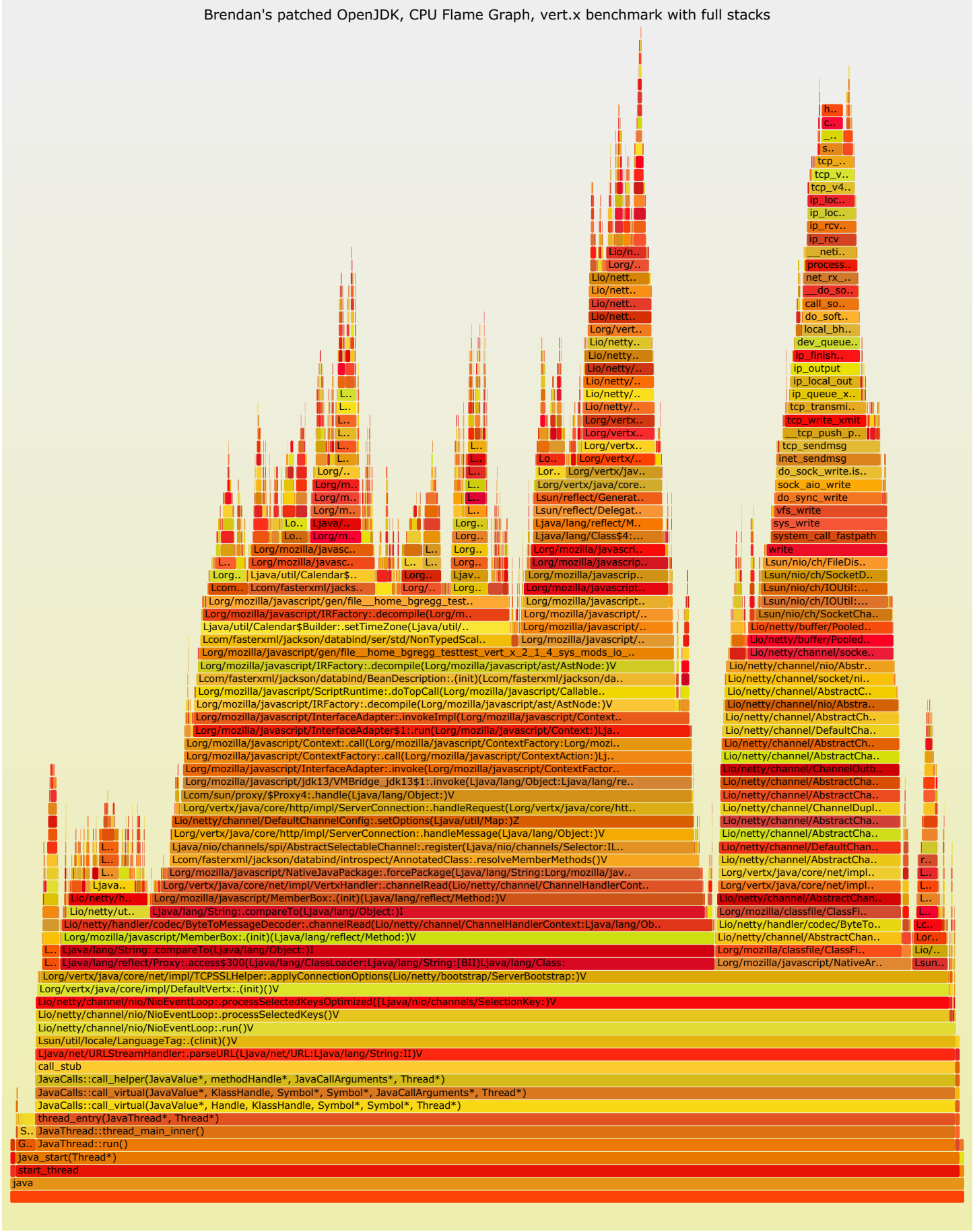
Coloring Flame Graphs: Code Hues

30 Jul 2017

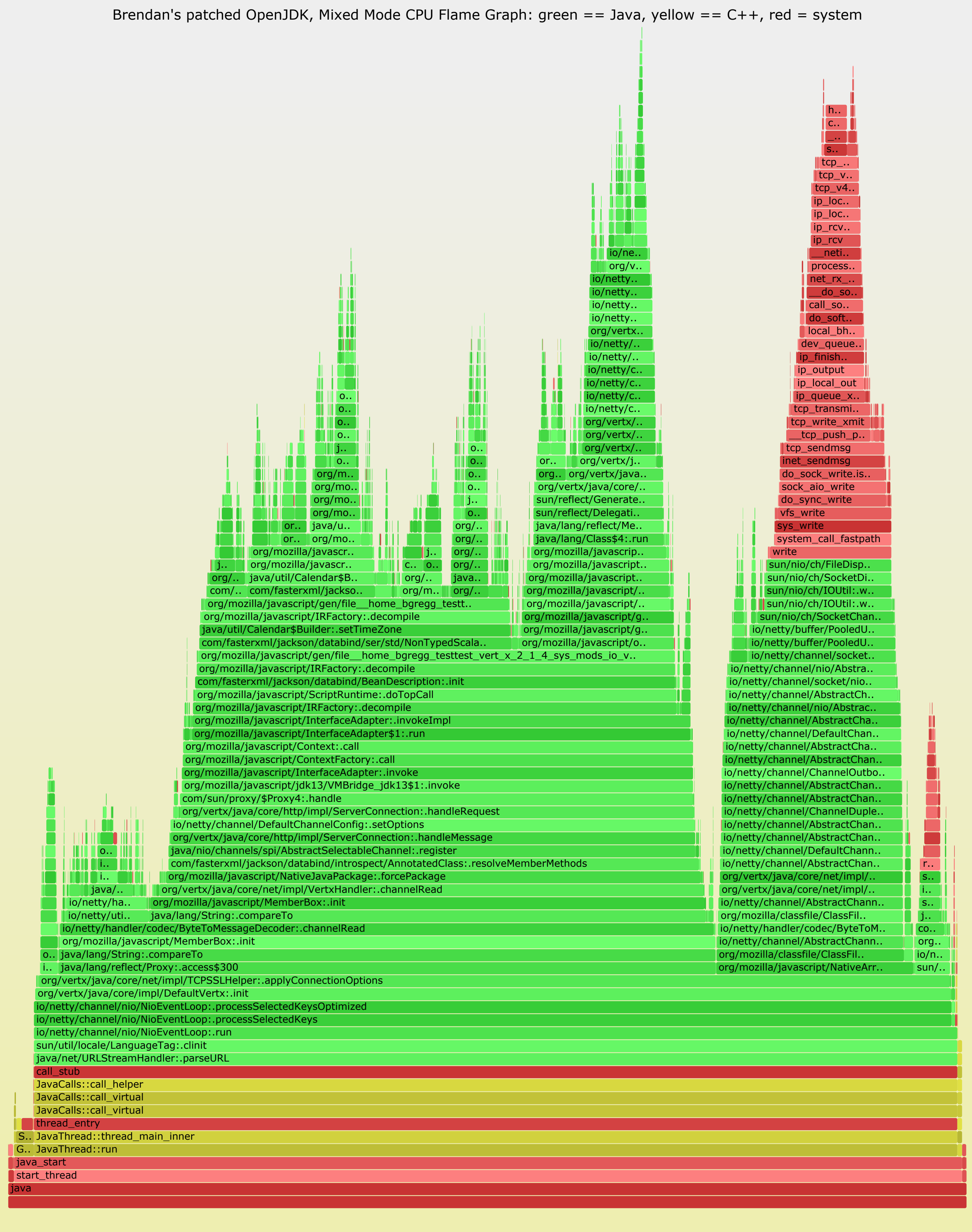
I recently improved flame graph code coloring. If you're automating or implementing flame graphs, this is a small detail that may interest you. (For an intro to flame graphs, see my website and github.)

First, a confession. Code-type coloring was a regex hack that took five minutes. In late 2014 I was modifying the JDK to preserve the frame pointer so that traditional stack walkers and profilers would work (an example of the problem is here, where Java methods lack ancestry). After I fixed the frame pointer, profiling Java looked like this (SVG):



It worked! Java methods now had ancesty (stack depth), and appear as towers.

I was delighted and showed my colleagues straight away. Amer Ather, another performance engineer at Netflix, suggested I color the Java and kernel frames differently. He was only back at his desk for five minutes when I called him back (SVG):



Done. (I also stripped the extra L from Java symbols.)

