of all branches         [51.21%]
     1,938,889,736 L1-dcache-loads           # 1203.741 M/sec                   [49.68%]
       154,380,395 L1-dcache-load-misses     #    7.96% of all L1-dcache hits   [49.66%]
                 0 LLC-loads                 #    0.000 K/sec                   [39.27%]
                 0 LLC-load-misses           #    0.00% of all LL-cache hits    [39.61%]

       1.614165346 seconds time elapsed
```

This includes additional counters for Level 1 data cache events, and last level cache (LLC) events.

Specific Counters

Hardware cache event counters, seen in **perf list**, can be instrumented. Eg:

```
# perf list | grep L1-dcache
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-dcache-store-misses                             [Hardware cache event]
  L1-dcache-prefetches                               [Hardware cache event]
  L1-dcache-prefetch-misses                          [Hardware cache event]
```

```
# perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores gzip file1

 Performance counter stats for 'gzip file1':

     1,947,551,657 L1-dcache-loads

       153,829,652 L1-dcache-misses
         #    7.90% of all L1-dcache hits
     1,171,475,286 L1-dcache-stores


       1.538038091 seconds time elapsed
```

The percentage printed is a convenient calculation that **perf_events** has included, based on the counters I specified. If you include the "cycles" and "instructions" counters, it will include an IPC calculation in the output.

These hardware events that can be measured are often specific to the processor model. Many may not be available from within a virtualized environment.

Raw Counters

The *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2* and the *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors* are full of interesting counters, but most cannot be found in perf list. If you find one you want to instrument, you can specify it as a raw event with the format: **rUUEE**, where **UU** == umask, and **EE** == event number. Here's an example where I've added a couple of raw counters:

```
# perf stat -e cycles,instructions,r80a2,r2b1 gzip file1

 Performance counter stats for 'gzip file1':

     5,586,963,328 cycles                    #    0.000 GHz
     8,608,237,932 instructions              #    1.54  insns per cycle
         9,448,159 raw 0x80a2
    11,855,777,803 raw 0x2b1

       1.588618969 seconds time elapsed
```

If I did this right, then **r80a2** has instrumented RESOURCE_STALLS.OTHER, and **r2b1** has instrumented UOPS_DISPATCHED.CORE: the number of uops dispatched each cycle. It's easy to mess this up, and you'll want to double check that you are on the right page of the manual for your processor.

If you do find an awesome raw counter, please suggest it be added as an alias in perf_events, so we all can find it in perf list.

Other Options

The **perf** subcommands, especially **perf stat**, have an extensive option set which can be listed using "**-h**". I've included the full output for perf stat here from version 3.9.3, not as a reference, but as an illustration of the interface:

```
# perf stat -h

 usage: perf stat [<options>] [<command>]

    -e, --event <event>   event selector. use 'perf list' to list available events
        --filter <filter>
                          event filter
    -i, --no-inherit      child tasks do not inherit counters
    -p, --pid <pid>       stat events on existing process id
    -t, --tid <tid>       stat events on existing thread id
    -a, --all-cpus        system-wide collection from all CPUs
    -g, --group           put the counters into a counter group
    -c, --scale           scale/normalize counters
    -v, --verbose         be more verbose (show counter open errors, etc)
    -r, --repeat <n>      repeat command and print average + stddev (max: 100)
    -n, --null            null run - dont start any counters
    -d, --detailed        detailed run - start a lot of events
    -S, --sync            call sync() before starting a run
    -B, --big-num         print large numbers with thousands' separators
    -C, --cpu <cpu>       list of cpus to monitor in system-wide
    -A, --no-aggr         disable CPU count aggregation
    -x, --field-separator <separator>
                          print counts with custom separator
    -G, --cgroup <name>   monitor event in cgroup name only
    -o, --output <file>   output file name
        --append          append to the output file
        --log-fd <n>      log output to fd, instead of stderr
        --pre <command>   command to run prior to the measured command
        --post <command>  command to run after to the measured command
    -I, --interval-print <n>
                          print counts at regular interval in ms (>= 100)
        --aggr-socket     aggregate counts per processor socket
```

Options such as **--repeat**, **--sync**, **--pre**, and **--post** can be quite useful when doing automated testing or micro-benchmarking.

1.6.2 - Timed Profiling

perf_events can profile CPU usage based on sampling the instruction pointer or stack trace at a fixed interval (timed profiling).

Sampling CPU stacks at 99 Hertz (**-F 99**), for the entire system (**-a**, for all CPUs), with stack traces (**-g**, for call graphs), for 10 seconds:

```
# perf record -F 99 -a -g -- sleep 30
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 3.135 MB perf.data (~136971 samples) ]
# ls -lh perf.data
-rw------- 1 root root 3.2M Jan 26 07:26 perf.data
```

The choice of 99 Hertz, instead of 100 Hertz, is to avoid accidentally sampling in lockstep with some periodic activity, which would produce skewed results. This is also coarse: you may want to increase that to higher rates (eg, up to 997 Hertz) for finer resolution, especially if you are sampling short bursts of activity and you'd still like enough resolution to be useful. Bear in mind that higher frequencies means higher overhead.

The **perf.data** file 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 use **--stdio** in the new version) print the call graph as a tree, annotated with percentages:

```
# perf report --stdio
# ========
# captured on: Mon Jan 26 07:26:40 2014
# hostname : dev2
# os release : 3.8.6-ubuntu-12-opt
# perf version : 3.8.6
# arch : x86_64
# nrcpus online : 8
# nrcpus avail : 8
# cpudesc : Intel(R) Xeon(R) CPU X5675 @ 3.07GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 8182008 kB
# cmdline : /usr/bin/perf record -F 99 -a -g -- sleep 30
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2 = ...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: software = 1, breakpoint = 5
# ========
#
# Samples: 22K of event 'cpu-clock'
# Event count (approx.): 22751
#
# Overhead  Command      Shared Object                           Symbol
# ........  .......  .................  ..............................
#
    94.12%       dd  [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
                 |
                 --- _raw_spin_unlock_irqrestore
                    |
                    |--96.67%-- extract_buf
                    |           extract_entropy_user
```

```
                    |           urandom_read
                    |           vfs_read
                    |           sys_read
                    |           system_call_fastpath
                    |           read
                    |
                    |--1.69%-- account
                    |          |
                    |          |--99.72%-- extract_entropy_user
                    |          |           urandom_read
                    |          |           vfs_read
                    |          |           sys_read
                    |          |           system_call_fastpath
                    |          |           read
                    |           --0.28%-- [...]
                    |
                    |--1.60%-- mix_pool_bytes.constprop.17
[...]
```

This tree starts with the on-CPU functions and works back through the ancestry. This approach is called a "callee based call graph". This can be flipped by using **-G** for an "inverted call graph", or by using the "caller" option to **-g**/**--call-graph**, instead of the "callee" default.

The hottest (most frequent) stack trace in this **perf** call graph occurred in 90.99% of samples, which is the product of the overhead percentage and top stack leaf (94.12% x 96.67%, which are relative rates). perf report can also be run with "**-g graph**" to show absolute overhead rates, in which case "90.99%" is directly displayed on the stack leaf:

```
    94.12%       dd  [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
                 |
                 --- _raw_spin_unlock_irqrestore
                    |
                    |--90.99%-- extract_buf
[...]
```

If user-level stacks look incomplete, you can try perf record with "**-g dwarf**" as a different technique to unwind them. See the Stacks section.

The output from perf report can be many pages long, which can become cumbersome to read. Try generating Flame Graphs from the same data.

1.6.3 - Event Profiling

Apart from sampling at a timed interval, taking samples triggered by CPU hardware counters is another form of CPU profiling, which can be used to shed more light on cache misses, memory stall cycles, and other low-level processor events. The available events can be found using perf list:

```
# perf list | grep Hardware
  cpu-cycles OR cycles                                [Hardware event]
  instructions                                        [Hardware event]
  cache-references                                    [Hardware event]
  cache-misses                                        [Hardware event]
  branch-instructions OR branches                     [Hardware event]
  branch-misses                                       [Hardware event]
  bus-cycles                                          [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend     [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend       [Hardware event]
  ref-cycles                                          [Hardware event]
  L1-dcache-loads                                     [Hardware cache event]
  L1-dcache-load-misses                               [Hardware cache event]
  L1-dcache-stores                                    [Hardware cache event]
  L1-dcache-store-misses                              [Hardware cache event]
[...]
```

For many of these, gathering a stack on every occurrence would induce far too much overhead, and would slow down the system and change the performance characteristics of the target. It's usually sufficient to only instrument a small fraction of their occurrences, rather than all of them. This can be done by specifying a threshold for triggering event collection, using "**-c**" and a count.

For example, the following one-liner instruments Level 1 data cache load misses, collecting a stack trace for one in every 10,000 occurrences:

```
# perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5
```

The mechanics of "**-c count**" are implemented by the processor, which only interrupts the kernel when the threshold has been reached.

See the earlier Raw Counters section for an example of specifying a custom counter, and the next section about skew.

Skew and PEBS

There's a problem with event profiling that you don't really encounter with CPU profiling (timed sampling). With timed sampling, it doesn't matter if there was a small sub-microsecond delay between the interrupt and reading the instruction pointer (IP). Some CPU profilers introduce this jitter on purpose, as another way to avoid lockstep sampling. But for event profiling, it does matter: if you're trying to capture the IP on some PMC event, and there's a delay between the PMC overflow and capturing the IP, then the IP will point to the wrong address. This is skew. Another contributing problem is that micro-ops are processed in parallel and out-of-order, while the instruction pointer points to the resumption instruction, not the instruction that caused the event. I've talked about this before.

The solution is "precise sampling", which on Intel is PEBS (Precise Event-Based Sampling), and on AMD it is IBS (Instruction-Based Sampling). These use CPU hardware support to capture the real state of the CPU at the time of the event. perf can use precise sampling by adding a :p modifier to the PMC event name, eg, "-e instructions:p". The more p's, the more accurate. Here are the docs from tools/perf/Documentation/perf-list.txt:

```
The 'p' modifier can be used for specifying how precise the instruction
address should be. The 'p' modifier can be specified multiple times:

 0 - SAMPLE_IP can have arbitrary skid
 1 - SAMPLE_IP must have constant skid
 2 - SAMPLE_IP requested to have 0 skid
 3 - SAMPLE_IP must have 0 skid
```

In some cases, perf will default to using precise sampling without you needing to specify it. Run "**perf record -vv ...**" to see the value of "**precise_ip**". Also note that only some PMCs support PEBS.

If PEBS isn't working at all for you, check **dmesg**:

```
# dmesg | grep -i pebs
[    0.387014] Performance Events: PEBS fmt1+, SandyBridge events, 16-deep LBR, full-width
counters, Intel PMU driver.
[    0.387034] core: PEBS disabled due to CPU errata, please upgrade microcode
```

The fix (on Intel):

```
# apt-get install -y intel-microcode
[...]
intel-microcode: microcode will be updated at next boot
Processing triggers for initramfs-tools (0.125ubuntu5) ...
update-initramfs: Generating /boot/initrd.img-4.8.0-41-generic
# reboot

(system reboots)

# dmesg | grep -i pebs
[    0.386596] Performance Events: PEBS fmt1+, SandyBridge events, 16-deep LBR, full-width
counters, Intel PMU driver.
#
```

XXX: Need to cover more PEBS problems and other caveats.

1.6.4 - Static Kernel Tracing

The following examples demonstrate static tracing: the instrumentation of tracepoints and other static events.

Counting Syscalls

The following simple one-liner counts system calls for the executed command, and prints a summary (of non-zero counts):

```
# perf stat -e 'syscalls:sys_enter_*' gzip file1 2>&1 | awk '$1 != 0'

 Performance counter stats for 'gzip file1':

                 1 syscalls:sys_enter_utimensat
                 1 syscalls:sys_enter_unlink
                 5 syscalls:sys_enter_newfstat
             1,603 syscalls:sys_enter_read
             3,201 syscalls:sys_enter_write
                 5 syscalls:sys_enter_access
                 1 syscalls:sys_enter_fchmod
                 1 syscalls:sys_enter_fchown
                 6 syscalls:sys_enter_open
                 9 syscalls:sys_enter_close
                 8 syscalls:sys_enter_mprotect
                 1 syscalls:sys_enter_brk
                 1 syscalls:sys_enter_munmap
                 1 syscalls:sys_enter_set_robust_list
                 1 syscalls:sys_enter_futex
                 1 syscalls:sys_enter_getrlimit
                 5 syscalls:sys_enter_rt_sigprocmask
                14 syscalls:sys_enter_rt_sigaction
                 1 syscalls:sys_enter_exit_group
                 1 syscalls:sys_enter_set_tid_address
                14 syscalls:sys_enter_mmap

       1.543990940 seconds time elapsed
```

In this case, a **gzip** command was analyzed. The report shows that there were 3,201 write() syscalls, and half that number of read() syscalls. Many of the other syscalls will be due to process and library initialization.

A similar report can be seen using **strace -c**, the system call tracer, however it may induce much higher overhead than perf, as perf buffers data in-kernel.

perf vs strace

To explain the difference a little further: the current implementation of **strace** uses ptrace(2) to attach to the target process and stop it during system calls, like a debugger. This is violent, and can cause serious overhead. To demonstrate this, the following syscall-heavy program was run by itself, with perf, and with strace. I've only included the line of output that shows its performance:

```
# dd if=/dev/zero of=/dev/null bs=512 count=10000k
5242880000 bytes (5.2 GB) copied, 3.53031 s, 1.5 GB/s

# perf stat -e 'syscalls:sys_enter_*' dd if=/dev/zero of=/dev/null bs=512 
count=10000k
5242880000 bytes (5.2 GB) copied, 9.14225 s, 573 MB/s

# strace -c dd if=/dev/zero of=/dev/null bs=512 count=10000k
5242880000 bytes (5.2 GB) copied, 218.915 s, 23.9 MB/s
```

With perf, the program ran 2.5x slower. But **with strace, it ran 62x slower**. That's likely to be a worst-case result: if syscalls are not so frequent, the difference between the tools will not be as great.

Recent version of perf have included a trace subcommand, to provide some similar functionality to strace, but with much lower overhead.

New Processes

Tracing new processes triggered by a "**`man ls`**":

```
# perf record -e sched:sched_process_exec -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.064 MB perf.data (~2788 samples) ]
# perf report -n --sort comm --stdio
[...]
# Overhead       Samples  Command
# ........  ............  .......
#
    11.11%             1    troff
    11.11%             1      tbl
    11.11%             1  preconv
    11.11%             1    pager
    11.11%             1    nroff
    11.11%             1      man
    11.11%             1   locale
    11.11%             1   grotty
    11.11%             1    groff
```

Nine different commands were executed, each once. I used **`-n`** to print the "Samples" column, and "**`--sort comm`**" to customize the remaining columns.

This works by tracing **`sched:sched_process_exec`**, when a process runs **`exec()`** to execute a different binary. This is often how new processes are created, but not always. An application may **`fork()`** to create a pool of worker processes, but not **`exec()`** a different binary. An application may also reexec: call **`exec()`** again, on itself, usually to clean up its address space. In that case, it's will be seen by this exec tracepoint, but it's not a new process.

The **`sched:sched_process_fork`** tracepoint can be traced to only catch new processes, created via **`fork()`**. The downside is that the process identified is the parent, not the new target, as the new process has yet to **`exec()`** it's final program.

Outbound Connections

There can be times when it's useful to double check what network connections are initiated by a server, from which processes, and why. You might be surprised. These connections can be important to understand, as they can be a source of latency.

For this example, I have a completely idle ubuntu server, and while tracing I'll login to it using ssh. I'm going to trace outbound connections via the **connect()** syscall. Given that I'm performing an *inbound* connection over SSH, will there be any outbound connections at all?

```
# perf record -e syscalls:sys_enter_connect -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.057 MB perf.data (~2489 samples) ]
# perf report --stdio
# ========
# captured on: Tue Jan 28 10:53:38 2014
# hostname : ubuntu
# os release : 3.5.0-23-generic
# perf version : 3.5.7.2
# arch : x86_64
# nrcpus online : 2
# nrcpus avail : 2
# cpudesc : Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz
# cpuid : GenuineIntel,6,58,9
# total memory : 1011932 kB
# cmdline : /usr/bin/perf_3.5.0-23 record -e syscalls:sys_enter_connect -a
# event : name = syscalls:sys_enter_connect, type = 2, config = 0x38b, ...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# ========
#
# Samples: 21  of event 'syscalls:sys_enter_connect'
# Event count (approx.): 21
#
# Overhead  Command       Shared Object                       Symbol
# ........  .......  ..................  ..........................
#
    52.38%     sshd  libc-2.15.so        [.] __GI___connect_internal
    19.05%   groups  libc-2.15.so        [.] __GI___connect_internal
     9.52%     sshd  libpthread-2.15.so  [.] __connect_internal
     9.52%     mesg  libc-2.15.so        [.] __GI___connect_internal
     9.52%     bash  libc-2.15.so        [.] __GI___connect_internal
```

The report shows that sshd, groups, mesg, and bash are all performing **connect()** syscalls. Ring a bell?

The stack traces that led to the **connect()** can explain why:

```
# perf record -e syscalls:sys_enter_connect -ag
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.057 MB perf.data (~2499 samples) ]
# perf report --stdio
[...]
    55.00%     sshd  libc-2.15.so        [.] __GI___connect_internal
               |
               --- __GI___connect_internal
                  |
                  |--27.27%-- add_one_listen_addr.isra.0
                  |
                  |--27.27%-- __nscd_get_mapping
                  |           __nscd_get_map_ref
                  |
```

```
                  |--27.27%-- __nscd_open_socket
                   --18.18%-- [...]
    20.00%   groups  libc-2.15.so        [.] __GI___connect_internal
             |
             --- __GI___connect_internal
                |
                |--50.00%-- __nscd_get_mapping
                |           __nscd_get_map_ref
                |
                 --50.00%-- __nscd_open_socket
    10.00%     mesg  libc-2.15.so        [.] __GI___connect_internal
               |
               --- __GI___connect_internal
                  |
                  |--50.00%-- __nscd_get_mapping
                  |           __nscd_get_map_ref
                  |
                   --50.00%-- __nscd_open_socket
    10.00%     bash  libc-2.15.so        [.] __GI___connect_internal
               |
               --- __GI___connect_internal
                  |
                  |--50.00%-- __nscd_get_mapping
                  |           __nscd_get_map_ref
                  |
                   --50.00%-- __nscd_open_socket
     5.00%     sshd  libpthread-2.15.so  [.] __connect_internal
               |
               --- __connect_internal
```

Ah, these are **nscd** calls: the name service cache daemon. If you see hexadecimal numbers and not function names, you will need to install debug info: see the earlier section on Symbols. These **nscd** calls are likely triggered by calling **getaddrinfo()**, which server software may be using to resolve IP addresses for logging, or for matching hostnames in config files. Browsing the stack traces should identify why.

For sshd, this was called via **add_one_listen_addr()**: a name that was only visible after adding the **openssh-server-dbgsym** package. Unfortunately, the stack trace doesn't continue after **add_one_listen_add()**. I can browse the OpenSSH code to figure out the reasons we're calling into **add_one_listen_add()**, or, I can get the stack traces to work. See the earlier section on Stack Traces.

I took a quick look at the OpenSSH code, and it looks like this code-path is due to parsing ListenAddress from the **sshd_config** file, which can contain either an IP address or a hostname.

Socket Buffers

Tracing the consumption of socket buffers, and the stack traces, is one way to identify what is leading to socket or network I/O.

```
# perf record -e 'skb:consume_skb' -ag
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.065 MB perf.data (~2851 samples) ]
# perf report
[...]
    74.42%  swapper  [kernel.kallsyms]  [k] consume_skb
            |
            --- consume_skb
                arp_process
                arp_rcv
                __netif_receive_skb_core
                __netif_receive_skb
                netif_receive_skb
                virtnet_poll
                net_rx_action
                __do_softirq
                irq_exit
                do_IRQ
                ret_from_intr
                default_idle
                cpu_idle
                start_secondary

    25.58%     sshd  [kernel.kallsyms]  [k] consume_skb
               |
               --- consume_skb
                   dev_kfree_skb_any
                   free_old_xmit_skbs.isra.24
                   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
                   __tcp_push_pending_frames
                   tcp_sendmsg
                   inet_sendmsg
                   sock_aio_write
                   do_sync_write
                   vfs_write
                   sys_write
                   system_call_fastpath
                   __write_nocancel
```

The swapper stack shows the network receive path, triggered by an interrupt. The **sshd** path shows writes.

1.6.5 - Static User Tracing

Support was added in later 4.x series kernels. The following demonstrates Linux 4.10 (with an additional patchset), and tracing the Node.js USDT probes:

```
# perf buildid-cache --add `which node`
# perf list | grep sdt_node
  sdt_node:gc__done                                  [SDT event]
  sdt_node:gc__start                                 [SDT event]
  sdt_node:http__client__request                     [SDT event]
  sdt_node:http__client__response                    [SDT event]
  sdt_node:http__server__request                     [SDT event]
  sdt_node:http__server__response                    [SDT event]
  sdt_node:net__server__connection                   [SDT event]
  sdt_node:net__stream__end                          [SDT event]
# perf record -e sdt_node:http__server__request -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.446 MB perf.data (3 samples) ]
# perf script
            node  7646 [002]   361.012364: sdt_node:http__server__request: (dc2e69)
            node  7646 [002]   361.204718: sdt_node:http__server__request: (dc2e69)
            node  7646 [002]   361.363043: sdt_node:http__server__request: (dc2e69)
```

XXX fill me in, including how to use arguments.

If you are on an older kernel, say, Linux 4.4-4.9, you can probably get these to work with adjustments (I've even hacked them up with ftrace for older kernels), but since they have been in development, I haven't seen documentation outside of lkml, so you'll need to figure it out. (On this kernel range, you might find more documentation for tracing these with bcc/eBPF, including using the **trace.py** tool.)

1.6.6 - Dynamic Tracing

For kernel analysis, I'm using **CONFIG_KPROBES=y** and **CONFIG_KPROBE_EVENTS=y**, to enable kernel dynamic tracing, and **CONFIG_FRAME_POINTER=y**, for frame pointer-based kernel stacks. For user-level analysis, **CONFIG_UPROBES=y** and **CONFIG_UPROBE_EVENTS=y**, for user-level dynamic tracing.

Kernel: tcp_sendmsg()

This example shows instrumenting the kernel tcp_sendmsg() function on the Linux 3.9.3 kernel:

```
# perf probe --add tcp_sendmsg
Failed to find path of kernel module.
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg)
```

```
You can now use it in all perf tools, such as:

        perf record -e probe:tcp_sendmsg -aR sleep 1
```

This adds a new tracepoint event. It suggests using the **-R** option, to collect raw sample records, which is already the default for tracepoints. Tracing this event for 5 seconds, recording stack traces:

```
# perf record -e probe:tcp_sendmsg -a -g -- sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.228 MB perf.data (~9974 samples) ]
```

And the report:

```
# perf report --stdio
# ========
# captured on: Fri Jan 31 20:10:14 2014
# hostname : pgbackup
# os release : 3.9.3-ubuntu-12-opt
# perf version : 3.9.3
# arch : x86_64
# nrcpus online : 8
# nrcpus avail : 8
# cpudesc : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
# cpuid : GenuineIntel,6,45,7
# total memory : 8179104 kB
# cmdline : /lib/modules/3.9.3/build/tools/perf/perf record -e probe:tcp_sendmsg
-a -g -- sleep 5
# event : name = probe:tcp_sendmsg, type = 2, config = 0x3b2, config1 = 0x0,
config2 = 0x0, ...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: software = 1, tracepoint = 2, breakpoint = 5
# ========
#
# Samples: 12  of event 'probe:tcp_sendmsg'
# Event count (approx.): 12
#
# Overhead  Command      Shared Object           Symbol
# ........  .......  .................  ...............
#
   100.00%     sshd  [kernel.kallsyms]  [k] tcp_sendmsg
               |
               --- tcp_sendmsg
                   sock_aio_write
                   do_sync_write
                   vfs_write
                   sys_write
                   system_call_fastpath
                   __write_nocancel
                  |
                  |--8.33%-- 0x50f00000001b810
                   --91.67%-- [...]
```

This shows the path from the **write()** system call to **tcp_sendmsg()**.

You can delete these dynamic tracepoints if you want after use, using **perf probe --del**.

Kernel: tcp_sendmsg() with size

If your kernel has debuginfo (**CONFIG_DEBUG_INFO=y**), you can fish out kernel variables from functions. This is a simple example of examining a size_t (integer), on Linux 3.13.1.

Listing variables available for **tcp_sendmsg()**:

```
# perf probe -V tcp_sendmsg
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                size_t  size
                struct kiocb*   iocb
                struct msghdr*  msg
                struct sock*    sk
```

Creating a probe for **tcp_sendmsg()** with the "size" variable:

```
# perf probe --add 'tcp_sendmsg size'
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with size)

You can now use it in all perf tools, such as:

        perf record -e probe:tcp_sendmsg -aR sleep 1
```

Tracing this probe:

```
# perf record -e probe:tcp_sendmsg -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.052 MB perf.data (~2252 samples) ]
# perf script
# ========
# captured on: Fri Jan 31 23:49:55 2014
# hostname : dev1
# os release : 3.13.1-ubuntu-12-opt
# perf version : 3.13.1
# arch : x86_64
# nrcpus online : 2
# nrcpus avail : 2
# cpudesc : Intel(R) Xeon(R) CPU E5645 @ 2.40GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 1796024 kB
# cmdline : /usr/bin/perf record -e probe:tcp_sendmsg -a
# event : name = probe:tcp_sendmsg, type = 2, config = 0x1dd, config1 = 0x0, config2 = ...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: software = 1, tracepoint = 2, breakpoint = 5
# ========
#
            sshd  1301 [001]   502.424719: probe:tcp_sendmsg: (ffffffff81505d80) size=b0
```

```
            sshd  1301 [001]   502.424814: probe:tcp_sendmsg: (ffffffff81505d80) size=40
            sshd  2371 [000]   502.952590: probe:tcp_sendmsg: (ffffffff81505d80) size=27
            sshd  2372 [000]   503.025023: probe:tcp_sendmsg: (ffffffff81505d80) size=3c0
            sshd  2372 [001]   503.203776: probe:tcp_sendmsg: (ffffffff81505d80) size=98
            sshd  2372 [001]   503.281312: probe:tcp_sendmsg: (ffffffff81505d80) size=2d0
            sshd  2372 [001]   503.461358: probe:tcp_sendmsg: (ffffffff81505d80) size=30
            sshd  2372 [001]   503.670239: probe:tcp_sendmsg: (ffffffff81505d80) size=40
            sshd  2372 [001]   503.742565: probe:tcp_sendmsg: (ffffffff81505d80) size=140
            sshd  2372 [001]   503.822005: probe:tcp_sendmsg: (ffffffff81505d80) size=20
            sshd  2371 [000]   504.118728: probe:tcp_sendmsg: (ffffffff81505d80) size=30
            sshd  2371 [000]   504.192575: probe:tcp_sendmsg: (ffffffff81505d80) size=70
[...]
```

The size is shown as hexadecimal.

Kernel: tcp_sendmsg() line number and local variable

With debuginfo, perf_events can create tracepoints for lines within kernel functions. Listing available line probes for **`tcp_sendmsg()`**:

```
# perf probe -L tcp_sendmsg
<tcp_sendmsg@/mnt/src/linux-3.14.5/net/ipv4/tcp.c:0>
      0  int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                        size_t size)
      2  {
                struct iovec *iov;
                struct tcp_sock *tp = tcp_sk(sk);
                struct sk_buff *skb;
      6         int iovlen, flags, err, copied = 0;
      7         int mss_now = 0, size_goal, copied_syn = 0, offset = 0;
                bool sg;
                long timeo;
[...]
     79                 while (seglen > 0) {
                                int copy = 0;
     81                         int max = size_goal;

                                skb = tcp_write_queue_tail(sk);
     84                         if (tcp_send_head(sk)) {
     85                                 if (skb->ip_summed == CHECKSUM_NONE)
                                                max = mss_now;
     87                                 copy = max - skb->len;
                                }

     90                         if (copy <= 0) {
         new_segment:
[...]
```

This is Linux 3.14.5; your kernel version may look different. Lets check what variables are available on line 81:

```
# perf probe -V tcp_sendmsg:81
Available variables at tcp_sendmsg:81
        @<tcp_sendmsg+537>
                bool    sg
                int     copied
                int     copied_syn
                int     flags
                int     mss_now
                int     offset
                int     size_goal
                long int        timeo
                size_t  seglen
                struct iovec*   iov
                struct sock*    sk
                unsigned char*  from
```

Now lets trace line 81, with the seglen variable that is checked in the loop:

```
# perf probe --add 'tcp_sendmsg:81 seglen'
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg:81 with seglen)

You can now use it in all perf tools, such as:

        perf record -e probe:tcp_sendmsg -aR sleep 1

# perf record -e probe:tcp_sendmsg -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.188 MB perf.data (~8200 samples) ]
# perf script
            sshd  4652 [001] 2082360.931086: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0x80
   app_plugin.pl  2400 [001] 2082360.970489: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0x20
        postgres  2422 [000] 2082360.970703: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0x52
   app_plugin.pl  2400 [000] 2082360.970890: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0x7b
        postgres  2422 [001] 2082360.971099: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0xb
   app_plugin.pl  2400 [000] 2082360.971140: probe:tcp_sendmsg: (ffffffff81642ca9) seglen=0x55
[...]
```

This is pretty amazing. Remember that you can also include in-kernel filtering using **`--filter`**, to match only the data you want.

User: malloc()

While this is an interesting example, I want to say right off the bat that **`malloc()`** calls are very frequent, so you will need to consider the overheads of tracing calls like this.

Adding a libc **`malloc()`** probe:

```
# perf probe -x /lib/x86_64-linux-gnu/libc-2.15.so --add malloc
Added new event:
  probe_libc:malloc    (on 0x82f20)

You can now use it in all perf tools, such as:
```

```
        perf record -e probe_libc:malloc -aR sleep 1
```

Tracing it system-wide:

```
# perf record -e probe_libc:malloc -a
^C[ perf record: Woken up 12 times to write data ]
[ perf record: Captured and wrote 3.522 MB perf.data (~153866 samples) ]
```

The report:

```
# perf report -n
[...]
# Samples: 45K of event 'probe_libc:malloc'
# Event count (approx.): 45158
#
# Overhead       Samples          Command  Shared Object      Symbol
# ........  ............  ...............  .............  ..........
#
    42.72%         19292       apt-config  libc-2.15.so   [.] malloc
    19.71%          8902             grep  libc-2.15.so   [.] malloc
     7.88%          3557             sshd  libc-2.15.so   [.] malloc
     6.25%          2824              sed  libc-2.15.so   [.] malloc
     6.06%          2738            which  libc-2.15.so   [.] malloc
     4.12%          1862  update-motd-upd  libc-2.15.so   [.] malloc
     3.72%          1680             stat  libc-2.15.so   [.] malloc
     1.68%           758            login  libc-2.15.so   [.] malloc
     1.21%           546        run-parts  libc-2.15.so   [.] malloc
     1.21%           545               ls  libc-2.15.so   [.] malloc
     0.80%           360        dircolors  libc-2.15.so   [.] malloc
     0.56%           252               tr  libc-2.15.so   [.] malloc
     0.54%           242              top  libc-2.15.so   [.] malloc
     0.49%           222       irqbalance  libc-2.15.so   [.] malloc
     0.44%           200             dpkg  libc-2.15.so   [.] malloc
     0.38%           173          lesspipe  libc-2.15.so   [.] malloc
     0.29%           130  update-motd-fsc  libc-2.15.so   [.] malloc
     0.25%           112            uname  libc-2.15.so   [.] malloc
     0.24%           108              cut  libc-2.15.so   [.] malloc
     0.23%           104           groups  libc-2.15.so   [.] malloc
     0.21%            94  release-upgrade  libc-2.15.so   [.] malloc
     0.18%            82        00-header  libc-2.15.so   [.] malloc
     0.14%            62             mesg  libc-2.15.so   [.] malloc
     0.09%            42  update-motd-reb  libc-2.15.so   [.] malloc
     0.09%            40             date  libc-2.15.so   [.] malloc
     0.08%            35             bash  libc-2.15.so   [.] malloc
     0.08%            35         basename  libc-2.15.so   [.] malloc
     0.08%            34          dirname  libc-2.15.so   [.] malloc
     0.06%            29               sh  libc-2.15.so   [.] malloc
     0.06%            26        99-footer  libc-2.15.so   [.] malloc
     0.05%            24              cat  libc-2.15.so   [.] malloc
     0.04%            18             expr  libc-2.15.so   [.] malloc
     0.04%            17         rsyslogd  libc-2.15.so   [.] malloc
     0.03%            12             stty  libc-2.15.so   [.] malloc
     0.00%             1             cron  libc-2.15.so   [.] malloc
```

This shows the most malloc() calls were by apt-config, while I was tracing.

User: malloc() with size

As of the Linux 3.13.1 kernel, this is not supported yet:

```
# perf probe -x /lib/x86_64-linux-gnu/libc-2.15.so --add 'malloc size'
Debuginfo-analysis is not yet supported with -x/--exec option.
  Error: Failed to add events. (-38)
```

As a workaround, you can access the registers (on Linux 3.7+). For example, on x86_64:

```
# perf probe -x /lib64/libc-2.17.so '--add=malloc size=%di'
       probe_libc:malloc    (on 0x800c0 with size=%di)
```

These registers ("**%di**" etc) are dependent on your processor architecture. To figure out which ones to use, see the X86 calling conventions on Wikipedia, or page 24 of the AMD64 ABI (PDF). (Thanks Jose E. Nunez for digging out these references.)

1.6.7 - Scheduler Analysis

The **perf sched** subcommand provides a number of tools for analyzing kernel CPU scheduler behavior. You can use this to identify and quantify issues of scheduler latency.

The current overhead of this tool (as of up to Linux 4.10) may be noticeable, as it instruments and dumps scheduler events to the perf.data file for later analysis. For example:

```
# perf sched record -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.886 MB perf.data (13502 samples) ]
```

That's 1.9 Mbytes for one second, including 13,502 samples. The size and rate will be relative to your workload and number of CPUs (this example is an 8 CPU server running a software build). How this is written to the file system has been optimized: it only woke up one time to read the event buffers and write them to disk, which greatly reduces overhead. That said, there are still significant overheads with instrumenting all scheduler events and writing event data to the file system. These events:

```
# perf script --header
# ========
# captured on: Sun Feb 26 19:40:00 2017
# hostname : bgregg-xenial
# os release : 4.10-virtual
# perf version : 4.10
# arch : x86_64
# nrcpus online : 8
# nrcpus avail : 8
# cpudesc : Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
# cpuid : GenuineIntel,6,62,4
# total memory : 15401700 kB
# cmdline : /usr/bin/perf sched record -- sleep 1
```

```
# event : name = sched:sched_switch, , id = { 2752, 2753, 2754, 2755, 2756, 2757, 2758, 2759 }, type = 2, size = 11...
# event : name = sched:sched_stat_wait, , id = { 2760, 2761, 2762, 2763, 2764, 2765, 2766, 2767 }, type = 2, size =...
# event : name = sched:sched_stat_sleep, , id = { 2768, 2769, 2770, 2771, 2772, 2773, 2774, 2775 }, type = 2, size ...
# event : name = sched:sched_stat_iowait, , id = { 2776, 2777, 2778, 2779, 2780, 2781, 2782, 2783 }, type = 2, size...
# event : name = sched:sched_stat_runtime, , id = { 2784, 2785, 2786, 2787, 2788, 2789, 2790, 2791 }, type = 2, siz...
# event : name = sched:sched_process_fork, , id = { 2792, 2793, 2794, 2795, 2796, 2797, 2798, 2799 }, type = 2, siz...
# event : name = sched:sched_wakeup, , id = { 2800, 2801, 2802, 2803, 2804, 2805, 2806, 2807 }, type = 2, size = 11...
# event : name = sched:sched_wakeup_new, , id = { 2808, 2809, 2810, 2811, 2812, 2813, 2814, 2815 }, type = 2, size ...
# event : name = sched:sched_migrate_task, , id = { 2816, 2817, 2818, 2819, 2820, 2821, 2822, 2823 }, type = 2, siz...
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: breakpoint = 5, power = 7, software = 1, tracepoint = 2, msr = 6
# HEADER_CACHE info available, use -I to display
# missing features: HEADER_BRANCH_STACK HEADER_GROUP_DESC HEADER_AUXTRACE HEADER_STAT
# ========
#
    perf 16984 [005] 991962.879966:       sched:sched_wakeup: comm=perf pid=16999 prio=120 target_cpu=005
[...]
```

If overhead is a problem, you can use my eBPF/bcc Tools including runqlat and runqlen which use in-kernel summaries of scheduler events, reducing overhead further. An advantage of **perf sched** dumping all events is that you aren't limited to the summary. If you caught an intermittent event, you can analyze those recorded events in custom ways until you understood the issue, rather than needing to catch it a second time.

The captured trace file can be reported in a number of ways, summarized by the help message:

```
# perf sched -h

 Usage: perf sched [] {record|latency|map|replay|script|timehist}

    -D, --dump-raw-trace  dump raw trace in ASCII
    -f, --force           don't complain, do it
    -i, --input     input file name
    -v, --verbose         be more verbose (show symbol address, etc)
```

perf sched latency will summarize scheduler latencies by task, including average and maximum delay:

```
# perf sched latency

 -----------------------------------------------------------------------------------------------------------------
  Task                  |   Runtime ms  | Switches | Average delay ms | Maximum delay ms | Maximum delay at       |
 -----------------------------------------------------------------------------------------------------------------
  cat:(6)               |     12.002 ms |        6 | avg:   17.541 ms | max:   29.702 ms | max at: 991962.948070 s
  ar:17043              |      3.191 ms |        1 | avg:   13.638 ms | max:   13.638 ms | max at: 991963.048070 s
  rm:(10)               |     20.955 ms |       10 | avg:   11.212 ms | max:   19.598 ms | max at: 991963.404069 s
  objdump:(6)           |     35.870 ms |        8 | avg:   10.969 ms | max:   16.509 ms | max at: 991963.424443 s
  :17008:17008          |    462.213 ms |       50 | avg:   10.464 ms | max:   35.999 ms | max at: 991963.120069 s
  grep:(7)              |     21.655 ms |       11 | avg:    9.465 ms | max:   24.502 ms | max at: 991963.464082 s
  fixdep:(6)            |     81.066 ms |        8 | avg:    9.023 ms | max:   19.521 ms | max at: 991963.120068 s
  mv:(10)               |     30.249 ms |       14 | avg:    8.380 ms | max:   21.688 ms | max at: 991963.200073 s
  ld:(3)                |     14.353 ms |        6 | avg:    7.376 ms | max:   15.498 ms | max at: 991963.452070 s
  recordmcount:(7)      |     14.629 ms |        9 | avg:    7.155 ms | max:   18.964 ms | max at: 991963.292100 s
  svstat:17067          |      1.862 ms |        1 | avg:    6.142 ms | max:    6.142 ms | max at: 991963.280069 s
  cc1:(21)              |   6013.457 ms |     1138 | avg:    5.305 ms | max:   44.001 ms | max at: 991963.436070 s
  gcc:(18)              |     43.596 ms |       40 | avg:    3.905 ms | max:   26.994 ms | max at: 991963.380069 s
  ps:17073              |     27.158 ms |        4 | avg:    3.751 ms | max:    8.000 ms | max at: 991963.332070 s
[...]
```

To shed some light as to how this is instrumented and calculated, I'll show the events that led to the top event's "Maximum delay at" of 29.702 ms. Here are the raw events from perf sched script:

```
      sh 17028 [001] 991962.918368:   sched:sched_wakeup_new: comm=sh pid=17030 prio=120 target_cpu=002
[...]
     cc1 16819 [002] 991962.948070:       sched:sched_switch: prev_comm=cc1 prev_pid=16819 prev_prio=120
                                                         prev_state=R ==> next_comm=sh next_pid=17030 next_prio=120
[...]
```

The time from the wakeup (991962.918368, which is in seconds) to the context switch (991962.948070) is 29.702 ms. This process is listed as "sh" (shell) in the raw events, but execs "cat" soon after, so is shown as "cat" in the perf sched latency output.

perf sched map shows all CPUs and context-switch events, with columns representing what each CPU was doing and when. It's the kind of data you see visualized in scheduler analysis GUIs (including perf timechart, with the layout rotated 90 degrees). Example output:

```
# perf sched map
                      *A0           991962.879971 secs A0 => perf:16999
                       A0     *B0   991962.880070 secs B0 => cc1:16863
          *C0          A0      B0   991962.880070 secs C0 => :17023:17023
  *D0      C0          A0      B0   991962.880078 secs D0 => ksoftirqd/0:6
   D0      C0 *E0      A0      B0   991962.880081 secs E0 => ksoftirqd/3:28
   D0      C0 *F0      A0      B0   991962.880093 secs F0 => :17022:17022
  *G0      C0  F0      A0      B0   991962.880108 secs G0 => :17016:17016
   G0      C0  F0     *H0      B0   991962.880256 secs H0 => migration/5:39
   G0      C0  F0     *I0      B0   991962.880276 secs I0 => perf:16984
   G0      C0  F0     *J0      B0   991962.880687 secs J0 => cc1:16996
   G0      C0 *K0      J0      B0   991962.881839 secs K0 => cc1:16945
   G0      C0  K0      J0 *L0  B0   991962.881841 secs L0 => :17020:17020
   G0      C0  K0      J0 *M0  B0   991962.882289 secs M0 => make:16637
   G0      C0  K0      J0 *N0  B0   991962.883102 secs N0 => make:16545
   G0     *O0  K0      J0  N0  B0   991962.883880 secs O0 => cc1:16819
   G0 *A0  O0  K0      J0  N0  B0   991962.884069 secs
   G0  A0  O0  K0 *P0  J0  N0  B0   991962.884076 secs P0 => rcu_sched:7
   G0  A0  O0  K0 *Q0  J0  N0  B0   991962.884084 secs Q0 => cc1:16831
   G0  A0  O0  K0  Q0  J0 *R0  B0   991962.884843 secs R0 => cc1:16825
   G0 *S0  O0  K0  Q0  J0  R0  B0   991962.885636 secs S0 => cc1:16900
   G0  S0  O0 *T0  Q0  J0  R0  B0   991962.886893 secs T0 => :17014:17014
   G0  S0  O0 *K0  Q0  J0  R0  B0   991962.886917 secs
[...]
```

This is an 8 CPU system, and you can see the 8 columns for each CPU starting from the left. Some CPU columns begin blank, as we've yet to trace an event on that CPU at the start of the profile. They quickly become populated.

The two character codes you see ("A0", "C0") are identifiers for tasks, which are mapped on the right ("=>"). This is more compact than using process (task) IDs. The "*" shows which CPU had the context switch event, and the new event that was running. For example, the very last line shows that at 991962.886917 (seconds) CPU 4 context-switched to K0 (a "cc1" process, PID 16945).

That example was from a busy system. Here's an idle system:

```
# perf sched map
                      *A0           993552.887633 secs A0 => perf:26596
  *.                   A0           993552.887781 secs .  => swapper:0
   .                  *B0           993552.887843 secs B0 => migration/5:39
```

```
   .                  *.            993552.887858 secs
   .                   .   *A0      993552.887861 secs
   .                  *C0   A0      993552.887903 secs C0 => bash:26622
   .                  *.    A0      993552.888020 secs
   .          *D0      .    A0      993552.888074 secs D0 => rcu_sched:7
   .          *.       .    A0      993552.888082 secs
   .           .      *C0   A0      993552.888143 secs
   .      *.   .       C0   A0      993552.888173 secs
   .       .   .      *B0   A0      993552.888439 secs
   .       .   .      *.    A0      993552.888454 secs
   .      *C0  .       .    A0      993552.888457 secs
   .       C0  .       .   *.       993552.889257 secs
   .      *.   .       .    .       993552.889764 secs
   .       .  *E0      .    .       993552.889767 secs E0 => bash:7902
[...]
```

Idle CPUs are shown as ".".

Remember to examine the timestamp column to make sense of this visualization (GUIs use that as a dimension, which is easier to comprehend, but here the numbers are just listed). It's also only showing context switch events, and not scheduler latency. The newer timehist command has a visualization (-V) that can include wakeup events.

`perf sched timehist` was added in Linux 4.10, and shows the scheduler latency by event, including the time the task was waiting to be woken up (wait time) and the scheduler latency after wakeup to running (sch delay). It's the scheduler latency that we're more interested in tuning. Example output:

```
# perf sched timehist
Samples do not have callchains.
           time    cpu  task name                       wait time  sch delay   run time
                        [tid/pid]                          (msec)     (msec)     (msec)
--------------- ------  ------------------------------  ---------  ---------  ---------
  991962.879971 [0005]  perf[16984]                         0.000      0.000      0.000
  991962.880070 [0007]  :17008[17008]                       0.000      0.000      0.000
  991962.880070 [0002]  cc1[16880]                          0.000      0.000      0.000
  991962.880078 [0000]  cc1[16881]                          0.000      0.000      0.000
  991962.880081 [0003]  cc1[16945]                          0.000      0.000      0.000
  991962.880093 [0003]  ksoftirqd/3[28]                     0.000      0.007      0.012
  991962.880108 [0000]  ksoftirqd/0[6]                      0.000      0.007      0.030
  991962.880256 [0005]  perf[16999]                         0.000      0.005      0.285
  991962.880276 [0005]  migration/5[39]                     0.000      0.007      0.019
  991962.880687 [0005]  perf[16984]                         0.304      0.000      0.411
  991962.881839 [0003]  cat[17022]                          0.000      0.000      1.746
  991962.881841 [0006]  cc1[16825]                          0.000      0.000      0.000
[...]
  991963.885740 [0001]  :17008[17008]                      25.613      0.000      0.057
  991963.886009 [0001]  sleep[16999]                     1000.104      0.006      0.269
  991963.886018 [0005]  cc1[17083]                         19.998      0.000      9.948
```

This output includes the sleep command run to set the duration of perf itself to one second. Note that sleep's wait time is 1000.104 milliseconds because I had run "**`sleep 1`**": that's the time it was asleep waiting its timer wakeup event. Its scheduler latency was only 0.006 milliseconds, and its time on-CPU was 0.269 milliseconds.

There are a number of options to timehist, including **-V** to add a CPU visualization column, **-M** to add migration events, and **-w** for wakeup events. For example:

```
# perf sched timehist -MVw
Samples do not have callchains.
           time    cpu  012345678  task name           wait time  sch delay   run time
                                   [tid/pid]              (msec)     (msec)     (msec)
--------------- ------  ---------  ------------------  ---------  ---------  ---------
  991962.879966 [0005]             perf[16984]                                          awakened: perf[16999]
  991962.879971 [0005]       s     perf[16984]             0.000      0.000      0.000
  991962.880070 [0007]         s   :17008[17008]           0.000      0.000      0.000
  991962.880070 [0002]    s        cc1[16880]              0.000      0.000      0.000
  991962.880071 [0000]             cc1[16881]                                           awakened: ksoftirqd/0[6]
  991962.880073 [0003]             cc1[16945]                                           awakened: ksoftirqd/3[28]
  991962.880078 [0000]  s          cc1[16881]              0.000      0.000      0.000
  991962.880081 [0003]     s       cc1[16945]              0.000      0.000      0.000
  991962.880093 [0003]     s       ksoftirqd/3[28]         0.000      0.007      0.012
  991962.880108 [0000]  s          ksoftirqd/0[6]          0.000      0.007      0.030
  991962.880249 [0005]             perf[16999]                                          awakened: migration/5[39]
  991962.880256 [0005]       s     perf[16999]             0.000      0.005      0.285
  991962.880264 [0005]        m      migration/5[39]                                      migrated: perf[16999] cpu 5 => 1
  991962.880276 [0005]       s     migration/5[39]         0.000      0.007      0.019
  991962.880682 [0005]        m      perf[16984]                                          migrated: cc1[16996] cpu 0 => 5
  991962.880687 [0005]       s     perf[16984]             0.304      0.000      0.411
  991962.881834 [0003]             cat[17022]                                           awakened: :17020
[...]
  991963.885734 [0001]             :17008[17008]                                        awakened: sleep[16999]
  991963.885740 [0001]   s         :17008[17008]          25.613      0.000      0.057
  991963.886005 [0001]             sleep[16999]                                         awakened: perf[16984]
  991963.886009 [0001]   s         sleep[16999]         1000.104      0.006      0.269
  991963.886018 [0005]       s     cc1[17083]             19.998      0.000      9.948
```

The CPU visualization column ("012345678") has "s" for context-switch events, and "m" for migration events, showing the CPU of the event. If you run perf sched record -g, then the stack traces are appended on the right in a single line (not shown here).

The last events in that output include those related to the "sleep 1" command used to time perf. The wakeup happened at 991963.885734, and at 991963.885740 (6 microseconds later) CPU 1 begins to context-switch to the sleep process. The column for that event still shows ":17008[17008]" for what was on-CPU, but the target of the context switch (sleep) is not shown. It is in the raw events:

```
  :17008 17008 [001] 991963.885740:       sched:sched_switch: prev_comm=cc1 prev_pid=17008 prev_prio=120
                                                  prev_state=R ==> next_comm=sleep next_pid=16999 next_prio=120
```

The 991963.886005 event shows that the perf command received a wakeup while sleep was running (almost certainly sleep waking up its parent process because it terminated), and then we have the context switch on 991963.886009 where sleep stops running, and a summary is printed out: 1000.104 ms waiting (the "sleep 1"), with 0.006 ms scheduler latency, and 0.269 ms of CPU runtime.

Here I've decorated the timehist output with the details of the context switch destination in red:

```
  991963.885734 [0001]             :17008[17008]                                        awakened: sleep[16999]
  991963.885740 [0001]   s         :17008[17008]          25.613      0.000      0.057  next: sleep[16999]
  991963.886005 [0001]             sleep[16999]                                         awakened: perf[16984]
  991963.886009 [0001]   s         sleep[16999]         1000.104      0.006      0.269  next: cc1[17008]
  991963.886018 [0005]       s     cc1[17083]             19.998      0.000      9.948  next: perf[16984]
```

When sleep finished, a waiting "cc1" process then executed. perf ran on the following context switch, and is the last event in the profile (perf terminated). I've added this as a -n/--next option to perf (should arrive in Linux 4.11 or 4.12).

perf sched script dumps all events (similar to perf script):

```
# perf sched script

    perf 16984 [005] 991962.879960: sched:sched_stat_runtime: comm=perf pid=16984 runtime=3901506 [ns] vruntime=165...
    perf 16984 [005] 991962.879966:       sched:sched_wakeup: comm=perf pid=16999 prio=120 target_cpu=005
    perf 16984 [005] 991962.879971:       sched:sched_switch: prev_comm=perf prev_pid=16984 prev_prio=120 prev_stat...
    perf 16999 [005] 991962.880058: sched:sched_stat_runtime: comm=perf pid=16999 runtime=98309 [ns] vruntime=16405...
     cc1 16881 [000] 991962.880058: sched:sched_stat_runtime: comm=cc1 pid=16881 runtime=3999231 [ns] vruntime=7897...
  :17024 17024 [004] 991962.880058: sched:sched_stat_runtime: comm=cc1 pid=17024 runtime=3866637 [ns] vruntime=7810...
     cc1 16900 [001] 991962.880058: sched:sched_stat_runtime: comm=cc1 pid=16900 runtime=3006028 [ns] vruntime=7772...
     cc1 16825 [006] 991962.880058: sched:sched_stat_runtime: comm=cc1 pid=16825 runtime=3999423 [ns] vruntime=7876...
```

Each of these events ("**sched:sched_stat_runtime**" etc) are tracepoints you can instrument directly using perf record.

As I've shown earlier, this raw output can be useful for digging further than the summary commands.

perf sched replay will take the recorded scheduler events, and then simulate the workload by spawning threads with similar runtimes and context switches. Useful for testing and developing scheduler changes and configuration. Don't put too much faith in this (and other) workload replayers: they can be a useful load generator, but it's difficult to simulate the real workload completely. Here I'm running replay with -r -1, to repeat the workload:

```
# perf sched replay -r -1
run measurement overhead: 84 nsecs
sleep measurement overhead: 146710 nsecs
the run test took 1000005 nsecs
the sleep test took 1107773 nsecs
nr_run_events:        4175
nr_sleep_events:      4710
nr_wakeup_events:     2138
task      0 (             swapper:         0), nr_events: 13
task      1 (             swapper:         1), nr_events: 1
task      2 (             swapper:         2), nr_events: 1
task      3 (            kthreadd:         4), nr_events: 1
task      4 (            kthreadd:         6), nr_events: 29
[...]
task    530 (                  sh:     17145), nr_events: 4
task    531 (                  sh:     17146), nr_events: 7
task    532 (                  sh:     17147), nr_events: 4
task    533 (                make:     17148), nr_events: 10
task    534 (                  sh:     17149), nr_events: 1
------------------------------------------------------------
#1  : 965.996, ravg: 966.00, cpu: 798.24 / 798.24
#2  : 902.647, ravg: 966.00, cpu: 1157.53 / 798.24
#3  : 945.482, ravg: 966.00, cpu: 925.25 / 798.24
#4  : 943.541, ravg: 966.00, cpu: 761.72 / 798.24
#5  : 914.643, ravg: 966.00, cpu: 1604.32 / 798.24
[...]
```

1.6.8 - eBPF

As of Linux 4.4, perf has some enhanced BPF support (aka eBPF or just "BPF"), with more in later kernels. BPF makes perf tracing programmatic, and takes perf from being a counting & sampling-with-post-processing tracer, to a fully in-kernel programmable tracer.

eBPF is currently a little restricted and difficult to use from perf. It's getting better all the time. A different and currently easier way to access eBPF is via the bcc Python interface, which is described on my eBPF Tools page. On this page, I'll discuss perf.

Prerequisites

Linux 4.4 at least. Newer versions have more perf/BPF features, so the newer the better. Also clang (eg, apt-get install clang).

kmem_cache_alloc from Example

This program traces the kernel kmem_cache_alloc() function, only if its calling function matches a specified range, filtered in kernel context. You can imagine doing this for efficiency: instead of tracing all allocations, which can be very frequent and add significant overhead, you filter for just a range of kernel calling functions of interest, such as a kernel module. I'll loosely match tcp functions as an example, which are in memory at these addresses:

```
# grep tcp /proc/kallsyms | more
[...]
ffffffff817c1bb0 t tcp_get_info_chrono_stats
ffffffff817c1c60 T tcp_init_sock
ffffffff817c1e30 t tcp_splice_data_recv
ffffffff817c1e70 t tcp_push
ffffffff817c20a0 t tcp_send_mss
ffffffff817c2170 t tcp_recv_skb
ffffffff817c2250 t tcp_cleanup_rbuf
[...]
ffffffff818524f0 T tcp6_proc_exit
ffffffff81852510 T tcpv6_exit
ffffffff818648a0 t tcp6_gro_complete
ffffffff81864910 t tcp6_gro_receive
ffffffff81864ae0 t tcp6_gso_segment
ffffffff8187bd89 t tcp_v4_inbound_md5_hash
```

I'll assume these functions are contiguous, so that by tracing the range **0xffffffff817c1bb0** to **0xffffffff8187bd89**, I'm matching much of tcp.

Here is my BPF program, **kca_from.c**:

```
#include <uapi/linux/bpf.h>
#include <uapi/linux/ptrace.h>
```

```
#define SEC(NAME) __attribute__((section(NAME), used))

/*
 * Edit the following to match the instruction address range you want to
 * sample. Eg, look in /proc/kallsyms. The addresses will change for each
 * kernel version and build.
 */
#define RANGE_START  0xffffffff817c1bb0
#define RANGE_END    0xffffffff8187bd89

struct bpf_map_def {
        unsigned int type;
        unsigned int key_size;
        unsigned int value_size;
        unsigned int max_entries;
};

static int (*probe_read)(void *dst, int size, void *src) =
    (void *)BPF_FUNC_probe_read;
static int (*get_smp_processor_id)(void) =
    (void *)BPF_FUNC_get_smp_processor_id;
static int (*perf_event_output)(void *, struct bpf_map_def *, int, void *,
    unsigned long) = (void *)BPF_FUNC_perf_event_output;

struct bpf_map_def SEC("maps") channel = {
        .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        .key_size = sizeof(int),
        .value_size = sizeof(u32),
        .max_entries = __NR_CPUS__,
};

SEC("func=kmem_cache_alloc")
int func(struct pt_regs *ctx)
{
        u64 ret = 0;
        // x86_64 specific:
        probe_read(&ret, sizeof(ret), (void *)(ctx->bp+8));
        if (ret >= RANGE_START && ret < RANGE_END) {
                perf_event_output(ctx, &channel, get_smp_processor_id(),
                    &ret, sizeof(ret));
        }
        return 0;
}

char _license[] SEC("license") = "GPL";
int _version SEC("version") = LINUX_VERSION_CODE;
```

Now I'll execute it, then dump the events:

```
# perf record -e bpf-output/no-inherit,name=evt/ -e ./kca_from.c/map:channel.event=evt/ -a -- sleep 1
bpf: builtin compilation failed: -95, try external compiler
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.214 MB perf.data (3 samples) ]
```

```
# perf script
 testserver00001 14337 [003] 481432.395181:          0     evt:  ffffffff81210f51 kmem_cache_alloc (/lib/modules/...)
      BPF output: 0000: 0f b4 7c 81 ff ff ff ff  ..|.....
                  0008: 00 00 00 00              ....

    redis-server  1871 [005] 481432.395258:          0     evt:  ffffffff81210f51 kmem_cache_alloc (/lib/modules/...)
      BPF output: 0000: 14 55 7c 81 ff ff ff ff  .U|.....
                  0008: 00 00 00 00              ....

    redis-server  1871 [005] 481432.395456:          0     evt:  ffffffff81210f51 kmem_cache_alloc (/lib/modules/...)
      BPF output: 0000: fe dc 7d 81 ff ff ff ff  ..}.....
                  0008: 00 00 00 00              ....
```

It worked: the "BPF output" records contain addresses in our range: **0xffffffff817cb40f**, and so on. **kmem_cache_alloc()** is a frequently called function, so that it only matched a few entries in one second of tracing is an indication it is working (I can also relax that range to confirm it).

Adding stack traces with **-g**:

```
# perf record -e bpf-output/no-inherit,name=evt/ -e ./kca_from.c/map:channel.event=evt/ -a -g -- sleep 
1
bpf: builtin compilation failed: -95, try external compiler
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.215 MB perf.data (3 samples) ]

# perf script
testserver00001 16744 [002] 481518.262579:          0                evt:
                  410f51 kmem_cache_alloc (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9cb40f tcp_conn_request (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9da243 tcp_v4_conn_request (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d0936 tcp_rcv_state_process (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9db102 tcp_v4_do_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9dcabf tcp_v4_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b4af4 ip_local_deliver_finish (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b4dff ip_local_deliver (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b477b ip_rcv_finish (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b50fb ip_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  97119e __netif_receive_skb_core (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  971708 __netif_receive_skb (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9725df process_backlog (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  971c8e net_rx_action (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8e58d __do_softirq (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8c9ac do_softirq_own_stack (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  28a061 do_softirq.part.18 (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  28a0ed __local_bh_enable_ip (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b8ff3 ip_finish_output2 (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b9f43 ip_finish_output (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba9f6 ip_output (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba155 ip_local_out (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba48a ip_queue_xmit (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d3823 tcp_transmit_skb (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d5345 tcp_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9da764 tcp_v4_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9f1abc __inet_stream_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9f1d38 inet_stream_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  952fd9 SYSC_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  953c1e sys_connect (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8b9fb entry_SYSCALL_64_fastpath (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                   10800 __GI___libc_connect (/lib/x86_64-linux-gnu/libpthread-2.23.so)

      BPF output: 0000: 0f b4 7c 81 ff ff ff ff  ..|.....
                  0008: 00 00 00 00              ....

redis-server  1871 [003] 481518.262670:          0                evt:
                  410f51 kmem_cache_alloc (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
```

```
                  9c5514 tcp_poll (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9515ba sock_poll (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  485699 sys_epoll_ctl (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8b9fb entry_SYSCALL_64_fastpath (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  106dca epoll_ctl (/lib/x86_64-linux-gnu/libc-2.23.so)

      BPF output: 0000: 14 55 7c 81 ff ff ff ff  .U|.....
                  0008: 00 00 00 00              ....

redis-server  1871 [003] 481518.262870:          0                 evt:
                  410f51 kmem_cache_alloc (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ddcfe tcp_time_wait (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9cefff tcp_fin (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9cf630 tcp_data_queue (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d0abd tcp_rcv_state_process (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9db102 tcp_v4_do_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9dca8b tcp_v4_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b4af4 ip_local_deliver_finish (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b4dff ip_local_deliver (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b477b ip_rcv_finish (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b50fb ip_rcv (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  97119e __netif_receive_skb_core (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  971708 __netif_receive_skb (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9725df process_backlog (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  971c8e net_rx_action (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8e58d __do_softirq (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8c9ac do_softirq_own_stack (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  28a061 do_softirq.part.18 (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  28a0ed __local_bh_enable_ip (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b8ff3 ip_finish_output2 (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9b9f43 ip_finish_output (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba9f6 ip_output (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba155 ip_local_out (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9ba48a ip_queue_xmit (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d3823 tcp_transmit_skb (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d3e24 tcp_write_xmit (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d4c31 __tcp_push_pending_frames (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9d6881 tcp_send_fin (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9c70b7 tcp_close (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  9f161c inet_release (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  95181f sock_release (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  951892 sock_close (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  43b2f7 __fput (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  43b46e ____fput (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  2a3cfe task_work_run (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  2032ba exit_to_usermode_loop (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  203b29 syscall_return_slowpath (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                  a8ba88 entry_SYSCALL_64_fastpath (/lib/modules/4.10.0-rc8-virtual/build/vmlinux)
                   105cd __GI___libc_close (/lib/x86_64-linux-gnu/libpthread-2.23.so)

      BPF output: 0000: fe dc 7d 81 ff ff ff ff  ..}.....
                  0008: 00 00 00 00              ....
```

This confirms the parent functions that were matched by the range.

1.7 - Visualizations

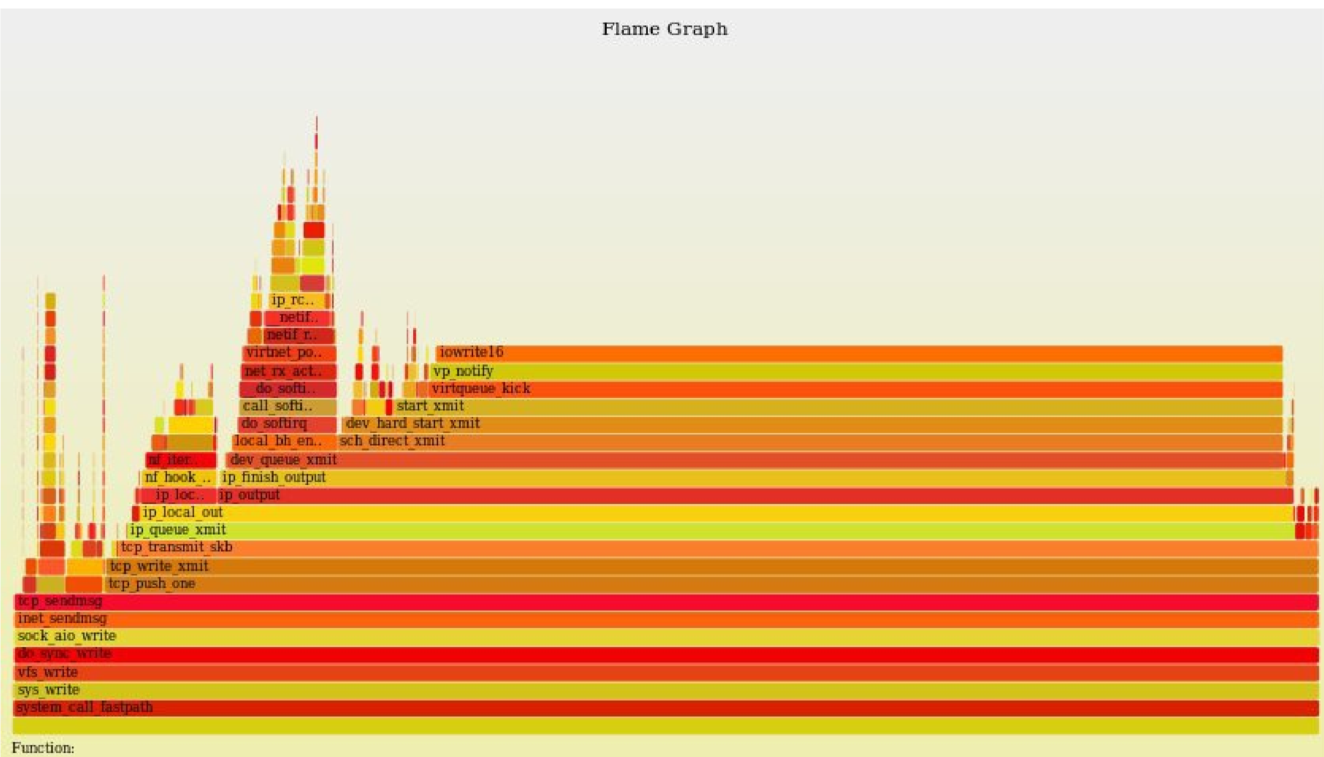
perf_events has a builtin visualization: timecharts, as well as text-style visualization via its text user interface (TUI) and tree reports. The following two sections show visualizations of my own: flame graphs and heat maps. The software I'm using is open source and on github, and produces these from **perf_events** collected data. (It'd be very handy to have these integrated into perf_events directly; if you want to do that, let me know if I can help.)

1.7.1 - Flame Graphs

Flame Graphs can be produced from **perf_events** profiling data using the FlameGraph tools software. This visualizes the same data you see in perf report, and works with any **perf.data** file that was captured with stack traces (**-g**).

Example

This example CPU flame graph shows a network workload for the 3.2.9-1 Linux kernel, running as a KVM instance (SVG, PNG):



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 the CPU Flame Graphs page 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(), which is handled by the KVM hypervisor. This information has been extremely useful for directing KVM performance efforts.

A similar network workload on a bare metal Linux system looks quite different, as networking isn't processed via the virtio-net driver, for a start.

Generation

The example flame graph was generated using **perf_events** and the FlameGraph tools:

```
# git clone https://github.com/brendangregg/FlameGraph  # or download it from github
# cd FlameGraph
# perf record -F 99 -ag -- sleep 60
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# cat out.perf-folded | ./flamegraph.pl > perf-kernel.svg
```

The first **perf** command profiles CPU stacks, as explained earlier. I adjusted the rate to 99 Hertz here; I actually generated the flame graph from a 1000 Hertz profile, but I'd only use that if you had a reason to go faster, which costs more in overhead. The samples are saved in a perf.data file, which can be viewed using **perf report**:

```
# perf report --stdio
[...]
# 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 follows the flame graph when reading it top-down. When using **-g/--call-graph** (for "caller", instead of the "callee" default), it generates a tree that follows the flame graph when read bottom-up. The hottest 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%).

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 dumps 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**).

Piping

A flame graph can be generated directly by piping all the steps:

```
# 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 dozens of 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, such as CPU idle threads.

Filtering

The one-line-per-stack output of **stackcollapse-perf.pl** is also convenient for **grep**(1). Eg:

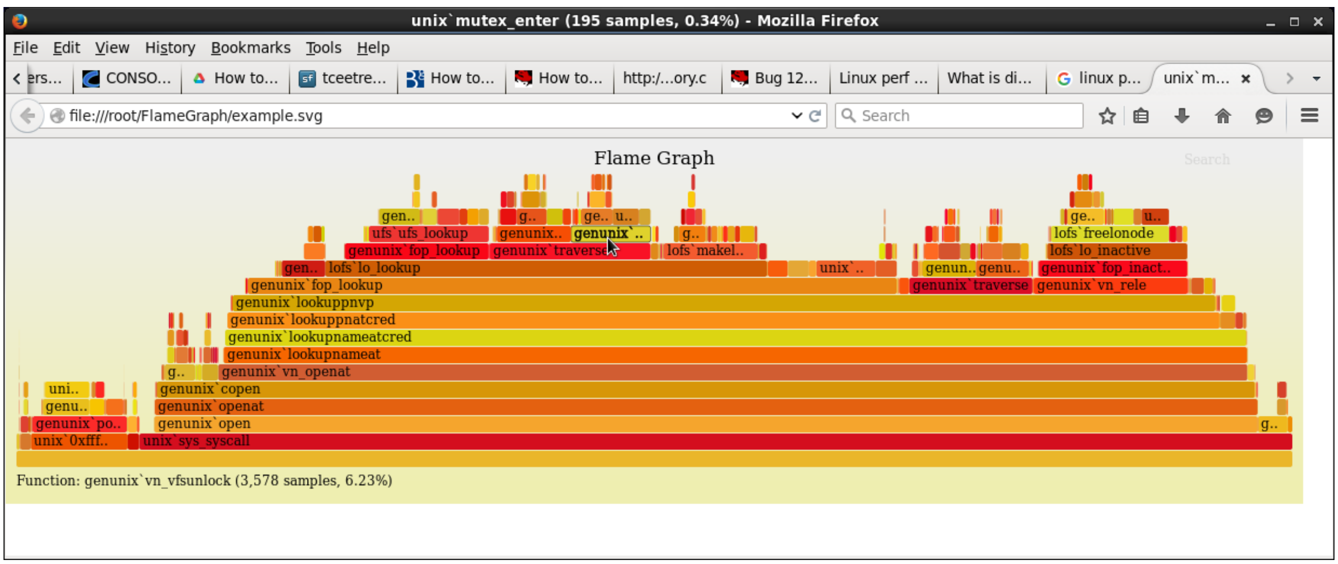
```
# perf script | ./stackcollapse-perf.pl > out.perf-folded
```

```
# 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
```

I frequently elide the **cpu_idle** threads in this way, to focus on the real threads that are consuming CPU resources. If I miss this step, the **cpu_idle** threads can often dominate the flame graph, squeezing the interesting code paths.

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).

There are more examples of **perf_events** CPU flame graphs on the [CPU flame graph] page, including a [summary] of these instructions.

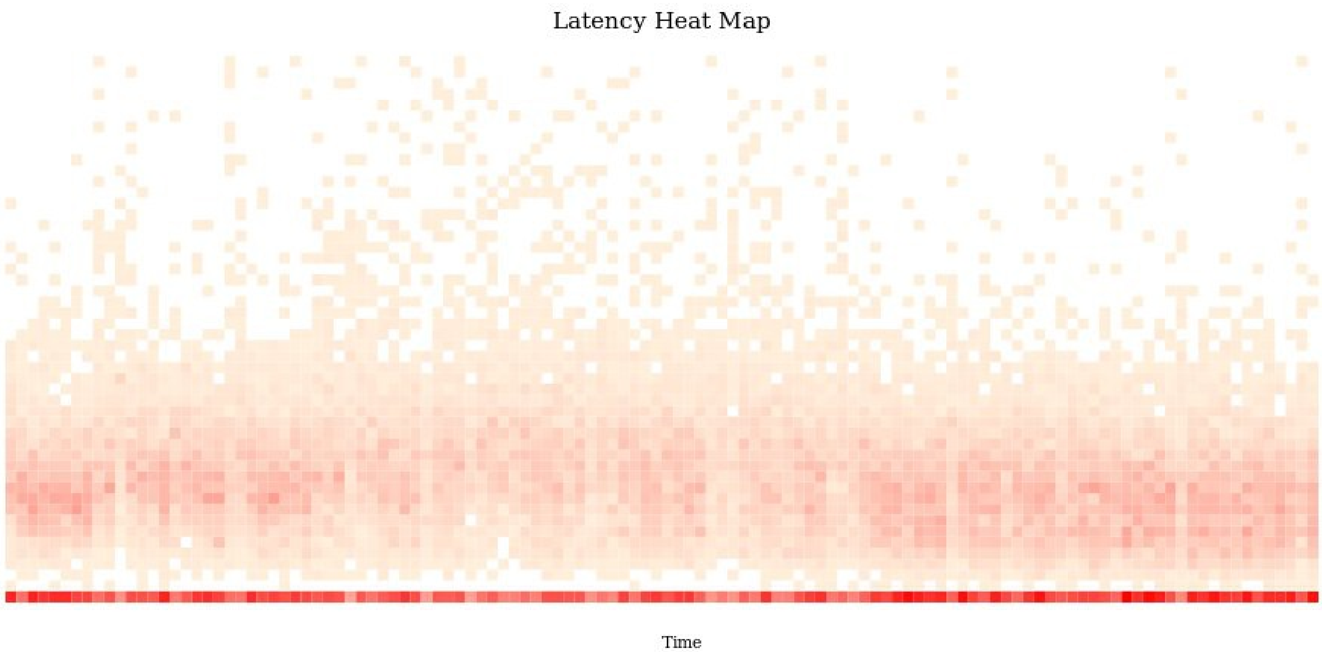


Heat Maps

Since **perf_events** can record high resolution timestamps (microseconds) for events, some latency measurements can be derived from trace data.

Example

The following heat map visualizes disk I/O latency data collected from **perf_events** (SVG, PNG):



Mouse-over blocks to explore the latency distribution over time. The x-axis is the passage of time, the y-axis latency, and the z-axis (color) is the number of I/O at that time and latency range. The distribution is bimodal, with the dark line at the bottom showing that many disk I/O completed with sub-millisecond latency: cache hits. There is a cloud of disk I/O from about 3 ms to 25 ms, which would be caused by random disk I/O (and queueing). Both these modes averaged to the 9 ms we saw earlier.

The following **iostat** output was collected at the same time as the heat map data was collected (shows a typical one second summary):

```
# iostat -x 1
[...]
Device: rrqm/s wrqm/s    r/s   w/s   rkB/s wkB/s avgrq-sz avgqu-sz await r_await w_await svctm  %util
vda       0.00   0.00   0.00  0.00    0.00  0.00     0.00     0.00  0.00    0.00    0.00  0.00   0.00
vdb       0.00   0.00 334.00  0.00 2672.00  0.00    16.00     2.97  9.01    9.01    0.00  2.99 100.00
```

This workload has an average I/O time (await) of 9 milliseconds, which sounds like a fairly random workload on 7200 RPM disks. The problem is that we don't know the distribution from the **iostat** output, or any similar latency average. There could be latency outliers present, which is not visible in the average, and yet are causing problems. The heat map did show I/O up to 50 ms, which you might not have expected from that iostat output. There could also be multiple modes, as we saw in the heat map, which are also not visible in an average.

Gathering

I used **perf_events** to record the block request (disk I/O) issue and completion static tracepoints:

```
# perf record -e block:block_rq_issue -e block:block_rq_complete -a sleep 120
[ perf record: Woken up 36 times to write data ]
[ perf record: Captured and wrote 8.885 MB perf.data (~388174 samples) ]
# perf script
[...]
     randread.pl  2522 [000]  6011.824759: block:block_rq_issue: 254,16 R 0 () 7322849 + 16 [randread.pl]
     randread.pl  2520 [000]  6011.824866: block:block_rq_issue: 254,16 R 0 () 26144801 + 16 [randread.pl]
         swapper     0 [000]  6011.828913: block:block_rq_complete: 254,16 R () 31262577 + 16 [0]
     randread.pl  2521 [000]  6011.828970: block:block_rq_issue: 254,16 R 0 () 70295937 + 16 [randread.pl]
         swapper     0 [000]  6011.835862: block:block_rq_complete: 254,16 R () 26144801 + 16 [0]
     randread.pl  2520 [000]  6011.835932: block:block_rq_issue: 254,16 R 0 () 5495681 + 16 [randread.pl]
         swapper     0 [000]  6011.837988: block:block_rq_complete: 254,16 R () 7322849 + 16 [0]
     randread.pl  2522 [000]  6011.838051: block:block_rq_issue: 254,16 R 0 () 108589633 + 16 [randread.pl]
         swapper     0 [000]  6011.850615: block:block_rq_complete: 254,16 R () 108589633 + 16 [0]
[...]
```

The full output from perf script is about 70,000 lines. I've included some here so that you can see the kind of data available.

Processing

To calculate latency for each I/O, I'll need to pair up the issue and completion events, so that I can calculate the timestamp delta. The columns look straightforward (and are in **include/trace/events/block.h**), with the 4th field the timestamp in seconds (with microsecond resolution), the 6th field the disk device ID (major, minor), and a later field (which varies based on the tracepoint) has the disk offset. I'll use the disk device ID and offset as the unique identifier, assuming the kernel will not issue concurrent I/O to the exact same location.

I'll use **awk** to do these calculations and print the completion times and latency:

```
# perf script | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 } $5 ~
/complete/ { if (l = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000, ($4 - l) *
1000000; ts[$6, $10] = 0 } }' > out.lat_us
# more out.lat_us
6011793689 8437
6011797306 3488
6011798851 1283
6011806422 11248
6011824680 18210
6011824693 21908
[...]
```

I converted both columns to be microseconds, to make the next step easier.

Generation

Now I can use my **trace2heatmap.pl** program (github), to generate the interactive SVG heatmap from the trace data (and uses microseconds by default):

```
# ./trace2heatmap.pl --unitstime=us --unitslat=us --maxlat=50000 out.lat_us > out.svg
```

When I generated the heatmap, I truncated the y scale to 50 ms. You can adjust it to suit your investigation, increasing it to see more of the latency outliers, or decreasing it to reveal more resolution for the lower latencies: for example, with a 250 us limit.

Overheads

While this can be useful to do, be mindful of overheads. In my case, I had a low rate of disk I/O (~300 IOPS), which generated an 8 Mbyte trace file after 2 minutes. If your disk IOPS were 100x that, your trace file will also be 100x, and the overheads for gathering and processing will add up.

For more about latency heatmaps, see my LISA 2010 presentation slides, and my CACM 2010 article, both about heat maps. Also see my Perf Heat Maps blog post.

1.8 - Targets

Notes on specific targets.

Under construction.

1.8.1 - Java

1.8.2 - Node.js

- Node.js V8 JIT internals with annotation support https://twitter.com/brendangregg/status/755838455549001728

1.9 - More

There's more capabilities to **`perf_events`** than I've demonstrated here. I'll add examples of the other subcommands when I get a chance.

Here's a preview of **`perf trace`**, which was added in 3.7, demonstrated on 3.13.1:

```
# perf trace ls
     0.109 ( 0.000 ms):  ... [continued]: read()) = 1
     0.430 ( 0.000 ms):  ... [continued]: execve()) = -2
     0.565 ( 0.051 ms): execve(arg0: 140734989338352, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     0.697 ( 0.051 ms): execve(arg0: 140734989338353, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     0.797 ( 0.046 ms): execve(arg0: 140734989338358, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     0.915 ( 0.045 ms): execve(arg0: 140734989338359, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     1.030 ( 0.044 ms): execve(arg0: 140734989338362, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     1.414 ( 0.311 ms): execve(arg0: 140734989338363, arg1: 140734989358048, arg2: 40612288, arg3: 1407...
     2.156 ( 1.053 ms):  ... [continued]: brk()) = 0xac9000
     2.319 ( 1.215 ms):  ... [continued]: access()) = -1 ENOENT No such file or directory
     2.479 ( 1.376 ms):  ... [continued]: mmap()) = 0xb3a84000
     2.634 ( 0.052 ms): access(arg0: 139967406289504, arg1: 4, arg2: 139967408408688, arg3: 13996740839...
     2.787 ( 0.205 ms):  ... [continued]: open()) = 3
     2.919 ( 0.337 ms):  ... [continued]: fstat()) = 0
     3.049 ( 0.057 ms): mmap(arg0: 0, arg1: 22200, arg2: 1, arg3: 2, arg4: 3, arg5: 0         ) = 0xb3a...
     3.177 ( 0.184 ms):  ... [continued]: close()) = 0
     3.298 ( 0.043 ms): access(arg0: 139967406278152, arg1: 0, arg2: 6, arg3: 7146772199173811245, arg4...
     3.432 ( 0.049 ms): open(arg0: 139967408376811, arg1: 524288, arg2: 0, arg3: 139967408376810, arg4:...
     3.560 ( 0.045 ms): read(arg0: 3, arg1: 140737350651528, arg2: 832, arg3: 139967408376810, arg4: 14...
     3.684 ( 0.042 ms): fstat(arg0: 3, arg1: 140737350651216, arg2: 140737350651216, arg3: 354389249727...
     3.814 ( 0.054 ms): mmap(arg0: 0, arg1: 2221680, arg2: 5, arg3: 2050, arg4: 3, arg5: 0    ) = 0xb36...
[...]
```

An advantage is that this is buffered tracing, which costs much less overhead than strace, as I described earlier. The perf trace output seen from this 3.13.1 kernel does, however, looks suspicious for a number of reasons. I think this is still an in-development feature. It reminds me of my dtruss tool, which has a similar role, before I added code to print each system call in a custom and appropriate way.

1.10 - Building

The steps to build **perf_events** depends on your kernel version and Linux distribution. In summary:

1. Get the Linux kernel source that matches your currently running kernel (eg, from the linux-source package, or kernel.org).
2. Unpack the kernel source.
3. **cd tools/perf**
4. **make**
5. Fix all errors, and most warnings, from (4).

The first error may be that you are missing make, or a compiler (**gcc**). Once you have those, you may then see various warnings about missing libraries, which disable **perf** features. I'd install as many as possible, and take note of the ones you are missing.

These **perf** build warnings are *really helpful*, and are generated by its **Makefile**. Here's the makefile from 3.9.3:

```
# grep found Makefile
msg := $(warning No libelf found, disables 'probe' tool, please install elfutils-libelf-devel/libelf-dev);
msg := $(error No gnu/libc-version.h found, please install glibc-dev[el]/glibc-static);
msg := $(warning No libdw.h found or old libdw.h found or elfutils is older than 0.138, disables dwarf support.
 Please install new elfutils-devel/libdw-dev);
msg := $(warning No libunwind found, disabling post unwind support. Please install libunwind-dev[el] >= 0.99);
msg := $(warning No libaudit.h found, disables 'trace' tool, please install audit-libs-devel or libaudit-dev);
msg := $(warning newt not found, disables TUI support. Please install newt-devel or libnewt-dev);
msg := $(warning GTK2 not found, disables GTK2 support. Please install gtk2-devel or libgtk2.0-dev);
$(if $(1),$(warning No $(1) was found))
msg := $(warning No bfd.h/libbfd found, install binutils-dev[el]/zlib-static to gain symbol demangling)
msg := $(warning No numa.h found, disables 'perf bench numa mem' benchmark, please install numa-libs-devel or
 libnuma-dev);
```

Take the time to read them. This list is likely to grow as new features are added to **perf_events**.

The following notes show what I've specifically done for kernel versions and distributions, in case it is helpful.

Packages: Ubuntu, 3.8.6

Packages required for key functionality: gcc make bison flex elfutils libelf-dev libdw-dev libaudit-dev. You may also consider python-dev (for python scripting) and binutils-dev (for symbol demangling), which are larger packages.

Kernel Config: 3.8.6

Here are some kernel **CONFIG** options for **perf_events** functionality:

```
# for perf_events:
```

```
CONFIG_PERF_EVENTS=y
# for stack traces:
CONFIG_FRAME_POINTER=y
# kernel symbols:
CONFIG_KALLSYMS=y
# tracepoints:
CONFIG_TRACEPOINTS=y
# kernel function trace:
CONFIG_FTRACE=y
# kernel-level dynamic tracing:
CONFIG_KPROBES=y
CONFIG_KPROBE_EVENTS=y
# user-level dynamic tracing:
CONFIG_UPROBES=y
CONFIG_UPROBE_EVENTS=y
# full kernel debug info:
CONFIG_DEBUG_INFO=y
# kernel lock tracing:
CONFIG_LOCKDEP=y
# kernel lock tracing:
CONFIG_LOCK_STAT=y
# kernel dynamic tracepoint variables:
CONFIG_DEBUG_INFO=y
```

You may need to build your own kernel to enable these. The exact set you need depends on your needs and kernel version, and list is likely to grow as new features are added to **`perf_events`**.

1.11 - Troubleshooting

If you see hexadecimal numbers instead of symbols, or have truncated stack traces, see the Prerequisites section.

Here are some rough notes from other issues I've encountered.

This sometimes works (3.5.7.2) and sometimes throws the following error (3.9.3):

```
ubuntu# perf stat -e 'syscalls:sys_enter_*' -a sleep 5
Error:
Too many events are opened.
Try again after reducing the number of events.
```

This can be fixed by increasing the file descriptor limit using **`ulimit -n`**.

Type 3 errors:

```
ubuntu# perf report
0xab7e48 [0x30]: failed to process type: 3
# ========
# captured on: Tue Jan 28 21:08:31 2014
# hostname : pgbackup
# os release : 3.9.3-ubuntu-12-opt
# perf version : 3.9.3
# arch : x86_64
# nrcpus online : 8
# nrcpus avail : 8
# cpudesc : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
# cpuid : GenuineIntel,6,45,7
# total memory : 8179104 kB
# cmdline : /lib/modules/3.9.3-ubuntu-12-opt/build/tools/perf/perf record
 -e sched:sched_process_exec -a
# event : name = sched:sched_process_exec, type = 2, config = 0x125, config1 = 0x0,
 config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0, excl_guest = 1, precise_ip = 0
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: software = 1, tracepoint = 2, breakpoint = 5
# ========
#
Warning: Timestamp below last timeslice flush
```

1.12 - Other Tools

perf_events has the capabilities from many other tools rolled into one: **strace**(1), for tracing system calls, **tcpdump**(8), for tracing network packets, and **blktrace**(1), for tracing block device I/O (disk I/O), and other targets including file system and scheduler events. Tracing all events from one tool is not only convenient, it also allows direct correlations, including timestamps, between different instrumentation sources. Unlike these other tools, some assembly is required, which may not be for everyone (as explained in Audience).

1.13 - Resources

Resources for further study.

1.13.1 - Posts

I've also been writing blog posts on specific **`perf_events`** topics. My suggested reading order is from oldest to newest (top down):

- 22 Jun 2014: perf CPU Sampling
- 29 Jun 2014: perf Static Tracepoints
- 01 Jul 2014: perf Heat Maps
- 03 Jul 2014: perf Counting
- 10 Jul 2014: perf Hacktogram
- 11 Sep 2014: Linux perf Rides the Rocket
- 17 Sep 2014: node.js Flame Graphs on Linux
- 26 Feb 2015: Linux perf_events Off-CPU Time Flame Graph
- 27 Feb 2015: Linux Profiling at Netflix
- 24 Jul 2015: Java Mixed-Mode Flame Graphs (PDF)
- 30 Apr 2016: Linux 4.5 perf folded format

And posts on **ftrace**:

- 13 Jul 2014: Linux ftrace Function Counting
- 16 Jul 2014: iosnoop for Linux
- 23 Jul 2014: Linux iosnoop Latency Heat Maps
- 25 Jul 2014: opensnoop for Linux
- 28 Jul 2014: execsnoop for Linux: See Short-Lived Processes
- 30 Aug 2014: ftrace: The Hidden Light Switch
- 06 Sep 2014: tcpretrans: Tracing TCP retransmits
- 31 Dec 2014: Linux Page Cache Hit Ratio
- 28 Jun 2015: uprobe: User-Level Dynamic Tracing
- 03 Jul 2015: Hacking Linux USDT

1.13.2 - Links

`perf_events`:

- perf-tools (github), a collection of my performance analysis tools based on Linux perf_events and ftrace.
- perf Main Page.
- The excellent perf Tutorial, which focuses more on CPU hardware counters.
- The Unofficial Linux Perf Events Web-Page by Vince Weaver.
- The perf user mailing list.
- Mischa Jonker's presentation Fighting latency: How to optimize your system using perf (PDF) (2013).
- The OMG SO PERF T-shirt (site has coarse language).
- Shannon Cepeda's great posts on pipeline speak: frontend and backend.
- Jiri Olsa's dwarf mode callchain patch.
- Linux kernel source: tools/perf/Documentation/examples.txt.
- Linux kernel source: tools/perf/Documentation/perf-record.txt.
- ... and other documentation under tools/perf/Documentation.
- A good case study for Transparent Hugepages: measuring the performance impact using perf and PMCs.
- Julia Evans created a perf cheatsheet based on my one-liners (2017).

`ftrace`:

- perf-tools (github), a collection of my performance analysis tools based on Linux perf_events and ftrace.
- Linux kernel source: Documentation/trace/ftrace.txt.
- lwn.net Secrets of the Ftrace function tracer, by Steven Rostedt, Jan 2010.
- lwn.net Debugging the kernel using Ftrace - part 1, by Steven Rostedt, Dec 2009.
- lwn.net Debugging the kernel using Ftrace - part 2, by Steven Rostedt, Dec 2009.

1.14 - Email

Have a question? If you work at Netflix, contact me. If not, please use the perf user mailing list, which I and other perf users are on.

2.0 - perf examples.txt in the supplied documentation

With the latest releases of **perf** there is now also an examples file worth pointing out:

```
# rpm -ql perf | grep example
/usr/share/doc/perf-2.6.32/examples.txt


            ------------------------------
            ****** perf by examples ******
            ------------------------------

[ From an e-mail by Ingo Molnar, http://lkml.org/lkml/2009/8/4/346 ]
```

First, discovery/enumeration of available counters can be done via '**perf list**':

```
titan:~> perf list
  [...]
  kmem:kmalloc                                [Tracepoint event]
  kmem:kmem_cache_alloc                       [Tracepoint event]
  kmem:kmalloc_node                           [Tracepoint event]
  kmem:kmem_cache_alloc_node                  [Tracepoint event]
  kmem:kfree                                  [Tracepoint event]
  kmem:kmem_cache_free                        [Tracepoint event]
  kmem:mm_page_free                           [Tracepoint event]
  kmem:mm_page_free_batched                   [Tracepoint event]
  kmem:mm_page_alloc                          [Tracepoint event]
  kmem:mm_page_alloc_zone_locked              [Tracepoint event]
  kmem:mm_page_pcpu_drain                     [Tracepoint event]
  kmem:mm_page_alloc_extfrag                  [Tracepoint event]
```

Then any (or all) of the above event sources can be activated and measured. For example the page alloc/free properties of a 'hackbench run' are:

```
titan:~> perf stat -e kmem:mm_page_pcpu_drain -e kmem:mm_page_alloc -e
kmem:mm_page_free_batched -e kmem:mm_page_free ./hackbench 10

 Time: 0.575

 Performance counter stats for './hackbench 10':

          13857  kmem:mm_page_pcpu_drain
          27576  kmem:mm_page_alloc
           6025  kmem:mm_page_free_batched
          20934  kmem:mm_page_free
```

You can observe the statistical properties as well, by using the 'repeat the workload N times' feature of perf stat:

```
titan:~> perf stat --repeat 5 -e kmem:mm_page_pcpu_drain -e
kmem:mm_page_alloc -e kmem:mm_page_free_batched -e kmem:mm_page_free
./hackbench 10

 Time: 0.627
 Time: 0.644
 Time: 0.564
 Time: 0.559
 Time: 0.626

 Performance counter stats for './hackbench 10' (5 runs):

          12920  kmem:mm_page_pcpu_drain    ( +-   3.359% )
          25035  kmem:mm_page_alloc         ( +-   3.783% )
           6104  kmem:mm_page_free_batched  ( +-   0.934% )
          18376  kmem:mm_page_free     ( +-   4.941% )

    0.643954516  seconds time elapsed   ( +-   2.363% )
```

Furthermore, these tracepoints can be used to sample the workload as well. For example the page allocations done by a '**git gc**' can be captured the following way:

```
 titan:~/git> perf record -e kmem:mm_page_alloc -c 1 ./git gc
 Counting objects: 1148, done.
 Delta compression using up to 2 threads.
 Compressing objects: 100% (450/450), done.
 Writing objects: 100% (1148/1148), done.
 Total 1148 (delta 690), reused 1148 (delta 690)
 [ perf record: Captured and wrote 0.267 MB perf.data (~11679 samples) ]
```

To check which functions generated page allocations:

```
 titan:~/git> perf report
 # Samples: 10646
 #
 # Overhead          Command               Shared Object
 # ........  ...............  ..........................
 #
    23.57%       git-repack  /lib64/libc-2.5.so
    21.81%              git  /lib64/libc-2.5.so
    14.59%              git  ./git
    11.79%       git-repack  ./git
     7.12%              git  /lib64/ld-2.5.so
     3.16%       git-repack  /lib64/libpthread-2.5.so
     2.09%       git-repack  /bin/bash
     1.97%               rm  /lib64/libc-2.5.so
     1.39%               mv  /lib64/ld-2.5.so
```

```
     1.37%               mv  /lib64/libc-2.5.so
     1.12%       git-repack  /lib64/ld-2.5.so
     0.95%               rm  /lib64/ld-2.5.so
     0.90%  git-update-serv  /lib64/libc-2.5.so
     0.73%  git-update-serv  /lib64/ld-2.5.so
     0.68%             perf  /lib64/libpthread-2.5.so
     0.64%       git-repack  /usr/lib64/libz.so.1.2.3
```

Or to see it on a more finegrained level:

```
titan:~/git> perf report --sort comm,dso,symbol
# Samples: 10646
#
# Overhead          Command               Shared Object  Symbol
# ........  ...............  ..........................  ......
#
     9.35%       git-repack  ./git                       [.] insert_obj_hash
     9.12%              git  ./git                       [.] insert_obj_hash
     7.31%              git  /lib64/libc-2.5.so          [.] memcpy
     6.34%       git-repack  /lib64/libc-2.5.so          [.] _int_malloc
     6.24%       git-repack  /lib64/libc-2.5.so          [.] memcpy
     5.82%       git-repack  /lib64/libc-2.5.so          [.] __GI___fork
     5.47%              git  /lib64/libc-2.5.so          [.] _int_malloc
     2.99%              git  /lib64/libc-2.5.so          [.] memset
```

Furthermore, call-graph sampling can be done too, of page allocations - to see precisely what kind of page allocations there are:

```
titan:~/git> perf record -g -e kmem:mm_page_alloc -c 1 ./git gc
Counting objects: 1148, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (450/450), done.
Writing objects: 100% (1148/1148), done.
Total 1148 (delta 690), reused 1148 (delta 690)
[ perf record: Captured and wrote 0.963 MB perf.data (~42069 samples) ]

titan:~/git> perf report -g
# Samples: 10686
#
# Overhead          Command               Shared Object
# ........  ...............  ..........................
#
    23.25%       git-repack  /lib64/libc-2.5.so
                |
                |--50.00%-- _int_free
                |
                |--37.50%-- __GI___fork
                |          make_child
                |
                |--12.50%-- ptmalloc_unlock_all2
                |          make_child
                |
```

```
                 --6.25%-- __GI_strcpy
    21.61%              git  /lib64/libc-2.5.so
                |
                |--30.00%-- __GI_read
                |          |
                |           --83.33%-- git_config_from_file
                |                     git_config
                |                      |
   [...]
```

Or you can observe the whole system's page allocations for 10 seconds:

```
titan:~/git> perf stat -a -e kmem:mm_page_pcpu_drain -e kmem:mm_page_alloc -e
kmem:mm_page_free_batched -e kmem:mm_page_free sleep 10

 Performance counter stats for 'sleep 10':

        171585  kmem:mm_page_pcpu_drain
        322114  kmem:mm_page_alloc
         73623  kmem:mm_page_free_batched
        254115  kmem:mm_page_free

   10.000591410  seconds time elapsed
```

Or observe how fluctuating the page allocations are, via statistical analysis done over ten 1-second intervals:

```
titan:~/git> perf stat --repeat 10 -a -e kmem:mm_page_pcpu_drain -e
kmem:mm_page_alloc -e kmem:mm_page_free_batched -e kmem:mm_page_free sleep 1

 Performance counter stats for 'sleep 1' (10 runs):

         17254  kmem:mm_page_pcpu_drain    ( +-   3.709% )
         34394  kmem:mm_page_alloc         ( +-   4.617% )
          7509  kmem:mm_page_free_batched  ( +-   4.820% )
         25653  kmem:mm_page_free     ( +-   3.672% )

    1.058135029  seconds time elapsed   ( +-   3.089% )
```

Or you can annotate the recorded '**git gc**' run on a per symbol basis and check which instructions/source-code generated page allocations:

```
titan:~/git> perf annotate __GI___fork
------------------------------------------------
 Percent |      Source code & Disassembly of libc-2.5.so
------------------------------------------------
         :
         :
         :      Disassembly of section .plt:
         :      Disassembly of section .text:
         :
         :      00000031a2e95560 <__fork>:
```

```
[...]
    0.00 :        31a2e95602:   b8 38 00 00 00          mov    $0x38,%eax
    0.00 :        31a2e95607:   0f 05                   syscall
   83.42 :        31a2e95609:   48 3d 00 f0 ff ff       cmp    $0xfffffffffffff000,%rax
    0.00 :        31a2e9560f:   0f 87 4d 01 00 00       ja     31a2e95762 <__fork+0x202>
    0.00 :        31a2e95615:   85 c0                   test   %eax,%eax
```

(this shows that 83.42% of **`__GI___fork`**'s page allocations come from the **`0x38`** system call it performs.)

etc. etc. - a lot more is possible. I could list a dozen of other different usecases straight away - neither of which is possible via **`/proc/vmstat`**.

`/proc/vmstat` is not in the same league really, in terms of expressive power of system analysis and performance analysis.

All that the above results needed were those new tracepoints in **`include/tracing/events/kmem.h`**.

Ingo

3.0 - Steve Johnston's "How to debug with perf, how it works etc."

One of the really neat things with **`perf`** is the ability to do more than meets the obvious eye. Here's a few tricks I've uncovered. Most are already documented, but they were not obvious and this section highlights them for our analysts reading this. I've also included some things I've discovered myself which I feel would be useful knowledge.

3.1 - Commands and One-Liners

3.1.1 - What commands does perf recognize?

`perf` has a few more commands that you might be aware. Here's a list of the most commonly used commands. If in doubt check the man page "**`man perf-<command>`**"

```
# perf help

 usage: perf [--version] [--help] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   inject          Filter to augment the events stream with additional information
   kmem            Tool to trace/measure kernel memory(slab) properties
   kvm             Tool to trace/measure kvm guest os
   list            List all symbolic event types
   lock            Analyze lock events
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   test            Runs sanity tests.
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   trace           strace inspired tool
   probe           Define new dynamic tracepoints
```

```
See 'perf help COMMAND' for more information on a specific command.
```

3.1.2 - Some useful One-Liners (Quick Reference List)

This list started originally from one compiled by Brendan Gregg who himself had gathered or had written. I've subsequently added a lot and also corrected some of Brendan's which were no longer valid for RHEL6 or RHEL7 due to changes in the software pacakge. I've also taken the liberty to reformat the list in keeping with the format I use in this and my other training documents.

1. You will see throughout this reference list, the use of '**`sleep n`**' and sometimes '**`-- sleep n`**'. There's a little confusion as to when to use the '**`--`**'. The actual explanation is, if a command follows the use of the stack option '**`-g`**', you SHOULD use '**`--`**' to precede the command. In reality, it works in most cases with and without, but to avoid confusion/problems, anytime you use a command to determine a period sampling time (which is what **`sleep`** is being used for), I'd recommend you always use it to avoid confusion.

2. When adding probes, while it is possible to do so without using the **`-a`** or **`--add`**, I recommend you ALWAYS use it. Failure to do so will catch you out when you use '**`-f`**' to add a duplicate as an example. It will say it is added then give you an error message and not add it.

3. **`-g dwarf`** not longer seems to work. RHEL6+, it looks like they changed it to **`-g --call-graph dwarf`**. The extension **`dwarf`** is why there was a need to use "**`--`**" so it may no longer be required as the extension has been removed and added to a new option. I'm still using it.

Listing Events

```
Listing all currently known events:
# perf list

Listing sched tracepoints (FYI. You cannot use 'sched*' but you can wildcard any
string after the ':'. EG. perf list 'sched:sched_stat*' will list 4 or 5 items):
# perf list 'sched:*'
```

Counting Events

```
CPU counter statistics for the specified command:
# perf stat <command>

Detailed CPU counter statistics (includes extras) for the specified command:
# perf stat -d <command>

CPU counter statistics for the specified PID or string of PIDs, until Ctrl-C:
# perf stat -p <PID>,<PID>,<PID>

CPU counter statistics for the entire system, for 5 seconds:
# perf stat -a sleep 5
```

```
CPU cycles counter showing seperate kernel and userspace for 10 seconds:
# perf stat --event=cycles:{k,u} -- sleep 10

CPU counter statistics for the entire system, reported by each logical CPU, for 10
seconds:
# perf stat -aA -- sleep 10

CPU counter statistics for the entire system, reported only for logical CPU's 0-3,
for 10 seconds:
# perf stat -aA -C0-3 -- sleep 10

CPU counter statistics for the entire system, reported by each core CPU, for 5
seconds:
# perf stat -a --per-core sleep 5

CPU counter statistics for the entire system, reported by each socketed CPU, for 5
seconds:
# perf stat -a --per-socket -- sleep 5

Various basic CPU statistics, system wide, for 10 seconds:
# perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10

Various CPU level 1 data cache statistics for the specified command:
# perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores <command>

Various CPU data TLB statistics for the specified command:
# perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses <command>

Various CPU last level cache statistics for the specified command:
# perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches <command>

Using raw PMC counters, eg, unhalted core cycles:
# perf stat -e r003c -a sleep 5

Count system calls for the specified PID, until Ctrl-C:
# perf stat -e 'syscalls:sys_enter_*' -p <PID>

Count system calls for the entire system, for 5 seconds:
# perf stat -e 'syscalls:sys_enter_*' -a -- sleep 5

Count scheduler events for the specified PIDs, until Ctrl-C:
# perf stat -e 'sched:*' -p <PID>,<PID>

Count scheduler events for the specified PID, for 10 seconds:
# perf stat -e 'sched:*' -p <PID> sleep 10

Count ext4 events for the entire system, for 10 seconds:
# perf stat -e 'ext4:*' -a -- sleep 10

Count block device I/O events for the entire system, for 10 seconds:
# perf stat -e 'block:*' -a sleep 10

Show system calls by process, refreshing every 2 seconds:
# perf top -e syscalls:sys_enter* -ns comm
```

```
Count bus-cycles for the entire system for 5 seconds, exporting the data in CSV
style (-x) using ':' as a separator:
# perf stat -e bus-cycles -a -x: -- sleep 5

Count a number of events for 5 seconds for the entire system and repeat 10 times
providing a standard deviation of counts for the events:
# perf stat --repeat 10 -a -e kmem:mm_page* -- sleep 5
```

Profiling

```
Sample on-CPU functions for the specified command, at 99 Hertz:
# perf record -F 99 <command>

Sample on-CPU functions for the specified PID, at 99 Hertz, until Ctrl-C:
# perf record -F 99 -p <PID>

Sample on-CPU functions for the specified PID, at 99 Hertz, for 10 seconds:
# perf record -F 99 -p <PID> sleep 10

Sample CPU stack traces for the specified PID, at 99 Hertz, for 10 seconds:
# perf record -F 99 -p <PID> -g -- sleep 10

Sample CPU stack traces for the PID, using dwarf to unwind stacks, at 99 Hertz, for
10 seconds:
# perf record -F 99 -p <PID> -g --call-graph dwarf -- sleep 10

Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds:
# perf record -F 99 -ag -- sleep 10

If the previous command didn't work, try forcing perf to use the cpu-clock event:
# perf record -F 99 -e cpu-clock -ag -- sleep 10

Sample CPU stack traces for the entire system, with dwarf stacks, at 99 Hertz, for
10 seconds:
# perf record -F 99 -ag --call-graph dwarf -- sleep 10

Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 seconds:
# perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5

Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds:
# perf record -e LLC-load-misses -c 100 -ag -- sleep 5

Sample on-CPU kernel instructions, for 5 seconds:
# perf record -e cycles:k -a -- sleep 5

Sample on-CPU user instructions, for 5 seconds:
# perf record -e cycles:u -a -- sleep 5

Sample on-CPU instructions precisely (using PEBS), for 5 seconds:
# perf record -e cycles:p -a -- sleep 5

Perform branch tracing (needs HW support), for 1 second:
# perf record -b -a sleep 1

Sample on-CPU kernel instructions just for logical CPU #4, for 10 seconds:
# perf record -e cycles -C4 -- sleep 10
```

```
Sample on-CPU instructions and cache-misses combined into a single report,for the
entire system, for 10 seconds:
# perf record -e '{cycles,cache-misses}' -a -- sleep 10
```

Static Tracing

```
Trace new processes, until Ctrl-C:
# perf record -e sched:sched_process_exec -a

Trace all context-switches, until Ctrl-C:
# perf record -e context-switches -a

Trace context-switches via sched tracepoint, until Ctrl-C:
# perf record -e sched:sched_switch -a

Trace all context-switches with stack traces, until Ctrl-C:
# perf record -e context-switches -ag

Trace all context-switches with stack traces, for 10 seconds:
# perf record -e context-switches -ag -- sleep 10

Trace CPU migrations, for 10 seconds:
# perf record -e migrations -a -- sleep 10

Trace all connect()s with stack traces (outbound connections), until Ctrl-C:
# perf record -e syscalls:sys_enter_connect -ag

Trace all accepts()s with stack traces (inbound connections), until Ctrl-C:
# perf record -e syscalls:sys_enter_accept* -ag

Trace all block device (disk I/O) requests with stack traces, until Ctrl-C:
# perf record -e block:block_rq_insert -ag

Trace all block device issues and completions (has timestamps), until Ctrl-C:
# perf record -e block:block_rq_issue -e block:block_rq_complete -a

Trace all block completions, of size at least 100 Kbytes, until Ctrl-C:
# perf record -e block:block_rq_complete --filter 'nr_sector > 200' -a

Trace all block completions, synchronous writes only, until Ctrl-C:
# perf record -e block:block_rq_complete --filter 'rwbs == "WS"' -ag

Trace all block completions, all types of writes, until Ctrl-C:
# perf record -e block:block_rq_complete --filter 'rwbs ~ "*W*"' -a

Trace all minor faults (RSS growth) with stack traces, until Ctrl-C:
# perf record -e minor-faults -ag

Trace all page faults with stack traces, until Ctrl-C:
# perf record -e page-faults -ag

Trace all ext4 calls, and write to a non-ext4 location, until Ctrl-C:
# perf record -e 'ext4:*' -o /tmp/perf.data -a
```

```
Trace kswapd wakeup events, until Ctrl-C:
# perf record -e vmscan:mm_vmscan_wakeup_kswapd -ag

Set a memory address hardware breakpoint (RHEL7 and above only):
# perf record -e mem:0xffffffff81943040:rw -a


Trace kernel slab memory for 10 seconds:
# perf kmem record -- sleep 10

Trace memory usage/profiling for 10 seconds:
# perf mem record -- sleep 10
```

Dynamic Tracing

```
Add a tracepoint for the kernel tcp_sendmsg() function entry ("--add" is optional):
# perf probe --add tcp_sendmsg

Remove the tcp_sendmsg() tracepoint (or use "--del"):
# perf probe -d tcp_sendmsg

Add a tracepoint for the kernel tcp_sendmsg() function return:
# perf probe 'tcp_sendmsg%return'

Show available variables for the kernel tcp_sendmsg() function (needs debuginfo):
# perf probe -V tcp_sendmsg

Show available variables for the kernel tcp_sendmsg() function, plus external vars
(needs debuginfo):
# perf probe -V tcp_sendmsg --externs

Show available line probes for tcp_sendmsg() (needs debuginfo):
# perf probe -L tcp_sendmsg

Show available variables for tcp_sendmsg() at line number 81 (needs debuginfo):
# perf probe -V tcp_sendmsg:81

