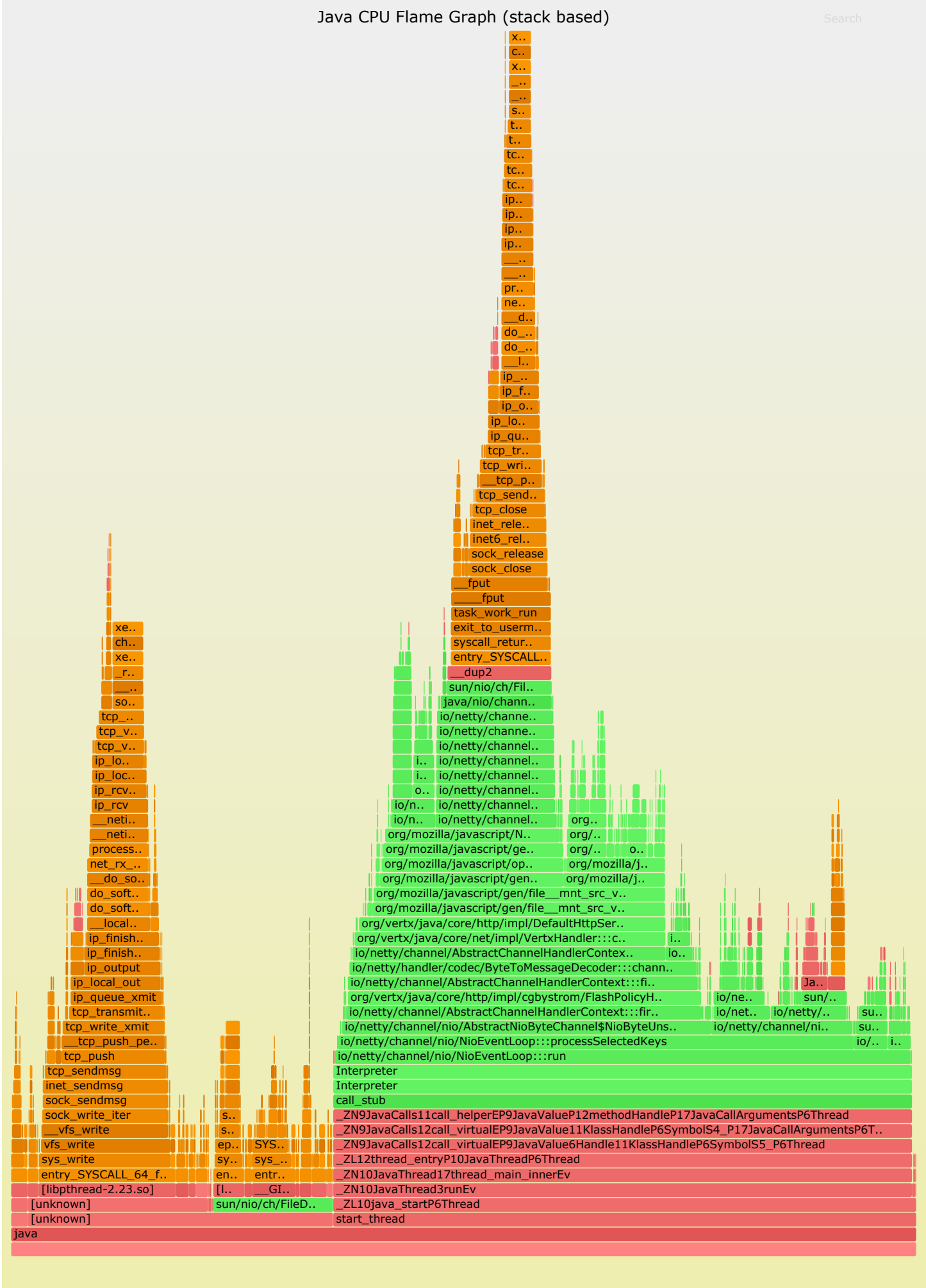
Java Package Flame Graph

30 Jun 2017

CPU flame graphs visualize running code based on its flow or stack trace ancestry, showing which functions called which other functions and so on. But with Java, there's another way to visualize the same CPU workload which provides some additional insight: **a Java package flame graph**. Instead of visualizing the stack trace hierarchy, this visualizes the Java package name hierarchy. I'll explain with a quick example.

