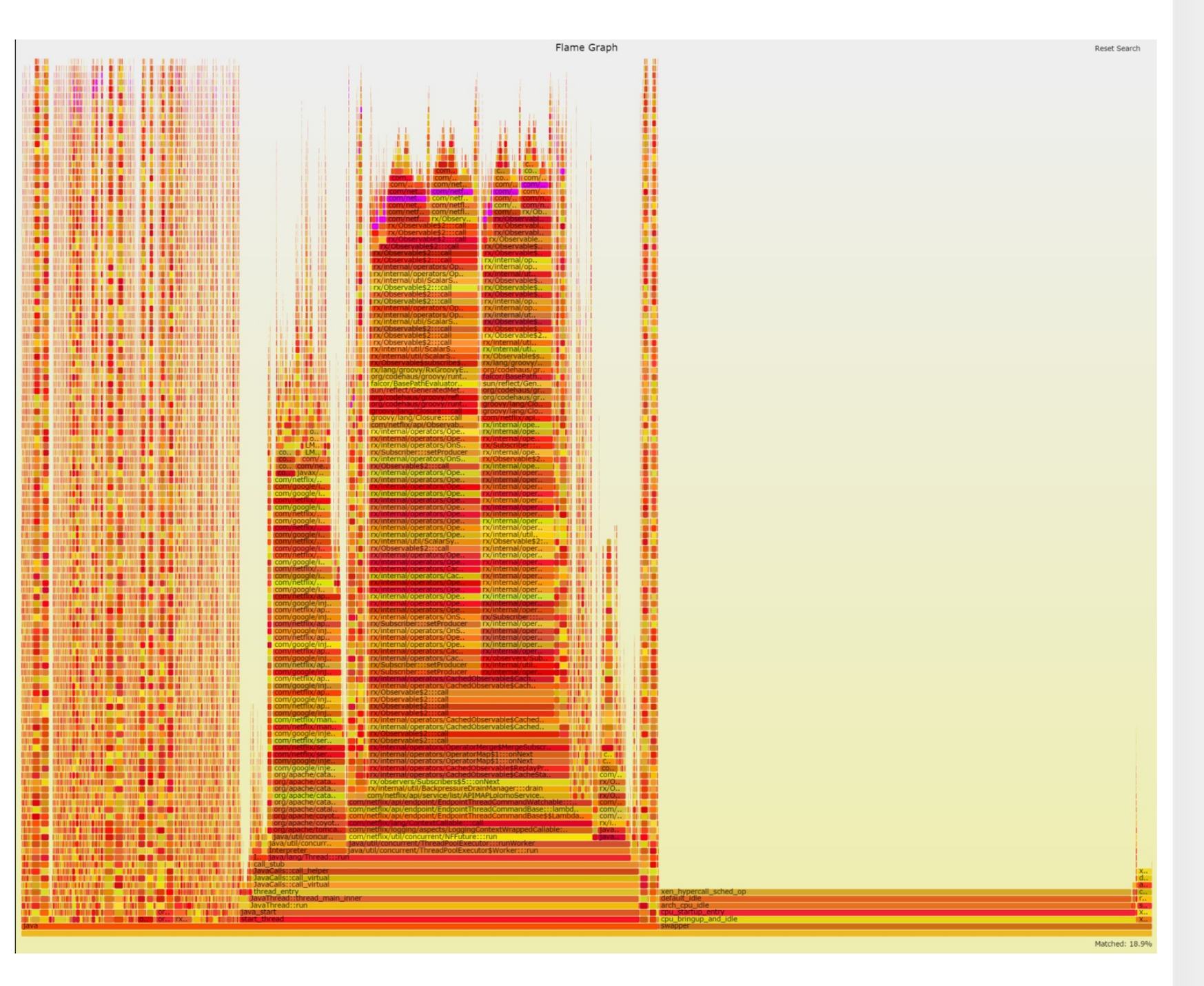
Monday, April 11, 2016

Saving 13 Million Computational Minutes per Day with Flame Graphs

We recently uncovered a significant performance win for a key microservice here at Netflix, and we want to share the problems, challenges, and new developments for the kind of analysis that lead us to the optimization.

Optimizing performance is critical for Netflix. Improvements enhance the customer experience by lowering latency while concurrently reducing capacity costs through increased efficiency. Performance wins and efficiency gains can be found in many places - tuning the platform and software stack configuration, selecting different AWS instance types or features that better fit unique workloads, honing autoscaling rules, and highlighting potential code optimizations. While many developers throughout Netflix have performance expertise, we also have a small team of performance engineers dedicated to providing the specialized tooling, methodologies, and analysis to achieve our goals.