Add a tracepoint for tcp_sendmsg()+2075 (Instruction offset from start of
function). Can also use '+0x8ab'. Needs debuginfo:
# perf probe --add 'tcp_sendmsg+2075'

Add a tracepoint for tcp_sendmsg(), with three entry argument registers (platform
specific):
Note. Possible register names you can use for Intel and AMD are:
       %di %si %dx %cx %ax %bx %bp %sp %ip %flags %cs %ss
       %r8 %r9 %r10 %r11 %r12 %r13 %r14 %r15
# perf probe 'tcp_sendmsg %ax %dx %cx'

Add multiple tracepoints for tcp_sendmsg() that each call a function containing
'*tcp_push*' (wildcard allowed):
# perf probe -a 'tcp_sendmsg;*tcp_push*'

Add a tracepoint for function getname() + line 27, rename it to 'Start' and display
the filename, renamed pathname, as a string:
# perf probe --add 'Start=getname_flags:27 pathname=filename:string'
```

```
Add a tracepoint for tcp_sendmsg(), with an alias ("bytes") for the %cx register
(platform specific):
# perf probe --add 'tcp_sendmsg bytes=%cx'

Trace previously created probe when the bytes (alias) variable is greater than 100:
# perf record -e probe:tcp_sendmsg --filter 'bytes > 100' -a

Add a tracepoint for tcp_sendmsg() return, and capture the return value:
# perf probe 'tcp_sendmsg%return $retval'

Add a tracepoint for tcp_sendmsg(), and "size" entry argument (reliable, but needs
debuginfo):
# perf probe -a 'tcp_sendmsg size'


Add a tracepoint for tcp_sendmsg(), with size and socket state (needs debuginfo):
# perf probe --add 'tcp_sendmsg size sock->state'

Check if you can do this, but don't actually do it (needs debuginfo). -n does a dry
run but does not add/delete, -v displays verbose
# perf probe -nv 'tcp_sendmsg size sock->state'

Trace previous probe when size is non-zero, and state is not TCP_ESTABLISHED(1)
(needs debuginfo):
# perf record -e probe:tcp_sendmsg --filter 'size > 0 && state != 1' -a

Add a tracepoint for tcp_sendmsg() line 81 with local variable seglen (needs
debuginfo):
# perf probe -a 'tcp_sendmsg:81 seglen'

Add a tracepoint for do_sys_open() with the filename as a string (needs debuginfo):
# perf probe -a 'do_sys_open filename:string'

Add a tracepoint for myfunc() return, and include the retval as a string:
# perf probe --add 'myfunc%return +0($retval):string'

Add a tracepoint for the user-level malloc() function from libc (Requires the
kernel is built with CONFIG_UPROBE_EVENTS):
# perf probe -x /lib64/libc.so.6 malloc

List currently available dynamic probes:
# perf probe -l

Add a tracepoint for drm_av_sync_delay() in kernel module drm:
# perf probe -m drm -a drm_av_sync_delay

Add a tracepoint for dm_region_hash_destroy() in specific kernel module file:
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-
327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug --add
dm_region_hash_destroy

Show available line probes for drm_av_sync_delay() in kernel module drm:
# perf probe -m drm -L drm_av_sync_delay
```

```
Show available line probes for dm_region_hash_destroy() in kernel module drm:
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-
327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug -L
dm_region_hash_destroy

List all available function calls in kernel module drm:
# perf probe -m drm -F

List available variables for compat_drm_agp_info() in kernel module drm:
# perf probe -m drm -V compat_drm_agp_info

List available variables for dm_region_hash_destroy() in speific kernel module
file:
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-
327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug -V
dm_region_hash_destroy

Trace a process by PID and add timestamps (-T) and a summary (-S):
# perf trace -p <PID> -TS
```

Reporting

```
Display what events are captured in the perf.data file
# perf evlist

Show perf.data in an ncurses browser (TUI) if possible:
# perf report

Show perf.data with a column for sample count:
# perf report -n

Show perf.data as a text report, with data coalesced and percentages:
# perf report --stdio

Show perf.data as a text report, in CSV style (-t) with ',' separator:
# perf report -t, --stdio

List all raw events from perf.data:
# perf script

List all python scripts:
# perf script --list

Report the netdevice times using the supplied python script. Ctrl/c to terminate:
# perf script netdev-times

Record the netdevice times in perf.data using the supplied python script:
# perf script record netdev-times

List all raw events from perf.data, with customized fields:
# perf script -F time,event,trace

Dump raw contents from perf.data as hex (for debugging):
# perf script -D
```

Disassemble and annotate instructions with percentages (needs some debuginfo):
perf annotate --stdio

Disassemble instructions with percentages and show the instructions bytes (needs some debuginfo):
perf annotate --stdio --asm-raw

Show benchmark futex operations:
perf bench futex all

Test that perf is installed correctly:
perf test

Report kernel slab memory usage for 10 seconds:
perf kmem record -- sleep 10
perf kmem stat

Report kernel memory allocation/usage for 10 seconds:
perf mem record -- sleep 10
perf script
perf mem report

Trace all lock usage for 30 seconds, review and then report (requires the DEBUG kernel):
perf lock record -- sleep 30
perf lock script
perf lock report

3.1.3 - Summary of various reports available

We have covered these before but this is worth a quick review:

perf report

Standard reporting using the TUI (Text user Interface)

```
# perf report

Samples: 1K of event 'probe:__do_page_fault_2', Event count (approx.): 1845
  Children      Self  Command          Shared Object          Symbol
+   44.93%     0.00%  DOM Worker       [kernel.vmlinux]       [k] page_fault
+   44.93%     0.00%  DOM Worker       [kernel.vmlinux]       [k] do_page_fault
+   44.93%    44.93%  DOM Worker       [kernel.vmlinux]       [k] __do_page_fault
+   28.08%     0.00%  chrome           [kernel.vmlinux]       [k] page_fault
+   28.08%     0.00%  chrome           [kernel.vmlinux]       [k] do_page_fault
+   28.08%    28.08%  chrome           [kernel.vmlinux]       [k] __do_page_fault
+   15.88%     0.00%  firefox          [kernel.vmlinux]       [k] page_fault
+   15.88%     0.00%  firefox          [kernel.vmlinux]       [k] do_page_fault
+   15.88%    15.88%  firefox          [kernel.vmlinux]       [k] __do_page_fault
+   13.88%     0.00%  DOM Worker       libxul.so              [.] 0xffff80264021ec15
+   12.14%     0.00%  chrome           chrome                 [.] 0xffff80f443390dde
+   12.14%     0.00%  chrome           perf-26348.map         [.] 0x00003151231ad301
+   12.14%     0.00%  chrome           perf-26348.map         [.] 0x00003151231d3e43
+    9.43%     0.00%  chrome           perf-26348.map         [.] 0x0000315123185cd5
```

The symbol is a useful column to note:

- [.] Userspace
- [k] Kernel
- [g] Guest kernel
- [u] Guest Userspace
- [H] Hypervisor

Standard reporting using the TUI and adding the number of samples

```
# perf report -n

Samples: 1K of event 'probe:__do_page_fault_2', Event count (approx.): 1845
  Children      Self       Samples  Command         Shared Object           Symbol
+   44.93%     0.00%             0  DOM Worker      [kernel.vmlinux]        [k] page_fault
+   44.93%     0.00%             0  DOM Worker      [kernel.vmlinux]        [k] do_page_fault
+   44.93%    44.93%           829  DOM Worker      [kernel.vmlinux]        [k] __do_page_fault
+   28.08%     0.00%             0  chrome          [kernel.vmlinux]        [k] page_fault
+   28.08%     0.00%             0  chrome          [kernel.vmlinux]        [k] do_page_fault
+   28.08%    28.08%           518  chrome          [kernel.vmlinux]        [k] __do_page_fault
+   15.88%     0.00%             0  firefox         [kernel.vmlinux]        [k] page_fault
+   15.88%     0.00%             0  firefox         [kernel.vmlinux]        [k] do_page_fault
+   15.88%    15.88%           293  firefox         [kernel.vmlinux]        [k] __do_page_fault
```

Standard reporting and outputting the data to standard-IO terminal output

```
# perf report --show_nr_samples --stdio
```

perf script

Displaying all the raw events. Very useful if recording also included all CPUs (-a) and backtraces (-g)

```
# perf script
```

Displaying all the raw events with a customized output display. **NOTE. Time stamps are available.** Note. Man page for latest rhel6 shows **-f** and is now **-F** (rhel7 is correct)

```
# perf script -F time,event,trace
```

Dump all the perf data in raw hex

```
# perf script -D
```

Display internal perf script functions that can provide additional performance data

```
# perf script --list
```

Run a perf top script

```
# perf script netdev-times
```

Record a perf top script

```
# perf script record netdev-times
```

perf annotate

Show the disassembly listing and source line numbers of **just the probes currently collected in the perf.data.** This requires the debuginfo.

```
# perf annotate --stdio
```

Show the disassembly listing and the disassembled instruction bytes for each instruction

```
# perf annotate --asm-raw --stdio
```

perf list

List all the default events that can be monitored and includes all currently added probes

```
# perf list

# perf list 'block:*'                ← Depending on release, may work without the ''
```

```
# perf list 'probe:*'
```

← Depending on release, probes may be listed before Tracepoint events, or may be listed as part of Tracepoint events in alphabetic order

perf probe

Show just a specific source code function (and optional line number **:n-m**). You can list ANY of the known functions. It is not dependent on a probe being added.

```
# perf probe -L __do_page_fault:173-175
```

List just current probes that have been added

```
# perf probe --list
```

or can be done with **perf list**

```
# perf list probe:*
```

Display a listing of all known functions that can be probed

```
# perf probe -F
```

Display all the variables that are known at a specific line of code for ANY known function

```
# perf probe -V tcp_sendmsg:179
```

Display a listing of all known functions that can be probed in a specific module

```
# perf probe -m drm -F
```

Display all the variables that are known at a specific line of code for a function in a module

```
# perf probe -m drm -V compat_drm_agp_info
```

3.2 - Listing features and capabilites

3.2.1 - Obtaining source code listings

I've “painted” some lines blue, those are lines that you cannot actually probe. It's a compiler 'thing'. We don't actually compile every line of C code. Perf shows the output exactly the same way. It presents the line numbers that CAN be probed. If you try to probe a line not listed, it will simply give you an error. No harm done.

```
# perf probe -L tcp_sendmsg
<tcp_sendmsg@/usr/src/debug/kernel-2.6.32-573.12.1.el6/linux-2.6.32-
573.12.1.el6.x86_64/net/ipv4/tcp.c:0>
      0  int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                        size_t size)
      2  {
      3          struct sock *sk = sock->sk;
                 struct iovec *iov;
                 struct tcp_sock *tp = tcp_sk(sk);
                 struct sk_buff *skb;
                 int iovlen, flags;
                 int mss_now, size_goal;
                 int err, copied;
                 long timeo;

     12          lock_sock(sk);
                 TCP_CHECK_TIMER(sk);

     15          flags = msg->msg_flags;
     16          timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);

                 /* Wait for a connection to finish. */
     19          if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
     20                  if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
                                 goto out_err;

                 /* This should be in poll */
     24          clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

     26          mss_now = tcp_send_mss(sk, &size_goal, flags);

                 /* Ok commence sending. */
     29          iovlen = msg->msg_iovlen;
     30          iov = msg->msg_iov;
                 copied = 0;

                 err = -EPIPE;
     34          if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
                         goto out_err;
```

```
	while (--iovlen >= 0) {
		size_t seglen = iov->iov_len;
		unsigned char __user *from = iov->iov_base;

		iov++;
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

3.2.2 - Obtaining disassembled code of source

Perf is blowing me away with its capabilities. If you exclude the **--stdio** you can see this display in a TUI (Text /graphical user interface). I suspect most folks are like me and would like to keep a “record” of the listing. I've color coded the output similar to what you'll see perf provide. This was produced from a **probe:tcp_sendmsg**:

```
# perf annotate --stdio
 Percent |      Source code & Disassembly of vmlinux for probe:tcp_sendmsg
------------------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      ffffffff814b3cf0 <tcp_sendmsg>:
         :              return tmp;
         :      }
         :
         :      int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
         :                      size_t size)
         :      {
    0.00 :        ffffffff814b3cf0:       push   %rbp
  100.00 :        ffffffff814b3cf1:       mov    %rsp,%rbp
    0.00 :        ffffffff814b3cf4:       push   %r15
    0.00 :        ffffffff814b3cf6:       push   %r14
    0.00 :        ffffffff814b3cf8:       push   %r13
    0.00 :        ffffffff814b3cfa:       push   %r12
    0.00 :        ffffffff814b3cfc:       push   %rbx
    0.00 :        ffffffff814b3cfd:       sub    $0x78,%rsp
    0.00 :        ffffffff814b3d01:       callq  ffffffff8100adc0 <mcount>
         :              struct sock *sk = sock->sk;
    0.00 :        ffffffff814b3d06:       mov    0x38(%rsi),%r12
         :
         :      extern void lock_sock_nested(struct sock *sk, int subclass);
         :
         :      static inline void lock_sock(struct sock *sk)
         :      {
         :              lock_sock_nested(sk, 0);
    0.00 :        ffffffff814b3d0a:       xor    %esi,%esi
         :              return tmp;
         :      }
         :
         :      int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
         :                      size_t size)
         :      {
    0.00 :        ffffffff814b3d0c:       mov    %rdx,%rbx
    0.00 :        ffffffff814b3d0f:       mov    %r12,%rdi
    0.00 :        ffffffff814b3d12:       callq  ffffffff81459b20 <lock_sock_nested>
         :              long timeo;
         :
         :              lock_sock(sk);
         :              TCP_CHECK_TIMER(sk);
         :
```

```
        :              flags = msg->msg_flags;
    0.00 :        ffffffff814b3d17:       mov    0x30(%rbx),%eax
    0.00 :        ffffffff814b3d1a:       mov    %eax,-0x44(%rbp)
         :              return noblock ? 0 : sk->sk_rcvtimeo;
         :      }
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

You can also get the instruction disassembly by adding the **--asm-raw** toping

```
# perf annotate --asm-raw __do_page_fault --stdio
 Percent |      Source code & Disassembly of vmlinux for probe:__do_page_fault
------------------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      ffffffff8104f010 <__do_page_fault>:
         :      {
         :              return address >= TASK_SIZE_MAX;
         :      }
         :
         :      static inline void __do_page_fault(struct pt_regs *regs, unsigned long address, unsigned
long error_code)
         :      {
    0.00 :        ffffffff8104f010:       55                      push   %rbp
  100.00 :        ffffffff8104f011:       48 89 e5                mov    %rsp,%rbp
    0.00 :        ffffffff8104f014:       48 81 ec 10 01 00 00    sub    $0x110,%rsp
    0.00 :        ffffffff8104f01b:       48 89 5d d8             mov    %rbx,-0x28(%rbp)
    0.00 :        ffffffff8104f01f:       4c 89 65 e0             mov    %r12,-0x20(%rbp)
    0.00 :        ffffffff8104f023:       4c 89 6d e8             mov    %r13,-0x18(%rbp)
    0.00 :        ffffffff8104f027:       4c 89 75 f0             mov    %r14,-0x10(%rbp)
    0.00 :        ffffffff8104f02b:       4c 89 7d f8             mov    %r15,-0x8(%rbp)
    0.00 :        ffffffff8104f02f:       e8 8c bd fb ff          callq  ffffffff8100adc0 <mcount>
         :
         :      DECLARE_PER_CPU(struct task_struct *, current_task);
         :
         :      static __always_inline struct task_struct *get_current(void)
         :      {
         :              return percpu_read_stable(current_task);
    0.00 :        ffffffff8104f034:       65 4c 8b 3c 25 00 bc    mov    %gs:0xbc00,%r15
    0.00 :        ffffffff8104f03b:       00 00
    0.00 :        ffffffff8104f03d:       48 89 bd 00 ff ff ff    mov    %rdi,-0x100(%rbp)
    0.00 :        ffffffff8104f044:       48 89 95 08 ff ff ff    mov    %rdx,-0xf8(%rbp)
    0.00 :        ffffffff8104f04b:       49 89 f4                mov    %rsi,%r12
         :              int fault;
         :              int write = error_code & PF_WRITE;
         - - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

3.2.3 - What local variables are being used at any given point?

You can find the local "variables" in use at any probe point in the kernel

```
# perf probe -V tcp_sendmsg
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                size_t  size
                struct kiocb*   iocb
                struct msghdr*  msg
                struct socket*  sock
```

You can also find the local variables at a specific line number.

```
# perf probe -V tcp_sendmsg:34
Available variables at tcp_sendmsg:34
        @<tcp_sendmsg+175>
                int                     mss_now
                struct msghdr*          msg
                struct sock*            sk
                struct tcp_sock*        tp
        @<tcp_sendmsg+191>
                int                     iovlen
                int                     mss_now
                struct iovec*           iov
                struct sock*            sk
                struct tcp_sock*        tp
        @<tcp_sendmsg+208>
                int                     iovlen
                struct iovec*           iov
                struct sock*            sk
                struct tcp_sock*        tp
```

```
# perf probe -V tcp_sendmsg%return
Available variables at tcp_sendmsg%return
        @<tcp_sendmsg+0>
                size_t          size
                struct kiocb*    iocb
                struct msghdr*   msg
                struct socket*   sock
```

You can display something called “external variables”. These are ALSO available to be included in a probe –add.

```
# perf probe --externs -V tcp_sendmsg
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                (function_type)*        ip_nat_decode_session
                __u8*   ip_tos2prio
                atomic_long_t   nr_swap_pages
                atomic_long_t*  vm_stat
                atomic_t        tcp_memory_allocated
                char*   __setup_str_set_thash_entries
```

```
char*   hex_asc
char*   inet_csk_timer_bug_msg
cpumask_var_t   cpu_callout_mask
cpumask_var_t   per_cpu__cpu_core_map
cpumask_var_t   per_cpu__cpu_sibling_map
cpumask_var_t*  node_to_cpumask_map
dma_addr_t      bad_dma_address
int     acpi_disabled
int     acpi_ht
int     acpi_noirq
int     acpi_pci_disabled
int     audit_enabled
int     debug_locks
int     disable_apic
int     nr_cpu_ids
int     nr_online_nodes
int     page_group_by_mobility_disabled
int     per_cpu__cpu_number
int     per_cpu__node_number
int     per_cpu__x86_cpu_to_node_map
int     percpu_counter_batch
int     prof_on
int     sched_mc_power_savings
int     sched_smt_power_savings
int     smp_found_config
int     smp_num_siblings
int     sysctl_max_syn_backlog
int     sysctl_tcp_adv_win_scale
int     sysctl_tcp_dma_copybreak
int     sysctl_tcp_ecn
int     sysctl_tcp_fin_timeout
int     sysctl_tcp_keepalive_intvl
int     sysctl_tcp_keepalive_probes
int     sysctl_tcp_keepalive_time
int     sysctl_tcp_low_latency
int     sysctl_tcp_max_orphans
int     sysctl_tcp_min_tso_segs
int     sysctl_tcp_syn_retries
int     tcp_memory_pressure
int     time_status
int     timer_stats_active
int     x2apic_phys
int*    console_printk
int*    sysctl_tcp_mem
int*    sysctl_tcp_rmem
int*    sysctl_tcp_wmem
int*    x86_cpu_to_node_map_early_ptr
irq_cpustat_t   per_cpu__irq_stat
long int        total_swap_pages
long long int   dynamic_debug_enabled
long long int   dynamic_debug_enabled2
long unsigned int       jiffies
long unsigned int       mmap_min_addr
long unsigned int       mmu_cr4_features
long unsigned int       per_cpu__kernel_stack
long unsigned int       per_cpu__this_cpu_off
long unsigned int       tcp_md5sig_users
long unsigned int       thash_entries
long unsigned int       totalram_pages
long unsigned int*      __per_cpu_offset
```

```
        long unsigned int*      cpu_bit_bitmap
        long unsigned int*      sysctl_local_reserved_ports
        nodemask_t*     node_states
        pg_data_t**     node_data
        physid_mask_t   phys_cpu_present_map
        pteval_t        __supported_pte_mask
        size_t  size
        spinlock_t      dcache_lock
        spinlock_t      dma_spin_lock
        spinlock_t      inet_peer_idlock
        spinlock_t      tcp_md5sig_pool_lock
        struct address_space    swapper_space
        struct apic*    apic
        struct cache_sizes*     malloc_sizes
        struct cgroup_subsys    mem_cgroup_subsys
        struct cpuinfo_x86      boot_cpu_data
        struct cpuinfo_x86_rh   boot_cpu_data_rh
        struct cpuinfo_x86_rh   per_cpu__cpu_info_rh
        struct cpumask* cpu_online_mask
        struct cpumask* cpu_possible_mask
        struct cpumask* cpu_present_mask
        struct device   x86_dma_fallback_dev
        struct dma_map_ops*     dma_ops
        struct inet_hashinfo    tcp_hashinfo
        struct inet_timewait_death_row  tcp_death_row
        struct kiocb*   iocb
        struct list_head*       nf_hooks
        struct mem_section**    mem_section
        struct memnode  memnode
        struct mm_struct*       swap_token_mm
        struct mm_tracker       mm_tracking_struct
        struct msghdr*  msg
        struct neigh_table      nd_tbl
        struct nf_afinfo**      nf_afinfo
        struct obs_kernel_param __setup_set_thash_entries
        struct percpu_counter   tcp_orphan_count
        struct percpu_counter   tcp_sockets_allocated
        struct percpu_rw_semaphore      cgroup_threadgroup_rwsem
        struct pglist_data**    node_data
        struct pid*     cad_pid
        struct pid_namespace    init_pid_ns
        struct pt_regs* per_cpu__irq_regs
        struct pv_apic_ops      pv_apic_ops
        struct pv_cpu_ops       pv_cpu_ops
        struct pv_info  pv_info
        struct pv_irq_ops       pv_irq_ops
        struct pv_mmu_ops       pv_mmu_ops
        struct pv_time_ops      pv_time_ops
        struct resource ioport_resource
        struct rps_sock_flow_table*     rps_sock_flow_table
        struct smp_ops  smp_ops
        struct socket*  sock
        struct task_struct*     per_cpu__current_task
        struct tcp_congestion_ops       tcp_reno
        struct tcp_md5sig_pool**        tcp_md5sig_pool
        struct tracepoint       __tracepoint_irq_handler_entry
        struct tracepoint       __tracepoint_irq_handler_exit
        struct tracepoint       __tracepoint_module_free
        struct tracepoint       __tracepoint_module_get
        struct tracepoint       __tracepoint_module_load
```

```
                struct tracepoint        __tracepoint_module_put
                struct tracepoint        __tracepoint_module_request
                struct tracepoint        __tracepoint_softirq_entry
                struct tracepoint        __tracepoint_softirq_exit
                struct tracepoint        __tracepoint_softirq_raise
                struct vm_event_state    per_cpu__vm_event_states
                struct x86_init_ops      x86_init
                struct x86_platform_ops x86_platform
                u16     per_cpu__x86_bios_cpu_apicid
                u32     inet_ehash_secret
                union irq_stack_union    per_cpu__irq_stack_union
                unsigned char*  new_state
                unsigned int    apic_verbosity
                unsigned int    sysctl_net_busy_poll
                unsigned int    sysctl_timer_migration
```

Example of adding internal and external variables:

```
# perf probe -V 'tcp_sendmsg'
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                int             size_goal
                long int        timeo
                size_t          size
                struct kiocb*   iocb
                struct msghdr*  msg
                struct sock*    sk

# perf probe --add 'tcp_sendmsg size timeo nr_swap_pages tcp_memory_pressure'
                                \Internals/ \           Externals            /
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with size timeo nr_swap_pages
tcp_memory_pressure)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

```
# perf record -e probe:tcp_sendmsg -a -- sleep 60
[ perf record: Woken up 15 times to write data ]
[ perf record: Captured and wrote 4.898 MB perf.data (1351 samples) ]
```

```
# perf script | head
   Socket Thread 15920 [001] 1030382.817270: probe:tcp_sendmsg: (ffffffffba843a10)
size=0xcb timeo=-107724897059504 nr_swap_pages=0xf76bd tcp_memory_pressure=0

   Socket Thread 15920 [001] 1030382.878909: probe:tcp_sendmsg: (ffffffffba843a10)
size=0x7e timeo=-107724897059504 nr_swap_pages=0xf76bd tcp_memory_pressure=0

   Socket Thread 15920 [001] 1030383.521158: probe:tcp_sendmsg: (ffffffffba843a10)
size=0x1af timeo=-107724897059504 nr_swap_pages=0xf76bd tcp_memory_pressure=0

   Socket Thread 15920 [001] 1030383.575681: probe:tcp_sendmsg: (ffffffffba843a10)
size=0x149 timeo=-107724897059504 nr_swap_pages=0xf76bd tcp_memory_pressure=0
```

3.2.4 - How to tell what events are/were sampled in the perf.data

This is actually quite simple. You simply run **perf evlist**.

```
# perf evlist
probe:__do_page_fault
```

You can supply the name of the corresponding **perf.data** file (if renamed) using **-i** if required.

Another useful feature, add **-F** and it will tell you what recording frequency (sampling per second) that was used:

```
# perf evlist -F
cycles: sample_freq=4000
```

Need to find even more info? Use **-v** to find the verbose list of event sampled data:

```
# perf evlist -v
cycles: size: 104, { sample_period, sample_freq }: 4000, sample_type: IP|TID|TIME|
PERIOD, disabled: 1, inherit: 1, mmap: 1, comm: 1, freq: 1, enable_on_exec: 1, task:
1, sample_id_all: 1, exclude_guest: 1, mmap2: 1, comm_exec: 1
```

3.2.5 - Displaying variables and structure elements

Can you do more? Yes you can. Let's not only probe a call, let's display arguments at the call. How do I find out the arguments? Use **-V**

```
# perf probe -V __do_page_fault
Available variables at __do_page_fault
        @<__do_page_fault+0>
                long unsigned int       address
                long unsigned int       error_code
                struct pt_regs* regs

# perf probe --add '__do_page_fault address error_code'
Added new event:
  probe:__do_page_fault (on __do_page_fault with address error_code)

You can now use it in all perf tools, such as:

      perf record -e probe:__do_page_fault -aR sleep 1
```

Start recording with **-a** (All CPUs) and **-g** (Capture Stack Frames), **sleep** for 5 seconds. Note, **sleep** is NOT a perf directive... we're simply executing a command... any command.

```
# perf record -e probe:__do_page_fault -ag sleep 5
[ perf record: Woken up 4 times to write data ]
[ perf record: Captured and wrote 1.757 MB perf.data (~76764 samples) ]

# perf script
perf  9241 [004] 885408.979095: probe:__do_page_fault: (ffffffff8104f010)
address=7fd77e116400 error_code=4
        ffffffff8104f011 __do_page_fault ([kernel.kallsyms])
        ffffffff8153c835 page_fault ([kernel.kallsyms])
                  427be7 cmd_record (/usr/bin/perf)
                  4196c3 [unknown] (/usr/bin/perf)
                  41a2fd main (/usr/bin/perf)
              39c821ed5d __libc_start_main (/lib64/libc-2.12.so)

perf  9242 [000] 885408.979099: probe:__do_page_fault: (ffffffff8104f010)
address=39c82ad280 error_code=14
        ffffffff8104f011 __do_page_fault ([kernel.kallsyms])
        ffffffff8153c835 page_fault ([kernel.kallsyms])
              39c82ad280 execvp (/lib64/libc-2.12.so)
                  427940 cmd_record (/usr/bin/perf)
                  4196c3 [unknown] (/usr/bin/perf)
                  41a2fd main (/usr/bin/perf)
              39c821ed5d __libc_start_main (/lib64/libc-2.12.so)
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

Can you see variables anywhere in code? Yes. Take our example line 173 above:

```
# perf probe -L __do_page_fault:173-175
<__do_page_fault@/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-
573.18.1.el6.x86_64/arch/x86/mm/fault.c:173>
    173         fault = handle_mm_fault(mm, vma, address, flags);

                /*

# perf probe -V __do_page_fault:173
Available variables at __do_page_fault:173
        @<__do_page_fault+310>
                long unsigned int       address
                struct mm_struct*       mm
                struct task_struct*     tsk
                struct vm_area_struct*  vma
                unsigned int    flags

# perf probe --del probe:*
/sys/kernel/debug/tracing/uprobe_events file does not exist - please rebuild kernel
with CONFIG_UPROBE_EVENTS.
Removed event: probe:__do_page_fault
```

And let's look at a sub-element in the **task_struct** (**tsk**):

```
# perf probe --add '__do_page_fault:173 tsk->flags'
Added new event:
  probe:__do_page_fault (on __do_page_fault:173 with flags=tsk->flags)

You can now use it in all perf tools, such as:

      perf record -e probe:__do_page_fault -aR sleep 1


# perf record -e probe:__do_page_fault -ag sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.860 MB perf.data (~37580 samples) ]

# perf script
perf  9298 [000] 886049.740992: probe:__do_page_fault: (ffffffff8104f146) flags=402100
        ffffffff8104f147 __do_page_fault ([kernel.kallsyms])
        ffffffff8153f48e do_page_fault ([kernel.kallsyms])
        ffffffff8153c835 page_fault ([kernel.kallsyms])
                  427be7 cmd_record (/usr/bin/perf)
                  4196c3 [unknown] (/usr/bin/perf)
                  41a2fd main (/usr/bin/perf)
              39c821ed5d __libc_start_main (/lib64/libc-2.12.so)
```

```
perf  9299 [001] 886049.740995: probe:__do_page_fault: (ffffffff8104f146) flags=402040
        ffffffff8104f147 __do_page_fault ([kernel.kallsyms])
        ffffffff8153f48e do_page_fault ([kernel.kallsyms])
        ffffffff8153c835 page_fault ([kernel.kallsyms])
              39c82ad280 execvp (/lib64/libc-2.12.so)
                  427940 cmd_record (/usr/bin/perf)
                  4196c3 [unknown] (/usr/bin/perf)
                  41a2fd main (/usr/bin/perf)
              39c821ed5d __libc_start_main (/lib64/libc-2.12.so)
```

3.2.6 - How do I find the variables for pre-defined tracepoint events?

This one stumped me for a while until a colleague who had read a LWN article just happened to recall something about this. Let's take 2 examples:

```
# perf record -e block:block_rq_complete -a --filter 'nr_sector > 200'
# perf record -e block:block_rq_complete -a --filter 'rwbs == "WS"'
```

Both of these work.... but how do you know that **nr-sector** and **rwbs** are available variables? Are there others? How about the other pre-defined events? You cannot use **perf probe -V <known event>**. That returns you errors:

```
# perf probe -V block_rq_complete
Failed to find the address of block_rq_complete
  Error: Failed to show vars.
```

The answer is a little cumbersome as I have not as yet found any method using any **perf** command to determine the variables. However, it is simple to find "once you know how". First you need to mount the debugfs:

```
# mount -t debugfs none /sys/kernel/debug/
```

And now the answer. First be aware of the tracing directory now visible in debugfs. Within that is an **events** directory and sub to that are the names of the directories that directly correlate to the pre-defined tracepoints events as seen in **perf list**:

```
# ll /sys/kernel/debug/tracing/events
total 0
drwxr-xr-x.  20 root root 0 Apr 12 18:52 block
drwxr-xr-x. 143 root root 0 Apr 12 18:52 cfg80211
drwxr-xr-x.   5 root root 0 Apr 12 18:52 drm
-rw-r--r--.   1 root root 0 Apr 12 18:52 enable
drwxr-xr-x.  35 root root 0 Apr 12 18:52 ext4
drwxr-xr-x.  10 root root 0 Apr 12 18:52 fence
drwxr-xr-x.  19 root root 0 Apr 12 18:52 ftrace
drwxr-xr-x.   9 root root 0 Apr 12 18:52 hda
drwxr-xr-x.   4 root root 0 Apr 12 18:52 hda_intel
-r--r--r--.   1 root root 0 Apr 12 18:52 header_event
-r--r--r--.   1 root root 0 Apr 12 18:52 header_page
drwxr-xr-x.  30 root root 0 Apr 12 18:52 i915
drwxr-xr-x.   7 root root 0 Apr 12 18:52 irq
drwxr-xr-x.  22 root root 0 Apr 12 18:52 irq_vectors
drwxr-xr-x.  13 root root 0 Apr 12 18:52 jbd2
drwxr-xr-x.  42 root root 0 Apr 12 18:52 kmem
drwxr-xr-x. 112 root root 0 Apr 12 18:52 mac80211
```

```
drwxr-xr-x.   3 root root 0 Apr 12 18:52 mce
drwxr-xr-x.   7 root root 0 Apr 12 18:52 module
drwxr-xr-x.   3 root root 0 Apr 12 18:52 napi
drwxr-xr-x.   6 root root 0 Apr 12 18:52 net
drwxr-xr-x.   6 root root 0 Apr 12 18:52 power
drwxr-xr-x.   3 root root 0 Apr 12 18:52 ras
drwxr-xr-x.  19 root root 0 Apr 12 18:52 sched
drwxr-xr-x.   7 root root 0 Apr 12 18:52 scsi
drwxr-xr-x.   4 root root 0 Apr 12 18:52 signal
drwxr-xr-x.   5 root root 0 Apr 12 18:52 skb
drwxr-xr-x.   4 root root 0 Apr 12 18:52 sock
drwxr-xr-x.  10 root root 0 Apr 12 18:52 sunrpc
drwxr-xr-x. 552 root root 0 Apr 12 18:52 syscalls
drwxr-xr-x.  14 root root 0 Apr 12 18:52 timer
drwxr-xr-x.   3 root root 0 Apr 12 18:52 udp
drwxr-xr-x.   6 root root 0 Apr 12 18:52 workqueue
drwxr-xr-x.  18 root root 0 Apr 12 18:52 writeback
drwxr-xr-x.  11 root root 0 Apr 12 18:52 xhci-hcd
```

Let's look at what is in the **block** directory for our example **block_rq_complete**:

```
# ll /sys/kernel/debug/tracing/events/block
total 0
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_bio_backmerge
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_bio_bounce
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_bio_complete
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_bio_frontmerge
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_bio_queue
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_getrq
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_plug
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_remap
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_abort
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_complete
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_insert
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_issue
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_remap
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_rq_requeue
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_sleeprq
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_split
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_unplug_io
drwxr-xr-x. 2 root root 0 Apr 12 18:52 block_unplug_timer
-rw-r--r--. 1 root root 0 Apr 12 18:52 enable
-rw-r--r--. 1 root root 0 Apr 12 18:52 filter
```

Next let's see what is in that subdirectory:

```
# ll /sys/kernel/debug/tracing/events/block/block_rq_complete/
total 0
-rw-r--r--. 1 root root 0 Apr 12 18:52 enable
-rw-r--r--. 1 root root 0 Apr 12 18:52 filter
-r--r--r--. 1 root root 0 Apr 12 18:52 format
-r--r--r--. 1 root root 0 Apr 12 18:52 id
```

Finally, we can see the **format** file for the named event and in there is the answer to our problem. (I've reformatted the output just to make it a little more legible):

```
# cat /sys/kernel/debug/tracing/events/block/block_rq_complete/format
name: block_rq_complete
ID: 675
format:
	field:unsigned short common_type;		offset:0;	size:2;	signed:0;
	field:unsigned char common_flags;		offset:2;	size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1;	signed:0;
	field:int common_pid;				offset:4;	size:4;	signed:1;
	field:int common_lock_depth;			offset:8;	size:4;	signed:1;

	field:dev_t dev;				offset:12;	size:4;	signed:0;
	field:sector_t sector;				offset:16;	size:8;	signed:0;
	field:unsigned int nr_sector;			offset:24;	size:4;	signed:0;
	field:int errors;				offset:28;	size:4;	signed:1;
	field:char rwbs[RWBS_LEN];			offset:32;	size:8;	signed:0;
	field:__data_loc char[] cmd;			offset:40;	size:4;	signed:0;

print fmt: "%d,%d %s (%s) %llu + %u [%d]", ((unsigned int) ((REC->dev) >> 20)), ((unsigned int)
((REC->dev) & ((1U << 20) - 1))), REC->rwbs, __get_str(cmd), (unsigned long long)REC->sector,
REC->nr_sector, REC->errors
```

Now we can see the 2 variable names that were used previously. FYI, dev is slightly unusual in that the Major No. is bits 20-39 and Minor No. is bits 0-19 so as examples 0x800000 would be 8,0 and 0xfd00002 would be 253,2)

What about the other events? From **perf**, here's the list of pre-defined events. This matches 1:1 with the directories in **/sys/kernel/debug/tracing/events.** So if you want to review all the **format** entries for all these events, you can see it's now easy to find the variables.

```
$ perf list | grep ":" | cut -d: -f1 | uniq | sort
  block
  cfg80211
  drm
  ext4
  fence
  hda
  hda_intel
  i915
```

```
irq
irq_vectors
jbd2
kmem
mac80211
mce
module
napi
net
power
ras
sched
scsi
signal
skb
sock
sunrpc
syscalls
timer
udp
workqueue
writeback
xhci-hcd
```

3.2.7 - What function do these pre-defined tracepoint events call/map to?

This is an often asked question and the answer is not difficult but also is not obvious.

Predefined events appear to discoverable by taking the event function name and preceding it with trace_ and search cscope or your favorite kernel source code viewer for it. eg block:block_rq_issue is trace_block_rq_issue(). Howewver that's not all there is to this....

These trace events are predefined and built into the kernel specifically for ftrace and perf. There is no easy command style list available to determine the actual function that is calling the trace event. The trace_ event itself is embedded in some actual kernel function call and that's what you have to locate. Also, the trace_ event may be found in MORE than 1 location. It's not neccessarily limited to one kernel function call.

Here's an example for block:block_rq_issue:

```
struct request *blk_peek_request(struct request_queue *q)
{
        struct request *rq;
        int ret;

        while ((rq = __elv_next_request(q)) != NULL) {

                rq = blk_pm_peek_request(q, rq);
                if (!rq)
                        break;

                if (!(rq->cmd_flags & REQ_STARTED)) {
                        /*
                         * This is the first time the device driver
                         * sees this request (possibly after
                         * requeueing).  Notify IO scheduler.
                         */
                        if (rq->cmd_flags & REQ_SORTED)
                                elv_activate_rq(q, rq);

                        /*
                         * just mark as started even if we don't start
                         * it, a request that has been delayed should
                         * not be passed by new incoming requests
                         */
                        rq->cmd_flags |= REQ_STARTED;
                        trace_block_rq_issue(q, rq);     ← Here is block_rq_issue
                }

                if (!q->boundary_rq || q->boundary_rq == rq) {
                        q->end_sector = rq_end_sector(rq);
```

```
                        q->boundary_rq = NULL;
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

…. and this function also calls it:

```
void blk_mq_start_request(struct request *rq)
{
        struct request_queue *q = rq->q;

        trace_block_rq_issue(q, rq);        ← This also is block:block_rq_issue

        rq->resid_len = blk_rq_bytes(rq);
        if (unlikely(blk_bidi_rq(rq)))
                rq->next_rq->resid_len = blk_rq_bytes(rq->next_rq);

        blk_add_timer(rq);
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

As you will discover in trying to chase this down, this is all part of the trace point feature now integral to the kernel. If you want to dig further, knock yourself out ☺.

include/trace/events/block.h

```
DEFINE_EVENT(block_rq, block_rq_issue,

        TP_PROTO(struct request_queue *q, struct request *rq),

        TP_ARGS(q, rq)
);
```

3.2.8 – How many events are there that can be probed?

The proverbial " How long is a piece of string....". The answer is 10's of thousands. You can find some initial numbers as follows. I'm showing the ftrace and systemtap for comparison although be advised the counts even for perf are NOT all possible probes as I will explain following:

FTRACE

```
# cat /sys/kernel/debug/tracing/available_filter_functions | wc -l
37996
```

PERF

```
# perf probe -F | wc -l
25112
```

STAP

```
# stap -L 'kernel.function("*")' | wc -l
35871
```

Why isn't this the complete answer? Because modules are not included in kernel functions; and there are trace functions; and there are pre-defined events; and it depends on the release you're looking at..... None the less, you have to start somewhere

3.3 - Probing, Recording, Statistics usages

3.3.1 - Getting registers from probe points

```
# perf probe --add 'tcp_sendmsg %di %si %dx %cx %ax %bx %bp %sp %flags %r8 %r9 %r10 %r11 
%r12 %r13 %r14 %r15'
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with %di %si %dx %cx %ax %bx %bp %sp %flags %r8 %r9 
%r10 %r11 %r12 %r13 %r14 %r15)

You can now use it in all perf tools, such as:

       perf record -e probe:tcp_sendmsg -aR sleep 1
```

```
# perf record -e probe:tcp_sendmsg -aR
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.685 MB perf.data (~29908 samples) ]
```

Only issue I see is that it doesn't show you the registers by name, they're all called **`arg`**. There's a way around that. First, here's what you get from the above:

```
# perf script
         firefox  4309 [001] 417818.286068: probe:tcp_sendmsg: (ffffffff814b3cf0) 
arg1=ffff88044a99fc98 arg2=ffff880355b49c00 arg3=ffff88044a99fe58 arg4=50b 
arg5=ffffffff816777e0 arg6=ffff880355b49c00 arg7=ffff88044a99fe38 arg8=ffff88044a99fc90 
arg9=282 arg10=0 arg11=0 arg12=7e9 arg13=4 arg14=ffff88044a99fe58 arg15=ffff88044a99fc98 
arg16=50b arg17=ffff88044a99fd88
         firefox  4309 [000] 417818.579068: probe:tcp_sendmsg: (ffffffff814b3cf0) 
arg1=ffff88044a99fc98 arg2=ffff88047465a400 arg3=ffff88044a99fe58 arg4=23 
arg5=ffffffff816777e0 arg6=ffff88047465a400 arg7=ffff88044a99fe38 arg8=ffff88044a99fc90 
arg9=282 arg10=0 arg11=0 arg12=7e9 arg13=4 arg14=ffff88044a99fe58 arg15=ffff88044a99fc98 
arg16=23 arg17=ffff88044a99fd88
         firefox  4309 [001] 417824.268696: probe:tcp_sendmsg: (ffffffff814b3cf0) 
arg1=ffff88044a99fc98 arg2=ffff880472e63140 arg3=ffff88044a99fe58 arg4=5b5 
arg5=ffffffff816777e0 arg6=ffff880472e63140 arg7=ffff88044a99fe38 arg8=ffff88044a99fc90 
arg9=282 arg10=0 arg11=0 arg12=7e9 arg13=4 arg14=ffff88044a99fe58 arg15=ffff88044a99fc98 
arg16=5b5 arg17=ffff88044a99fd88
         firefox  4309 [004] 417826.283054: probe:tcp_sendmsg: (ffffffff814b3cf0) 
arg1=ffff88044a99fc98 arg2=ffff880456a22cc0 arg3=ffff88044a99fe58 arg4=2e 
arg5=ffffffff816777e0 arg6=ffff880456a22cc0 arg7=ffff88044a99fe38 arg8=ffff88044a99fc90 
arg9=282 arg10=0 arg11=0 arg12=7e9 arg13=4 arg14=ffff88044a99fe58 arg15=ffff88044a99fc98 
arg16=2e arg17=ffff88044a99fd88
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

Can you fix the arg issue? Yes you can. You can assign “names” to variables and registers which makes the output display of your **`perf.dat`** far more legible. BTW, if you know what the struct name is for a specific register, you could always just call it that. For example '**`sock=%cx`**'. Naming convention is yours to choose.

```
# perf probe --add 'tcp_sendmsg rdi=%di rsi=%si rdx=%dx rcx=%cx rax=%ax rbx=%bx rbp=%bp rsp=
```

```
%sp rflags=%flags r8=%r8 r9=%r9 r10=%r10 r11=%r11 r12=%r12 r13=%r13 r14=%r14 r15=%r15'
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with rdi=%di rsi=%si rdx=%dx rcx=%cx rax=%ax rbx=%bx
rbp=%bp rsp=%sp rflags=%flags r8=%r8 r9=%r9 r10=%r10 r11=%r11 r12=%r12 r13=%r13 r14=%r14
r15=%r15)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

```
# perf record -e probe:tcp_sendmsg -aR
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.701 MB perf.data (~30647 samples) ]
```

```
# perf script
         firefox  4309 [000] 420919.224878: probe:tcp_sendmsg: (ffffffff814b3cf0)
rdi=ffff88044a99fc98 rsi=ffff88045871e080 rdx=ffff88044a99fe58 rcx=5b5 rax=ffffffff816777e0
rbx=ffff88045871e080 rbp=ffff88044a99fe38 rsp=ffff88044a99fc90 rflags=282 r8=0 r9=0 r10=7e9
r11=4 r12=ffff88044a99fe58 r13=ffff88044a99fc98 r14=5b5 r15=ffff88044a99fd88
         firefox  4309 [000] 420920.255058: probe:tcp_sendmsg: (ffffffff814b3cf0)
rdi=ffff88044a99fc98 rsi=ffff8804758cc900 rdx=ffff88044a99fe58 rcx=2e rax=ffffffff816777e0
rbx=ffff8804758cc900 rbp=ffff88044a99fe38 rsp=ffff88044a99fc90 rflags=282 r8=0 r9=0 r10=7e9
r11=4 r12=ffff88044a99fe58 r13=ffff88044a99fc98 r14=2e r15=ffff88044a99fd88
         firefox  4309 [001] 420920.258868: probe:tcp_sendmsg: (ffffffff814b3cf0)
rdi=ffff88044a99fc98 rsi=ffff880458543b80 rdx=ffff88044a99fe58 rcx=143 rax=ffffffff816777e0
rbx=ffff880458543b80 rbp=ffff88044a99fe38 rsp=ffff88044a99fc90 rflags=282 r8=0 r9=0 r10=7e9
r11=4 r12=ffff88044a99fe58 r13=ffff88044a99fc98 r14=143 r15=ffff88044a99fd88
         firefox  4309 [001] 420920.258914: probe:tcp_sendmsg: (ffffffff814b3cf0)
rdi=ffff88044a99fc98 rsi=ffff880458543b80 rdx=ffff88044a99fe58 rcx=26 rax=ffffffff816777e0
rbx=ffff880458543b80 rbp=ffff88044a99fe38 rsp=ffff88044a99fc90 rflags=282 r8=0 r9=0 r10=7e9
r11=4 r12=ffff88044a99fe58 r13=ffff88044a99fc98 r14=26 r15=ffff88044a99fd88
```

I've since discovered you can also get the **rip, cs, ss** using **rip=%ip cs=%cs ss=%ss**

```
# perf probe --add 'tcp_sendmsg rip=%ip rdi=%di rsi=%si rdx=%dx rcx=%cx rax=%ax rbx=%bx rbp=
%bp rsp=%sp rflags=%flags r8=%r8 r9=%r9 r10=%r10 r11=%r11 r12=%r12 r13=%r13 r14=%r14 r15=%r15
cs=%cs ss=%ss'

Added new event:
  probe:tcp_sendmsg_1  (on tcp_sendmsg with rip=%ip rdi=%di rsi=%si rdx=%dx rcx=%cx rax=%ax
rbx=%bx rbp=%bp rsp=%sp rflags=%flags r8=%r8 r9=%r9 r10=%r10 r11=%r11 r12=%r12 r13=%r13 r14=
%r14 r15=%r15 cs=%cs ss=%ss)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_1 -aR sleep 1


# perf record -e probe:tcp_sendmsg_1 -a
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.715 MB perf.data (170 samples) ]


# perf script > perf-script
```

```
# head perf-script
   Socket Thread 25905 [001] 33143.273803: probe:tcp_sendmsg_1: (ffffffff81576bd0)
rip=ffffffff81576bd1 rdi=ffff88011706bd80 rsi=ffff8800d2f12d00 rdx=ffff88011706be80 rcx=205
rax=ffffffff81a2ab60 rbx=ffff88011706bd80 rbp=ffff88011706bcf0 rsp=ffff88011706bcc8
rflags=282 r8=0 r9=0 r10=92e r11=0 r12=ffff8800a1953980 r13=ffff88011706be80 r14=205 r15=0
cs=10 ss=18
   Socket Thread 25905 [001] 33143.316599: probe:tcp_sendmsg_1: (ffffffff81576bd0)
rip=ffffffff81576bd1 rdi=ffff88011706bd80 rsi=ffff8800d2f12d00 rdx=ffff88011706be80 rcx=33
rax=ffffffff81a2ab60 rbx=ffff88011706bd80 rbp=ffff88011706bcf0 rsp=ffff88011706bcc8
rflags=282 r8=0 r9=0 r10=92e r11=0 r12=ffff8800a1953980 r13=ffff88011706be80 r14=33 r15=0
cs=10 ss=18
   Socket Thread 25905 [001] 33143.339716: probe:tcp_sendmsg_1: (ffffffff81576bd0)
rip=ffffffff81576bd1 rdi=ffff88011706bd80 rsi=ffff8800d2f12d00 rdx=ffff88011706be80 rcx=9d
rax=ffffffff81a2ab60 rbx=ffff88011706bd80 rbp=ffff88011706bcf0 rsp=ffff88011706bcc8
rflags=282 r8=0 r9=0 r10=92e r11=0 r12=ffff8800a1953980 r13=ffff88011706be80 r14=9d r15=0
cs=10 ss=18
   Socket Thread 25905 [001] 33143.339829: probe:tcp_sendmsg_1: (ffffffff81576bd0)
rip=ffffffff81576bd1 rdi=ffff88011706bd80 rsi=ffff8800d2f12d00 rdx=ffff88011706be80 rcx=163
rax=ffffffff81a2ab60 rbx=ffff88011706bd80 rbp=ffff88011706bcf0 rsp=ffff88011706bcc8
rflags=282 r8=0 r9=0 r10=92e r11=0 r12=ffff8800a1953980 r13=ffff88011706be80 r14=163 r15=0
cs=10 ss=18
- - - - - - - - - - - 8< - - - - - - - - - - - -

crash> dis ffffffff81576bd1
0xffffffff81576bd1 <tcp_sendmsg+0x1>:   (bad)
```

3.3.2 - Adding multiple probe points "automagically" based on a string

When you want to add a probe to a function, sometimes there's a sub-function called and you want to probe that but the source code has multiple places in the function that the call is made. Do you have to probe each and everyone? No. You can supply what perf calls a "lazy matching pattern" and it will create the multiple probes for you in one simple go. For example. I'm tracing a problem and discover that **tcp_sendmsg()** has calls to **tcp_push()** and I'm not sure which one is the one we take. I can manually create a probe for each. Or.... let perf do that for you. First let's look at the function. Indeed, let's go wider, any sub-function called with the string **tcp_push** in it:

```
# perf probe -L tcp_sendmsg | grep tcp_push
    190                         __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
    192                         tcp_push_one(sk, mss_now);
    199                         tcp_push(sk, flags & ~MSG_MORE, mss_now,
    211             tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
```

So there are 4 calls. 2 to **tcp_push()**, 1 to **__tcp_push_pending_frames()** and 1 to **tcp_push_one()**

```
# perf probe --add 'tcp_sendmsg;*tcp_push*'
Added new events:
  probe:tcp_sendmsg    (on tcp_sendmsg)
  probe:tcp_sendmsg_1  (on tcp_sendmsg)
  probe:tcp_sendmsg_2  (on tcp_sendmsg)
  probe:tcp_sendmsg_3  (on tcp_sendmsg)
```

And to list them....

```
# perf probe -l
  probe:tcp_sendmsg    (on tcp_sendmsg:211@net/ipv4/tcp.c)
  probe:tcp_sendmsg_1  (on tcp_sendmsg:192@net/ipv4/tcp.c)
  probe:tcp_sendmsg_2  (on tcp_sendmsg:199@net/ipv4/tcp.c)
  probe:tcp_sendmsg_3  (on tcp_sendmsg:190@net/ipv4/tcp.c)
```

What if I ONLY wanted the 2 at lines 199 and 211 to **tcp_push**. Well this is lazy pattern matching after all. So you can do this:

```
# perf probe --add 'tcp_sendmsg;tcp_push(*'
Added new events:
  probe:tcp_sendmsg    (on tcp_sendmsg)
  probe:tcp_sendmsg_1  (on tcp_sendmsg)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_1 -aR sleep 1
```

```
# perf probe -l
  probe:tcp_sendmsg    (on tcp_sendmsg:211@net/ipv4/tcp.c)
  probe:tcp_sendmsg_1  (on tcp_sendmsg:199@net/ipv4/tcp.c)
```

If you are wondering why I used the pattern "**`tcp_push(*`**". If you use just **`tcp_push`**, you will only get the entry point. If you use **`tcp_push*`** you will also get **`tcp_push_one()`**. It's matching the string.

Just a comment to be clear about this. The **`perf.data`** will record the caller, not the callee. So if you review the output with **`perf script`** you will see **`tcp_sendmsg()`** BUT, the address at the end is the **`%RIP`**. None the less, you need to be aware of what's going on. The address is NOT the actual call as corroborated by **`crash`**. It is the start of the source line number corresponding to the call.

```
# perf script
   Socket Thread 25905 [000] 176337.882895: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.325390: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [002] 176339.561641: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.602420: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.610785: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [003] 176339.614205: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.617693: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.625891: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.626079: probe:tcp_sendmsg: (ffffffff81576c78)
   Socket Thread 25905 [001] 176339.626161: probe:tcp_sendmsg: (ffffffff81576c78)
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

Here's where one of the probes was inserted:

```
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/net/ipv4/tcp.c: 1264
0xffffffff81576c78 <tcp_sendmsg+0xa8>:  movzbl 0x5d3(%r13),%ecx
0xffffffff81576c80 <tcp_sendmsg+0xb0>:  mov    -0x40(%rbp),%r8d
0xffffffff81576c84 <tcp_sendmsg+0xb4>:  mov    %r13,%rdi
0xffffffff81576c87 <tcp_sendmsg+0xb7>:  mov    -0x4c(%rbp),%edx
0xffffffff81576c8a <tcp_sendmsg+0xba>:  mov    -0x44(%rbp),%esi
0xffffffff81576c8d <tcp_sendmsg+0xbd>:  mov    %eax,-0x48(%rbp)
0xffffffff81576c90 <tcp_sendmsg+0xc0>:  and    $0xf,%ecx
0xffffffff81576c93 <tcp_sendmsg+0xc3>:  callq  0xffffffff815733c0 <tcp_push>
0xffffffff81576c98 <tcp_sendmsg+0xc8>:  mov    -0x48(%rbp),%eax

crash> dis -l tcp_sendmsg | grep \<tcp_push\>
0xffffffff81576c93 <tcp_sendmsg+0xc3>: callq  0xffffffff815733c0 <tcp_push>
0xffffffff81577208 <tcp_sendmsg+0x638>: callq  0xffffffff815733c0 <tcp_push>
```

One last thing, you can add variables also to the multi probe:

```
# perf probe --add 'tcp_sendmsg;tcp_push(* copied flags' -f
Added new events:
```

```
  probe:tcp_sendmsg_1  (on tcp_sendmsg with copied flags)
  probe:tcp_sendmsg_2  (on tcp_sendmsg with copied flags)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_2 -aR sleep 1
```

3.3.3 - How can I see backtraces?

This is all dependent on an option you use.... **-a** tells the recorder you want ALL CPU's, and **-g** tells it you want the backtraces recorded.

