Perf

Important Notice : Courseware - Legal

Perf is a multi-tool utility aimed at performance monitoring, cpu profiling and capture for later analysis. It is heavily maintained- mostly because it lives in the kernel source tree (under the *tools/* folder).

From *<kernel-src-tree>/tools/perf/design.txt :*

```
Performance Counters for Linux
------------------------------

Performance counters are special hardware registers available on most
modern CPUs. These registers count the number of certain types of hw
events: such as instructions executed, cache-misses suffered, or branches
mis-predicted - without slowing down the kernel or applications. These
registers can also trigger interrupts when a threshold number of events
have passed - and can thus be used to profile the code that runs on that
CPU.

The Linux Performance Counter subsystem provides an abstraction of these
hardware capabilities. It provides per task and per CPU counters, counter
groups, and it provides event capabilities on top of those.  It provides
"virtual" 64-bit counters, regardless of the width of the underlying
hardware counters.
...
```

From the Wikipedia entry on Perf:

perf (sometimes "Perf Events"[1] or perf tools, originally "Performance Counters for Linux", PCL[2]) - is a performance analyzing tool in Linux, available from kernel version 2.6.31 << July 2009 >>.[3] User-space controlling utility, called 'perf' has git-like interface with subcommands. It is capable of statistical profiling of entire system (both kernel and user code), single CPU or severals threads.

It supports hardware performance counters, tracepoints, software performance counters (e.g. hrtimer), and dynamic probes (e.g. kprobes or uprobes).[4]

In 2012 IBM recognized perf (along with OProfile) as one of the two most commonly used performance counter profiling tools on Linux.[5]

…

Subcommands [edit]

perf is used with several subcommands:

- 'stat': measure total event count for single program or for system for some time
- 'top': top-like dynamic view of hottest functions
- 'record': measure and save sampling data for single program[6]
- 'report': analyze file generated by perf record; can generate flat, or graph profile.[6]
- 'annotate': annotate sources or assembly

• 'sched': tracing/measuring of scheduler actions and latencies[7]

• 'list': list available events

...

Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple commandline interface. Perf is based on the *perf_events* interface exported by recent versions << **from ver 2.6.31** >> of the Linux kernel.

Installation

```
$ perf stat ps
WARNING: perf not found for kernel 3.16.0-38

  You may need to install the following packages for this specific
kernel:
    linux-tools-3.16.0-38-generic
    linux-cloud-tools-3.16.0-38-generic

  You may also want to install one of the following packages to keep up
to date:
    linux-tools-generic
    linux-cloud-tools-generic
$
```

Alternatively, install from Source:

On any recent (2.6.31 onwards) kernel source tree:

```
cd <root-of-kernel-src>
cd tools/perf
make
<< sudo cp ./perf /usr/bin >>
```

-OR- (easier)

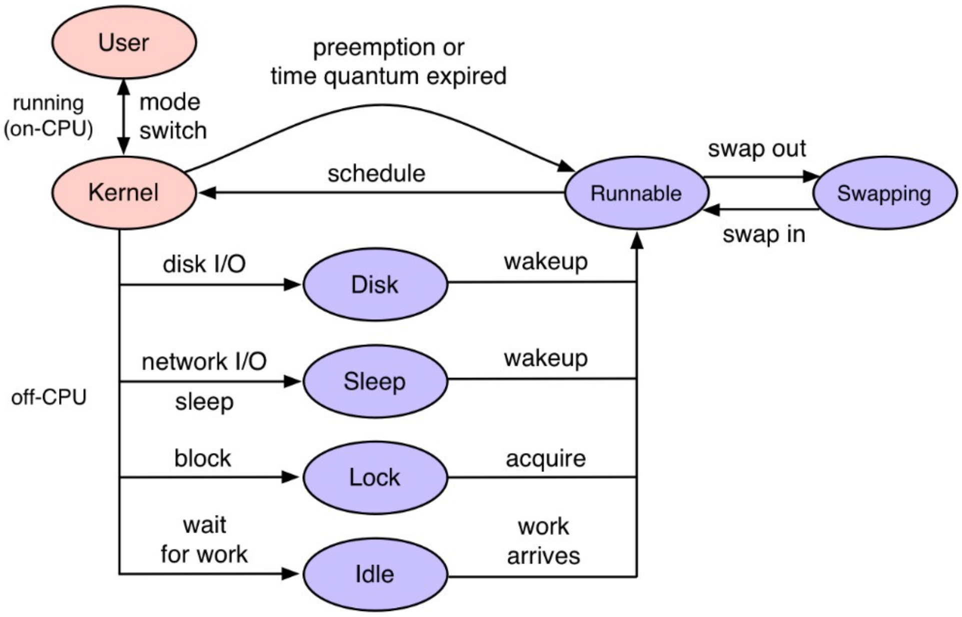
```
sudo apt-get install linux-tools-$(uname -r)
```

Resources

- Linux Perf by Brendan Gregg
- Linux Profiling - Perf

- A tutorial on Perf from the Perf wiki guide
- perf Examples
- Playing around with perf
- *tools/perf/Documentation*
  - *includes tools/perf/Documentation/examples.txt*

- **IMP! Off-CPU Analysis**



- Memory Access Patterns are Important

Source

Events

perf_events instruments "events", which are a unified interface for different kernel instrumentation frameworks. 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.
- **Tracepoint Events**: These are kernel-level events based on the ftrace framework. These tracepoints are placed in interesting and logical locations of the kernel, so that higher-level behavior can be easily traced. For example, system calls, TCP events, file system I/O, disk I/

O, etc. These are grouped into libraries of tracepoints; eg, "sock:" for socket events, "sched:" for CPU scheduler 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.

Apart from those categories of events, there are two other ways perf_events can instrument the system:

- **Profiling**: Snapshots can be collected at an arbitrary frequency, using `perf record -F` *`Hz`*. This is commonly used for CPU usage profiling.
- **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.

Tracepoints

List All available events that one can track:

Note!!
The capability and behaviour of perf ***varies*** *with:*
a) the hardware events supported by the processor
b) user privilege: running as root user shows all usable events; running as a regular user shows a subset of those
c) running perf on a virtual machine: likely you would have less events available than on a physical machine.

```
$ sudo /bin/bash
Password: <xxx>
# 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]
  dummy                                                [Software 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]

--snip--

  exceptions:page_fault_kernel                         [Tracepoint event]
  exceptions:page_fault_user                           [Tracepoint event]
<<
An example of the code within the kernel that 'counts' this perf event:
arch/mips/mm/fault.c
  perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS, 1, regs, address);
>>

--snip--

  kvmmmu:fast_page_fault                               [Tracepoint event]
  kvmmmu:kvm_mmu_invalidate_zap_all_pages              [Tracepoint event]
  kvmmmu:check_mmio_spte                               [Tracepoint event]
  kvm:kvm_entry                                        [Tracepoint event]
  kvm:kvm_hypercall                                    [Tracepoint event]
  kvm:kvm_hv_hypercall                                 [Tracepoint event]
  kvm:kvm_pio                                          [Tracepoint event]
  kvm:kvm_cpuid                                        [Tracepoint event]

--snip--

  drm:drm_vblank_event_queued                          [Tracepoint event]
  drm:drm_vblank_event_delivered                       [Tracepoint event]
  skb:kfree_skb                                        [Tracepoint event]
  skb:consume_skb                                      [Tracepoint event]
  skb:skb_copy_datagram_iovec                          [Tracepoint event]
  net:net_dev_start_xmit                               [Tracepoint event]
  net:net_dev_xmit                                     [Tracepoint event]
  net:net_dev_queue                                    [Tracepoint event]
  net:netif_receive_skb                                [Tracepoint event]
  net:netif_rx                                         [Tracepoint event]
  net:napi_gro_frags_entry                             [Tracepoint event]
  net:napi_gro_receive_entry                           [Tracepoint event]

--snip--

  block:block_split                                    [Tracepoint event]
  block:block_bio_remap                                [Tracepoint event]
  block:block_rq_remap                                 [Tracepoint event]
  jbd2:jbd2_checkpoint                                 [Tracepoint event]
  jbd2:jbd2_start_commit                               [Tracepoint event]
  jbd2:jbd2_commit_locking                             [Tracepoint event]

--snip--

  fs:do_sys_open                                       [Tracepoint event]
  fs:open_exec                                         [Tracepoint event]
```

```
  migrate:mm_migrate_pages                             [Tracepoint event]
  migrate:mm_numa_migrate_ratelimit                    [Tracepoint event]
  compaction:mm_compaction_isolate_migratepages        [Tracepoint event]
  compaction:mm_compaction_isolate_freepages           [Tracepoint event]
  compaction:mm_compaction_migratepages                [Tracepoint event]
  compaction:mm_compaction_begin                       [Tracepoint event]
  compaction:mm_compaction_end                         [Tracepoint event]
  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]

--snip--

  syscalls:sys_exit_unshare                            [Tracepoint event]
  syscalls:sys_enter_mmap                              [Tracepoint event]
  syscalls:sys_exit_mmap                               [Tracepoint event]
  nmi:nmi_handler                                      [Tracepoint event]
  irq_vectors:local_timer_entry                        [Tracepoint event]
  irq_vectors:local_timer_exit                         [Tracepoint event]

--snip--

  xen:xen_cpu_write_gdt_entry                          [Tracepoint event]
  xen:xen_cpu_set_ldt                                  [Tracepoint event]
#

# perf list 'kmem:*'
  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]
#
```

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 "  %-16s %d\n", l, lib[l] } }' | sort | column
    block           19      iwlwifi_io        7     rcu              1
    btrfs           40      iwlwifi_msg       5     regmap          15
    cfg80211        137     iwlwifi_ucode     2     regulator        7
    compaction      5       jbd2             16     rpm              4
    context_tracking 2      kmem             12     sched           23
    drm             3       kvm              42     scsi             5
    exceptions      2       kvmmmu           14     signal           2
```

```
    ext4            91      mac80211        101         skb             3
    filelock        6       mac80211_msg      3         sock            2
    filemap         2       mce               1          spi            7
    fs              2       migrate           2          swiotlb        1
    ftrace          1       module            5          syscalls     588
    gpio            2       napi              1          task           2
    hda             7       net              10          timer         13
    hda_intel       2       nmi               1          udp            1
    i2c             8       oom               1          vmscan        15
    i915            29      pagemap           2          vsyscall       1
    iommu           7       power            21          workqueue      4
    irq             5       printk            1          writeback     25
    irq_vectors     20      random           15          xen           35
    iwlwifi         5       ras               1          xfs          348
    iwlwifi_data    2       raw_syscalls      2          xhci-hcd       9
#
```

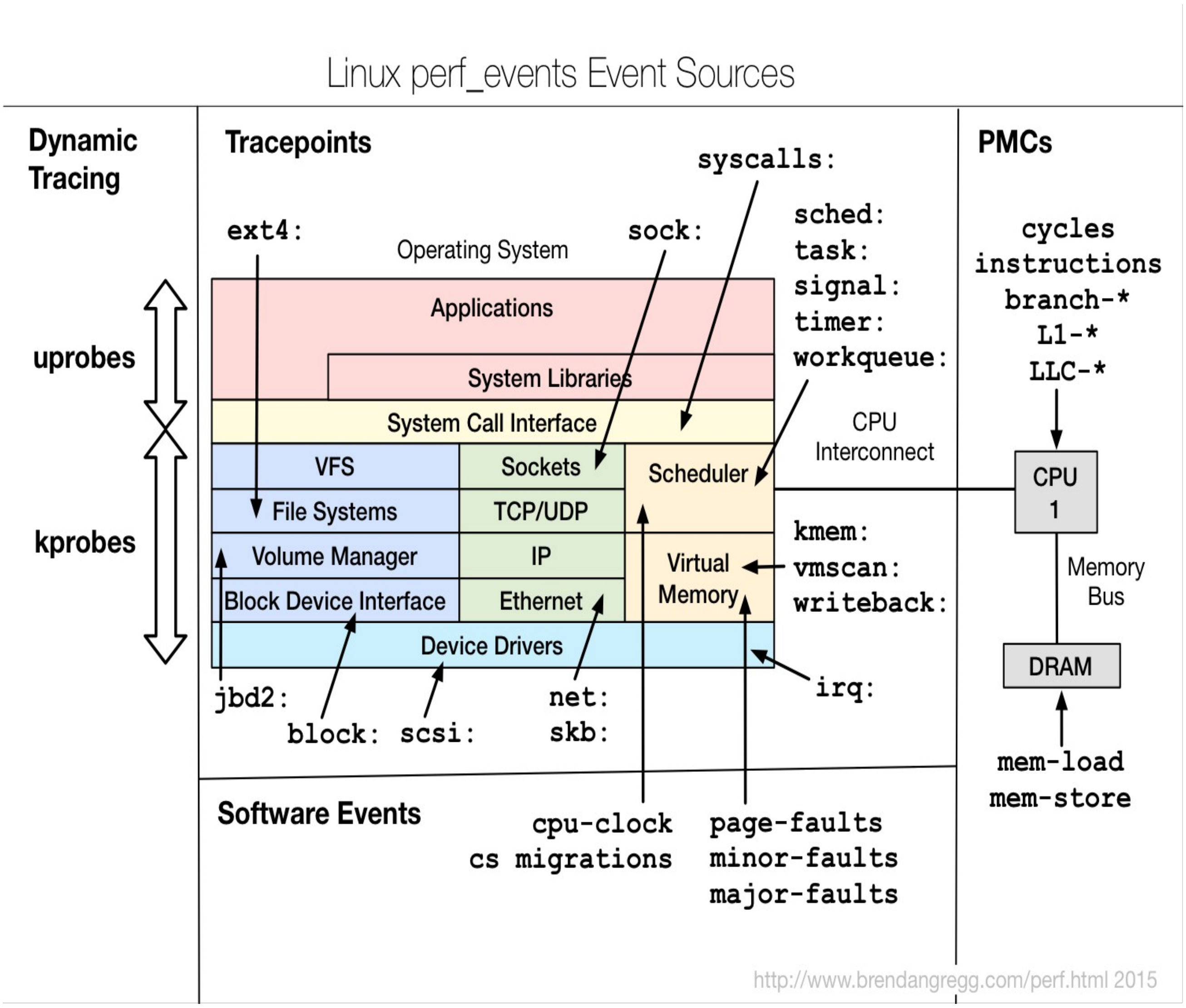
These include:

- **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.

[P.T.O.]

All events diagram:



[PMC = Performance Measurement Counter]

Also:
<< Source >> …
Preflight check

First thing to do when you start working with Perf is to launch **perf test**. This will check your system and kernel features and report if something isn't available. Usually, you need to make as much as possible «OK"s.

Beware though that perf will behave differently when launched under «root» and ordinary user. It's smart enough to let you do some things without root priveleges. There is a control file at */proc/sys/kernel/perf_event_paranoid* that you can tweak in order to change access to perf events:

```
$ perf stat -a
Error:
You may not have permission to collect system-wide stats.
```

```
Consider tweaking /proc/sys/kernel/perf_event_paranoid:
 -1 - Not paranoid at all
  0 - Disallow raw tracepoint access for unpriv
  1 - Disallow cpu events for unpriv
  2 - Disallow kernel profiling for unpriv
```

After you played with *perf test*, you can see what hardware events are available to you with perf list. Again, the list will differ depending on current user id. Also, the amount of events will depend on your hardware: x86_64 CPUs have much more hardware events than some low-end ARM processors.
...

Simple Example

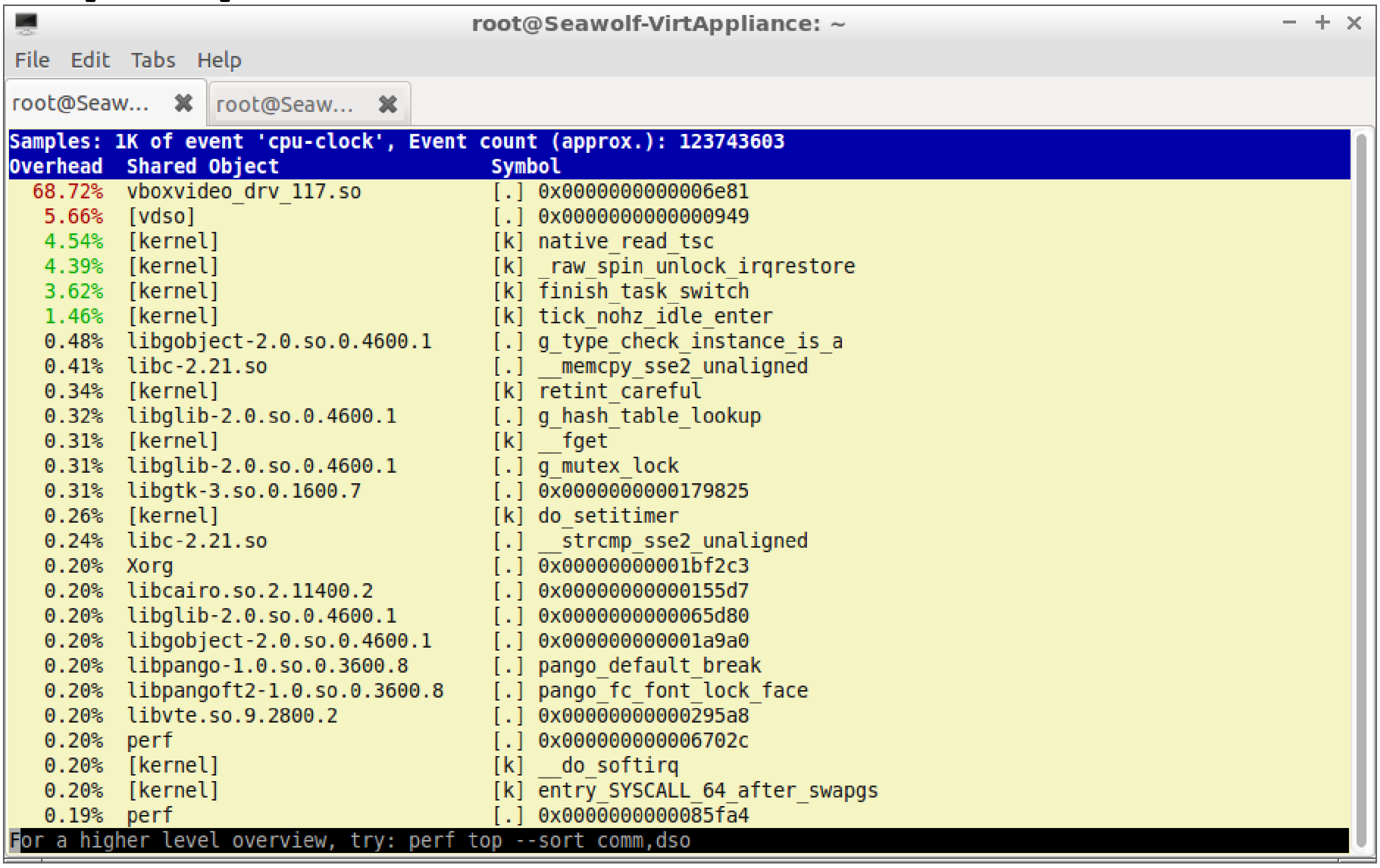
```
# perf stat -a -e 'kmem:kmalloc' sleep 1   << -a => system wide; -e => 
tracepoint event  >>

 Performance counter stats for 'system wide':

             5,347      kmem:kmalloc

       1.001556007 seconds time elapsed

#
# perf stat -e kmem:* sleep 1    << only for the 'sleep' process >>

 Performance counter stats for 'sleep 1':

                 6      kmem:kmalloc
                53      kmem:kmem_cache_alloc
                 0      kmem:kmalloc_node
                 0      kmem:kmem_cache_alloc_node
                15      kmem:kfree
                55      kmem:kmem_cache_free
                43      kmem:mm_page_free
                40      kmem:mm_page_free_batched
                38      kmem:mm_page_alloc
                 6      kmem:mm_page_alloc_zone_locked
                 6      kmem:mm_page_pcpu_drain
                 0      kmem:mm_page_alloc_extfrag

       1.001691044 seconds time elapsed

#
```

Tip: Leave out the '-a' option to report the number of kmalloc's that occurred for precisely the process that was run ('sleep 1' in the above example).

Top-like system-wide Profiling

To get a quick view of overall system state, do:

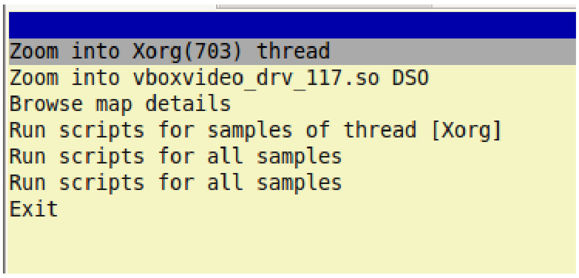
```
# perf top
```



```
root@Seawolf-VirtAppliance: ~
File  Edit  Tabs  Help
root@Seaw...   root@Seaw...
Samples: 1K of event 'cpu-clock', Event count (approx.): 123743603
Overhead  Shared Object                  Symbol
  68.72%  vboxvideo_drv_117.so           [.] 0x0000000000006e81
   5.66%  [vdso]                         [.] 0x0000000000000949
   4.54%  [kernel]                       [k] native_read_tsc
   4.39%  [kernel]                       [k] _raw_spin_unlock_irqrestore
   3.62%  [kernel]                       [k] finish_task_switch
   1.46%  [kernel]                       [k] tick_nohz_idle_enter
   0.48%  libgobject-2.0.so.0.4600.1     [.] g_type_check_instance_is_a
   0.41%  libc-2.21.so                   [.] __memcpy_sse2_unaligned
   0.34%  [kernel]                       [k] retint_careful
   0.32%  libglib-2.0.so.0.4600.1        [.] g_hash_table_lookup
   0.31%  [kernel]                       [k] __fget
   0.31%  libglib-2.0.so.0.4600.1        [.] g_mutex_lock
   0.31%  libgtk-3.so.0.1600.7           [.] 0x0000000000179825
   0.26%  [kernel]                       [k] do_setitimer
   0.24%  libc-2.21.so                   [.] __strcmp_sse2_unaligned
   0.20%  Xorg                           [.] 0x00000000001bf2c3
   0.20%  libcairo.so.2.11400.2          [.] 0x00000000000155d7
   0.20%  libglib-2.0.so.0.4600.1        [.] 0x0000000000065d80
   0.20%  libgobject-2.0.so.0.4600.1     [.] 0x000000000001a9a0
   0.20%  libpango-1.0.so.0.3600.8       [.] pango_default_break
   0.20%  libpangoft2-1.0.so.0.3600.8    [.] pango_fc_font_lock_face
   0.20%  libvte.so.9.2800.2             [.] 0x00000000000295a8
   0.20%  perf                           [.] 0x000000000006702c
   0.20%  [kernel]                       [k] __do_softirq
   0.20%  [kernel]                       [k] entry_SYSCALL_64_after_swapgs
   0.19%  perf                           [.] 0x0000000000085fa4
For a higher level overview, try: perf top --sort comm,dso
```

The above screenshot was taken on a Linux appliance running on VirtualBox. As can be clearly seen, the majority of cpu overhead is that of the VirtualBox video driver (almost 70%!).

Now keep the highlight bar (grey colour), on the top line and press right-arrow key: the following shows up:



```
Zoom into Xorg(703) thread
Zoom into vboxvideo_drv_117.so DSO
Browse map details
Run scripts for samples of thread [Xorg]
Run scripts for all samples
Run scripts for all samples
Exit
```

So we now see that this runs in the process context of Xorg pid 703, i.e., the X server process of course.

Another example: on an Ubuntu VM, some of the Intel e1000 NIC driver's 'methods' seem to be banging on CPU:

```
Samples: 974  of event 'cpu-clock', Event count (approx.): 11622035
Overhead  Shared Object        Symbol
  19.88%  [kernel]             [k] e1000_xmit_frame
  16.37%  [kernel]             [k] e1000_alloc_rx_buffers
  14.70%  [kernel]             [k] e1000_watchdog
  12.98%  [kernel]             [k] _raw_spin_unlock_irqrestore
   9.59%  [kernel]             [k] __softirqentry_text_start
   6.72%  [kernel]             [k] tick_nohz_idle_enter
   6.30%  [kernel]             [k] e1000_clean
   2.15%  libslang.so.2.3.1    [.] SLtt_smart_puts
```

```
Samples: 164  of event 'cpu-clock', Event count (approx.): 9634037
Overhead  Shared Object        Symbol
  22.26%  [kernel]             [k] e1000_xmit_frame
  18.39%  [kernel]             [k] e1000_watchdog
  15.60%  [kernel]             [k] _raw_spin_unlock_irqrestore
  14.86%  [kernel]             [k] e1000_alloc_rx_buffers
   7.96%  [kernel]             [k] tick_nohz_idle_enter
   6.50%  [kernel]             [k] __softirqentry_text_start
   3.07%  [kernel]             [k] e1000_clean
   1.99%  libslang.so.2.3.1    [.] 0x000000000009affb
   1.99%  perf                 [.] 0x00000000001e7864
   1.16%  perf                 [.] 0x00000000001ab96c
```

Quick Summary by process (context):

 # perf top --sort comm,dso

```
Samples: 3K of event 'cpu-clock', Event count (approx.): 550103079
Overhead  Command          Shared Object
  25.78%  :1709            [kernel]
  24.07%  Xorg             vboxvideo_drv_117.so
  14.33%  :1703            [kernel]
   9.15%  :0               [kernel]
   8.47%  :1709            libbz2.so.1.0.4
   5.24%  :1703            libapt-pkg.so.4.16.0
   4.00%  :1709            libapt-pkg.so.4.16.0
   1.32%  :1702            [kernel]
   1.10%  :1656            [kernel]
   1.05%  Xorg             [kernel]
   0.52%  Xorg             Xorg
   0.50%  :1699            [kernel]
```

TIP- to see PID:name, shared object and symbolic name simultaneously:

$ sudo perf top -r 90 --sort pid,comm,dso,symbol ***<< -r 90 => collect data with SCHED_FIFO RT scheduling class and priority 90 [1-99]) >>***

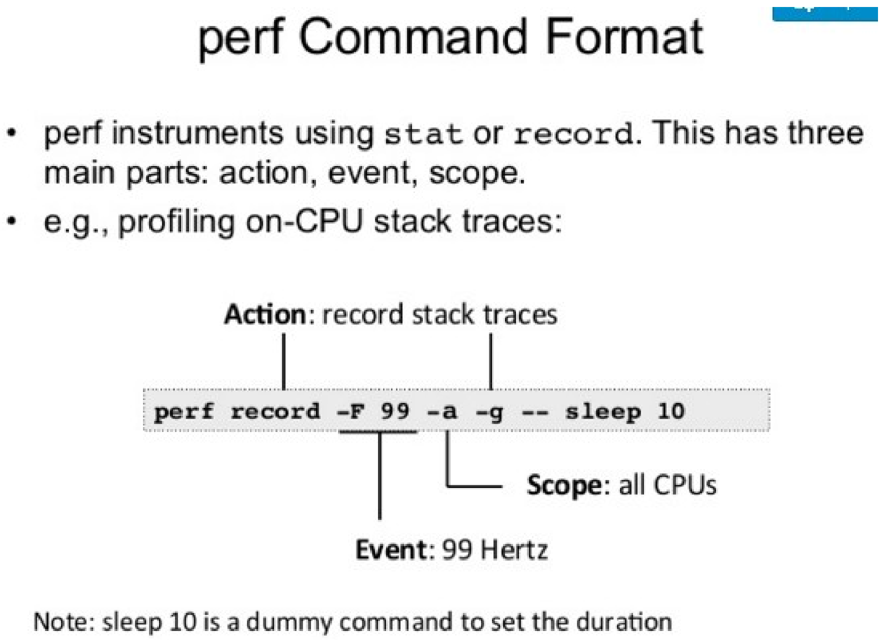
```
Samples: 1M of event 'cycles', Event count (approx.): 259022836877
Overhead      Pid:Command          Command          Shared Object                            Symbol
   1.63%     17213:perf            perf             perf                                     [.] 0x000000000023a0a6
   0.64%     17213:perf            perf             perf                                     [.] 0x000000000019fa07
   0.18%     17213:perf            perf             perf                                     [.] 0x00000000001e8304
   0.17%     17210:perf            perf             perf                                     [.] 0x00000000001ab0ed
   0.15%     17210:perf            perf             perf                                     [.] 0x00000000001e896f
   0.15%      2095:Xorg            Xorg             Xorg                                     [.] 0x000000000013d16f
   0.14%      7891:python2         python2          python2.7                                [.] PyEval_EvalFrameEx
   0.14%      2095:Xorg            Xorg             [kernel]                                 [k] fw_domains_get
   0.13%      3772:compiz          compiz           libcompiz_core.so.0.9.13.1               [.] CompWindow::destroyed
   0.12%     17210:perf            perf             perf                                     [.] 0x000000000019fa07
   0.11%      3772:compiz          compiz           libopengl.so                             [.] PrivateGLScreen::paintOutp
   0.11%     17213:perf            perf             perf                                     [.] 0x00000000001dfea5
   0.10%      2095:Xorg            Xorg             Xorg                                     [.] 0x000000000013906b
   0.10%      3772:compiz          compiz           [kernel]                                 [k] fw_domains_get
   0.09%     17213:perf            perf             perf                                     [.] 0x000000000019f1c5
   0.09%      7507:firefox         firefox          libxcb.so.1.1.0                          [.] 0x000000000000cc55
   0.09%     17213:perf            perf             perf                                     [.] 0x0000000000249be7
   0.09%     17210:perf            perf             libc-2.24.so                             [.] __strcmp_sse2_unaligned
   0.08%     17210:perf            perf             perf                                     [.] 0x00000000001dfe8f
   0.07%      7507:firefox         firefox          libpthread-2.24.so                       [.] pthread_mutex_lock
   0.07%      2095:Xorg            Xorg             libfb.so                                 [.] _fbGetWindowPixmap
   0.07%     17213:perf            perf             perf                                     [.] 0x00000000001e8267
   0.07%      2095:Xorg            Xorg             Xorg                                     [.] 0x00000000000ef5c6
   0.07%      7507:firefox         firefox          libxul.so                                [.] 0x00000000009d43cd
   0.07%     17213:perf            perf             libc-2.24.so                             [.] __strcmp_sse2_unaligned
   0.07%      3772:compiz          compiz           libc-2.24.so                             [.] _int_malloc
   0.07%     17213:perf            perf             perf                                     [.] 0x00000000001e9596
   0.06%     17213:perf            perf             perf                                     [.] 0x00000000001dfea8
no symbols found in /home/kaiwan/MentorGraphics/Sourcery CodeBench Lite for ARM GNU Linux/bin/arm-none-linux-gnueabi-ld, mayb
```

For convenience

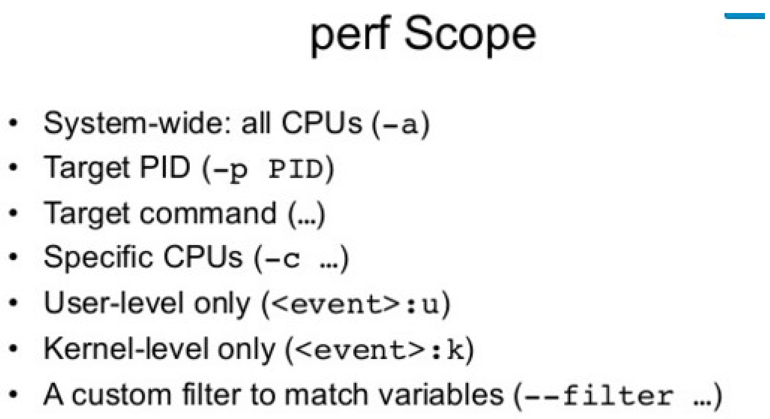
```
$ alias ptop
alias ptop='sudo perf top'
$ alias ptopv
alias ptopv='sudo perf top -r 80 -f 99 --sort pid,comm,dso,symbol \
--demangle-kernel -v --call-graph dwarf,fractal'
$
```

Q. How does one get a report?
A. 2 steps:

1. Use the
perf record -e <events> [-F <sampling_frequency> -a] [command]
syntax to capture data to a binary file (called *perf.data* by default, use –output <fname> to save to an alternate pathname).



<< Note: substitute '-g' with '--call-graph dwarf' >>



2. Use the
`perf report [--stdio]`
syntax to display & analyze the report.

<<
SIDEBAR

Perf for Android?
This page from Linaro - *Using Perf on Android* - covers it quite well.

>>

Eg. *<See the "Profiling" and "Static Tracing" section [here](#) for many examples >*
Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds:
 # perf record -F 99 -a --call-graph dwarf sleep 10

-F 99 => frequency of 99 (i.e. sample @ a rate of 99 times / second)
-a => scope: all cpu's
--call-graph dwarf (older: -g) => Enables call-graph (stack chain/backtrace) recording. (More details follow).