Here is a normal stack trace-based CPU flame graph for Java, running a microbenchmark (SVG):

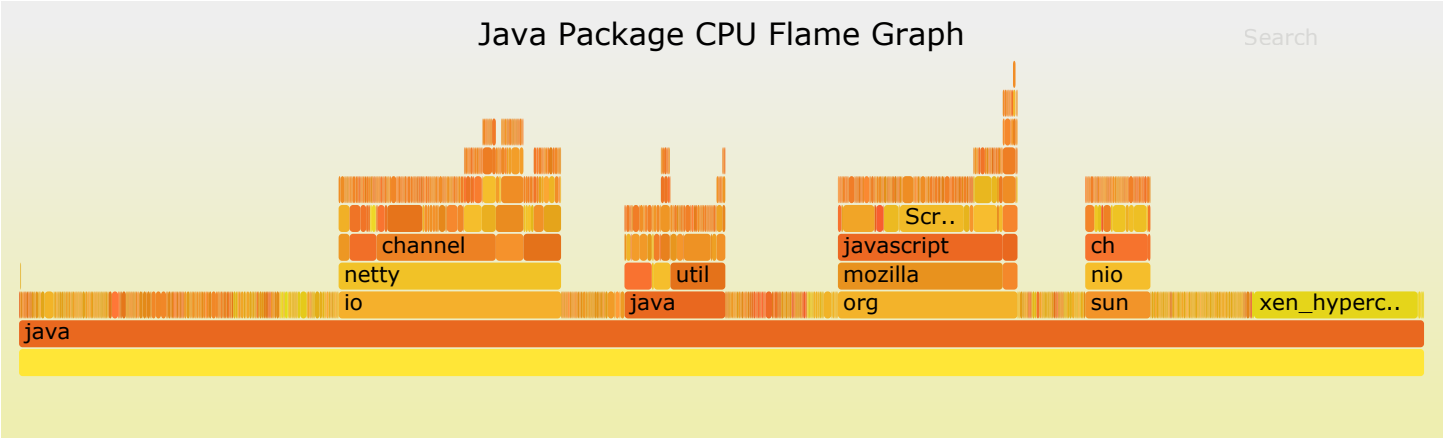


The y-axis is stack depth. From bottom to top are parent to child functions, and the top edge shows the functions running on CPU.

These flame graphs answer many questions easily, such as where the bulk of the CPU time is spent, with ancestry and child functions. But there's one line of questioning that's still tricky: How much CPU time is spent

in `java/util/*` for example? The Search button (top right) lets you answer this by searching on "java/util", and the bottom right will show 4.3%. But this includes child functions (on purpose). How much CPU time was in java/util methods directly, excluding child functions calls? That takes a bit of effort to figure out, involving zooming on each call and excluding child calls manually. A package flame graph can help here.

Now for a Java package flame graph for the same workload, also showing CPU samples (SVG):



The y-axis now spans the package name. Click to navigate. This visualizes the on-CPU functions only, so function ancestry is excluded. The time in java/util is grouped together, which can be identified visually: it's 3.91% (it should be less than the earlier flame graph, as it excludes child calls; however, this is also a separate profile and the workload may have varied). There seems to be a grass of many thin rectangles: these are not Java methods, and so don't have a package name to spilt.

Is this package flame graph better than the normal stack trace flame graph? Definitely not. I use it in addition, as a different perspective for understanding the same CPU workload.

Here's how you can make a package flame graph, using the software from my FlameGraph repository:

```
# perf record -F 99 -a -- sleep 30; ./jmaps
# perf script | ./pkgsplit-perf.pl | grep java | ./flamegraph.pl > out.svg
```

Notice something? I'm not using `-g` with `perf record`, like I normally do, so this is not collecting stack traces. That means that this type of profiling has lower overhead, which is a bonus. It also means that Java doesn't need to be running with -XX:+PreserveFramePointer, although you probably still want to so that you can collect the normal (stack trace) flame graphs.

Also, some workloads can bust perf's 127 stack frame limit (tunable in Linux 4.8 onwards), which can badly mess up a normal flame graph to the point where it's unreadable. The package name flame graph will work fine in this situation.

I introduced Java package flame graphs in my JavaOne talk last year, and was just using them again to find some extra clues. I hope they are useful for you too.

shivam bharuka • a year ago

Hey Berndan,
I really like your work on flamegraphs. But I am facing a small issue. Can you help by answering to this post: https://stackoverflow.com/q... ?
Thanks

Share ›

Oleg Mazurov • a year ago

Hi Brendan,
thought you may be interested to know that Flame Graph is now part of Oracle Developer Studio Performance Analyzer: http://www.oracle.com/techn...
Works for Java too (AsyncGetCallTrace callstacks).

Share ›

Manu Zhang • 2 years ago

I'm new to flamegraph. How can this be done with dtrace on Mac ? Thanks.

Share ›

Benjamin Schindler • 2 years ago

I really love flamegraphs, but there is one thing which they are not really good at - I frequently have to optimize code where a lot of time is spent in a few small functions, which are however called from a multitude of locations. A flamegraph-like visualization (or we used timelines which gave us something quite similar) just doesn't manage to highlight this because of its bottom-up approach. Reversing the flame graph would probably quickly reveal such cases. Have you ever tried something like that?

Share ›

brendangregg Mod ➚ Benjamin Schindler • 2 years ago

Yes, try generating the flamegraph with: --reverse --inverted

The reverse option reverses the stack merge order. Inverted prints it upside down, so it becomes an icicle graph.

The other option is to use the search option to search on those few small functions, and look at the cumulative time in the bottom right.

Share ›

About this blog