Using Linux `perf` at Netflix

Brendan Gregg
Senior Performance Architect
Sep 2017
Case Study: ZFS is eating my CPU

- Easy to debug using Netflix Vector & flame graphs
- How I expected it to look:
Case Study: ZFS is eating my CPU (cont.)

• How it really looked:

Application (truncated)

38% kernel time (why?)
Case Study: ZFS is eating my CPU (cont.)

- ZFS ARC (adaptive replacement cache) reclaim.
- But... ZFS is not in use. No pools, datasets, or ARC buffers.
- CPU time is in random entropy, picking which (empty) list to evict.

Bug: https://github.com/zfsonlinux/zfs/issues/6531
REGIONS WHERE NETFLIX IS AVAILABLE
Agenda

1. Why Netflix Needs Linux Profiling
2. perf Basics
3. CPU Profiling & Gotchas
   - Stacks (gcc, Java)
   - Symbols (Node.js, Java)
   - Guest PMCs
   - PEBS
   - Overheads
4. perf Advanced
1. Why Netflix Needs Linux Profiling
Understand CPU usage quickly and completely
Quickly

Eg, Netflix Vector (self-service UI):

Flame Graphs
Heat Maps
Completely
Why Linux perf?

• Available
  – Linux, open source

• Low overhead
  – Tunable sampling, ring buffers

• Accurate
  – Application-basic samplers don't know what's really RUNNING; eg, Java and epoll

• No blind spots
  – See user, library, kernel with CPU sampling
  – With some work: hardirqs & SMI as well

• No sample skew
  – Unlike Java safety point skew
Why is this so important

• We typically scale microservices based on %CPU
  – Small %CPU improvements can mean big $avings

• CPU profiling is used by many activities
  – Explaining regressions in new software versions
  – Incident response
  – 3rd party software evaluations
  – Identify performance tuning targets
  – Part of CPU workload characterization

• perf does lots more, but we spend ~95% of our time looking at CPU profiles, and 5% on everything else
  – With new BPF capabilities (off-CPU analysis), that might go from 95 to 90%
CPU profiling should be easy, but…

- JIT runtimes
- no frame pointers
- no debuginfo
- stale symbol maps
- container namespaces
...
2. perf Basics
perf (aka "perf_events")

• The official Linux profiler
  – In the linux-tools-common package
  – Source code & docs in Linux: tools/perf

• Supports many profiling/tracing features:
  – CPU Performance Monitoring Counters (PMCs)
  – Statically defined tracepoints
  – User and kernel dynamic tracing
  – Kernel line and local variable tracing
  – Efficient in-kernel counts and filters
  – Stack tracing, libunwind
  – Code annotation

• Some bugs in the past; has been stable for us
# 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
- **c2c**: Shared Data C2C/HITM Analyzer.
- **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
- **ftrace**: Simple wrapper for kernel's ftrace functionality
- **inject**: Filter to augment the events stream with additional information
- **kallsyms**: Searches running kernel for symbols
- **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.
perf Basic Workflow

1. list -> find events
2. stat -> count them
3. record-> write event data to file
4. report -> browse summary
5. script -> event dump for post processing
Basic Workflow Example

1. found an event of interest
2. 19 per 10 sec is a very low rate, so safe to record
3. 21 samples captured
4. summary style may be sufficient, or,
5. script output in time order
perf stat/record Format

- These have three main parts: action, event, scope.
- e.g., profiling on-CPU stack traces:

```bash
perf record -F 99 -a -g -- sleep 10
```

**Action:** record stack traces

**Event:** 99 Hertz

**Scope:** all CPUs

Note: sleep 10 is a dummy command to set the duration
perf Actions

• Count events (perf stat ...)  
  – Uses an efficient in-kernel counter, and prints the results

• Sample events (perf record ...)  
  – Records details of every event to a dump file (perf.data)  
    • Timestamp, CPU, PID, instruction pointer, ...
  – This incurs higher overhead, relative to the rate of events
  – Include the call graph (stack trace) using -g