```
# perf record -e block:block_rq_issue -e block:block_rq_complete -ag
^C
[ perf record: Woken up 82 times to write data ]
[ perf record: Captured and wrote 21.302 MB perf.data (~930692 samples) ]
```

```
# perf script
jbd2/dm-0-8   646 [000] 425434.609222: block:block_rq_issue: 8,0 WS 0 () 11168024 + 8 [jbd2/dm-0-8]
ffffffff812770f6 ftrace_profile_block_rq_issue ([kernel.kallsyms])
ffffffff81275b90 blk_peek_request ([kernel.kallsyms])
ffffffff81392a93 scsi_request_fn ([kernel.kallsyms])
ffffffff81273921 __blk_run_queue ([kernel.kallsyms])
ffffffff8128cf6b cfq_insert_request ([kernel.kallsyms])
ffffffff8126ed78 elv_insert ([kernel.kallsyms])
ffffffff8126eea0 __elv_add_request ([kernel.kallsyms])
ffffffff8127656c blk_queue_bio ([kernel.kallsyms])
ffffffff81274cb0 generic_make_request ([kernel.kallsyms])
ffffffff81275080 submit_bio ([kernel.kallsyms])
ffffffff811c7bad submit_bh ([kernel.kallsyms])
ffffffff811ca368 __block_write_full_page ([kernel.kallsyms])
ffffffff811ca5b0 block_write_full_page_endio ([kernel.kallsyms])
ffffffff811ca605 block_write_full_page ([kernel.kallsyms])
ffffffffa03d5b22 ext4_writepage (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/ext4/ext4.ko)
ffffffff8113b387 __writepage ([kernel.kallsyms])
ffffffff8113c64d write_cache_pages ([kernel.kallsyms])
ffffffff8113c934 generic_writepages ([kernel.kallsyms])
ffffffffa03b54d7 journal_submit_inode_data_buffers (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
ffffffffa03b59ed jbd2_journal_commit_transaction (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
ffffffffa03bba38 kjournald2 (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
ffffffff810a0fce kthread ([kernel.kallsyms])
ffffffff8100c28a child_rip ([kernel.kallsyms])

jbd2/dm-0-8   646 [000] 425434.609394: block:block_rq_issue: 8,0 WS 0 () 51774928 + 88 [jbd2/dm-0-8]
ffffffff812770f6 ftrace_profile_block_rq_issue ([kernel.kallsyms])
ffffffff81275b90 blk_peek_request ([kernel.kallsyms])
ffffffff81392a93 scsi_request_fn ([kernel.kallsyms])
ffffffff81273ac2 __generic_unplug_device ([kernel.kallsyms])
ffffffff81273afe generic_unplug_device ([kernel.kallsyms])
ffffffff8126f944 blk_unplug ([kernel.kallsyms])
ffffffffa0004d9f dm_table_unplug_all (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/drivers/md/dm-mod.ko)
ffffffffa0001036 dm_unplug_all (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/drivers/md/dm-mod.ko)
ffffffff8126f944 blk_unplug ([kernel.kallsyms])
ffffffff8126f992 blk_backing_dev_unplug ([kernel.kallsyms])
ffffffff811c73ce block_sync_page ([kernel.kallsyms])
ffffffff81127578 sync_page ([kernel.kallsyms])
ffffffff81539bef __wait_on_bit ([kernel.kallsyms])
ffffffff811277b3 wait_on_page_bit ([kernel.kallsyms])
ffffffff81127bdb wait_on_page_writeback_range ([kernel.kallsyms])
ffffffff81127c9f filemap_fdatawait ([kernel.kallsyms])
ffffffffa03b5e59 jbd2_journal_commit_transaction (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
ffffffffa03bba38 kjournald2 (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
ffffffff810a0fce kthread ([kernel.kallsyms])
ffffffff8100c28a child_rip ([kernel.kallsyms])

jbd2/dm-0-8   646 [000] 425434.609425: block:block_rq_issue: 8,0 WS 0 () 11095216 + 8 [jbd2/dm-0-8]
ffffffff812770f6 ftrace_profile_block_rq_issue ([kernel.kallsyms])
ffffffff81275b90 blk_peek_request ([kernel.kallsyms])
ffffffff81392a93 scsi_request_fn ([kernel.kallsyms])
ffffffff81273ac2 __generic_unplug_device ([kernel.kallsyms])
ffffffff81273afe generic_unplug_device ([kernel.kallsyms])
```

```
        ffffffff8126f944 blk_unplug ([kernel.kallsyms])
        ffffffffa0004d9f dm_table_unplug_all (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/drivers/md/dm-mod.ko)
        ffffffffa0001036 dm_unplug_all (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/drivers/md/dm-mod.ko)
        ffffffff8126f944 blk_unplug ([kernel.kallsyms])
        ffffffff8126f992 blk_backing_dev_unplug ([kernel.kallsyms])
        ffffffff811c73ce block_sync_page ([kernel.kallsyms])
        ffffffff81127578 sync_page ([kernel.kallsyms])
        ffffffff81539bef __wait_on_bit ([kernel.kallsyms])
        ffffffff811277b3 wait_on_page_bit ([kernel.kallsyms])
        ffffffff81127bdb wait_on_page_writeback_range ([kernel.kallsyms])
        ffffffff81127c9f filemap_fdatawait ([kernel.kallsyms])
        ffffffffa03b5e59 jbd2_journal_commit_transaction (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
        ffffffffa03bba38 kjournald2 (/lib/modules/2.6.32-573.12.1.el6.x86_64/kernel/fs/jbd2/jbd2.ko)
        ffffffff810a0fce kthread ([kernel.kallsyms])
        ffffffff8100c28a child_rip ([kernel.kallsyms])
       - - - - - - - - - - - - - - - - - - 8< - - - - - - - - - - - - - - - - - - - -
```

There is a change in this **`perf script`** behavior in later RHEL6 and into RHEL7 versions of **`perf`**. They no longer show the full "kernel" address. Let's look at an extracted example:

```
sleep 23649 [003] 780638.470134: probe:file_read_actor: (ffffffff8112d950)
                  32d951 file_read_actor (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  399aaa do_sync_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  39a3a5 vfs_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  39a6f1 sys_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  20b0d2 system_call_fastpath (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                   17557 read (/lib64/ld-2.12.so)
```

At first I thought perhaps something was broken but actually these are addresses relative to the associated library or debug file. A dump of the raw record (**`perf script -D`**) shows the actual addresses are still held in the **`perf.data`** file)

```
0xc8030 [0x98]: event: 9
.
. ... raw event: size 152 bytes
.  0000:  09 00 00 00 01 00 98 00 51 d9 12 81 ff ff ff ff  ........Q.......
.  0010:  61 5c 00 00 61 5c 00 00 8c da b5 8e fc c5 02 00  a\..a\..........
.  0020:  03 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
.  0030:  08 00 00 00 00 00 00 00 80 ff ff ff ff ff ff ff  ................
.  0040:  51 d9 12 81 ff ff ff ff aa 9a 19 81 ff ff ff ff  Q...............
.  0050:  a5 a3 19 81 ff ff ff ff f1 a6 19 81 ff ff ff ff  ................
.  0060:  d2 b0 00 81 ff ff ff ff 00 fe ff ff ff ff ff ff  ................
.  0070:  57 75 81 f1 3d 00 00 00 1c 00 00 00 5f 04 01 00  Wu..=......._...
.  0080:  61 5c 00 00 ff ff ff ff 00 00 00 00 50 d9 12 81  a\..........P...
.  0090:  ff ff ff ff 00 00 00 00                          ........
.
3 780638470134412 0xc8030 [0x98]: PERF_RECORD_SAMPLE(IP, 0x1): 23649/23649: 0xffffffff8112d951 period: 1 addr: 0
... FP chain: nr:8
.....  0: ffffffffffffff80
.....  1: ffffffff8112d951
.....  2: ffffffff81199aaa
.....  3: ffffffff8119a3a5
.....  4: ffffffff8119a6f1
.....  5: ffffffff8100b0d2
.....  6: fffffffffffffe00
.....  7: 0000003df1817557
 ... thread: sleep:23649
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
sleep 23649 [003] 780638.470134: probe:file_read_actor: (ffffffff8112d950)
                  32d951 file_read_actor (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  399aaa do_sync_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  39a3a5 vfs_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  39a6f1 sys_read (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
                  20b0d2 system_call_fastpath (/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)
```

```
                    17557 read (/lib64/ld-2.12.so)
```

Now look at the kernel address and the “offset” address side by side

```
.....   0: ffffffffffffff80
.....   1: ffffffff8112d951            32d951 file_read_actor
.....   2: ffffffff81199aaa            399aaa do_sync_read
.....   3: ffffffff8119a3a5            39a3a5 vfs_read
.....   4: ffffffff8119a6f1            39a6f1 sys_read
.....   5: ffffffff8100b0d2            20b0d2 system_call_fastpath
.....   6: fffffffffffffe00
.....   7: 0000003df1817557             17557 read
```

From **crash** using KERNEL: **/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux**:

```
0xffffffff81199a8b <do_sync_read+0xdb>:        callq  0xffffffff811996e0 <wait_on_retry_sync_kiocb>
0xffffffff81199a90 <do_sync_read+0xe0>:        mov    -0xa0(%rbp),%rcx
0xffffffff81199a97 <do_sync_read+0xe7>:        mov    0x20(%r12),%rax
0xffffffff81199a9c <do_sync_read+0xec>:        mov    $0x1,%edx
0xffffffff81199aa1 <do_sync_read+0xf1>:        mov    %r13,%rsi
0xffffffff81199aa4 <do_sync_read+0xf4>:        mov    %rbx,%rdi
0xffffffff81199aa7 <do_sync_read+0xf7>:        callq  *0x20(%rax)
0xffffffff81199aaa <do_sync_read+0xfa>:        cmp    $0xfffffffffffffdee,%rax
0xffffffff81199ab0 <do_sync_read+0x100>:       je     0xffffffff81199a88 <do_sync_read+0xd8>
```

From **objdump** of **/lib64/ld-2.12.so**:

```
0000003df1817550 <__libc_read>:
  3df1817550:     b8 00 00 00 00           mov    $0x0,%eax
  3df1817555:     0f 05                    syscall
  3df1817557:     48 3d 01 f0 ff ff        cmp    $0xfffffffffffff001,%rax
  3df181755d:     73 01                    jae    3df1817560 <__libc_read+0x10>
```

There may be a way to display this in the form of the full kernel address, after all it is held in the **perf.data** file. However as yet, I have not found the “magic” command sequence to make it happen.

3.3.4 - Probe a function by Instruction offset instead of line number

One additional feature of perf probing, all of our examples so far have been based on probe'able line numbers. As it turns out, you can probe a function on an instruction address. Let's say I have a function that I want to add probes at various points (See the highlighted lines I randomly selected)

```
0xffffffff815773dd <tcp_sendmsg+2061>:  movabs $0x160000000000,%rdx
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1918
0xffffffff815773e7 <tcp_sendmsg+2071>:  mov    0x68(%r14),%r11d
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/linux/mm.h: 886
0xffffffff815773eb <tcp_sendmsg+2075>:  add    %rdx,%rdi
0xffffffff815773ee <tcp_sendmsg+2078>:  movabs $0xffff880000000000,%rdx
0xffffffff815773f8 <tcp_sendmsg+2088>:  sar    $0x6,%rdi
0xffffffff815773fc <tcp_sendmsg+2092>:  shl    $0xc,%rdi
0xffffffff81577400 <tcp_sendmsg+2096>:  add    %rdx,%rdi
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1917
0xffffffff81577403 <tcp_sendmsg+2099>:  add    %rax,%rdi
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1881
0xffffffff81577406 <tcp_sendmsg+2102>:  testb  $0xc,0x7c(%r14)
0xffffffff8157740b <tcp_sendmsg+2107>:  je     0xffffffff815775a0 <tcp_sendmsg+2512>
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1887
0xffffffff81577411 <tcp_sendmsg+2113>:  testb  $0x2,0x164(%r13)
0xffffffff81577419 <tcp_sendmsg+2121>:  mov    %r10,-0xb8(%rbp)
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1888
0xffffffff81577420 <tcp_sendmsg+2128>:  movslq %r12d,%r8
0xffffffff81577423 <tcp_sendmsg+2131>:  mov    %r9d,-0xb0(%rbp)
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1887
0xffffffff8157742a <tcp_sendmsg+2138>:  je     0xffffffff815776d4 <tcp_sendmsg+2820>
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/net/sock.h: 1888
0xffffffff81577430 <tcp_sendmsg+2144>:  mov    -0x58(%rbp),%rsi
0xffffffff81577434 <tcp_sendmsg+2148>:  mov    -0xa0(%rbp),%rcx
0xffffffff8157743b <tcp_sendmsg+2155>:  mov    %r8,-0xa8(%rbp)
0xffffffff81577442 <tcp_sendmsg+2162>:  mov    %rsi,%rax
0xffffffff81577445 <tcp_sendmsg+2165>:  add    %r8,%rax
0xffffffff81577448 <tcp_sendmsg+2168>:  sbb    %rdx,%rdx
0xffffffff8157744b <tcp_sendmsg+2171>:  cmp    %rax,-0x3fb8(%rcx)
0xffffffff81577452 <tcp_sendmsg+2178>:  sbb    $0x0,%rdx
0xffffffff81577456 <tcp_sendmsg+2182>:  test   %rdx,%rdx
0xffffffff81577459 <tcp_sendmsg+2185>:  jne    0xffffffff81577705 <tcp_sendmsg+2869>
```

Now let's add our probes for all 3 instructions. Note 2 of these were in the same Source code line.

```
# perf probe --add 'tcp_sendmsg+2075' -f
Added new event:
  probe:tcp_sendmsg_2  (on tcp_sendmsg+2075)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_2 -aR sleep 1
```

```
# perf probe --add 'tcp_sendmsg+2096' -f
Added new event:
  probe:tcp_sendmsg_3  (on tcp_sendmsg+2096)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_3 -aR sleep 1

# perf probe --add 'tcp_sendmsg+2171' -f
Added new event:
  probe:tcp_sendmsg_4  (on tcp_sendmsg+2171)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_4 -aR sleep 1

# perf probe -l
  probe:tcp_sendmsg    (on tcp_sendmsg:211@net/ipv4/tcp.c with RIP)
  probe:tcp_sendmsg_1  (on tcp_sendmsg:199@net/ipv4/tcp.c with RIP)
  probe:tcp_sendmsg_2  (on tcp_sendmsg+2075@net/ipv4/tcp.c)
  probe:tcp_sendmsg_3  (on tcp_sendmsg+2096@net/ipv4/tcp.c)
  probe:tcp_sendmsg_4  (on tcp_sendmsg+2171@net/ipv4/tcp.c)
```

You can use a hex offset:

```
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/linux/skbuff.h: 939
0xffffffff81576e5b <tcp_sendmsg+0x28b>: add    0xe0(%r14),%rsi
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/linux/skbuff.h: 2518
0xffffffff81576e62 <tcp_sendmsg+0x292>: movslq %ecx,%rcx
0xffffffff81576e65 <tcp_sendmsg+0x295>: add    $0x3,%rcx
0xffffffff81576e69 <tcp_sendmsg+0x299>: shl    $0x4,%rcx
0xffffffff81576e6d <tcp_sendmsg+0x29d>: add    %rsi,%rcx
/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/include/linux/skbuff.h: 2520
0xffffffff81576e70 <tcp_sendmsg+0x2a0>: cmp    (%rcx),%rdi
0xffffffff81576e73 <tcp_sendmsg+0x2a3>: je     0xffffffff81577638 <tcp_sendmsg+0xa68>

# perf probe --add 'tcp_sendmsg+0x299' -f
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg+665)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

Are there any restrictions. Well yes.

1. I tried a dozen or so offset addresses in **`__do_page_fault()`**. It would not allow me to do any of them. I've not been able to figure out why yet.
2. An obvious point, you can only probe on an instruction boundary.

3.3.5 - Getting CPU statistics (EG. TLB hits and misses)

You can actually find some really slick CPU statistics. Here's a way of getting TLB hits and misses for a command you want to run. You can do the same for a pid use **-p <PID>**.

```
# perf stat -e dTLB-loads,dTLB-load-misses,dTLB-stores,dTLB-store-misses,iTLB-
loads,iTLB-load-misses df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  42475796  25972020  63% /
tmpfs                  7977236      3288   7973948   1% /dev/shm
/dev/sda1               487652    215233    246819  47% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 299200488  67098840  82% /home

 Performance counter stats for 'df':

     <not counted>      dTLB-loads
             1,451      dTLB-load-misses          #    0.00% of all dTLB cache hits
           146,062      dTLB-stores
               235      dTLB-store-misses
               609      iTLB-loads
             1,210      iTLB-load-misses          #  198.69% of all iTLB cache hits
[17.20%]

       0.001352096 seconds time elapsed
```

What are **dTLB** vs **iTLB**?

- **d** = Data (operand) addresses that caused a TLB hit miss. As we can perform stores or loads, that's why you'll see both reported
- **i** = Instruction addresses. You will only see instruction fetches reported (instruction fetches into the CPU's pipeline).

BTW. Can you think of any examples where we modify instructions? There are a few. These only happen in the kernel....

And the answer to that last question...

1. **ftrace** modifies kernel code putting nop's in and then changing code to **call ftrace()**
2. **stap** modifies code inserting an **int3** instruction
3. **perf** probing modifies code inserting a **jmp** as the probe
4. and I'm sure they could be others....

So why isn't there a iTLB -store? Because when the above noted examples modify the code, they're actually storing 'data' into a page, albeit, an instruction page.

What else can you get from perf that's hardware specific?

```
# perf list | grep Hardware
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  cache-references                                   [Hardware event]
```

```
  cache-misses                                       [Hardware event]
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  ref-cycles                                         [Hardware event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
```

3.3.6 - Restricting event counting (E.G. Just userspace or kernel)

There's another nice feature with perf. You can limit the event counting to various aspects of the System

```
u - user-space counting
k - kernel counting
h - hypervisor counting
G - guest counting (in KVM guests)
H - host counting (not in KVM guests)
p - precise level
         The p modifier can be used for specifying how precise the instruction
         address should be. The p modifier can be specified multiple times:
              0 - SAMPLE_IP can have arbitrary skid
              1 - SAMPLE_IP must have constant skid
              2 - SAMPLE_IP requested to have 0 skid
              3 - SAMPLE_IP must have 0 skid
         For Intel systems precise event sampling is implemented with PEBS which
         supports up to precise-level 2.
S - read sample value (PERF_SAMPLE_READ)
D - pin the event to the PMU
```

Here's a few examples to give you an idea:

```
# perf stat -e r00c0:u,rfec1:u,r00c5:u,r1fc7:u,r01cb:u,r003c:u -ag sleep 5

 Performance counter stats for 'system wide':

       28,853,903      r00c0:u                                                     [ 0.03%]
                0      rfec1:u                                                     [ 0.02%]
          122,543      r00c5:u                                                     [ 0.02%]
           34,292      r1fc7:u                                                     [ 0.01%]
                0      r01cb:u                                                     [ 0.01%]
      265,606,366      r003c:u                                                     [ 0.00%]

      5.001682929 seconds time elapsed

# perf stat -e cpu-cycles:u df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31961844  36485972  47% /
tmpfs                  7977236       564   7976672   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315637320  50662008  87% /home

 Performance counter stats for 'df':

          602,365      cpu-cycles:u

      0.000565763 seconds time elapsed
```

```
# perf stat -e cpu-cycles:k df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31961856  36485960  47% /
tmpfs                  7977236       564   7976672   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315637320  50662008  87% /home

 Performance counter stats for 'df':

           776,074      cpu-cycles:k

# perf stat -e cpu-cycles:pp df
```

(Note on Intel, ppp is not supported)

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31949844  36497972  47% /
tmpfs                  7977236       564   7976672   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315637348  50661980  87% /home

 Performance counter stats for 'df':

         1,247,184      cpu-cycles:pp

       0.002164481 seconds time elapsed

# perf record -ag -e cycles:p -- sleep 10
```

You can combine multiple restrictive events however I've noted a quirk which may be specific to a release and may be fixed at some time. The first 2 work:

```
# perf stat -e cycles:k,cycles:u -- sleep 10
# perf stat --event=cycles:{k,u} -- sleep 10
```

This one however does not work

```
# perf stat -e 'cycles:{k,u}' -- sleep 10
invalid or unsupported event: 'cycles:{k,u}'
Run 'perf list' for a list of valid events

 usage: perf stat [<options>] [<command>]

    -e, --event <event>   event selector. use 'perf list' to list available events
```

3.3.7 - Selective “tracing” - Filtering/Testing flags, states etc.

What if you only want to produce output when a specific condition is met? Let's check for **flags** set to **0x400000** only. You can use **--filter** when you record.

NOTE. Some data is display in decimal and some in Hex. However, perf display does NOT differentiate when/which it is using. Therefore if filtering, remember to use 0x...... if the value contains hex characters to ensure proper filtering

```
# perf probe --add '__do_page_fault:173 tsk->flags'
Added new event:
  probe:__do_page_fault (on __do_page_fault:173 with flags=tsk->flags)

You can now use it in all perf tools, such as:

      perf record -e probe:__do_page_fault -aR sleep 1

# perf probe -V __do_page_fault:173
Available variables at __do_page_fault:173
        @<__do_page_fault+310>
                long unsigned int       address
                struct mm_struct*       mm
                struct task_struct*     tsk
                struct vm_area_struct*  vma
                unsigned int    flags

# perf record -e probe:__do_page_fault --filter 'flags == 0x400000' -ag sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.663 MB perf.data (~28967 samples) ]

# perf script
sleep  9403 [005] 887039.245896: probe:__do_page_fault: (ffffffff8104f146) flags=400000
        ffffffff8104f147 __do_page_fault ([kernel.kallsyms])
        ffffffff8153f48e do_page_fault ([kernel.kallsyms])
        ffffffff8153c835 page_fault ([kernel.kallsyms])
        ffffffff8129e0e8 clear_user ([kernel.kallsyms])
        ffffffff811ee26d padzero ([kernel.kallsyms])
        ffffffff811f0083 load_elf_binary ([kernel.kallsyms])
        ffffffff8119b167 search_binary_handler ([kernel.kallsyms])
        ffffffff8119b6d7 do_execve ([kernel.kallsyms])
        ffffffff810095ea sys_execve ([kernel.kallsyms])
        ffffffff8100b52a stub_execve ([kernel.kallsyms])
              39c82acde7 __execve (/lib64/libc-2.12.so)

sleep  9403 [005] 887039.245904: probe:__do_page_fault: (ffffffff8104f146) flags=400000
        ffffffff8104f147 __do_page_fault ([kernel.kallsyms])
        ffffffff8153f48e do_page_fault ([kernel.kallsyms])
```

```
        ffffffff8153c835 page_fault ([kernel.kallsyms])
        ffffffff811ef0e7 load_elf_binary ([kernel.kallsyms])
        ffffffff8119b167 search_binary_handler ([kernel.kallsyms])
        ffffffff8119b6d7 do_execve ([kernel.kallsyms])
        ffffffff810095ea sys_execve ([kernel.kallsyms])
        ffffffff8100b52a stub_execve ([kernel.kallsyms])
              39c82acde7 __execve (/lib64/libc-2.12.so)
```

You can also look at an element within a structure and test it. Let's take the following example use **`tcp_sendmsg()`**. Note, that the element name that we are going to be testing in '**`record`**' does not require the full structure name, just the final element name... in this case **`skc_state.`**

Note. The following example will not work on RHEL6/RHEL7. However as a challenge, see if you can figure out why. If you want to see the RHEL6 corrected 'probe --add' then look in the One-Liners I listed earlier in 3.4.4.1.2. There's a reason I've left this asis. It is impossible to keep up with what structures will and will not change with each release so any and all examples included in this document should be respected accordingly for this limitation. If you cut and paste and one does not work, you should look deeper into the code/function, the structures and even the perf command itself as things often change with release changes in all of these aspects.

```
# perf probe -V tcp_sendmsg
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                size_t  size
                struct kiocb*   iocb
                struct socket*  sock
```

Let's find the element we're interested in:

```
struct socket {
   [0x0] socket_state state;
   [0x4] int type_begin[];
   [0x4] short type;
   [0x8] int type_end[];
   [0x8] unsigned long flags;
  [0x10] struct fasync_struct *fasync_list;
  [0x18] wait_queue_head_t wait;
  [0x30] struct file *file;
  [0x38] struct sock *sk;
  [0x40] const struct proto_ops *ops;
}
SIZE: 0x48
```

```
struct sock {
    [0x0] struct sock_common __sk_common;
   [0x40] int flags_begin[];
   [0x40] unsigned int sk_shutdown : 2;
   [0x40] unsigned int sk_no_check : 2;
- - - - - - - - - - 8< - - - - - - - - - -
```

```
struct sock_common {
         union {
   [0x0]     struct hlist_node skc_node;
   [0x0]     struct hlist_nulls_node skc_nulls_node;
         };
  [0x10] atomic_t skc_refcnt;
  [0x14] unsigned int skc_hash;
  [0x18] unsigned short skc_family;
  [0x1a] volatile unsigned char skc_state;
- - - - - - - - - - 8< - - - - - - - - - -
```

```
# perf probe --add 'tcp_sendmsg size sock->sk->__sk_common.skc_state'
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg with size skc_state=sock->sk-
>__sk_common.skc_state)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

Note the highlighted name that perf has given to our structure sequence... **`skc-state`**

```
# perf record -e probe:tcp_sendmsg -ag sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.686 MB perf.data (~29960 samples) ]

# perf script
firefox  4262 [000] 889024.862036: probe:tcp_sendmsg: (ffffffff814b3f60) size=5a6 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)

firefox  4262 [000] 889025.364757: probe:tcp_sendmsg: (ffffffff814b3f60) size=185 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)

firefox  4262 [000] 889025.591112: probe:tcp_sendmsg: (ffffffff814b3f60) size=13c skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
```

```
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

Now let's re-test but this time check **`size`** and **`skc_state`** for specific ranges. **Note this is checking for a size of 400 (decimal). You could/should use 0x400 when checking hexadecimal sized fields.**

```
# perf record -e probe:tcp_sendmsg --filter 'size > 400 && skc_state == 1' -ag sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.663 MB perf.data (~28981 samples) ]

# perf script
firefox  4262 [001] 889500.448524: probe:tcp_sendmsg: (ffffffff814b3f60) size=5c5 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)

firefox  4262 [001] 889501.232660: probe:tcp_sendmsg: (ffffffff814b3f60) size=5a6 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)

firefox  4262 [000] 889501.455222: probe:tcp_sendmsg: (ffffffff814b3f60) size=445 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)

firefox  4262 [000] 889501.455783: probe:tcp_sendmsg: (ffffffff814b3f60) size=455 skc_state=1
        ffffffff814b3f61 tcp_sendmsg ([kernel.kallsyms])
        ffffffff81456b59 sys_sendto ([kernel.kallsyms])
        ffffffff8100b0d2 system_call_fastpath ([kernel.kallsyms])
              39c8a0edac __libc_send (/lib64/libpthread-2.12.so)
```

I think you get the picture now. You can certainly gather information a lot simpler than with **`stap`**. That doesn't mean that **`stap`** is no longer necessary. However, with **`perf`** you can gather much of the information we could have previously written an **`stap`** script for, but now it's a simple command line.

FILTERING – Just what can you do with “--filter”?

Here's a summary of one-liners that might help show the range of filtering that you can do. It's by no means complete. Unfortunately, I've not discovererd any documentation that clearly outlines all the filtering operands that are available. I have spent considerable hours trying various that didn't work so what's listed here are those I have tried and do work. I have not as yet, discovered any way to test for a specific bit in a flag/word.

Filtering using predefined tracepoint events and their variables:

Trace all block completions, of size greater than 200 Kbytes:
perf record -e block:block_rq_complete --filter 'nr_sector > 200' -a

Trace all block completions, of size at least 100 Kbytes:
perf record -e block:block_rq_complete --filter 'nr_sector => 100' -ag

Trace all block completions, synchronous writes only:
perf record -e block:block_rq_complete --filter 'rwbs == "WS"' -a

Trace all block completions, specifically any type of writes:
perf record -e block:block_rq_complete --filter 'rwbs ~ "*W*"' -a

Filtering using probed events with user defined variables:

Trace when the bytes (alias) variable is less than 100:
perf probe --add 'tcp_sendmsg bytes=%cx'
perf record -e probe:tcp_sendmsg --filter 'bytes < 100' -ag

Trace when size is non-zero, AND state is not TCP_ESTABLISHED(1):
perf probe --add 'tcp_sendmsg size sock->state'
perf record -e probe:tcp_sendmsg --filter 'size > 0 && state != 1' -a

Trace when size is between 40 AND 400, AND state is not TCP_ESTABLISHED(1):
perf probe --add 'tcp_sendmsg size sock->state'
perf record -e probe:tcp_sendmsg --filter '(size >= 40 && size <= 400) && state != 1' -a

Trace when size is between 40 or less OR 400 or greater, AND state is not TCP_ESTABLISHED(1):
perf probe --add 'tcp_sendmsg size sock->state'
perf record -e probe:tcp_sendmsg --filter '(size <= 40 || size >= 400) && state != 1' -a

Trace page fault handling at line 173 and match task flags specifically equal to 0x400000:
perf probe --add '__do_page_fault:173 flags=tsk->flags'
perf record -e probe:__do_page_fault --filter 'flags == 0x400000' -ag -- sleep 5

Trace page fault handling at line 173 and match task flags specifically 0x400000 OR 0x402000:
perf probe --add '__do_page_fault:173 tsk->flags'

perf record -e probe:__do_page_fault --filter 'flags == 0x400000 || flags == 0x402000' -ag sleep 10

Trace when size is between 40 AND 400, AND state is 0 OR 2:

 # perf probe --add 'tcp_sendmsg size sock->state'

 # perf record -e probe:tcp_sendmsg --filter '(size >= 40 && size <= 400) && (state == 0 || state == 2)' -a

Trace block request issues for only device 8,0 (Major No. is shifted 20 bits, Minor No. is lower 20 bits)

 # perf record -e block:block_rq_issue -a --filter 'dev == 0x800000' -- sleep 10

Relational filter operators allowed	
==	Equal to
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
<	Less than
>	Greater than
~	Wildcard String compare

Logical multiple filter operators alllowed	
&&	And
||	Or

3.3.8 - Displaying strings instead of hex addresses

There are times when you are looking at variables and discover in your recording and reporting phase that the address is s pointer to a string variable. One caveat before going any further. **I cannot get strings to display correctly on RHEL6.** The examples were taken from a RHEL7 system which does allow this to work.

Here's a RHEL7 example of the lack of string definition problem. Add a problem and ask to see the **filename**.

```
# perf probe --add 'do_sys_open filename'
# perf record -e probe:do_sys_open -a
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.615 MB perf.data (33 samples) ]

# perf script
 tracker-miner-f  3179 [003]  8970.528499: probe:do_sys_open_1: (ffffffff811dd710) filename=125ebe0
            pool  3200 [002]  8970.529905: probe:do_sys_open_1: (ffffffff811dd710) filename=7fe32c54abb0
 tracker-miner-f  3179 [003]  8970.532147: probe:do_sys_open_1: (ffffffff811dd710) filename=1270890
            pool  3200 [002]  8970.532833: probe:do_sys_open_1: (ffffffff811dd710) filename=7fe32c54abb0
            pool  3200 [002]  8970.533173: probe:do_sys_open_1: (ffffffff811dd710) filename=7fe32c54abb0
            pool  3200 [002]  8970.534403: probe:do_sys_open_1: (ffffffff811dd710) filename=7fe313ffd9c0
            pool  3200 [002]  8970.535990: probe:do_sys_open_1: (ffffffff811dd710) filename=7fe32c54abb0
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

It displays the **filename** pointer. How can I display the actual string of data for the **filename**? Use the **:string** identifier.

```
# perf probe --add 'do_sys_open filename:string'
Added new event:
  probe:do_sys_open    (on do_sys_open with filename:string)

You can now use it in all perf tools, such as:

        perf record -e probe:do_sys_open -aR sleep 1

# perf record -e probe:do_sys_open -a
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.719 MB perf.data (903 samples) ]


# perf script
       sleep  4745 [003]   982.578084: probe:do_sys_open: (ffffffff811dd710) filename_string="/etc/ld.so.cache"
       sleep  4745 [003]   982.578118: probe:do_sys_open: (ffffffff811dd710) filename_string="/lib64/libc.so.6"
       sleep  4745 [003]   982.578513: probe:do_sys_open: (ffffffff811dd710) filename_string=""
```

```
tracker-miner-f  3179 [001]   983.410458: probe:do_sys_open: (ffffffff811dd710) filename_string="/root/.hidden"
           pool  3200 [002]   983.413530: probe:do_sys_open: (ffffffff811dd710) filename_string="/dev/urandom"
tracker-miner-f  3179 [001]   983.415669: probe:do_sys_open: (ffffffff811dd710) filename_string="/root/perf.data.old"
           pool  3200 [002]   983.416373: probe:do_sys_open: (ffffffff811dd710) filename_string="/dev/urandom"
           pool  3200 [002]   983.416650: probe:do_sys_open: (ffffffff811dd710) filename_string="/dev/urandom"
           pool  3200 [002]   983.418053: probe:do_sys_open: (ffffffff811dd710) filename_string="/var/tmp/etilqs_SNbzwYfSHn
           pool  3200 [002]   983.419666: probe:do_sys_open: (ffffffff811dd710) filename_string="/dev/urandom"
     irqbalance   656 [000]   984.428066: probe:do_sys_open: (ffffffff811dd710) filename_string="/proc/interrupts"
     irqbalance   656 [000]   984.428240: probe:do_sys_open: (ffffffff811dd710) filename_string="/proc/stat"
```

I did uncover another example which is shown for clarification of when you can and cannot use a variable. The call to **getname_flags()** has a filename passed. Here's a part of the source code:

```
# perf probe -L getname_flags
<getname_flags@/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/fs/namei.c:0>
      0  getname_flags(const char __user *filename, int flags, int *empty)
      1  {
                struct filename *result, *err;
                int len;
                long max;
                char *kname;

      7         result = audit_reusename(filename);
      8         if (result)
                        return result;

     11         result = __getname();
     12         if (unlikely(!result))
     13                 return ERR_PTR(-ENOMEM);
     14         result->refcnt = 1;

                /*
                 * First, try to embed the struct filename inside the names_cache
                 * allocation
                 */
     20         kname = (char *)result + sizeof(*result);
     21         result->name = kname;
     22         result->separate = false;
     23         max = EMBEDDED_NAME_MAX;

         recopy:
     26         len = strncpy_from_user(kname, filename, max);
     27         if (unlikely(len < 0)) {
     28                 err = ERR_PTR(len);
                        goto error;
                }

                /*
                 * Uh-oh. We have a name that's approaching PATH_MAX. Allocate a
                 * separate struct filename so we can dedicate the entire
                 * names_cache allocation for the pathname, and re-do the copy from
```

```
                 * userland.
                 */
    38          if (len == EMBEDDED_NAME_MAX && max == EMBEDDED_NAME_MAX) {
                        kname = (char *)result;

                        result = kzalloc(sizeof(*result), GFP_KERNEL);
    - - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

Now let's look at the variables at 3 different locations (entry point, line 1 and line 27)

```
# perf probe -V getname_flags
Available variables at getname_flags
        @<getname_flags+0>
                char*                   filename
                int                     flags
                int*                    empty
        @<getname+18>
                (No matched variables)
        @<user_path_create+37>
                (No matched variables)
        @<user_path_parent+37>
                (No matched variables)
        @<user_path_at_empty+69>
                (No matched variables)
        @<user_path_mountpoint_at+34>
                (No matched variables)
        @<SyS_symlinkat+52>
                (No matched variables)
```

```
# perf probe -V getname_flags:1
Available variables at getname_flags:1
        @<getname_flags+0>
                char*   filename
                int     flags
                int*    empty
        @<getname_flags+15>
                char*   filename
                int     flags
                int*    empty
```

```
# perf probe -V getname_flags:27
Available variables at getname_flags:27
        @<getname_flags+196>
                char*                   filename
                int                     flags
                int*                    empty
```

```
                long int                max
                struct filename*       result
        @<getname_flags+200>
                char*                   filename
                int                     flags
                int                     len
                int*                    empty
                long int                max
                struct filename*       result
```

We have a variable **filename** in all 3 cases. However, look what happens on RHEL7 if you try to get the variable **filename** at the Entry point (Note the function is renamed to vfs_getname. More on this in the next sub-section):

```
# perf probe --add 'vfs_getname=getname_flags pathname=filename:string' -f
Failed to find 'filename' in this function.
  Error: Failed to add events.
```

It will let you probe at line 1 and in the following I selected line 27 also. However, be careful because when I probed line 38, the **perf.data** would never display anything with **perf script**.

```
# perf probe --add 'vfs_getname=getname_flags:27 pathname=filename:string'
Added new events:
  probe:vfs_getname  (on getname_flags:27 with pathname=filename:string)

You can now use it in all perf tools, such as:

      perf record -e probe:vfs_getname -aR sleep 1


# perf record -e probe:vfs_getname -a -- sleep 10

# perf script
           perf  4137 [000]   849.114491: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/libexec/perf-core/sleep"
           perf  4137 [000]   849.114519: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/local/sbin/sleep"
           perf  4137 [000]   849.114532: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/local/bin/sleep"
           perf  4137 [000]   849.114544: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/sbin/sleep"
           perf  4137 [000]   849.114554: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/bin/sleep"
          sleep  4137 [001]   849.115174: probe:vfs_getname: (ffffffff811eeca8) pathname="/etc/ld.so.preload"
          sleep  4137 [001]   849.115197: probe:vfs_getname: (ffffffff811eeca8) pathname="/etc/ld.so.cache"
          sleep  4137 [001]   849.115237: probe:vfs_getname: (ffffffff811eeca8) pathname="/lib64/libc.so.6"
          sleep  4137 [001]   849.115630: probe:vfs_getname: (ffffffff811eeca8) pathname="/usr/lib/locale/locale-archive"
tracker-miner-f  3179 [000]   849.969304: probe:vfs_getname: (ffffffff811eeca8) pathname="/root/perf.data.old"
tracker-miner-f  3179 [000]   849.969411: probe:vfs_getname: (ffffffff811eeca8) pathname="/root/perf.data.old"
tracker-miner-f  3179 [000]   849.969432: probe:vfs_getname: (ffffffff811eeca8) pathname="/root/.hidden"
       nautilus  3169 [002]   849.969882: probe:vfs_getname: (ffffffff811eeca8) pathname="MARK: nautilus nautilus_directo
tracker-miner-f  3179 [000]   849.969940: probe:vfs_getname: (ffffffff811eeca8) pathname="/root/perf.data.old"
       nautilus  3169 [002]   849.969970: probe:vfs_getname: (ffffffff811eeca8) pathname="MARK: nautilus nautilus_directo
       nautilus  3169 [002]   849.970050: probe:vfs_getname: (ffffffff811eeca8) pathname="MARK: nautilus nautilus_directo
tracker-miner-f  3179 [000]   849.970121: probe:vfs_getname: (ffffffff811eeca8) pathname="/root/perf.data.old"
```

3.3.9 - Renaming events (functions) like you rename variables

You might be wondering what the heck does that **vfs_getname=** do in section 3.4.3.15.34 previously? Well not only can you add your own name to variables, you can add your own event name to functions also:

```
# perf probe --add 'Donald_Duck=getname_flags:27 pathname=filename:string'
Added new events:
  probe:Donald_Duck    (on getname_flags:27 with pathname=filename:string)

You can now use it in all perf tools, such as:

      perf record -e probe:Donald_Duck -aR sleep 1

# perf record -e probe:Donald_Duck -a -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.628 MB perf.data (114 samples) ]


# perf script | head
           perf 24546 [000] 30211.120492: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/libexec/perf-core/sleep"
           perf 24546 [000] 30211.120514: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/local/sbin/sleep"
           perf 24546 [000] 30211.120521: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/local/bin/sleep"
           perf 24546 [000] 30211.120526: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/sbin/sleep"
           perf 24546 [000] 30211.120531: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/bin/sleep"
          sleep 24546 [001] 30211.121178: probe:Donald_Duck: (ffffffff811eeca4) pathname="/etc/ld.so.preload"
          sleep 24546 [001] 30211.121199: probe:Donald_Duck: (ffffffff811eeca4) pathname="/etc/ld.so.cache"
          sleep 24546 [001] 30211.121233: probe:Donald_Duck: (ffffffff811eeca4) pathname="/lib64/libc.so.6"
          sleep 24546 [001] 30211.121624: probe:Donald_Duck: (ffffffff811eeca4) pathname="/usr/lib/locale/locale-archive"
```

3.3.10 - Tracing calls, returns & mid function lines of code.

In order to trace events in perf you need to add a probe. Adding a probe doesn't actually start tracing, it simply adds the tracepoint to a table of traceable events. This is done via '**`perf probe --add xxxxx`**". In actuality you don't need **`--add`**, it will add the probe without the **`--add`** option.

How do you probe a call or return???

`perf probe --add __do_page_fault`	Probe a call entry
`perf probe --add '__do_page_fault%return ret=$retval'`	Probe a return

Can you probe anywhere? Not totally.... Let's add a probe a source line 173 of **`__do_page_fault()`**.

```
perf probe --add '__do_page_fault:173'
```

So where can you not probe???? You cannot probe 'inline' statements right now. Seems there's a little problem with that. Also, you cannot probe lines that don't actually produce assembler code (obvious as that seems but has to be re-stated here). One other point, you can't use line numbers without the debuginfo (another obvious point that must be restated). You can can probe the calls and returns without such.

```
# perf probe -L __do_page_fault:0-5
<__do_page_fault@/usr/src/debug/kernel-2.6.32-621.el6/linux-2.6.32-621.el6.x86_64/arch/x86/mm/fault.c:0>
      0  static inline void __do_page_fault(struct pt_regs *regs, unsigned long address, unsigned long error_
      1  {
                 struct vm_area_struct *vma;
                 struct task_struct *tsk;                  ← Line 3 is not noted so cannot be probed
                 struct mm_struct *mm;
                 int fault;


# perf probe --add '__do_page_fault:3'
Probe point '__do_page_fault:3' not found.
  Error: Failed to add events.
```

3.3.11 - Probing/source listing/variables of a module/mod

You can probe modules. Of course you must have the **`debuginfo`** loaded.

```
# perf probe -m drm drm_av_sync_delay
Added new event:
  probe:drm_av_sync_delay (on drm_av_sync_delay in drm)

You can now use it in all perf tools, such as:

        perf record -e probe:drm_av_sync_delay -aR sleep 1
```

You can list the source in the same way:

```
# perf probe -m drm -L drm_av_sync_delay
<drm_av_sync_delay@/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-
327.13.1.el7.x86_64/drivers/gpu/drm/drm_edid.c:0>
      0  int drm_av_sync_delay(struct drm_connector *connector,
                              struct drm_display_mode *mode)
      2  {
      3         int i = !!(mode->flags & DRM_MODE_FLAG_INTERLACE);
                int a, v;

      6         if (!connector->latency_present[0])
      7                 return 0;
      8         if (!connector->latency_present[1])
                        i = 0;

     11         a = connector->audio_latency[i];
     12         v = connector->video_latency[i];

                /*
                 * HDMI/DP sink doesn't support audio or video?
                 */
     17         if (a == 255 || v == 255)
                        return 0;

                /*
                 * Convert raw EDID values to millisecond.
                 * Treat unknown latency as 0ms.
                 */
     24         if (a)
     25                 a = min(2 * (a - 1), 500);
     26         if (v)
     27                 v = min(2 * (v - 1), 500);

     29         return max(v - a, 0);
```

I had tried a number of ways to probe the **dm_mod** but could not get it to work (eventually I did...).

```
# perf probe -m dm_mod dm_region_hash_destroy
Probe point 'dm_region_hash_destroy' not found.
  Error: Failed to add events.
```

This took some work to figure out (perhaps that was my issue...). I finally discovered how to find the **.ko** and was able to get the listing and **--add** the probe (follows the listing below). First... how did I find the info? Looked in the source code for the function I wanted. I had used **cscope** but you can use whatever tool works for you. This gave me the source file name **drivers/md/dm-region-hash.c**.

```
void dm_region_hash_destroy(struct dm_region_hash *rh)
{
        unsigned h;
        struct dm_region *reg, *nreg;

        BUG_ON(!list_empty(&rh->quiesced_regions));
        for (h = 0; h < rh->nr_buckets; h++) {
                list_for_each_entry_safe(reg, nreg, rh->buckets + h,
                                         hash_list) {
                        BUG_ON(atomic_read(&reg->pending));
                        mempool_free(reg, rh->region_pool);
                }
"drivers/md/dm-region-hash.c" 723L, 18485C
```

So now I know the source path. It's in **drivers/md**. Now I looked in the **debug** dir where we store all the debuginfo files and found the debug file for the function.

```
# ll /usr/lib/debug/lib/modules/3.10.0-327.13.1.el7.x86_64/kernel/drivers/md/
total 12704
-r--r--r--. 1 root root  209296 Feb 29 11:44 dm-bio-prison.ko.debug
-r--r--r--. 1 root root  342240 Feb 29 11:44 dm-bufio.ko.debug
-r--r--r--. 1 root root  211336 Feb 29 11:44 dm-cache-cleaner.ko.debug
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
-r--r--r--. 1 root root  405208 Feb 29 11:44 dm-log-userspace.ko.debug
-r--r--r--. 1 root root  311800 Feb 29 11:44 dm-mirror.ko.debug
-r--r--r--. 1 root root 2089152 Feb 29 11:44 dm-mod.ko.debug
-r--r--r--. 1 root root  425264 Feb 29 11:44 dm-multipath.ko.debug
-r--r--r--. 1 root root  171456 Feb 29 11:44 dm-queue-length.ko.debug
-r--r--r--. 1 root root  304104 Feb 29 11:44 dm-raid.ko.debug
-r--r--r--. 1 root root  263288 Feb 29 11:44 dm-region-hash.ko.debug
-r--r--r--. 1 root root  167448 Feb 29 11:44 dm-round-robin.ko.debug
-r--r--r--. 1 root root  175128 Feb 29 11:44 dm-service-time.ko.debug
-r--r--r--. 1 root root  703560 Feb 29 11:44 dm-snapshot.ko.debug
-r--r--r--. 1 root root  197464 Feb 29 11:44 dm-switch.ko.debug
```

```
- - - - - - - - - - - - - - - 8< - - - - - - - - - - - - - - -
```

Now I know the debug file name, it's easy to link it into the **`perf`** command to list the source, find the variables, and add probes:

```
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug -L dm_region_hash_destroy
<dm_region_hash_destroy@/usr/src/debug/kernel-3.10.0-327.13.1.el7/linux-3.10.0-327.13.1.el7.x86_64/drivers/md/dm-region-hash.c:0>
      0  void dm_region_hash_destroy(struct dm_region_hash *rh)
      1  {
                unsigned h;
                struct dm_region *reg, *nreg;

      5         BUG_ON(!list_empty(&rh->quiesced_regions));
      6         for (h = 0; h < rh->nr_buckets; h++) {
      7                 list_for_each_entry_safe(reg, nreg, rh->buckets + h,
                                                 hash_list) {
      9                         BUG_ON(atomic_read(&reg->pending));
     10                         mempool_free(reg, rh->region_pool);
                        }
                }

     14         if (rh->log)
     15                 dm_dirty_log_destroy(rh->log);
- - - - - - - - - - - - - - - 8< - - - - - - - - - - - - - - - -
```

```
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug -V dm_region_hash_destroy
Available variables at dm_region_hash_destroy
        @<dm_region_hash_destroy+0>
                struct dm_region_hash*  rh
```

```
# perf probe -m /usr/lib/debug/lib/modules/3.10.0-327.13.1.el7.x86_64/kernel/drivers/md/dm-region-hash.ko.debug --add dm_region_hash_destroy
Added new event:
  probe:dm_region_hash_destroy (on dm_region_hash_destroy in dm-region-hash)

You can now use it in all perf tools, such as:

      perf record -e probe:dm_region_hash_destroy -aR sleep 1
```

One nice addition, you can list the function calls within a module also and therefore also see the local variables they use:

```
# perf probe -m drm -F
agp_remap
```

```
alloc_anon_inode
anon_set_page_dirty
cea_mode_alternate_clock
check_src_coords
cleanup_module
compat_drm_addbufs
compat_drm_addmap
compat_drm_agp_alloc
compat_drm_agp_bind
compat_drm_agp_enable
compat_drm_agp_free
compat_drm_agp_info
compat_drm_agp_unbind
compat_drm_dma
- - - - - - 8< - - - - - -
```

```
# perf probe -m drm -V compat_drm_agp_info
Available variables at compat_drm_agp_info
        @<compat_drm_agp_info+0>
                drm_agp_info32_t        i32
                long unsigned int       arg
                struct file*    file
                unsigned int    cmd
```

Here's another example. This one also uses a filter to test for a specific device. Note the structure linkage. Why the '.' period in that linkage? Because there's an embedded structure and we want to access a specific element within that.

```
# perf probe  -m xfs -V  xfs_dir_removename
Available variables at xfs_dir_removename
      @<xfs_dir_removename+0>
              int     v
              struct xfs_name*        name
              xfs_bmap_free_t*        flist
              xfs_extlen_t    total
              xfs_fsblock_t*  first
              xfs_ino_t       ino
              xfs_inode_t*    dp              ← We want this structure pointer
              xfs_trans_t*    tp
```

What is **xfs_inode_t**? Check the source code. It's a renamed **xfs_inode** structure.

```
typedef struct xfs_inode {
      /* Inode linking and identification information. */
        ...
      struct inode            i_vnode;        /* embedded VFS inode */
} xfs_inode_t;
```

```
crash> struct xfs_inode ffff8801392ad700
struct xfs_inode {
  i_mount = 0x1eb80,
        ...    xfs_inode->i_vnode.i_sb->s_dev
        i_vnode = {
        ...
    i_opflags = 0x0,
    i_op = 0xffff8801392ad968,
    i_sb = 0xffff880034c88600,            ← Now we want this subelement
```

```
crash> super_block -o
struct super_block {
    [0x0] struct list_head s_list;
   [0x10] dev_t s_dev;                    ← And this is the actual variable we want
```

```
# perf probe -m xfs  --add 'xfs_dir_removename dp->i_vnode.i_sb->s_dev'
Added new event:
  probe:xfs_dir_removename (on xfs_dir_removename in xfs with s_dev=dp->i_vnode.i_sb->s_dev)

You can now use it in all perf tools, such as:

      perf record -e probe:xfs_dir_removename -aR sleep 1
```

```
# perf record -a -g -e 'probe:xfs_dir*' --filter 's_dev==0xfd00003' -- sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.381 MB perf.data ]
```

What we were doing here was to probe a module's function which is of value but doesn't directly have the data we want to see and so we determined the variable we wanted deeper in the underlying structure and then used filtering to select only events that matched our device (**`0xfd00003`** = 253,3).

3.3.12 - Tracing multiple probes and repeating probes

Can you trace multiple probes at the same time? Yes. With **perf record**, supply each probe with a **-e**. OR.... if you want to record all the probes, use **-e probe:***

```
# perf probe -l
/sys/kernel/debug/tracing/uprobe_events file does not exist - please
rebuild kernel with CONFIG_UPROBE_EVENTS.
  probe:__do_page_fault (on __do_page_fault@arch/x86/mm/fault.c)
  probe:tcp_sendmsg    (on tcp_sendmsg@net/ipv4/tcp.c with size
skc_state)

# perf record -e probe:tcp_sendmsg -e probe:__do_page_fault -ag sleep 5
[ perf record: Woken up 4 times to write data ]
[ perf record: Captured and wrote 1.972 MB perf.data (~86177 samples) ]
```

-OR-

```
# dd if=/dev/sda of=/dev/null bs=4k count=100000

# perf report

Available samples
7 probe:tcp_sendmsg                                                    ◆
9K probe:__do_page_fault                                               ▒
```

You can add multiple probes in **perf stat** also. In the following example, I'm not just sleeping 10 seconds, I'm telling perf with **--repeat** I want to execute this stat 10 times. (This results in displaying the mean and standard deviation NOT the total for all repeated runs)

```
# perf stat --repeat 10 -a -e kmem:mm_page_pcpu_drain -e kmem:mm_page_alloc sleep 10

 Performance counter stats for 'system wide' (10 runs):

           11,656      kmem:mm_page_pcpu_drain                 ( +-   9.87% ) [100.00%]
           13,109      kmem:mm_page_alloc                      ( +-   6.66% )

     10.001828804 seconds time elapsed                    ( +-  0.00% )
```

You can also use the wildcard as follows:

```
# perf stat --repeat 10 -a -e kmem:mm_page* sleep 10

 Performance counter stats for 'system wide' (10 runs):

             2,571      kmem:mm_page_free_direct             ( +- 19.10% ) [100.00%]
            12,513      kmem:mm_pagevec_free                 ( +-  7.85% ) [100.00%]
            13,854      kmem:mm_page_alloc                   ( +-  6.45% ) [100.00%]
            11,490      kmem:mm_page_alloc_zone_locked       ( +-  7.02% ) [100.00%]
            11,445      kmem:mm_page_pcpu_drain              ( +-  8.15% ) [100.00%]
                 0      kmem:mm_page_alloc_extfrag           [100.00%]
                 0      kmem:mm_pagereclaim_pgout            [100.00%]
                 0      kmem:mm_pagereclaim_free             [100.00%]
                 0      kmem:mm_pagereclaim_shrinkzone       [100.00%]
                 0      kmem:mm_pagereclaim_shrinkactive     [100.00%]
                 0      kmem:mm_pagereclaim_shrinkinactive

      10.001688534 seconds time elapsed                ( +-  0.00% )
```

When using **--repeat** it is displaying the mean value/count for the repeated runs and the standard deviation from the mean.

Another nice feature of **stat** is that you can supply commands to execute prior to and after the monitored command. I acknowledge this isn't the best example but you'll get the idea. I've added **-d** also to get more CPU detail:

```
# perf stat -d --pre free --post date -- df
             total       used       free     shared    buffers     cached
Mem:      15954472    7777552    8176920     259224    1024016    3329748
-/+ buffers/cache:    3423788   12530684
Swap:     20529148          0   20529148
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31960008  36487808  47% /
tmpfs                  7977236       552   7976684   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315640004  50659324  87% /home
Thu Mar 31 19:11:09 MST 2016

 Performance counter stats for 'df':

          0.360454      task-clock (msec)         #    0.704 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               233      page-faults               #    0.646 M/sec
         1,320,405      cycles                    #    3.663 GHz
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
           949,382      instructions              #    0.72  insns per cycle
```

```
          194,152      branches                  #  538.632 M/sec
            9,559      branch-misses             #    4.92% of all branches
          239,178      L1-dcache-loads           #  663.547 M/sec
    <not counted>      L1-dcache-load-misses
    <not counted>      LLC-loads
  <not supported>      LLC-load-misses:HG

      0.000512108 seconds time elapsed
```

A more meaningful example? What if you wanted to **`stat`** a **`dd`** of a file but you want to force the file to disk first so you need to run a **`--pre sync`**. Or how about running a script in **`--pre`** to setup a condition, **`stat`** your system (or a command) and then **`--post`** to reset things back again. I'm sure in time, we'll think of some really wizard ideas 😃

3.3.13 - “Event Monitoring” a running process

We've been looking at probing the kernel and gathering all kinds of useful data. You can also “monitor” a running process.

You can use **-p *PID*** with **perf stat** to monitor all kinds of events for just the noted PID. Here are a few examples. First monitoring all calls that ***PID*** makes. Secondly, monitoring all scheduler activity for just this ***PID*** without and with a sleep timer.