```
# perf report --stdio
Failed to open /tmp/perf-3465.map, continuing without symbols
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 49250631497
#
# Children      Self          Command                Shared Object
# ........  ........  ...............  ............................  ..........................
..................................................
#
     5.83%     0.00%          firefox  [unknown]                     [.] 0000000000000000
                    |
                    --- (nil)
                       |
                       |--16.38%-- 0x0
                       |          |
                       |          |--92.00%-- 0x7f7a198351c4
                       |          |          0xf88948fffa8f17e9
                       |          |
                       |           --8.00%-- 0x0
                       |
                       |--14.00%-- 0x10200
                       |          0x7573736920746f6e
                       |
--snip--

    4.45%     0.00%  evince-thumbnai  [kernel.kallsyms]             [k] system_call_fastpath
            |
            --- system_call_fastpath
               |
               |--40.15%-- open64
               |          |
               |          |--96.40%-- _dl_map_object
               |          |
               |           --3.60%-- open_path
               |                     _dl_map_object
               |
               |--8.30%-- mprotect
               |          |
               |          |--65.98%-- _dl_map_object
               |          |
               |           --34.02%-- dl_main
               |                     _dl_sysdep_start

--snip--
#
```

Example 1 : What's causing system boot to take so long?

On a custom product (based on Ubuntu 15.10), boot was taking (much) longer than expected. How does one investigate this?

I setup a cron job (as root) to let perf sample and record system-wide:

```
# crontab -l
...
@reboot /usr/bin/perf record -F 512 -a --call-graph dwarf --output
</path/to/perf-out> -s -T --realtime=99
#

[ -F 512 : record @ freq of 512 HZ
  -a     : all cpu's, full system
  --call-graph dwarf : record stack using DWARF
  --output : specify perf data output file (default: perf.data)
  -s     : per-thread counts
  -T     : sample timestamps (see with perf script)
  --realtime=99 : run perf with scheduling policy SCHED_FIFO max
priority (99)
                    (root access required)
]
```

After booting:

```
# perf report --input </path/to/perf-out> -f
Samples: 2K of event 'cycles', Event count (approx.): 22247721974
  Children      Self  Command          Shared Object                            Symbol
+   16.88%     0.06%  apt-check        python3.4                                [.] PyEval_EvalCodeEx
+   16.88%     0.29%  apt-check        python3.4                                [.] PyEval_EvalFrameEx
+   15.81%     0.00%  apt-check        libc-2.21.so                             [.] __libc_start_main
+   15.81%     0.00%  apt-check        python3.4                                [.] main
+   15.81%     0.00%  apt-check        python3.4                                [.] Py_Main
+   15.79%     0.00%  apt-check        python3.4                                [.] _start
+   15.49%     0.00%  apt-check        python3.4                                [.] PyEval_EvalCode
+   15.47%     0.00%  apt-check        python3.4                                [.] PyRun_FileExFlags
+   15.47%     0.00%  apt-check        python3.4                                [.] 0x00000000001f7452
+   15.45%     0.00%  apt-check        python3.4                                [.] PyRun_SimpleFileExFlags
+   10.76%     0.00%  freshclam        libclamav.so.6.1.26                      [.] 0x0000000000042d52
+   10.76%     0.00%  freshclam        libclamav.so.6.1.26                      [.] 0x0000000000041c20
+   10.76%     0.00%  freshclam        libz.so.1.2.8                            [.] gzread
+   10.46%     0.00%  freshclam        libz.so.1.2.8                            [.] 0x0000000000010a7a
+   10.46%     0.06%  freshclam        libz.so.1.2.8                            [.] inflate
+   10.38%     0.00%  apt-check        python3.4                                [.] PyObject_Call
+    9.02%     0.00%  apt-check        python3.4                                [.] 0x0000000000159c55
+    8.57%     0.10%  swapper          [kernel.kallsyms]                        [k] cpu_startup_entry
+    7.88%     0.00%  apt-check        apt_pkg.cpython-34m-x86_64-linux-gnu.so  [.] 0x000000000002064f
+    7.88%     0.00%  apt-check        libapt-pkg.so.4.16.0                     [.] _ZN12pkgCacheFile4OpenEP10OpProgres
+    7.67%     0.00%  apt-check        libapt-pkg.so.4.16.0                     [.] _ZN12pkgCacheFile13BuildDepCacheEP1
+    7.67%     0.30%  apt-check        libapt-pkg.so.4.16.0                     [.] _ZN11pkgDepCache4InitEP10OpProgress
+    6.87%     0.01%  swapper          [kernel.kallsyms]                        [k] call_cpuidle
+    6.85%     0.00%  swapper          [kernel.kallsyms]                        [k] cpuidle_enter
+    5.55%     0.75%  apt-check        libapt-pkg.so.4.16.0                     [.] _ZN11pkgDepCache6UpdateEP10OpProgre
+    4.76%     0.00%  apt-check        python3.4                                [.] 0x00000000000cf143
+    4.73%     0.00%  apt-check        python3.4                                [.] 0x0000000000159cac
+    4.63%     0.00%  swapper          [kernel.kallsyms]                        [k] x86_64_start_kernel
```

The screenshot above shows that 'apt-check' and 'freshclam' are consuming large amounts of cpu. (apt-check, indeed, has been known to consume cpu at boot. Disabling it helps!
Regarding freshclam/clamav – it's an opensource antivirus solution – there are two choices: either uninstall clamav or reconfigure freshclam to have the daemon work in the background).

After reconfiguring freshclam and rebooting the system, the perf report shows:

```
# perf report -i perf_all.data --sort comm,dso

+   28.49%     2.90%  apt-check         python3.4
+   27.39%     3.95%  apt-check         libc-2.21.so
+   26.52%    22.00%  apt-check         libapt-pkg.so.4.16.0
+   24.89%     0.45%  apt-check         apt_pkg.cpython-34m-x86_64-linux-
gnu.so
+    8.12%     8.12%  swapper           [kernel.kallsyms]
+    5.17%     0.99%  Xorg              Xorg
+    4.85%     0.31%  Xorg              libc-2.21.so
+    3.99%     0.41%  zenity            libgtk-3.so.0.1600.7
+    3.81%     1.03%  lxpanel           libc-2.21.so
+    3.47%     0.01%  lxpanel           lxpanel
+    3.24%     0.46%  pcmanfm           libc-2.21.so
...
...
```

Disabling apt-check at boot then eliminates it's overhead.

[BTW there are interesting tools like *bootchart* too, a visualization tool to show boot-time activity (it renders a Gantt-like chart displaying which processes are taking cpu)].

<<
*Hey, guess what, there's an easier (and very detailed) way to see 'who took so damn long' to startup – if you're using **systemd**:*

```
$ systemd-analyze <tab-tab>
blame              critical-chain  dot              dump             plot
set-log-level      syscall-filter  time             verify
$
```

Eg.
CLI: systemd-analyze blame
GUI: systemd-analyze *plot*   -writes an SVG!

Eg.

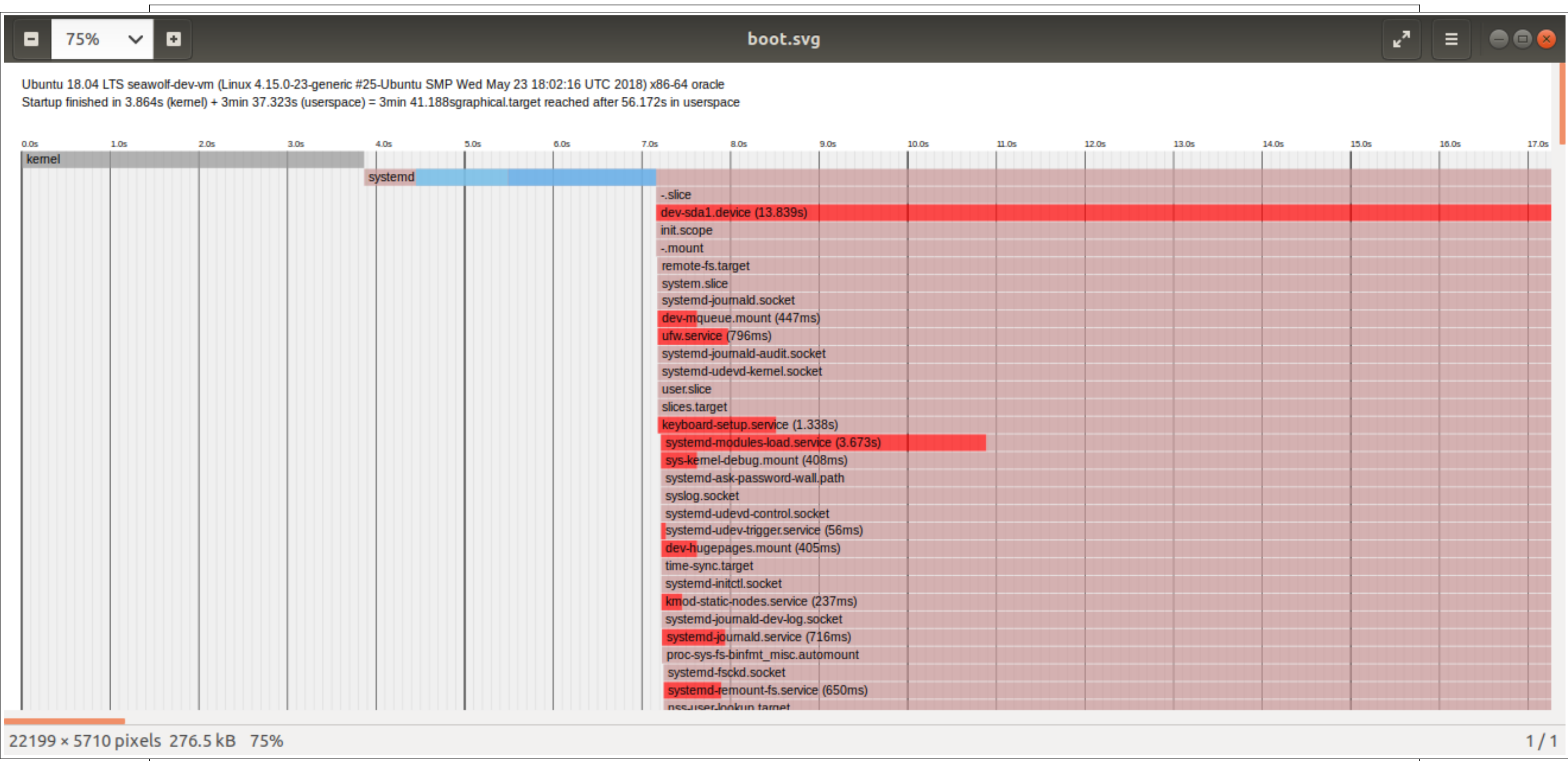
```
$ systemd-analyze blame
         32.050s apt-daily-upgrade.service
         30.276s plymouth-quit-wait.service
         27.698s apt-daily.service
         14.548s snapd.service
         13.839s dev-sda1.device
         11.632s ModemManager.service
         10.680s udisks2.service
          9.950s dev-loop10.device
          9.904s dev-loop11.device
          9.857s dev-loop12.device
          9.839s dev-loop6.device
...
```

```
          6.252s apparmor.service
          6.209s apport.service
          5.481s NetworkManager-wait-online.service
          5.336s polkit.service
          4.648s avahi-daemon.service
          4.543s systemd-udevd.service
          4.378s wpa_supplicant.service

...