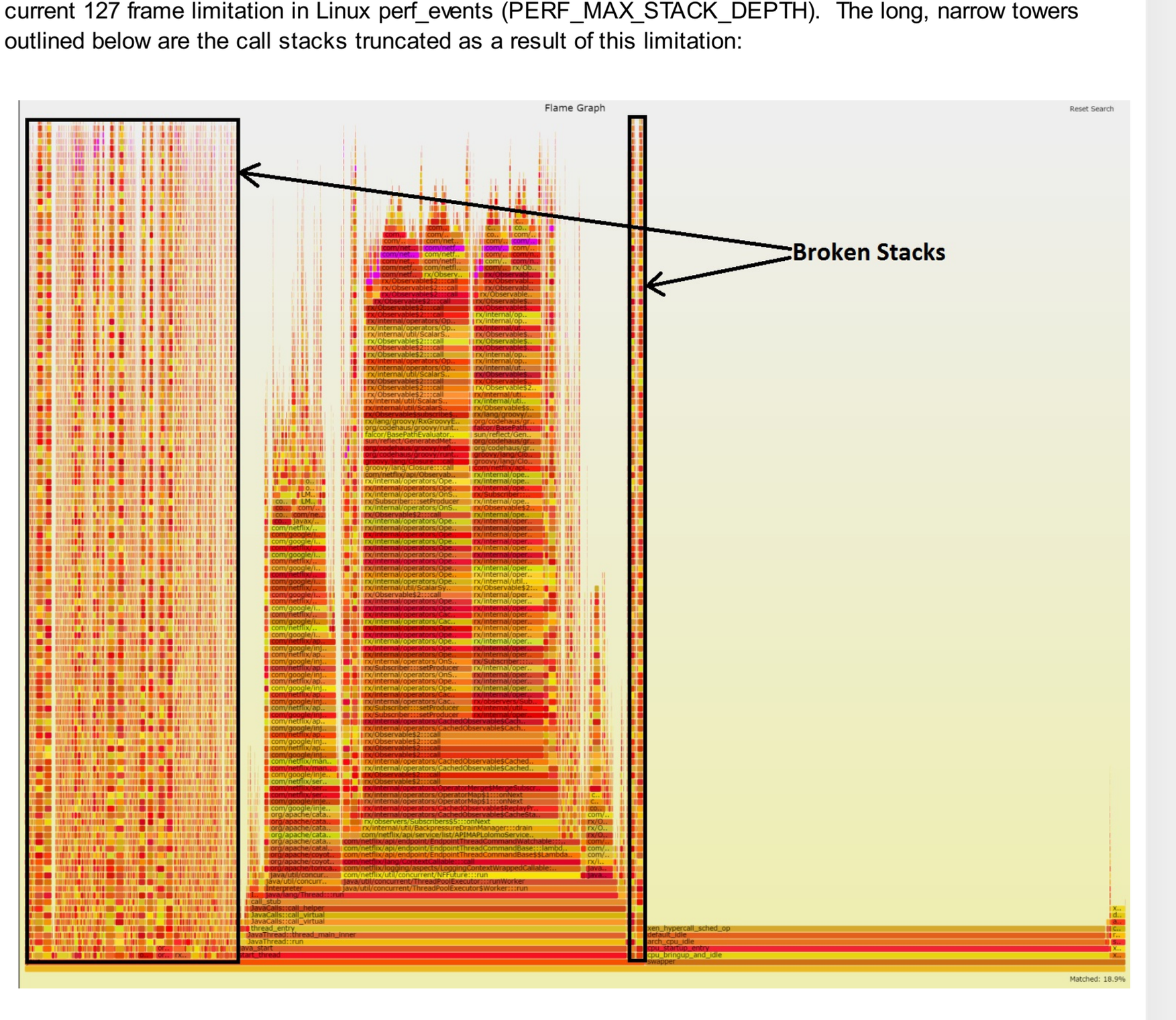
An on-going focus for the Netflix performance team is to proactively look for optimizations in our service and infrastructure tiers. It's possible to save hundreds of thousands of dollars with just a few percentage points of improvement. Given that one of our largest workloads is primarily CPU-bound, we focused on collecting and analyzing CPU profiles in that tier. Fortunately with the work Brendan Gregg pushed forward to preserve the frame pointer in the JVM we can easily capture CPU sampling profiles using Linux perf_events on live production instances and visualize them with flame graphs:



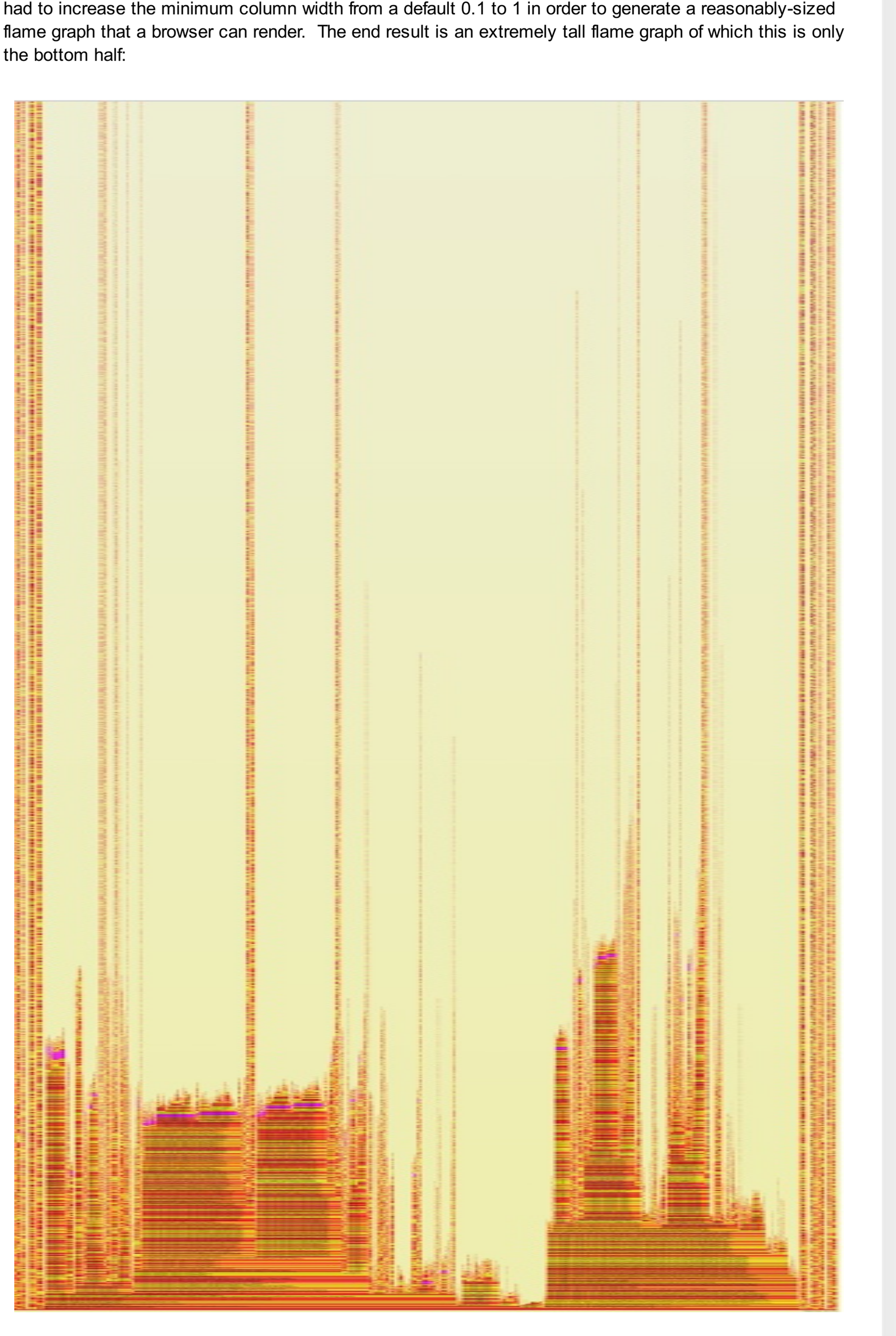
For those unfamiliar with flame graphs, the default visualization places the initial frame in a call stack at the bottom of the graph and stacks subsequent frames on top of each other with the deepest frame at the top. Stacks are ordered alphabetically to maximize the merging of common adjacent frames that call the same method. As a quick side note, the magenta frames in the flame graphs are those frames that match a search phrase. Their particular significance in these specific visualizations will be discussed later on.

Broken Stacks

Unfortunately 20% of what we see above contains broken stack traces for this specific workload. This is due to the complex call patterns in certain frameworks that generate extremely deep stacks, far exceeding the current 127 frame limitation in Linux perf_events (PERF_MAX_STACK_DEPTH). The long, narrow towers outlined below are the call stacks truncated as a result of this limitation:



We experimented with Java profilers to get around this stack depth limitation and were able to hack together a flame graph with just few percent of broken stacks. In fact we captured so many unique stacks that we had to increase the minimum column width from a default 0.1 to 1 in order to generate a reasonably-sized flame graph that a browser can render. The end result is an extremely tall flame graph of which this is only the bottom half:



Although there are still a significant number of broken stacks, we begin to see a more complete picture materialize with the full call stacks intact in most of the executing code.

Hot Spot Analysis

Without specific domain knowledge of the application code there are generally two techniques to finding potential hot spots in a CPU profile. Based on the default stack ordering in a flame graph, a bottom-up approach starts at the base of the visualization and advances upwards to identify interesting branch points where large chunks of the sampled cost either split into multiple subsequent call paths or simply terminate at the current frame. To maximize potential impact from an optimization, we prioritize looking at the widest candidate frames first before assessing the narrower candidates.

