Visualizing Performance with Flame Graphs

Brendan Gregg
Senior Performance Architect
Jul 2017
Visualize CPU time consumed by all software

Kernel

Java

User-level
Agenda

1. CPU Flame graphs
2. Fixing Stacks & Symbols
3. Advanced flame graphs
Take aways

1. Interpret CPU flame graphs
2. Understand pitfalls with stack traces and symbols
3. Discover opportunities for future development
Case Study

Exception handling consuming CPU
Summary

CPU PROFILING
CPU Profiling

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

• A code path snapshot. e.g., from jstack(1):

```
$ jstack 1819
[...]
"main" prio=10 tid=0x00007ff304009000 nid=0x7361
  runnable [0x00007ff30d4f9000]
    java.lang.Thread.State: RUNNABLE
      at Func_abc.func_c(Func_abc.java:6)
      at Func_abc.func_b(Func_abc.java:16)
      at Func_abc.func_a(Func_abc.java:23)
      at Func_abc.main(Func_abc.java:27)
```
System Profilers

• Linux
  – perf_events (aka "perf")

• Oracle Solaris
  – DTrace

• OS X
  – Instruments

• Windows
  – XPerf, WPA (which now has flame graphs!)

• And many others...
Linux perf_events

• Standard Linux profiler
  – Provides the `perf` command (multi-tool)
  – Usually pkg added by linux-tools-common, etc.

• Many event sources:
  – Timer-based sampling
  – Hardware events
  – Tracepoints
  – Dynamic tracing

• Can sample stacks of (almost) everything on CPU
  – Can miss hard interrupt ISRs, but these should be near-zero. They can be measured if needed (I wrote my own tools).
#perf Profiling

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

```
# Overhead       Samples  Command      Shared Object                         Symbol
# ........  ............  .......  .................  .............................
#
# 20.42%           605     bash  [kernel.kallsyms] [k] xen_hypercall_xen_version

|--- xen_hypercall_xen_version
     check_events

|--44.13%-- syscall_trace_enter
     tracesys

|--35.58%-- __GI___libc_fcntl

|--65.26%-- do_redirection_internal
           do_redirections
           execute_builtin_or_function
           execute_simple_command

[... ~13,000 lines truncated ...]
```
Full perf report Output
... as a Flame Graph
Flame Graph Summary

• Visualizes a collection of stack traces
  – **x-axis**: alphabetical stack sort, to maximize merging
  – **y-axis**: stack depth
  – **color**: random (default), or a dimension

• Currently made from Perl + SVG + JavaScript
  – [https://github.com/brendangregg/FlameGraph](https://github.com/brendangregg/FlameGraph)
  – Takes input from many different profilers
  – Multiple d3 versions are being developed

• References:
  – [http://queue.acm.org/detail.cfm?id=2927301](http://queue.acm.org/detail.cfm?id=2927301)
  – "The Flame Graph" CACM, June 2016
Flame Graph Interpretation
Flame Graph Interpretation (1/3)

Top edge shows who is running on-CPU, and how much (width)
Flame Graph Interpretation (2/3)

Top-down shows ancestry

e.g., from g():
Flame Graph Interpretation (3/3)

Widths are proportional to presence in samples
e.g., comparing b() to h() (incl. children)
### Mixed-Mode Flame Graphs

- **Hues:**
  - green == JIT (e.g., Java)
  - aqua == inlined
    - if included
  - red == user-level*
  - orange == kernel
  - yellow == C++

- **Intensity:**
  - Randomized to differentiate frames
  - Or hashed on function name

* new palette uses red for kernel modules too
Differential Flame Graphs

- **Hues:**
  - red == more samples
  - blue == less samples

- **Intensity:**
  - Degree of difference

- **Compares two profiles**

- **Can show other metrics:** e.g., CPI

- **Other types exist**
  - flamegraphdiff
Icicle Graph

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw_spin_lock</td>
<td>xen_hypercall_event_channel_op</td>
</tr>
<tr>
<td>do_numa_page</td>
<td>xen_send_IPI_one</td>
</tr>
<tr>
<td>handle_mm_fault</td>
<td>xen_send_SMMIO</td>
</tr>
<tr>
<td>do_page_fault</td>
<td>xen_atomic_create_page</td>
</tr>
<tr>
<td>page_fault</td>
<td>_page_page_fault</td>
</tr>
<tr>
<td>flush_HIS_page</td>
<td>_page_page_fault</td>
</tr>
<tr>
<td>pte_get_clear_flush</td>
<td>_do_page_fault</td>
</tr>
<tr>
<td>try_to_unmap</td>
<td>_do_page_fault</td>
</tr>
<tr>
<td>migrate_pages</td>
<td>page_fault</td>
</tr>
<tr>
<td>migrate_missplaced_page</td>
<td></td>
</tr>
<tr>
<td>do_numa_page</td>
<td></td>
</tr>
<tr>
<td>handle_mm_fault</td>
<td></td>
</tr>
<tr>
<td>do_page_fault</td>
<td></td>
</tr>
</tbody>
</table>