            28ms dev-loop17.device
            19ms snapd.socket
            13ms systemd-update-utmp-runlevel.service
             6ms sys-kernel-config.mount
             4ms sys-fs-fuse-connections.mount
$
```

GUI: systemd-analyze *plot* -writes an SVG!

```
$ systemd-analyze plot > boot-time.svg
$
```

View the SVG within a web browser (or image viewer app):



Very interestingly, once [snap](#) was purged from Ubuntu 18.04 LTS, the startup time reduced dramatically:

	Kernel	*Userspace*
With snap	3.864s	3min 31.323s
Without snap	2.906s	26.191s

>>

Example 2 : firefox hanging

On my Ubuntu 15.10 box, firefox seemed to be hung up. Tools like top or htop could not reveal much. *(Recall, capturing the "Who? and How?" are easy, but the "What? and Why?" are not so easy!).*

So, lets use perf:

```
# pgrep firefox    << get firefox's PID >>
26189
# perf record --call-graph dwarf -p 26189  << record stats including
stack >>
Lowering default frequency rate to 3250.
```

```
Please consider tweaking /proc/sys/kernel/perf_event_max_sample_rate.
...
^C[ perf record: Woken up 15 times to write data ]
[ perf record: Captured and wrote 8.671 MB perf.data (557 samples) ]
#
#
# ls -lh perf.data
-rw------- 1 root root 8.7M Dec 29 13:11 perf.data
# perf report --stdio

[P.T.O.]
```

```
root@kaiwan-ThinkPad-X220: ~    kaiwan@kaiwan-ThinkPad-X220: ~    kaiwan@kaiwan-ThinkPad-X220: ~
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 557  of event 'cycles'
# Event count (approx.): 207872625
#
# Children      Self  Command          Shared Object       Symbol
# ........  ........  ...............  ..................  ..........................................................
#
    63.43%     0.00%  DOM Worker       libc-2.21.so        [.] __clone
              |
              ---__clone

    63.43%     0.00%  DOM Worker       libpthread-2.21.so  [.] start_thread
              |
              ---start_thread
                 __clone

    63.43%     0.00%  DOM Worker       libnspr4.so         [.] 0x00000000000296c8
              |
              ---0x7f2b0abe86c8
                 start_thread
                 __clone

    63.43%     0.00%  DOM Worker       libxul.so           [.] 0x00000000009e0442
              |
              ---0x7f2b05428442
                 0x7f2b0abe86c8
                 start_thread
                 __clone

    63.43%     0.00%  DOM Worker       libxul.so           [.] 0x0000000000bbc962
              |
              ---0x7f2b05604962
                 0x7f2b05428442
:
```

Clearly, firefox seems to be stuck in spawning threads (notice the multiple "__clone"s in the stack traces).
Furthermore, a quick check with ps shows that "DOM Worker" is a child thread of the firefox process:

```
$ ps -LA|grep "^26189" |wc -l        << ps: -L options shows threads >>
81
$ ps -LA|grep "^26189"
26189 26189 ?        01:50:24 firefox
26189 26292 ?        00:00:00 gmain
26189 26293 ?        00:00:11 gdbus
26189 26294 ?        00:00:00 dconf worker
26189 26295 ?        00:00:07 Gecko_IOThread
26189 26296 ?        00:00:00 Link Monitor
26189 26297 ?        00:06:59 Socket Thread
26189 26298 ?        00:00:00 firefox
26189 26299 ?        00:01:11 JS Helper
```

```
26189 26300 ?        00:01:12 JS Helper
26189 26301 ?        00:01:14 JS Helper
26189 26302 ?        00:01:15 JS Helper
26189 26303 ?        00:01:13 JS Helper
26189 26304 ?        00:01:13 JS Helper
26189 26305 ?        00:01:12 JS Helper
26189 26306 ?        00:01:15 JS Helper
26189 26307 ?        00:00:01 JS Watchdog
26189 26308 ?        00:00:00 Hang Monitor
26189 26312 ?        00:00:15 Cache2 I/O
26189 26313 ?        00:05:11 Timer
26189 26314 ?        00:00:00 firefox
26189 26315 ?        00:00:00 Cert Verify
26189 26317 ?        00:00:00 GMPThread
26189 26319 ?        00:03:16 Compositor
26189 26320 ?        00:00:23 ImageBridgeChil
26189 26321 ?        00:00:14 ImgDecoder #1
26189 26322 ?        00:00:13 ImgDecoder #2
26189 26323 ?        00:00:13 ImgDecoder #3
26189 26324 ?        00:00:00 ImageIO
26189 26325 ?        00:05:03 SoftwareVsyncTh
26189 26326 ?        00:00:01 HTML5 Parser
26189 26327 ?        00:00:00 IPDL Background
26189 26331 ?        00:00:14 DOM Worker
26189 26335 ?        00:00:21 DOM Worker
26189 26341 ?        00:00:00 DOM Worker
26189 26342 ?        00:00:01 mozStorage #1
26189 26343 ?        00:00:00 firefox
26189 26344 ?        00:00:00 Cache I/O
...
26189 14708 ?        00:00:00 firefox
$
```

Found them! But which thread of the several (below) "DOM Worker" threads above is the culprit? Lets look at individual threads:

```
$ ps -LA|grep "^26189.*DOM Worker"
26189 26331 ?        00:00:14 DOM Worker
26189 26335 ?        00:00:21 DOM Worker
26189 26341 ?        00:00:00 DOM Worker
26189 26375 ?        00:00:13 DOM Worker
26189  1573 ?        00:00:30 DOM Worker
26189 12012 ?        00:00:37 DOM Worker
$
```

To see the report by thread, use the --*tid* switch:

```
$ perf report --stdio --tid=26331
# To display the perf.data header info, please use --header/--header-
only options.
#
#
# Total Lost Samples: 0
```

```
#
# Samples: 557  of event 'cycles'
# Event count (approx.): 207872625
#
# Children      Self  Command     Shared Object        Symbol
# ........  ........  ..........  ..................  ..................
.....................
#
     1.23%     0.00%  DOM Worker  libxul.so           [.]
0x00000000029de5b0
            |
            ---0x7f2b074265b0
               0x7f2b072284e0
               0x7f2b074852b5
               0x7f2b0748617e
               0x7f2b07486522
               0x7f2b074883d0
               0x7f2b0647f23f
               0x7f2b0647f29e
               0x7f2b0648786b
               0x7f2b0648d167
               0x7f2b0648ef7e
               0x7f2b06467a3e
               0x7f2b05425268
               0x7f2b05441667
               0x7f2b056146bd
               0x7f2b05604962
               0x7f2b05428442
               0x7f2b0abe86c8
               start_thread
               __clone
...
…
$
```

Does not seem to be the thread causes CPU overhead (only 1.23% overhead). So we check out the other “DOM Worker” threads until we find the one eating CPU:

…

```
$ perf report --stdio --tid=1573 -v
 symsrc__init: cannot get elf header.
Looking at the vmlinux_path (7 entries long)
Using /proc/kcore for kernel object code
Using /proc/kallsyms for symbols
unwind: pthread_cond_timedwait@@GLIBC_2.3.2:ip = 0x7f2b0bf75149 (0xd149)
unwind: reg 16, val 7f2b0bf75149
unwind: reg 7, val 7f2ae82f2b00
unwind: find_proc_info dso /lib/x86_64-linux-gnu/libpthread-2.21.so
unwind: no map for 7f2b0c1812b0
unwind: access_mem 0x7f2b0c1812b0 not inside range 0x7f2ae82f2b00-
0x7f2ae82f4000
unwind: pthread_cond_timedwait@@GLIBC_2.3.2:ip = 0x7f2b0bf75149 (0xd149)
...
```

```
...
unwind: __clone:ip = 0x7f2b0b1ffeed (0x106eed)
# To display the perf.data header info, please use --header/--header-
only options.
#
#
# Total Lost Samples: 0
#
# Samples: 557  of event 'cycles'
# Event count (approx.): 207872625
#
# Children      Self  Command     Shared Object
Symbol
# ........  ........  ..........  .....................................
..  ..................................................
#
    63.43%     0.00%  DOM Worker  /lib/x86_64-linux-gnu/libc-2.21.so
0x106eed            u [.] __clone
            |
            ---__clone

    63.43%     0.00%  DOM Worker  /lib/x86_64-linux-gnu/libpthread-
2.21.so  0x76aa              l [.] start_thread
            |
            ---start_thread
               __clone

    63.43%     0.00%  DOM Worker  /usr/lib/firefox/libnspr4.so
0x296c8             l [.] 0x00000000000296c8
            |
            ---0x7f2b0abe86c8
               start_thread
               __clone

    63.43%     0.00%  DOM Worker  /usr/lib/firefox/libxul.so
0x9e0442            l [.] 0x00000000009e0442
            |
            ---0x7f2b05428442
               0x7f2b0abe86c8
               start_thread
               __clone

    63.43%     0.00%  DOM Worker  /usr/lib/firefox/libxul.so
0xbbc962            l [.] 0x0000000000bbc962
            |
            ---0x7f2b05604962
               0x7f2b05428442
               0x7f2b0abe86c8
               start_thread
               __clone

    63.43%     0.00%  DOM Worker  /usr/lib/firefox/libxul.so
0xbcc6bd            l [.] 0x0000000000bcc6bd
            |
```

```
              ---0x7f2b056146bd
                 0x7f2b05604962
                 0x7f2b05428442
                 0x7f2b0abe86c8
                 start_thread
                 __clone
...
$
```

Example 3 : Qemu/KVM: Running an x86_64 (Debian) guest with and without KVM hypervisor acceleration

Environment:

Host: Ubuntu 17.04, 4.10.0-35-generic kernel.

Host CPU:

```
$ lscpu |egrep -i "name|vt|ept"
Model name:             Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
Virtualization:         VT-x
Flags:                  fpu vme [...] flexpriority ept vpid [...]
$
```

qemu-system-x86_64 full virtualization:

```
$ qemu-system-x86_64 --version
QEMU emulator version 2.8.0(Debian 1:2.8+dfsg-3ubuntu2.5)
Copyright (c) 2003-2016 Fabrice Bellard and the QEMU Project developers
$
```

A helper script

```
[...]
QEMU_ARGS_COMMON="-kernel ${KERNEL_IMG} -drive file=$
{ROOTFS},if=virtio,format=raw -append root=/dev/vda"

[...]
# Perf stuff
PERF_REC_ARGS="-F 256 -s -T --realtime=99 --call-graph dwarf"
PERF_RAWDATA_WITHKVM=perf_raw_withkvm.data
PERF_RAWDATA_NOKVM=perf_raw_nokvm.data

[...]
```

3.1 WITH KVM

<< First, we make sure the kvm.ko and kvm-[intel|amd].ko kernel modules are loaded up >>

```
...

 qemu-system-x86_64 --enable-kvm ${QEMU_ARGS_COMMON} \
    -chardev stdio,id=stdio,mux=on,signal=off \
    -device virtio-serial-pci -device virtio-net-pci \
```

```
    -device virtconsole,chardev=stdio \
    -net nic,model=virtio \
    -mon chardev=stdio \
     -fsdev local,id=fs1,path=$
{SHARED_HOST_FOLDER},security_model=none \
     -device virtio-9p-pci,fsdev=fs1,mount_tag=host-code &  << run in
background >>

  qpid=$(pidof -s ${QEMUPRCS})
  [ -z "${qpid}" ] && {
   echo "Failed to get PID of ${QEMUPRCS}, abandoning perf run.."
  } || {
   HZ=512
   sudo perf record -p ${qpid} ${PERF_REC_ARGS} --output $
{PERF_RAWDATA_WITHKVM}
  }

<< After the run, perf record saves the raw data file (3 MB) >>


$ sudo perf report -i perf_raw_withkvm.data
Samples: 333  of event 'cycles', Event count (approx.): 1995020328
  Children      Self  Command          Shared Object             Symbol
+   61.10%     0.00%  qemu-system-x86  libc-2.24.so              [.] __clone
+   61.10%     0.00%  qemu-system-x86  libpthread-2.24.so        [.] start_thread
+   59.89%     0.06%  qemu-system-x86  [kernel.kallsyms]         [k] entry_SYSCALL_64_fastpath
-   58.88%     0.00%  qemu-system-x86  qemu-system-x86_64        [.] kvm_cpu_exec
   - kvm_cpu_exec
      - 41.37% kvm_vcpu_ioctl
         - __GI___ioctl
            - 39.20% entry_SYSCALL_64_fastpath
               - 37.77% sys_ioctl
                  - 37.69% do_vfs_ioctl
                     - kvm_vcpu_ioctl
                        - 34.95% kvm_arch_vcpu_ioctl_run
                           - 28.86% vcpu_enter_guest
                              - 20.72% vmx_handle_exit
                                 + 9.98% handle_ept_misconfig
                                 + 3.40% handle_cpuid
                                 + 2.97% handle_ept_violation
                                 + 1.65% handle_apic_access
                                 + 1.36% handle_io
                              + 2.79% vmx_save_host_state
                                1.81% __srcu_read_unlock
                                1.16% __srcu_read_lock
                           + 1.69% kvm_vcpu_block
                             0.86% __srcu_read_lock
                             0.85% vmx_handle_exit
                           + 0.77% complete_emulated_mmio
                           + 0.66% _cond_resched
                        + 2.70% vcpu_put
               + 1.37% syscall_return_slowpath
              0.61% vmx_vcpu_run
      + 17.07% address_space_write
+   58.88%     0.00%  qemu-system-x86  qemu-system-x86_64        [.] 0xffffa9cad3195918
+   42.52%     0.00%  qemu-system-x86  libc-2.24.so              [.] __GI___ioctl
+   41.37%     0.00%  qemu-system-x86  qemu-system-x86_64        [.] kvm_vcpu_ioctl
```

Majority of time spent by qemu-system-x86_64 is in kvm_cpu_exec → kvm_vcpu_ioctl → ...

3.2 WITHOUT KVM

<< First, we make sure the kvm.ko and kvm-[intel|amd].ko kernel modules are removed >>

```
[...]
   qemu-system-x86_64 ${QEMU_ARGS_COMMON} &  << run in background >>

  qpid=$(pidof -s ${QEMUPRCS})
  [ -z "${qpid}" ] && {
   echo "Failed to get PID of ${QEMUPRCS}, abandoning perf run.."
  } || {
   HZ=99
   sudo perf record -p ${qpid} ${PERF_REC_ARGS} --output
    ${PERF_RAWDATA_NOKVM}
  }

[ perf record: Captured and wrote 102.977 MB perf_raw_nokvm.data (12763
samples) ]
$ ls -lh perf_raw_nokvm.data
-rw------- 1 root root 103M Sep 30 12:15 perf_raw_nokvm.data
$
```

We can literally "feel" that it's much much slower!

