[Java][Profiling] Async-profiler - manual by use cases

Just a homepage

[Java][Profiling] Async-profiler - manual by use cases

This blog post contains examples of async-profiler usages that I have found helpful in my job. Some content of this post is copy-pasted from previous entries, as I just wanted to avoid unnecessary jumps between articles.

The goal of that post is to give examples. It's not a replacement for the project README. I advise you to read it, as it tells you how to obtain async-profiler binaries and more.

All the examples that you are going to see here are synthetic reproductions of real-world problems that I solved during my career. Even if some examples look like "it's too stupid to happen anywhere," well, it isn't.

That post will be maintained. Whenever I find a new use case that I think is worth sharing, I will update this post.

Change log

- 2022-12-16 - Initial version

Acknowledgments

I would like to say thank you to Andrei Pangin (Lightrun) for all the work he did to create async-profiler and for his time and remarks on that article, Johannes Bechberger (SapMachine team at SAP) for all the work on making OpenJDK more stable with profilers, the input he gave me on overhead and stability, and the copy editing of this document, Marcin Grzejszczak (VMware) for great insight on how to integrate this profiler with Spring, Krystian Zybała for the review.

Profiled application

I've created a Spring Boot application for this post so that you can run the following examples on your own. It's available on my GitHub. To build the application, do the following:

```
git clone https://github.com/krzysztofslusarski/async-profiler-demos
cd async-profiler-demos
mvn clean package
```

To run the application, you need three terminals where you run the following (you need the ports 8081, 8082, and 8083 available):

```
java -Xms1G -Xmx1G -XX:+AlwaysPreTouch \
-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints \
-Duser.language=en-US \
-Xlog:safepoint,gc+humongous=trace \
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar

java -Xms1G -Xmx1G -XX:+AlwaysPreTouch \
-jar second-application/target/second-application-0.0.1-SNAPSHOT.jar

java -Xms1G -Xmx1G -XX:+AlwaysPreTouch \
-jar third-application/target/third-application-0.0.1-SNAPSHOT.jar
```

I'm using Corretto 17.0.2:

```
$ java -version

openjdk version "17.0.2" 2022-01-18 LTS
OpenJDK Runtime Environment Corretto-17.0.2.8.1 (build 17.0.2+8-LTS)
OpenJDK 64-Bit Server VM Corretto-17.0.2.8.1 (build 17.0.2+8-LTS, mixed mode, sl
```

And to create simple load tests, I'm using an ancient tool:

```
$ ab -V

This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

I will not go through the source code to explain all the details before I jump into examples. I often profile applications with source code I haven't seen in my work. Just consider it as a typical micro-service build with Spring Boot and Hibernate.

In all the examples, I will assume that the application is started with the `java -jar` command. If you are running the application from the IDE, then the name of the application is switched from `first-application-0.0.1-SNAPSHOT.jar` to `FirstApplication`.

All the JFRs generated while writing that post are available here.

How to run an Async-profiler

Command line

One of the easiest ways of running the async-profiler is using the command line. You just need to execute the following in the profiler folder:

```
./profiler.sh -e <event type> -d <duration in seconds> \
-f <output file name> <pid or application name>

# examples
./profiler.sh -e cpu -d 10 -f prof.jfr first-application-0.0.1-SNAPSHOT.jar
./profiler.sh -e wall -d 10 -f prof.html 1234 # where 1234 is the PID of the Jav
```

There are a lot of additional switches that are explained in the README.

You can also use async-profiler to output JFR files:

```
./profiler.sh start -e <event type> -f <output JFR file name> <pid or applicati
# do something
./profiler.sh stop -f <output JFR file name> <pid or application name>
```

For formats different from JFR, you need to pass the file name during `stop`, but for JFR it is needed during `start`.

WARNING: This way of attaching any profiler to a JVM is vulnerable to JDK-8212155. That issue can crash your JVM during attachment. It has been fixed in JDK 17.

If you are attaching a profiler this way, it is recommended to use `-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints` JVM flags (see this blog post by Jean-Philippe Bempel for more information on why these flags are essential).

During JVM startup

You can add a parameter when you are starting a `java` process:

```
java -agentpath:/path/to/libasyncProfiler.so=start,event=cpu,file=prof.jfr
```

The parameters passed this way differ from the switches used in the command line approach. You can find the list of parameters in the arguments.cpp file and the mapping between those in the profiler.sh file in the source code.

You can also attach a profiler without starting it using `-agentpath`, which is the safest way of starting your JVM if you want to profile it anytime soon.

From Java API

The Java API is published to maven central. All you need to do is to include a dependency:

```
<dependency>
  <groupId>tools.profiler</groupId>
  <artifactId>async-profiler</artifactId>
  <version>2.9</version>
</dependency>
```

That gives you an API where you can use Async-profiler from Java code. Example usage:

