Flame Graphs on FreeBSD

Brendan Gregg
Senior Performance Architect
Performance Engineering Team
bgregg@netflix.com @brendangregg

FreeBSD Developer and Vendor Summit, Nov, 2014
Agenda

1. Genesis
2. Generation
3. CPU
4. Memory
5. Disk I/O
6. Off-CPU
7. Chain
1. Genesis
The Problem

• The same MySQL load on one host runs at 30% higher CPU than another. Why?
• CPU profiling should answer this easily
```bash
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"' {
  @[ustack()] = count(); } tick-60s { exit(0); }

dtrace: description 'profile-997 ' matched 2 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75195</td>
<td>:tick-60s</td>
</tr>
</tbody>
</table>

[...]

lib.so.1`priocntlset+0xa
lib.so.1`getparam+0x83
lib.so.1`pthread_getschedparam+0x3c
lib.so.1`pthread_setschedprio+0x1f
mysqld`dispatch_command19enum_server_commandP3THDPcj+0x9ab
mysqld`do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
lib.so.1`thrp_setup+0x8d
lib.so.1`lwp_start

4884

mysqld`add_to_statusP17system_status_varS0_+0x47
mysqld`calc_sum_of_all_statusP17system_status_var+0x67
mysqld`dispatch_command19enum_server_commandP3THDPcj+0x1222
mysqld`do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
lib.so.1`thrp_setup+0x8d
lib.so.1`lwp_start

5530
```
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"/ {
    @[ustack()] = count(); } tick-60s { exit(0); }'

dtrace: description 'profile-997 ' matched 2 probes

CPU       ID                    FUNCTION:NAME
1  75195                        :tick-60s

[...]

lib.so.1`__priocntlset+0xa
lib.so.1`getparam+0x83
lib.so.1`pthread_getschedparam+0x3c
lib.so.1`pthread_setschedprio+0x1f
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x9ab
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
lib.so.1`_thrp_setup+0x8d
lib.so.1`_lwp_start

4884

mysqld`_Z13add_to_statusP17system_status_varS0_+0x47
mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222
mysqld`_Z10do_commandP3THD+0x198

------ this stack
was sampled
this many times

Only unique stacks are shown, with their counts. This compresses the output.
This stack – the most frequent – is <2% of the samples
Despite the terse output, I elided over 500,000 lines
Here is what all the output looks like...
Size of one stack
These are just the unique stacks.

I have to compare this grey featureless square, with a grey square from the other host, and explain the 30% CPU difference.

And I need to do this by Friday.
Flame Graph of the same data
Flame Graph of the same data

stack depth

number of samples

one stack sample
Problem Solved

• Comparing two flame graphs was trivial
  – MySQL codepath difference suggested different compiler optimizations, which was confirmed

• Flame graph needed in this case
  – Profile data was too large to consume otherwise
  – Not always the case: the profiler output might be small enough to read directly. For CPU profiles, it often isn’t.
Flame Graphs: Definition

• **Boxes**: are functions
  – Visualizes a frame of a stack trace

• **y-axis**: stack depth
  – The top function led to the profiling event, everything beneath it is ancestry: explains why

• **x-axis**: spans samples, sorted alphabetically
  – Box width shows sample count: bigger for more
  – Alphabetical sort improves merging of like-frames

• **Colors**: either random or a dimension
  – Random helps separate columns
Flame Graphs: Presentation

- All threads can be shown in the same graph
  - So can multiple distributed systems
- Can be interactive
  - Mouse over for details
  - Click to zoom
- Can be invented
  - Eg, Facebook’s icicle-like flame graphs
- Uses for color:
  - Differentials
  - Modes: user/library/kernel
2. Generation
Examples

• Using DTrace to profile kernel CPU usage:

```bash
# git clone https://github.com/brendangregg/FlameGraph
# cd FlameGraph
# kldload dtraceall    # if needed
# dtrace -x stackframes=100 -n 'profile-197 /arg0/ {
      @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks
# ./stackcollapse.pl out.stacks | ./flamegraph.pl > out.svg
```

• Using pmcstat to profile stall cycles:

```bash
...
# pmcstat -S RESOURCE_STALLS.ANY -O out.pmcstat sleep 10
# pmcstat -R out.pmcstat -z100 -G out.stacks
# ./stackcollapse-pmc.pl out.stacks | ./flamegraph.pl > out.svg
```
Steps

1. Profile Stacks

2. Fold Stacks

3. Flame Graph
Step 1. Profile Stacks

- FreeBSD data sources:
  - DTrace stack() or ustack()
  - pmcstat -G stacks
  - Application profilers
  - Anything that can gather full stacks
Step 2. Fold Stacks

- Profiled stacks are “folded” to 1 line per stack:

  \[
  \text{func1;func2;func3;... count}
  \]

- Many converters exist (usually Perl). Eg:

<table>
<thead>
<tr>
<th>Format</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTrace</td>
<td>stackcollapse.pl</td>
</tr>
<tr>
<td>FreeBSD pmcstat</td>
<td>stackcollapse-pmc.pl</td>
</tr>
<tr>
<td>Linux perf_events</td>
<td>stackcollapse-perf.pl</td>
</tr>
<tr>
<td>OS X Instruments</td>
<td>stackcollapse-instruments.pl</td>
</tr>
<tr>
<td>Lightweight Java Profiler</td>
<td>stackcollapse-ljp.awk</td>
</tr>
</tbody>
</table>
Step 3. Flame Graph

- flamegraph.pl converts folded stacks into an interactive SVG Flame Graph
  - Uses JavaScript. Open in a browser.