• Other actions include:  
  – List events (perf list)
  – Report from a perf.data file (perf report)
  – Dump a perf.data file as text (perf script)
  – top style profiling (perf top)
perf Events

- **Custom timers**
  - e.g., 99 Hertz (samples per second)

- **Hardware events**
  - CPU Performance Monitoring Counters (PMCs)

- **Tracepoints**
  - Statically defined in software

- **Dynamic tracing**
  - Created using uprobes (user) or kprobes (kernel)
  - Can do kernel line tracing with local variables (needs kernel debuginfo)
perf Events: Map

Dynamic Tracing

uprobes

kprobes

Tracepoints

ext4:

Operating System

Applications

System Libraries

System Call Interface

VFS

File Systems

TCP/UDP

Sockets

sched:
signal:
timer:
workqueue:
syscalls:

CPU Interconnect

Volume Manager

IP

Block Device Interface

Ethernet

Virtual Memory

Device Drivers

Software Events

jbd2:

block:

scsi:

net:

skb:

cpu-clock
cs migrations

cycle faults

page-faults

minor-faults

major-faults

PMCs

cycles
instructions
branch-*

L1-**

L2-**

kmem:
vmscan:
writeback:

irq:

Memory Bus

DRAM

mem-load
mem-store

http://www.brendangregg.com/perf.html 2017
perf Events: List

```plaintext
# 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]
  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]
  L1-dcache-loads [Hardware cache event]
  L1-dcache-load-misses [Hardware cache event]
  L1-dcache-stores [Hardware cache event]
  skb:kfree_skb [Tracepoint event]
  skb:consume_skb [Tracepoint event]
  skb:skb_copy_datagram_iovec [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]
```

perf Scope

- System-wide: all CPUs (-a)
- Target PID (-p PID)
- Target command (…)
- Specific CPUs (-c …)
- User-level only (<event>:u)
- Kernel-level only (<event>:k)
- A custom filter to match variables (--filter …)
- This cgroup (container) only (--cgroup …)
One-Liners: Listing Events

# Listing all currently known events:
perf list

# Searching for "sched" tracepoints:
perf list | grep sched

# Listing sched tracepoints:
perf list 'sched:*

Dozens of perf one-liners:
http://www.brendangregg.com/perf.html#OneLiners
### One-Liners: Counting Events

- **CPU counter statistics for the specified command:**
  ```
  perf stat command
  ```

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

- **Detailed CPU counter statistics for the specified PID, until Ctrl-C:**
  ```
  perf stat -dp PID
  ```

- **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 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
  ```
One-Liners: Profiling Events

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

# 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 stacks, once every 10,000 Level 1 data cache misses, for 5 secs:
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 secs:
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
One-Liners: Reporting

```bash
# 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 debuginfo):
perf annotate --stdio
```
3. CPU Profiling
CPU Profiling

- Record stacks at a timed interval: simple and effective
  - Pros: Low (deterministic) overhead
  - Cons: Coarse accuracy, but usually sufficient
perf Record