```
AsyncProfiler profiler = AsyncProfiler.getInstance();
```

That gives you an instance of `AsyncProfiler` object, with which you can send orders to the profiler:

```
profiler.execute(String.format("start,jfr,event=wall,file=%s.jfr", fileName));
// do something, like sleep
profiler.execute(String.format("stop,file=%s.jfr", fileName));
```

Since async-profiler 2.9, the `AsyncProfiler.getInstance()` extracts and loads the `libasyncProfiler.so` from the JAR. In the previous version, this file needed to be in one of the following directories:

- `/usr/java/packages/lib`
- `/usr/lib64`
- `/lib64`
- `/lib`
- `/usr/lib`

You can also point to any location of that file with API:

```
AsyncProfiler profiler = AsyncProfiler.getInstance("/path/to/libasyncProfiler.s
```

From JMH benchmark

It's worth mentioning that the async-profiler is supported in JMH benchmarks. If you have one, you just need to run the following:

```
java -jar benchmarks.jar -prof async:libPath=/path/to/libasyncProfiler.so\;outpu
```

JMH will take care of every magic, and you get proper JFR files from async-profiler.

AP-Loader

There is a pretty new project called AP-Loader by Johannes Bechberger that can also be helpful to you. This project packages all native distributions into a single JAR, so It is convenient when deploying on different CPU architectures. You can also use the Java API with this loader without caring where the binary of the profiler is located and which platform you're running on. I recommend reading the README of that project. It may be suitable for you.

IntelliJ Idea Ultimate

If you are using IntelliJ Idea Ultimate, you have a built-in async-profiler at your fingertips. You can profile any JVM running on your machine and visualize the results. Honestly, I don't use it that much. Most of the time, I run profilers on remote machines, and I've got used to it, so I run it the same way on my localhost.

Output formats

Async-profiler gives you a choice of how the results should be saved:

- default - printing results to the terminal
- JFR
- Collapsed stack
- Flame graphs
- ...

From that list, I choose JFR 95% of the time. It's a binary format containing all the information gathered by the profiler. That file can be post-processed later by some external tool. I'm using my own open-sourced JVM profiling toolkit, which can read JFR files with additional filters and gives me the possibility to add/remove additional levels during the conversion to a flame

graph. I will use the filter names of my viewer in the following and the names of filters from my viewer. There are other products (including the `jfr2flame` converter that is a part of async-profiler) that can visualize the JFR output, but you should use the tool that works for you. None worked for me, so I wrote my own, but it doesn't mean it is the best choice for everybody.

