Java Performance Analysis on Linux with Flame Graphs

Brendan Gregg  Senior Performance Architect
Complete Visibility

Java Mixed-Mode Flame Graph via Linux perf_events

C++ (JVM)

Java

Java (Inlined)

C (User)

C (Kernel)
• Tens of thousands of AWS EC2 instances
• Mostly Java (Oracle JVM)

Instance: usually Ubuntu Linux

Optional Apache, memcached, Node.js, ...

Java (JDK 8)

- GC and thread dump logging
- Tomcat
- hystrix, servo

Atlas, Vector, S3 logs, sar, trace, perf, perf-tools, (BPF soon)

Auto Scaling Group

Scaling Policy

- loadavg, latency, ...

CloudWatch, servo

Instance
The Problem with Profilers
Java Profilers

Kernel, libraries, JVM

Java

GC
Java Profilers

• Visibility
  – Java method execution
  – Object usage
  – GC logs
  – Custom Java context

• Typical problems:
  – Sampling often happens at safety/yield points (skew)
  – Method tracing has massive observer effect
  – Misidentifies RUNNING as on-CPU (e.g., epoll)
  – Doesn't include or profile GC or JVM CPU time
  – Tree views not quick (proportional) to comprehend

• Inaccurate (skewed) and incomplete profiles
System Profilers

Java

JVM

Locks

Time

GC

Idle thread

TCP/IP

epoll
System Profilers

• Visibility
  – JVM (C++)
  – GC (C++)
  – libraries (C)
  – kernel (C)

• Typical problems (x86):
  – Stacks missing for Java
  – Symbols missing for Java methods

• Other architectures (e.g., SPARC) have fared better

• Profile everything except Java
Workaround

• Capture both:

• An improvement, but system stacks are missing Java context, and therefore hard to interpret
Java Mixed-Mode Flame Graph

Solution
Solution

• Fix system profiling, see everything:
  – Java methods
  – JVM (C++)
  – GC (C++)
  – libraries (C)
  – kernel (C)
  – Other apps

• Minor Problems:
  – 0-3% CPU overhead to enable frame pointers (usually <1%).
  – Symbol dumps can consume a burst of CPU

• Complete and accurate (asynchronous) profiling
Saving 13M CPU Minutes Per Day

System Example

Exception handling consuming CPU
Profiling GC

GC internals, visualized:
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  B  A  B  A  A
        ↓  ↓  ↓  ↓
       ___ ___ ___ ___
      |  |  |  |  |
      |  |  |  |  |
    A  B  syscalls  off-CPU
        ___ ___ ___ ___
      |  |  |  |  |
      |  |  |  |  |
    A  A  A
```
Stack Traces

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

```
$ jstack 1819
[...]
"main" prio=10 tid=0x00007ff304009000
nid=0x7361 runnable [0x00007ff30d4f9000]
   at 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 Graphs

- Flame Graphs:
  - **x-axis**: alphabetical stack sort, to maximize merging
  - **y-axis**: stack depth
  - **color**: random (default), or a dimension

- Currently made from Perl + SVG + JavaScript
  - Multiple d3 versions are being developed

- References:
  - http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  - http://queue.acm.org/detail.cfm?id=2927301
  - "The Flame Graph" CACM, June 2016

- Easy to make
  - Converters for many profilers
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():

- a()
- b()
- c()
- d()
- e()
- f()
- 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 == Java
  – aqua == Java (inlined)
    • if included
  – red == system
  – yellow == C++

• Intensity:
  – Randomized to differentiate frames
  – Or hashed on function name
Differential Flame Graphs

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

- **Intensity:**
  - Degree of difference

- **Compared two profiles**

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

- **Other types exist**
  - flamegraphdiff
Flame Graph Search

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

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

<table>
<thead>
<tr>
<th>Time</th>
<th>1800 ms</th>
<th>1900 ms</th>
<th>2000 ms</th>
<th>2100 ms</th>
<th>2200 ms</th>
<th>2300 ms</th>
<th>2400 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>949.6 ms</td>
<td>76.8 ms</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>Main Thread</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluate Script (<a href="https://jo...ff55ce4516f0d8684102c.js:1">https://jo...ff55ce4516f0d8684102c.js:1</a>)</td>
<td>Parse HTML (anonymous function)</td>
<td>Event (DOM...ntLoaded) (anonymous...ction)</td>
<td>Function Ca...102c.js:1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Flame charts:** x-axis is time
- **Flame graphs:** x-axis is population (maximize merging)
Stack Tracing
Broken Java Stacks on x86