```
$ sudo perf report -i perf_raw_nokvm.data
```

```
Samples: 12K of event 'cycles', Event count (approx.): 227457053992
  Children      Self  Command          Shared Object             Symbol
+   81.72%     0.00%  qemu-system-x86  libc-2.24.so              [.] __clone
+   81.72%     0.00%  qemu-system-x86  libpthread-2.24.so        [.] start_thread
+   81.14%     5.59%  qemu-system-x86  qemu-system-x86_64        [.] cpu_exec
+   81.14%     0.00%  qemu-system-x86  qemu-system-x86_64        [.] 0xffffaa79cd0ff491
-   61.10%     1.30%  qemu-system-x86  qemu-system-x86_64        [.] tb_gen_code
   - 59.80% tb_gen_code
      - 48.95% tcg_gen_code
           11.41% tcg_optimize
         + 1.09% 0x2d018d
           1.01% 0x2cf01c
           0.85% 0x2cf9f8
           0.56% 0x2cedc9
           0.52% 0x2cf779
           0.51% 0x2cf5a8
      + 8.56% gen_intermediate_code
      + 1.46% qht_insert
   + 1.30% __clone
-   48.95%    20.31%  qemu-system-x86  qemu-system-x86_64        [.] tcg_gen_code
   - 28.64% tcg_gen_code
        11.41% tcg_optimize
      + 1.09% 0x2d018d
        1.01% 0x2cf01c
        0.85% 0x2cf9f8
        0.56% 0x2cedc9
        0.52% 0x2cf779
        0.51% 0x2cf5a8
   + 20.31% __clone
+   13.41%     0.37%  qemu-system-x86  qemu-system-x86_64        [.] _init
+   11.41%    10.12%  qemu-system-x86  qemu-system-x86_64        [.] tcg_optimize
+    8.56%     0.25%  qemu-system-x86  qemu-system-x86_64        [.] gen_intermediate_code
+    7.80%     0.38%  qemu-system-x86  qemu-system-x86_64        [.] get_page_addr_code
+    5.72%     0.02%  qemu-system-x86  qemu-system-x86_64        [.] tlb_fill
+    5.70%     0.34%  qemu-system-x86  qemu-system-x86_64        [.] x86_cpu_handle_mmu_fault
+    5.43%     4.53%  qemu-system-x86  qemu-system-x86_64        [.] qht_lookup
+    3.85%     3.57%  qemu-system-x86  qemu-system-x86_64        [.] iotlb_to_region
```

This time, the majority of time spent by qemu-system-x86_64 is in the ***TCG*** *(Tiny Code Generator – a JIT compiler) code paths: tcg_gen_code → … , implying that it's Qemu's dynamic binary translation engine. Makes sense, as we now can't run guest code directly on the host cpu and instead have to rely on - much slower – binary translation techniques.*

Example 4 : calling fork() / exit() in a loop 60,000 times

```
Samples: 11K of event 'cycles:ppp', Event count (approx.): 10380008305
  Children      Self  Command     Shared Object
-   96.67%    96.67%  fork_test   [kernel.kallsyms]
   - 81.33% __GI___fork (inlined)
      - 69.81% entry_SYSCALL_64_after_hwframe
         - do_syscall_64
            - 69.50% _do_fork
               - 65.97% copy_process.part.35
                  - 17.39% copy_page_range
                     + 5.63% __pte_alloc
                     + 3.11% __pmd_alloc
                     + 2.17% __pud_alloc
                  + 6.98% anon_vma_fork
                  + 6.34% __vmalloc_node_range
                  + 5.94% perf_event_init_task
                  + 3.38% kmem_cache_alloc
                  + 2.66% security_vm_enough_memory_mm
                    1.98% dup_userfaultfd
                  + 1.44% copy_fs_struct
                  + 1.33% arch_dup_task_struct
                  + 1.32% mm_init
                  + 1.18% sched_fork
```

Partial screenshot above: we can see that the majority of the work is in *copy_process()* and within that in the function *copy_page_range();* within that, its the *page table setup* code that takes time.

<< Related Tutorial: [“PERF tutorial: Finding execution hot spots”](#) >>

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.

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.

…

<<

On a Fedora 26 box

perf crashed, with error message “*** Error in `perf': free(): invalid pointer: 0x0000559531a30d20 ***”.
So, we fire up GDB to examine the corefile :

```
$ gdb -q -c
corefile:host=localhost.fc26:gPID=24912:gTID=24912:ruid=1000:sig=6:exe=\!
usr\!bin\!perf.24912 /usr/bin/perf
Reading symbols from /usr/bin/perf...(no debugging symbols found)...done.
[New LWP 24912]
```

```
warning: Unexpected size of section `.reg-xstate/24912' in core file.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Core was generated by `perf report --stdio -T --input=perf.data'.
Program terminated with signal SIGABRT, Aborted.

warning: Unexpected size of section `.reg-xstate/24912' in core file.
#0  0x00007ff95772a69b in raise () from /lib64/libc.so.6
Missing separate debuginfos, use: dnf debuginfo-install perf-4.13.5-
200.fc26.x86_64
  << Interesting! Fedora asks us to install the appropriate dbgsym ver
of perf! >>
(gdb) bt
#0  0x00007ff95772a69b in raise () from /lib64/libc.so.6
#1  0x00007ff95772c4a0 in abort () from /lib64/libc.so.6
#2  0x00007ff9577708e1 in __libc_message () from /lib64/libc.so.6
#3  0x00007ff95777b789 in _int_free () from /lib64/libc.so.6
#4  0x00007ff95777810ee in free () from /lib64/libc.so.6
#5  0x000055a6069265b5 in perf_read_values_destroy ()
#6  0x000055a60687bde1 in cmd_report ()
#7  0x000055a6068dc8b1 in ?? ()
#8  0x000055a6068dcbae in ?? ()
#9  0x000055a606864130 in main ()
(gdb)

>>
```

Another thing that can happen when not running perf as root:

```
$ perf record -F 200 -a sleep 3
WARNING: Kernel address maps (/proc/{kallsyms,modules}) are restricted,
check /proc/sys/kernel/kptr_restrict.

Samples in kernel functions may not be resolved if a suitable vmlinux
file is not found in the buildid cache or in the vmlinux path.

Samples in kernel modules won't be resolved at all.

If some relocation was applied (e.g. kexec) symbols may be misresolved
even with a suitable vmlinux or kallsyms file.

Cannot read kernel map
Error:
You may not have permission to collect system-wide stats.
Consider tweaking /proc/sys/kernel/perf_event_paranoid:
 -1 - Not paranoid at all
  0 - Disallow raw tracepoint access for unpriv
  1 - Disallow cpu events for unpriv
  2 - Disallow kernel profiling for unpriv
$
```

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.

Note!

For compiled code (for development time at least), always use the "`-fno-omit-frame-pointer`" gcc switch !!

Dwarf [1]

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 "`-g dwarf`".

```
Eg. # 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
```

<<
Common Error:

```
# perf record -F 99 -g dwarf -p 2869
Workload failed: No such file or directory
#
```

See this post.
" … That error is telling you have a bad command line. I remember running into this error in the past and basically if perf fails to parse the command line it will give this nondescript "Workload failed: No such file or directory" error in some cases. It's since been fixed in new versions of perf.

In your first example your passing in: `-g dwarf` which is wrong. If you read the man page that comes up when you run perf help report you'll see that the `-g` flag doesn't take any parameters. Instead, you have to specify that after `--call-graph FOO`, like in your second example, which takes the callgraph method as a parameter. Also, `--call-graph FOO` already implies -g. "
Ok, bottom-line: don't use the arg as "`-g dwarf`", instead use "`--call-graph dwarf`" :

```
# perf record -F 99 --call-graph dwarf -p 2869
<< ...wait a few seconds... >>
^C[ perf record: Woken up 154 times to write data ]
[ perf record: Captured and wrote 38.802 MB perf.data (~1695291 samples) ]
#
```
>>

...

If user-level stacks look incomplete, you can try perf record with ~~"-g dwarf"~~ "`--call-graph 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

1 What exactly is the DWARF format and how can one see it? See this article here.

generating Flame Graphs from the same data.

…

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

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

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

```
# 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 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 entire system, at 99 Hertz, for 10 seconds:
perf record -F 99 -ag -- sleep 10

# Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 s:
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
```

Interesting:

Look carefully at the (5^{th} and) 6^{th} example above:

```
perf record -e LLC-load-misses -c 100 -a --call-graph dwarf --
sleep 5
```

LLC: Last Level Cache.
Meaning: for all CPUs system-wide, record stats for 5 sec. Every time we get 100 cache load misses, sample the cpu stack trace. This helps in profling apps that have cache misses. Fine tune for your app by using the -p (PID) switch.

<<
Monitoring software for cache misses can be useful. What perf events are available for this?
Eg. on a laptop with an Intel Core-i7 processor:

```
# perf list|grep -i miss
  cache-misses                                         [Hardware event]
  branch-misses                                        [Hardware event]
  L1-dcache-load-misses                                [Hardware cache event]
  L1-dcache-store-misses                               [Hardware cache event]
  L1-dcache-prefetch-misses                            [Hardware cache event]
  L1-icache-load-misses                                [Hardware cache event]
  dTLB-load-misses                                     [Hardware cache event]
  dTLB-store-misses                                    [Hardware cache event]
  iTLB-load-misses                                     [Hardware cache event]
  branch-load-misses                                   [Hardware cache event]
  branch-misses OR cpu/branch-misses/                  [Kernel PMU event]
  cache-misses OR cpu/cache-misses/                    [Kernel PMU event]
```

```
#
```

Example:

```
# perf stat -e cache-misses vim /etc/passwd

 Performance counter stats for 'vim /etc/passwd':

          2,07,480      cache-misses

       3.042336654 seconds time elapsed

#
```

>>

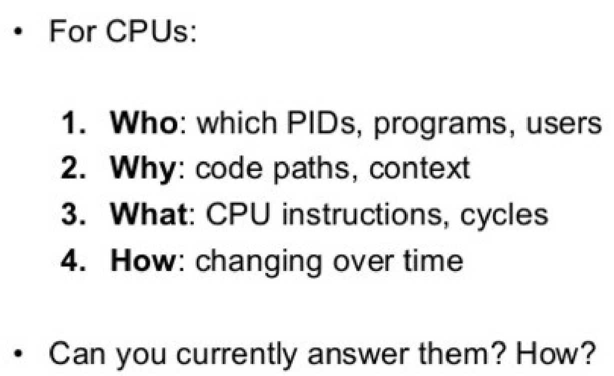
Reporting

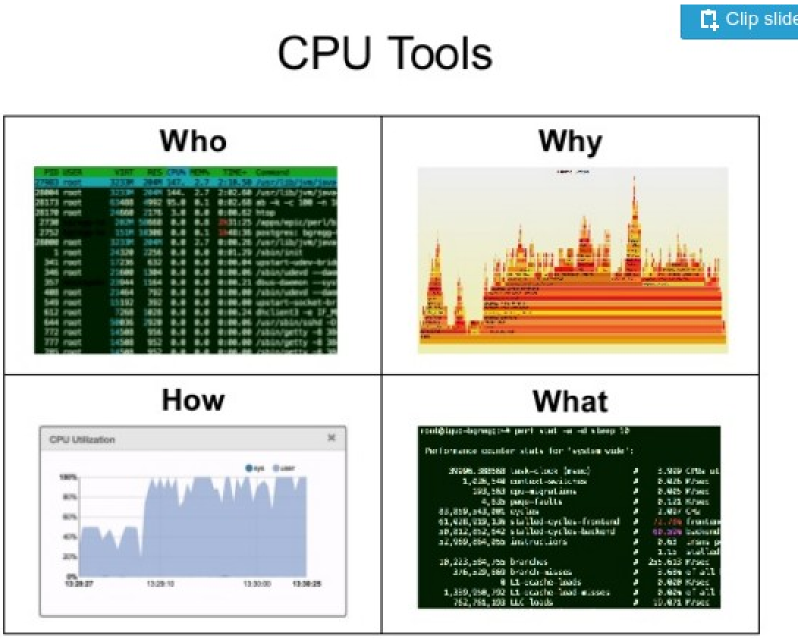
```
# Show perf.data in an ncurses browser (TUI) if possible:
perf report

