Linux Profiling at Netflix

using perf_events (aka "perf")

Brendan Gregg

Senior Performance Architect
Performance Engineering Team
bgregg@netflix.com @brendangregg
This Talk

• This talk is about Linux profiling using perf_events
  – How to get CPU profiling to work, and overcome gotchas
  – A tour of perf_events and its features
• This is based on our use of perf_events at Netflix
• Massive Amazon EC2 Linux cloud
  – Tens of thousands of instances
  – Autoscale by ~3k each day
  – CentOS and Ubuntu, Java and Node.js

• FreeBSD for content delivery
  – Approx 33% of the US Internet traffic at night

• Performance is critical
  – Customer satisfaction: >50M subscribers
  – $$$ price/performance
  – Develop tools for cloud-wide analysis, and make them open source: NetflixOSS
  – Use server tools as needed
Agenda

1. Why We Need Linux Profiling
2. Crash Course
3. CPU Profiling
4. Gotchas
   – Stacks (gcc, Java)
   – Symbols (Node.js, Java)
   – Guest PMCs
   – PEBS
   – Overheads
5. Tracing
1. Why We Need Linux Profiling
Why We Need Linux Profiling

• Our primary motivation is simple:
  Understand CPU usage quickly and completely
Quickly:

1. Observe high CPU usage
2. Generate a perf_events-based flame graph
Brendan's patched OpenJDK, Mixed Mode CPU Flame Graph: vert.x

Flame Graph

Completely:

Kernel (C)

JVM (C++)

Java

Function: io.netty.channel.nio.NioEventLoop.run (2,885 samples, 94.87%)
Value for Netflix

• Uses for CPU profiling:
  – Help with incident response
  – Non-regression testing
  – Software evaluations
  – Identify performance tuning targets
  – Part of CPU workload characterization

• Built into Netflix Vector
  – A near real-time instance analysis tool (will be NetflixOSS)
Workload Characterization

• For CPUs:

 1. Who
 2. Why
 3. What
 4. How
Workload Characterization

• For CPUs:

  1. **Who**: which PIDs, programs, users
  2. **Why**: code paths, context
  3. **What**: CPU instructions, cycles
  4. **How**: changing over time

• Can you currently answer them? How?
CPU Tools

Who

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>VIRT</th>
<th>RES</th>
<th>ADDR</th>
<th>CPU</th>
<th>MEM</th>
<th>TIME+</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>27313</td>
<td>root</td>
<td>2753</td>
<td>324</td>
<td>7282</td>
<td>2.3</td>
<td>0.1</td>
<td>0:05:59</td>
<td>/usr/lib/jvm/java</td>
</tr>
<tr>
<td>28004</td>
<td>root</td>
<td>3232</td>
<td>2049</td>
<td>1044</td>
<td>2.7</td>
<td>2:02:39</td>
<td>/usr/lib/jvm/java</td>
<td></td>
</tr>
<tr>
<td>28173</td>
<td>root</td>
<td>6348</td>
<td>4992</td>
<td>95.0</td>
<td>0.1</td>
<td>0:02:68</td>
<td>ab -k -c 100 -n 1</td>
<td></td>
</tr>
<tr>
<td>28170</td>
<td>root</td>
<td>4660</td>
<td>2176</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:62</td>
<td>htop</td>
<td></td>
</tr>
<tr>
<td>2730</td>
<td>bgrege-te</td>
<td>268M</td>
<td>58668</td>
<td>0.0</td>
<td>0.0</td>
<td>2:31:25</td>
<td>/apps/epic/perl/b</td>
<td></td>
</tr>
<tr>
<td>2752</td>
<td>bgrege-te</td>
<td>151M</td>
<td>10388</td>
<td>0.0</td>
<td>0.1</td>
<td>1:40:38</td>
<td>postgres; bgrege-</td>
<td></td>
</tr>
<tr>
<td>28000</td>
<td>root</td>
<td>3233</td>
<td>2049</td>
<td>0.0</td>
<td>0.7</td>
<td>0:00:26</td>
<td>/usr/lib/jvm/java</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>2432</td>
<td>2256</td>
<td>0.0</td>
<td>0.0</td>
<td>0:01:29</td>
<td>/sbin/init</td>
<td></td>
</tr>
<tr>
<td>341</td>
<td>root</td>
<td>17236</td>
<td>632</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:4</td>
<td>/sbin/init</td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>root</td>
<td>21800</td>
<td>1304</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:6</td>
<td>/sbin/udev --deactivate</td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>root</td>
<td>2934</td>
<td>1164</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:21</td>
<td>dbus-daemon --sys</td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>root</td>
<td>2145</td>
<td>792</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/sbin/socket-br</td>
<td></td>
</tr>
<tr>
<td>549</td>
<td>root</td>
<td>15192</td>
<td>392</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/sbin/socket-br</td>
<td></td>
</tr>
<tr>
<td>612</td>
<td>root</td>
<td>7268</td>
<td>1828</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:24</td>
<td>dhcclient3 -e IF_M</td>
<td></td>
</tr>
<tr>
<td>644</td>
<td>root</td>
<td>50830</td>
<td>2928</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/usr/sbin/sshd -D</td>
<td></td>
</tr>
<tr>
<td>772</td>
<td>root</td>
<td>14508</td>
<td>956</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/sbin/getty -R 38</td>
<td></td>
</tr>
<tr>
<td>777</td>
<td>root</td>
<td>14508</td>
<td>952</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/sbin/getty -R 38</td>
<td></td>
</tr>
<tr>
<td>785</td>
<td>root</td>
<td>14508</td>
<td>952</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:0</td>
<td>/sbin/getty -R 38</td>
<td></td>
</tr>
</tbody>
</table>