**Usage:** ./flamegraph.pl [options] infile > outfile.svg

```
  --title     # change title text
  --width     # width of image (default 1200)
  --height    # height of each frame (default 16)
  --minwidth  # omit smaller functions (default 0.1 pixels)
  --fonttype  # font type (default "Verdana")
  --fontsize  # font size (default 12)
  --countname # count type label (default "samples")
  --nametype  # name type label (default "Function:")
  --colors    # "hot", "mem", "io" palette (default "hot")
  --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)
```
Extra Step: Filtering

• Folded stacks (single line records) are easy to process with grep/sed/awk

• For CPU profiles, I commonly exclude cpu_idle():

```
# ./stackcollapse.pl out.stacks | grep -v cpu_idle | \
./flamegraph.pl out.folded > out.svg
```

• Or click-to-zoom
3. CPU
CPU: Stack Sampling

A

B

A

B

A

A

A

A

A

A

A

user-level

kernel

On-CPU

X

block . . . . . . . . . . . interrupt

 syscall
CPU: Commands

- DTrace kernel stack sampling at 199 Hertz, 60 s:

```bash
# dtrace -x stackframes=100 -n 'profile-199 /arg0/ {
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks
```

- DTrace user stack sampling at 99 Hertz, 60 s:

```bash
# dtrace -x ustackframes=100 -n 'profile-99 /arg1/ {
    @[ustack()] = count(); } tick-60s { exit(0); }' -o out.stacks
```

- Warnings:
  - ustack() more expensive
  - Short-lived processes will miss symbol translation
CPU: Commands

- DTrace both stack sampling at 99 Hertz, 30 s:

```bash
# dtrace -x stackframes=100 -x ustackframes=100 -n ' profile-99 { @[stack(), ustack(), execname] = count(); }
tick-30s { printa("%k-%k%s\n%@d\n", @); exit(0); }
' -o out.stacks
```

- This prints kernel then user stacks, separated by a “-” line. flamegraph.pl makes “-” lines grey (separators)

- pmcstat for everything beyond sampling
4. Memory
Memory: 4 Targets

1. malloc()
2. brk()
3. mmap()
4. page fault

Process Address Space

Virtual Memory

Heap

Mappings

Physical Memory

MMU

lookup
DEMO
Memory: Commands

• DTrace page fault profiling of kernel stacks, 30 s:

```
# dtrace -x stackframes=100 -n 'fbt::vm_fault:entry { @[stack()] = count(); } tick-30s { exit(0) }' -o out.stacks
```

• Flame graphs generated with --colors=mem

```
# cat out.stacks | ../stackcollapse.pl | ../flamegraph.pl \
   --title="FreeBSD vm_fault() kernel stacks" --colors=mem \
   --countname=pages --width=800 > vm_faults-kernel01.svg
```

• See earlier diagram for other targets
Memory: Commands

• DTrace page fault profiling of user stacks, 5 s:

```bash
# dtrace -x ustackframes=100 -n 'fbt::vm_fault:entry {
    @[ustack(), execname] = count(); } tick-5s { exit(0) }
' -o out.stacks
```

• Warnings:
  – Overhead for user-level stack translation relative to number of unique stacks, and might be significant
  – Stacks for short-lived processes may be hex, as translation is performed after the process has exited

• See also other memory target types (earlier pic)
5. Disk I/O
FreeBSD Storage I/O Kernel Flame Graph

kernel\`ufs_str...
kernell\`VOP_S..
kernell\`bufstra...
kernell\`bufwrite
kernell\`cluster..
kernell\`cluster..
kernell\`ffs_write
kernell\`VOP_W..
kernell\`vn_write
kernell\`vn_io_f...
kernell\`dofilewr...

......kernel\`g_disk_start

......kernel\`kern_wr...
......kernel\`g_io_schedule_down

......kernel\`sys_write
......kernel\`g_down_procbody

......kernel\`amd64..
......kernel\`fork_exit

......kernel\`0xffffffff
......kernel\`0xffffffff806c5a1e

all

iostart01.svg
DEMO
Disk I/O: Commands

• Device I/O issued with kernel stacks, 30 s:

```bash
# dtrace -x stackframes=100 -n 'io:::start {
    @[stack()] = count(); } tick-10s { exit(0) }' -o out.stacks
```

• Flame graphs generated with --colors=io

```bash
# cat out.stacks | ./stackcollapse.pl | ./flamegraph.pl \
    --title="FreeBSD Storage I/O Kernel Flame Graph" --colors=io\n    --countname=io--width=800 > iostart01.svg
```