# 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

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

# List all raw events from perf.data, with customized fields:
perf script -f comm,tid,pid,time,cpu,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
```

Before getting into the how of a FlameGraph:
<< from [here](#) and [here](#) : slides of Brendan Gregg's talk at Xscale, Feb 2015 >>

CPU Workload Characterization

- Who
- Why
- What
- How



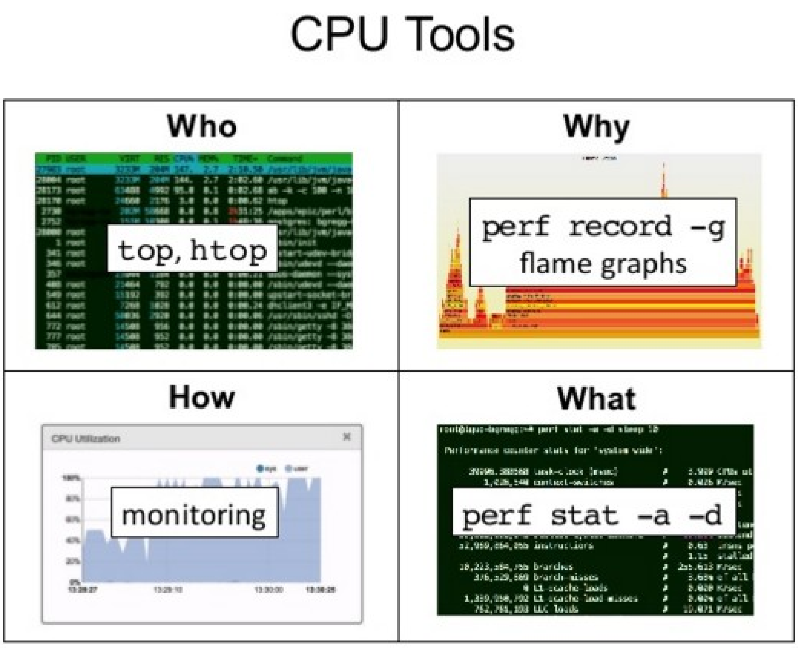


The “Who” is usually easy: tools like top, htop, mpstat, perf top, etc will show you.

The “How” is also relatively easy- lots of system monitoring tools available (sar, nagios, cacti, nmon, sysmon, etc).

The problem areas are usually the “Why” and “What”!

- Who
- Why
- What
- How



CPU Profiling with Flame Graphs

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

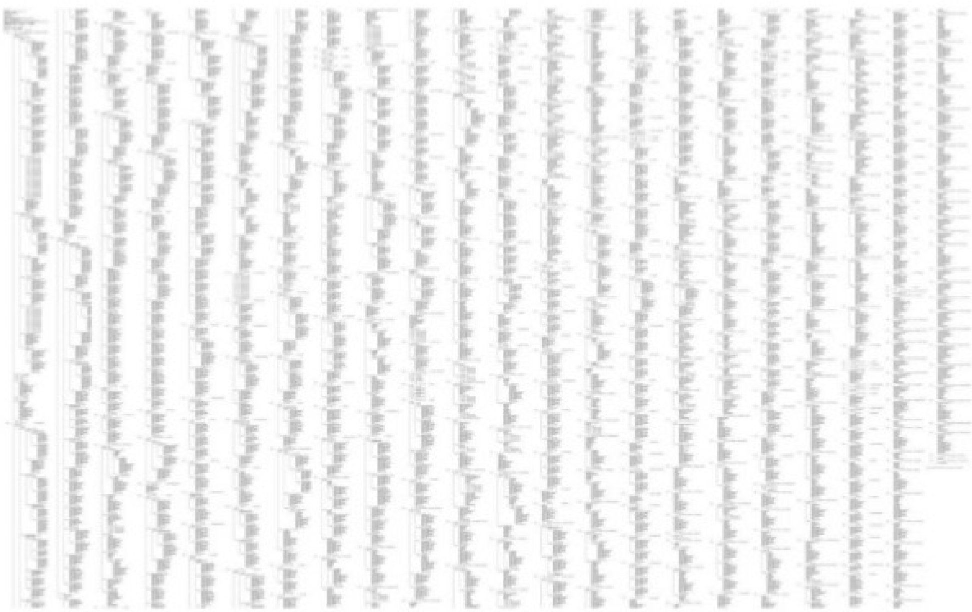
- In a “regular” analysis with perf, you'd sample cpu (perf record …) and then report your findings (perf report …)
- Here's a quick sample:

  - Sampling full stack traces at 99 Hertz:

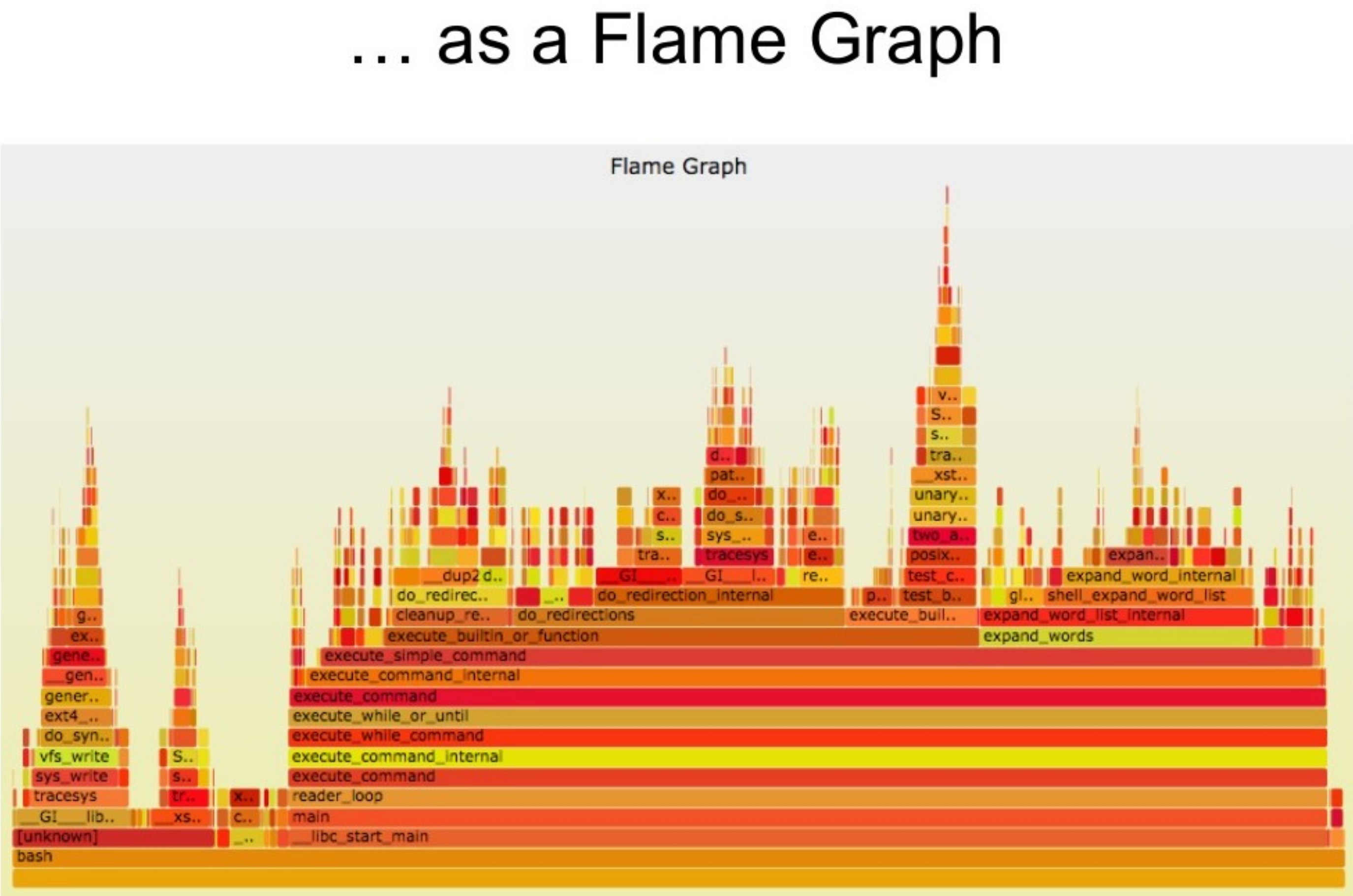
```
# perf record -F 99 -ag -- sleep 30
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.745 MB perf.data (~119930 samples) ]
# perf report -n --stdio
1.40%     162              java   [kernel.kallsyms]        [k] _raw_spin_lock
             |
             --- _raw_spin_lock
                |
                |--63.21%-- try_to_wake_up
                |           |
                |           |--63.91%-- default_wake_function
                |           |           |
                |           |           |--56.11%-- __wake_up_common
                |           |           |           __wake_up_locked
                |           |           |           ep_poll_callback
                |           |           |           __wake_up_common
                |           |           |           __wake_up_sync_key
                |           |           |           |
                |           |           |           |--59.19%-- sock_def_readable
[…78,000 lines truncated…]
```

- Trouble is, on a production server, the ouput of perf report –-stdio could go into tens of thousands of lines! (78k above).
- Around 8000 lines would look like this:

perf report



- That's why we need an effective visualization tool: Flame Graphs!



Flame Graphs: along with Linux perf you get to see the *entire stack.*

Take sampling a typical (large) Java application scenario. With the Flame Grphs visualization method, we get stack traces for *all layers*:

Java app (Java) → JVM (C/C++) → Middleware/Libs (C/C++) → system-level (kernel C) !

Flame Graphs are a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately. They can be generated using my open source programs on github.com/brendangregg/FlameGraph, which create interactive SVGs. See the Updates section for other implementations.

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record -F 99 -a –g -- sleep 30
perf script | ./stackcollapse-perf.pl |./flamegraph.pl > perf.svg
```

- Flame Graphs:
  - **x-axis**: alphabetical stack sort, to maximize merging
  - **y-axis**: stack depth
  - **color**: random, or hue can be a dimension
    - e.g., software type, or difference between two profiles for non-regression testing ("differential flame graphs")
  - interpretation: top edge is on-CPU, beneath it is ancestry
- Just a Perl program to convert perf stacks into SVG
  - Includes JavaScript: open in a browser for interactivity
- Easy to get working

The following pages (or posts) introduce different types of flame graphs:

- CPU
- Memory
- Off-CPU
- Hot/Cold
- Differential

The example on the right is a portion of a CPU flame graph, showing MySQL codepaths that are consuming CPU cycles, and by how much.

Summary

The x-axis shows the stack profile population, sorted alphabetically (it is not the passage of time), and the y-axis shows stack depth. Each rectangle represents a stack frame. The wider a frame is is, the more often it was present in the stacks. The top edge shows what is on-CPU, and beneath it is its ancestry. The colors are usually not significant, picked randomly to differentiate frames.

This visualization is fully introduced and explained in the CPU Flame Graphs page, and in the presentation below.

…

Generating a FlameGraph

1. Install the FlameGraph scripts [from here](#) or do (in an empty folder):

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph.git
```

2. Generate a perf.data sample set

```
perf record -F 99 --call-graph dwarf [-a]|[-p pid]
```

Generates perf.data
[-a => all cpu's; in effect, if specified, the sample is system-wide
-p => sample a particular process.]

3. Convert using perf script:

```
perf script > perfscript_out.dat
```

Generate the "flamegraph" SVG

```
cat perfscript_out.dat | FlameGraph/stackcollapse-perf.pl |
  FlameGraph/flamegraph.pl > out.svg
```

4. Open the SVG in a web browser, move the mouse over stack frames (each rectangle represents a single stack frame: the width is representative of the frequency of the function call, the height is representative of the depth of the stack; the order from left-to-right is just alphabetical, it's *not* a timeline), click on stack frames to zoom into it.

In effect: the hottest code-paths are the widest sections!
The top-edge is the stuff on-CPU; beneath is ancestry.

To zoom out to the whole picture again, click on the "Reset Zoom" hyperlink on the upper-left corner.
Enjoy!

Participants with the [Seawolf pendrive](#) :

Prebuilt scripts will let you generate FlameGraph's very easily.
Just navigate to the `~/kaiwanTECH/FlameGraph` folder, read the `Readme_FlameGraph.txt` there and proceed.

Source: CPU Flame Graphs

...

Description

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

- Each box represents a function in the stack (a "stack frame").
- The **y-axis** shows stack depth (number of frames on the stack). The top box shows the function that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent, just like the stack traces shown earlier.
- The **x-axis** spans the sample population. It does *not* show the passing of time from left to right, as most graphs do. The left to right ordering has no meaning (it's sorted alphabetically).
- The width of the box shows the *total* time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Wider box functions may be slower than narrow box functions, or, they may simply be called more often. The call count is not shown (or known via sampling).
- The sample count can exceed elapsed time if multiple threads were running and sampled concurrently.

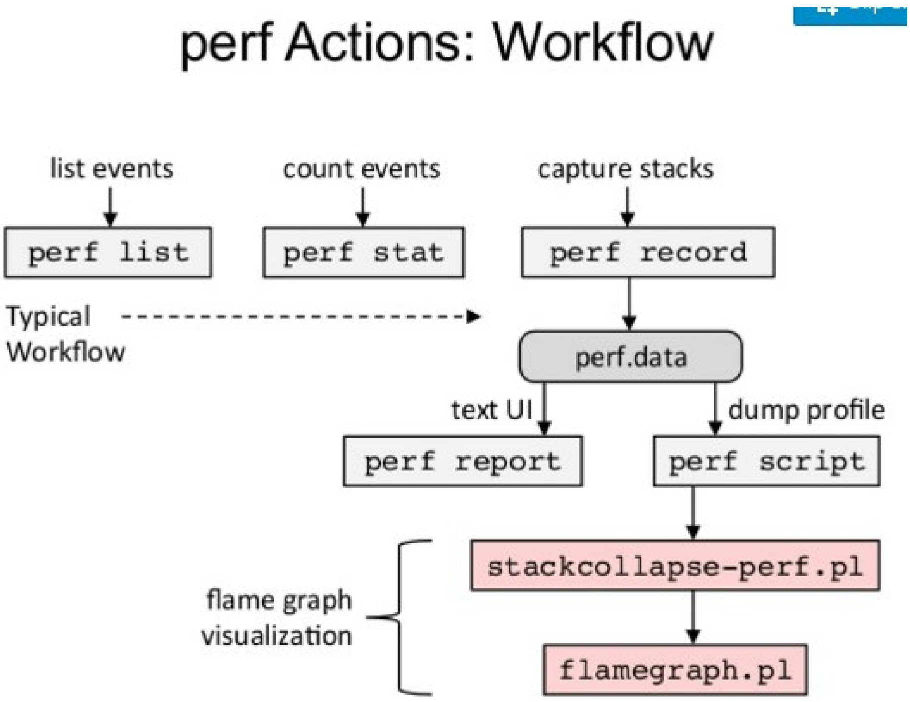
The colors aren't significant, and are usually picked at random to be warm colors (other meaningful palettes are supported). It's called "flame graph" as it's showing what is hot on-CPU. And, it's interactive: mouse over the SVGs to reveal details, and click to zoom.

…

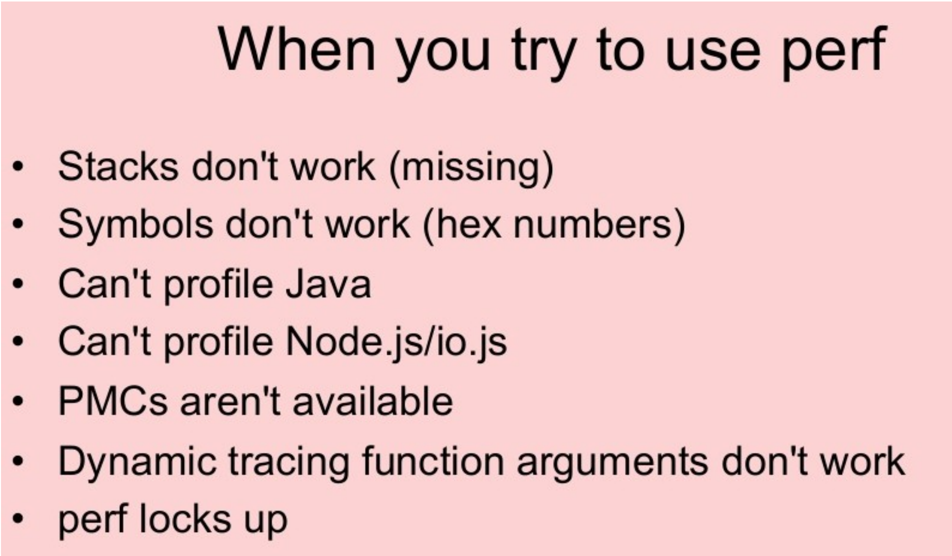
```
# cd FlameGraph
# perf record -F 99 -a -g -- sleep 60
# perf script | ./stackcollapse-perf.pl > out.perf-folded
# ./flamegraph.pl out.perf-folded > perf-kernel.svg
```

The `perf record` command samples at 99 Hertz (-F 99) across all CPUs (-a), capturing stack traces so that a call graph (-g) of function ancestry can be generated later. The samples are saved in a perf.data file, which are read by `perf script`.

…

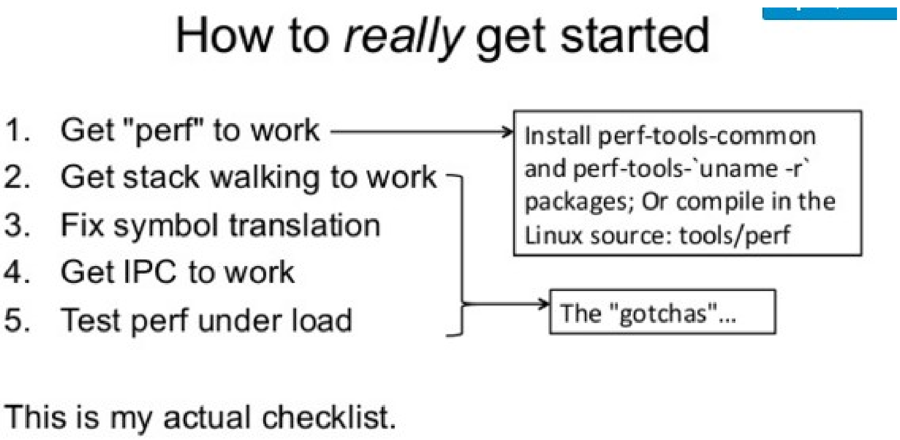


Gotchas!
(see below)



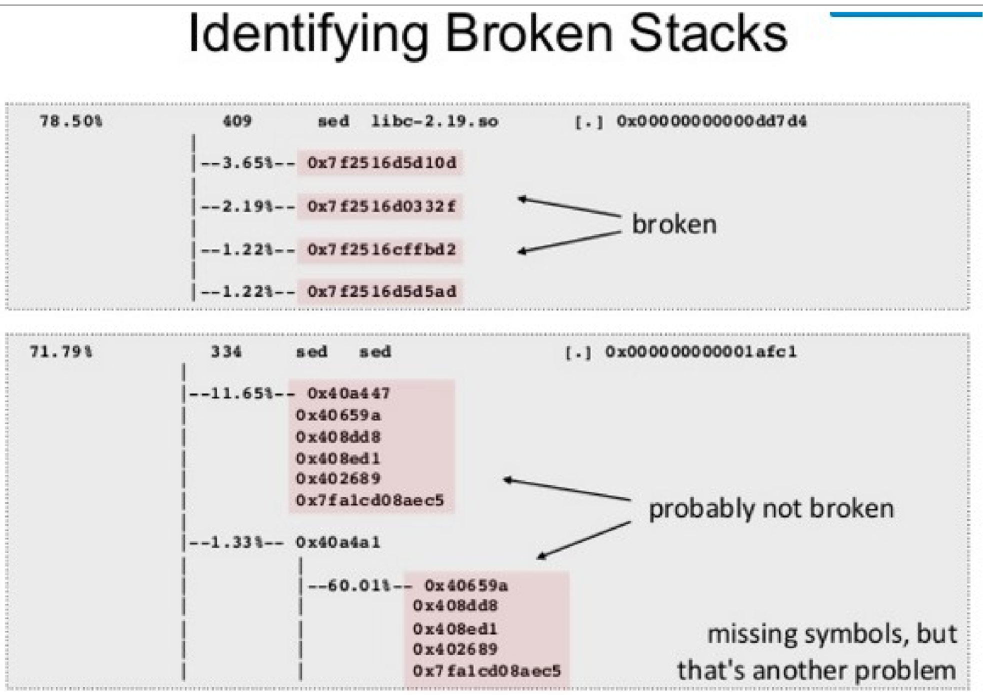
PMC = (cpu) Performance Measurement Counters

The GOTCHA's!

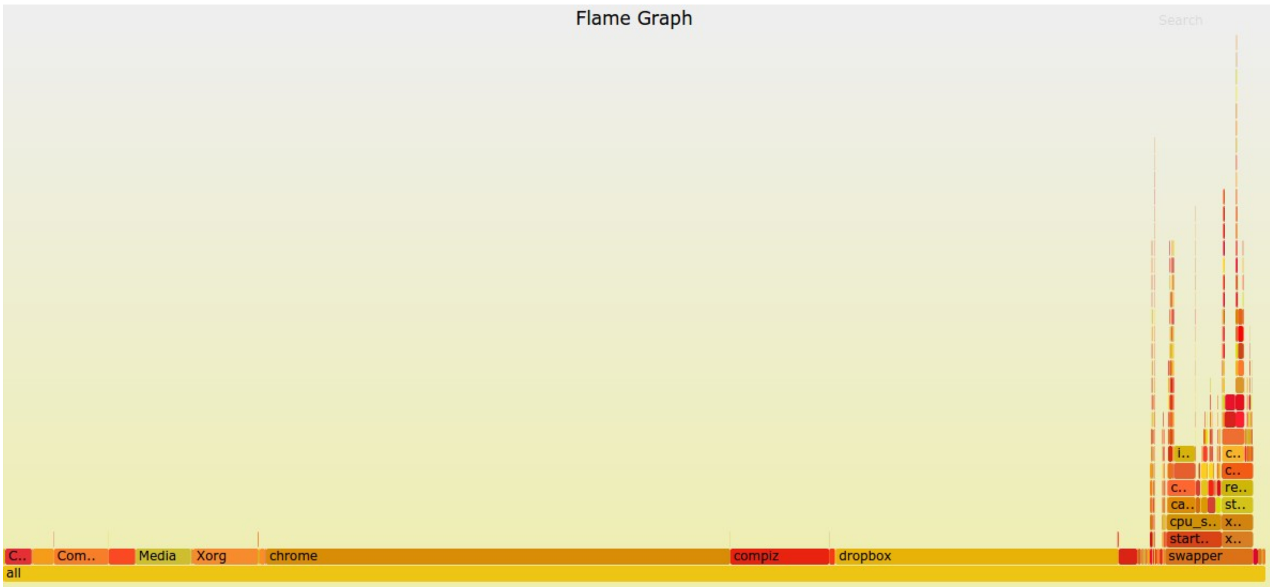


IPC: (cpu) Instructions Per Cycle

Identifying Broken Stacks (1)



Identifying Broken Stacks (2)
Symbols missing here..



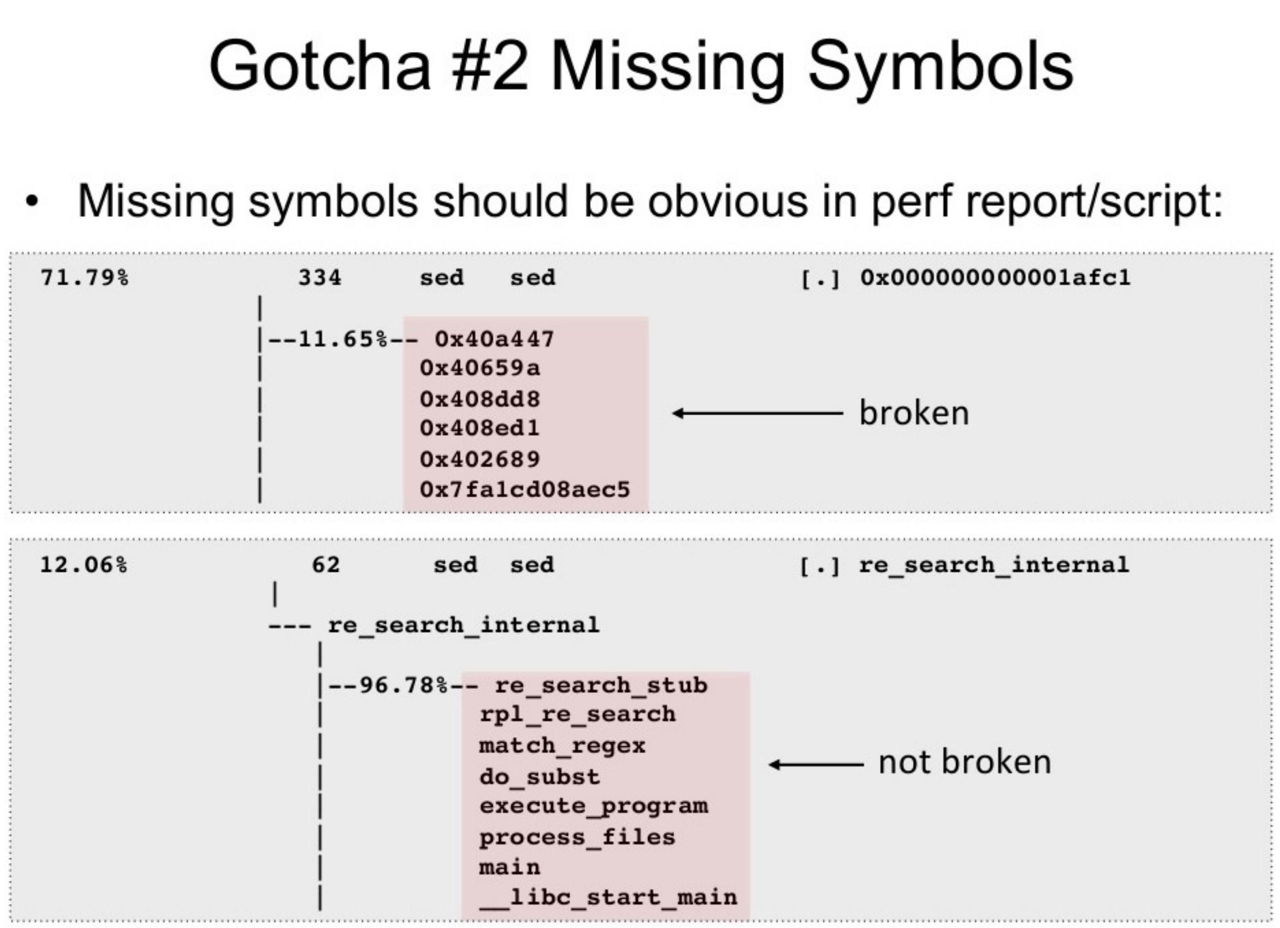
Identifying Broken Stacks (3)

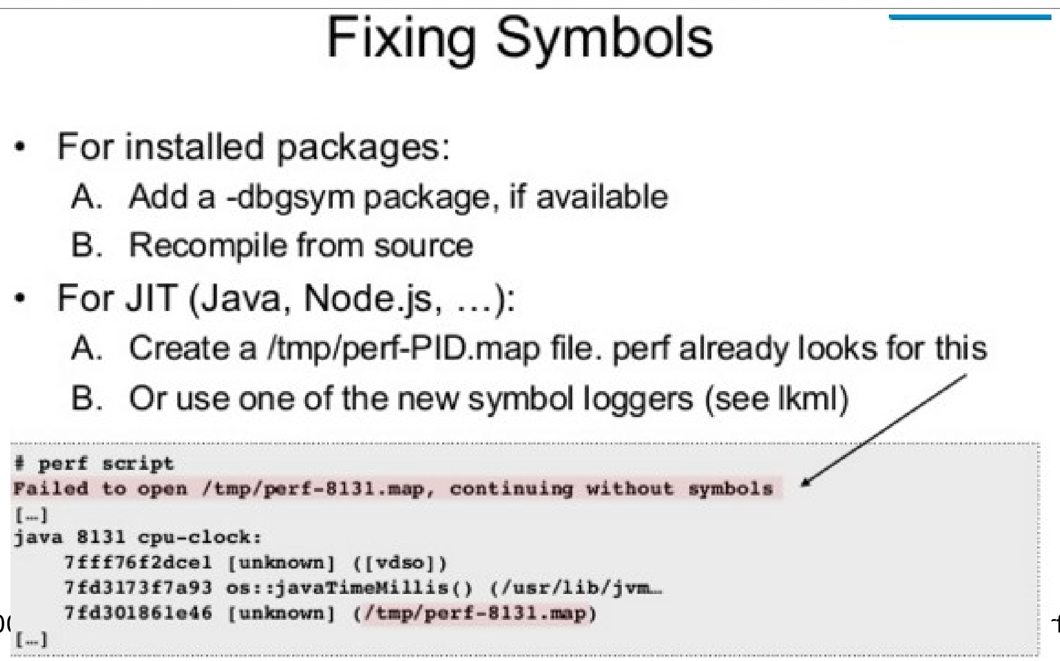
Notice how “low & flat” the Flame Graph shown above is: implies no stack info – broken stacks. Only on the extreme right is there a “tower” implying availability of stack info.

Turns out this was kernel-mode code.

Solution to Gotcha #1 : Broken Stacks

- Compilable code: use the '`-fno-omit-frame-pointer`' gcc switch *[preferred]*
- `perf … --call-graph dwarf` [use DWARF & libunwind]
- custom stack walker (probably needs kernel support, slow)





Compiled code: Use `gcc -g -ggdb` switches to include symbolic info
-dbgsym package for the app / lib (even a debug *vmlinux* kernel image should be available for most platforms).

FYI: Java

perf JIT symbols: Java, Node.js

- Using the /tmp map file for symbol translation:
  - Pros: simple, can be low overhead (snapshot on demand)
  - Cons: stale symbols
  - Map format is "START SIZE symbolname"
  - Another gotcha: If perf is run as root, chown root <mapfile>
- Java
  - https://github.com/jrudolph/perf-map-agent
  - Agent attaches and writes the map file on demand (previous versions attached on Java start, and wrote continually)
- Node.js
  - `node --perf_basic_prof` writes the map file continually
  - Available from 0.11.13+
  - Currently has a file growth issue; see my patch in https://code.google.com/p/v8/issues/detail?id=3453

Gotcha #3 Guest PMCs

- Using PMCs from a Xen guest (currently):

```
# perf stat -a -d sleep 5

 Performance counter stats for 'system wide':

      10003.718595 task-clock (msec)       #    2.000 CPUs utilized       [100.00%]
               323 context-switches        #    0.032 K/sec               [100.00%]
                17 cpu-migrations          #    0.002 K/sec               [100.00%]
               233 page-faults             #    0.023 K/sec
   <not supported> cycles
   <not supported> stalled-cycles-frontend
   <not supported> stalled-cycles-backend
   <not supported> instructions
   <not supported> branches
   <not supported> branch-misses
   <not supported> L1-dcache-loads
   <not supported> L1-dcache-load-misses
   <not supported> LLC-loads
   <not supported> LLC-load-misses