Why

Flame Graph

How

CPU Utilization

What

root@lgu-d-bregg:~ # perf stat -a -d sleep 10

Performance counter stats for 'system wide':

- 3.999 CPUs used
- 1,026,540 context-switches
- 193,563 CPU migrations
- 4,435 page faults
- 83,859,543,801 cycles
- 61,028,919,136 stalled-cycles-frontend # 72.78% frontend
- 50,812,852,642 stalled-cycles-backend # 68.59% backend
- 52,969,864,955 instructions
- 0.63 insns per cycle
- 1,15 stalled cycles
- 10,223,584,755 branches
- 376,529,869 branch-misses
- 3.68% of all branches
- 0 L1-cache loads
- 1,339,958,792 L1-cache load-misses
- 0.00% of all branches
- 762,761,193 LLC loads
- 19.071 M/sec
CPU Tools

**Who**
- top
- htop

**Why**
- perf record -g
- flame graphs

**How**
- monitoring

**What**
- perf stat -a -d
Most companies & monitoring products today use top, htop, perf, and flame Graphs.
Re-setting Expectations

• That was pretty good… 20 years ago.
• Today you should easily understand why CPUs are used:
  – A profile of all CPU consumers and code paths
  – Visualized effectively
  – This should be easy to do
• Best done using:
  – A perf_events CPU profile of stack traces
  – Visualized as a flame graph
  – This will usually mean some sysadmin/devops work, to get perf_events working, and to automate profiling
Recent Example

1. Poor performance, and 1 CPU at 100%
2. perf_events flame graph shows JVM stuck compiling
2. Crash Course
perf_events

- The main Linux profiler, used via the "perf" command
- Add from linux-tools-common, etc.
- 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] 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
bench       General framework for benchmark suites
buildid-cache Manage build-id cache.
buildid-list List the buildids in a perf.data file
diff        Read two perf.data files and display the differential profile
evlist      List the event names in a perf.data file
inject      Filter to augment the events stream with additional information
kmem        Tool to trace/measure kernel memory(slab) properties
kvm         Tool to trace/measure kvm guest os
list        List all symbolic event types
lock        Analyze lock events
probe       Define new dynamic tracepoints
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.