**top (leaf) merge**
Flame Graph Search

- **Color:** magenta to show matched frames
Flame Charts

- Final note: these are useful, but are not flame graphs

- Flame **charts**: x-axis is time
- Flame **graphs**: x-axis is population (maximize merging)
Pitfalls and fixes

STACK TRACING
Broken Stack Traces are Common

Because:

A. Profilers use frame pointer walking by default
B. Compilers reuse the frame pointer register as a general purpose register: a (usually very small) performance optimization.

```bash
# perf record -F 99 -a -g - sleep 30
# perf script
[...]
java  4579  cpu-clock:
   7f417908c10b  [unknown] (/tmp/perf-4458.map)
java  4579  cpu-clock:
   7f41792fc65f  [unknown] (/tmp/perf-4458.map)
a2d53351ff7da603  [unknown] ([unknown])
[...]
```
... as a Flame Graph

Broken Java stacks (missing frame pointer)
Fixing Stack Walking

A. Frame pointer-based
   – Fix by disabling that compiler optimization: gcc's -fno-omit-frame-pointer
   – Pros: simple, supported by many tools
   – Cons: might cost a little extra CPU

B. Debug info (DWARF) walking
   – Cons: costs disk space, and not supported by all profilers. Even possible with JIT?

C. JIT runtime walkers
   – Pros: include more internals, such as inlined frames
   – Cons: limited to application internals - no kernel

D. Last branch record
Fixing Java Stack Traces

I prototyped JVM frame pointers. Oracle rewrote it and added it to Java as -XX:+PreserveFramePointer (JDK 8 u60b19)
Fixed Stacks Flame Graph

Java stacks (but no symbols, yet)
Inlining

• Many frames may be missing (inlineed)
  – Flame graph may still make enough sense

• Inlining can often be be tuned
  – e.g. Java's -XX:-Inline to disable, but can be 80% slower
  – Java's -XX:MaxInlineSize and -XX:InlineSmallCode can be tuned
    a little to reveal more frames: can even improve performance!

• Runtimes can un-inline on demand
  – So that exception stack traces make sense
  – e.g. Java's perf-map-agent can un-inline (unfoldall option)
Stack Depth

• perf had a 127 frame limit
• Now tunable in Linux 4.8
  – sysctl -w kernel.perf_event_max_stack=512
  – Thanks Arnaldo Carvalho de Melo!

A Java microservice with a stack depth of > 900

broken stacks

perf_event_max_stack=1024
Fixing

SYMBOLS
Fixing Native Symbols

A. Add a -dbgsym package, if available
B. Recompile from source
Fixing JIT Symbols (Java, Node.js, …)

• Just-in-time runtimes don't have a pre-compiled symbol table
• So Linux perf looks for an externally provided JIT symbol file: /tmp/perf-PID.map

```bash
# perf script
Failed to open /tmp/perf-8131.map, continuing without symbols
[...]
java 8131 cpu-clock:
    7fff6f2dce1 [unknown] ([vdso])
    7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm...
    7fd30186e46 [unknown] (/tmp/perf-8131.map)
[...]
```

• This can be created by runtimes; eg, Java's perf-map-agent
Fixed Stacks & Symbols
Stacks & Symbols (zoom)
Symbol Churn

- For JIT runtimes, symbols can change during a profile
- Symbols may be mistranslated by perf's map snapshot
- Solutions:
  A. Take a before & after snapshot, and compare
  B. perf's new support for timestamped symbol logs
Containers

• perf can't find any symbol sources
  – Unless you copy them into the host

• I'm testing Krister Johansen's fix, hopefully for Linux 4.13
  – lkml: "[PATCH tip/perf/core 0/7] namespace tracing improvements"
INSTRUCTIONS

For Linux
Linux CPU Flame Graphs

Linux 2.6+, via perf.data and perf script:

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

Linux 4.5+ can use folded output

- Skips the CPU-costly stackcollapse-perf.pl step; see:

Linux 4.9+, via BPF:

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

```bash
git clone --depth 1 https://github.com/iovisor/bcc
./bcc/tools/profile.py -dF 99 30 | ./FlameGraph/flamegraph.pl > perf.svg
```

- Most efficient: no perf.data file, summarizes in-kernel
Linux Profiling Optimizations

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

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

**Linux 4.9**
- count stacks (BPF)
  - `profile.py`
  - folded output
  - `flamegraph.pl`
Language/Runtime Instructions

• Each may have special stack/symbol instructions
  – Java, Node.js, Python, Ruby, C++, Go, ...