All flame graphs in this post are generated by my tool from JFR files. The JFR file format for each sample contains the following:

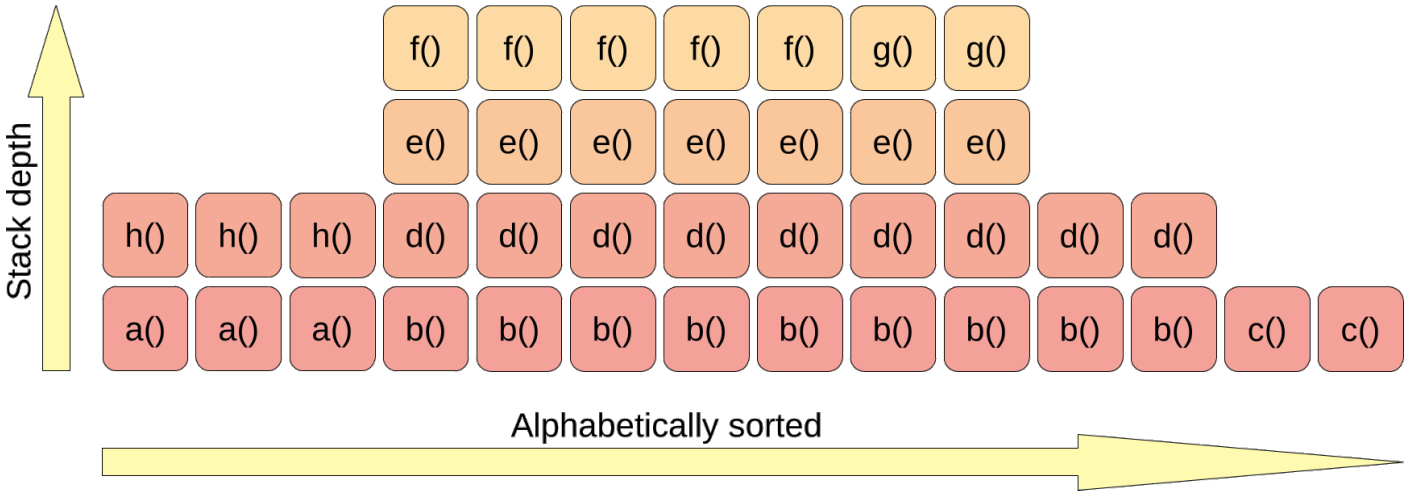
- Stack trace
- Java thread name
- Thread state (is the Java thread consumes CPU)
- Timestamp
- Monitor class - for `lock` mode
- Waiting for lock duration - for `lock` mode
- Allocated object class - for `alloc` mode
- Allocated object size - for `alloc` mode
- Context ID - if you are using async-profiler with Context ID PR merged

So all the information is already there. We just need to extract what we need and present it visually.

Flame graphs

If you do **sampling profiling** you need to visualize the results. The results are nothing more than a **set of stack traces**. My favorite visualization is a **flame graph**. The easiest way to understand what flame graphs are is to know how they are created.

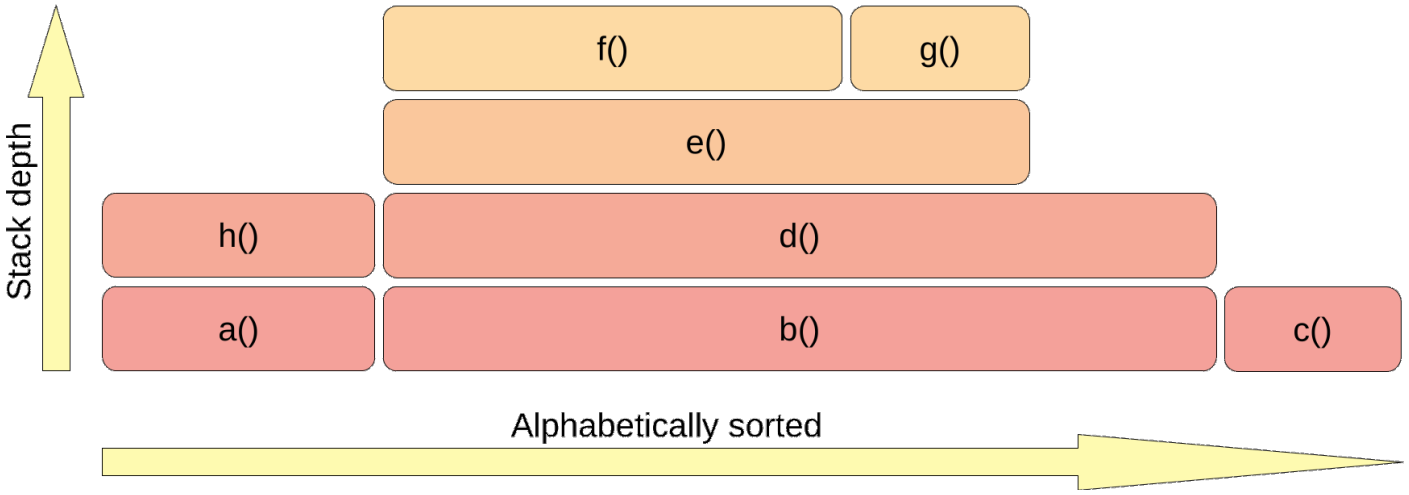
First, we draw a rectangle for each frame of each stack trace. The stack traces are drawn bottom-up and sorted alphabetically. For example, such a graph looks like this:



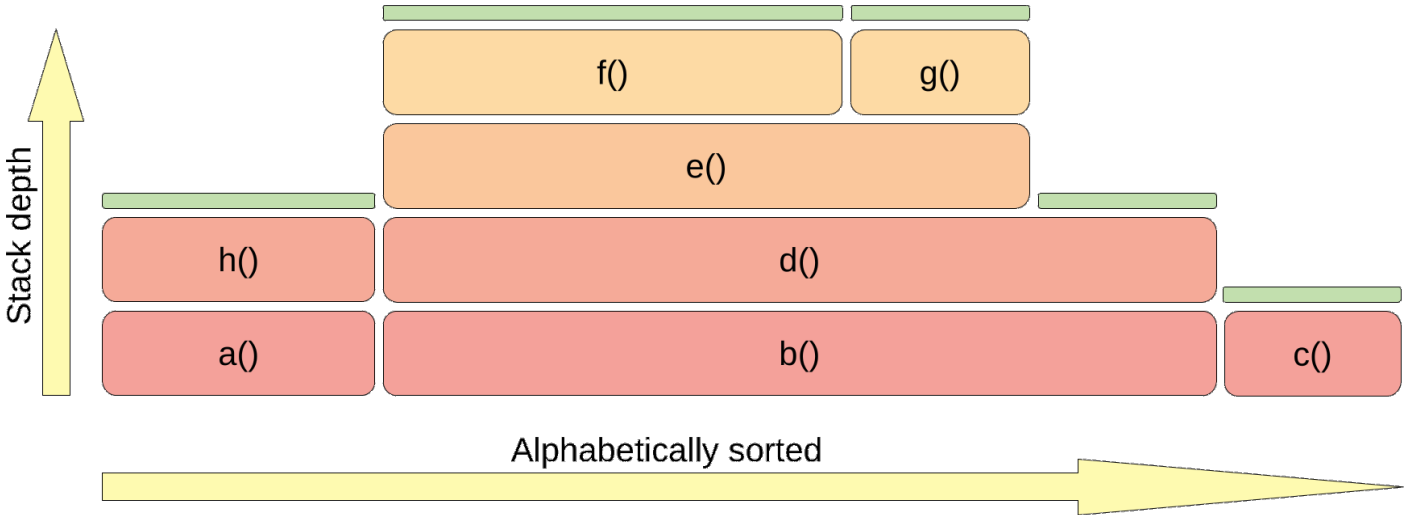
This corresponds to the following set of stack traces:

- 3 samples - `a() -> h()`
- 5 samples - `b() -> d() -> e() -> f()`
- 2 samples - `b() -> d() -> e() -> g()`
- 2 samples - `b() -> d()`
- 2 samples - `c()`

The next step is **joining** the rectangles with the same method name to one bar:



The flame graph usually shows you how your application utilizes a specific resource. The resource is utilized **by the top methods** of that graph (visualized with green bar):



So in this example, method `b()` does not utilize the resource. It just invokes methods that transitively use it. Flame graphs are commonly used for the **CPU utilization**, but the CPU is just one of the resources we can visualize this way. If you use **wall-clock mode**, your resource is **time**. If you use **allocation mode**, then your resource is **heap**. If you want to learn more about flame graphs, you can check Brendan's Gregg video, he invented flame graphs.

Basic resources profiling

Before you start any profiler, the first thing you need to know is what your goal is. Only after that can you choose the proper mode of async-profiler. Let's start with the basics.

Wall-clock

If your goal is to optimize time, you should run the async-profiler in wall-clock mode. This is the most common mistake made by engineers starting their journey with profilers. The majority of applications that I profiled so far were applications that were working with a distributed architecture, using some DBs, MQ, Kafka, ... In such applications, the majority of time is spent on IO - waiting for other services/DB/... to respond. During such actions, Java is not using the CPU.