- These stacks are 1 or 2 levels deep, with junk values.
- On x86 (x86_64), hotspot uses the frame pointer register (RBP) as a general purpose.
- This "compiler optimization" breaks (RBP-based) stack walking.
- *Once upon a time*, x86 had fewer registers, and this made more sense.
- gcc provides `-fno-omit-frame-pointer` to avoid doing this, but the JVM had no such option...

```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

Possibilities:

A. Fix frame pointer-based stack walking (the default)
   – Pros: simple, supported by many tools
   – Cons: might cost a little extra CPU

B. Use a custom walker (likely needing kernel support)
   – Pros: full stack walking (incl. inlining) & arguments
   – Cons: custom kernel code, can cost more CPU when in use

C. Try libunwind and DWARF
   – Even feasible with JIT?

Our current preference is (A)
I hacked OpenJDK x86_64 to support frame pointers
   - Taking RBP out of register pools, and adding function prologues. It worked, I shared the patch.
   - It became JDK-8068945 for JDK 9 and JDK-8072465 for JDK 8

Zoltán Majó (Oracle) rewrote it, and it is now:
   - -XX:+PreserveFramePointer in JDK 9 and JDK 8 u60b19
   - Thanks to Zoltán, Oracle, and the other hotspot engineers for helping get this done!

It might cost 0 – 3% CPU, depending on workload
# perf script

```
[...
java  4579 cpu-clock:
    7f417908c10b [unknown] (/tmp/...

java  4579 cpu-clock:
    7f41792fc65f [unknown] (/tmp/...
    a2d53351ff7da603 [unknown] ([unkn...[...
```

# perf script

```
[...
java  8131 cpu-clock:
    7ff76f2dce1 [unknown] ([vdso])
    7fd3173f7a93 os::javaTimeMillis() (/usr/lib/jvm...
    7fd301861e46 [unknown] (/tmp/perf-8131.map)
    7fd30184def8 [unknown] (/tmp/perf-8131.map)
    7fd30174f544 [unknown] (/tmp/perf-8131.map)
    7fd30175d3a8 [unknown] (/tmp/perf-8131.map)
    7fd30166d51c [unknown] (/tmp/perf-8131.map)
    7fd301750f34 [unknown] (/tmp/perf-8131.map)
    7fd3016c2280 [unknown] (/tmp/perf-8131.map)
    7fd301b02ec0 [unknown] (/tmp/perf-8131.map)
    7fd3016f9888 [unknown] (/tmp/perf-8131.map)
    7fd3016ece04 [unknown] (/tmp/perf-8131.map)
    7fd30177783c [unknown] (/tmp/perf-8131.map)
    7fd301600aa8 [unknown] (/tmp/perf-8131.map)
    7fd301a4484c [unknown] (/tmp/perf-8131.map)
    7fd3010072e0 [unknown] (/tmp/perf-8131.map)
    7fd301007325 [unknown] (/tmp/perf-8131.map)
    7fd301007325 [unknown] (/tmp/perf-8131.map)
    7fd3010004e7 [unknown] (/tmp/perf-8131.map)
    7fd3171df76a JavaCalls::call_helper(JavaValue*,...
    7fd3171dce44 JavaCalls::call_virtual(JavaValue*...
    7fd3171dd43a JavaCalls::call_virtual(JavaValue*...
    7fd31721b6ce thread_entry(JavaThread*, Thread*)
    7fd3175389e0 JavaThread::thread_main_inner() (/...
    7fd317538cb2 JavaThread::run() (/usr/lib/jvm/ni...
    7fd3173f6f52 java_start(Thread*) (/usr/lib/jvm/ni...
    7fd317a7e182 start_thread (/lib/x86_64-linux-gnu...```
Fixed Stacks Flame Graph

Java stacks (but no symbols)
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
Symbols
Fixing Symbols

• For JIT'd code, Linux perf already looks for an externally provided symbol file: /tmp/perf-PID.map, and warns if it doesn't exist

```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)

[...]
```

• This file can be created by a Java agent
Java Symbols for perf

• perf-map-agent
  – Agent attaches and writes the /tmp file on demand (previous versions attached on Java start, wrote continually)
  – Thanks Johannes Rudolph!

• Use of a /tmp symbol file
  – Pros: simple, can be low overhead (snapshot on demand)
  – Cons: stale symbols

• Using a symbol logger with perf instead
  – Stephane Eranian contributed this to perf
  – See lkml for "perf: add support for profiling jitted code"

```
# perf script
  java 14025 [017]  8048.157085: cpu-clock: 7fd781253265 Ljava/util/HashMap::get (/tmp/perf-12149.map)
[...]
```
Stacks & Symbols (zoom)
Inlining

- Many frames may be missing (inlined)
  - Flame graph may still make enough sense
- Inlining can be tuned
  - `-XX:-Inline` to disable, but can be 80% slower!
  - `-XX:MaxInlineSize` and `-XX:InlineSmallCode` can be tuned a little to reveal more frames
    - Can even improve performance!
- `perf-map-agent` can un-inline (unfoldall)
  - Adds inlined frames to symbol dump
  - `flamegraph.pl --color=java` will color these aqua
  - Thanks Johannes Rudolph, T Jake Luciani, and Nitsan Wakart
Instructions
Instructions

1. Check Java version
2. Install perf-map-agent
3. Set -XX:+PreserveFramePointer
4. Profile Java
5. Dump symbols
6. Generate Mixed-Mode Flame Graph