       5.001607197 seconds time elapsed
```

Virtual Machines: they could, but most do not as of today, implement CPU PMCs (Performance Measurement Counters). Minimally require: IPC (Instructions Per Cycle) Workaround: making use of MSRs (Model Specific Registers): usually are implemented in VMs, and can help.

More examples explained
…

(More) 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:

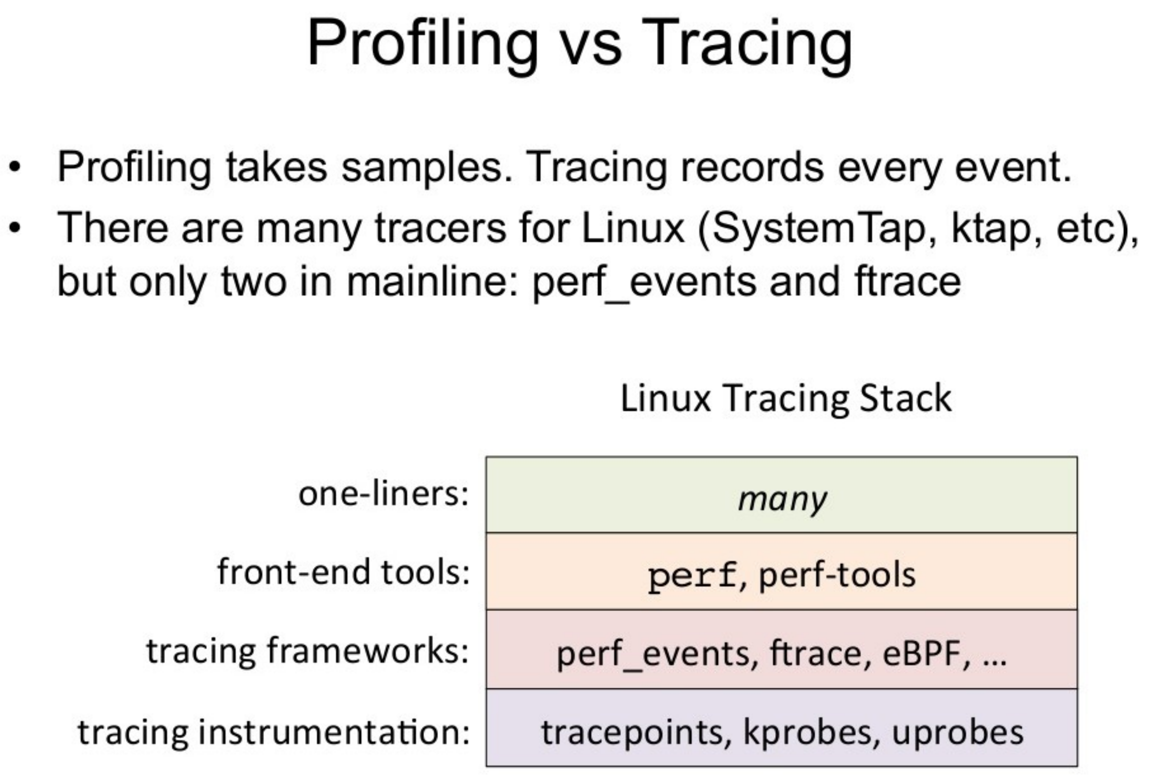
```
# perf record -F 99 -ag -- sleep 5
# perf report
```


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.

...



Tracing Example

```
# perf record -e block:block_rq_insert -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.172 MB perf.data (~7527 samples) ]

# perf script
[…]
  java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 () 4783360 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783448 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783536 + 24 [java]
  java 9940 [000] 1199510.065195: block_rq_insert: 202,1 R 0 () 4864088 + 88 [java]
[…]
```

process PID [CPU] timestamp: eventname: format string

```
include/trace/events/block.h: java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 ()
4783360 + 88 [java]
DECLARE_EVENT_CLASS(block_rq,
[...]
 TP_printk("%d,%d %s %u (%s) %llu + %u [%s]",
                  MAJOR(__entry->dev), MINOR(__entry->dev),
                  __entry->rwbs, __entry->bytes, __get_str(cmd),
                  (unsigned long long)__entry->sector,
                  __entry->nr_sector, __entry->comm)
```

kernel source may be the only docs

More Examples

One-Liners: Static Tracing

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

# Trace all context-switches with stack traces, for 1 second:
perf record -e context-switches —ag -- sleep 1

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

Perf – Dynamic Tracing

One can add tracepoints dynamically via perf probe!

This works on any functions within the kernel, except for those marked as 'static'.
Even works for functions that are private (that are not exported via EXPORT_SYMBOL macro).

One-Liners: Dynamic Tracing

```
# Add a tracepoint for the kernel tcp_sendmsg() function entry (--add 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 avail vars for the tcp_sendmsg(), 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

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

# Add a tracepoint for do_sys_open() with the filename as a string (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
```

Eg.1 : Kernel function: ip_rcv

```
# perf probe --add ip_rcv
Added new event:
  probe:ip_rcv         (on ip_rcv)

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

      perf record -e probe:ip_rcv -aR sleep 1

# perf probe -l
  probe:ip_rcv         (on ip_rcv)
#
```

Eg.2 : Libc function: malloc

```
# perf probe -x /lib/x86_64-linux-gnu/libc-2.21.so malloc
  << -x : -x, --exec=PATH
           Specify path to the executable or shared library file for user
space tracing.
           Can also be used with --funcs option.
>>
Added new events:
  probe_libc:malloc    (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_1  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_2  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_3  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_4  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_5  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_6  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_7  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_8  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_9  (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_10 (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_11 (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_12 (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_13 (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)
  probe_libc:malloc_14 (on malloc in /lib/x86_64-linux-gnu/libc-2.21.so)

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

      perf record -e probe_libc:malloc_14 -aR sleep 1

#
# perf record -e probe_libc:malloc_1 -aR --call-graph dwarf sleep 3
[ perf record: Woken up 2443 times to write data ]
Warning:
Processed 188146 events and lost 945 chunks!

Check IO/CPU overload!

[ perf record: Captured and wrote 1332.009 MB perf.data (164998 samples) ]
# l perf.data
-rw------- 1 root root 1.4G Dec 23 11:55 perf.data
#
```

```
# perf report
Samples: 164K of event 'probe_libc:malloc_1', Event count (approx.): 164998
  Children      Self  Command          Shared Object                  Symbol
-   83.37%    83.37%  dropbox          libc-2.21.so                   [.] malloc
   - malloc
      + 42.42% 0x53a707
      + 33.49% realloc
      + 8.44% PyList_New
      + 6.46% 0x4c8f99
      + 3.20% _nl_make_l10nflist
      + 1.96% PyString_FromStringAndSize
      + 1.43% 0x45e921
      - 0.79% CRYPTO_malloc
         - EVP_MD_CTX_copy_ex
            + 50.05% 0x4ff3f5
            + 49.95% 0x4ff82c
+   82.54%     0.00%  dropbox          dropbox                        [.] PyEval_EvalCodeEx
+   82.54%     0.00%  dropbox          dropbox                        [.] PyObject_Call
+   82.54%     0.00%  dropbox          dropbox                        [.] PyEval_EvalFrameEx
+   76.73%     0.00%  dropbox          dropbox                        [.] 0x00000000001e15c5
+   58.16%     0.00%  dropbox          dropbox                        [.] 0x0000000000206a5f
+   41.75%     0.00%  dropbox          dropbox                        [.] 0x0000000000206990
+   40.70%     0.00%  dropbox          dropbox                        [.] 0x00000000001f985a
+   35.37%     0.00%  dropbox          dropbox                        [.] 0x000000000010f1b9
+   35.37%     0.00%  dropbox          dropbox                        [.] 0x000000000013a707
+   34.74%     0.00%  dropbox          _functools.so                  [.] 0x0000000000001971
+   27.92%     0.00%  dropbox          libc-2.21.so                   [.] realloc
+   26.55%     0.00%  dropbox          dropbox                        [.] 0x0000000000080a2a
+   26.17%     0.00%  dropbox          dropbox                        [.] 0x0000000000187c50
+   25.83%     0.00%  dropbox          dropbox                        [.] PyObject_CallFunctionObjArgs
```

Tip:
`perf report -stdio`
to see everything at once with expanded stack traces.

Brendan Gregg's perf-tools-unstable

Interesting:
Brendan Gregg has developed useful and powerful bash script wrappers over ftrace, kprobes, etc. Install the '`perf-tools-unstable`' package to try them out (*pre-installed on the Seawolf virtual appliance).*

They come with man pages too! :-)

Git repo.

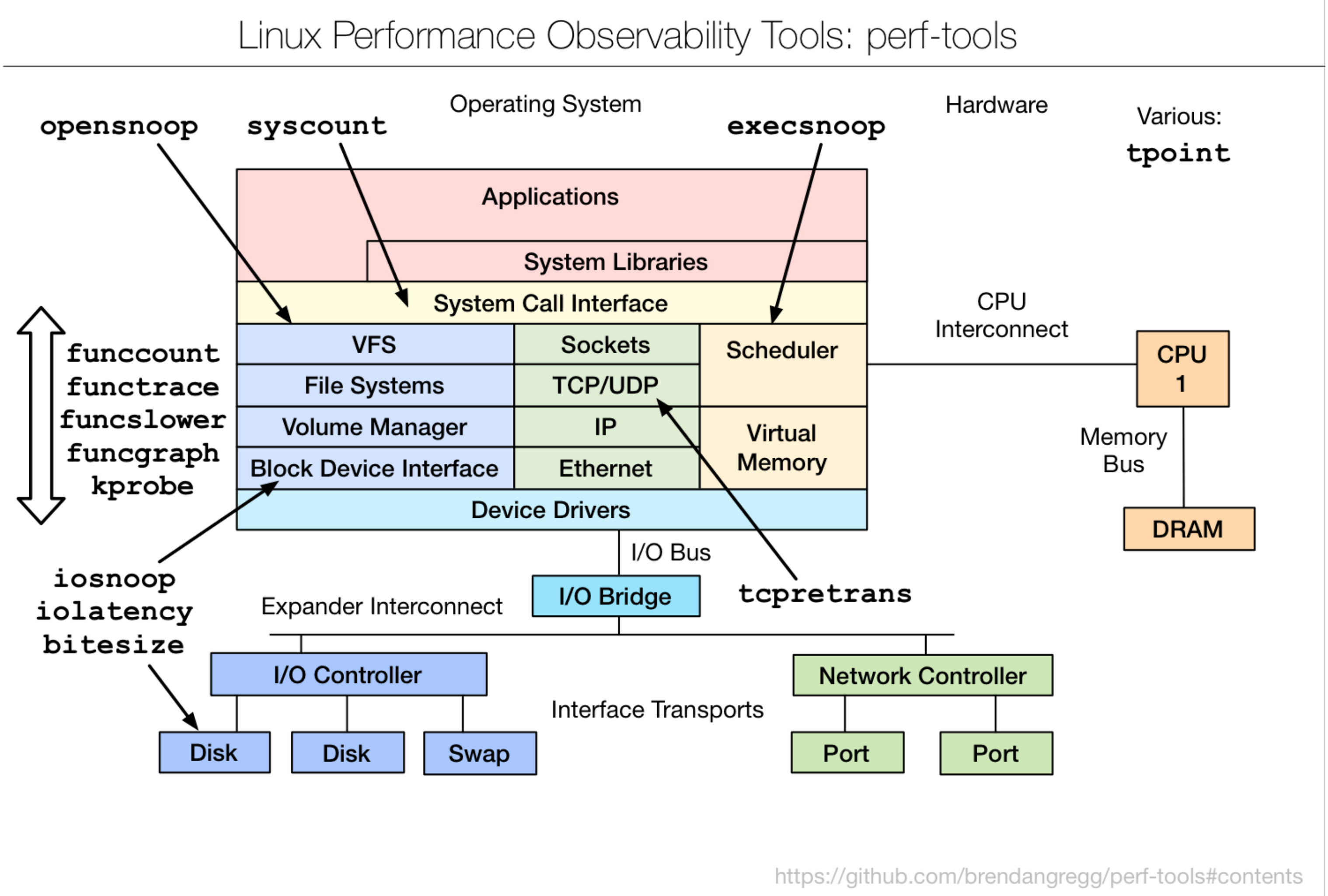
```
$ dpkg -L perf-tools-unstable
/.
/usr
/usr/bin
/usr/bin/bitesize
/usr/bin/cachestat
/usr/bin/execsnoop
/usr/bin/funccount
/usr/bin/funcgraph
```

```
/usr/bin/funcslower
/usr/bin/functrace
/usr/bin/iolatency
/usr/bin/iosnoop
/usr/bin/killsnoop
/usr/bin/kprobe
/usr/bin/opensnoop
/usr/bin/perf-stat-hist
/usr/bin/reset-ftrace
/usr/bin/syscount
/usr/bin/tcpretrans
/usr/bin/tpoint
/usr/bin/uprobe
...
$
```

(On some installations, the script name is suffixed with -perf).

!WARNING! Many of these tools are considered to be Experimental and Unstable! Should only be used on kernel ver >= 4.0.

Source
Linux Performance Observability Tools: Perf Tools, Brendan Gregg



!WARNING! Many of these tools are considered to be Experimental and Unstable!

DTrace for Linux 2016, Oct 2016
or
BPF (Berkeley Packet Filter) Tracing

Linux 4.9 on now has advanced tracing capabilities, similar to Solaris's dtrace.

"… On Linux, you can now analyze the performance of applications and the kernel using production-safe low-overhead custom tracing, with latency histograms, frequency counts, and more."
- eBPF (enhanced Berkeley Packet Filter) timed sampling now merged into 4.9-rc1.

Brendan Gregg

Talks

Velocity 2015

At Velocity 2015, I gave a 90 minute tutorial on Linux performance tools, summarizing performance observability, benchmarking, tuning, static performance tuning, and tracing tools. I also covered performance methodology, and included some live demos. This should be useful for everyone working on Linux systems.

A video of the talk is on youtube (playlist; part 1, part 2) and the slides are on slideshare or as a PDF.

This was similar to my SCaLE11x and LinuxCon talks, however, with 90 minutes I was able to cover more tools and methodologies, making it the most complete tour of the topic I've done. I also posted about it on the Netflix Tech Blog.

SCALE13x (2015)

At the Southern California Linux Expo (SCALE 13x), I << ***Brendan Gregg*** >> gave a talk on Linux Profiling at Netflix using perf_events (aka "perf"), covering CPU profiling and a tour of other features. This talk includes a crash course on perf_events that you may find useful, plus it covers gotchas such as fixing stack traces and symbols when profiling Java and Node.js.

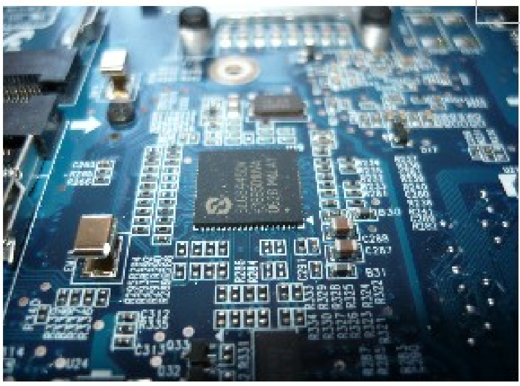
A video of the talk is on youtube, and the slides are on slideshare.

In a post about this talk, I included the interactive CPU flame graph SVG I was demonstrating.

…

<<
Participants are *strongly encouraged* to see Brendan Gregg's video's!
The SCALE13x video focusses on perf.
>>

Linux Operating System Specialized



The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! *Linux OS for Technical Managers*

Please do visit our website for details:
http://kaiwantech.in