```
# 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...]
```

Sampling full stack traces at 99 Hertz
perf Reporting

- perf report summarizes by combining common paths
- Previous output truncated 78,000 lines of summary
- The following is what a mere 8,000 lines looks like...
perf report
... as a Flame Graph
Flame Graphs

- 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 [http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html](http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html)
flamegraph.pl Options

$ flamegraph.pl --help
USAGE: flamegraph.pl [options] infile > outfile.svg

--title TEXT   # change title text
--subtitle TEXT # second level title (optional)
--width NUM    # width of image (default 1200)
--height NUM   # height of each frame (default 16)
--minwidth NUM # omit smaller functions (default 0.1 pixels)
--fonttype FONT # font type (default "Verdana")
--fontsize NUM  # font size (default 12)
--countname TEXT # count type label (default "samples")
--nametype TEXT # name type label (default "Function:"
--colors PALETTE # set color palette. choices are: hot (default), mem,
                 #  io, wakeup, chain, java, js, perl, red, green, blue,
                 #  aqua, yellow, purple, orange
--hash          # colors are keyed by function name hash
--cp            # use consistent palette (palette.map)
--reverse       # generate stack-reversed flame graph
--inverted      # icicle graph
--negate        # switch differential hues (blue<->red)
--notes TEXT    # add notes comment in SVG (for debugging)
--help          # this message

eg,
flamegraph.pl --title="Flame Graph: malloc()" trace.txt > graph.svg
perf Flame Graph Workflow (Linux 2.6+)

Typical Workflow:
- **list events**: `perf list`
- **count events**: `perf stat`
- **capture stacks**: `perf record` → `perf.data`
  - text UI: `perf report`
  - dump profile: `perf script` → `stackcollapse-perf.pl`

flame graph visualization:
- `flamegraph.pl`
perf Flame Graph Workflow (Linux 4.5+)

Typical Workflow

- **list events**
  - `perf list`

- **count events**
  - `perf stat`

- **capture stacks**
  - `perf record`

  - `perf.data`

- **summary**
  - `perf report -g folded`

- **dump**
  - `perf script`

- **text UI**
  - `perf report`

- **flame graph visualization**
  - `awk`
  - `flamegraph.pl`
Flame Graph Optimizations

Linux 2.6
- capture stacks
  - `perf record`
  - `write samples`
  - `perf.data`
  - `read samples`
  - `perf script`
  - `write text`
  - `stackcollapse-perf.pl`
  - `folded output`
  - `flamegraph.pl`

Linux 4.5
- capture stacks
  - `perf record`
  - `write samples`
  - `perf.data`
  - `read samples`
  - `perf report -g folded`
  - `folded report`
  - `awk`
  - `folded output`
  - `flamegraph.pl`

Linux 4.9
- count stacks (BPF)
  - `profile.py`
  - `not perf`
  - `folded output`
  - `flamegraph.pl`
Gotchas
When we've tried to use perf

- Stacks don't work (missing)
- Symbols don't work (hex numbers)
- Instruction profiling looks bogus
- PMCs don't work in VM guests
- Container break things
- Overhead is too high
How to *really* get started

1. Get "perf" to work
2. Get stack walking to work
3. Fix symbol translation
4. Get IPC to work
5. Test perf under load

Install perf-tools-common and perf-tools-`uname -r` packages;
Or compile in the Linux source: tools/perf

The "gotchas"...
Gotcha #1 Broken Stacks