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

• perf instruments using `stat` or `record`. This has three main parts: action, event, scope.
• e.g., profiling on-CPU stack traces:

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

**Action**: record stack traces
**Scope**: all CPUs
**Event**: 99 Hertz

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 Actions: 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 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: List

# 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 …)

The following one-liner tour includes some complex action, event, and scope combinations.
# Listing all currently known events:
perf list

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

# Listing sched tracepoints:
perf list 'sched:*'
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 'syscall: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
One-Liners: Profiling Events

# 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
One-Liners: 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
And More

• perf can also probe and record dynamic tracepoints, and custom CPU PMCs
• This can get a little advanced
• I'll start with CPU profiling, then gotchas
3. CPU Profiling
CPU Profiling

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

stack samples: A B A B A

A B ➔ syscall

A ➔ on-CPU

-----------

off-CPU

block

interrupt

time
perf Screenshot

- Sampling full stack traces at 99 Hertz:

```sh
# 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 --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...]
```
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)
4. Gotchas
When you try to use perf

- Stacks don't work (missing)
- Symbols don't work (hex numbers)
- Can't profile Java
- Can't profile Node.js/io.js
- PMCs aren't available
- Dynamic tracing function arguments don't work
- perf locks up
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

This is my actual checklist.
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

This is my actual checklist.

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

---
Gotcha #1 Broken Stacks

Start by testing stacks:
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.

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

28.10%
146    sed  libc-2.19.so  [.] re_search_internal
      |               |
      --- re_search_internal
      |               |
      --12.25%-- 0x3
      |       0x100007

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
      |               |
      --3.22%-- rpl_re_search
      |     match_regex
      |       do_subst
      |       execute_program
      |       process_files
      |       main
      |       __libc_start_main

not broken
Identifying Broken Stacks

78.50%  409  sed  libc-2.19.so  [.] 0x0000000000000dd7d4

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>--3.65%--</td>
<td>0x7f2516d5d10d</td>
</tr>
<tr>
<td>--2.19%--</td>
<td>0x7f2516d0332f</td>
</tr>
<tr>
<td>--1.22%-- &amp; 0x7f2516cfbd2</td>
<td></td>
</tr>
<tr>
<td>--1.22%--</td>
<td>0x7f2516d5d5ad</td>
</tr>
</tbody>
</table>

broken

71.79%  334  sed  sed  [.] 0x000000000001afcl

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>--11.65%--</td>
<td>0x40a447</td>
</tr>
<tr>
<td></td>
<td>0x40659a</td>
</tr>
<tr>
<td></td>
<td>0x408dd8</td>
</tr>
<tr>
<td></td>
<td>0x408ed1</td>
</tr>
<tr>
<td></td>
<td>0x402689</td>
</tr>
<tr>
<td></td>
<td>0x7fa1cd08aec5</td>
</tr>
<tr>
<td>--1.33%-- &amp; 0x40a4a1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--60.01%--</td>
</tr>
<tr>
<td></td>
<td>0x40659a</td>
</tr>
<tr>
<td></td>
<td>0x408dd8</td>
</tr>
<tr>
<td></td>
<td>0x408ed1</td>
</tr>
<tr>
<td></td>
<td>0x402689</td>
</tr>
<tr>
<td></td>
<td>0x7fa1cd08aec5</td>
</tr>
</tbody>
</table>

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:

A. 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

B. Use libunwind and DWARF: `perf record -g dwarf`
   - Pros: more debug info
   - Cons: not on older kernels, and inflates instance size

C. Use a custom walker (probably needs kernel support)
   - Pros: full stack walking (e.g., unwind inlined frames) & args
   - Cons: custom kernel code, can cost more CPU when in use

Our current preference is (A)
   - So how do we fix the frame pointer…


gcc -fno-omit-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!
JDK-8068945

• Java's compilers also reuse the frame pointer, and unfortunately there's no -XX:no-omit-frame-pointer (yet)
• I hacked hotspot to fix the frame pointer, and published the patch as a prototype suggestion (JDK-8068945)

--- openjdk8clean/hotspot/src/cpu/x86/vm/x86_64.ad 2014-03-04 ...
+++ openjdk8/hotspot/src/cpu/x86/vm/x86_64.ad 2014-11-08 ...
@@ -166,10 +166,9 @@

    reg_class stack_slots( /* one chunk of stack-based "registers" */ )
    //

-// Class for all pointer registers (including RSP)
+// Class for all pointer registers (including RSP, excluding RBP)
      reg_class any_reg(RAX, RAX_H,
                       RDX, RDX_H,
                       RBP, RBP_H,
                       RDI, RDI_H,
                       RSI, RSI_H,
                       RCX, RCX_H,
-   [...]