• I'm documenting some on:
  – Also try an Internet search
GUI Automation

Eg, Netflix Vector (self-service UI):

Should be open sourced; you may also build/buy your own
Future Work

ADVANCED FLAME GRAPHS
Flame graphs can be generated for stack traces from any Linux event source.
Page Faults

• Show what triggered main memory (resident) to grow:

```
# perf record -e page-faults -p PID -g -- sleep 120
```

• "fault" as (physical) main memory is allocated on-demand, when a virtual page is first populated

• Low overhead tool to solve some types of memory leak

RES column in top(1) grows because
Other Memory Sources

http://www.brendangregg.com/FlameGraphs/memoryflamegraphs.html
Context Switches

• Show why Java blocked and stopped running on-CPU:

```
# perf record -e context-switches -p PID -g -- sleep 5
```

• Identifies locks, I/O, sleeps
  – If code path shouldn't block and looks random, it's an involuntary context switch. I could filter these, but you should have solved them beforehand (CPU load).

• e.g., was used to understand framework differences:
Disk I/O Requests

- Shows who issued disk I/O (sync reads & writes):

```bash
# perf record -e block:block_rq_insert -a -g -- sleep 60
```

- e.g.: page faults in GC? This JVM has swapped out!:

[Block I/O Flame Graph]
TCP Events

• TCP transmit, using dynamic tracing:

```
# perf probe tcp_sendmsg
# perf record -e probe:tcp_sendmsg -a -g -- sleep 1; jmaps
# perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso,trace > out.stacks
# perf probe --del tcp_sendmsg
```

• Note: can be high overhead for high packet rates
  – For the current perf trace, dump, post-process cycle

• Can also trace TCP connect & accept
  – Lower frequency, therefore lower overhead

• TCP receive is async
  – Could trace via socket read
CPU Cache Misses

• In this example, sampling via Last Level Cache loads:

```bash
# perf record -e LLC-loads -c 10000 -a -g -- sleep 5; jmaps
# perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso > out.stacks
```

• -c is the count (samples once per count)

• Use other CPU counters to sample hits, misses, stalls
CPI Flame Graph

- Cycles Per Instruction
  - red == instruction heavy
  - blue == cycle heavy
  (likely memory stall cycles)

zoomed:
Off-CPU Analysis

Off-CPU analysis is the study of blocking states, or the code-path (stack trace) that led to these states.
Off-CPU Time Flame Graph

Off-CPU Time (zoomed): tar(1)

Only showing kernel stacks in this example
CPU + Off-CPU Flame Graphs: See Everything

CPU

http://www.brendangregg.com/flamegraphs.html
The off-CPU stack trace often doesn't show the root cause of latency. What is gzip blocked on?
Off-Wake Time Flame Graph

Uses Linux enhanced BPF to merge off-CPU and waker stack in kernel context.
Off-Wake Time Flame Graph (zoomed)

- Stack Direction

- Stack
  - tar
  - entry_SYSCALL_64_fastpath
  - sys_write
  - vfs_write
  - __vfs_write
  - pipe_write
  - __wake_up_sync_key
  - __wake_up_common
  - autoremove_wake_function
  - schedule
  - pipe_wait
  - pipe_read
  - __vfs_read
  - vfs_read
  - sys_read
  - entry_SYSCALL_64_fastpath
  - gzip
  - all

- Waker task
- Waker stack
- Wokeup
- Blocked stack
- Blocked task
Chain Graphs

Walking the chain of wakeup stacks to reach root cause
Hot Cold Flame Graphs

Includes both CPU & Off-CPU (or chain) stacks in one flame graph

• However, Off-CPU time often dominates: threads waiting or polling

http://www.brendangregg.com/FlameGraphs/hotcoldflamegraphs.html
Flame Graph Diff

DFG1:

DFG2:

DFG_diff:

https://github.com/corpaul/flamegraphdiff
Take aways

1. Interpret CPU flame graphs
2. Understand pitfalls with stack traces and symbols
3. Discover opportunities for future development
Links & References

- Flame Graphs
  - "The Flame Graph" Communications of the ACM, Vol. 56, No. 6 (June 2016)
  - http://queue.acm.org/detail.cfm?id=2927301
  - http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  - http://www.brendangregg.com/FlameGraphs/memoryflamegraphs.html
  - http://www.brendangregg.com/blog/2016-02-05/ebpf-chaingraph-prototype.html
  - http://corpaul.github.io/flamegraphdiff/

- Linux perf_events
  - https://perf.wiki.kernel.org/index.php/Main_Page
  - http://www.brendangregg.com/perf.html

- Netflix Vector
  - https://github.com/netflix/vector
Thank You

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

Next topic: Performance Superpowers with Enhanced BPF