Troubleshooting code bottlenecks

As a developer or service owner you’re responsible for writing new code, troubleshooting latency, and optimizing service performance. Code profiling provides visibility of code-level performance to help identify and isolate service bottlenecks. Code profiling works by periodically taking CPU snapshots, or call stacks, from a runtime environment, and visualizing them in flame graphs to help you easily understand which code negatively impacts service performance and customer experience the most.

The challenge though is that code profiling can incur notable performance overhead. Some profiling solutions require you to manually “switch on and off,” creating a tradeoff between application performance and available data to troubleshoot issues.

Data required

Build systems

How to use Splunk software for this use case

Splunk APM’s AlwaysOn Profiling for Java applications provides continuous monitoring and visibility of code-level performance, linked with unsampled trace data, with minimal overhead. AlwaysOn is included at no cost within Splunk APM.

Unlike dedicated code profiling solutions, Splunk’s AlwaysOn Profiler links collected call stacks to spans that are being executed at the time of call stack collection. This helps separate data about the background threads from active threads which service incoming requests, greatly reducing the amount of time you need to analyze profiling data.

Additionally, with Splunk’s AlwaysOn profiler, all of the data collection is automatic, and low overhead. Instead of having to switch the profiler on during production incidents, you only need to re-deploy Splunk’s Open Telemetry Collector and it begins to continuously collect data in the background.

Here are two examples of how AlwaysOn can help identify production issues.

Viewing common code in your slowest traces

If you are troubleshooting production issues, you’ll often sort through example traces looking for common attributes in their slowest spans. AlwaysOn’s call stacks are linked to trace data, providing context into which code is executed during each trace.

Within Splunk APM you can easily view latency within your production environment.
1. Open Splunk APM and click on any service to navigate to the service map, which provides additional context on bottlenecks within that service and its dependencies.

2. From here, you can click to explore example traces.

You can filter the "min" by 10,000, or ten seconds, to focus specifically on the slowest traces. Requests to /stats/races/fastest repeatedly respond in around 40+ seconds.

3. By clicking into one of these long traces, the following screen opens:

4. In the example below, you can see that while the StatsController.fastestRace operation was being executed, 21 call stacks were collected. As the Java agent continuously collects call stacks, the longer the spans, the more call stacks they will have. When you open this span, you'll see the metadata on the left, and the call stacks that the agent collected on the right. You can use the Previous and Next buttons to flip through all call stacks.
5. If you see several consecutive call stacks pointing to the same line of code, it indicates that these lines take a long time to execute, or execute many times in a row. This is often an indication of a performance bottleneck.

**Viewing aggregate performance of services over time**

Before you begin optimizing code, it’s always helpful to understand which part of your source code impacts performance the most. How do you know which part is the biggest bottleneck? This is where aggregation of collected call stacks, in the form of flame graphs, helps.

1. Open up your service map. When viewing your service map, notice the AlwaysOn Profiling addition on your right side panel, which automatically shows you the top five frames from the call stacks you’ve collected for your selected time range, that already point to bottlenecks in code.

2. By clicking into the feature, you’re taken to a flame graph, which is a visual aggregation of call stacks collected from the time range you’ve specified. Flame graphs visualize call stacks across a time range - the larger the horizontal bar, the more frequently that line of code is found in the collected call stacks. When viewing the flamegraph, focus on larger top down “pillars”, which indicate lines of code that use the CPU the most. If you want to highlight your own code classes in the flamegraph, use the filter in the top left.
3. Within each horizontal bar of the flame graph, there are class names and line numbers for your code. Flame graphs point you to the bottleneck causing the slowness, and the final step in troubleshooting is returning to your source code itself to fix the problem.

Next steps

The content in this guide comes from a previously published blog, one of the thousands of Splunk resources available to help users succeed. In addition, these Splunk resources might help you understand and implement this use case:

- Splunk Docs: Always-On Profiling documentation