```
perf record -F 99 -a -g -- sleep 30
perf report -n --stdio
```

1. Take a CPU profile
2. Run perf report
3. If stacks are often < 3 frames, or don't reach "thread start" or "main", they are probably broken. Fix them.
Identifying Broken Stacks

28.10% | 146 | sed libc-2.19.so | [.] re_search_internal

--- re_search_internal

--96.78% -- 0x3

broken

--12.25% -- 0x100007

not broken

28.10% | 146 | sed libc-2.19.so | [.] re_search_internal

--- re_search_internal

--96.78% -- 0x3

broken

--12.25% -- 0x100007

not broken
Identifying Broken Stacks

- 78.50% 409 sed libc-2.19.so [.] 0x00000000000dd7d4
  - 3.65% 0x7f2516d5d10d
  - 2.19% 0x7f2516d0332f
  - 1.22% 0x7f2516c9fbd2
  - 1.22% 0x7f2516d555ad

- 11.65% 0x40a4a47
  - 0x40659a
  - 0x406dd8
  - 0x408ed1
  - 0x402689
  - 0x7f1cd08aec5

- 3.65% 0x7f2516d5d10d

- 2.19% 0x7f2516d0332f

- 1.22% 0x7f2516c9fbd2

- 1.22% 0x7f2516d555ad

- 1.33% 0x40a4a41

- 60.01% 0x40659a

broken

probably not broken

missing symbols, but that's another problem
Broken Stacks Flame Graph

Broken Java stacks
(missing frame pointer)

Java == green
system == red
C++ == yellow
Fixing Broken Stacks

• Either:
  • Fix frame pointer-based stack walking (the default)
    – Pros: simple, supports any system stack walker
    – Cons: might cost a little extra CPU to make available
  • Use libunwind and DWARF: perf record -g dwarf
    – Pros: more debug info
    – Cons: not on older kernels, and inflates instance size
    – ... there's also ORC on the latest kernel

• Application support
  – https://github.com/jvm-profiling-tools/async-profiler

• Our current preference is (A), but (C) is also promising
  – So how do we fix the frame pointer...
Once upon a time, x86 had fewer registers, and the frame pointer register was reused for general purpose to improve performance. This breaks system stack walking.

gcc provides -fno-omit-frame-pointer to fix this

– Please make this the default in gcc!
Java -XX:+PreserveFramePointer

• I hacked frame pointers in the JVM (JDK-8068945) and Oracle rewrote it as -XX:+PreserveFramePointer. Lets perf do FP stack walks of Java.

--- openjdk8clean/hotspot/src/cpu/x86/vm/macroAssembler_x86.cpp 2014-03-04…
+++ openjdk8/hotspot/src/cpu/x86/vm/macroAssembler_x86.cpp 2014-11-07 …
@@ -5236,6 +5236,7 @@
 // We always push rbp, so that on return to interpreter rbp, will be
 // restored correctly and we can correct the stack.
 push(rbp);
+  mov(rbp, rsp);
// Remove word for ebp
framesize -= wordSize;

--- openjdk8clean/hotspot/src/cpu/x86/vm/cl_MacroAssembler_x86.cpp …
+++ openjdk8/hotspot/src/cpu/x86/vm/cl_MacroAssembler_x86.cpp …
[++]

• Costs some overhead to use. Usually <1%. Rare cases 10%.
Broken Java Stacks

• Check with "perf script" to see stack samples
• These are 1 or 2 levels deep (junk values)
Fixed Java Stacks

- With -XX:+PreserveFramePointer stacks are full, and go all the way to start_thread()
- This is what the CPUs are really running: inlined frames are not present
Fixed Stacks Flame Graph

Java (no symbols)
Gotcha #2 Missing Symbols

- Missing symbols should be obvious in perf report/script:

```
71.79% 334  sed  sed  [.] 0x000000000001afcl

--11.65-- 0x40a447
0x40659a
0x408dd8
0x408ed1
0x402689
0x7fa1cd08aec5

broken
```

```
12.06%  62  sed  sed  [.]  re_search_internal

--- re_search_internal

--96.78-- re_search_stub
rpl_re_search
match_regex
do_subst
execute_program
process_files
main
_libc_start_main

not broken
```
Fixing Symbols

• For installed packages:
  A. Add a -dbgsym package, if available
  B. Recompile from source

• For JIT (Java, Node.js, ...):
  A. Create a /tmp/perf-PID.map file. perf already looks for this
     • Map format is "START SIZE symbolname"
  B. Or use a symbol loggers. Eg tools/perf/jvmti.

```bash
# perf script
Failed to open /tmp/perf-8131.map, continuing without symbols
 [...] java 8131 cpu-clock:
   7fff76f2dce1 [unknown] ([vdso])
   7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm...
   7fd301861e46 [unknown] (/tmp/perf-8131.map)
 [...]```
Java Symbols

• perf-map-agent
  – Agent attaches and writes the map file on demand (previous versions attached on Java start, and wrote continually)
    (was https://github.com/jrudolph/perf-map-agent)

• Automation: jmaps
  – We use scripts to find Java processes and dump their map files, paying attention to file ownership etc
  – https://github.com/brendangregg/FlameGraph/blob/master/jmaps
  – Needs to run as close as possible to the profile, to minimize symbol churn

```bash
# perf record -F 99 -a -g -- sleep 30; jmaps
```
Java: Inlining

A. Disabling inlining:
   - `-XX:-Inline`
   - Many more Java frames
   - 80% slower (in this case)
   - May not be necessary: inlined flame graphs often make enough sense
   - Or tune `-XX:MaxInlineSize` and `-XX:InlineSmallCode` to reveal more frames, without costing much perf: can even go faster!