```
# warmup
ab -n 100 -c 4 http://localhost:8081/examples/wall/first
ab -n 100 -c 4 http://localhost:8081/examples/wall/second

# profiling of the first request
./profiler.sh start -e cpu -f first-cpu.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 100 -c 4 http://localhost:8081/examples/wall/first
./profiler.sh stop -f first-cpu.jfr first-application-0.0.1-SNAPSHOT.jar

./profiler.sh start -e wall -f first-wall.jfr first-application-0.0.1-SNAPSHOT.j
ab -n 100 -c 4 http://localhost:8081/examples/wall/first
./profiler.sh stop -f first-wall.jfr first-application-0.0.1-SNAPSHOT.jar

# profiling of the second request
./profiler.sh start -e cpu -f second-cpu.jfr first-application-0.0.1-SNAPSHOT.ja
ab -n 100 -c 4 http://localhost:8081/examples/wall/second
./profiler.sh stop -f second-cpu.jfr first-application-0.0.1-SNAPSHOT.jar

./profiler.sh start -e wall -f second-wall.jfr first-application-0.0.1-SNAPSHOT
ab -n 100 -c 4 http://localhost:8081/examples/wall/second
./profiler.sh stop -f second-wall.jfr first-application-0.0.1-SNAPSHOT.jar
```

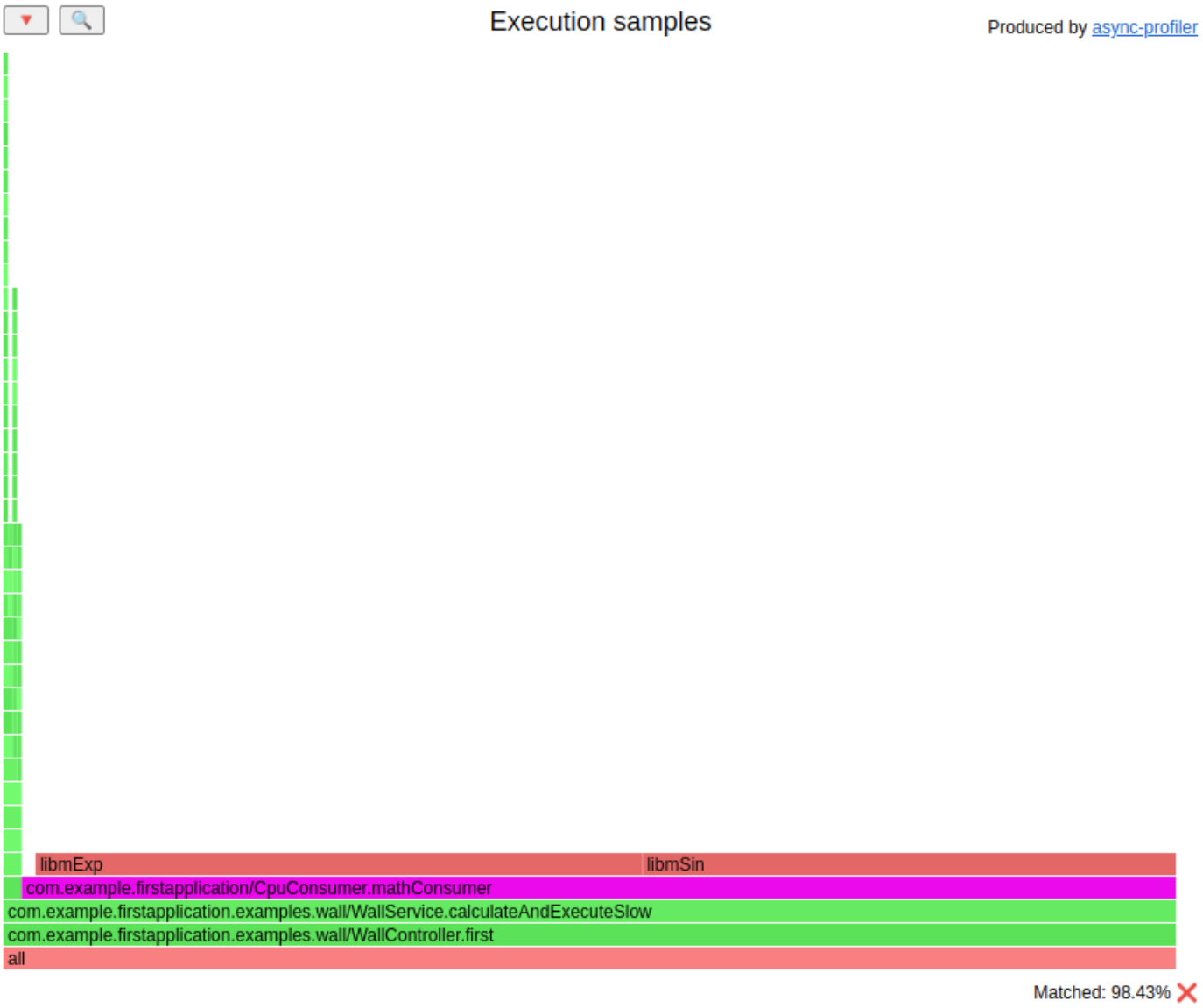
In the `ab` output, we can see that the basic stats are similar for each request:

```
# first request
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:   521  654 217.9    528    1055
Waiting:      521  654 217.9    528    1054
Total:        521  654 217.9    528    1055