```
# perf stat -e 'syscalls:sys_enter_*' -p PID

# perf stat -e 'sched:*' -p PID
# perf stat -e 'sched:*' -p PID sleep 10
```

You can use **-p *PID*** with **perf record** also.

```
# perf record -p PID
# perf record -p PID sleep 10
# perf record -p PID -g -- sleep 10
```

And now another slick feature, you can **stat** and **record** more than just a single PID or Thread (TID). Just supply a comma separated list:

```
# perf stat -p PID,PID,PID,PID
# perf stat -t TID,TID,TID,TID

# perf record -p PID,PID,PID,PID
# perf record -p PID,PID,PID sleep 10
# perf record -p PID,PID,PID -g -- sleep 10
```

3.3.14 - I want to see individual CPU stat/record

There are times this will definitely be handy and yes it is possible. It ONLY works when you are in "system wide" display mode. Use **-A** (with **-a**) in **perf stat**. I've spaced the CPU listings below to make it a little more viewable. **perf record** follows this explanation.

```
# perf stat -aA -- df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32260528  36187288  48% /
tmpfs                  7977236       580   7976656   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315651472  50647856  87% /home

 Performance counter stats for 'system wide':

CPU0             3.108973      task-clock (msec)         #    1.258 CPUs utilized            (99.97%)
CPU1             3.109934      task-clock (msec)         #    1.258 CPUs utilized            (99.98%)
CPU2             3.107986      task-clock (msec)         #    1.257 CPUs utilized            (99.97%)
CPU3             3.106259      task-clock (msec)         #    1.257 CPUs utilized            (99.98%)
CPU4             3.089281      task-clock (msec)         #    1.250 CPUs utilized            (99.98%)
CPU5             3.086380      task-clock (msec)         #    1.249 CPUs utilized            (99.98%)
CPU6             3.068205      task-clock (msec)         #    1.241 CPUs utilized            (99.98%)
CPU7             3.066400      task-clock (msec)         #    1.240 CPUs utilized            (99.98%)

CPU0                    0      context-switches          #    0.000 K/sec                    (99.98%)
CPU1                    2      context-switches                                              (99.99%)
CPU2                    5      context-switches                                              (99.99%)
CPU3                    0      context-switches                                              (99.99%)
CPU4                    0      context-switches                                              (99.99%)
CPU5                    2      context-switches                                              (99.99%)
CPU6                    0      context-switches                                              (99.99%)
CPU7                    0      context-switches                                              (99.99%)

CPU0                    0      cpu-migrations            #    0.000 K/sec                    (99.99%)
CPU1                    1      cpu-migrations                                                (99.99%)
CPU2                    1      cpu-migrations                                                (99.99%)
CPU3                    0      cpu-migrations                                                (99.99%)
CPU4                    0      cpu-migrations                                                (99.99%)
CPU5                    0      cpu-migrations                                                (100.00%)
CPU6                    0      cpu-migrations                                                (99.99%)
CPU7                    0      cpu-migrations                                                (99.99%)

CPU0                    0      page-faults               #    0.000 K/sec
CPU1                   11      page-faults
CPU2                    9      page-faults
CPU3                    0      page-faults
CPU4                    0      page-faults
CPU5                  233      page-faults
CPU6                    0      page-faults
CPU7                    0      page-faults
```

```
CPU0          10,426,902      cycles                    #    3.371 GHz                    (99.92%)
CPU1           1,834,762      cycles                     (99.92%)
CPU2             516,333      cycles                     (99.94%)
CPU3             138,760      cycles                     (99.94%)
CPU4             156,848      cycles                     (99.94%)
CPU5           8,604,365      cycles                     (99.94%)
CPU6             312,569      cycles                     (99.93%)
CPU7             145,643      cycles                     (99.94%)

CPU0     <not supported>      stalled-cycles-frontend
CPU1     <not supported>      stalled-cycles-frontend
CPU2     <not supported>      stalled-cycles-frontend
CPU3     <not supported>      stalled-cycles-frontend
CPU4     <not supported>      stalled-cycles-frontend
CPU5     <not supported>      stalled-cycles-frontend
CPU6     <not supported>      stalled-cycles-frontend
CPU7     <not supported>      stalled-cycles-frontend

CPU0     <not supported>      stalled-cycles-backend
CPU1     <not supported>      stalled-cycles-backend
CPU2     <not supported>      stalled-cycles-backend
CPU3     <not supported>      stalled-cycles-backend
CPU4     <not supported>      stalled-cycles-backend
CPU5     <not supported>      stalled-cycles-backend
CPU6     <not supported>      stalled-cycles-backend
CPU7     <not supported>      stalled-cycles-backend

CPU0          22,055,912      instructions              #    7.97  insns per cycle         (99.94%)
CPU1             456,481      instructions               (99.94%)
CPU2             100,174      instructions               (99.95%)
CPU3              34,170      instructions               (99.95%)
CPU4              31,065      instructions               (99.96%)
CPU5           2,912,502      instructions               (99.95%)
CPU6              26,916      instructions               (99.95%)
CPU7              35,420      instructions               (99.96%)

CPU0           3,880,111      branches                  # 1254.511 M/sec                   (99.97%)
CPU1             103,712      branches                                                     (99.96%)
CPU2              17,703      branches                                                     (99.97%)
CPU3               6,553      branches                                                     (99.97%)
CPU4               6,159      branches                                                     (99.98%)
CPU5             991,518      branches                                                     (99.97%)
CPU6               5,218      branches                                                     (99.97%)
CPU7               6,776      branches                                                     (99.97%)

CPU0                 167      branch-misses             #    0.03% of all branches
CPU1                 850      branch-misses
CPU2               1,355      branch-misses
CPU3                 273      branch-misses
CPU4                 126      branch-misses
CPU5              15,782      branch-misses
CPU6                 133      branch-misses
CPU7                 404      branch-misses

       0.002472041 seconds time elapsed
```

What if you have 8 CPU's but you only want to see the output for a selected set of 4 CPU's? Yes it can do that:

```
# perf stat -aA -C0-3 -- df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32264216  36183600  48% /
tmpfs                  7977236       580   7976656   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315651552  50647776  87% /home

 Performance counter stats for 'system wide':

CPU0              2.746256      task-clock (msec)         #    1.299 CPUs utilized            (99.89%)
CPU1              2.746958      task-clock (msec)         #    1.299 CPUs utilized            (99.91%)
CPU2              2.742429      task-clock (msec)         #    1.297 CPUs utilized            (99.89%)
CPU3              2.712198      task-clock (msec)         #    1.283 CPUs utilized            (99.94%)

CPU0                     2      context-switches          #    0.731 K/sec                    (99.95%)
CPU1                     3      context-switches                                              (99.94%)
CPU2                     0      context-switches                                              (99.94%)
CPU3                     2      context-switches                                              (99.96%)

CPU0                     1      cpu-migrations            #    0.365 K/sec                    (99.98%)
CPU1                     1      cpu-migrations                                                (99.97%)
CPU2                     0      cpu-migrations                                                (99.98%)
CPU3                     0      cpu-migrations                                                (99.98%)

CPU0                     9      page-faults               #    0.003 M/sec
CPU1                     9      page-faults
CPU2                     0      page-faults
CPU3                   233      page-faults

CPU0               593,569      cycles                    #    0.217 GHz                      (99.70%)
CPU1               216,167      cycles                     (99.65%)
CPU2               183,073      cycles                     (99.69%)
CPU3             1,616,084      cycles                     (99.71%)

CPU0       <not supported>      stalled-cycles-frontend
CPU1       <not supported>      stalled-cycles-frontend
CPU2       <not supported>      stalled-cycles-frontend
CPU3       <not supported>      stalled-cycles-frontend

CPU0       <not supported>      stalled-cycles-backend
CPU1       <not supported>      stalled-cycles-backend
CPU2       <not supported>      stalled-cycles-backend
CPU3       <not supported>      stalled-cycles-backend

CPU0               205,950      instructions              #    0.32  insns per cycle          (99.77%)
CPU1                87,605      instructions               (99.73%)
CPU2                41,497      instructions               (99.77%)
CPU3             1,325,770      instructions               (99.78%)

CPU0                40,801      branches                  #   14.907 M/sec                    (99.88%)
```

```
CPU1                15,864      branches                                                  (99.85%)
CPU2                 8,076      branches                                                  (99.87%)
CPU3               282,001      branches                                                  (99.88%)

CPU0                 1,023      branch-misses             #    1.18% of all branches
CPU1                   628      branch-misses
CPU2                   358      branch-misses
CPU3                11,467      branch-misses

       0.002114598 seconds time elapsed
```

Don't forget, you can use selective CPU's:

```
# perf stat -aA -C0,2,4,6 -- df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32264224  36183592  48% /
tmpfs                  7977236       580   7976656   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315651540  50647788  87% /home

 Performance counter stats for 'system wide':

CPU0              4.881311      task-clock (msec)         #    1.303 CPUs utilized            (99.94%)
CPU2              4.867413      task-clock (msec)         #    1.299 CPUs utilized            (99.88%)
CPU4              4.861462      task-clock (msec)         #    1.298 CPUs utilized            (99.89%)
CPU6              4.870357      task-clock (msec)         #    1.300 CPUs utilized            (99.92%)

CPU0                     6      context-switches          #    0.001 M/sec                    (99.97%)
CPU2                     0      context-switches                                              (99.92%)
CPU4                     4      context-switches                                              (99.92%)
CPU6                     0      context-switches                                              (99.95%)

CPU0                     0      cpu-migrations            #    0.000 K/sec                    (99.98%)
CPU2                     0      cpu-migrations                                                (99.96%)
CPU4                     0      cpu-migrations                                                (99.96%)
CPU6                     0      cpu-migrations                                                (99.98%)

CPU0                    10      page-faults               #    0.002 M/sec
CPU2                     0      page-faults
CPU4                     0      page-faults
CPU6                     0      page-faults

CPU0             1,568,986      cycles                    #    0.322 GHz                      (99.84%)
CPU2               347,142      cycles                     (99.74%)
CPU4               548,914      cycles                     (99.76%)
CPU6               445,221      cycles                     (99.75%)

CPU0       <not supported>      stalled-cycles-frontend
CPU2       <not supported>      stalled-cycles-frontend
CPU4       <not supported>      stalled-cycles-frontend
CPU6       <not supported>      stalled-cycles-frontend

CPU0       <not supported>      stalled-cycles-backend
```

```
CPU2       <not supported>      stalled-cycles-backend
CPU4       <not supported>      stalled-cycles-backend
CPU6       <not supported>      stalled-cycles-backend

CPU0              458,328      instructions              #    0.63  insns per cycle         (99.88%)
CPU2               26,953      instructions               (99.81%)
CPU4               56,659      instructions               (99.83%)
CPU6               31,868      instructions               (99.81%)

CPU0              107,631      branches                  #   22.100 M/sec                   (99.93%)
CPU2                5,254      branches                                                     (99.90%)
CPU4               10,952      branches                                                     (99.92%)
CPU6                6,278      branches                                                     (99.89%)

CPU0                4,226      branch-misses             #   12.99% of all branches
CPU2                  330      branch-misses
CPU4                1,462      branch-misses
CPU6                  916      branch-misses

       0.003745655 seconds time elapsed
```

It gets better. You can actually produce stats based on the numebr of cores in a socket and you can display the stats based on the socket itself. This is from a system which has 1 socket and 4 cores and hyperthreading enabled (8 logicaal CPU's):

```
# perf stat -a --per-core df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32261376  36186440  48% /
tmpfs                  7977236       624   7976612   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315649532  50649796  87% /home

 Performance counter stats for 'system wide':

S0-C0           2          22.631450      task-clock (msec)         #    2.369 CPUs utilized           (99.98%)
S0-C0           2                 19      context-switches          #    0.002 M/sec                   (99.99%)
S0-C0           2                  2      cpu-migrations            #    0.177 K/sec                   (99.99%)
S0-C0           2                 27      page-faults               #    0.002 M/sec
S0-C0           2          5,729,924      cycles                    #    0.507 GHz                     (99.92%)
S0-C0           2    <not supported>      stalled-cycles-frontend
S0-C0           2    <not supported>      stalled-cycles-backend
S0-C0           2          3,404,102      instructions              #    1.78  insns per cycle         (99.94%)
S0-C0           2            681,782      branches                  #   60.378 M/sec                   (99.97%)
S0-C0           2             25,871      branch-misses             #   10.64% of all branches

S0-C1           2          22.573727      task-clock (msec)         #    2.363 CPUs utilized           (99.98%)
S0-C1           2                  8      context-switches                                             (99.99%)
S0-C1           2                  0      cpu-migrations                                               (99.99%)
S0-C1           2                233      page-faults
S0-C1           2          7,903,179      cycles                     (99.91%)
S0-C1           2    <not supported>      stalled-cycles-frontend
S0-C1           2    <not supported>      stalled-cycles-backend
S0-C1           2          3,464,821      instructions               (99.93%)
S0-C1           2          1,219,022      branches                                                     (99.96%)
S0-C1           2             19,665      branch-misses

S0-C2           2          22.573378      task-clock (msec)         #    2.363 CPUs utilized           (99.97%)
```

```
S0-C2           2                4      context-switches                                  (99.98%)
S0-C2           2                0      cpu-migrations                                    (99.99%)
S0-C2           2                0      page-faults
S0-C2           2          875,120      cycles                    (99.92%)
S0-C2           2    <not supported>    stalled-cycles-frontend
S0-C2           2    <not supported>    stalled-cycles-backend
S0-C2           2          101,079      instructions              (99.94%)
S0-C2           2           19,125      branches                                          (99.97%)
S0-C2           2            2,039      branch-misses

S0-C3           2        22.556041      task-clock (msec)        #    2.361 CPUs utilized            (99.96%)
S0-C3           2                2      context-switches                                  (99.97%)
S0-C3           2                0      cpu-migrations                                    (99.98%)
S0-C3           2                0      page-faults
S0-C3           2          755,825      cycles                    (99.89%)
S0-C3           2    <not supported>    stalled-cycles-frontend
S0-C3           2    <not supported>    stalled-cycles-backend
S0-C3           2          123,700      instructions              (99.92%)
S0-C3           2           26,049      branches                                          (99.96%)
S0-C3           2            1,999      branch-misses

       0.009551751 seconds time elapsed
```

```
# perf stat -a --per-socket df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32261432  36186384  48% /
tmpfs                  7977236       624   7976612   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315649572  50649756  87% /home

 Performance counter stats for 'system wide':

S0        8         38.268656      task-clock (msec)         #   12.588 CPUs utilized            (99.91%)
S0        8                13      context-switches          #    0.003 M/sec                    (99.94%)
S0        8                 4      cpu-migrations            #    0.836 K/sec                    (99.97%)
S0        8               251      page-faults               #    0.052 M/sec
S0        8         5,969,095      cycles                    #    1.248 GHz                      (99.79%)
S0        8    <not supported>     stalled-cycles-frontend
S0        8    <not supported>     stalled-cycles-backend
S0        8         2,378,951      instructions              #    3.19  insns per cycle          (99.83%)
S0        8           549,316      branches                  #  114.834 M/sec                    (99.91%)
S0        8            18,101      branch-misses             #   26.36% of all branches

       0.003040097 seconds time elapsed
```

You can also enable specific CPU recording with **perf record**. I've noted the CPU column in the **perf script** output.

```
# perf record -e cycles -C4 -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.651 MB perf.data (~28442 samples) ]

# perf script
                        CPU
                        vvv
            init     1 [004] 67923.507851: cycles:  ffffffff8104315a native_write_msr_safe ([kernel.kallsyms])
         firefox  4468 [004] 67923.597971: cycles:  ffffffff8105fbba update_curr ([kernel.kallsyms])
         firefox  4468 [004] 67923.597979: cycles:  ffffffff8105e64d enqueue_task ([kernel.kallsyms])
         firefox  4468 [004] 67923.597988: cycles:  ffffffff8108a3cd run_timer_softirq ([kernel.kallsyms])
```

```
     firefox  4468 [004] 67923.598019: cycles:      7f251f18ee32 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.598144: cycles:      7f251f1958e3 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.598378: cycles:      7f251f18ef23 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.598639: cycles:      7f251f121681 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.598903: cycles:      7f251f1958e3 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.599168: cycles:      7f251f1958a4 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.599427: cycles:      7f251f2b3e57 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.599685: cycles:      7f251da87051 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.599941: cycles:      7f251e5551f8 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4468 [004] 67923.600199: cycles:      7f251ddb796c [unknown] (/usr/lib64/firefox/libxul.so)
        init     1 [004] 67923.659651: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
     firefox  4495 [004] 67923.881579: cycles:      7f251f465e70 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4495 [004] 67923.881634: cycles:      7f251f430170 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4495 [004] 67923.881688: cycles:      7f251f459f35 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4495 [004] 67923.881767: cycles:      7f251f401c5e [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4495 [004] 67923.881881: cycles:      7f251f3f164e [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4495 [004] 67923.882026: cycles:      7f251f137e5f [unknown] (/usr/lib64/firefox/libxul.so)
        init     1 [004] 67924.062004: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
        init     1 [004] 67925.301992: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
        init     1 [004] 67926.478591: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
        init     1 [004] 67927.066615: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
        init     1 [004] 67928.068437: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
        init     1 [004] 67929.069086: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
     firefox  4501 [004] 67929.121957: cycles:      7f251f1d082c [unknown] (/usr/lib64/firefox/libxul.so)
        init     1 [004] 67929.622361: cycles:  ffffffff812f1661 intel_idle ([kernel.kallsyms])
     firefox  4501 [004] 67929.951766: cycles:      7f251f41f238 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4501 [004] 67929.952261: cycles:      7f251f40d750 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4501 [004] 67929.952761: cycles:      7f251f3f21c4 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4501 [004] 67929.953199: cycles:      7f251f3f3323 [unknown] (/usr/lib64/firefox/libxul.so)
     firefox  4501 [004] 67929.953614: cycles:      7f251f2ffadb [unknown] (/usr/lib64/firefox/libxul.so)
        init     1 [004] 67930.070272: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
        init     1 [004] 67930.261750: cycles:  ffffffff810ece31 touch_softlockup_watchdog ([kernel.kallsyms])
    events/4    39 [004] 67930.262111: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
    events/4    39 [004] 67930.262408: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
    events/4    39 [004] 67930.262693: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
    events/4    39 [004] 67930.262960: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
    events/4    39 [004] 67930.263019: cycles:  ffffffff812a326a ioread32 ([kernel.kallsyms])
    - - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

3.3.15 - Recording/profiling at a specific frequency or count

Frequency

There is an option in record that allows you set the frequency at which you want to profile/record data. Now searching on the web I've seen different explanations of what this frequency means. One person stated this as the number of times per second that we sample. On the other hand Brendan Gregg notes it as the Hz Frequency Sampling Rate. From what I've now determined, it is indeed in RHEL6 and RHEL7, **the number of samples per second.**

Here's a simple example of what it can be used for. I want to sample the whole system and record the bt/stacks. First, let's see what the system gives us without the **-F** option.

```
# perf record -ag -- sleep 10
[ perf record: Woken up 5 times to write data ]
[ perf record: Captured and wrote 2.314 MB perf.data (19238 samples) ]
```

And now let's adjust the frequency. To give you an idea, I started at 1,000,000 and decreased by a power of 10 each time, finally reducing to 1. Note the number of samples differences!!!!!

```
# perf record -F 1000000 -ag -- sleep 10
Maximum frequency rate (100000) reached.
Please use -F freq option with lower value or consider
tweaking /proc/sys/kernel/perf_event_max_sample_rate.
```

```
# perf record -F 100000 -ag -- sleep 10
[ perf record: Woken up 141 times to write data ]
[ perf record: Captured and wrote 37.587 MB perf.data (401240 samples) ]
```

```
# perf record -F 10000 -ag -- sleep 10
[ perf record: Woken up 12 times to write data ]
[ perf record: Captured and wrote 4.284 MB perf.data (40935 samples) ]
```

```
# perf record -F 4000 -ag -- sleep 10
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 1.858 MB perf.data (12668 samples) ]
```

```
# perf record -F 1000 -ag -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.102 MB perf.data (4311 samples) ]
```

```
# perf record -F 100 -ag -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.757 MB perf.data (570 samples) ]
```

```
# perf record -F 10 -ag -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.711 MB perf.data (75 samples) ]
```

```
# perf record -F 1 -ag -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.710 MB perf.data (19 samples) ]
```

This started me wondering. Then what is the sample rate without specifying the **-F**. As it turns out **the default is [4000]**. That's why I highlighted the 4000 entry.

> A note here, a frequency of 1 did not seem to work early RHEL6 but RHEL6.8 does seem to work. As with ANY of the functions and features of perf, be cognizant that it is continually being improved so any issues or limitations in this document may not exist in your release.

Therefore while I can see there is some uses where sampling counters at a higher frequency may help, you certainly have to be cognizant that the size of the **perf.data** file may grow considerably.

```
# perf record -F 4000 -ag -- sleep 10
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 2.162 MB perf.data (16757 samples) ]
```

```
# perf record -ag -- sleep 10
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 1.858 MB perf.data (13034 samples) ]
```

Why would you want to change the frequency? In some cases, the amount of captured data may be so extensive as to impact the performance of **perf**. Therefore there are going to occassions where it is more important to have a lower frequency and smaller sample.

<u>perf record: Woken up 1 times to write data</u>

I'm sure you're wondering what this means? Look at the table below

```
# perf record -F 100000 -ag -- sleep 10         [ perf record: Woken up 141 times to write data ]
# perf record -F 10000 -ag -- sleep 10          [ perf record: Woken up 12 times to write data ]
# perf record -F 4000 -ag -- sleep 10           [ perf record: Woken up 3 times to write data ]
# perf record -F 1000 -ag -- sleep 10           [ perf record: Woken up 1 times to write data ]
# perf record -F 100 -ag -- sleep 10            [ perf record: Woken up 1 times to write data ]
# perf record -F 10 -ag -- sleep 10             [ perf record: Woken up 1 times to write data ]
# perf record -F 1 -ag -- sleep 10              [ perf record: Woken up 1 times to write data ]
```

So the higher the frequency the higher the count of wakeups needed to keep **perf** capturing at the required sample rate.

Count

You can also set a counter which tells perf to collect a sample every **n** occurrences. Like frequency, you have to be a little careful and knowledgeable to use this effectively. Take this simple example. Sampling every 200,000 occurrences creates a file of 41,000 samples. Changed to every 2,000 occurrences, the number of samples rise considerably to nearly 149,000.

```
# perf record -e instructions:u -a -c 200000 -- sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.942 MB perf.data (~41,177 samples) ]
```

```
# perf record -e instructions:u -a -c 2000 -- sleep 5
[ perf record: Woken up 10 times to write data ]
[ perf record: Captured and wrote 3.404 MB perf.data (~148,724 samples) ]
```

```
# perf record -e instructions:u -a -- sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.717 MB perf.data (~31,347 samples) ]
```

3.3.16 - Grouping multiple events into one combined (group) report

One of things you might have realized is that when you record multiple events perf, it reports them all individually. You might be wondering, how do you combine all the events into one report. The answer is **--group**. Well... almost... that's used in the perf report but you have to combine the events in per record using a different technique. Let's take an example of how we have been recording events:

```
# perf record -e cycles,cache-misses -a -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.109 MB perf.data (~48473 samples) ]
```

Now when you **`report`**, it shows you 2 separate events and you can select which you want to investigate:

```
Available samples
12K cycles                                                                                          ◆
7K cache-misses                                                                                     ▒
                                                                                                    ▒
                                                                                                    ▒
                                                                                                    ▒
                                                                                                    ▒
                                                                                                    ▒
                                                                                                    ▒
                                                                                                    ▒
```

If you use evlist, it too reports the individual events:

```
# perf evlist
cycles
cache-misses
```

If however you want to have a combined report

```
# perf record -e '{cycles,cache-misses}' -a -- sleep 10
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 1.178 MB perf.data (~51478 samples) ]
```

Checking with evlist

```
# perf evlist --group
{cycles,cache-misses}
```

Now use **--group** on **report** and see the combined report

```
Samples: 9K of event 'anon group { cycles, cache-misses }', Event count (approx.): 1393423229
   6.98%   0.00%        events/2  [kernel.kallsyms]             [k] ioread32
   6.30%   1.44%            init  [kernel.kallsyms]             [k] intel_idle
   2.37%   0.12%         swapper  [kernel.kallsyms]             [k] intel_idle
   0.86%   0.85%         firefox  libpthread-2.12.so            [.] pthread_mutex_lock
   0.72%   0.44%            Xorg  [kernel.kallsyms]             [k] sock_poll
   0.58%   0.12%         firefox  [kernel.kallsyms]             [k] avtab_search_node
   0.58%   0.22%            Xorg  [kernel.kallsyms]             [k] do_select
   0.54%   0.39%         firefox  libpthread-2.12.so            [.] pthread_mutex_unlock
   0.46%   0.34%         firefox  [kernel.kallsyms]             [k] find_busiest_group
   0.41%   0.21%            init  [kernel.kallsyms]             [k] native_write_msr_safe
   0.37%   0.74%         firefox  libglib-2.0.so.0.2800.8       [.] g_main_context_check
   0.34%   0.48%         firefox  [kernel.kallsyms]             [k] fget_light
   0.33%   0.25%         firefox  [kernel.kallsyms]             [k] do_sys_poll
   0.32%   0.36%            init  [kernel.kallsyms]             [k] cpuidle_idle_call
   0.31%   0.02%            Xorg  [kernel.kallsyms]             [k] _spin_lock_irqsave
   0.30%   0.25%         firefox  libpthread-2.12.so            [.] pthread_cond_timedwait@@GLIBC_2.3.2
   0.29%   0.06%         firefox  firefox                       [.] 0x000000000000e0b6
   0.28%   0.00%            perf  [kernel.kallsyms]             [k] generic_exec_single
   0.26%   0.40%  plugin-containe  [kernel.kallsyms]            [k] find_busiest_group
   0.25%   0.52%         firefox  [kernel.kallsyms]             [k] sock_poll
   0.25%   0.00%           sleep  [kernel.kallsyms]             [k] native_write_msr_safe
   0.24%   0.05%         firefox  libpthread-2.12.so            [.] __pthread_mutex_cond_lock
   0.23%   0.29%         firefox  libpthread-2.12.so            [.] pthread_getspecific
   0.23%   0.36%            init  [kernel.kallsyms]             [k] _spin_lock
   0.23%   0.18%         firefox  [kernel.kallsyms]             [k] _spin_lock_irqsave
   0.22%   0.38%         firefox  [kernel.kallsyms]             [k] avc_has_perm_noaudit
   0.22%   0.11%            init  [kernel.kallsyms]             [k] ktime_get_real
   0.21%   0.23%         firefox  libxul.so                     [.] 0x000000000016229f
   0.21%   0.00%            init  [kernel.kallsyms]             [k] vsnprintf
```

Now you might be asking, can't I just use **--group** whenever I want? No. You have to use the “grouping” method of event **perf record -e '{<events>,<events>, , , , , }'**.

Let's look at another example with a slightly different record method to make sure it all makes sense. Let's add some probes ourselves:

```
# perf probe --add get_page
Added new event:
  probe:get_page       (on get_page)

You can now use it in all perf tools, such as:

      perf record -e probe:get_page -aR sleep 1

# perf probe --add __do_page_fault
Added new event:
  probe:__do_page_fault (on __do_page_fault)
```

```
You can now use it in all perf tools, such as:

      perf record -e probe:__do_page_fault -aR sleep 1

# perf probe --add zone_reclaimable_pages
Added new event:
  probe:zone_reclaimable_pages (on zone_reclaimable_pages)

You can now use it in all perf tools, such as:

      perf record -e probe:zone_reclaimable_pages -aR sleep 1
```

Check with **perf probe** what we've got so far:

```
# perf probe -l
/sys/kernel/debug/tracing/uprobe_events file does not exist - please rebuild kernel
with CONFIG_UPROBE_EVENTS.
  probe:__do_page_fault (on __do_page_fault@arch/x86/mm/fault.c)
  probe:get_page       (on get_page@mm/swap.c)
  probe:zone_reclaimable_pages (on zone_reclaimable_pages@mm/vmscan.c)
```

Let's start a recording session with all of our probes:

```
# perf record -e '{probe:*}' -ag -- sleep 10
[ perf record: Woken up 3 times to write data ]
[ perf record: Captured and wrote 1.591 MB perf.data (~69527 samples) ]
```

```
# perf evlist --group
{probe:zone_reclaimable_pages,probe:__do_page_fault,probe:get_page}
```

```
# perf report --group

Samples: 5K of event 'anon group { probe:zone_reclaimable_pages, probe:__do_page_fault, probe:get_page }', Event count (approx.): 553
     0.00%   0.00%  44.39%     0.00%   0.00%  44.39%         firefox  [kernel.kallsyms]      [k] get_page
     0.00%  78.39%  43.71%     0.00%  78.39%   0.00%         firefox  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   0.00%  26.06%     0.00%   0.00%  26.06%            perf  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%  24.54%     0.00%   0.00%  24.54%       khugepaged  [kernel.kallsyms]      [k] get_page
     0.00%  12.87%   7.04%     0.00%  12.87%   0.00%            perf  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   8.08%   0.56%     0.00%   8.08%   0.00%           sleep  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   0.00%   3.38%     0.00%   0.00%   3.38%  irq/39-iwlwifi  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%   0.56%     0.00%   0.00%   0.56%           sleep  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%   0.34%     0.00%   0.00%   0.34%      jbd2/dm-2-8  [kernel.kallsyms]      [k] get_page
     0.00%   0.56%   0.31%     0.00%   0.56%   0.00%       irqbalance  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   0.00%   0.31%     0.00%   0.00%   0.31%       irqbalance  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%   0.23%     0.00%   0.00%   0.23%            Xorg  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%   0.17%     0.00%   0.00%   0.17%      jbd2/dm-0-8  [kernel.kallsyms]      [k] get_page
     0.00%   0.05%   0.00%     0.00%   0.05%   0.00%            Xorg  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   0.05%   0.03%     0.00%   0.05%   0.00%            ntpd  [kernel.kallsyms]      [k] __do_page_fault
     0.00%   0.00%   0.03%     0.00%   0.00%   0.03%            ntpd  [kernel.kallsyms]      [k] get_page
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [drm]                  [k] drm_ioctl
```

```
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [i915]                 [k] i915_gem_object_get_pages
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [i915]                 [k] i915_gem_object_get_pages_gtt
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [i915]                 [k] i915_gem_pwrite_ioctl
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] do_vfs_ioctl
     0.00%   0.05%   0.00%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] page_fault
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] shmem_getpage_gfp
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] shmem_read_mapping_page_gfp
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] sys_ioctl
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] system_call_fastpath
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  [kernel.kallsyms]      [k] vfs_ioctl
     0.00%   0.00%   0.23%     0.00%   0.00%   0.00%            Xorg  libc-2.12.so           [.] __GI___ioctl
     0.00%   0.05%   0.00%     0.00%   0.00%   0.00%            Xorg  libpixman-1.so.0.32.4  [.] 0x0000000000079688
Press '?' for help on key bindings
```

3.4 - Graphs & Maps

3.4.1 - Creating HeatMap pictures

FlameGraphs are nice but I also think that HeatMaps are very effective. It may be personal preference so you should try both. This example shows how to produce a HeatMap which displays the actual time of a block IO. In the HeatMap, the darker the color, the more samples that fit that count. The x (horizontal) axis will generally be the elapsed time of the sampling; and the y (vertical) axis the length of the sampled item.

First. An example of displaying raw data to report block's requested and completed:

```
# perf record -e block:block_rq_issue -e block:block_rq_complete -a sleep 60
[ perf record: Woken up 21 times to write data ]
[ perf record: Captured and wrote 5.965 MB perf.data (~260630 samples) ]
```

```
# perf script
     flush-253:2  3189 [000] 12335.781590: block:block_rq_issue: 8,0 W 0 () 267375432 + 8 [flush-253:2]
            init     1 [004] 12335.783436: block:block_rq_complete: 8,0 W () 267375432 + 8 [0]
            init     1 [004] 12335.783480: block:block_rq_issue: 8,0 W 0 () 267375464 + 32 [swapper]
            init     1 [004] 12335.783566: block:block_rq_complete: 8,0 W () 267375464 + 32 [0]
     jbd2/dm-2-8  1914 [004] 12336.763855: block:block_rq_issue: 8,0 WS 0 () 362126360 + 8 [jbd2/dm-2-8]
     jbd2/dm-2-8  1914 [004] 12336.763911: block:block_rq_issue: 8,0 WS 0 () 362126368 + 40 [jbd2/dm-2-8]
            init     1 [004] 12336.765668: block:block_rq_complete: 8,0 WS () 362126360 + 8 [0]
            init     1 [004] 12336.765699: block:block_rq_complete: 8,0 WS () 362126368 + 40 [0]
     jbd2/dm-2-8  1914 [004] 12336.765753: block:block_rq_issue: 8,0 FWS 0 () 18446744073709551615 + 0 [jbd2/dm-2-8]
            init     1 [004] 12336.767417: block:block_rq_complete: 8,0 WS () 0 + 0 [0]
            init     1 [004] 12336.767425: block:block_rq_issue: 8,0 FWS 0 () 18446744073709551615 + 0 [swapper]
            init     1 [004] 12336.767630: block:block_rq_complete: 8,0 WS () 0 + 0 [0]
        kdmflush  1865 [004] 12336.767647: block:block_rq_issue: 8,0 WS 0 () 362126408 + 8 [kdmflush]
            init     1 [004] 12336.767677: block:block_rq_complete: 8,0 WS () 362126408 + 8 [0]
            init     1 [004] 12336.767683: block:block_rq_issue: 8,0 FWS 0 () 18446744073709551615 + 0 [swapper]
            init     1 [004] 12336.769243: block:block_rq_complete: 8,0 WS () 362126408 + 0 [0]
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

Second. And now using **`perf script`** to read and extract data in preparation to creating a HeatMap picture:

```
# perf script | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 } $5 ~
/complete/ { if (itime = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000, ($4 - itime)
* 1000000; ts[$6, $10] = 0 } }' > out.lat_us
```

Let's look at this in detail. I've renamed the element '**`l`**' from Brendan's original command line as **`1`** (one) looks like **`l`** (L) and it was so confusing. So I renamed the element to **`itime`**.

PART 1 – Remove the '**`:`**'

```
# perf script | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 } $5 ~
/complete/ { if (itime = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000, ($4 - itime)
* 1000000; ts[$6, $10] = 0 } }'

     flush-2532  3189 [000] 12335.781590 blockblock_rq_issue 8,0 W 0 () 267375432 + 8 [flush-2532]
          init     1 [004] 12335.783436 blockblock_rq_complete 8,0 W () 267375432 + 8 [0]
```

```
           init     1 [004] 12335.783480 blockblock_rq_issue 8,0 W 0 () 267375464 + 32 [swapper]
           init     1 [004] 12335.783566 blockblock_rq_complete 8,0 W () 267375464 + 32 [0]
    jbd2/dm-2-8  1914 [004] 12336.763855 blockblock_rq_issue 8,0 WS 0 () 362126360 + 8 [jbd2/dm-2-8]
    jbd2/dm-2-8  1914 [004] 12336.763911 blockblock_rq_issue 8,0 WS 0 () 362126368 + 40 [jbd2/dm-2-8]
           init     1 [004] 12336.765668 blockblock_rq_complete 8,0 WS () 362126360 + 8 [0]
           init     1 [004] 12336.765699 blockblock_rq_complete 8,0 WS () 362126368 + 40 [0]
    jbd2/dm-2-8  1914 [004] 12336.765753 blockblock_rq_issue 8,0 FWS 0 () 18446744073709551615 + 0 [jbd2/dm-2-8]
           init     1 [004] 12336.767417 blockblock_rq_complete 8,0 WS () 0 + 0 [0]
```

PART 2 – Now test if arg5 contains either the string "**issue**" or "**complete**". If is contains **issue** then save the start (**issue**) time from arg4 into our **ts[]** array (using the elements disk device ID and block offset). If we find a **complete** for the same disk device ID and block offset then we can proceed with the **printf**. If there's a **complete** and no **issue** time, we ignore it. Why? What if when we started recording, the **issue** had already passed?

Note. If **issue**, arg10 is the block address.
If **complete**, arg9 is the block address.
This Note should explain why we are gathering **$6**,**$10** and **$6**,**$9**

```
# perf

     arg1      arg2 arg3      arg4             arg5            arg6 arg7 arg8  arg9       arg10
  flush-2532   3189 [000] 12335.781590 blockblock_rq_issue       8,0   W    0    ()     267375432    + 8 [flush-2532]
        init      1 [004] 12335.783436 blockblock_rq_complete   8,0   W   () 267375432     +        8 [0]
        init      1 [004] 12335.783480 blockblock_rq_issue       8,0   W    0    ()     267375464    + 32 [swapper]
        init      1 [004] 12335.783566 blockblock_rq_complete   8,0   W   () 267375464     +        32 [0]
 jbd2/dm-2-8   1914 [004] 12336.763855 blockblock_rq_issue       8,0   WS   0    ()     362126360    + 8 [jbd2/dm-2-8]
 jbd2/dm-2-8   1914 [004] 12336.763911 blockblock_rq_issue       8,0   WS   0    ()     362126368    + 40 [jbd2/dm-2-8]
        init      1 [004] 12336.765668 blockblock_rq_complete   8,0   WS  () 362126360     +        8 [0]
        init      1 [004] 12336.765699 blockblock_rq_complete   8,0   WS  () 362126368     +        40 [0]
```

PART 3 – To detemine latency time we need to subtract the time of the **issue** (was held in our **ts[]** array and moved into **itime**) from the **complete** time currently held in arg4. We can then print the **complete** time in microseconds and the microsecond latency time.

```
# perf script | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 } $5 ~
/complete/ { if (itime = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000, ($4 - itime)
* 1000000; ts[$6, $10] = 0 } }' | head

  complete  latency
    time
12335783436 1846
12335783566 86
12336765668 1813
12336765699 1788
12336767677 30
12336769243 1596
12338764995 1806
12338765015 1686
12338765079 1728
12338774381 48
```

PART 4 – Once you've printed the **complete** time and latency time, clean out the array element **ts[]** for this disk device ID and block offset.

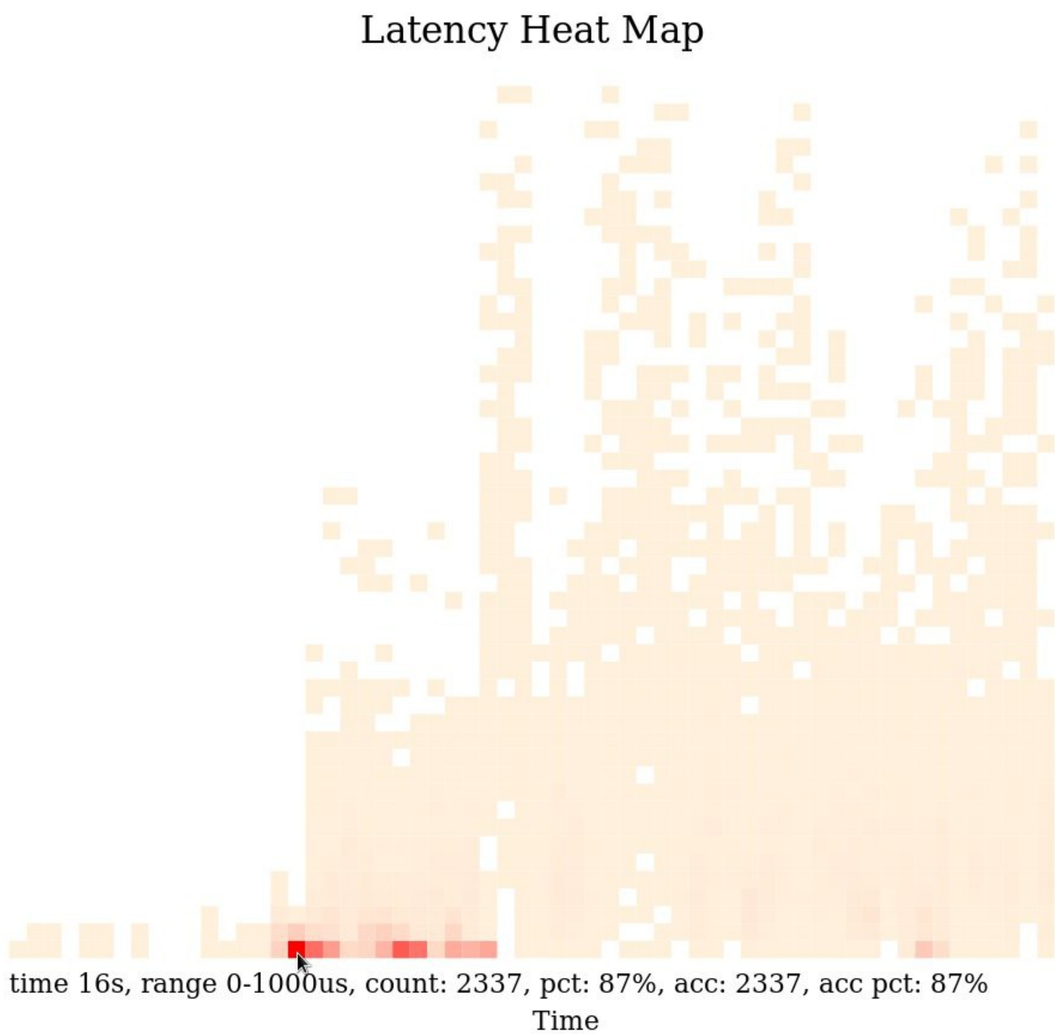
```
perf script | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 } $5 ~
/complete/ { if itime = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000, ($4 - itime)
```

```
* 1000000; ts[$6, $10] = 0 } }'
```

Third and finally. Creating the HeatMap picture:

See Brendan's documentation on how to acquire his **trace2heatmap.pl** script

```
# ./trace2heatmap.pl --unitstime=us --unitslat=us --maxlat=50000 out.lat_us > out.svg
```



The x (horizontal) axis is the elapsed time of the sampling period.
The y (vertical) axis is the latency period.

If you Mouse over the RED sample it reports a time period of 16 secs (x-axis is 0-60secs). It shows a range of 0-1000us (0-1millisec). A count of 2337 with a % of 87%. Meaning 87% of all samples in this 16 second sample time produced an IO Latency of less that 1millisec (1000us). If you went one blocm up on the 16 sec time sample, the next entry reports 1000-2000us and a count os 333 (12%). Meaning 99% of all samples at this time slot were less than 2millisecs. Make sense now?

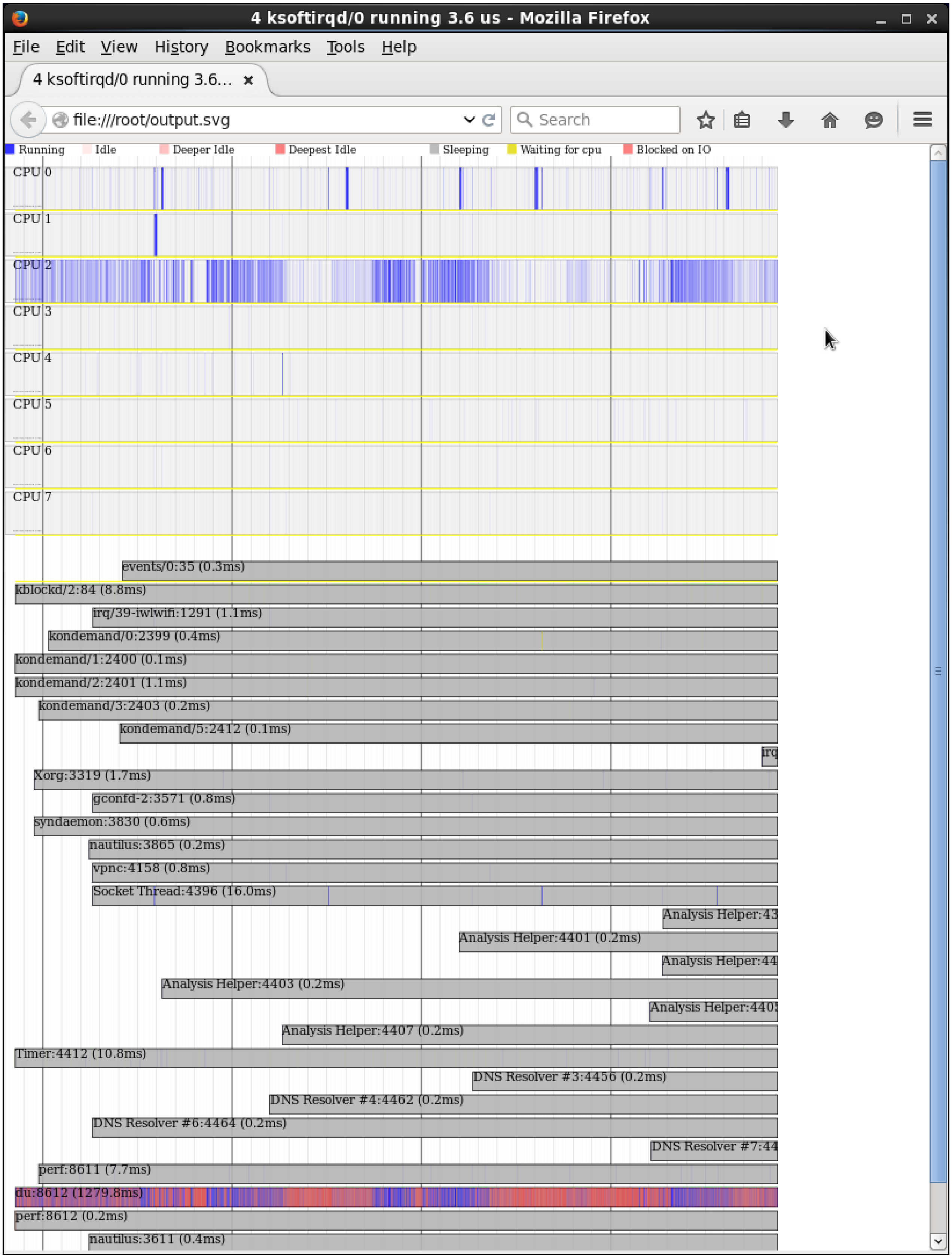
3.4.2 - Timecharts with perf

You can generate some useful timecharts which show overall system activity during the monitored period

```
# perf timechart record du -s
19024228      .
```

```
[ perf record: Woken up 41 times to write data ]
[ perf record: Captured and wrote 11.438 MB perf.data (~499722 samples) ]

# perf timechart
Written 4.0 seconds of trace to output.svg.
```

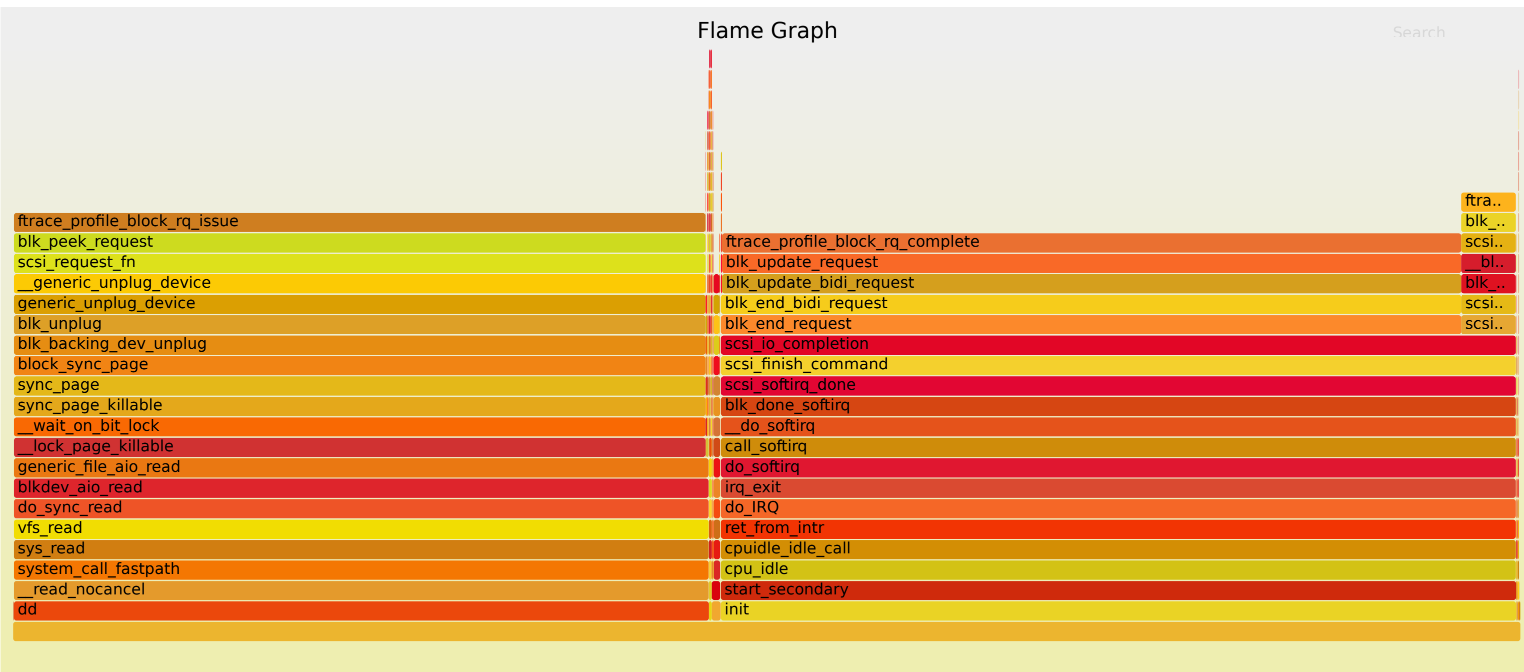
Now you can display the output.svg file. Just one comment, I tried this on RHEL6 and it took many seconds to minutes to properly display all the svg file especially after trying to re-size it. Here's an example of what it can produce:


4 ksoftirqd/0 running 3.6 us - Mozilla Firefox
File Edit View History Bookmarks Tools Help
4 ksoftirqd/0 running 3.6...
file:///root/output.svg
Search
Running Idle Deeper Idle Deepest Idle Sleeping Waiting for cpu Blocked on IO
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
events/0:35 (0.3ms)
kblockd/2:84 (8.8ms)
irq/39-iwlwifi:1291 (1.1ms)
kondemand/0:2399 (0.4ms)
kondemand/1:2400 (0.1ms)
kondemand/2:2401 (1.1ms)
kondemand/3:2403 (0.2ms)
kondemand/5:2412 (0.1ms)
irq
Xorg:3319 (1.7ms)
gconfd-2:3571 (0.8ms)
syndaemon:3830 (0.6ms)
nautilus:3865 (0.2ms)
vpnc:4158 (0.8ms)
Socket Thread:4396 (16.0ms)
Analysis Helper:43
Analysis Helper:4401 (0.2ms)
Analysis Helper:44
Analysis Helper:4403 (0.2ms)
Analysis Helper:440
Analysis Helper:4407 (0.2ms)
Timer:4412 (10.8ms)
DNS Resolver #3:4456 (0.2ms)
DNS Resolver #4:4462 (0.2ms)
DNS Resolver #6:4464 (0.2ms)
DNS Resolver #7:44
perf:8611 (7.7ms)
du:8612 (1279.8ms)
perf:8612 (0.2ms)
nautilus:3611 (0.4ms)

3.4.3 - An example of a FlameGraph

This was covered in Brendan Gregg's document but I thought I'd select something different that might resonate more with Red Hatters. FlameGraphs are nice visuals. I'm not sure how much value they are for nuts and bolts investigations but like any type of graphical display, they have the possibility of highlighting events or sequences. Here's the sequence for block request issues and completes.

