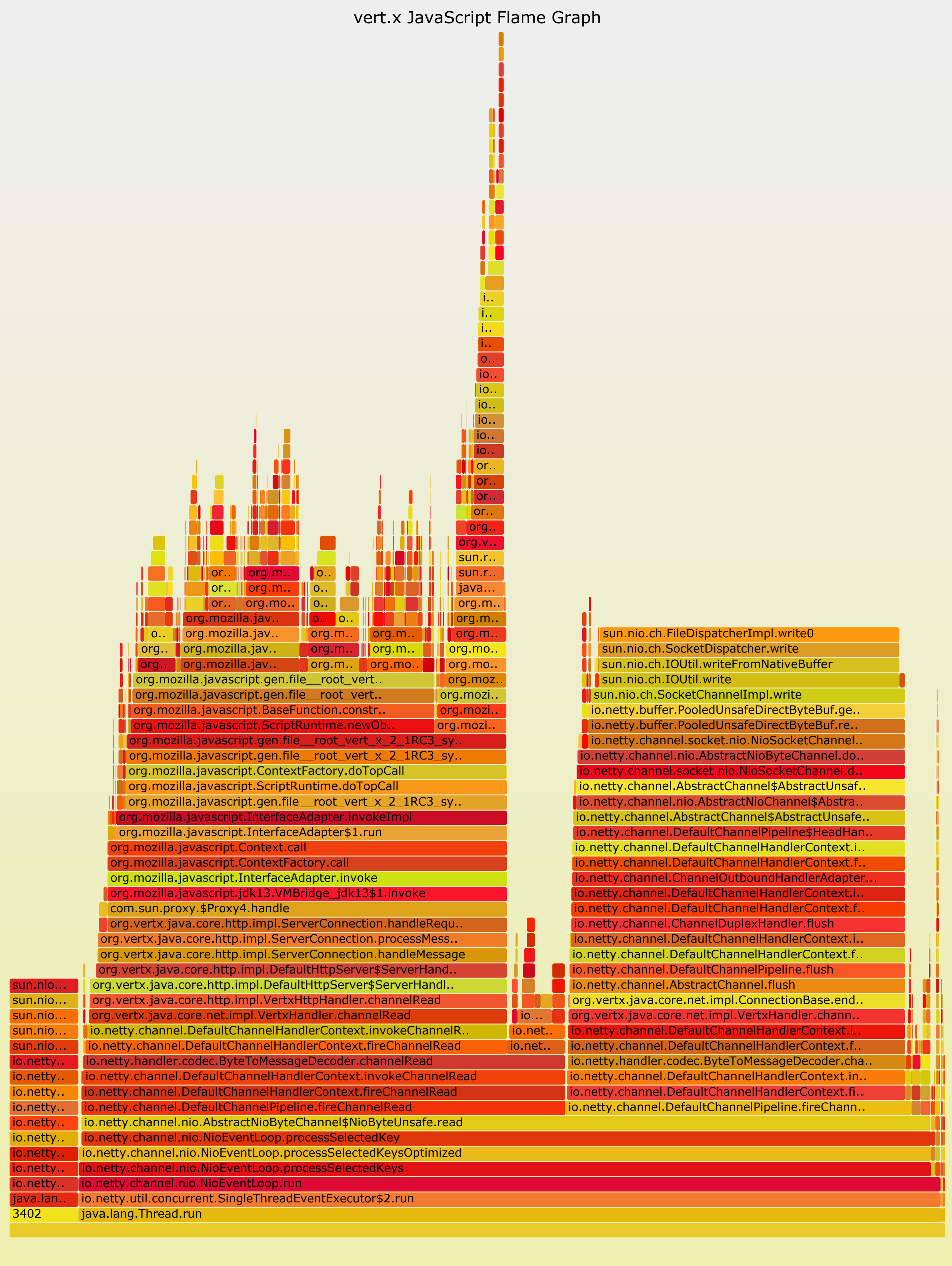
Java Flame Graphs

12 Jun 2014

Java flame graphs are a hot new way to visualize CPU usage. I'll show how to create them using free and open source tools: Google's lightweight-java-profiler (code.google.com) and my flame graph software (github). Hopefully, one day, flame graphs will be a standard feature on all Java profilers, alongside the call tree graphs that are standard today.

UPDATE (Aug 2015): See the Java in Flames Netflix Tech Blog post for the latest and best method for Java flame graphs, which uses Linux perf_events to show Java and system code paths together. I also describe the steps in Java CPU Flame Graphs. This blog post refers to a Java-only JVMTI-based method, that while works, has the caveats described in those links.