# second request
```

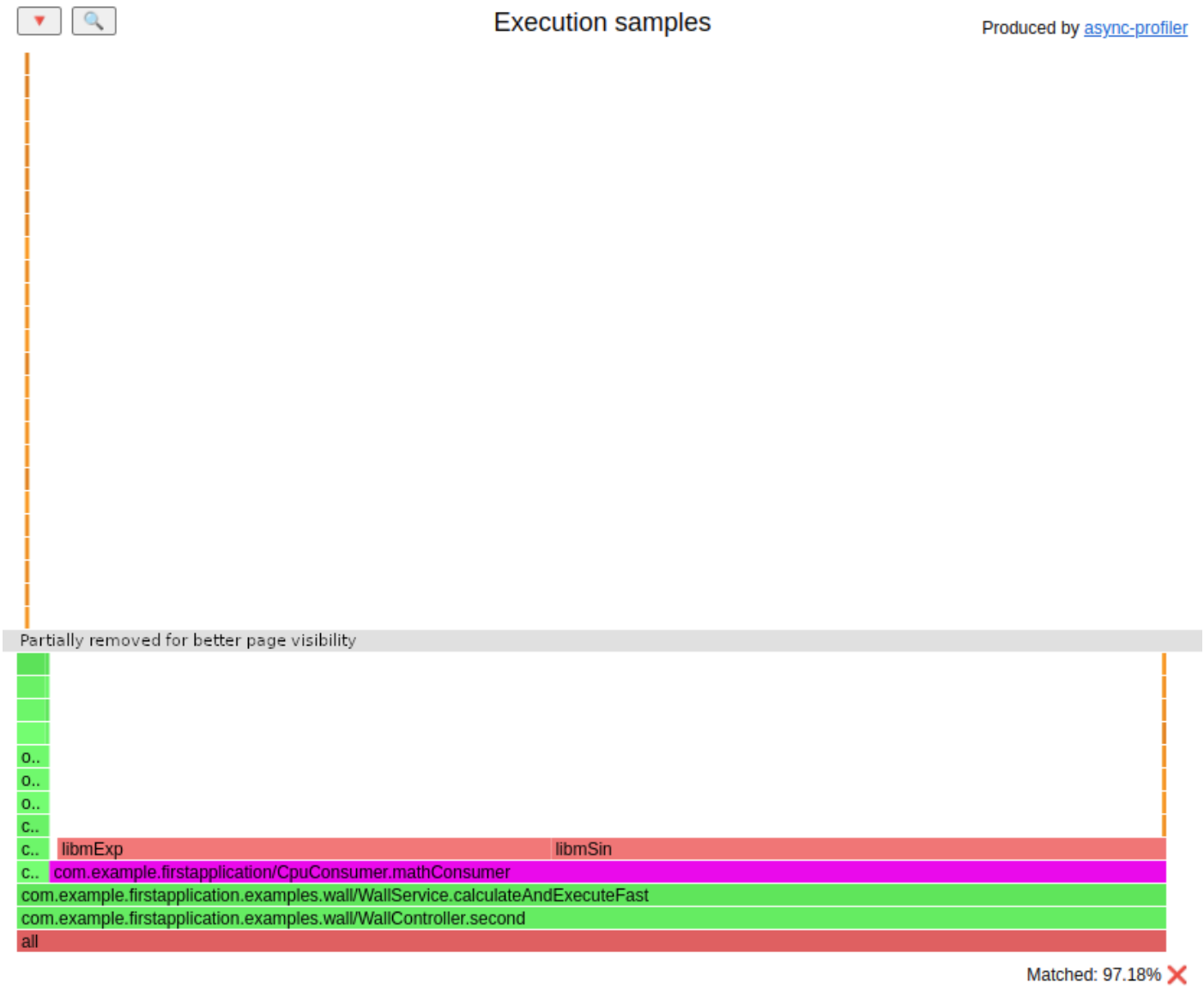
```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:   522  665 190.2    548    1028
Waiting:      522  665 190.1    548    1028
Total:        522  665 190.2    549    1029
```

To give you a taste of how the CPU profile can mislead you, here are flame graphs for those two executions in CPU mode.

First execution: (HTML)



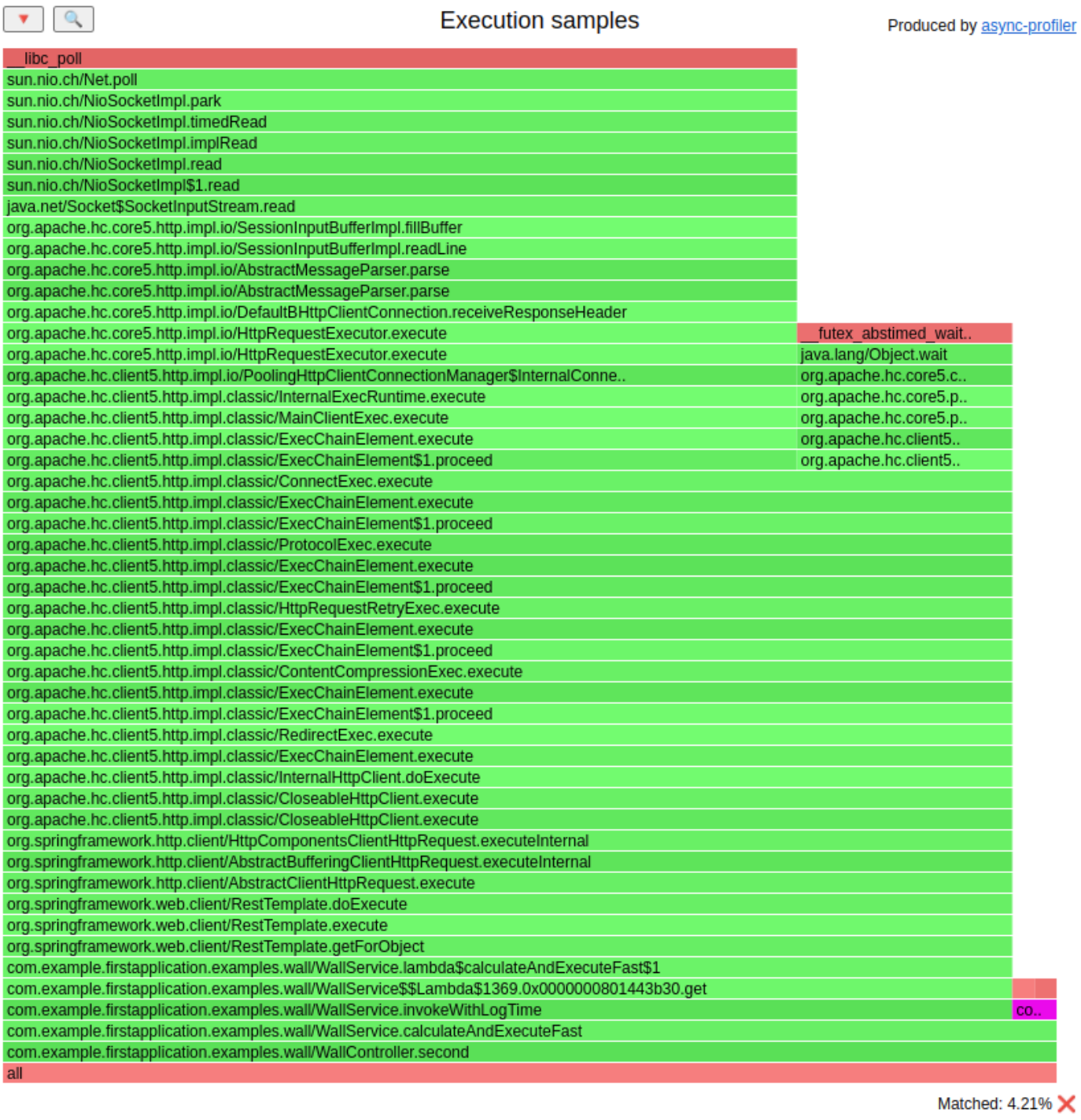
Second execution: (HTML)