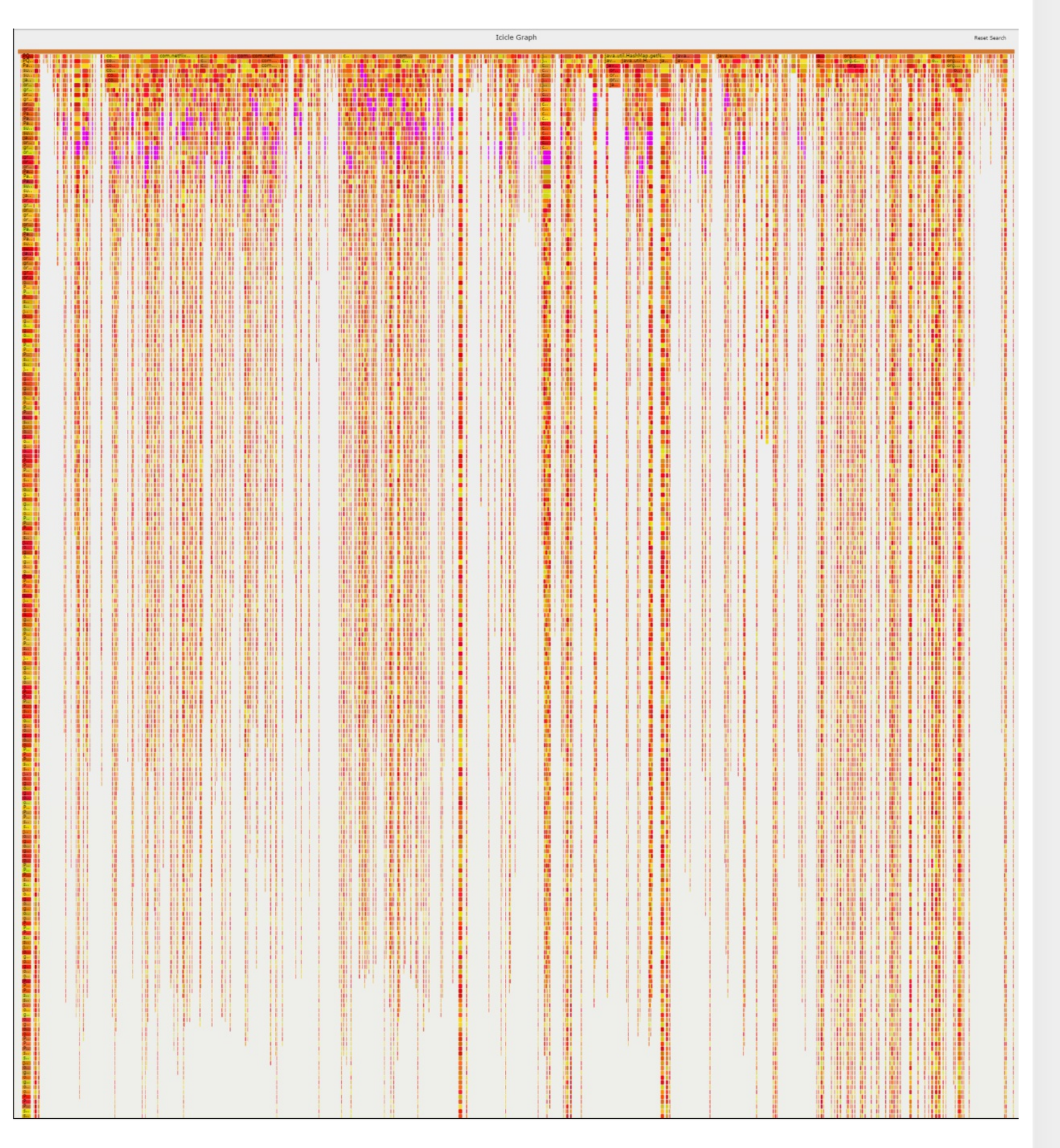
Using this bottom-up approach for the above flame graph, we derived the following observations:

- Most of the lower frames in the stacks are calls executing various framework code
- We generally find the interesting application-specific code in the upper portion of the stacks
- Sampled costs have whittled down to just a few percentage points at most by the time we reach the interesting code in the thicker towers

While it's still worthwhile to pursue optimizing some of these costs, this visualization doesn't point to any obvious hot spot.

Top-Down Approach

The second technique is a top-down approach where we visualize the sampled call stacks aggregated in reverse order starting from the leaves. This visualization is simple to generate with Brendan's flame graph script using the options `--inverted --reverse`, which merges call stacks top-down and inverts the layout. Instead of a flame graph, the visualization becomes an icicle graph. Here are the same stacks from the previous flame graph reprocessed with those options:



In this approach we start at the top-most frames again prioritizing the wider frames over the more narrow ones. An icicle graph can surface some obvious methods that are called excessively or are by themselves computationally expensive. Similarly it can help highlight common code executed in multiple unique call paths.

Analyzing the icicle graph above, the widest frames point to map get/put calls. Some of the other thicker columns that stand out are related to a known inefficiency within one of the frameworks utilized. However, this visualization still doesn't illustrate any specific code path that may produce a major win.