```
# perf record -e block:block_rq_issue -e block:block_rq_complete -ag -- sleep 60
[ perf record: Woken up 115 times to write data ]
[ perf record: Captured and wrote 29.661 MB perf.data (105249 samples) ]

# perf script | ./stackcollapse-perf.pl > out.perf-folded
Failed to open /tmp/perf-14604.map, continuing without symbols
Failed to open [vsyscall], continuing without symbols

# cat out.perf-folded | ./flamegraph.pl > perf-kernel2.svg
```



The x-axis (horizontal) is showing the number of samples. The samples are stacks. The bottom line is all stacks. The line above comprises the processes and the number of stack samples captured for each.

The y-axis (vertical) is show the stack depth that each process attained and the functions gathered from the stack for each stack sample.

What we're seeing here is quite easy to explain. I was running a **`dd if=/dev/sda of=/dev/null`**

bs=4k count=100000000 while the **perf record** was running. So in this FlameGraph, there are 105,249 samples. **dd** comprised 48,511 samples and as can be seen from the stack followed up the y-axis, the last entry is for **ftrace_profile_block_rq_issue()**. Well that makes sense. The other large sampling process was **init** which shows 55,512 samples. Follow that up the y-axis (horizontal) and we see the process was sitting idle for 55,500 samples and took a ret_from_intr() for 55,490 samples. The top stack call displayed was the **ftrace_profile_block_rq_complete()** which also makes sense.

So what is this telling us. For the vast mority of the 60 second sampling period, there wasn't much else running that required disk I/O as **dd** was responsible for both the issue's and complete's. That can be assertained by the fact that the interrupts for the completion of the IO which are reported for **init** are actually those that belong to **dd**.

What you can see additional in the interrupt handling is that there were also cases where **scsi_io_completion()** also called **scsi_next_command()** (3,808 samples).

There's also another process between **dd** and **init**. If you leave the mouse over the **svg** file it shows that belongs to **firefox**.

While this FlameGraph looks simple, the intention was to give you a more friendly example of what can be produced. Put into perspective, it's showing the detailed levels for each process stack, the counts of that function in the stack and the % of that count based on the total number of samples captured for the period.

FYI. Brendan provides **perl** scripts that can be used with other tools including FreeBSD pmcstat (hwpmc), DTrace, SystemTap, and many other profilers.

```
# ll FlameGraph
- - - - - 8< - - - - -
-rwxr-xr-x. 1 root root    2304 Nov  5 14:47 stackcollapse-elfutils.pl
-rwxr-xr-x. 1 root root    1816 Nov  5 14:47 stackcollapse-gdb.pl          <= gdb stacks
-rwxr-xr-x. 1 root root     504 Nov  5 14:47 stackcollapse-instruments.pl  <= Xcode Instruments
-rwxr-xr-x. 1 root root    4501 Nov  5 14:47 stackcollapse-jstack.pl       <= Java jstack
-rwxr-xr-x. 1 root root    1859 Nov  5 14:47 stackcollapse-ljp.awk         <= Lightweight Java
-rwxr-xr-x. 1 root root    6815 Nov  5 14:47 stackcollapse-perf.pl         <= perf script stacks
-rwxr-xr-x. 1 root root    2554 Nov  5 14:47 stackcollapse.pl              <= Dtrace stacks
-rwxr-xr-x. 1 root root    2663 Nov  5 14:47 stackcollapse-pmc.pl          <= FreeBSD pmcstat
-rwxr-xr-x. 1 root root    1569 Nov  5 14:47 stackcollapse-recursive.pl
-rwxr-xr-x. 1 root root    2310 Nov  5 14:47 stackcollapse-stap.pl         <= SystemTap stacks
-rw-r--r--. 1 root root    2030 Nov  5 14:47 stackcollapse-vtune.pl        <= Intel Vtune
```

For more information, you should read the **README.md** file in the **FlameGraph** directory

Additionally, Brendan also supplies a number of demo graphs which might spur thoughts and interests.

```
# ll FlameGraph/demos/
total 4972
-rw-r--r--. 1 root root 120082 Nov  5 14:47 brkbytes-mysql.svg
-rw-r--r--. 1 root root  10733 Nov  5 14:47 cpu-grep.svg
-rw-r--r--. 1 root root 416888 Nov  5 14:47 cpu-illumos-ipdce.svg
-rw-r--r--. 1 root root 586699 Nov  5 14:47 cpu-illumos-syscalls.svg
-rw-r--r--. 1 root root 115788 Nov  5 14:47 cpu-illumos-tcpfuse.svg
-rw-r--r--. 1 root root 166796 Nov  5 14:47 cpu-iozone.svg
-rw-r--r--. 1 root root  96240 Nov  5 14:47 cpu-ipnet-diff.svg
-rw-r--r--. 1 root root 111366 Nov  5 14:47 cpu-linux-tar.svg
-rw-r--r--. 1 root root  70785 Nov  5 14:47 cpu-linux-tcpsend.svg
-rw-r--r--. 1 root root 301303 Nov  5 14:47 cpu-mixedmode-flamegraph-java.svg
-rw-r--r--. 1 root root 809833 Nov  5 14:47 cpu-mysql-filt.svg
-rw-r--r--. 1 root root 629353 Nov  5 14:47 cpu-mysql.svg
-rw-r--r--. 1 root root 210664 Nov  5 14:47 cpu-qemu-both.svg
-rw-r--r--. 1 root root 460507 Nov  5 14:47 cpu-zoomable.html
-rw-r--r--. 1 root root  82855 Nov  5 14:47 hotcold-kernelthread.svg
-rw-r--r--. 1 root root  12225 Nov  5 14:47 io-gzip.svg
-rw-r--r--. 1 root root  52226 Nov  5 14:47 io-mysql.svg
-rw-r--r--. 1 root root  54088 Nov  5 14:47 mallocbytes-bash.svg
-rw-r--r--. 1 root root  10458 Nov  5 14:47 off-bash.svg
-rw-r--r--. 1 root root  78558 Nov  5 14:47 off-mysql-busy.svg
-rw-r--r--. 1 root root  53869 Nov  5 14:47 off-mysql-idle.svg
-rw-r--r--. 1 root root 304118 Nov  5 14:47 palette-example-broken.svg
-rw-r--r--. 1 root root 281453 Nov  5 14:47 palette-example-working.svg
-rw-r--r--. 1 root root    252 Nov  5 14:47 README
```

3.4.4 - Brendan Gregg's "perf to histogram"

Brendan has a script he has developed which can produce some nice Historgrams. It is located at:

https://github.com/brendangregg/perf-tools/blob/master/misc/perf-stat-hist

I did find a typo in the version I just reviewed. Very simple to spot and fix so I'll leave that for you readers to also determine if it has not been corrected when you pull it down. (Experience is everything....)

What can it do? First make sure you have the **debugfs** mounted before you start

```
# mount -t debugfs none /sys/kernel/debug/

# /home/sjoh/scripts2/perf-stat-hist.sh net:net_dev_xmit len 10
Tracing net:net_dev_xmit, power-of-4, max 1048576, for 10 seconds...

            Range          : Count    Distribution
              -> -1        : 0        |                                      |
            0 -> 0         : 0        |                                      |
            1 -> 3         : 0        |                                      |
            4 -> 15        : 0        |                                      |
           16 -> 63        : 0        |                                      |
           64 -> 255       : 207      |######################################|
          256 -> 1023      : 19       |####                                  |
         1024 -> 4095      : 86       |################                      |
         4096 -> 16383     : 0        |                                      |
        16384 -> 65535     : 0        |                                      |
        65536 -> 262143    : 0        |                                      |
       262144 -> 1048575   : 0        |                                      |
      1048576 ->           : 0        |                                      |
```

Another example:

```
# time /home/sjoh/scripts2/perf-stat-hist.sh syscalls:sys_enter_read count 10
Tracing syscalls:sys_enter_read, power-of-4, max 1048576, for 10 seconds...

            Range          : Count    Distribution
              -> -1        : 11       |#                                     |
            0 -> 0         : 11       |#                                     |
            1 -> 3         : 513      |######################################|
            4 -> 15        : 73       |######                                |
           16 -> 63        : 44       |####                                  |
           64 -> 255       : 71       |######                                |
          256 -> 1023      : 12       |#                                     |
```

```
         1024 -> 4095       : 94       |#######                                  |
         4096 -> 16383      : 63       |#####                                    |
        16384 -> 65535      : 11       |#                                        |
        65536 -> 262143     : 0        |                                         |
       262144 -> 1048575    : 0        |                                         |
      1048576 ->            : 0        |                                         |

real    0m10.053s
user    0m0.029s
sys     0m0.016s
```

In case you are wondering about the format of the command lines from above, here's the breakdown. The items color coded GREEN and BLUE highlighted above are variables. As we discussed previously, this needs some exploration into the **/sys/kernel/debug/tracing/events/......** directories:

```
# /home/sjoh/scripts2/perf-stat-hist.sh net:net_dev_xmit len 10
                                        ^^^^^^^^^^^^^^^^
                                            event        ^^^
                                                    variable ^^
                                                             # Seconds to run
```

```
# cat /sys/kernel/debug/tracing/events/net/net_dev_xmit/format
name: net_dev_xmit
ID: 735
format:
      field:unsigned short common_type;            offset:0;    size:2;     signed:0;
      field:unsigned char common_flags;            offset:2;    size:1;     signed:0;
      field:unsigned char common_preempt_count;    offset:3;    size:1;     signed:0;
      field:int common_pid;                        offset:4;    size:4;     signed:1;
      field:int common_lock_depth;                 offset:8;    size:4;     signed:1;

      field:void * skbaddr;                        offset:16;   size:8;     signed:0;
      field:unsigned int len;                      offset:24;   size:4;     signed:0;
      field:int rc;                                offset:28;   size:4;     signed:1;
      field:__data_loc char[] name;                offset:32;   size:4;     signed:0;

print fmt: "dev=%s skbaddr=%p len=%u rc=%d", __get_str(name), REC->skbaddr, REC->len, REC-
>rc
```

```
# time /home/sjoh/scripts2/perf-stat-hist.sh syscalls:sys_enter_read count 10
                                             ^^^^^^^^^^^^^^^^^^^^^^^
                                                       event          ^^^^^
                                                                 variable ^^
                                                                          # Seconds
                                                                          to run
```

```
# cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_read/format
name: sys_enter_read
ID: 423
format:
	field:unsigned short common_type;	offset:0;	size:2;	signed:0;
	field:unsigned char common_flags;	offset:2;	size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1;	signed:0;
	field:int common_pid;	offset:4;	size:4;	signed:1;
	field:int common_lock_depth;	offset:8;	size:4;	signed:1;

	field:unsigned int fd;	offset:16;	size:8;	signed:0;
	field:char * buf;	offset:24;	size:8;	signed:0;
	field:size_t count;	offset:32;	size:8;	signed:0;

print fmt: "fd: 0x%08lx, buf: 0x%08lx, count: 0x%08lx", ((unsigned long)(REC->fd)),
((unsigned long)(REC->buf)), ((unsigned long)(REC->count))
```

Here's another example. This time I'm crudely using the sector number from **block:block_rq_issue** to get a map of the area of the disk. Actually, I used this example to show that you can change the bucket ranges easily to suit your needs. The first two (default and a supplied bucket range) I used logarithmic bucket scaling and the 3rd report I used a linear bucket scaling.

```
# perf-stat-hist.sh block:block_rq_issue sector 20
Tracing block:block_rq_issue, power-of-4, max 1048576, for 20 seconds...

            Range          : Count    Distribution
              -> -1        : 1        |#                                      |
            0 -> 0         : 2        |#                                      |
            1 -> 3         : 2        |#                                      |
            4 -> 15        : 2        |#                                      |
           16 -> 63        : 2        |#                                      |
           64 -> 255       : 3        |#                                      |
          256 -> 1023      : 2        |#                                      |
         1024 -> 4095      : 2        |#                                      |
         4096 -> 16383     : 3        |#                                      |
        16384 -> 65535     : 3        |#                                      |
        65536 -> 262143    : 4        |#                                      |
       262144 -> 1048575   : 4        |#                                      |
      1048576 ->           : 81131    |#######################################|
      ^^^^^^^    ^^^^^^^
```

This is the range of sector numbers for all block IO requests. (basically it is showing us the sector range across the device (where is the device most busy), although I will emphasize very crudely)

```
# perf-stat-hist.sh -b "100 1000 10000 100000 1000000 10000000 100000000 1000000000 10000000000"  block:block_rq_issue sector 20
Tracing block:block_rq_issue, specified buckets, for 20 seconds...

             Range          : Count    Distribution
               -> 99        : 4        |#                                      |
           100 -> 999       : 4        |#                                      |
          1000 -> 9999      : 4        |#                                      |
         10000 -> 99999     : 5        |#                                      |
        100000 -> 999999    : 5        |#                                      |
       1000000 -> 9999999   : 9        |#                                      |
      10000000 -> 99999999  : 37       |#                                      |
     100000000 -> 999999999 : 81154    |#######################################|
    1000000000 -> 9999999999: 12       |#                                      |
   10000000000 ->           : 31       |#                                      |
#
```

```
# perf-stat-hist.sh -b "100000000 200000000 300000000 400000000 500000000 600000000 700000000 800000000 900000000 1000000000 1100000000 1200000000"  block:block_rq_issue sector 20
Tracing block:block_rq_issue, specified buckets, for 20 seconds...

            Range          : Count    Distribution
              -> 99999999  : 136      |#                                      |
    100000000 -> 199999999 : 18       |#                                      |
    200000000 -> 299999999 : 9        |#                                      |
    300000000 -> 399999999 : 38627    |#######################################|
    400000000 -> 499999999 : 38273    |#######################################|
    500000000 -> 599999999 : 6        |#                                      |
    600000000 -> 699999999 : 6        |#                                      |
    700000000 -> 799999999 : 5        |#                                      |
    800000000 -> 899999999 : 6        |#                                      |
    900000000 -> 999999999 : 4        |#                                      |
   1000000000 -> 1099999999: 6        |#                                      |
   1100000000 -> 1199999999: 21       |#                                      |
   1200000000 ->           : 36       |#                                      |
```

sector number range

from -> to

3.5 - Tracing processes, locks, memory, other tools etc

3.5.1 - How performant perf is compared to strace

Some lines removed for easier reading. Timing highlighted.

dd of a device (it was previously read and assured to be in cache):

```
# dd if=/dev/sda of=/dev/null bs=4k count=100000
100000+0 records in
100000+0 records out
409600000 bytes (410 MB) copied, 0.0670868 s, 6.1 GB/s
```

Now “monitor” the **dd** with **perf**:

```
# perf stat -e 'syscalls:sys_enter_*' dd if=/dev/sda of=/dev/null bs=4k count=100000
100000+0 records in
100000+0 records out
409600000 bytes (410 MB) copied, 0.110195 s, 3.7 GB/s

 Performance counter stats for 'dd if=/dev/sda of=/dev/null bs=4k count=100000':

                 0      syscalls:sys_enter_socket
                 0      syscalls:sys_enter_socketpair
- - - - - - - - 8< - - - - - - - - -
                16      syscalls:sys_enter_mmap
                 0      syscalls:sys_enter_uname

       0.111018443 seconds time elapsed
```

And finally, **strace** of a **dd**:

```
# strace -c dd if=/dev/sda of=/dev/null bs=4k count=100000
100000+0 records in
100000+0 records out
409600000 bytes (410 MB) copied, 2.08917 s, 196 MB/s

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 59.61    0.000865           0    100005           read
- - - - - - - - - - 8< - - - - - - - - -
  0.00    0.000000           0         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    0.001451                200087         7 total
```

3.5.2 - Tracing processes (similar to strace)

perf has a strace type ability. It's worth checking out. You can trace a process (and all of it's threads), just a TID, all calls made by a CPU (or range of CPU's), a range of syscalls or all of them amongst other features. Check the perf-trace manpage. Why use this over strace? It appears to be far more performant.

Here's an example of tracing a process (and it's threads). Note that the time stamp is based on the interval time from the first sample.

```
# perf trace -p 4239
     0.000 ( 0.000 ms): Timer/4278  ... [continued]: futex()) = -1 ETIMEDOUT Connection timed out
     0.016 ( 0.002 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
     0.030 ( 0.007 ms): Timer/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1       ) = 1
     0.055 ( 0.000 ms): firefox/4239  ... [continued]: poll()) = 1
     0.073 ( 0.003 ms): firefox/4239 read(fd: 25<pipe:[32217]>, buf: 0x7fff6537105f, count: 1        ) = 1
     0.149 ( 0.003 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
     0.159 ( 0.002 ms): firefox/4239 poll(ufds: 0x7fcf1904e0e0, nfds: 11                         ) = 0 Timeout
     0.162 ( 0.001 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
     0.164 ( 0.001 ms): firefox/4239 poll(ufds: 0x7fcf1904e0e0, nfds: 11                         ) = 0 Timeout
     0.168 ( 0.001 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
     0.170 ( 0.001 ms): firefox/4239 poll(ufds: 0x7fcf1904e0e0, nfds: 11                         ) = 0 Timeout
     0.172 ( 0.001 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

Add **-T** and you can see the timestamp based on uptime.

```
# perf trace -p 4239 -T
1393091029.386 ( 0.000 ms): Timer/4278  ... [continued]: futex()) = -1 ETIMEDOUT Connection timed out
1393091029.399 ( 0.001 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091040.494 (11.086 ms): Timer/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268849123, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393091040.500 ( 0.001 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091049.587 ( 9.083 ms): Timer/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268849125, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393091049.593 ( 0.001 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091062.687 (13.089 ms): Timer/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268849127, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393091062.698 ( 0.002 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091063.759 ( 1.054 ms): Timer/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268849129, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393091063.763 ( 0.001 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091072.851 ( 9.084 ms): Timer/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268849131, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393091072.857 ( 0.001 ms): Timer/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1             ) = 0
1393091076.307 ( 0.017 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = 32
1393091076.313 ( 0.001 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
1393091076.326 ( 0.004 ms): firefox/4239 poll(ufds: 0x7fcf1904e0e0, nfds: 11                         ) = 1
1393091076.341 ( 0.003 ms): firefox/4239 read(fd: 25<pipe:[32217]>, buf: 0x7fff6537105f, count: 1        ) = 1
1393091076.343 ( 0.001 ms): firefox/4239 recvfrom(fd: 4, ubuf: 0x7fcfb65db074, size: 4096              ) = -1 EAGAIN Resource
temporarily unavailable
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

To sample a CPU use -C and the CPU or range of CPUs (**-C0** or **-C1,4** or **-C10-13**)

```
# perf trace -C3 -T
1393388965.306 ( 0.000 ms): firefox/4278  ... [continued]: futex()) = -1 ETIMEDOUT Connection timed out
1393388965.322 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393388965.327 ( 0.003 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389063.534 (98.194 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880671, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389063.557 ( 0.002 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389092.679 (29.114 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880673, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389092.706 ( 0.004 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389102.905 (10.190 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880675, utime: 0x7fcfa25fec10,
val3: 4294967295) = 0
1393389102.913 ( 0.002 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389115.022 (12.107 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880677, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389115.027 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389115.042 ( 0.003 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389136.167 (21.123 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880679, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389136.186 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389137.267 ( 1.080 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880681, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389137.272 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389137.291 ( 0.005 ms): firefox/4278 write(fd: 23<pipe:[32214]>, buf: 0x39d6a2cfd4, count: 1             ) = 1
1393389161.423 (24.129 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880683, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389161.446 ( 0.002 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389161.457 ( 0.005 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389210.621 (49.159 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880685, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389210.639 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389210.645 ( 0.003 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389250.101 ( 0.000 ms): gnome-settings/3596  ... [continued]: poll()) = 1
1393389250.123 ( 0.003 ms): gnome-settings/3596 recvfrom(fd: 3, ubuf: 0x17986c4, size: 4096                          ) = 32
1393389250.128 ( 0.001 ms): gnome-settings/3596 recvfrom(fd: 3, ubuf: 0x17986c4, size: 4096                          ) = -1 EAGAIN
Resource temporarily unavailable
1393389250.147 ( 0.001 ms): gnome-settings/3596 recvfrom(fd: 3, ubuf: 0x17986c4, size: 4096                          ) = -1 EAGAIN
Resource temporarily unavailable
1393389397.004 (186.356 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880687, utime:
0x7fcfa25fec10, val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389397.028 ( 0.002 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389397.041 ( 0.006 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389458.255 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389458.266 ( 0.005 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389458.349 ( 0.079 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880691, utime: 0x7fcfa25fec10,
val3: 4294967295) = 0
1393389458.365 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389461.439 ( 3.073 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880693, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389461.444 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389461.459 ( 0.003 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389474.590 (13.129 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880695, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389474.614 ( 0.002 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389474.626 ( 0.006 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389475.732 ( 1.070 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880697, utime: 0x7fcfa25fec10,
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
1393389475.747 ( 0.001 ms): firefox/4278 futex(uaddr: 0x7fcfaa80b240, op: WAKE|PRIV, val: 1                  ) = 0
1393389475.750 ( 0.002 ms): firefox/4278 write(fd: 26<pipe:[32217]>, buf: 0x7fcfa25feb9f, count: 1           ) = 1
1393389477.823 ( 2.072 ms): firefox/4278 futex(uaddr: 0x7fcfaa85878c, op: WAIT_BITSET|PRIV|CLKRT, val: 268880699, utime: 0x7fcfa25fec10,
```

```
val3: 4294967295) = -1 ETIMEDOUT Connection timed out
```

Be aware, you can overrun the server with tracing. Here's an example. The start of the tracing was in fact, perfectly good... then a few seconds into the streaming these errors started appearing. There is a comment in the code about this:

```
/*
 * The kernel collects the number of events it couldn't send in a stretch and
 * when possible sends this number in a PERF_RECORD_LOST event. The number of
 * such "chunks" of lost events is stored in .nr_events[PERF_EVENT_LOST] while
 * total_lost tells exactly how many events the kernel in fact lost, i.e. it is
 * the sum of all struct lost_event.lost fields reported.
 *
 * The total_period is needed because by default auto-freq is used, so
 * multipling nr_events[PERF_EVENT_SAMPLE] by a frequency isn't possible to get
 * the total number of low level events, it is necessary to to sum all struct
 * sample_event.period and stash the result in total_period.
 */
```

```
# perf trace -C1-3
- - - - - - - - - - - 8< - - - - - - - - - - - -
   459.105 ( 0.002 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 113              ) = 113
   459.109 ( 0.002 ms): perf/13181 lstat(filename: 0x7ffe1a58a190, statbuf: 0x7ffe1a589100              ) = 0
LOST 1 events!
LOST 1 events!
LOST 1 events!
LOST 2 events!
   459.126 ( 0.003 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 112              ) = 0
LOST 3 events!
   459.132 ( 0.009 ms): perf/13181  ... [continued]: write()) = 113
   459.136 ( 0.001 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 111              ) = 111
   459.141 ( 0.001 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 112              ) = 112
LOST 4 events!
LOST 1 events!
   459.154 ( 0.002 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 111              ) = 111
   459.158 ( 0.001 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 112              ) = 112
LOST 4 events!
LOST 5 events!
LOST 2 events!
   459.176 ( 0.003 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 113              ) = 0
LOST 3 events!
   459.181 ( 0.009 ms): perf/13181  ... [continued]: write()) = 113
   459.186 ( 0.001 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 111              ) = 111
   459.190 ( 0.001 ms): perf/13181 write(fd: 1</dev/pts/2>, buf: 0x7f8f315b6000, count: 112              ) = 112
LOST 4 events!
LOST 1 events!
LOST 1 events!
- - - - - - - - - - - 8< - - - - - - - - - - - -
```

I have come across one anomaly in RHEL6 (to be verified if occurs in RHEL7). It won't let you trace a PID or TID by CPU. I'm not suggesting this is a problem, just an observation.

```
# perf trace -p 4239 -C0
PID/TID switch overriding CPU
```

Another nice feature of trace is the **-S** option which provides a summary at the end of the trace

```
# perf trace -TS df
349851544.629 ( 0.000 ms):  ... [continued]: read()) = 1
349851544.658 ( 0.000 ms):  ... [continued]: execve()) = -2
349851544.666 ( 0.006 ms): execve(arg0: 140730400036510, arg1: 140730400056720, arg2: 40128560, arg3: 8, arg4: 7955998171588342573, arg5: 2000) = -2
349851544.671 ( 0.004 ms): execve(arg0: 140730400036516, arg1: 140730400056720, arg2: 40128560, arg3: 8, arg4: 7955998171588342573, arg5: 2000) = -2
349851544.675 ( 0.003 ms): execve(arg0: 140730400036517, arg1: 140730400056720, arg2: 40128560, arg3: 8, arg4: 7955998171588342573, arg5: 2000) = -2
349851544.679 ( 0.003 ms): execve(arg0: 140730400036526, arg1: 140730400056720, arg2: 40128560, arg3: 8, arg4: 7955998171588342573, arg5: 2000) = -2
349851544.990 ( 0.310 ms): execve(arg0: 140730400036527, arg1: 140730400056720, arg2: 40128560, arg3: 8, arg4: 7955998171588342573, arg5: 2000) = 0
349851545.003 ( 0.001 ms): brk(                                                                   ) = 0xdb9000
349851545.015 ( 0.002 ms): mmap(len: 4096, prot: READ|WRITE, flags: PRIVATE|ANONYMOUS, fd: -1    ) = 0xea927000
349851545.024 ( 0.004 ms): access(filename: 0x39c7e1d280, mode: R                                 ) = -1 ENOENT No such file or directory
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31960904  36486912  47% /
tmpfs                  7977236       552   7976684   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
349851545.034 ( 0.003 ms): open(filename: 0x39c7e1b901, mode: 1                                   ) = 3
/dev/mapper/VolGroup-lv_home
                     385901524 315649340  50649988  87% /home
349851545.035 ( 0.001 ms): fstat(fd: 3, statbuf: 0x7ffe54698ab0                                   ) = 0
349851545.038 ( 0.002 ms): mmap(len: 91571, prot: READ, flags: PRIVATE, fd: 3                     ) = 0xea910000
349851545.040 ( 0.001 ms): close(fd: 3                                                            ) = 0
349851545.051 ( 0.003 ms): open(filename: 0x7f95ea927640                                          ) = 3
349851545.053 ( 0.001 ms): read(fd: 3, buf: 0x7ffe54698c68, count: 832                            ) = 832
349851545.056 ( 0.001 ms): fstat(fd: 3, statbuf: 0x7ffe54698b10                                   ) = 0
349851545.060 ( 0.003 ms): mmap(addr: 0x39c8200000, len: 3750152, prot: EXEC|READ, flags: PRIVATE|DENYWRITE, fd: 3) = 0xc8200000
349851545.064 ( 0.003 ms): mprotect(start: 0x39c838a000, len: 2097152                             ) = 0
349851545.068 ( 0.003 ms): mmap(addr: 0x39c858a000, len: 20480, prot: READ|WRITE, flags: PRIVATE|DENYWRITE|FIXED, fd: 3, off: 1613824) = 0xc858a000
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
349851545.490 ( 0.001 ms): close(fd: 2                                                            ) = 0
349851545.492 ( 0.000 ms): exit_group(

 Summary of events:

 df (2001), 165 events, 95.4%, 0.000 msec

   syscall            calls      min       avg       max      stddev
                               (msec)    (msec)    (msec)        (%)
   --------------- -------- --------- --------- ---------     ------
   read                   6     0.000     0.001     0.002     27.70%
   write                  7     0.001     0.002     0.003     12.42%
   open                  12     0.002     0.003     0.005      8.82%
   close                  9     0.001     0.001     0.002     13.27%
   fstat                  8     0.001     0.001     0.001      2.91%
   mmap                  14     0.001     0.002     0.003      9.03%
   mprotect               3     0.002     0.002     0.003     12.55%
   munmap                 4     0.002     0.002     0.003     14.73%
   brk                    3     0.001     0.001     0.001     16.00%
   access                 1     0.004     0.004     0.004      0.00%
   execve                 6     0.000     0.054     0.310     93.98%
   statfs                 9     0.002     0.004     0.009     19.61%
   arch_prctl             1     0.001     0.001     0.001      0.00%

#
```

3.5.3 - Lock Tracing

This does work but requires the appropriate debug config options are enabled.

```
# perf lock record df
tracepoint lock:lock_acquire is not enabled. Are CONFIG_LOCKDEP and
CONFIG_LOCK_STAT enabled?
```

Rebooted into the debug kernel where the debug options are enabled.....

```
# perf lock record df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31877224  36570592  47% /
tmpfs                  7962884        84   7962800   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315649336  50649992  87% /home
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.530 MB perf.data (~23155 samples) ]
```

```
# perf lock script                    ← You can just type 'perf script'
          :14609 14609 [003]   913.085141: lock:lock_release: 0xffff88046876a080 &ctx->lock
          :14609 14609 [003]   913.085143: lock:lock_release: 0xffff8800387d8c50 &cpuctx_lock
          :14609 14609 [003]   913.085144: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
          :14609 14609 [003]   913.085145: lock:lock_acquire: 0xffffffff818cf060 read rcu_read_lock
          :14609 14609 [003]   913.085147: lock:lock_acquire: 0xffffffff818cf060 read rcu_read_lock
          :14609 14609 [003]   913.085148: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
              df 14609 [003]   913.085149: lock:lock_acquire: 0xffffffff818cf060 read rcu_read_lock
              df 14609 [003]   913.085150: lock:lock_acquire: 0xffffffff818cf060 read rcu_read_lock
              df 14609 [003]   913.085152: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
              df 14609 [003]   913.085153: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
              df 14609 [003]   913.085154: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
              df 14609 [003]   913.085156: lock:lock_acquire: 0xffffffff818cf060 read rcu_read_lock
              df 14609 [003]   913.085157: lock:lock_release: 0xffffffff818cf060 rcu_read_lock
              df 14609 [003]   913.085159: lock:lock_acquire: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085160: lock:lock_acquired: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085161: lock:lock_release: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085162: lock:lock_acquire: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085163: lock:lock_acquired: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085164: lock:lock_release: 0xffff88046ddc9d48 &newf->file_lock
              df 14609 [003]   913.085166: lock:lock_acquire: 0xffff8802e9484790 &sb->s_type->i_mutex_key
              df 14609 [003]   913.085167: lock:lock_acquired: 0xffff8802e9484790 &sb->s_type->i_mutex_key
              df 14609 [003]   913.085168: lock:lock_acquire: 0xffffffff82f8dfe8 &obj_hash[i].lock
              df 14609 [003]   913.085171: lock:lock_acquired: 0xffffffff82f8dfe8 &obj_hash[i].lock
               - - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

Now let's look at the report.

```
# perf lock report
               Name   acquired  contended   avg wait (ns) total wait (ns)   max wait (ns)   min wait (ns)

 &mm->page_table_...        296          0               0               0               0               0
          &fs->lock         37          0               0               0               0               0
 &newf->file_lock...         36          0               0               0               0               0
                key         35          0               0               0               0               0
      vfsmount_lock         30          0               0               0               0               0
     &tty->buf.lock         28          1            1586            1586            1586            1586
        dcache_lock         27          0               0               0               0               0
 &obj_hash[i].loc...         23          0               0               0               0               0
 &obj_hash[i].loc...         23          0               0               0               0               0
         files_lock         16          0               0               0               0               0
 &obj_hash[i].loc...         16          0               0               0               0               0
    &cpu_base->lock         16          0               0               0               0               0
    &zone->lru_lock         15          0               0               0               0               0
 &obj_hash[i].loc...         14          0               0               0               0               0
 &tty->output_loc...         14          0               0               0               0               0
        &base->lock         13          0               0               0               0               0
 &inode->i_data.i...         12          0               0               0               0               0
    &anon_vma->lock         10          0               0               0               0               0
         &ctx->lock         10          0               0               0               0               0
 &obj_hash[i].loc...          9          0               0               0               0               0
    &dentry->d_lock          9          0               0               0               0               0
         files_lock          9          0               0               0               0               0
 &inode->i_data.i...          8          0               0               0               0               0
         &rnp->lock          8          0               0               0               0               0
 &obj_hash[i].loc...          8          0               0               0               0               0
       &cpuctx_lock          8          0               0               0               0               0
 tty_ldisc_idle.l...          7          0               0               0               0               0
     tty_ldisc_lock          7          0               0               0               0               0
    &dentry->d_lock          7          0               0               0               0               0
    &dentry->d_lock          7          0               0               0               0               0
 &obj_hash[i].loc...          6          0               0               0               0               0
 &inode->i_data.i...          6          0               0               0               0               0
 &obj_hash[i].loc...          6          0               0               0               0               0
    &anon_vma->lock          5          0               0               0               0               0
    &dentry->d_lock          5          0               0               0               0               0
        sysctl_lock          5          0               0               0               0               0
    &anon_vma->lock          5          0               0               0               0               0
      policy_rwlock          4          0               0               0               0               0
    &dentry->d_lock          4          0               0               0               0               0
     &p->alloc_lock          4          0               0               0               0               0
    &anon_vma->lock          4          0               0               0               0               0
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
         &rnp->lock          1          0               0               0               0               0
 &child->perf_eve...          1          0               0               0               0               0
         &rnp->lock          1          0               0               0               0               0
         &rnp->lock          1          0               0               0               0               0
         &rnp->lock          1          0               0               0               0               0
    &dentry->d_lock          1          0               0               0               0               0
    &dentry->d_lock          1          0               0               0               0               0
         &rnp->lock          1          0               0               0               0               0
```

```
 &p->cred_guard_m...          0          0          0          0          0          0

=== output for debug===

bad: 7, total: 270
bad rate: 2.59 %
histogram of events caused bad sequence
    acquire: 2
   acquired: 0
  contended: 0
    release: 5
```

3.5.4 - Memory (mem, kmem(slab) & kvm) Tracing

perf can record and report on both **kmem** (kernel slab memory) and memory profiling. **perf kmem stat** is simply providing current totals. You cannot get anything more. So to be effective you could run this prior to a test then at the conclusion for the variances. Additionally you can run **kvm** which allows you to select information from the host or guest.

KMEM

```
# perf kmem stat

SUMMARY
=======
Total bytes requested: 512592
Total bytes allocated: 545848
Total bytes wasted on internal fragmentation: 33256
Internal fragmentation: 6.092539%
Cross CPU allocations: 308/1569
```

You can of course just start a recording process. You'll need to **Ctrl/C** to terminate or use sleep

```
# perf kmem record
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.751 MB perf.data (~32828 samples) ]

# perf kmem record sleep 10
[ perf record: Woken up 0 times to write data ]
[ perf record: Captured and wrote 1428.643 MB perf.data (~62418377 samples) ]

# perf kmem stat

SUMMARY
=======
Total bytes requested: 201010338
Total bytes allocated: 201104136
Total bytes wasted on internal fragmentation: 93798
Internal fragmentation: 0.046642%
Cross CPU allocations: 864/6981162
```

Reviewing the **kmem** data is simple enough:

```
# perf script
   perf 14760 [003]  1828.072076: kmem:kmem_cache_alloc: (jbd2_journal_start+0x89) call_site=ffffffffa03bf919
                                              ptr=0xffff8803508ee400 bytes_req=64 bytes_alloc=88 gfp_flags=GFP_NOFS
   perf 14760 [003]  1828.072083: kmem:kmem_cache_free: (jbd2_journal_stop+0x1cf) call_site=ffffffffa03bea0f
                                              ptr=0xffff8803508ee400
   init     1 [003]  1828.072546: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88040cd377f8
   init     1 [003]  1828.072548: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88043b798db0
```

```
    init     1 [003]  1828.072549: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032fa6a610
    init     1 [003]  1828.072550: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff8804565ecdb0
    init     1 [003]  1828.072551: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880455db47f8
    init     1 [003]  1828.072552: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88040cffb7f8
    init     1 [003]  1828.072553: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880315155610
    init     1 [003]  1828.072555: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880315155db0
    init     1 [003]  1828.072556: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032fa6a428
    init     1 [003]  1828.072557: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880341c68058
    init     1 [003]  1828.072559: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880455db4240
    init     1 [003]  1828.072560: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032fa6bbc8
    init     1 [003]  1828.072561: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880455db5428
    init     1 [003]  1828.072562: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff880341c527f8
    init     1 [003]  1828.072563: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032fa68610
    init     1 [003]  1828.072564: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88044bb2ddb0
    init     1 [003]  1828.072566: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88044bb2d9e0
    init     1 [003]  1828.072567: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88044f41b058
    init     1 [003]  1828.072568: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032fbd29e0
    init     1 [003]  1828.072569: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88044fa30bc8
    init     1 [003]  1828.072571: kmem:kmem_cache_free: (file_free_rcu+0x4d) call_site=ffffffff811b11fd ptr=0xffff88032c2ab240
```

```
# perf kmem stat --caller
Warning:
1 out of order events recorded.
------------------------------------------------------------------------------------------------------
 Callsite                           | Total_alloc/Per | Total_req/Per    | Hit      | Ping-pong | Frag
------------------------------------------------------------------------------------------------------
 kcalloc.clone.0+1c                 |      4608/32    |       864/6      |      144 |         0 | 81.250%
 vmap_batch+7b                      |      1664/32    |       416/8      |       52 |         0 | 75.000%
 proc_self_follow_link+7e           |       288/32    |       108/12     |        9 |         0 | 62.500%
 kstrdup+41                         |       288/32    |       138/15     |        9 |         0 | 52.083%
 selinux_file_alloc_security+46     |     19488/32    |      9744/16     |      609 |        18 | 50.000%
 i915_gem_object_get_pages_gtt+5a   |        96/32    |        48/16     |        3 |         0 | 50.000%
 __scm_send+3f9                     |     20480/4096  |     10360/2072   |        5 |         0 | 49.414%
 xhci_urb_enqueue+128               |    650240/512   |    335280/264    |     1270 |         0 | 48.438%
 context_struct_to_string+e7        |        64/64    |        38/38     |        1 |         0 | 40.625%
 memdup_user+2c                     |   1906688/2048  |   1139544/1224   |      931 |         0 | 40.234%
 kmemdup+29                         |       448/49    |       296/32     |        9 |         4 | 33.929%
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
 ext4_free_blocks+eb                |       136/136   |       136/136    |        1 |         0 |  0.000%
 ext4_mb_new_blocks+152             |       136/136   |       136/136    |        1 |         0 |  0.000%
 ext4_mb_new_inode_pa+5e            |       104/104   |       104/104    |        1 |         1 |  0.000%
 ext4_free_blocks+5ba               |        56/56    |        56/56     |        1 |         0 |  0.000%
------------------------------------------------------------------------------------------------------

SUMMARY (SLAB allocator)
========================
Total bytes requested: 111,420,948
Total bytes allocated: 116,063,632
Total bytes wasted on internal fragmentation: 4,642,684
Internal fragmentation: 4.000120%
Cross CPU allocations: 29,244/2,960,897
```

```
# perf kmem stat --alloc
Warning:
1 out of order events recorded.
------------------------------------------------------------------------------------------------------
 Alloc Ptr                          | Total_alloc/Per | Total_req/Per    | Hit      | Ping-pong | Frag
```

```
--------------------------------------------------------------------------------------------------------------
 0xffff8803e5d102c0                  |       2624/32     |        572/6      |        82 |           0 | 78.201%
 0xffff8804552fbac0                  |       3712/32     |       1136/9      |       116 |           1 | 69.397%
 0xffff8803e58c0ec0                  |        224/32     |         88/12     |         7 |           0 | 60.714%
 0xffff8803e440e8e0                  |        256/32     |        112/14     |         8 |           0 | 56.250%
 0xffff8803e58c0600                  |       2880/32     |       1368/15     |        90 |           0 | 52.500%
 0xffff8803e5d105c0                  |       3392/32     |       1618/15     |       106 |           5 | 52.300%
 0xffff8803e58c0dc0                  |        832/32     |        400/15     |        26 |           1 | 51.923%
 0xffff8803e5d10560                  |       1920/32     |        928/15     |        60 |           1 | 51.667%
 0xffff88044b6b5e80                  |       3488/32     |       1736/15     |       109 |           1 | 50.229%
 0xffff88044b6b5e60                  |        288/32     |        144/16     |         9 |           0 | 50.000%
 0xffff88044b5f3560                  |        256/32     |        128/16     |         8 |           1 | 50.000%
 0xffff88045401bac0                  |        256/32     |        128/16     |         8 |           0 | 50.000%
 0xffff88045401b980                  |        128/32     |         64/16     |         4 |           0 | 50.000%
 0xffff8803e5dd1580                  |         96/32     |         48/16     |         3 |           0 | 50.000%
 0xffff8803e5dd1d00                  |         96/32     |         48/16     |         3 |           0 | 50.000%
 0xffff8803e5d10960                  |       2208/32     |       1120/16     |        69 |           3 | 49.275%
- - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

MEM

perf mem on the other hand, is profiling memory. You can do so generically or for a specific command

```
# perf mem record df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31877544  36570272  47% /
tmpfs                  7962884       316   7962568   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315649332  50649996  87% /home
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.018 MB perf.data (~793 samples) ]
```

```
# perf script
      df  4852  1650.438634: cpu/mem-loads/pp:  ffffffff810a8967 sched_clock_cpu ([kernel.kallsyms])
      df  4852  1650.438637: cpu/mem-loads/pp:  ffffffff81125dd6 perf_output_copy ([kernel.kallsyms])
      df  4852  1650.438640: cpu/mem-loads/pp:  ffffffff8119c3fb __free_pipe_info ([kernel.kallsyms])
      df  4852  1650.438643: cpu/mem-loads/pp:  ffffffff8118a1fd __vma_adjust_trans_huge ([kernel.kallsyms])
      df  4852  1650.438671: cpu/mem-loads/pp:  ffffffff811ef01b load_elf_binary ([kernel.kallsyms])
      df  4852  1650.438921: cpu/mem-loads/pp:        39c822f321 read_alias_file (/lib64/libc-2.12.so)
```

```
# perf mem record -- sleep 15
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.019 MB perf.data (13 samples) ]
```

```
# perf script
   :6115  6115  6391.402169:          1 cpu/mem-stores/pp:  ffffffff8112759f perf_event_comm (/usr/lib/debug/lib/
   :6115  6115  6391.402170:          1 cpu/mem-loads/pp:  ffffffff811275a0 perf_event_comm (/usr/lib/debug/lib/m
   :6115  6115  6391.402172:          1 cpu/mem-stores/pp:  ffffffff8154d66a perf_event_nmi_handler (/usr/lib/deb
   :6115  6115  6391.402173:          1 cpu/mem-loads/pp:  ffffffff810f84c5 rcu_nmi_enter (/usr/lib/debug/lib/mod
   :6115  6115  6391.402174:         11 cpu/mem-stores/pp:  ffffffff8154d66a perf_event_nmi_handler (/usr/lib/deb
```

```
   :6115  6115  6391.402175:         14 cpu/mem-loads/pp:  ffffffff8154cc78 do_nmi (/usr/lib/debug/lib/modules/2.
   :6115  6115  6391.402176:        186 cpu/mem-stores/pp:  ffffffff8154cc78 do_nmi (/usr/lib/debug/lib/modules/2
   :6115  6115  6391.402177:       3322 cpu/mem-stores/pp:  ffffffff811265af perf_event_comm_output (/usr/lib/deb
   sleep  6115  6391.402180:        169 cpu/mem-loads/pp:  ffffffff8119b536 __fput (/usr/lib/debug/lib/modules/2.
   sleep  6115  6391.402191:      66989 cpu/mem-stores/pp:  ffffffff81160240 mmap_region (/usr/lib/debug/lib/modu
   sleep  6115  6391.402202:       1259 cpu/mem-loads/pp:  ffffffff8112cc21 perf_output_copy (/usr/lib/debug/lib/
   sleep  6115  6391.402420:       2905 cpu/mem-loads/pp:  ffffffff8112f6e3 filemap_fault (/usr/lib/debug/lib/mod
   sleep  6115  6406.420535:     218235 cpu/mem-stores/pp:  ffffffff8113e7f9 free_hot_cold_page (/usr/lib/debug/l
```

You might be wondering, what exactly does "**cpu/mem-loads/pp**" mean? This event is included in the **perf list** in the "**Kernel PMU event**" series. To find out what this means requires you read the "**Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3**" available freely from the Intel website:

https://software.intel.com/en-us/articles/intel-sdm

Specifically, review **Chapter 18 Performance Monitoring** and **Chapter 19 Performance-Monitoring Events** for the very descriptive explanation. Be advised, this is a manual written by technicians so it requires some prior CPU Processor knowledge to fully comprehend. This is also a good time to point out the this applies to anyone using the "**Kernel PMU event**" series of events in perf.

KVM

perf kvm is a little more complex. I found the help information to be almost useless but I finally got the idea of how to make it work. Here's a few of examples.

```
Usage: perf kvm [<options>] {top|record|report|diff|buildid-list|stat}

   -i, --input <file>    Input file name
   -o, --output <file>   Output file name
   -v, --verbose         be more verbose (show counter open errors, etc)
       --guest           Collect guest os data
       --guestkallsyms <file>
                         file saving guest os /proc/kallsyms
       --guestmodules <file>
                         file saving guest os /proc/modules
       --guestmount <directory>
                         guest mount directory under which every guest os instance has a subdir
       --guestvmlinux <file>
                         file saving guest os vmlinux
       --host            Collect host os data
```

From what I gathered, you basically use the **stat/record** just as you would normally set them up but precede them with the **kvm <options>**.

```
# perf kvm --host --guest stat df
Filesystem            1K-blocks     Used Available Use% Mounted on
/dev/mapper/rhel-root  47781076 12415464  35365612  26% /
devtmpfs                1910596        0   1910596   0% /dev
tmpfs                   1928728       88   1928640   1% /dev/shm
tmpfs                   1928728     8972   1919756   1% /run
tmpfs                   1928728        0   1928728   0% /sys/fs/cgroup
/dev/vda1                508588   329812    178776  65% /boot
tmpfs                    385748       12    385736   1% /run/user/0

 Performance counter stats for 'df':

         5.235946      task-clock:HG (msec)      #    0.724 CPUs utilized
                2      context-switches:HG       #    0.382 K/sec
                0      cpu-migrations:HG         #    0.000 K/sec
              242      page-faults:HG            #    0.046 M/sec
   <not supported>      cycles:HG
   <not supported>      instructions:HG
   <not supported>      branches:HG
   <not supported>      branch-misses:HG

       0.007234388 seconds time elapsed
```

```
# perf kvm --guest stat -ag df
Filesystem            1K-blocks     Used Available Use% Mounted on
/dev/mapper/rhel-root  47781076 12415464  35365612  26% /
devtmpfs                1910596        0   1910596   0% /dev
tmpfs                   1928728       88   1928640   1% /dev/shm
tmpfs                   1928728     8972   1919756   1% /run
tmpfs                   1928728        0   1928728   0% /sys/fs/cgroup
/dev/vda1                508588   329812    178776  65% /boot
tmpfs                    385748       12    385736   1% /run/user/0

 Performance counter stats for 'system wide':

        29.038618      task-clock:G (msec)       #    4.003 CPUs utilized
               22      context-switches:G        #    0.758 K/sec
                2      cpu-migrations:G          #    0.069 K/sec
              276      page-faults:G             #    0.010 M/sec
   <not supported>      cycles:G
   <not supported>      instructions:G
   <not supported>      branches:G
   <not supported>      branch-misses:G

       0.007253710 seconds time elapsed
```

When you record anything in **perf kvm --host**, it gets written to a file **perf.data.kvm**, not **perf.data** so you need to specify that if you use **perf script**

```
# perf kvm --host record -e cycles -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.014 MB perf.data.kvm (11 samples) ]
```

```
# perf script -i perf.data.kvm
         sleep  4186  3167.543603:     250000 cpu-clock:HG:  ffffffff811fc2d7 __vm_enough_memory (/usr/lib/debug/lib/mo
         sleep  4186  3167.543851:     250000 cpu-clock:HG:      7f1b5c0ca1b6 dl_main (/usr/lib64/ld-2.17.so)
         sleep  4186  3167.544101:     250000 cpu-clock:HG:  ffffffff811227b1 lock_acquire (/usr/lib/debug/lib/modules/
         sleep  4186  3167.544351:     250000 cpu-clock:HG:  ffffffff8111c59c lock_is_held (/usr/lib/debug/lib/modules/
         sleep  4186  3167.544601:     250000 cpu-clock:HG:      7f1b5c0d13b3 do_lookup_x (/usr/lib64/ld-2.17.so)
         sleep  4186  3167.544852:     250000 cpu-clock:HG:  ffffffff817680f4 __do_page_fault (/usr/lib/debug/lib/modul
         sleep  4186  3167.545100:     250000 cpu-clock:HG:  ffffffff812613da link_path_walk (/usr/lib/debug/lib/module
         sleep  4186  3177.545972:     250000 cpu-clock:HG:  ffffffff81763003 _raw_spin_unlock_irq (/usr/lib/debug/lib/
         sleep  4186  3177.546123:     250000 cpu-clock:HG:      7f1b5bd80660 _IO_unsave_markers (/usr/lib64/libc-2.17.
         sleep  4186  3177.546374:     250000 cpu-clock:HG:  ffffffff811227b1 lock_acquire (/usr/lib/debug/lib/modules/
         sleep  4186  3177.546627:     250000 cpu-clock:HG:  ffffffff81762f9b _raw_spin_unlock_irqrestore (/usr/lib/deb
(END)
```

```
# perf kvm --host report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 11  of event 'cpu-clock:HG'
# Event count (approx.): 2750000
#
# Overhead  Command  Shared Object     Symbol
# ........  .......  ................  ..............................
#
    18.18%  sleep    [kernel.vmlinux]  [k] lock_acquire
     9.09%  sleep    [kernel.vmlinux]  [k] __do_page_fault
     9.09%  sleep    [kernel.vmlinux]  [k] __vm_enough_memory
     9.09%  sleep    [kernel.vmlinux]  [k] _raw_spin_unlock_irq
     9.09%  sleep    [kernel.vmlinux]  [k] _raw_spin_unlock_irqrestore
     9.09%  sleep    [kernel.vmlinux]  [k] link_path_walk
     9.09%  sleep    [kernel.vmlinux]  [k] lock_is_held
     9.09%  sleep    ld-2.17.so        [.] dl_main
     9.09%  sleep    ld-2.17.so        [.] do_lookup_x
     9.09%  sleep    libc-2.17.so      [.] _IO_unsave_markers


#
# (Tip: To record every process run by an user: perf record -u <user>)
#
```

When you record anything with **perf kvm --guest**, the output is save in a file **perf.data.guest** not **perf.data**, nor **perf.data.kvm**. So again, you need to specifically identify the input file when displaying.

```
# perf kvm --guest record -e cycles -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.014 MB perf.data.guest (9 samples) ]
```

```
# perf script -i perf.data.guest
         sleep  4210  3321.053326:     250000 cpu-clock:G:  ffffffff811227b1 lock_acquire ([kernel.kallsyms])
         sleep  4210  3321.053574:     250000 cpu-clock:G:  ffffffff811227b1 lock_acquire ([kernel.kallsyms])
         sleep  4210  3321.053822:     250000 cpu-clock:G:  ffffffff811f8960 handle_mm_fault ([kernel.kallsyms])
         sleep  4210  3321.054078:     250000 cpu-clock:G:  ffffffff8111c5cb lock_is_held ([kernel.kallsyms])
         sleep  4210  3321.054324:     250000 cpu-clock:G:  ffffffff811227b1 lock_acquire ([kernel.kallsyms])
```

```
          sleep  4210  3321.054571:     250000 cpu-clock:G:  ffffffff8116efe8 rcu_is_watching ([kernel.kallsyms])
          sleep  4210  3321.054824:     250000 cpu-clock:G:  ffffffff81122ce9 lock_release ([kernel.kallsyms])
          sleep  4210  3331.055162:     250000 cpu-clock:G:  ffffffff81122ce9 lock_release ([kernel.kallsyms])
          sleep  4210  3331.055408:     250000 cpu-clock:G:  ffffffff811cc560 free_hot_cold_page ([kernel.kallsyms])
```

```
# perf kvm --guest report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 9  of event 'cpu-clock:G'
# Event count (approx.): 2250000
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......
#


#
# (Tip: List events using substring match: perf list <keyword>)
#
```

3.5.5 - Testing perf is installed ok

There may well be a time that you want to verify **perf** on a customer system. **perf test** will tell you if everything is installed ok although be careful as there are some features that are only relevant to RHEL7. You can also gain considerably more output using **-v** (verbose)

```
# perf test
 1: vmlinux symtab matches kallsyms                        : FAILED!
 2: detect open syscall event                              : Ok
 3: detect open syscall event on all cpus                  : Ok
 4: read samples using the mmap interface                  :  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
 Ok
 5: parse events tests                                     :  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: [scsi:scsi_dispatch_cmd_start] function ftrace_print_hex_seq not defined
  Warning: [scsi:scsi_dispatch_cmd_error] function ftrace_print_hex_seq not defined
  Warning: [scsi:scsi_dispatch_cmd_done] function ftrace_print_hex_seq not defined
  Warning: [scsi:scsi_dispatch_cmd_timeout] function ftrace_print_hex_seq not defined
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
```

```
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
 Ok
 6: Validate PERF_RECORD_* events & perf_sample fields     : Ok
 7: Test perf pmu format parsing                           : Ok
 8: Test dso data read                                     : Ok
 9: Test dso data cache                                    : Ok
10: Test dso data reopen                                   : Ok
11: roundtrip evsel->name check                            : Ok
12: Check parsing of sched tracepoints fields              : Ok
13: Generate and check syscalls:sys_enter_open event fields: Ok
14: struct perf_event_attr setup                           : Ok
15: Test matching and linking multiple hists               : Ok
16: Try 'use perf' in python, checking link problems       : FAILED!
17: Test number of exit event of a simple workload         : Ok
18: Test software clock events have valid period values    : Ok
19: Test object code reading                               : FAILED!
20: Test sample parsing                                    : Ok
21: Test using a dummy software event to keep tracking     : Ok
22: Test parsing with no sample_id_all bit set             : Ok
23: Test dwarf unwind                                      : FAILED!
24: Test filtering hist entries                            : Ok
25: Test mmap thread lookup                                : Ok
26: Test thread mg sharing                                 : Ok
27: Test output sorting of hist entries                    : Ok
28: Test cumulation of child hist entries                  : Ok
#
```

3.5.6 - perf benchmarking

Perf has a **bench** command. It's actually used for benchmarking and has a small number of options. An example where this could prove useful is where a customer has multiple servers with the same hardware and OS config (quite common) and complaining that one server runs much slower than another/rest of the servers.