I agree that they are not identical. But they have one thing in common. They show that the most CPU-consuming method invoked by my controller is `CpuConsumer.mathConsumer()`. It is not a lie: It consumes CPU. But does it consume most of the time of the request? Look at the flame graphs in wall-clock mode:

First execution: (HTML)

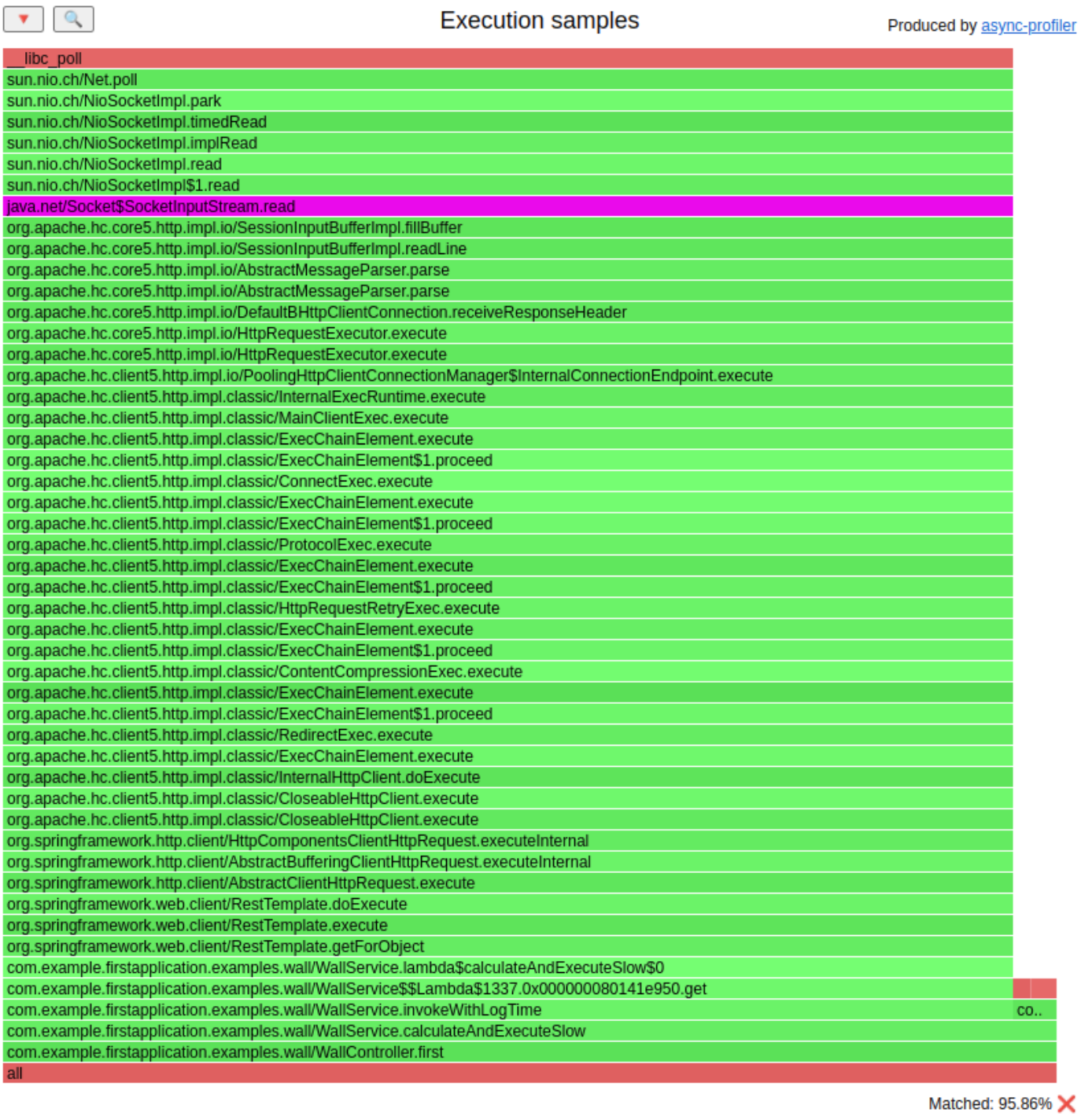
Second execution: (HTML)