B. Symbol agents can uninline
   - `perf-map-agent unfoldall`
   - We sometimes need and use this
Node.js: Stacks & Symbols

• Frame pointer stacks work

• Symbols currently via a logger
  – --perf-basic-prof: everything. We found it can log over 1 Gbyte/day.
  – --perf-basic-prof-only-functions: tries to only log symbols we care about.

• perf may not use the most recent symbol in the log
  – We tidy logs before using them: https://raw.githubusercontent.com/brendangregg/Misc/master/perf_events/perfmaptopidy.pl

• Future v8's may support on-demand symbol dumps
# perf annotate -i perf.data.noplooper --stdio

## Gotcha #3 Instruction Profiling

Disassembly of section .text:

```
00000000004004ed <main>:
  4004ed:   push  %rbp
  4004ee:   mov   %rsp,%rbp
  4004f1:   nop
  4004f2:   nop
  4004f3:   nop
  4004f4:   nop
  4004f5:   nop
  4004f6:   nop
  4004f7:   nop
  4004f8:   nop
  4004f9:   nop
  4004fa:   nop
  4004fb:   nop
  4004fc:   nop
  4004fd:   nop
  4004fe:   nop
  4004ff:   nop
  400500:   nop
  400501:   jmp  4004f1 <main+0x4>
```

16 NOPs in a loop

Let's profile instructions to see which are hot!

(have I lost my mind?)
Instruction Profiling

• Even distribution (A)? Or something else?
Instruction Profiling

# perf annotate -i perf.data.noplooper --stdio

<table>
<thead>
<tr>
<th>Percent</th>
<th>Source code &amp; Disassembly of noplooper</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Disassembly of section .text:</td>
</tr>
<tr>
<td></td>
<td>000000000004004ed &lt;main&gt;:</td>
</tr>
<tr>
<td>0.00</td>
<td>4004ed: push %rbp</td>
</tr>
<tr>
<td>0.00</td>
<td>4004ee: mov %rsp,%rbp</td>
</tr>
<tr>
<td>20.86</td>
<td>4004f1: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f2: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f3: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f4: nop</td>
</tr>
<tr>
<td>19.84</td>
<td>4004f5: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f6: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f7: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004f8: nop</td>
</tr>
<tr>
<td>18.73</td>
<td>4004f9: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004fa: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004fb: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004fc: nop</td>
</tr>
<tr>
<td>19.08</td>
<td>4004fd: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004fe: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>4004ff: nop</td>
</tr>
<tr>
<td>0.00</td>
<td>400500: nop</td>
</tr>
<tr>
<td>21.49</td>
<td>400501: jmp 4004f1 &lt;main+0x4&gt;</td>
</tr>
</tbody>
</table>

Go home instruction pointer, you're drunk
PEBS

• I believe this is due to parallel and out-of-order execution of micro-ops: the sampled IP is the resumption instruction, not what is currently executing. And skid.