Flame Graphs show the big picture. Here is vert.x serving a simple JavaScript program:



Awesome! Mouse over elements to see details. Here is the direct SVG, and a non-interactive PNG version.

The y-axis is stack depth, and the x-axis spans the sample population. Each rectangle is a stack frame. Color is not important, it's randomized to differentiate frames. The ordering from left to right is also unimportant.

You look for the widest frames, from bottom up, and forks in the "flames", which indicate different code paths taken. See my CPU flame graphs page for a longer explanation, and the flame graphs presentation from USENIX/LISA`13. Once you get the hang of these, you can quickly identify and quantify CPU usage.

The large tower on the left is the Mozilla Rhino JavaScript engine, which is eating 42.70% CPU (look for the lowest mozilla frame; the percentage is inclusive of all child frames above it). vert.x with Java doesn't need to run this engine, freeing up those CPU resources, and roughly doubling maximum possible performance.

Other details are also interesting: the large plateau in write0(), at 31.99%, is desirable – that's vert.x responding to requests and doing work. To improve the performance further (aside from tuning write0()), examine and reduce time spent in other code paths. Eliminating Rhino provides the biggest win.

Collect flame graphs over different days or software versions, and you can also quickly quantify performance changes by comparing them.

Profile Collection

These flame graphs visualize sampled stack traces that were running on-CPU. You can use any profiler that gives you full stack traces, provided the profiler is accurate. I first tried CPU sampling using hprof, but found that it had issues.

For this example, I used the lightweight-java-profiler by Jeremy Manson. This is an open source demonstration of an accurate profiling technique, and not a point-and-click production-ready product, so some assembly (and caution) is required. My steps were:

1. Get software

```
svn checkout http://lightweight-java-profiler.googlecode.com/svn/trunk/ lightweight-java-profiler
cd lightweight-java-profiler-read-only
```

2. Customize Makefile

I edited the Makefile using vi, and set BITS to 64, and added an include path for my system. My changes looked like this (diff), yours may vary depending on include paths required:

```
4c4
< BITS?=32
---
> BITS?=64
49c49
< INCLUDES=-I$(JAVA_HOME)/$(HEADERS) -I$(JAVA_HOME)/$(HEADERS)/$(UNAME)
---
> INCLUDES=-I$(JAVA_HOME)/$(HEADERS) -I$(JAVA_HOME)/$(HEADERS)/$(UNAME) -I/usr/include/x86_64-lin
```

3. Build software:

```
make all
```

4. Set agent

I added the following option when running java (change path to match yours):

```
-agentpath:/usr/local/lightweight-java-profiler-read-only/build-64/liblagent.so
```

This samples when java starts until it exits, writing the report to a traces.txt file, which has a similar layout to hprof. The flame graph software will read this traces.txt file.

There are some overheads to have this running. For my program it was negligible (~1% loss in request rate, and a 7% increase in CPU consumption, for which headroom was available), but your mileage will vary. See the later Customizing the Profiler section for reducing the sampling rate if needed.

Flame Graph

Now to turn traces.txt into a flame graph:

```
git clone http://github.com/brendangregg/FlameGraph
cd FlameGraph
./stackcollapse-ljp.awk < ../traces.txt | ./flamegraph.pl > ../traces.svg
```

stackcollapse-ljp.awk is a simple awk program to convert the output of lightweight-java-profiler into the stack trace format that flame graph reads (one stack per line). The flamegraph.pl program has various options (list using -h) to customize the output, including changing the title.

Now open the traces.svg file in a browser.

Customizing the Profiler

It's worth mentioning that you can configure some options in lightweight-java-profiler by editing the source. See src/globals.h for:

```
// Number of times per second that we profile
static const int kNumInterrupts = 100;

// Maximum number of stack traces
static const int kMaxStackTraces = 3000;

// Maximum number of frames to store from the stack traces sampled.
static const int kMaxFramesToCapture = 128;
```

I may be inclined to reduce the sampling rate from 100 hertz to 50, or lower, if I'd like to collect data over a long run (minutes), to reduce the sampling overheads. I'd also increase the maximum stack frames, if my code exceeded 128. Flame graphs need full stacks to be drawn.

There's also Richard Warburton's honest-profiler, which builds upon the same accurate profiling technique. Like the lightweight-java-profiler, some assembly is required.

While these profilers see inside Java, my ideal profiler would include native user-level and kernel stacks. This would allow us to include JVM GC code paths, as well as other JVM internals, and kernel internals.

Flame graphs already have a history of proving their usefulness in other areas, including for kernel performance, Node.JS, Ruby, Perl, and others. See the updates section on my Flame Graphs page.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).