```
# perf bench futex all
# Running futex/hash benchmark...
Run summary [PID 25775]: 8 threads, each operating on 1024 [private] futexes for 10
secs.

[thread  0] futexes: 0x16ad440 ... 0x16ae43c [ 4527923 ops/sec ]
[thread  1] futexes: 0x16ae750 ... 0x16af74c [ 4666470 ops/sec ]
[thread  2] futexes: 0x16af9d0 ... 0x16b09cc [ 4368896 ops/sec ]
[thread  3] futexes: 0x16b0c50 ... 0x16b1c4c [ 4661350 ops/sec ]
[thread  4] futexes: 0x16b1ed0 ... 0x16b2ecc [ 4321689 ops/sec ]
[thread  5] futexes: 0x16b3150 ... 0x16b414c [ 4517888 ops/sec ]
[thread  6] futexes: 0x16b43d0 ... 0x16b53cc [ 4929536 ops/sec ]
[thread  7] futexes: 0x16b5650 ... 0x16b664c [ 5270732 ops/sec ]

Averaged 4658060 operations/sec (+- 2.37%), total secs = 10

# Running futex/wake benchmark...
Run summary [PID 25775]: blocking on 8 threads (at futex 0x7ed384), waking up 1 at a
time.

[Run 1]: Wokeup 8 of 8 threads in 0.0210 ms
[Run 2]: Wokeup 8 of 8 threads in 0.0270 ms
[Run 3]: Wokeup 8 of 8 threads in 0.0290 ms
[Run 4]: Wokeup 8 of 8 threads in 0.0230 ms
[Run 5]: Wokeup 8 of 8 threads in 0.0290 ms
[Run 6]: Wokeup 8 of 8 threads in 0.0200 ms
[Run 7]: Wokeup 8 of 8 threads in 0.0140 ms
[Run 8]: Wokeup 8 of 8 threads in 0.0550 ms
[Run 9]: Wokeup 8 of 8 threads in 0.0300 ms
[Run 10]: Wokeup 8 of 8 threads in 0.0190 ms
Wokeup 8 of 8 threads in 0.0267 ms (+-13.30%)

# Running futex/requeue benchmark...
Run summary [PID 25775]: Requeuing 8 threads (from 0x7ed4e4 to 0x7ed4e8), 1 at a
time.

[Run 1]: Requeued 8 of 8 threads in 0.0030 ms
[Run 2]: Requeued 8 of 8 threads in 0.0120 ms
[Run 3]: Requeued 8 of 8 threads in 0.0120 ms
[Run 4]: Requeued 8 of 8 threads in 0.0110 ms
```

```
[Run 5]: Requeued 8 of 8 threads in 0.0120 ms
[Run 6]: Requeued 8 of 8 threads in 0.0110 ms
[Run 7]: Requeued 8 of 8 threads in 0.0030 ms
[Run 8]: Requeued 8 of 8 threads in 0.0110 ms
[Run 9]: Requeued 8 of 8 threads in 0.0130 ms
[Run 10]: Requeued 8 of 8 threads in 0.0040 ms
Requeued 8 of 8 threads in 0.0092 ms (+-14.11%)
```

What can you benchmark????

```
# perf bench
Usage:
      perf bench [<common options>] <collection> <benchmark> [<options>]

        # List of all available benchmark collections:

         sched: Scheduler and IPC benchmarks
           mem: Memory access benchmarks
         futex: Futex stressing benchmarks
           all: All benchmarks
```

So what are the actual options??

`perf bench sched messaging`	Benchmark for scheduling and IPC
`"  pipe`	Benchmark for pipe() between two processes
`"  all`	Test all scheduler benchmarks
`perf bench mem memcpy`	Benchmark for memcpy()
`"  all`	Test all memory benchmarks
`perf bench numa mem`	Benchmark for NUMA workloads
`"  all`	Run all NUMA benchmarks
`perf bench futex hash`	Benchmark for futex hash table
`"  wake`	Benchmark for futex wake calls
`"  requeue`	Benchmark for futex requeue calls
`"  all`	Test all futex benchmarks
`perf bench all`	Test all sched, mem and futex benchmarks

You DO need to check the man page for **`perf-bench`**. There are options available specific to each subset test

3.5.7 - Using perf top

Perf top is a system profiling tool and displays performance counters in real time. You can select CPU's using **-C** or select system wide (**-a**) for all CPU's. You can select an event(s) you want to monitor (**-e <event>**) and you can set the delay between refreshes (**-d <seconds>**). Like many other perf commands, you can select a PID (**-p**) or list of PIDs, a TID (**-t**) or list of TIDs so it can be very specific if you so wish. You can **top** the whole system or specific events or set/series of events.

```
# perf top -e sched:*
```

....produces a real time, incrementing display similar to this:

```
Available samples
0 sched:sched_kthread_stop                                                    ◆
0 sched:sched_kthread_stop_ret                                                ▒
0 sched:sched_wait_task                                                       ▒
32K sched:sched_wakeup                                                        ▒
16 sched:sched_wakeup_new                                                     ▒
59K sched:sched_switch                                                        ▒
1K sched:sched_migrate_task                                                   ▒
14 sched:sched_process_free                                                   ▒
14 sched:sched_process_exit                                                   ▒
7 sched:sched_process_wait                                                    ▒
16 sched:sched_process_fork                                                   ▒
35K sched:sched_stat_wait                                                     ▒
31K sched:sched_stat_sleep                                                    ▒
54 sched:sched_stat_iowait                                                    ▒
147 sched:sched_stat_blocked                                                  ▒
53K sched:sched_stat_runtime                                                  ▒
0 sched:sched_process_hang                                                    ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
                                                                              ▒
ESC: exit, ENTER|->: Browse histograms
```

3.5.8 - 'perf script <script>' – Some nice pre-built scripts and how to use them

perf continues its amazing run of features with this little tidbit. perf script has some built in monitoring features which some folks will find amazingly powerful and easy. How do you find out what they are?

```
# perf script --list
List of available trace scripts:
  rwtop [interval]                     system-wide r/w top
  rw-by-pid                            system-wide r/w activity
  rw-by-file <comm>                    r/w activity for a program, by file
  wakeup-latency                       system-wide min/max/avg wakeup latency
  failed-syscalls [comm]               system-wide failed syscalls
  net_dropmonitor                      display a table of dropped frames
  event_analyzing_sample               analyze all perf samples
  sctop [comm] [interval]              syscall top
  netdev-times [tx] [rx] [dev=] [debug] display a process of packet and processing time
  sched-migration                      sched migration overview
  futex-contention                     futext contention measurement
  syscall-counts-by-pid [comm]         system-wide syscall counts, by pid
  syscall-counts [comm]                system-wide syscall counts
  failed-syscalls-by-pid [comm]        system-wide failed syscalls, by pid
```

How do they work? Let's look at a couple of examples.

```
# perf script netdev-times
Install the audit-libs-python package to get syscall names
```

As the message above states:
yum install audit-libs-python
removes this message

```
^C
20171.134331sec cpu=0
  irq_entry(+0.000msec irq=36:eth1)
         |
  softirq_entry(+0.005msec)
         |
  napi_poll_exit(+0.009msec eth1)

20173.136793sec cpu=0
  irq_entry(+0.000msec irq=36:eth1)
         |
  softirq_entry(+0.013msec)
         |
  napi_poll_exit(+0.020msec eth1)

20175.139187sec cpu=0
```

```
  irq_entry(+0.000msec irq=36:eth1)
         |
  softirq_entry(+0.013msec)
         |
  napi_poll_exit(+0.020msec eth1)

   dev    len      Qdisc               netdevice             free
  wlan1    66  20170.908813sec        0.031msec             0.340msec
   tun0    68  20171.497179sec        0.014msec             0.051msec
   tun0    68  20171.497208sec        0.003msec             0.156msec
  wlan1   158  20171.497328sec        0.029msec             0.767msec
  wlan1  1414  20171.497423sec        0.025msec             0.680msec
  wlan1   158  20171.497450sec        0.008msec             0.673msec
  wlan1   179  20171.497463sec        0.017msec             0.652msec
  wlan1    66  20171.792215sec        0.027msec             1.865msec
  wlan1    66  20171.793564sec        0.021msec             0.893msec
  wlan1    66  20171.793623sec        0.007msec             0.850msec
  wlan1    66  20171.793645sec        0.006msec             0.832msec
   tun0    68  20173.515108sec        0.014msec             0.056msec
   tun0    68  20173.515139sec        0.003msec             0.266msec
  wlan1  1414  20173.515289sec        0.038msec             1.556msec
  wlan1   163  20173.515339sec        0.007msec             1.540msec
  wlan1   158  20173.515375sec        0.023msec             1.492msec
  wlan1   158  20173.515446sec        0.008msec             1.439msec
  wlan1    66  20173.746119sec        0.028msec             0.297msec
  wlan1    66  20173.746729sec        0.020msec             0.740msec
  wlan1    66  20173.747150sec        0.017msec             0.325msec
```

What about those netdev-times options it shows??? If you try to run **netdev-times rx** or any other option it fails.

```
# perf script netdev-times tx
Workload failed: No such file or directory
   dev    len      Qdisc               netdevice             free
#
```

Took me a little while to figure this out. It's not very obvious because the error message is so cryptic and gives no clue as to a cause. Here's how you use those options. First you record the data and then you use the reporting feature which then allows you to use the scripts embedded options:

```
# perf script record netdev-times
^C[ perf record: Woken up 10 times to write data ]
[ perf record: Captured and wrote 3.563 MB perf.data (36217 samples) ]

#
```

```
# perf script report netdev-times dev=tun0
   dev    len      Qdisc               netdevice             free
```

```
   tun0    52  53749.431305sec        0.005msec             0.010msec
   tun0    52  53749.433771sec        0.002msec             0.007msec
   tun0  1329  53749.880695sec        0.005msec             0.009msec
   tun0  1329  53749.880704sec        0.001msec             0.071msec
   tun0   183  53749.880708sec        0.001msec             0.084msec
#
```

Thios issue with options is not uniform across all scripts. Some scripts require command line options to work.

Another script example:

```
# perf script syscall-counts df
Press control+C to stop and show the summary
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  32605412  35842404  48% /
tmpfs                  7977236       320   7976916   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315972304  50327024  87% /home

syscall events:

event                                          count
----------------------------------------  -----------
mmap                                              14
open                                              12
statfs                                             9
close                                              9
fstat                                              8
write                                              7
read                                               5
munmap                                             4
brk                                                3
mprotect                                           3
exit_group                                         1
arch_prctl                                         1
access                                             1
```

And where are these scripts???? Perhaps you want to gain more information as to what some of the data is that is presented or perhaps even use/modify/correct a script. There are two places.

```
/usr/libexec/perf-core/scripts/python
/usr/libexec/perf-core/scripts/perl
```

```
# ll /usr/libexec/perf-core/scripts/python/
```

```
total 272
drwxr-xr-x. 2 root root  4096 Dec 12  2016 bin
-rw-r--r--. 1 root root  2539 Oct 26  2016 check-perf-trace.py
-rw-r--r--. 2 root root  3027 Oct 26  2016 check-perf-trace.pyc
-rw-r--r--. 2 root root  3027 Oct 26  2016 check-perf-trace.pyo
-rw-r--r--. 1 root root  7393 Oct 26  2016 event_analyzing_sample.py
-rw-r--r--. 2 root root  6069 Oct 26  2016 event_analyzing_sample.pyc
-rw-r--r--. 2 root root  6069 Oct 26  2016 event_analyzing_sample.pyo
-rw-r--r--. 1 root root 16128 Oct 26  2016 export-to-postgresql.py
-rw-r--r--. 2 root root 17352 Oct 26  2016 export-to-postgresql.pyc
-rw-r--r--. 2 root root 17352 Oct 26  2016 export-to-postgresql.pyo
-rw-r--r--. 1 root root  2229 Oct 26  2016 failed-syscalls-by-pid.py
-rw-r--r--. 2 root root  3039 Oct 26  2016 failed-syscalls-by-pid.pyc
-rw-r--r--. 2 root root  3039 Oct 26  2016 failed-syscalls-by-pid.pyo
-rw-r--r--. 1 root root  1508 Oct 26  2016 futex-contention.py
-rw-r--r--. 2 root root  1845 Oct 26  2016 futex-contention.pyc
-rw-r--r--. 2 root root  1845 Oct 26  2016 futex-contention.pyo
-rw-r--r--. 1 root root 15191 Oct 26  2016 netdev-times.py
-rw-r--r--. 2 root root 15827 Oct 26  2016 netdev-times.pyc
-rw-r--r--. 2 root root 15827 Oct 26  2016 netdev-times.pyo
-rw-r--r--. 1 root root  1749 Oct 26  2016 net_dropmonitor.py
-rw-r--r--. 2 root root  2484 Oct 26  2016 net_dropmonitor.pyc
-rw-r--r--. 2 root root  2484 Oct 26  2016 net_dropmonitor.pyo
drwxr-xr-x. 3 root root  4096 Oct 26  2016 Perf-Trace-Util
-rw-r--r--. 1 root root 11965 Oct 26  2016 sched-migration.py
-rw-r--r--. 2 root root 19920 Oct 26  2016 sched-migration.pyc
-rw-r--r--. 2 root root 19920 Oct 26  2016 sched-migration.pyo
-rw-r--r--. 1 root root  2098 Oct 26  2016 sctop.py
-rw-r--r--. 2 root root  2685 Oct 26  2016 sctop.pyc
-rw-r--r--. 2 root root  2685 Oct 26  2016 sctop.pyo
-rw-r--r--. 1 root root  2101 Oct 26  2016 syscall-counts-by-pid.py
-rw-r--r--. 2 root root  2944 Oct 26  2016 syscall-counts-by-pid.pyc
-rw-r--r--. 2 root root  2944 Oct 26  2016 syscall-counts-by-pid.pyo
-rw-r--r--. 1 root root  1696 Oct 26  2016 syscall-counts.py
-rw-r--r--. 2 root root  2553 Oct 26  2016 syscall-counts.pyc
-rw-r--r--. 2 root root  2553 Oct 26  2016 syscall-counts.pyo
```

```
# ll /usr/libexec/perf-core/scripts/perl
total 40
drwxr-xr-x. 2 root root 4096 Dec 12  2016 bin
-rw-r--r--. 1 root root 2626 Oct 26  2016 check-perf-trace.pl
-rw-r--r--. 1 root root 1153 Oct 26  2016 failed-syscalls.pl
drwxr-xr-x. 3 root root 4096 Oct 26  2016 Perf-Trace-Util
-rw-r--r--. 1 root root 2897 Oct 26  2016 rw-by-file.pl
-rw-r--r--. 1 root root 5186 Oct 26  2016 rw-by-pid.pl
-rw-r--r--. 1 root root 4550 Oct 26  2016 rwtop.pl
-rw-r--r--. 1 root root 2784 Oct 26  2016 wakeup-latency.pl
```

What about doing something more fancy? Here's a simple idea of how to use the scripts and heatmaps combined. First I ran the futex-contention script, recording the data to the perf.data file:

```
# perf script record futex-contention
^C[ perf record: Woken up 8 times to write data ]
[ perf record: Captured and wrote 3.671 MB perf.data (28253 samples) ]
```

Next I displayed the information with selected headings. The reason for this is that I intend to pass this output through an **awk** script and the first field is the command which in some cases is multiple fields space separated. Rather than make the **awk** more complex than it already is, I focused on the four fields I was primarily interested (although I ended up only using 3 of them)

```
# perf script -F pid,cpu,time,event | head
 4559 [000] 57175.418873: syscalls:sys_exit_futex:
 4559 [000] 57175.418885: syscalls:sys_enter_futex:
 4559 [000] 57175.418886: syscalls:sys_exit_futex:
 4559 [000] 57175.418898: syscalls:sys_enter_futex:
 4559 [000] 57175.423978: syscalls:sys_exit_futex:
 4559 [000] 57175.423987: syscalls:sys_enter_futex:
 4559 [000] 57175.423988: syscalls:sys_exit_futex:
 4559 [000] 57175.423999: syscalls:sys_enter_futex:
 4559 [000] 57175.425070: syscalls:sys_exit_futex:
 4559 [000] 57175.425073: syscalls:sys_enter_futex:
```

```
# perf script -F pid,cpu,time,event| awk '{ gsub(/:/, "") } $4 ~ /enter/ \
       { ts[$1] = $3 } $4 ~ /exit/ { if (itime = ts[$1]) { printf "%.f %.f\n", \
       $3 * 1000000, ($3 - itime) * 1000000; ts[$1] = 0 } }' > futex_lat.out
```

```
# head futex_lat.out
56119610607 1
56119611659 1051
56119611663 0
56119683820 72148
56119683833 2
56119711937 28094
56119711946 1
56119758116 1
56119794123 82167
56119794133 1
```

And now let's create a heatmap. The maxlat value was discovered after a few very quick map creations and realizing the need for a larger value to display all the samples

```
# trace2heatmap.pl --unitstime=us --unitslat=us --maxlat=250000 futex_lat.out > futex_lat.svg
```

Latency Heat Map

time 12s, range 0-5000us, count: 429, pct: 91%, acc: 429,
Time

3.6 - Other oddities of perf

3.6.1 - How does the perf probe into the kernel work?

We know that **stap** uses a rather intrusive mechanism of using **int3** and the debug fault handling mechansism for its probe handling. That in itself is only a small insight into the overhead of **stap**. **perf** on the other hand uses a far less invasive mechanism which also helps to make it far more performant with considerably less overhead as there is no "faulting" involved. It tries to use a **jmp** to replace what **/arch/x86/kernel/kprobes.c** calls a 'boostable' instruction. There are some instructions that cannot be replaced (boosted). When that happens, **perf** drops back to using the **int3** mechanism. Following are 2 examples of where a jmp was used, and also where an **int3** was used. I sould also note that I encountered a "bug" on rhel6 where after a series of training I provided on perf, I got my system into a state where it ALWAYS used int3. After a reboot it went back to normal. Still investigating how/why that happened.

BOOSTING with jmp

1. Let's check a function we are going to **perf** before we start

```
crash> dis __do_page_fault
0xffffffff8104f010 <__do_page_fault>:         push   %rbp
0xffffffff8104f011 <__do_page_fault+0x1>:     mov    %rsp,%rbp
0xffffffff8104f014 <__do_page_fault+0x4>:     sub    $0x110,%rsp
0xffffffff8104f01b <__do_page_fault+0xb>:     mov    %rbx,-0x28(%rbp)
0xffffffff8104f01f <__do_page_fault+0xf>:     mov    %r12,-0x20(%rbp)
0xffffffff8104f023 <__do_page_fault+0x13>:    mov    %r13,-0x18(%rbp)
0xffffffff8104f027 <__do_page_fault+0x17>:    mov    %r14,-0x10(%rbp)
0xffffffff8104f02b <__do_page_fault+0x1b>:    mov    %r15,-0x8(%rbp)
0xffffffff8104f02f <__do_page_fault+0x1f>:    nopl   0x0(%rax,%rax,1)
0xffffffff8104f034 <__do_page_fault+0x24>:    mov    %gs:0xbc00,%r15
0xffffffff8104f03d <__do_page_fault+0x2d>:    mov    %rdi,-0x100(%rbp)
0xffffffff8104f044 <__do_page_fault+0x34>:    mov    %rdx,-0xf8(%rbp)
0xffffffff8104f04b <__do_page_fault+0x3b>:    mov    %rsi,%r12
0xffffffff8104f04e <__do_page_fault+0x3e>:    mov    0x480(%r15),%r13
0xffffffff8104f055 <__do_page_fault+0x45>:    lea    0x68(%r13),%rax
0xffffffff8104f059 <__do_page_fault+0x49>:    mov    %rax,-0xf0(%rbp)
0xffffffff8104f060 <__do_page_fault+0x50>:    prefetcht0 (%rax)
0xffffffff8104f063 <__do_page_fault+0x53>:    movabs $0x7ffffffefff,%rax
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

2. Add a probe (this DOES NOT apply the hook at this point)

```
# perf probe --add __do_page_fault
Added new event:
  probe:__do_page_fault (on __do_page_fault)

You can now use it in all perf tools, such as:
```

```
perf record -e probe:__do_page_fault -aR sleep 1
```

3. Now start a perf recording session

```
# perf record -e probe:__do_page_fault -ag
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.858 MB perf.data (~37506 samples) ]
```

4. Check the function we have probed for perf

```
crash> dis __do_page_fault
0xffffffff8104f010 <__do_page_fault>:          jmpq   0xffffffffa0042000
0xffffffff8104f015 <__do_page_fault+0x5>:      sub    $0x110,%esp
0xffffffff8104f01b <__do_page_fault+0xb>:      mov    %rbx,-0x28(%rbp)
0xffffffff8104f01f <__do_page_fault+0xf>:      mov    %r12,-0x20(%rbp)
0xffffffff8104f023 <__do_page_fault+0x13>:     mov    %r13,-0x18(%rbp)
0xffffffff8104f027 <__do_page_fault+0x17>:     mov    %r14,-0x10(%rbp)
0xffffffff8104f02b <__do_page_fault+0x1b>:     mov    %r15,-0x8(%rbp)
0xffffffff8104f02f <__do_page_fault+0x1f>:     nopl   0x0(%rax,%rax,1)
0xffffffff8104f034 <__do_page_fault+0x24>:     mov    %gs:0xbc00,%r15
0xffffffff8104f03d <__do_page_fault+0x2d>:     mov    %rdi,-0x100(%rbp)
0xffffffff8104f044 <__do_page_fault+0x34>:     mov    %rdx,-0xf8(%rbp)
0xffffffff8104f04b <__do_page_fault+0x3b>:     mov    %rsi,%r12
0xffffffff8104f04e <__do_page_fault+0x3e>:     mov    0x480(%r15),%r13
0xffffffff8104f055 <__do_page_fault+0x45>:     lea    0x68(%r13),%rax
0xffffffff8104f059 <__do_page_fault+0x49>:     mov    %rax,-0xf0(%rbp)
0xffffffff8104f060 <__do_page_fault+0x50>:     prefetcht0 (%rax)
0xffffffff8104f063 <__do_page_fault+0x53>:     movabs $0x7ffffffefff,%rax
- - - - - - - - - - - - - - 8< - - - - - - - - - - - - - -
```

```
crash> dis 0xffffffffa0042000 60
dis: WARNING: ffffffffa0042000: no associated kernel symbol found
   0xffffffffa0042000:  push   %rsp
   0xffffffffa0042001:  pushfq
   0xffffffffa0042002:  sub    $0x18,%rsp
   0xffffffffa0042006:  push   %rdi        \
   0xffffffffa0042007:  push   %rsi         \
   0xffffffffa0042008:  push   %rdx          \
   0xffffffffa0042009:  push   %rcx           \
   0xffffffffa004200a:  push   %rax            \
   0xffffffffa004200b:  push   %r8              \
   0xffffffffa004200d:  push   %r9               \
   0xffffffffa004200f:  push   %r10               > Save all the regs
   0xffffffffa0042011:  push   %r11              /
   0xffffffffa0042013:  push   %rbx             /
   0xffffffffa0042014:  push   %rbp            /
   0xffffffffa0042015:  push   %r12           /
   0xffffffffa0042017:  push   %r13          /
   0xffffffffa0042019:  push   %r14         /
```

```
    0xffffffffa004201b:  push   %r15         /
    0xffffffffa004201d:  mov    %rsp,%rsi
    0xffffffffa0042020:  movabs $0xffff88041e4c86c0,%rdi
    0xffffffffa004202a:  callq  0xffffffff8153e980 <optimized_callback>
    0xffffffffa004202f:  mov    0x90(%rsp),%rdx
    0xffffffffa0042037:  mov    %rdx,0x98(%rsp)
    0xffffffffa004203f:  pop    %r15         \
    0xffffffffa0042041:  pop    %r14          \
    0xffffffffa0042043:  pop    %r13           \
    0xffffffffa0042045:  pop    %r12            \
    0xffffffffa0042047:  pop    %rbp             \
    0xffffffffa0042048:  pop    %rbx              \
    0xffffffffa0042049:  pop    %r11               \
    0xffffffffa004204b:  pop    %r10                > Restore all the regs
    0xffffffffa004204d:  pop    %r9                /
    0xffffffffa004204f:  pop    %r8               /
    0xffffffffa0042051:  pop    %rax             /
    0xffffffffa0042052:  pop    %rcx            /
    0xffffffffa0042053:  pop    %rdx           /
    0xffffffffa0042054:  pop    %rsi          /
    0xffffffffa0042055:  pop    %rdi         /
    0xffffffffa0042056:  add    $0x18,%rsp
    0xffffffffa004205a:  add    $0x8,%rsp
    0xffffffffa004205e:  popfq
    0xffffffffa004205f:  push   %rbp         \
    0xffffffffa0042060:  mov    %rsp,%rbp     >  These are the 3 instructions overwriten
    0xffffffffa0042063:  sub    $0x110,%rsp  /
    0xffffffffa004206a:  jmpq   0xffffffff8104f01b <__do_page_fault+0xb>  ← Back to our function
```

You may be wondering why we are executing the 3 instructions highlight that I note were overwritten. The jmpq that was inserted is a 5 byte instruction so it overwrites from `0xffffffff8104f010` to `0xffffffff8104f014` inclusive (`__do_page_fault` to `__do_page_fault+0x4`). Let's recap, the first 5 bytes of the function were:

```
0xffffffff8104f010 <__do_page_fault>:          push   %rbp
0xffffffff8104f011 <__do_page_fault+0x1>:      mov    %rsp,%rbp
0xffffffff8104f014 <__do_page_fault+0x4>:      sub    $0x110,%rsp
```

Now it should make sense.

Unable to boost with a jmp, must use an int3

`perf` can use the **`int3`** mechanism as noted. I installed a probe to **`tcp_sendmsg:215`**. Before I recorded, here's what the code looked like.

```
- - - - - - - - - - 8< - - - - - - - - - -
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1074
0xffffffff814b454b <tcp_sendmsg+1515>:  mov    0xe0(%r13),%eax
0xffffffff814b4552 <tcp_sendmsg+1522>:  sub    %eax,0xf8(%r12)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1075
0xffffffff814b455a <tcp_sendmsg+1530>:  mov    0xe0(%r13),%eax
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1066
```

```
0xffffffff814b4561 <tcp_sendmsg+1537>:  cmpq   $0x0,0xb0(%rdx)
0xffffffff814b4569 <tcp_sendmsg+1545>:  je     0xffffffff814b4573 <tcp_sendmsg+1555>
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1068
0xffffffff814b456b <tcp_sendmsg+1547>:  add    %eax,0xfc(%r12)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1076
0xffffffff814b4573 <tcp_sendmsg+1555>:  mov    %r13,%rdi
0xffffffff814b4576 <tcp_sendmsg+1558>:  mov    %r11d,-0x80(%rbp)
0xffffffff814b457a <tcp_sendmsg+1562>:  callq  0xffffffff8145e3d0 <__kfree_skb>
0xffffffff814b457f <tcp_sendmsg+1567>:  mov    -0x80(%rbp),%r11d
0xffffffff814b4583 <tcp_sendmsg+1571>:  mov    $0xfffffff2,%eax
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1146
0xffffffff814b4588 <tcp_sendmsg+1576>:  test   %r11d,%r11d
0xffffffff814b458b <tcp_sendmsg+1579>:  je     0xffffffff814b3fc1 <tcp_sendmsg+97>
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1130
0xffffffff814b4591 <tcp_sendmsg+1585>:  movzbl 0x4e7(%r12),%ecx
0xffffffff814b459a <tcp_sendmsg+1594>:  mov    -0x5c(%rbp),%edx
0xffffffff814b459d <tcp_sendmsg+1597>:  mov    %r12,%rdi
0xffffffff814b45a0 <tcp_sendmsg+1600>:  mov    -0x44(%rbp),%esi
0xffffffff814b45a3 <tcp_sendmsg+1603>:  mov    %r11d,-0x80(%rbp)
0xffffffff814b45a7 <tcp_sendmsg+1607>:  callq  0xffffffff814b3540 <tcp_push>
0xffffffff814b45ac <tcp_sendmsg+1612>:  mov    -0x80(%rbp),%r11d
0xffffffff814b45b0 <tcp_sendmsg+1616>:  jmpq   0xffffffff814b4690 <tcp_sendmsg+1840>
- - - - - - - - - - 8< - - - - - - - - - -
```

if you are wondering why cannot it boost the **test** with a **jmp**? Think about the first example previous where it was able to boost.... it excecuted the overwritten instructions in the boosted buffer than **jmp**'d back. However in this case, the **test** and **js** would be in the boosted buffer. Now you might be saying but that doesn't make a difference.. yes it does. The probe **jmp**'s are 5 byte instructions, they have a 4 byte offset from the current **rip** NOT absolute addresses so you would have to recalculate the address for the **je** to make it absolute (or relocate the address) as it's a relative address based on it's original **rip** NOT the **rip** it's now at in the boosted buffer.. After all, you don't know where the **perf** boosted buffer could end up? And don't forget, you'd also then have to make a second copy of the "boosted" code so you can set it back to what it was originally after you remove the probe. There's a lot more reasons why we sometimes have to use **int3** beyond this example.

```
- - - - - - - - - - 8< - - - - - - - - - -
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1074
0xffffffff814b454b <tcp_sendmsg+1515>:  mov    0xe0(%r13),%eax
0xffffffff814b4552 <tcp_sendmsg+1522>:  sub    %eax,0xf8(%r12)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1075
0xffffffff814b455a <tcp_sendmsg+1530>:  mov    0xe0(%r13),%eax
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1066
0xffffffff814b4561 <tcp_sendmsg+1537>:  cmpq   $0x0,0xb0(%rdx)
0xffffffff814b4569 <tcp_sendmsg+1545>:  je     0xffffffff814b4573 <tcp_sendmsg+1555>
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1068
0xffffffff814b456b <tcp_sendmsg+1547>:  add    %eax,0xfc(%r12)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/include/net/sock.h: 1076
0xffffffff814b4573 <tcp_sendmsg+1555>:  mov    %r13,%rdi
0xffffffff814b4576 <tcp_sendmsg+1558>:  mov    %r11d,-0x80(%rbp)
0xffffffff814b457a <tcp_sendmsg+1562>:  callq  0xffffffff8145e3d0 <__kfree_skb>
0xffffffff814b457f <tcp_sendmsg+1567>:  mov    -0x80(%rbp),%r11d
0xffffffff814b4583 <tcp_sendmsg+1571>:  mov    $0xfffffff2,%eax
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1146
```

```
0xffffffff814b4588 <tcp_sendmsg+1576>:  int3
0xffffffff814b4589 <tcp_sendmsg+1577>:  test   %ebx,%ebx
0xffffffff814b458b <tcp_sendmsg+1579>:  je     0xffffffff814b3fc1 <tcp_sendmsg+97>
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1130
0xffffffff814b4591 <tcp_sendmsg+1585>:  movzbl 0x4e7(%r12),%ecx
0xffffffff814b459a <tcp_sendmsg+1594>:  mov    -0x5c(%rbp),%edx
0xffffffff814b459d <tcp_sendmsg+1597>:  mov    %r12,%rdi
0xffffffff814b45a0 <tcp_sendmsg+1600>:  mov    -0x44(%rbp),%esi
0xffffffff814b45a3 <tcp_sendmsg+1603>:  mov    %r11d,-0x80(%rbp)
0xffffffff814b45a7 <tcp_sendmsg+1607>:  callq  0xffffffff814b3540 <tcp_push>
0xffffffff814b45ac <tcp_sendmsg+1612>:  mov    -0x80(%rbp),%r11d
0xffffffff814b45b0 <tcp_sendmsg+1616>:  jmpq   0xffffffff814b4690 <tcp_sendmsg+1840>
- - - - - - - - - - 8< - - - - - - - - - -
```

int3 or jmp? Who makes the decision?

If you're interested to know more, look in the source code module **/arch/x86/kernel/kprobes.c**. Here's a piece of that code that might help. There's a lot more to it than this, but for those who want to delve, it's a starting point.

```
static int __kprobes copy_optimized_instructions(u8 *dest, u8 *src)
{
        int len = 0, ret;

        while (len < RELATIVEJUMP_SIZE) {
                ret = __copy_instruction(dest + len, src + len);
                if (!ret || !can_boost(dest + len))
                        return -EINVAL;
                len += ret;
        }
        /* Check whether the address range is reserved */
        if (ftrace_text_reserved(src, src + len - 1) ||
            alternatives_text_reserved(src, src + len - 1))
                return -EBUSY;

        return len;
}
```

```
/*
 * Returns non-zero if opcode is boostable.
 * RIP relative instructions are adjusted at copying time in 64 bits mode
 */
static int __kprobes can_boost(kprobe_opcode_t *opcodes)
{
        kprobe_opcode_t opcode;
        kprobe_opcode_t *orig_opcodes = opcodes;

        if (search_exception_tables((unsigned long)opcodes))
```

```
                return 0;       /* Page fault may occur on this address. */

retry:
        if (opcodes - orig_opcodes > MAX_INSN_SIZE - 1)
                return 0;
        opcode = *(opcodes++);

        /* 2nd-byte opcode */
        if (opcode == 0x0f) {
                if (opcodes - orig_opcodes > MAX_INSN_SIZE - 1)
                        return 0;
                return test_bit(*opcodes,
                                (unsigned long *)twobyte_is_boostable);
        }

        switch (opcode & 0xf0) {
#ifdef CONFIG_X86_64
        case 0x40:
                goto retry; /* REX prefix is boostable */
#endif
        case 0x60:
                if (0x63 < opcode && opcode < 0x67)
                        goto retry; /* prefixes */
                /* can't boost Address-size override and bound */
                return (opcode != 0x62 && opcode != 0x67);
        case 0x70:
                return 0; /* can't boost conditional jump */
        case 0xc0:
                /* can't boost software-interruptions */
                return (0xc1 < opcode && opcode < 0xcc) || opcode == 0xcf;
        case 0xd0:
                /* can boost AA* and XLAT */
                return (opcode == 0xd4 || opcode == 0xd5 || opcode == 0xd7);
        case 0xe0:
                /* can boost in/out and absolute jmps */
                return ((opcode & 0x04) || opcode == 0xea);
        case 0xf0:
                if ((opcode & 0x0c) == 0 && opcode != 0xf1)
                        goto retry; /* lock/rep(ne) prefix */
                /* clear and set flags are boostable */
                return (opcode == 0xf5 || (0xf7 < opcode && opcode < 0xfe));
        default:
                /* segment override prefixes are boostable */
                if (opcode == 0x26 || opcode == 0x36 || opcode == 0x3e)
                        goto retry; /* prefixes */
                /* CS override prefix and call are not boostable */
                return (opcode != 0x2e && opcode != 0x9a);
        }
}
```

3.6.2 - Using the raw event register mechanism

You may have read/noticed that perf has the ability to monitor/report raw events. It is noted in some man pages as well as the perf listing:

```
# perf list | grep Raw
  rNNN                                               [Raw hardware event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier                  [Raw hardware event descriptor]
```

It is not easy to find the information on this. NNN is NOT an event, it's not a register number, it's not an MSR... It is actually 2 fields and should be better presented as **rMMEE**. Where **MM** is the mask and **EE** is the event. These are values referred to by Intel as PEBS events (Processor Event Based Sampling).

To use this feature you need the **Intel Architectures Software Developer's Manual**. The events are dotted throughout **Chapter 18** on **Performance Monitoring**. You might ask, why don't I just do all the hard work for you and list them for you? Well.... That's because Event and Umask identifiers are specific to the Intel Model Processor in your system. Here's an example from my system to show how to use the feature. The following table was an extract from the Intel manual Chapter 18 as I noted and is applicable to the general Intel CPU range. There are also a series of masks and events noted in the Intel manual for specific CPU models, hence why you need to refer to the manual which also explains each event in technical detail.

PEBS Generic Events for all Intel CPUs

	Hex Values	
	MM	**EE**
Event Name	UMask	Event Select
UnHalted Core Cycles	00	3C
UnHalted Reference Cycles	01	3C
LLC Reference	4F	2E
LLC Misses	41	2E
Branch Instruction Retired	00	C4
Branch Misses Retired	00	C5
ITLB_MISS_RETIRED	00	C9
INSTR_RETIRED.ANY_P	00	C0
X87_OPS_RETIRED.ANY	FE	C1
BR_INST_RETIRED.MISPRED	00	C5
SIMD_INST_RETIRED.ANY	1F	C7
MEM_LOAD_RETIRED.L1D_MISS	01	CB
MEM_LOAD_RETIRED.L1D_LINE_MISS	02	CB
MEM_LOAD_RETIRED.L2_MISS	04	CB
MEM_LOAD_RETIRED.L2_LINE_MISS	08	CB
MEM_LOAD_RETIRED.DTLB_MISS	10	CB

What does "**RETIRED**" mean in the table above?

For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction.

PEBS events for Sandy Bridge Intel CPU's

```
                                                   Hex Values
                                                    MM     EE
Event Name                                        UMask  Event Select
INST_RETIRED.PREC_DIST                              01     C0

UOPS_RETIRED.All                                    01     C2
            .Retire_Slots                           02     C2

BR_INST_RETIRED.Conditional                         01     C4
               .Near_Call                           02     C4
               .All_Branches                        04     C4
               .Near_Return                         08     C4
               .Near_Taken                          20     C4

BR_MISP_RETIRED.Conditional                         01     C5
               .Near_Call                           02     C5
               .All_Branches                        04     C5
               .Not_Taken                           10     C5
               .Taken                               20     C5

MEM_UOPS_RETIRED.STLB_MISS_LOADS                    11     D0
                .STLB_MISS_STORE                    12     D0
                .LOCK_LOADS                         21     D0
                .SPLIT_LOADS                        41     D0
                .SPLIT_STORES                       42     D0
                .ALL_LOADS                          81     D0
                .ALL_STORES                         82     D0

MEM_LOAD_UOPS_RETIRED.L1_Hit                        01     D1
                     .L2_Hit                        02     D1
                     .L3_Hit                        04     D1
                     .Hit_LFB                       40     D1

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_Miss             01     D2
                             .XSNP_Hit              02     D2
                             .XSNP_Hitm             04     D2
                             .XNSP_None             08     D2
```

The **r** value is determined by combining both fields into the **rMMEE** as follows (You can use this in :

```
# perf stat -e r00c0,rfec1,r00c5,r1fc7,r01cb,r003c -ag sleep 5

 Performance counter stats for 'system wide':

     1,758,830,854      r00c0                                                   [ 0.03%]
                 0      rfec1                                                   [ 0.02%]
```

```
        11,699,921      r00c5                                                        [ 0.02%]
         1,270,963      r1fc7                                                        [ 0.01%]
           158,988      r01cb                                                        [ 0.01%]
    29,935,011,405      r003c                                                        [ 0.00%]

       5.001728229 seconds time elapsed
```

Of course to understand what these **rMMEE** (PEBS) events are, you will need to look in the Intel manual for the detailed explanation.

Be advised, you can use PEBS events and static events (those available from **perf list**) together but watch what happens if you use the register that also matches a named item:

```
# perf stat -e r01d1,r02d1,r04d1,r40d1,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-
icache-load-misses  -ag sleep 5

 Performance counter stats for 'system wide':

     3,323,328,321      r01d1                                                   (0.00%)
        43,971,738      r02d1                                                   (0.00%)
     <not counted>      r04d1                     (0.00%)
     <not counted>      r40d1                     (0.00%)
   <not supported>      L1-dcache-load-misses
   <not supported>      L1-dcache-loads
   <not supported>      L1-dcache-stores
   <not supported>      L1-icache-load-misses

       5.000588957 seconds time elapsed
```

Now a re-re-run with the static event names:

```
# perf stat -e L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-icache-load-misses  -ag sleep 5

 Performance counter stats for 'system wide':

        87,488,139      L1-dcache-load-misses     #    5.87% of all L1-dcache hits    (0.00%)
     1,491,462,680      L1-dcache-loads                                                (0.00%)
     <not counted>      L1-dcache-stores           (0.00%)
       152,129,782      L1-icache-load-misses                                          (0.00%)

       5.000648282 seconds time elapsed
```

Let's look at using the PEBS events specifically for the correct model CPU. This test system was running an Intel(R) Core(TM) i7-4810MQ CPU @ 2.80GHz (Haswell) - 4th Generation i7

A review of the Intel manual previously noted and reviewing the table for 4th Generation CPU's shows:

```
Mask  Event       Event mask Mnemonic          Description
D0H   81H   MEM_UOPS_RETIRED.ALL_LOADS  All retired load uops.
D0H   82H   MEM_UOPS_RETIRED.ALL_STORES All retired store uops.
```

Another example... order matters:

```
# perf stat -e r81d0,r82d0,mem-loads,mem-stores -ag sleep 5

 Performance counter stats for 'system wide':

     5,621,436,706      r81d0           (All Loads)                    (0.00%)
     2,361,049,380      r82d0           (All Stores)                   (0.00%)
     <not counted>      mem-loads                                      (0.00%)
     <not counted>      mem-stores                                     (0.00%)

       5.000635452 seconds time elapsed
```

```
# perf stat -e mem-loads,mem-stores,r81d0,r82d0 -ag sleep 5

 Performance counter stats for 'system wide':

                 0      mem-loads          ← ????????                 (0.00%)
     5,858,557,652      mem-stores                                    (0.00%)
     <not counted>      r81d0                                         (0.00%)
     <not counted>      r82d0                                         (0.00%)

       5.000651261 seconds time elapsed
```

Notice here that I am using the correct MMEE names that intel documentation provides but the perf mem-loads (static event name) always returned 0. You may also notice that the Sandy Bridge values are also the same for the Intel 4th generation CPU chips. While true in this case, don't presume they are the same for all CPU models. Check the manual!!

Now here's a another quirk. If I use the register for stores and the event for stores it DOES work???? If you are wondering why the 2 values are not 100% the same, I have to say I suspect that this is a +/- variance due to the nanosecond timing. By that I mean, when perf events are being processed by the kernel it will be implementing them sequentially. That is, it is not programming all PEBS at exactly the same CPU cycle. It engages one, then the kernel code identifies the next in the list and engages that one and the next and so on. Clearly if you look at the totals involved you'll notice that the variation is small.

```
# perf stat -e r82d0,mem-stores -ag sleep 5

 Performance counter stats for 'system wide':

       863,144,880      r82d0                                         (100.00%)
       863,108,647      mem-stores

       5.000764041 seconds time elapsed
```

```
# perf stat -e r82d0,mem-stores -ag sleep 5

 Performance counter stats for 'system wide':
```

```
   1,024,216,283      r82d0                                                    (100.00%)
   1,024,184,501      mem-stores

     5.000598210 seconds time elapsed
```

In order to determine the accuracy of the variation in the results, I ran this 10 times:

```
# perf stat -e r82d0,mem-stores --repeat 10 -ag sleep 5

 Performance counter stats for 'system wide' (10 runs):

     1,032,614,525      r82d0                                ( +- 12.36% )  (100.00%)
     1,032,621,236      mem-stores                           ( +- 12.36% )

       5.000867962 seconds time elapsed                      ( +-  0.00% )
```

While there is a 12.36% standard deviation, note that the mean counts for each of the 2 methods of event identification, of the 10 runs, are less than .001% different.

Clearly there are some oddities to be aware of and be careful of interpreting results. This doesn't just apply to using the registers (which I believe ARE accurate) but also to the static Hardware and Processor events that perf lists.

There is no question that the ability to access these PEBS events is a very useful feature but it does require you review the Intel manual not only for an accurate understanding of the event being monitored but also for the correct PEBS Event values for the CPU model you are running on.

3.6.3 - Some quirks of perf to be cautious about

Let's look at some quirks that might slew your output. I'm sure as time progresses more will be added.

EXAMPLE 1.

Adding **-a** (All CPU's) to the command **stat** changes the output. See the Orange highlighted items. Note that without **-a**, we're stating **df**. With **-a** we're stat'ing the system while **df** ran NOT just **df**.

```
# perf stat -d df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31949892  36497924  47% /
tmpfs                  7977236       564   7976672   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315637416  50661912  87% /home

 Performance counter stats for 'df':

          1.565273      task-clock (msec)         #    0.725 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               234      page-faults               #    0.149 M/sec
     <not counted>      cycles
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
           529,699      instructions
           194,741      branches                  #  124.413 M/sec
             8,851      branch-misses             #    4.55% of all branches
           240,221      L1-dcache-loads           #  153.469 M/sec
            13,363      L1-dcache-load-misses     #    5.56% of all L1-dcache hits
     <not counted>      LLC-loads
   <not supported>      LLC-load-misses:HG

       0.002160440 seconds time elapsed
```

```
# perf stat -d -a df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
                      72117576  31949904  36497912  47% /
tmpfs                  7977236       564   7976672   1% /dev/shm
/dev/sda1               487652    222859    239193  49% /boot
/dev/mapper/VolGroup-lv_home
                     385901524 315637416  50661912  87% /home

 Performance counter stats for 'system wide':

         38.122198      task-clock (msec)         #   17.612 CPUs utilized           [99.93%]
                23      context-switches          #    0.603 K/sec                   [99.96%]
                 2      cpu-migrations            #    0.052 K/sec                   [99.98%]
               255      page-faults               #    0.007 M/sec
         5,065,864      cycles                    #    0.133 GHz                     [69.21%]
```

```
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
         2,042,396      instructions              #    0.40  insns per cycle         [79.13%]
           462,428      branches                  #   12.130 M/sec                   [83.22%]
            19,826      branch-misses             #    4.29% of all branches         [92.06%]
           834,371      L1-dcache-loads           #   21.887 M/sec                   [99.75%]
           108,665      L1-dcache-load-misses     #   13.02% of all L1-dcache hits   [46.01%]
            47,377      LLC-loads                 #    1.243 M/sec                   [25.65%]
   <not supported>      LLC-load-misses:HG

       0.002164537 seconds time elapsed

#
```

EXAMPLE 2.

An observation: looking at variables. Sometimes it shows multiple lines with slightly differing variables (all can be used)

```
# perf probe -V tcp_sendmsg:34
Available variables at tcp_sendmsg:34
        @<tcp_sendmsg+175>
                int     mss_now
                struct msghdr*  msg
                struct sock*    sk
                struct tcp_sock*        tp
        @<tcp_sendmsg+191>
                int     iovlen
                int     mss_now
                struct iovec*   iov
                struct sock*    sk
                struct tcp_sock*        tp
        @<tcp_sendmsg+208>
                int     iovlen
                struct iovec*   iov
                struct sock*    sk
                struct tcp_sock*        tp
[root@localhost ~]#
```

THIS IS NOT A PROBLEM. It's simply because the source line was broken down into multiple machine code pieces. See the same in crash. It's down to the compiler. Also note that source line 35 is a goto but IS NOT actually coded into disassembler code. It's all about compiler optimization.

```
0xffffffff814b400c <tcp_sendmsg+0xac>:  mov    %eax,-0x5c(%rbp)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 965   <<< PART 1
0xffffffff814b400f <tcp_sendmsg+0xaf>:  mov    0x14c(%r12),%esi
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 960
0xffffffff814b4017 <tcp_sendmsg+0xb7>:  mov    0x18(%rbx),%r15
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 961
0xffffffff814b401b <tcp_sendmsg+0xbb>:  mov    0x10(%rbx),%rbx
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 965   <<< PART 2
0xffffffff814b401f <tcp_sendmsg+0xbf>:  test   %esi,%esi
0xffffffff814b4021 <tcp_sendmsg+0xc1>:  je     0xffffffff814b4030 <tcp_sendmsg+0xd0>
```

```
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1146
0xffffffff814b4023 <tcp_sendmsg+0xc3>:  mov    $0xffffffe0,%eax
0xffffffff814b4028 <tcp_sendmsg+0xc8>:  jmp    0xffffffff814b3fc1 <tcp_sendmsg+0x61>
0xffffffff814b402a <tcp_sendmsg+0xca>:  nopw   0x0(%rax,%rax,1)
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 965   <<< PART 3
0xffffffff814b4030 <tcp_sendmsg+0xd0>:  testb  $0x2,0x40(%r12)
0xffffffff814b4036 <tcp_sendmsg+0xd6>:  jne    0xffffffff814b4023 <tcp_sendmsg+0xc3>
```

```
# perf probe -L tcp_sendmsg
<tcp_sendmsg@/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-
573.18.1.el6.x86_64/net/ipv4/tcp.c:0>
      0  int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                        size_t size)
      2  {
      3         struct sock *sk = sock->sk;
                struct iovec *iov;
                struct tcp_sock *tp = tcp_sk(sk);
                struct sk_buff *skb;
                int iovlen, flags;
                int mss_now, size_goal;
                int err, copied;
                long timeo;

     12         lock_sock(sk);
                TCP_CHECK_TIMER(sk);

     15         flags = msg->msg_flags;
     16         timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);

                /* Wait for a connection to finish. */
     19         if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
     20                 if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
                                goto out_err;

                /* This should be in poll */
     24         clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

     26         mss_now = tcp_send_mss(sk, &size_goal, flags);

                /* Ok commence sending. */
     29         iovlen = msg->msg_iovlen;
     30         iov = msg->msg_iov;
                copied = 0;

                err = -EPIPE;
     34         if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))     <<<<<<<<<<<<<
                        goto out_err;

     37         while (--iovlen >= 0) {
     38                 size_t seglen = iov->iov_len;
     - - - - - - - - - - - 8< - - - - - - - - - - -
```

EXAMPLE 3

You cannot 'perf trace' a process running on a specific CPU. It's one or the other (trace a process or trace a processor)

```
# perf trace -T -p 20993 -- sleep 1
616534033.610 ( 0.000 ms):  ... [continued]: futex()) = -1 ETIMEDOUT Connection timed out
616534033.615 ( 0.001 ms): futex(uaddr: 0x7f8547379af0, op: WAKE|PRIV, val: 1            ) = 0

# perf trace -T -C1 -p 20993 -- sleep 1
PID/TID switch overriding CPU

# perf trace -T -C1 -- sleep 1
```

EXAMPLE 4

Here's another quirk.... **--externs** is positional

```
# perf probe -V --externs tcp_sendmsg    <<<<  Doesn't work
  Error: Don't use --vars with --add/--del.

 usage: perf probe [<options>] 'PROBEDEF' ['PROBEDEF' ...]
    or: perf probe [<options>] --add 'PROBEDEF' [--add 'PROBEDEF' ...]
    or: perf probe [<options>] --del '[GROUP:]EVENT' ...
    or: perf probe --list
    or: perf probe [<options>] --line 'LINEDESC'
    or: perf probe [<options>] --vars 'PROBEPOINT'
   - - - - - - - - - - 8< - - - - - - - - - -
```

```
# perf probe tcp_sendmsg -V --externs   <<<<  Doesn't work
Added new event:                        <<<<  Huh ???????
Error: event "tcp_sendmsg" already exists. (Use -f to force duplicates.)
  Error: Failed to add events.
```

```
# perf probe -V tcp_sendmsg --externs   <<<<  Does work
Available variables at tcp_sendmsg
        @<tcp_sendmsg+0>
                (function_type)*        ip_nat_decode_session
                __u8*   ip_tos2prio
                atomic_long_t   nr_swap_pages
                atomic_long_t*  vm_stat
                atomic_t        tcp_memory_allocated
                char*   __setup_str_set_thash_entries
                cpumask_var_t   cpu_callout_mask
                - - - - - - - - 8< - - - - - - - -
```

EXAMPLE 5

This is a bug. Nothing to be overly concerned about. It happened on RHEL6. I was testing for another issue and found that adding some (not all) probes was sometimes adding duplicates.

```
# perf probe --add tcp_sendmsg:215 -f
Added new events:
  probe:tcp_sendmsg_2  (on tcp_sendmsg:215)
  probe:tcp_sendmsg_3  (on tcp_sendmsg:215)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_3 -aR sleep 1
```

```
# perf probe --add tcp_sendmsg:220 -f
Added new events:
  probe:tcp_sendmsg_4  (on tcp_sendmsg:220)
  probe:tcp_sendmsg_5  (on tcp_sendmsg:220)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_5 -aR sleep 1
```

```
# perf probe -l
/sys/kernel/debug/tracing/uprobe_events file does not exist - please rebuild kernel with
CONFIG_UPROBE_EVENTS.
  probe:__do_page_fault (on __do_page_fault@arch/x86/mm/fault.c)
  probe:__do_page_fault_1 (on __do_page_fault:36@arch/x86/mm/fault.c)
  probe:__do_page_fault_2 (on __do_page_fault@arch/x86/mm/fault.c with address error_code)
  probe:tcp_sendmsg    (on tcp_sendmsg@net/ipv4/tcp.c)
  probe:tcp_sendmsg_1  (on tcp_sendmsg:198@net/ipv4/tcp.c)
  probe:tcp_sendmsg_2  (on tcp_sendmsg:215@net/ipv4/tcp.c)
  probe:tcp_sendmsg_3  (on tcp_sendmsg:215@net/ipv4/tcp.c)
  probe:tcp_sendmsg_4  (on tcp_sendmsg:220@net/ipv4/tcp.c)
  probe:tcp_sendmsg_5  (on tcp_sendmsg:220@net/ipv4/tcp.c)
```

After clearing out all probes and starting again, I was still able to reproduce it although it was sporadic

```
# perf probe --add tcp_sendmsg:222 -f                Worked fine
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg:222)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

```
# perf probe --add tcp_sendmsg:215 -f                    Did not work!!!
Added new events:
  probe:tcp_sendmsg_1  (on tcp_sendmsg:215)
  probe:tcp_sendmsg_2  (on tcp_sendmsg:215)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_2 -aR sleep 1
```

```
# perf probe --add tcp_sendmsg:199 -f                    Worked fine
Added new event:
  probe:tcp_sendmsg_3  (on tcp_sendmsg:199)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_3 -aR sleep 1
```

```
# perf probe --add tcp_sendmsg:201 -f                    Worked fine
Added new event:
  probe:tcp_sendmsg_4  (on tcp_sendmsg:201)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_4 -aR sleep 1
```

EXAMPLE 6

This is a slightly different view of “quirks”... the error messages that perf gives you can often be bland.

ERROR 1 - Error message relating to CONFIG_UPROBE_EVENTS

```
# perf probe --list
/sys/kernel/debug/tracing/uprobe_events file does not exist - please rebuild kernel with
CONFIG_UPROBE_EVENTS.
  probe:tcp_sendmsg    (on tcp_sendmsg@net/ipv4/tcp.c with size skc_refcnt)
```

This is not a problem. If there are probes to list it will display them (as it did here). It's simply saying that while you have kprobes events enabled, you don't have uprobes.

Fixed in later rhel6 and rhel7

ERROR 2 - Forgetting to use --add or -a when adding probes

```
# perf probe tcp_sendmsg:20 -f
Added new event:
Failed to write event: Invalid argument
  Error: Failed to add events.
```

```
# perf probe tcp_sendmsg:20
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg:20)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg -aR sleep 1
```

```
# perf probe tcp_sendmsg:19 -f
Added new event:
Error: event "tcp_sendmsg" already exists. (Use -f to force duplicates.)
  Error: Failed to add events.
```

```
# perf probe --add tcp_sendmsg:19 -f
Added new event:
  probe:tcp_sendmsg_1  (on tcp_sendmsg:19)

You can now use it in all perf tools, such as:

      perf record -e probe:tcp_sendmsg_1 -aR sleep 1
```

You forgot to add **-a** or **--add**. This can be a confusing one. I added **--add** for the last of the 4 commands to show you what happens by ommitting **--add** (or **-a**) for the earlier 3 samples.

ERROR 3 - "Failed to add events" when using a correct line number when adding probes

```
# perf probe --add tcp_sendmsg:24
Added new event:
Error: event "tcp_sendmsg" already exists. (Use -f to force duplicates.)
  Error: Failed to add events.
```

```
# perf probe --add tcp_sendmsg:24 -f
Added new event:
Failed to write event: Invalid argument
  Error: Failed to add events.
```

Okay, so now added the -f

```
# perf probe --list
/sys/kernel/debug/tracing/uprobe_events file does not exist - please rebuild kernel with
CONFIG_UPROBE_EVENTS.
  probe:tcp_sendmsg    (on tcp_sendmsg:20@net/ipv4/tcp.c)
  probe:tcp_sendmsg_1  (on tcp_sendmsg:19@net/ipv4/tcp.c)
```

You check the probes listed

```
# perf probe -L tcp_sendmsg:15-30
<tcp_sendmsg@/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-
573.18.1.el6.x86_64/net/ipv4/tcp.c:15>
     15         flags = msg->msg_flags;
     16         timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
```

```
          /* Wait for a connection to finish. */
19        if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
20                if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
                        goto out_err;

          /* This should be in poll */
24        clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

26        mss_now = tcp_send_mss(sk, &size_goal, flags);

          /* Ok commence sending. */
29        iovlen = msg->msg_iovlen;
30        iov = msg->msg_iov;
```

This one is a weird one. **`clear_bit()`** is an in-line function and a dis of the code does show the code (see following).