• PEBS may help: Intel's Precise Event Based Sampling

• perf_events has support:
  – perf record -e cycles:pp
  – The 'p' 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
  – … from tools/perf/Documentation/perf-list.txt
Gotcha #4 VM Guests

• Using PMCs from most VM guests:

```sh
# 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
```
VM Guest PMCs

- Without PMCs, %CPU is ambiguous. We need IPC.
  - Can't measure instructions per cycle (IPC), cache hits/misses, MMU/TLB events, etc.
- Is fixable: eg, Xen can enable PMCs (\texttt{vpmu} boot option)
  - I added vpmu support for subsets, eg, vpmu=arch for Intel architectural set (7 PMCs only)
  - Now available on the largest AWS EC2 instance types

<table>
<thead>
<tr>
<th>Event Name</th>
<th>UMask</th>
<th>Event Select</th>
<th>Example Event Mask Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnHalted Core Cycles</td>
<td>00H</td>
<td>3CH</td>
<td>CPU_CLK_UNHALTED.THREAD_P</td>
</tr>
<tr>
<td>Instruction Retired</td>
<td>00H</td>
<td>C0H</td>
<td>INST RETIRED.ANY_P</td>
</tr>
<tr>
<td>UnHalted Reference Cycles</td>
<td>01H</td>
<td>3CH</td>
<td>CPU_CLK_THREAD_UNHALTED.REF_XCLK</td>
</tr>
<tr>
<td>LLC Reference</td>
<td>4FH</td>
<td>2EH</td>
<td>LONGEST_LAT CACHE.REFERENCE</td>
</tr>
<tr>
<td>LLC Misses</td>
<td>41H</td>
<td>2EH</td>
<td>LONGEST_LAT CACHE.MISS</td>
</tr>
<tr>
<td>Branch Instruction Retired</td>
<td>00H</td>
<td>C4H</td>
<td>BR_INST RETIRED.ALL BRANCHES</td>
</tr>
<tr>
<td>Branch Misses Retired</td>
<td>00H</td>
<td>C5H</td>
<td>BR_MISP RETIRED.ALL BRANCHES</td>
</tr>
</tbody>
</table>
VM Guest MSRs

- Model Specific Registers (MSRs) may be exposed when PMCs are not
- Better than nothing. Can solve some issues.

```bash
# ./showboost
CPU MHz     : 2500
Turbo MHz   : 2900 (10 active)
Turbo Ratio : 116% (10 active)
CPU 0 summary every 5 seconds...

<table>
<thead>
<tr>
<th>TIME</th>
<th>C0_MCYC</th>
<th>C0_ACYC</th>
<th>UTIL</th>
<th>RATIO</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>17:28:03</td>
<td>4226511637</td>
<td>4902783333</td>
<td>33%</td>
<td>116%</td>
<td>2900</td>
</tr>
<tr>
<td>17:28:08</td>
<td>4397892841</td>
<td>5101713941</td>
<td>35%</td>
<td>116%</td>
<td>2900</td>
</tr>
<tr>
<td>17:28:13</td>
<td>4550831380</td>
<td>5279462058</td>
<td>36%</td>
<td>116%</td>
<td>2900</td>
</tr>
<tr>
<td>17:28:18</td>
<td>4680962051</td>
<td>5429605341</td>
<td>37%</td>
<td>115%</td>
<td>2899</td>
</tr>
<tr>
<td>17:28:23</td>
<td>4782942155</td>
<td>5547813280</td>
<td>38%</td>
<td>115%</td>
<td>2899</td>
</tr>
</tbody>
</table>
[...]        |
```

- showboost is from my msr-cloud-tools collection (on github)
VM Guest PEBS

- Not possible yet in Xen
  - please fix
- Ditto for LBR, BTS, processor trace
Gotcha #5 Containers

• perf from the host can't find symbol files in different mount namespaces
• We currently workaround it
• Should be fixed in 4.14
  – Krister Johansen's patches
Gotcha #6 Overhead

- Overhead is relative to the rate of events instrumented
- `perf stat` does in-kernel counts: relatively low overhead
- `perf record` writes `perf.data`, which has slightly higher CPU overhead, plus file system and disk I/O
- Test before use
  - In the lab
  - Run `perf stat` to understand rate, before `perf record`
- Also consider `--filter`, to filter events in-kernel
4. perf Advanced
perf for Tracing Events