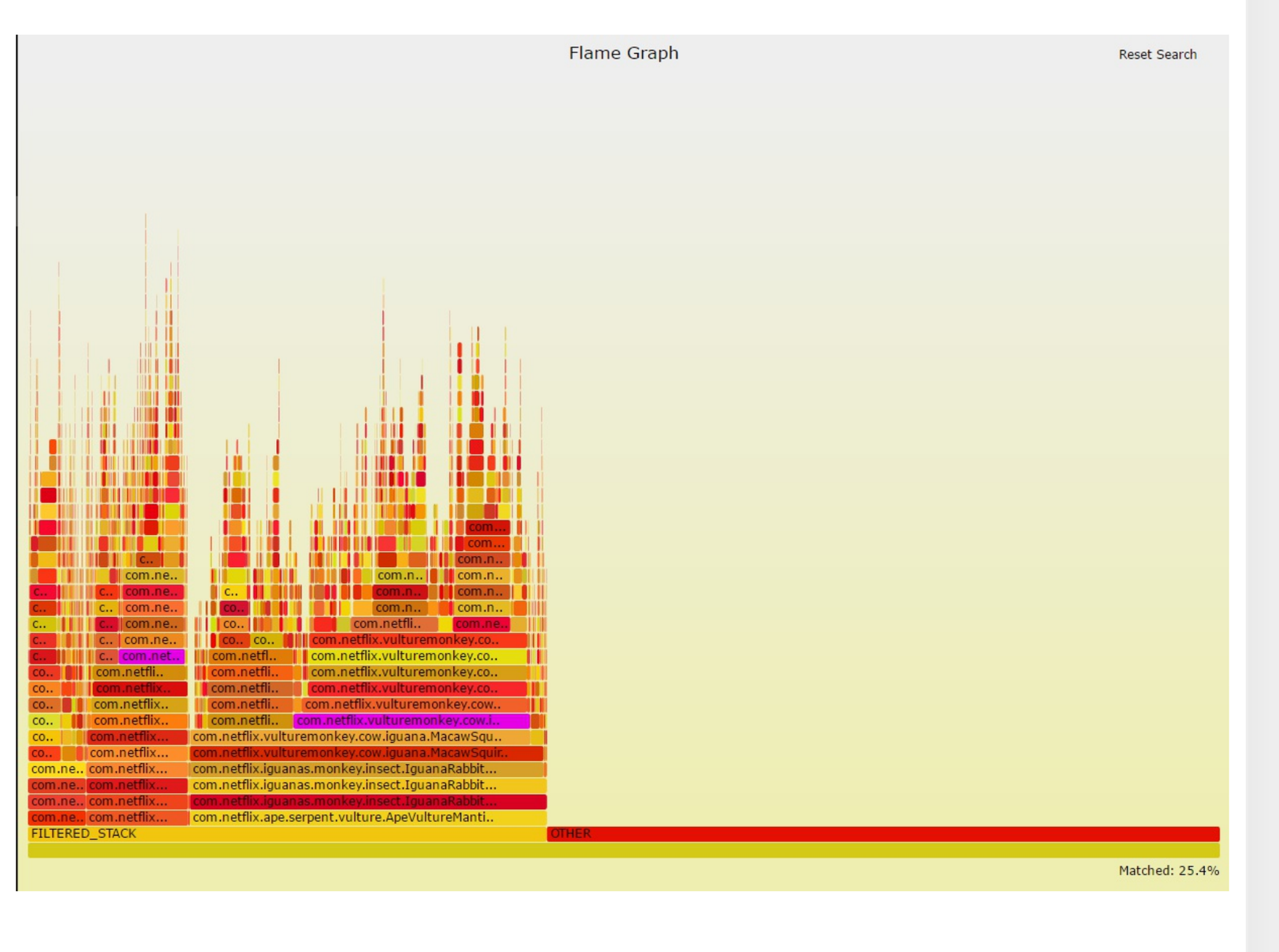
A Middle-Out Approach

Since we didn't find any obvious hotspots from the bottom-up and top-down approaches, we thought about how to improve our focus on just the application-specific code. We began by reprocessing the stack traces to collapse the repetitive framework calls that constitute most of the content in the tall stacks. As we iterated through removing the uninteresting frames, we saw some towers in the flame graph coalescing around a specific application package.

Remember those magenta-colored frames in the images above? Those frames match the package name we uncovered in this exercise. While it's somewhat arduous to visualize the package's cost because the matching frames are scattered throughout different towers, the flame graph's embedded search functionality attributed more than 30% of the samples to the matching package.

This cost was obscured in the flame graphs above because the matching frames are split across multiple towers and appear at different stack depths. Similarly, the icicle graph didn't fare any better because the matching frames diverge into different call paths that extend to varying stack heights. These factors defeat the merging of common frames in both visualizations for the bottom-up and top-down approaches because there isn't a common, consolidated pathway to the costly code. We needed a new way to tackle this problem.