• Note that this shows IOPS; would be better to measure and show latency
6. Off-CPU
Off-CPU Profiling

On-CPU Profiling

Off-CPU Profiling (everything else)

Thread State Transition Diagram
DEMO
Off-CPU: Commands

• DTrace off-CPU time kernel stacks, 10 s:

```bash
# dtrace -x dynvarsize=8m -x stackframes=100 –n ' sched:::off-cpu { self->ts = timestamp; } sched:::on-cpu /self->ts/ { @[stack()] = sum(timestamp - self->ts); self->ts = 0; } tick-10s { normalize(@, 1000000); exit(0); }' -o out.stacks
```

• Flame graph: countname=ms

• Warning: Often high overhead. DTrace will drop:

```bash
dtrace: 886 dynamic variable drops with non-empty dirty list
```

• User-level stacks more interesting, and expensive
Off-CPU: Commands

• DTrace off-CPU time kernel & user stacks, 10 s:

```bash
# dtrace -x dynvarsize=8m -x stackframes=100 -x ustackframes=100 -n '
  sched:::off-cpu { self->ts = timestamp; }
  sched:::on-cpu /self->ts/ {
    @[stack(), ustack(), execname] = sum(timestamp - self->ts);
    self->ts = 0; }
  tick-10s { normalize(@, 1000000);
    printa("%k-%k%s\n%@d\n", @); exit(0); } '
  -o out.offcpu
```

• Beware overheads

• Real reason for blocking often obscured
  – Need to trace the wakeups, and examine their stacks
Solve ALL The Issues

• Tantalizingly close to solving all perf issues:

<table>
<thead>
<tr>
<th>On-CPU Issues</th>
<th>Off-CPU Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Common</td>
</tr>
<tr>
<td>Some solved using <code>top</code> alone</td>
<td>Some solved using <code>iostat/systat</code></td>
</tr>
<tr>
<td>Many solved using CPU (Sample) Flame Graphs</td>
<td>Some solved using lock profiling</td>
</tr>
<tr>
<td>Most of the remainder solved using CPU performance counters</td>
<td>Some solved using Off-CPU Flame Graphs</td>
</tr>
<tr>
<td>Usually a solved problem</td>
<td>Many not straightforward</td>
</tr>
</tbody>
</table>
7. Chain Graphs
Walking the Wakeups...

1. I/O
2. Off-CPU
3. Wakeup
4. Chain
DEMO
Chain Graphs: Commands

• This may be too advanced for current DTrace
  – Can’t save stacks as variables
  – Overheads for tracing everything can become serious

• My prototype involved workarounds
  – Aggregating off-CPU->on-CPU time by:
    • execname, pid, blocking CV address, and stacks
  – Aggregating sleep->wakeup time by the same
  – Perl post-processing to connect the dots
  – Assuming that CV addrs aren’t reused during tracing
Grand Unified Analysis

• There are two types of performance issues:
  – On-CPU: Usually solved using CPU flame graphs
  – Off-CPU: Can be solved using chain graphs

• Combining CPU & chain graphs would provide a unified visualization for all perf issues

• Similar work to chain graphs:
Other Topics
Hot/Cold Flame Graphs

- CPU samples & off-CPU time in one flame graph
  - Off-CPU time often dominates & compresses CPU time too much. By-thread flame graphs helps.

- Example by Vladimir Kirillov, adding a blue frame:
CPU Counters

- Use pmcstat to make flame graphs for cycles, instructions, cache misses, stall cycles, CPI, …
Differential Flame Graphs

• Just added (used to make the CPI flame graph)
• Useful for non-regression testing: hue shows difference between profile1 and profile2
Flame Charts

• Similar to flame graphs, but different x-axis
• x-axis: passage of time
• Created by Google for Chrome/v8, inspired by flame graphs
• Needs timestamped stacks
  – Flame graphs just need stacks & counts, which is usually much less data
Other Implementations/Uses

• See
  http://www.brendangregg.com/flamegraphs.html#Updates

• Some use application profile sources, which should work on FreeBSD

• Thanks everyone who has contributed!
Other Text -> Interactive SVG Tools

Latency Heat Map

https://github.com/brendangregg/HeatMap
References & Links

• Flame Graphs:
  • http://www.brendangregg.com/flamegraphs.html
  • http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  • http://www.brendangregg.com/FlameGraphs/memoryflamegraphs.html
  • http://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html
  • http://www.brendangregg.com/FlameGraphs/hotcoldflamegraphs.html
  • http://www.slideshare.net/brendangregg/blazing-performance-with-flame-graphs
  and https://www.youtube.com/watch?v=nZfNehCzGdw
  • https://github.com/brendangregg/FlameGraph
  • http://agentzh.org/misc/slides/off-cpu-flame-graphs.pdf

• Netflix Open Connect Appliance (FreeBSD):
  • https://openconnect.itp.netflix.com/

• Systems Performance, Prentice Hall:
  • http://www.brendangregg.com/sysperfbook.html
Thanks

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