Dynamic Tracing

Tracepoints

- Ext4:
  - Operating System
  - Applications
  - System Libraries
  - System Call Interface
  - VFS
  - File Systems
  - TCP/UDP
  - Sockets
  - Volume Manager
  - IP
  - Block Device Interface
  - Ethernet
  - Device Drivers
  - jbd2:
    - block:
    - scsi:
    - skb:

- sock:
  - sched:
  - task:
  - signal:
  - timer:
  - workqueue:
  - syscalls:

Software Events

- cpu-clock
- cs migrations
- page-faults
- minor-faults
- major-faults

PMCs

- cycles
- instructions
- branch-*
- L1-*
- LLC-*
- Memory Bus
- CPU 1
- DRAM
- mem-load
- mem-store

http://www.brendangregg.com/perf.html 2017
Tracepoints

#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)

Also see: cat /sys/kernel/debug/tracing/events/block/block_rq_insert/format

# 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

kernel source may be the only docs

Also see: cat /sys/kernel/debug/tracing/events/block/block_rq_insert/format
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
# 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 available 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
# Add a tracepoint for tcp_sendmsg(), with three entry regs (platform specific):
perf probe 'tcp_sendmsg %ax %dx %cx'

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

# Trace previously created probe when the bytes (alias) var 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 (debuginfo):
perf probe 'tcp_sendmsg size'

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

# Trace previous probe when size > 0, and state != TCP_ESTABLISHED(1) (debuginfo):
perf record -e probe:tcp_sendmsg --filter 'size > 0 && skc_state != 1' -a

- Kernel debuginfo is an onerous requirement for the Netflix cloud
- We can use registers instead (as above). But which registers?
The Rosetta Stone of Registers

One server instance with kernel debuginfo, and -nv (dry run, verbose):

```bash
# perf probe -nv 'tcp_sendmsg size sk->_sk_common.skc_state'
 [...] 
Added new event:
Writing event: p:probe/tcp_sendmsg tcp_sendmsg+0 size=%cx:u64 skc_state=+18(%si):u8
  probe:tcp_sendmsg (on tcp_sendmsg with size skc_state=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
```

All other instances (of the same kernel version):

```bash
# perf probe 'tcp_sendmsg:0 size=%cx:u64 skc_state=+18(%si):u8'
Failed to find path of kernel module.
Added new event:
  probe:tcp_sendmsg (on tcp_sendmsg with size=%cx:u64 skc_state=+18(%si):u8)
You can now use it in all perf tools, such as:
  perf record -e probe:tcp_sendmsg -aR sleep 1
```

Masami Hiramatsu was investigating a way to better automate this
perf Visualizations: Block I/O Latency Heat Map

- We automated this for analyzing disk I/O latency issues

There's still a lot more to perf...

- Using PMCs
- perf scripting interface
- perf + eBPF
- perf sched
- perf timechart
- perf trace
- perf c2c (new!)
- perf ftrace (new!)
- ...

Links & References

- perf_events
  - Kernel source: tools/perf/Documentation
    - https://perf.wiki.kernel.org/index.php/Main_Page
    - http://www.brendangregg.com/perf.html
    - http://web.eece.maine.edu/~vweaver/projects/perf_events/
  - Mailing list http://vger.kernel.org/vger-lists.html#linux-perf-users
- perf-tools: https://github.com/brendangregg/perf-tools
- PMU tools: https://github.com/andikleen/pmu-tools
- perf, ftrace, and more: http://www.brendangregg.com/linuxperf.html
- Java frame pointer patch
  - https://bugs.openjdk.java.net/browse/JDK-8068945
- Flame graphs: http://www.brendangregg.com/flamegraphs.html
- eBPF: http://lwn.net/Articles/603983/
Thank You

– Questions?
– http://www.brendangregg.com
– http://slideshare.net/brendangregg
– bgregg@netflix.com
– @brendangregg