My hack was the following eight lines of code:

```
        if (defined $type and $type eq "java") {
                if ($name =~ /::/) {            # C++
                        $type = "yellow";
                } elsif ($name =~ m:/:) {       # Java (match "/" in path)
                        $type = "green"
                } else {                        # system
                        $type = "red";
                }
```

The "java" $type is from the command line option: `--color=java`. The $name is the function name. Here are some sample function names:

- **Java**

```
io/netty/channel/nio/NioEventLoop;.run
org/mozilla/classfile/ClassFileWriter;.addLoadConstant
```

- **C++**

```
JavaCalls::call_helper
JavaThread::thread_main_inner
```

- **C**

```
tcp_v4_do_rcv
start_thread
write
```

If you cast your regular expression eye over these, you'll quickly see patterns. If it contains "::" it's C++, "/" it's Java, else it's C. And that's what I coded.

It mostly worked. But I've noticed the odd case where it gets things wrong. Sometimes the profiled Java symbols use "." instead of "/" as a delimiter. Or, somehow, I have Java methods that lack any package delimiter, so were colored red. I had similar issues with JIT'd code for Node.js.

Revisiting how flame graphs for Linux perf are generated (full instructions in Java Flame Graphs):

```
perf record -F 49 -a -g -- sleep 30; ./jmaps
perf script | ./stackcollapse-perf.pl | grep -v cpu_idle | ./flamegraph.pl --color=java > out.svg
```

It's beginning with the output of `perf script` (later perf versions added a way to emit a folded summary directly). Here is some truncated `perf script` output:

```
java  4811 cpu-clock:
    ffffffff8100122a hypercall_page ([kernel.kallsyms])
    ffffffff8100aca2 check_events ([kernel.kallsyms])
    ffffffff8104dffe __wake_up_sync_key ([kernel.kallsyms])
    ffffffff8152f86e sock_def_readable ([kernel.kallsyms])
[...]
    ffffffff81662142 system_call_fastpath ([kernel.kallsyms])
        7f62aadf2f7d write (/lib/x86_64-linux-gnu/libc-2.15.so)
        7f62961a5e8b Lsun/nio/ch/FileDispatcherImpl;.write0(Ljava/io/FileDescriptor;JI)I (/tmp/pe
        7f629619dd64 Lsun/nio/ch/SocketDispatcher;.write(Ljava/io/FileDescriptor;JI)I (/tmp/perf-
        7f62961b3330 Lsun/nio/ch/IOUtil;.writeFromNativeBuffer(Ljava/io/FileDescriptor;Ljava/nio/
[...]
        7f62aa3b1618 JavaThread::thread_main_inner() (/mnt/openjdk8/build/linux-x86_64-normal-ser
        7f62aa3b186c JavaThread::run() (/mnt/openjdk8/build/linux-x86_64-normal-server-release/jd
        7f62aa272bf2 java_start(Thread*) (/mnt/openjdk8/build/linux-x86_64-normal-server-release/
        7f62aa8f2e9a start_thread (/lib/x86_64-linux-gnu/libpthread-2.15.so)
```

The stackcollapse-perf.pl tool plucks out the symbol name (second column) and discards everything else. But the last column – the segment printed in ( ) – provides more details for identifying code types. Eg:

- **`[kernel.kallsyms]`**: kernel code (I could also match the addr vs the kernel base address for this)
- **`/tmp/perf-PID.map`**: JIT'd code (Java, Node.js, ...)

This is what I made use of recently, by adding an `--all` option to stackcollapse-perf.pl to turn on all annotations. Annotations are inspired by the "[k]" annotations seen in `perf report --stdio` output. I append them after the function name, so `tcp_sengmsg` becomes `tcp_sengmsg_[k]`, and that annotation is used and then stripped by flamegraph.pl.

Annontation suffixes:

- **`_[k]`**: kernel
- **`_[j]`**: JIT
- **`_[i]`**: inlined function
- **`_[w]`**: waker stack (for offwake or chain graphs)

Making use of both annotations and pattern matching, the "java" palette is now:

- green: JIT (Java, Node.js, ...)
- aqua: inlined
- yellow: C++
- orange: kernel
- red: native (user-level)

If you're automating flame graphs using my original tools, you might want to consider adding `--all` to the normal workflow for annotations. These are currently used by the "java" and "js" palettes. Eg:

```
perf record -F 49 -a -g -- sleep 30; ./jmaps
perf script | ./stackcollapse-perf.pl --all | grep -v cpu_idle | ./flamegraph.pl --color=java > c
```

If you are using a different profiler (not Linux perf), you might want to consider enhancing its stackcollapse program to have an option to turn on annotations (or I can do it next time I use them). If you are implementing your own flame graph software, you might want to add similar color hues for code types.

Finally, it should be clear that changing the hue of code based on a regex is a trivial change to flamegraph.pl. You could add custom rules to your version to highlight your team's code, for example.