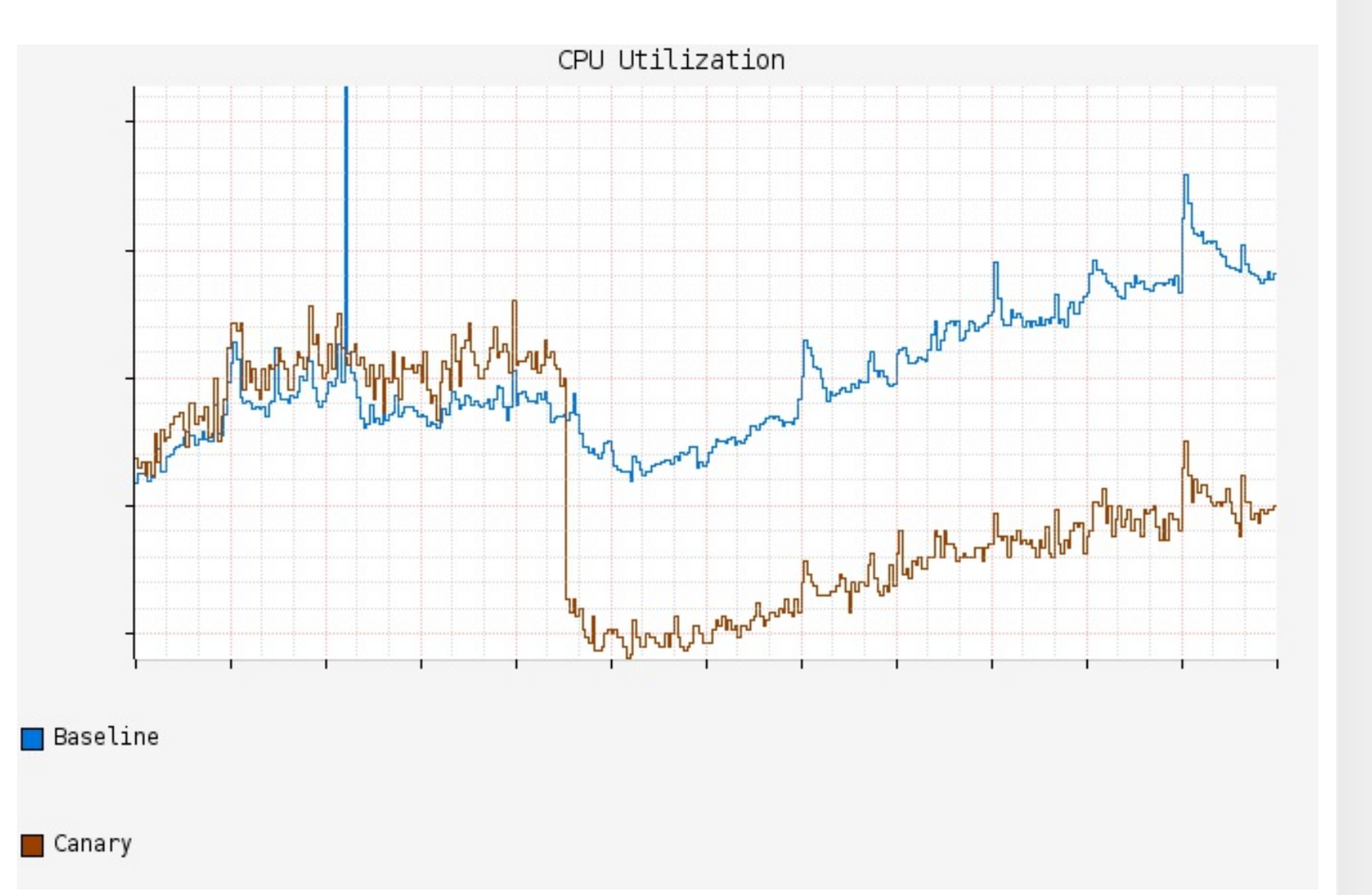
Given the discovery of the potentially expensive package above, we filtered out all the frames in each call stack until we reached a method calling this package. It's important to clarify that we are not simply discarding non-matching stacks. Rather, we are reshaping the existing stack traces in different and better ways around the frames of interest to facilitate stack merging in the flame graph. Also, to keep the matching stack cost in context within the overall sampled cost, we truncated the non-matching stacks to merge into an "OTHER" frame. The resulting flame graph below reveals that the matching stacks account for almost 44% of CPU samples, and we now have a high degree of clarity of the costs within the package:

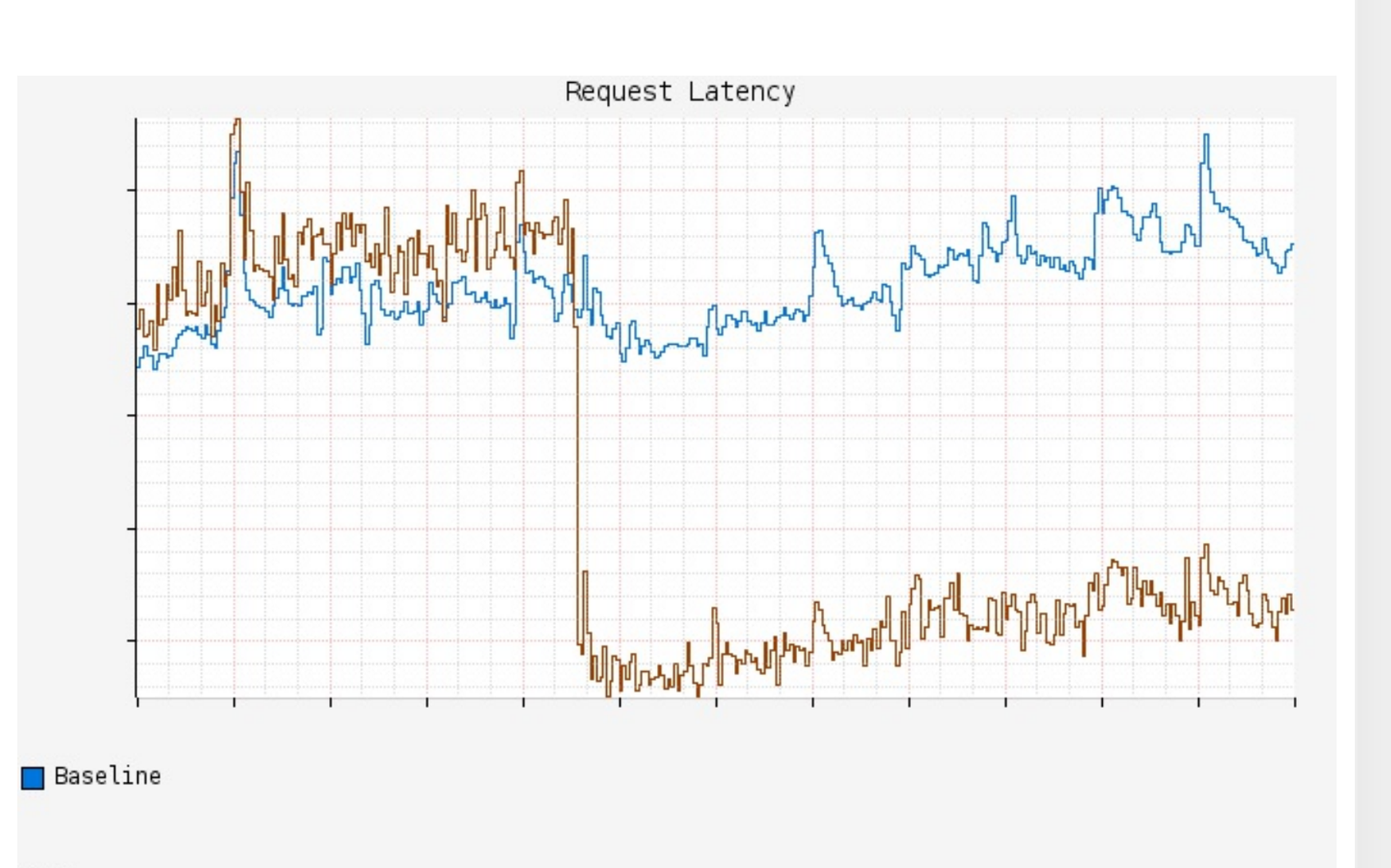


To explain how the matching cost jumped from 30% to 44% between the original flame graphs and the current one, we recall that earlier we needed to increase the column minimum width from 0.1 to 1 in the unfiltered flame graph. That surprisingly removed almost a third of the matching stack traces, which further highlights how the call patterns dispersed the package's cost across the profile. Not only have we simplified and consolidated the visualization for the package through reshaping the stack traces, but we have also improved the precision when accounting for its costs.