Note these are unsupported: use at your own risk

1. Check Java Version

- Need JDK8u60 or better
  - for -XX:+PreserveFramePointer

```bash
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

- Upgrade Java if necessary
2. Install perf-map-agent

- Check [https://github.com/jrudolph/perf-map-agent](https://github.com/jrudolph/perf-map-agent) for the latest instructions. e.g.:

```
$ sudo bash
# apt-get install -y cmake
# export JAVA_HOME=/usr/lib/jvm/java-8-oracle
# cd /usr/lib/jvm
# git clone --depth=1 https://github.com/jrudolph/perf-map-agent
# cd perf-map-agent
# cmake .
# make
```
3. Set `-XX:+PreserveFramePointer`

- Needs to be set on Java startup
- Check it is enabled (on Linux):
  
  ```
  $ ps wwp `pgrep -n java` | grep PreserveFramePointer
  or
  $ jcmd `pgrep -n java` VM.flags | grep PreserveFramePointer
  ```

- Measure overhead (should be small)
4. Profile Java

• Using Linux perf_events to profile all processes, at 99 Hertz, for 30 seconds (as root):

```bash
# perf record -F 99 -a -g -- sleep 30
```

• Just profile one PID (broken on some older kernels):

```bash
# perf record -F 99 -p PID -g -- sleep 30
```

• These create a perf.data file
5. Dump Symbols

- See perf-map-agent docs for updated usage
- e.g., as the same user as java:

```bash
$ cd /usr/lib/jvm/perf-map-agent/out
$ java -cp attach-main.jar:$JAVA_HOME/lib/tools.jar \net.virtualvoid.perf.AttachOnce PID
```

- perf-map-agent contains helper scripts. I wrote my own:
  - https://github.com/brendangregg/Misc/blob/master/java/jmaps
- Dump symbols quickly after perf record to minimize stale symbols. How I do it:

```bash
# perf record -F 99 -a -g -- sleep 30; jmaps
```
6. Generate a Mixed-Mode Flame Graph

- Using my FlameGraph software:

  ```
  # perf script > out.stacks01
  # git clone --depth=1 https://github.com/brendangregg/FlameGraph
  # cat out.stacks01 | ./FlameGraph/stackcollapse-perf.pl | \
  ./FlameGraph/flamegraph.pl --color=java --hash > flame01.svg
  ```

  - perf script reads perf.data with /tmp/* .map
  - out.stacks01 is an intermediate file; can be handy to keep

- Finally open flame01.svg in a browser
- Check for newer flame graph implementations (e.g., d3)
Optimizations

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

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

• Linux 4.5+ can use folded output
  – Skips the CPU-costly stackcollapse-perf.pl step; see:

• Linux 4.9+, via BPF:

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

Near real-time, per-second metrics

Select Instance

Select Metrics

Flame Graphs
Netflix Vector

- Open source, on-demand, instance analysis tool
  - [https://github.com/netflix/vector](https://github.com/netflix/vector)
- Shows various real-time metrics
- Flame graph support currently in development
  - Automating previous steps
  - Using it internally already
  - Also developing a new d3 front end
Advanced Analysis
Linux perf_events Coverage

Dynamic Tracing

u-probes

k-probes

Tracepoints

Operating System

Applications

System Libraries

System Call Interface

Sockets

TCP/UDP

Scheduler

Virtual Memory

Device Drivers

Device Interconnect

Software Events

cpu-clock

cs migrations

page-faults

minor-faults

major-faults

syscalls:

sched:

task:

signal:

timer:

workqueue:

CPU Interconnect

kmem:

vmscan:

writeback:

irq:

cycles

instructions

branch- *

L1- *

LLC- *

Memory Bus

DRAM

... all possible with Java stacks
Advanced Flame Graphs

• Any event can be flame graphed, provided it is issued in synchronous Java context
  – Java thread still on-CPU, and event is directly triggered
  – On-CPU Java context is valid

• Synchronous examples:
  – Disk I/O requests issued directly by Java → yes
    • direct reads, sync writes, page faults
  – Disk I/O completion interrupts → no*
  – Disk I/O requests triggered async, e.g., readahead → no*
    * can be made yes by tracing and associating context
Page Faults

- Show what triggered main memory (resident) to grow:
  
  ```sh
  # 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
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):

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

- e.g.: page faults in GC? This JVM has swapped out!
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 overhead)
- TCP receive is async
  - Could trace via socket read

TCP sends
CPU Cache Misses

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

  ```
  # 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 mem stall cycles)

zoomed:
Java Package Flame Graph

• Sample on-CPU instruction pointer only (no stack)
  – Don't need -XX:+PreserveFramePointer

• y-axis: package name hierarchy
  – java / util / ArrayList / ::size

Linux 2.6+ (pre-BPF):

# perf record -F 199 -a -- sleep 30; ./jmaps
# perf script | ./pkgsplit-perf.sh | ./flamegraph.pl > out.svg
Links & References

- Flame Graphs
  - http://www.brendangregg.com/flamegraphs.html
  - http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  - http://queue.acm.org/detail.cfm?id=2927301
  - "The Flame Graph" CACM, Vol. 56, No. 6 (June 2016)

- 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

Thanks

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