Remove RBP from register pools
We've been using our patched OpenJDK for profiling

To do: make this an option (-XX:MoreFramePointer), and (at some point) fix for invokedynamic

- See "A hotspot patch for stack profiling (frame pointer)" on the hotspot compiler dev mailing list, being discussed now

--- openjdk8clean/hotspot/src/cpu/x86/vm/macroAsmblr_x86.cpp 2014-03-04...
+++ openjdk8/hotspot/src/cpu/x86/vm/macroAsmblr_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/c1_MacroAsmblr_x86.cpp ...
+++ openjdk8/hotspot/src/cpu/x86/vm/c1_MacroAsmblr_x86.cpp ...
[...]

Fix x86-64 function prologues
Broken Java Stacks

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

• With JDK-8068945 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  [.] 0x0000000000001afcl
|  --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
  B. Or use one of the new symbol loggers (see lkml)

```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)
[...]
```
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
  – 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](https://code.google.com/p/v8/issues/detail?id=3453)
Java: Stacks & Symbols

flamegraph.pl --color=java
Java: Inlining

• Disabling inlining:
  – -XX:-Inline
  – Many more Java frames
  – 80% slower (in this case)
• Not really necessary
  – Inlined flame graphs often make enough sense
  – Or tune -XX:MaxInlineSize and -XX:InlineSmallCode a little to reveal more frames, without costing much perf
  – Can even go faster!
Node.js: Stacks & Symbols

- Covered previously on the Netflix Tech Blog
Gotcha #3 Guest PMCs

- Using PMCs from a Xen guest (currently):

```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
```
Guest PMCs

• Without PMCs, %CPU is ambiguous. We can't measure:
  – **Instructions Per Cycle** (IPC)
  – CPU cache hits/misses
  – MMU TLB hits/misses
  – Branch misprediction
  – Stall cycles

• Should be fixable: hypervisors can expose PMCs
  – At the very least, enough PMCs for IPC to work:
    INST RETIRED.ANY_P & CPU_CLK_UNHALTED.THREAD_P

• In the meantime
  – I'm using a physical server for PMC analysis
  – Also some MSRs on the cloud
MSRs

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

```sh
# ./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)
# Gotcha #4 Instruction Profiling

```
# perf annotate -i perf.data.noplooper --stdio
Percent | Source code & Disassembly of noplooper

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>00000000004004ed &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
I believe this is sample "skid", plus parallel and out-of-order execution of micro-ops: the sampled IP is the resumption instruction, not what is currently executing.

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 #5 Overhead

• Overhead is relative to the rate of events instrumented
• `perf stat` does in-kernel counts, with relatively low CPU 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
5. Tracing
Profiling vs Tracing

- Profiling takes samples. Tracing records every event.
- There are many tracers for Linux (SystemTap, ktap, etc), but only two in mainline: perf_events and ftrace

### Linux Tracing Stack

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-liners:</td>
<td>many</td>
</tr>
<tr>
<td>front-end tools:</td>
<td>perf, perf-tools</td>
</tr>
<tr>
<td>tracing frameworks:</td>
<td>perf_events, ftrace, eBPF, ...</td>
</tr>
<tr>
<td>tracing instrumentation:</td>
<td>tracepoints, kprobes, uprobes</td>
</tr>
</tbody>
</table>
Tracing Example

```bash
# 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]
[...]```
# 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]
[...]

Tracing Example

#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)

process PID [CPU] timestamp: eventname: format string

kernel source may be the only docs
perf Block I/O Latency Heat Map

- Trace data may only make sense when visualized
- e.g., block I/O latency from perf_events trace data:

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
Tracepoint Variables

- Some previous one-liners used variables with --filter
- The ftrace interface has a way to print them:

```bash
# cat /sys/kernel/debug/tracing/events/block/block_rq_insert/format
name: block_rq_insert
ID: 884
format:
  field: unsigned short common_type; offset:0; size:2; signed:0;
  field: unsigned char common_flags; offset:2; size:1; signed:0;
  field: unsigned char common_preempt_count; offset:3; size:1; signed:0;
  field: int common_pid; offset:4; size:4; signed:1;
  field: dev_t dev; offset:8; size:4; signed:0;
  field: sector_t sector; offset:16; size:8; signed:0;
  field: unsigned int nr_sector; offset:24; size:4; signed:0;
  field: unsigned int bytes; offset:28; size:4; signed:0;
  field: char rwbs[8]; offset:32; size:8; signed:1;
  field: char comm[16]; offset:40; size:16; signed:1;
  field: __data_loc char[] cmd; offset:56; size:4; signed:1;

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

- perf also has a scripting interface (Perl or Python)
  - These run perf and post-process trace data in user-space
  - Data path has some optimizations
    - Kernel buffering, and dynamic (optimal) number of buffer reads
    - But may still not be enough for high volume events
- Andi Kleen has scripted perf for his PMC tools
  - [https://github.com/andikleen/pmu-tools](https://github.com/andikleen/pmu-tools)
  - Includes toplev for applying "Top Down" methodology
- I've developed my own tools for perf & ftrace
  - [https://github.com/brendangregg/perf-tools](https://github.com/brendangregg/perf-tools)
  - Each tool has a man page and examples file
  - These are unsupported temporary hacks; their internals should be rewritten when kernel features arrive (e.g., eBPF)
perf-tools: bitesize

- Block (disk) I/O size distributions:

```bash
# ./bitesize
Tracing block I/O size (bytes), until Ctrl-C...
^C
```

<table>
<thead>
<tr>
<th>Kbytes</th>
<th>I/O</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt; 0.9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1.0 -&gt; 7.9</td>
<td>38</td>
<td>#</td>
</tr>
<tr>
<td>8.0 -&gt; 63.9</td>
<td>10108</td>
<td>######################################</td>
</tr>
<tr>
<td>64.0 -&gt; 127.9</td>
<td>13</td>
<td>#</td>
</tr>
<tr>
<td>128.0 -&gt;</td>
<td>1</td>
<td>#</td>
</tr>
</tbody>
</table>

- This automates perf with a set of in-kernel filters and counts for each bucket, to reduce overhead
- Will be much easier with eBPF
eBPF

- Extended BPF: programs on tracepoints
  - High-performance filtering: JIT
  - In-kernel summaries: maps
- e.g., in-kernel latency heat map (showing bimodal):
Linux Profiling Future

• eBPF is integrating, and provides the final missing piece of tracing infrastructure: efficient kernel programming
  – perf_events + eBPF?
  – ftrace + eBPF?
  – Other tracers + eBPF?
• At Netflix, the future is Vector, and more self-service automation of perf_events
Summary & Your Action Items

• Short term: **get full CPU profiling to work**
  A. Automate perf CPU profiles with flame graphs. See this talk!
  B. … or use Netflix Vector when it is open sourced
  C. … or ask performance monitoring vendors for this
     – Most importantly, you should *expect* that full CPU profiles are available at your company. The ROI is worth it.

• Long term: **PMCs & tracing**
  – Use perf_events to profile other targets: CPU cycles, file system I/O, disk I/O, memory usage, …

• Go forth and profile!

---

The "real" checklist reminder:
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
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
  • http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2014-December/016477.html
  • https://bugs.openjdk.java.net/browse/JDK-8068945
• Methodology: http://www.brendangregg.com/methodology.html
• Flame graphs: http://www.brendangregg.com/flamegraphs.html
• Heat maps: http://www.brendangregg.com/heatmaps.html
• eBPF: http://lwn.net/Articles/603983/
Thanks

• Questions?
• http://slideshare.net/brendangregg
• http://www.brendangregg.com
• bgregg@netflix.com
  – Performance and Reliability Engineering
• @brendangregg