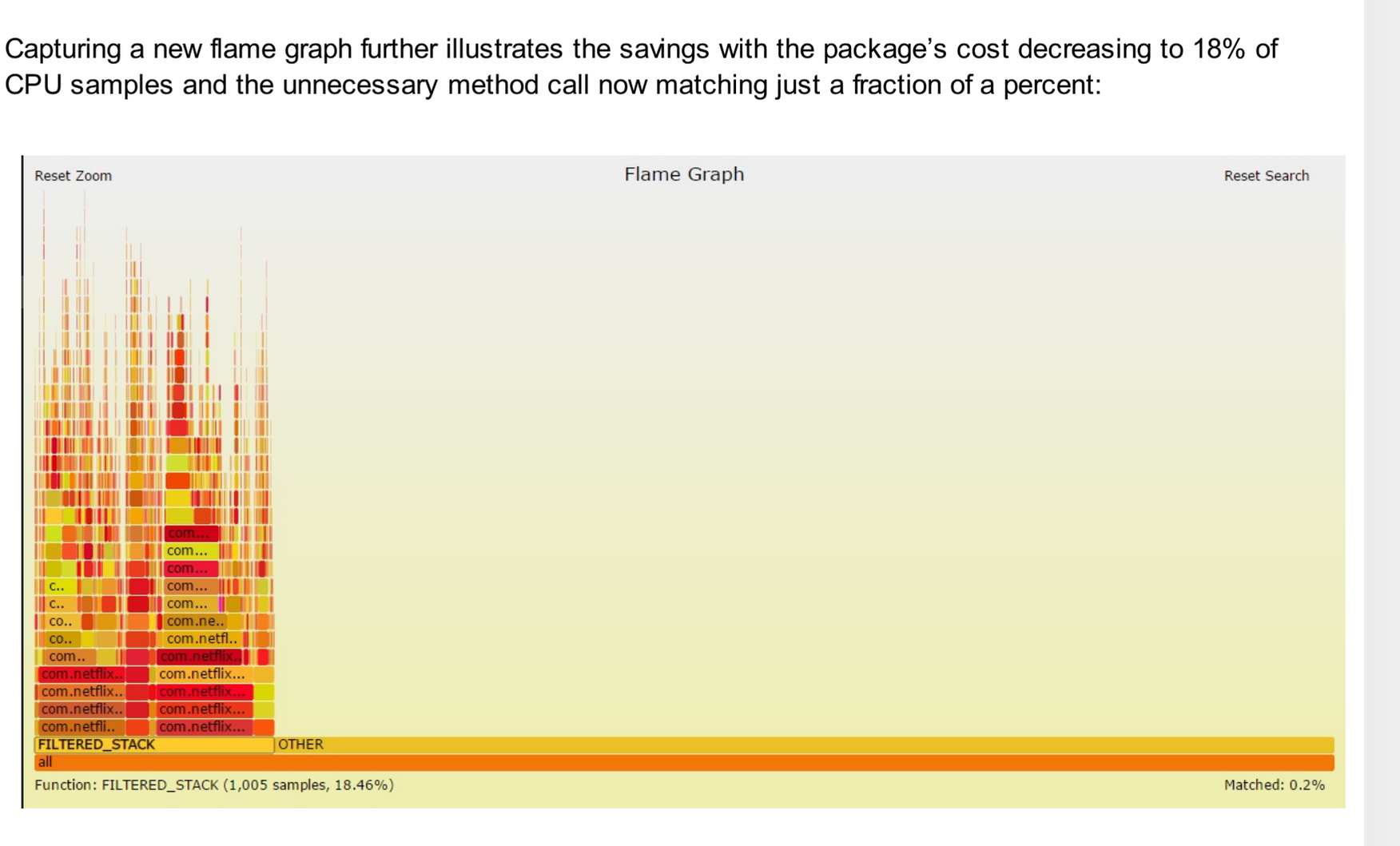
Hitting the Jackpot

Once we had this consolidated visualization, we began unraveling the application logic executing the call stack in the main tower. Reviewing this flame graph with the feature owners, it was quickly apparent that a method call into legacy functionality accounted for an unexpectedly large proportion of the profiled cost. The flame graph above highlights the suspect method in magenta and shows it matching 25% of CPU samples. As luck would have it, a dynamic flag already existed that gates the execution of this method. After validating that the method no longer returns distinct results, the code path was disabled in a canary cluster resulting in a dramatic drop in CPU usage and request latencies:





Capturing a new flame graph further illustrates the savings with the package's cost decreasing to 18% of CPU samples and the unnecessary method call now matching just a fraction of a percent:



Quantifying the actual CPU savings, we calculated that this optimization reduces the service's computation time by more than 13 million minutes (or almost 25 years) of CPU time per day.

Looking Forward

Aggregating costs based on a filter is admittedly not a new concept. The novelty of what we've done here is to maximize the merging of interesting frames through the use of variable stack trace filters within a flame graph visualization.

In the future we'd like to be able to define ad hoc inclusion or exclusion filters in a full flame graph to dynamically update a visualization into a more focused flame graph such as the one above. It will also be useful to apply inclusion filters in reverse stack order to visualize the call stacks leading into matching frames. In the meantime, we are exploring ways to intelligently generate filtered flame graphs to help teams visualize the cost of their code running within shared platforms.

Our goal as a performance team is to help scale not only the company by also our own capabilities. To achieve this we look to develop new approaches and tools for others at Netflix to consume in a self-service fashion. The flame graph work in this post is part of a profiling platform we have been building that can generate on-demand CPU flame graphs for Java and Node.js applications and is available 24x7 across all services. We'll be using this platform for automated regression analysis as well to push actionable data directly to engineering teams. Lots of exciting and new work in this space is coming up.

1 Comment   Sort by Oldest

Add a comment...

Jeff Turner
Absolutely brilliant explanation of why performance profiling is so important to IT effeciency. Well done, Brendan.
Like · Reply · 8 hrs

Facebook Comments Plugin

Posted by Mike Huang at 10:20 AM

G+1 +7 Recommend this on Google

Labels: efficiency, flamegraphs, optimization, performance

About the Netflix Tech Blog

This is a Netflix blog focused on technology and technology issues. We'll share our perspectives, decisions and challenges regarding the software we build and use to create the Netflix service.