```
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/net/ipv4/tcp.c: 1102
0xffffffff814b41b7 <tcp_sendmsg+0x257>: sub    %rdx,%r14
0xffffffff814b41ba <tcp_sendmsg+0x25a>: jne    0xffffffff814b41c9 <tcp_sendmsg+0x269>
0xffffffff814b41bc <tcp_sendmsg+0x25c>: mov    -0x60(%rbp),%r10d
0xffffffff814b41c0 <tcp_sendmsg+0x260>: test   %r10d,%r10d
0xffffffff814b41c3 <tcp_sendmsg+0x263>: je     0xffffffff814b4931 <tcp_sendmsg+0x9d1>
```

A further check shows that perf will display the variables but the code offset values are not matching

```
# perf probe -V tcp_sendmsg:24
Available variables at tcp_sendmsg:24
        @<tcp_sendmsg+152>
                long unsigned int*      addr
                struct msghdr*  msg
                struct sock*    sk
                struct tcp_sock*        tp
```

```
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/arch/x86/include/asm/bitops.h: 104
0xffffffff814b3ff0 <tcp_sendmsg+144>:   mov    0x1e8(%r12),%rax
/usr/src/debug/kernel-2.6.32-573.18.1.el6/linux-2.6.32-573.18.1.el6.x86_64/arch/x86/include/asm/bitops.h: 103
0xffffffff814b3ff8 <tcp_sendmsg+152>:   lock andb $0xfe,0x8(%rax)
```

This is a **`perf`** anomaly. As it turns out, **`tcp_sendmsg+152`** is the code source match for the **`clear_bit()`** function which is always inline. This happens on a RHEL6 release and RHEL7.

EXAMPLE 7

There was always some confusion in my mind about the use of '**--**' prior to the command:

e.g. **`perf record -e <EVENT> -ag -- sleep 1`**

Reading about it on the web gave me some inference that it was associated with the command having or not having an argument. As it turns out, I did find a blog written by Brendan Gregg which clearly states that the use of '--' should be used when preceded by -g (acquire backtrace stacks) because in later versions of perf, -g can have an additional argument. I can say that I've been using the double dash prior to commands on the perf command line that had no -g for some time and never had an issue. So I just thought I'd clarify its proper usage.

EXAMPLE 8

No idea what causes this but I've subsequently discovered it's benign and seems related only to the RHEL6 versions. It shows up in various command sequences.

```
# perf stat -e 'syscalls:sys_enter_*' dd if=/dev/sda of=/dev/null bs=4k count=100000
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'   why??????
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
  Warning: Error: expected 'field' but read 'print'
100000+0 records in
100000+0 records out
409600000 bytes (410 MB) copied, 0.114878 s, 3.6 GB/s
- - - - - - - - - - - - 8< - - - - - - - - - - - -
```

3.6.4 - What default values or variables can you use (eg. $retval)

This is not definitive. The more we use perf, the more we might uncover additionals. Here's what I've uncovered so far.

`%return`	Used when you want to probe a function return E.G. **`perf probe --add __do_page_fault%return`**

`%[registers]`
`%cs %ss %ip %sp %flags`
`%di %si %ax %bx %cx %dx %bp`
`%r8 %r9 %r10 %r11 %r12 %r13`
`%r14 %r15`
	You can use any/all of these as variables. However, if you do not name the variable it will be displayed by perf script default as **`arg`*`nn`*`=`**. So alias name it.
	E.G. **`RAX=%ax`**.

`$retval`	Used when you want to test any return value. This would be seen as a variable. It can be renamed. E.G. **`perf probe --add '__do_page_fault%return ret=$retval'`**
`$stack`	Can be used as an alternative to **`%sp`**.

3.6.5 - An unusual probing requirement/example

Sometimes a real world issue comes along which takes a little more thought and work than just a simple single probe. Such was this case on a RHEL6 kernel but the implications exist on any RHEL release.

There is a Linux function called **unix_stream_connect()**. What makes this one unusual is that the source code shows 6 **'goto out:'** branches. The question was "I need to determine which **'goto'** we're taking. What made this even more complicated was 2 factors:

1. The compiler optimized this code and actually created 2 separate "**out:**" sections in the disassembled listing.
2. Not all the branches were conditional, as there were now 2 separate "**out:**" sections, there were also 2 places where the compiler chose to let the code "drop through" to each **out:**.

Now to complicate the probing. The only way you can tell if you are taking the conditional **jmp** is to probe each one. You can't just probe **out:** as you have no way of knowing which conditional **jmp** got you there. And probing the conditional **jmp** doesn't mean we actually take it.

To solve the problem, I created the following probes. To assist the engineer when reviewing the perf output, I modified the names of the variables being captured to include helpful information. So for example, the first 5 probes capture the %rax and the name also tells you about the conditional jmp:

rax_nFFs

n = the bit number in the **rflags**
FF = the name of the flag (**Sign Flag** or **Zero Flag**)
s = the state condition to transfer on. Bit is off = 0, on = 1

5 conditional branches probed:

```
perf probe --add 'unix_stream_connect+0x42 rax_7SF1=%ax rflags=%flags'
perf probe -f --add 'unix_stream_connect+0x89 rax_6ZF1=%ax rflags=%flags'
perf probe -f --add 'unix_stream_connect+0xaa rax_6ZF1=%ax rflags=%flags'
perf probe -f --add 'unix_stream_connect+0xe9 rax_6ZF1=%ax rflags=%flags'
perf probe -f --add 'unix_stream_connect+0x189 rax_6ZF0=%ax rflags=%flags'
```

2 locations we drop through into the 2 x **out:** sections:

```
perf probe -f --add 'unix_stream_connect+0x2b3 rax=%ax rflags=%flags'
perf probe -f --add 'unix_stream_connect+0x1d0 rax=%ax rflags=%flags'
```

2 locations probing the actual call to **kfree_skb()** in each **out:**

```
perf probe -f --add 'unix_stream_connect+0x1d7 kfree_skb_sk_buff=%di'
```

```
perf probe -f --add 'unix_stream_connect+0x2ba kfree_skb_sk_buff=%di'
```

You will notice I also changed the name of the **%rdi** register to reflect that this is the **sk_buf** struct address and is used in the call to **kfree_skb()**. Reason being in **stap**, you can always format and print extra lines for help but with **perf**, you don't have that ability so using a combined name can help. I'm not suggesting this is the best/perfect answer. The point of this example is many.

- Don't assume that the compiler will generate the code you think (Source has 1 **out:** but the compiler optimized and generated 2 seperate sections of code in the disassembled listing)
- Probing conditional branches can be done but you will should consider capturing the **%flags** in order to determine if the branch was actually taken (assuming like this case, multiple branches = multiple choices)
- There's always the option we don't take the branch to **out:** at all, so don't overlook you need to probe that code area also to determine if you did or did not take that code flow.
- Don't forget that the the compiler optimizes, one of the "**goto**'s" you see in the source code may end up dropping through into the **out:** section.
- There's no ability to print personalized data/headings with **perf** but you can use the ***name=variable*** on the probe as a way to help you identify things more clearly in the output.

3.6.6 - Setting Memory address hardware breakpoints

RHEL6

Sadly this doesn't work on any default RHEL6 kernel or any kernel-debug. That doesn't mean it cannot work on RHEL6. However, to perform hardware address breakpoint monitoring on RHEL6 requires a special .ko kernel module written by Vern Lovejoy. I'm not documenting this here. If this becomes a requirement, contact a PSME so they can assess if this is a practical option for the problem you are chasing. This .ko does not require any special kernel nor debug. It can be installed on any standard RHEL6 customer kernel.

RHEL7

In the following example we're monitoring the kernel's **`tasklist_lock`** cell for anyone who reads or writes the memory address. I obtained the address by simply looking in the **`/boot/System.map.<RELEASE>`** file:

```
ffffffff81943040 D tasklist_lock
```

```
# perf record -e mem:0xffffffff81943040:rw -a
```

```
# perf report
      [perf enters the text user interface]

_raw_read_lock  /proc/kcore
       │
       │
       │
       │    Disassembly of section load2:
       │
       │    ffffffff8163daa0 <load2+0x63daa0>:
       │      nop
       │      push   %rbp
       │      mov    %rsp,%rbp
       │      lock   decq (%rdi)
100.00 │   ┌──jns    14
       │   │  callq  0xffffffff81302090
       │14:└─→pop    %rbp
       │    ← retq
```

```
# perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 4  of event 'mem:0xffffffff81943040:rw'
# Event count (approx.): 4
#
# Overhead  Command      Shared Object      Symbol
# ........  ...........  .................  ..................
```

```
#
    25.00%  kworker/0:1  [kernel.kallsyms]  [k] _raw_read_lock
    25.00%  kworker/0:1  [kernel.kallsyms]  [k] kill_pgrp
    25.00%  lsmd         [kernel.kallsyms]  [k] _raw_read_lock
    25.00%  lsmd         [kernel.kallsyms]  [k] do_wait

#
# (For a higher level overview, try: perf report --sort comm,dso)
#

Press 'h' for help on key bindings
```

As an fyi, you can monitor up to 4 different addresses. Format would be :

```
# perf record -e
mem:0xffffffff819b3080:rw,mem:0xffffffff819b3088:rw,mem:0xffffffff819b3090:rw,mem:0xffffffff819b3098:rw -a
```

3.6.7 - Making the data CSV style for importing into a spreadsheet

This is quite simple but the parameter may be a confusing at first.... For **perf stat** use **-x** and also supply the delimiter you want:

```
# perf stat -e bus-cycles -a -x: sleep 5
97377846::bus-cycles
```

```
# perf stat -e bus-cycles -a -x, sleep 5
99705745,,bus-cycles
```

```
# perf stat -a -d -x, sleep 5
40060.033833,,task-clock
4651,,context-switches
108,,cpu-migrations
1055,,page-faults
3243301936,,cycles
<not supported>,,stalled-cycles-frontend
<not supported>,,stalled-cycles-backend
5816433379,,instructions
928231084,,branches
3732906,,branch-misses
1386223370,,L1-dcache-loads
18408034,,L1-dcache-load-misses
10275840,,LLC-loads
<not supported>,,LLC-load-misses:HG
```

You can do something similar with **report** also and again, supply the delimeter:

```
# perf report -t, --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 0  of event 'sched:sched_kthread_stop'
# Event count (approx.): 0
#
# Overhead,Command,Shared Object,Symbol


# Samples: 0  of event 'sched:sched_kthread_stop_ret'
# Event count (approx.): 0
#
# Overhead,Command,Shared Object,Symbol


# Samples: 0  of event 'sched:sched_wait_task'
# Event count (approx.): 0
#
```

```
# Overhead,Command,Shared Object,Symbol


# Samples: 33  of event 'sched:sched_wakeup'
# Event count (approx.): 53765
#
# Overhead,       Command,    Shared Object,                         Symbol
 99.31,          init,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.44,     syndaemon,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.24,       swapper,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.00,      nautilus,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.00,          perf,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.00,irq/39-iwlwifi,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup
 0.00,         sleep,[kernel.kallsyms],[k] ftrace_profile_sched_wakeup


# Samples: 1  of event 'sched:sched_wakeup_new'
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

Otherwise, what it would display would be:

```
# perf report  --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 0  of event 'sched:sched_kthread_stop'
# Event count (approx.): 0
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......
#


# Samples: 0  of event 'sched:sched_kthread_stop_ret'
# Event count (approx.): 0
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......
#


# Samples: 0  of event 'sched:sched_wait_task'
# Event count (approx.): 0
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......
#


# Samples: 33  of event 'sched:sched_wakeup'
# Event count (approx.): 53765
```

```
#
# Overhead         Command      Shared Object                              Symbol
# ........  ..............  .................  ..............................
#
    99.31%            init  [kernel.kallsyms]  [k] ftrace_profile_sched_wakeup
     0.44%       syndaemon  [kernel.kallsyms]  [k] ftrace_profile_sched_wakeup
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
```

3.6.8 - Looking at the raw data in the perf.data file

From what I've uncovered so far, I believe there are 11 possible types of event. I've included these because it is possible to write your own program to interrogate the **perf.data** file in some manner that you specifically desire. This is by no means comprehensive but does at least give you a starting point.

1. PERF_RECORD_MMAP: Memory map event for .ko modules
2. PERF_RECORD_MMAP2: Memory map event for .so libraries and processes
3. PERF_RECORD_LOST: An unknown event (Perf report shows “Lost”)
4. PERF_RECORD_COMM: Maps a command name string to a process and thread ID
5. PERF_RECORD_EXIT: Process exit
6. PERF_RECORD_THROTTLE:
7. PERF_RECORD_UNTHROTTLE:
8. PERF_RECORD_FORK: Process creation
9. PERF_RECORD_READ:
10. PERF_RECORD_FINISHED_ROUND: Usually follows the PERF_RECORD_MMAP before sampling
11. PERF_RECORD_SAMPLE: Sample of actual hardware counter or software events

The PERF_RECORD_SAMPLE events (samples) are the most interesting ones in terms of program profiling. These are the actual records of captured data. The other event types seem to be mostly useful for keeping track of the perf process starting, memory allocation, exiting etc. Samples are timestamped with an unsigned 64 bit word, which records elapsed nanoseconds since some point in time (system running time, based on the kernel scheduler clock). Samples have themselves a type which is defined in the file header and linked to the sample by an integer identifier.

```
# perf script -D | grep PERF_RECORD
0 0xe0 [0x50]: PERF_RECORD_MMAP -1/0: [0xffffffff81000000(0x1f000000) @ 0xffffffff81000000]: x [kernel.kallsyms]_text
0 0x130 [0x80]: PERF_RECORD_MMAP -1/0: [0xffffffffa0000000(0x1b000) @ 0]: x /lib/modules/2.6.32-
642.11.1.el6.x86_64/kernel/drivers/md/dm-mod.ko
- - - - - - - - - - - - - 8< - - - - - - - - - - - - -
0 0x3cd8 [0x78]: PERF_RECORD_MMAP -1/0: [0xffffffffa0c44000(0x5f3bbfff) @ 0]: x /lib/modules/2.6.32-
642.11.1.el6.x86_64/kernel/crypto/ecb.ko
0x4150 [0x8]: PERF_RECORD_FINISHED_ROUND
347799469617298 0x3d50 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff8104615a period: 1 addr: 0
347799469618928 0x3d78 [0x28]: PERF_RECORD_COMM exec: df:2646/2646
347799469628313 0x3da0 [0x60]: PERF_RECORD_MMAP2 2646/2646: [0x400000(0x15000) @ 0 fd:00 2752610 1797039952]: r-xp /bin/df
347799469635905 0x3e00 [0x70]: PERF_RECORD_MMAP2 2646/2646: [0x3df1800000(0x223000) @ 0 fd:00 3148945 1681226159]: r-xp
/lib64/ld-2.12.so
347799469641638 0x3e70 [0x60]: PERF_RECORD_MMAP2 2646/2646: [0x7ffc789fd000(0x1000) @ 0x7ffc789fd000 00:00 0 0]: ---p [vdso]
347799469699900 0x3ed0 [0x70]: PERF_RECORD_MMAP2 2646/2646: [0x3df1c00000(0x394000) @ 0 fd:00 3152483 1681226160]: r-xp
```

```
/lib64/libc-2.12.so
347799470622474 0x3f40 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 108816 addr: 0
347799470651875 0x3f68 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 98598 addr: 0
347799470678460 0x3f90 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 190586 addr: 0
347799470729289 0x3fb8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 390683 addr: 0
347799470833014 0x3fe0 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 582047 addr: 0
347799471006222 0x4008 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 684669 addr: 0
347799471206060 0x4030 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 722629 addr: 0
347799471408262 0x4058 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 745283 addr: 0
347799471635263 0x4080 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 779986 addr: 0
347799471841893 0x40a8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 789867 addr: 0
347799472069669 0x40d0 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 810586 addr: 0
347799472284113 0x40f8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e24 period: 820475 addr: 0
347799472430854 0x4120 [0x30]: PERF_RECORD_EXIT(2646:2646):(2646:2646)
#
```

Here's the actual data reported by **`perf script`** to show the correlation to the records noted above. The end of the line is truncated. It simply shows you the name of the **`debuginfo vmlinux`** file path:

```
# perf script
  :2646  2646 347799.469617:          1 cycles:  ffffffff8104615a native_write_msr_safe (/usr/lib/debug/lib/modules/2.
     df  2646 347799.470622:     108816 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.470651:      98598 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.470678:     190586 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.470729:     390683 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.470833:     582047 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.471006:     684669 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.471206:     722629 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.471408:     745283 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.471635:     779986 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.471841:     789867 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.472069:     810586 cycles:  ffffffff81250e1e avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
     df  2646 347799.472284:     820475 cycles:  ffffffff81250e24 avtab_search_node (/usr/lib/debug/lib/modules/2.6.32
```

For those of you interested, here's a raw breakdown of the actual records. If you look closely, you can spot the data. Note also, the records do not begin with “PERF_RECORD_SAMPLE”, they start with an event identifier. The PERF_RECORD_SAMPLE that follows the event is a formatted display of that raw data:

```
0x3f40 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 0a 47 e6 5f 52 3c 01 00  V...V....G._R<..
.  0020:  10 a9 01 00 00 00 00 00                          ........
.
347799470622474 0x3f40 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 108816 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.470622:     108816 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x3f68 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 e3 b9 e6 5f 52 3c 01 00  V...V......_R<..
.  0020:  26 81 01 00 00 00 00 00                          &.......
.
```

```
347799470651875 0x3f68 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 98598 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.470651:      98598 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x3f90 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 bc 21 e7 5f 52 3c 01 00  V...V....!._R<..
.  0020:  7a e8 02 00 00 00 00 00                          z.......
.
347799470678460 0x3f90 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 190586 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.470678:     190586 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x3fb8 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 49 e8 e7 5f 52 3c 01 00  V...V...I.._R<..
.  0020:  1b f6 05 00 00 00 00 00                          ........
.
347799470729289 0x3fb8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 390683 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.470729:     390683 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x3fe0 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 76 7d e9 5f 52 3c 01 00  V...V...v}._R<..
.  0020:  9f e1 08 00 00 00 00 00                          ........
.
347799470833014 0x3fe0 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 582047 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.470833:     582047 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x4008 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 0e 22 ec 5f 52 3c 01 00  V...V...."._R<..
.  0020:  7d 72 0a 00 00 00 00 00                          }r......
.
347799471006222 0x4008 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 684669 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.471006:     684669 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x4030 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 ac 2e ef 5f 52 3c 01 00  V...V......_R<..
.  0020:  c5 06 0b 00 00 00 00 00                          ........
.
```

```
347799471206060 0x4030 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 722629 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.471206:     722629 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x4058 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 86 44 f2 5f 52 3c 01 00  V...V....D._R<..
.  0020:  43 5f 0b 00 00 00 00 00                          C_......
.
347799471408262 0x4058 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 745283 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.471408:     745283 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x4080 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 3f bb f5 5f 52 3c 01 00  V...V...?.._R<..
.  0020:  d2 e6 0b 00 00 00 00 00                          ........
.
347799471635263 0x4080 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 779986 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.471635:     779986 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x40a8 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 65 e2 f8 5f 52 3c 01 00  V...V...e.._R<..
.  0020:  6b 0d 0c 00 00 00 00 00                          k.......
.
347799471841893 0x40a8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 789867 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.471841:     789867 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x40d0 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 1e 0e 25 81 ff ff ff ff  ......(...%.....
.  0010:  56 0a 00 00 56 0a 00 00 25 5c fc 5f 52 3c 01 00  V...V...%\._R<..
.  0020:  5a 5e 0c 00 00 00 00 00                          Z^......
.
347799472069669 0x40d0 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e1e period: 810586 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.472069:     810586 cycles:  ffffffff81250e1e avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x40f8 [0x28]: event: 9
.
. ... raw event: size 40 bytes
.  0000:  09 00 00 00 01 00 28 00 24 0e 25 81 ff ff ff ff  ......(.$.%.....
.  0010:  56 0a 00 00 56 0a 00 00 d1 a1 ff 5f 52 3c 01 00  V...V......_R<..
.  0020:  fb 84 0c 00 00 00 00 00                          ........
.
```

```
347799472284113 0x40f8 [0x28]: PERF_RECORD_SAMPLE(IP, 0x1): 2646/2646: 0xffffffff81250e24 period: 820475 addr: 0
 ... thread: df:2646
 ...... dso: /root/.debug/.build-id/61/7f3960c266eacbc1cb06822a1e6dfff7a76837
              df  2646 347799.472284:     820475 cycles:  ffffffff81250e24 avtab_search_node
(/usr/lib/debug/lib/modules/2.6.32-642.11.1.el6.x86_64/vmlinux)

0x4120 [0x30]: event: 4
.
. ... raw event: size 48 bytes
.  0000:  04 00 00 00 00 00 30 00 56 0a 00 00 56 0a 00 00  ......0.V...V...
.  0010:  56 0a 00 00 56 0a 00 00 ae dd 01 60 52 3c 01 00  V...V......`R<..
.  0020:  56 0a 00 00 56 0a 00 00 06 df 01 60 52 3c 01 00  V...V......`R<..
.
347799472430854 0x4120 [0x30]: PERF_RECORD_EXIT(2646:2646):(2646:2646)
```

3.6.9 - Looking at perf.data submitted via an archive

When customers send in perf "data" created with the perf archive command, you may well receive 2 files:

```
$ ll perf*
-rw-rw-r--. 1 sjoh sjoh  230067598 Jun 12 10:08 perf.data.gz
-rw-rw-r--. 1 sjoh sjoh   13328344 Jun 12 10:00 perf.data.tar.bz2
```

First, you need to unpack the **perf.data.tar.bz2** file which contains all the customers symbols etc for their specific release. You can see the directories this builds further on in this section.

```
$ tar -xvf perf.data.tar.bz2
.build-id/15/4d40f7261216cb5b3d2bfc54f66a525faac619
[kernel.kallsyms]/154d40f7261216cb5b3d2bfc54f66a525faac619
.build-id/8a/7e7404a2335231be759cb54f8041344cac0c1b
lib64/libc-2.12.so/8a7e7404a2335231be759cb54f8041344cac0c1b
.build-id/c3/c1efabde9070c96e1785051f892b78926bc3e9
lib64/libgcc_s-4.4.7-20120601.so.1/c3c1efabde9070c96e1785051f892b78926bc3e9
.build-id/fd/f3a36fffe08375456d59da959eab2fc30b6186
lib64/librt-2.12.so/fdf3a36fffe08375456d59da959eab2fc30b6186
.build-id/85/104ecfe42c606b31c2d0d0d2e5dacd3286a341
lib64/libpthread-2.12.so/85104ecfe42c606b31c2d0d0d2e5dacd3286a341
.build-id/78/3dc74cad554282ca738053793f30aca93d38d0
lib64/libdbus-1.so.3.4.0/783dc74cad554282ca738053793f30aca93d38d0
.build-id/1c/c2165e019d43f71fde0a47af9f4c8eb5e51963
lib64/ld-2.12.so/1cc2165e019d43f71fde0a47af9f4c8eb5e51963
.build-id/aa/0f765de0737754d553345a59e9108904dbc572
sbin/init/aa0f765de0737754d553345a59e9108904dbc572
.build-id/72/6e43cab5290d9a9e70b8a3bf69133c0e3188f9
[vdso]/726e43cab5290d9a9e70b8a3bf69133c0e3188f9
.build-id/c3/c08437519a282f8947e8dae1fba4f796dd1990
usr/lib64/libstdc++.so.6.0.13/c3c08437519a282f8947e8dae1fba4f796dd1990
.build-id/8a/852ac42f0b64f0f30c760ebbcfa3fe4a228f12
lib64/libm-2.12.so/8a852ac42f0b64f0f30c760ebbcfa3fe4a228f12
.build-id/1f/7e85410384392bc51fa7324961719a10125f31
lib64/libdl-2.12.so/1f7e85410384392bc51fa7324961719a10125f31
.build-id/f5/42c8acd4ad1f2c6a551043bdfbab051905da1c
lib64/libcrypt-2.12.so/f542c8acd4ad1f2c6a551043bdfbab051905da1c
.build-id/67/5c7288304142c7bca6475b19a1212587dd2dab
usr/lib64/gconv/ISO8859-1.so/675c7288304142c7bca6475b19a1212587dd2dab
.build-id/5c/f511c7790b48fc827a2d85c9a27c2b6ce6fe89
bin/bash/5cf511c7790b48fc827a2d85c9a27c2b6ce6fe89
.build-id/e1/c5b0ff1602f22de18b8bf866c7b39b303096c4
lib64/libtinfo.so.5.7/e1c5b0ff1602f22de18b8bf866c7b39b303096c4
.build-id/42/a1108b8055334a0aa01e3d120edd64151593c9
```

```
lib64/libaio.so.1.0.1/42a1108b8055334a0aa01e3d120edd64151593c9
.build-id/f9/cb2c87b0f2ac3ec7edeea8a210d73de4a83551
lib64/libuuid.so.1.3.0/f9cb2c87b0f2ac3ec7edeea8a210d73de4a83551
.build-id/2d/d93739792452babac950675da93f7170edabe6
lib64/libaudit.so.1.0.0/2dd93739792452babac950675da93f7170edabe6
.build-id/2a/a6119c1c39780a4d841590436bb321c74fa1c1
lib64/libpam.so.0.82.2/2aa6119c1c39780a4d841590436bb321c74fa1c1
.build-id/0e/cb2ae108822ad54382a2df9a3eccf055dc16ca
lib64/security/pam_cracklib.so/0ecb2ae108822ad54382a2df9a3eccf055dc16ca
.build-id/7a/40f30082117824b5b2f24b8421e729ddf0fe18
lib64/security/pam_localuser.so/7a40f30082117824b5b2f24b8421e729ddf0fe18
.build-id/16/90b2ab6ff4dbe1bee903ec9c1ef89297ffd2f4
lib64/security/pam_nologin.so/1690b2ab6ff4dbe1bee903ec9c1ef89297ffd2f4
.build-id/a4/36538388f1f25113fda834ca2eed524efa17d6
lib64/libcap.so.2.16/a436538388f1f25113fda834ca2eed524efa17d6
.build-id/92/21b9cd4b38c4c3fe22b82aa65e2405860e79ca
usr/lib64/libnss3.so/9221b9cd4b38c4c3fe22b82aa65e2405860e79ca
.build-id/62/4c7056b8bbe6ba758def557f516fbdbd01e1fd
lib64/libkrb5.so.3.3/624c7056b8bbe6ba758def557f516fbdbd01e1fd
.build-id/0c/249df4d77989253ccd859956bf50749308a16a
lib64/libgssapi_krb5.so.2.2/0c249df4d77989253ccd859956bf50749308a16a
.build-id/d0/53bb4ff0c2fc983842f81598813b9b931ad0d1
lib64/libz.so.1.2.3/d053bb4ff0c2fc983842f81598813b9b931ad0d1
.build-id/1e/db45c205a844a75ebbb4f0075e705803ffb85b
usr/lib64/libcrypto.so.1.0.1e/1edb45c205a844a75ebbb4f0075e705803ffb85b
.build-id/dc/2b039c31f579b02c0edc80d7e0046e61cbe03c
usr/sbin/sshd/dc2b039c31f579b02c0edc80d7e0046e61cbe03c
.build-id/86/0d7a53e53a7dc49b3e068f961ecc1b586be655
bin/ksh93/860d7a53e53a7dc49b3e068f961ecc1b586be655
.build-id/70/d3b57317ab1e2c9f6c12e59b19f7ba6b489831
lib/ld-2.12.so/70d3b57317ab1e2c9f6c12e59b19f7ba6b489831
.build-id/dc/18560f7a4f807314cf46f3935bb986ac1e6800
lib/libc-2.12.so/dc18560f7a4f807314cf46f3935bb986ac1e6800
.build-id/a2/52f69109451ba0b0d02cb0e87ac789999dec33
lib/libpthread-2.12.so/a252f69109451ba0b0d02cb0e87ac789999dec33
.build-id/7a/e7e0766db0fbdff7dfc3e708f7323dc2f7c1be
usr/bin/perf/7ae7e0766db0fbdff7dfc3e708f7323dc2f7c1be
.build-id/53/842c2896ded0063e1be5c650ce97c67ae97973
usr/lib64/perl5/CORE/libperl.so/53842c2896ded0063e1be5c650ce97c67ae97973
.build-id/98/03b0ba84b681983e5984be4477201f4d8ac918
lib64/rsyslog/imuxsock.so/9803b0ba84b681983e5984be4477201f4d8ac918
.build-id/a1/43cf171a7555801183e8abfab90fa8383ee4f8
sbin/rsyslogd/a143cf171a7555801183e8abfab90fa8383ee4f8
.build-id/f7/7187ad7a3a819bf32ec7e6597be3c6aff2707d
lib64/libglib-2.0.so.0.2800.8/f77187ad7a3a819bf32ec7e6597be3c6aff2707d
.build-id/42/48380901bec4c3c32dffa3780c620f84cd90ea
usr/sbin/irqbalance/4248380901bec4c3c32dffa3780c620f84cd90ea
.build-id/41/bec34d85ba413c9a167dbd9e039cb122bf4f3f
sbin/vxconfigd/41bec34d85ba413c9a167dbd9e039cb122bf4f3f
.build-id/72/37e86b77377f854f0d12738df99cbba6d1352a
```

```
usr/sbin/lldpad/7237e86b77377f854f0d12738df99cbba6d1352a
.build-id/81/bc5c3e6edfdc5cf1617cf6f9ea3fc116b7e2d0
usr/lib64/libconfig.so.8.0.0/81bc5c3e6edfdc5cf1617cf6f9ea3fc116b7e2d0
.build-id/87/9881e85d59c582c6f32e48685a5d5775b3ccd4
usr/sbin/nscd/879881e85d59c582c6f32e48685a5d5775b3ccd4
.build-id/c8/c00281e62ede7a2e33863eb4b4d9163f130867
usr/bin/top/c8c00281e62ede7a2e33863eb4b4d9163f130867
.build-id/5c/26a6a304b52de274d39f0570e3b97d756afeca
lib64/libproc-3.2.8.so/5c26a6a304b52de274d39f0570e3b97d756afeca
.build-id/1e/1c500161fb7fb9d15fa8fa70511d2853075f44
usr/sbin/sendmail.sendmail/1e1c500161fb7fb9d15fa8fa70511d2853075f44
.build-id/d0/6466b4d055a8ca42fb526f499742d45037d7cf
opt/hp/hp-health/bin/hpasmlited/d06466b4d055a8ca42fb526f499742d45037d7cf
.build-id/9a/937791361dfccea6ba96e1da8de157c7679034
usr/lib64/libhponcfg64.so/9a937791361dfccea6ba96e1da8de157c7679034
.build-id/a2/6b0e49cd879eadef12e73a5a30bd7fea9d91df
opt/hp/hp-health/bin/hp-asrd/a26b0e49cd879eadef12e73a5a30bd7fea9d91df
.build-id/5d/324bd01821a190ad13d8b727586894a7cb4d64
opt/BESClient/bin/BESClient/5d324bd01821a190ad13d8b727586894a7cb4d64

bzip2: Compressed file ends unexpectedly;
       perhaps it is corrupted?  *Possible* reason follows.
bzip2: Inappropriate ioctl for device
       Input file = (stdin), output file = (stdout)

It is possible that the compressed file(s) have become corrupted.
You can use the -tvv option to test integrity of such files.

You can use the `bzip2recover' program to attempt to recover
data from undamaged sections of corrupted files.

tar: Unexpected EOF in archive
tar: Unexpected EOF in archive
tar: Error is not recoverable: exiting now
```

Next, you need to unpack the compressed **perf.data.gz** file to a raw **perf.data** file:

```
$ gunzip perf.data.gz
$ ll perf*
-rw-rw-r--. 1 sjoh sjoh 1473693528 Jun 12 10:46 perf.data
-rw-rw-r--. 1 sjoh sjoh  230067598 Jun 12 10:08 perf.data.gz
-rw-rw-r--. 1 sjoh sjoh   13328344 Jun 12 10:00 perf.data.tar.bz2
```

Now let's look at all the unpacked directories (in **Green**) as well. There were other files in the working directory for this customer case which are not highlighted as they are not applicable to **perf**.

```
$ ll
```

```
total 8952396
-rw-rw-r--.  1 sjoh sjoh   94679040 Jun 12 09:59 benwdbs001-20180606-152614 (1).raw
-rw-rw-r--.  1 sjoh sjoh  304484352 Jun 12 09:59 benwdbs001-20180606-152614.raw
-rw-rw-r--.  1 sjoh sjoh        327 Jun 12 10:04 benwdbs001-collectl-201806 (1).log
-rw-rw-r--.  1 sjoh sjoh        327 Jun 12 10:04 benwdbs001-collectl-201806.log
-rw-rw-r--.  1 sjoh sjoh      83292 Jun 12 09:50 benwdbs001.jpg
drwxrwxr-x.  4 sjoh sjoh       4096 Jun 12 10:51 bin
-rw-rw-r--.  1 sjoh sjoh  122276936 Jun 12 10:45 collectl.txt
-rw-rw-r--.  1 sjoh sjoh      83599 Jun 12 09:58 image001.jpg
drwxrwxr-x.  2 sjoh sjoh       4096 Jun 12 10:51 [kernel.kallsyms]
drwxrwxr-x.  5 sjoh sjoh       4096 Jun 12 10:51 lib
drwxrwxr-x. 24 sjoh sjoh       4096 Jun 12 10:51 lib64
drwxrwxr-x.  4 sjoh sjoh       4096 Jun 12 10:51 opt
-rw-rw-r--.  1 sjoh sjoh 1473693528 Jun 12 10:46 perf.data
-rw-rw-r--.  1 sjoh sjoh  230067598 Jun 12 10:08 perf.data.gz
-rw-rw-r--.  1 sjoh sjoh   13328344 Jun 12 10:00 perf.data.tar.bz2
drwxr-xr-x.  2 sjoh sjoh       4096 Jun  5 13:50 sa
drwxrwxr-x.  5 sjoh sjoh       4096 Jun 12 10:51 sbin
-rw-rw-r--.  1 sjoh sjoh     157374 Jun 12 09:59 schbdbs001_dmesg.txt
drwx------. 15 sjoh sjoh       4096 Jun  5 10:09 sosreport-benwdbs001-20180605130631
drwx------. 15 sjoh sjoh       4096 Jun  8 12:08 sosreport-gonangin.02114173-
20180608150547
drwxrwxr-x.  5 sjoh sjoh       4096 Jun 12 10:51 usr
drwxrwxr-x.  2 sjoh sjoh       4096 Jun 12 10:51 [vdso]
```

Now we can see the kernel symbols file that was unpacked and use that in the perf reporting:

```
$ perf script -i perf.data --kallsyms=\
[kernel.kallsyms\]/154d40f7261216cb5b3d2bfc54f66a525faac619 > perf.txt
```

As you can see, with a 1.4GB perf.data file, the raw perf script output created a nearly 7GB text file. This is one of the issue with perf in that it can collect an enormous amount of data in a very short period of time.

```
$ ll perf*
-rw-rw-r--. 1 sjoh sjoh 1473693528 Jun 12 10:46 perf.data
-rw-rw-r--. 1 sjoh sjoh  230067598 Jun 12 10:08 perf.data.gz
-rw-rw-r--. 1 sjoh sjoh   13328344 Jun 12 10:00 perf.data.tar.bz2
-rw-rw-r--. 1 sjoh sjoh 6928279655 Jun 12 11:11 perf.txt
```

3.7 - Real world examples of how perf can provide useful data

This section is not intended to house every possible way perf can be used effectively but instead I've added examples from actual cases where it was able to specifically support and prove conditions. The previous aspects of this course already show actual useful examples of perf but this section is more specific.

3.7.1 - Diagnosing a "retpoline" branch overhead.

Customer reported a problem where **`gettimeofday()`** calls seemed to be taking an excessive amount of extra time after they upgraded to a new kernel. The cause of problem was not recognized at first. However the engineer assigned investigated the code and discovered a coding change. As a result, they wrote 2 small programs to emulate the "before" and "after" kernel code changes and simply using time, determined that on a loop of one billion (1,000,000,000), there was a noticeable difference in overall run time. At this point things stalled what to do next and I was asked for my intervention.

Vern Lovejoy wrote both these 2 super little 'C' programs so full credit to him for those. I've highlighted the code sequence changes between the two releases. Also note that this directly involves the VDSO memory segment in the Userspace as gettimeofday() is one of the last/few remnants of the VDSO feature used any more. Without going off on a tangent explaining this, here's the description from the 'man' page which should help most of you:

> *The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.*
>
> *Why does the vDSO exist at all? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. This is due both to the frequency of the call as well as the context-switch overhead that results from exiting user space and entering the kernel.*
>
> *The rest of this documentation is geared toward the curious and/or C library writers rather than general developers. If you're trying to call the vDSO in your own application rather than using the C library, you're most likely doing it wrong.*

test2a.c – pre 'retpoline' code sequence

```
#include <stdio.h>
#include <sys/time.h>
```

```
typedef struct {
    unsigned int sequence;
    unsigned int lock;
} seqlock_t;

typedef long time_t;
typedef unsigned int u32;
typedef unsigned long long cycle_t;
typedef long __kernel_time_t;
struct vsyscall_gtod_data __vsyscall_gtod_data;
int i;



struct vsyscall_gtod_data {
        seqlock_t       lock;

        /* open coded 'struct timespec' */
        time_t          wall_time_sec;
        u32             wall_time_nsec;

        int             sysctl_enabled;
        struct timezone sys_tz;
        struct { /* extract of a clocksource struct */
                cycle_t (*vread)(void);
                cycle_t cycle_last;
                cycle_t mask;
                u32     mult;
                u32     shift;
        } clock;
        struct timespec wall_to_monotonic;
        struct timespec wall_time_coarse;
};

typedef unsigned long long u64;
#define DECLARE_ARGS(val, low, high)    unsigned low, high
#define EAX_EDX_RET(val, low, high)     "=a" (low), "=d" (high)
#define EAX_EDX_VAL(val, low, high)     ((low) | ((u64)(high) << 32))

static  cycle_t vget_cycles(void)
{
    DECLARE_ARGS(val, low, high);
    asm volatile("rdtsc" : EAX_EDX_RET(val, low, high));
    return EAX_EDX_VAL(val, low, high);
}
static inline void rdtsc_barrier(void)
{
  asm("lfence");
}
```

```
static cycle_t vread_tsc(void)
{
        cycle_t ret;

        /*
 *          * Surround the RDTSC by barriers, to make sure it's not
 *                   * speculated to outside the seqlock critical section and
 *                            * does not cause time warps:
 *                                     */
        rdtsc_barrier();
        ret = (cycle_t)vget_cycles();
        rdtsc_barrier();

        return ret >= __vsyscall_gtod_data.clock.cycle_last ?
                ret : __vsyscall_gtod_data.clock.cycle_last;
}


static cycle_t dyn_vread_tsc(void)
{
    void * fn;

      fn = &vread_tsc;
     asm( "mov %0,%%rax"
          :: "m" (fn));
     asm( "callq *%rax      \n");
}


void main(int argc, char **argv) {

    cycle_t rcycl0,rcycl1;


     rcycl0 =  dyn_vread_tsc();
     for (i=0;i<1000000000;i++) {
      rcycl1 = dyn_vread_tsc();
     }

    printf("%d iterations: alt_dyn_vread_tsc() tsc taken %ld\n", i,rcycl1-rcycl0);
}
```

test-2b.c - post 'retpoline' code sequence

```
#include <stdio.h>
#include <sys/time.h>
```

```
typedef struct {
    unsigned int sequence;
    unsigned int lock;
} seqlock_t;

typedef long time_t;
typedef unsigned int u32;
typedef unsigned long long cycle_t;
typedef long __kernel_time_t;
struct vsyscall_gtod_data __vsyscall_gtod_data;
int i;



struct vsyscall_gtod_data {
        seqlock_t       lock;

        /* open coded 'struct timespec' */
        time_t          wall_time_sec;
        u32             wall_time_nsec;

        int             sysctl_enabled;
        struct timezone sys_tz;
        struct { /* extract of a clocksource struct */
                cycle_t (*vread)(void);
                cycle_t cycle_last;
                cycle_t mask;
                u32     mult;
                u32     shift;
        } clock;
        struct timespec wall_to_monotonic;
        struct timespec wall_time_coarse;
};

typedef unsigned long long u64;
#define DECLARE_ARGS(val, low, high)    unsigned low, high
#define EAX_EDX_RET(val, low, high)     "=a" (low), "=d" (high)
#define EAX_EDX_VAL(val, low, high)     ((low) | ((u64)(high) << 32))

static  cycle_t vget_cycles(void)
{
    DECLARE_ARGS(val, low, high);
    asm volatile("rdtsc" : EAX_EDX_RET(val, low, high));
    return EAX_EDX_VAL(val, low, high);
}
static inline void rdtsc_barrier(void)
{
  asm("lfence");
}
```

```
static cycle_t vread_tsc(void)
{
        cycle_t ret;

        /*
 *           * Surround the RDTSC by barriers, to make sure it's not
 *                    * speculated to outside the seqlock critical section and
 *                             * does not cause time warps:
 *                                      */
        rdtsc_barrier();
        ret = (cycle_t)vget_cycles();
        rdtsc_barrier();

        return ret >= __vsyscall_gtod_data.clock.cycle_last ?
                ret : __vsyscall_gtod_data.clock.cycle_last;
}


static cycle_t alt_dyn_vread_tsc(void)
{
    void * fn;

      fn = &vread_tsc;
     asm( "mov %0,%%rax"
          :: "m" (fn));
     asm( "jmp j_1             \n"
          "c_2:                \n"
          "callq c_1           \n"
          "j_2:                \n"
          "pause               \n"
          "lfence              \n"
          "jmp j_2             \n"
          "c_1:                \n"
          "mov %rax,(%rsp)     \n"
          "retq                \n"
          "j_1:                \n"
          "call c_2            \n"
        );
}


void main(int argc, char **argv) {

    cycle_t rcycl0,rcycl1;


     rcycl0 =  alt_dyn_vread_tsc();
     for (i=0;i<1000000000;i++) {
      rcycl1=alt_dyn_vread_tsc();
```

```
    }

    printf("%d iterations: alt_dyn_vread_tsc() tsc taken %ld\n", i,rcycl1-rcycl0);
}
```

After playing with perf a few seconds, I narrowed down the parameters to specifically show me the PMU stats for Userspace (excluding the kernel using the appropriate optional parameter as shown)

```
# perf stat -e branch-instructions:u,branch-misses:u,bus-cycles:u,cache-misses:u,cache-references:u,cpu-cycles:u,cycles-ct:u,cycles-t:u,el-abort:u,el-capacity:u,el-commit:u,el-conflict:u,el-start:u,instructions:u,mem-loads:u,mem-stores:u,tx-abort:u,tx-capacity:u,tx-commit:u,tx-conflict:u,tx-start:u ./test-2a
1000000000 iterations: alt_dyn_vread_tsc() tsc taken 66545048981

 Performance counter stats for './test-2a':

    10,999,020,836      branch-instructions:u                                         (19.05%)
            47,023      branch-misses:u           #    0.00% of all branches          (19.05%)
     2,353,614,229      bus-cycles:u                                                  (19.06%)
             5,288      cache-misses:u            #   28.129 % of all cache refs      (19.06%)
            18,799      cache-references:u                                            (19.06%)
    88,802,680,874      cpu-cycles:u                                                  (19.06%)
    88,805,173,252      cycles-ct:u                                                   (19.05%)
    88,805,524,269      cycles-t:u                                                    (19.05%)
                 0      el-abort:u                                                    (19.05%)
                 0      el-capacity:u                                                 (19.04%)
                 0      el-commit:u                                                   (19.05%)
                 0      el-conflict:u                                                 (19.05%)
                 0      el-start:u                                                    (19.05%)
    48,951,888,929      instructions:u            #    0.55  insns per cycle          (23.81%)
                 0      mem-loads:u                                                   (23.82%)
    15,996,684,615      mem-stores:u                                                  (23.81%)
                 0      tx-abort:u                                                    (9.53%)
                 0      tx-capacity:u                                                 (9.53%)
                 0      tx-commit:u                                                   (14.29%)
                 0      tx-conflict:u                                                 (19.05%)
                 0      tx-start:u                                                    (19.05%)

      23.821615433 seconds time elapsed

# perf stat -e branch-instructions:u,branch-misses:u,bus-cycles:u,cache-misses:u,cache-references:u,cpu-cycles:u,cycles-ct:u,cycles-t:u,el-abort:u,el-capacity:u,el-commit:u,el-conflict:u,el-start:u,instructions:u,mem-loads:u,mem-stores:u,tx-abort:u,tx-capacity:u,tx-commit:u,tx-conflict:u,tx-start:u ./test-2b
1000000000 iterations: alt_vread_tsc() tsc taken 88047905160

 Performance counter stats for './test-2b':

    13,995,061,584      branch-instructions:u                                         (19.04%)
       999,647,584      branch-misses:u           #    7.14% of all branches          (19.05%)
     3,109,764,279      bus-cycles:u                                                  (19.06%)
             9,833      cache-misses:u            #   61.380 % of all cache refs      (19.06%)
            16,020      cache-references:u                                            (19.06%)
   117,266,870,313      cpu-cycles:u                                                  (19.06%)
   117,268,848,823      cycles-ct:u                                                   (19.06%)
```

```
  117,296,022,132      cycles-t:u                                                    (19.06%)
                0      el-abort:u                                                    (19.06%)
                0      el-capacity:u                                                 (19.06%)
                0      el-commit:u                                                   (19.05%)
                0      el-conflict:u                                                 (19.05%)
                0      el-start:u                                                    (19.05%)
   52,982,367,550      instructions:u            #    0.45  insns per cycle          (23.81%)
                0      mem-loads:u                                                   (23.81%)
   18,013,181,138      mem-stores:u                                                  (23.81%)
                0      tx-abort:u                                                    (9.53%)
                0      tx-capacity:u                                                 (9.52%)
                0      tx-commit:u                                                   (14.28%)
                0      tx-conflict:u                                                 (19.04%)
                0      tx-start:u                                                    (19.04%)

     31.519007965 seconds time elapsed
```

Let me clarify things, I ran these test repeatedely a then b, many times to ensure consistency in the numbers. DO NOT take results from a SINGLE run and assume they are showing you a problem. It is very easy to overlook the impact of cache etc on performance statistics so as in this case, run them repeatedly to ensure you understand what is going on and that the results are 100% consistent.

I highlighted in “red” what was going on. The retpoline code introduced a significant change in the pipeline code flow. Let's lookmat that in more detail to explain what is really going on here. I've also “spaced” out the code and line numbered it to make it more obvious too.

```
a                fn = &vread_tsc;
b             asm( "mov %0,%%rax"
c                   :: "m" (fn));
1      Asm( "       jmp j_1           \n"
2           "c_2:                 \n"
3           "       callq c_1         \n"
4           "j_2:                 \n"
5           "       pause             \n"
6           "       lfence            \n"
7           "       jmp j_2           \n"
8           "c_1:                 \n"
9           "       mov %rax,(%rsp)   \n"
10          "       retq              \n"
11          "j_1:                 \n"
12          "       call c_2          \n"
        );
```

Line 1		jump to j_1
Line 11	j_1	call c_2 (pushes the **%rip** onto the stack) – Basically the address of “line 13”
Line 2&3	c_2	call c_1 (pushes the **%rip** onto the stack) – The address of line 4
Line 8&9	c_1	move the address of the function held in **%rax** onto the stack (address of **vread_tsc()**). This OVERWRITES the **%rip** that was saved by Line 2
Line 10		Return by branching to the address held on the stack. In this case, its the address

of our function **`vread_tsc()`** that we adjusted by saving over the top of the **`%rip`** at line 9. This is the fundamental crux of “retpoline”. We are actually trampolining to a function by actually manipulating the **`%rip`** held on the stack so that we can use the **`ret`** instruction to get there instead of a direct call.

So what was the purpose of line 11?

Very simply, once we return back normally from the called function, the **`%rip`** on the stack will be “line 13” (effectively) and we bypass all the “retpoline” code and continue on as normal.

What's the purpose of line 4 thru 7?

We NEVER execute it, we simply fill the pipeline with code that branches to itself. This is fundamentally required also for the CPUs actual pipeline operation but never gets executed. It causes Pipeline break conditions/issues (branch misses) which helps avoid the pipeline hack that retpoline is designed to eliminate.

So now, taking into account the increase in **`branch-misses`** you see highlighted in red in the perf output, you can see that this is directly responsible for the increase in time it takes to execute this Userspace code which went from 23.8 seconds to 31.5 seconds. Before you think how could this be so bad!!! Remember that we are simply executing Userspace code 1,000,000,000 times so this code is useful to show the issue but not very practical. As far as this piece of code is concerned it does absolutely nothing useful other than literally waste time. However the actual customer's issue was directly involved with this **`gettimeofday()`** issue and they did have a valid concern. As it turns out, there was a simple solution. Remove the retpoline code for VDSO. This had already been done in a later kernel so the change was backported. Problem solved. None the less, it shows you a simple way in which perf could be used specifically and accurately to validate a “performance” degradation issue and identify the specific details behind why it was occurring.

4.0 - Documentation changes

March 1st 2018

1. Updated Brendan's section – 1.0
2. Added some additional links from Brendan's WEB page – 1.13
3. Renumered all the perf sections

March 23rd 2018

1. Added details on how to find the scripts shown by “`perf script --list`” - 3.5.8
2. Added example on how to create a heatmap from futex-contention script – 3.5.8
3. Added how to locate the kernel code for pre-defined events – 3.7
4. Added an additional example of producing histograms – 3..4
5. Added short note on how many functions can be probed using stap, ftrace and perf – 3.2.8

June 29th 2018

1. Added subsection on unpacking customer submitted perf files – 3.6.9
2. Added example of using probe external variables - 3.2.3