I highlighted the ```CpuConsumer.mathConsumer()`` method. Wall-clock mode shows us that this method is responsible just for **~4%** of execution time.

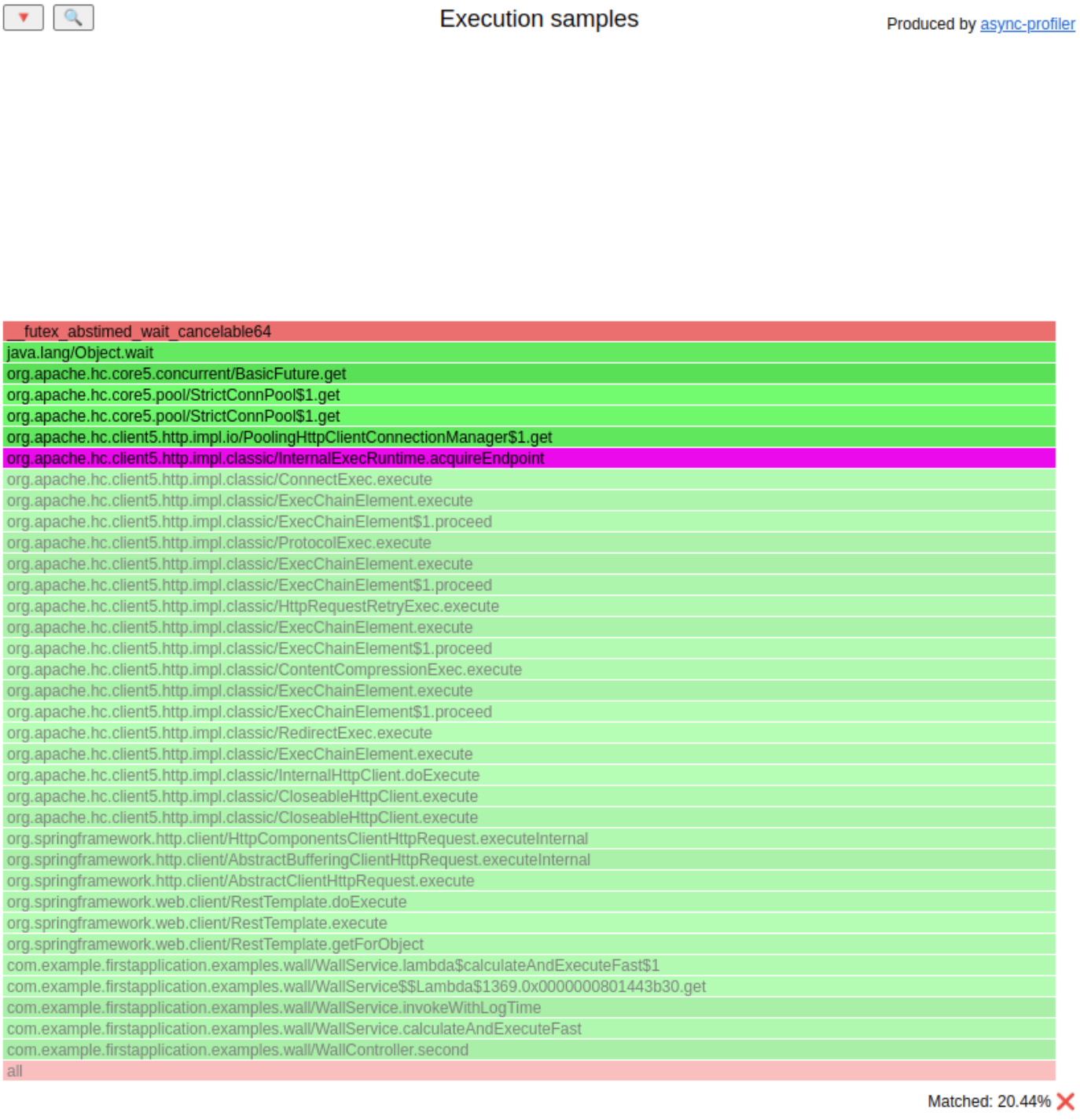
Thing to remember: if your goal is to optimize time, and you use external systems (including DBs, queues, topics, microservices) or locks, sleeps, disk IO, It would help if you started with wall-clock mode.

In wall-clock mode, we can also see that these flame graphs differ. The first execution spends most of its time in `SocketInputStream.read()`:



Over **95%** of the time is consumed there. But the second execution:

spends just **75%** on the socket. To the right of the method `SocketInputStream.read()` you can spot an additional bar. Let's zoom in:



It's the `InternalExecRuntime.acquireEndpoint()` method, which executes `PoolingHttpClientConnectionManager$1.get()` from Apache HTTP Client, which in the end executes `Object.wait()`. What does it do? Basically, what we are trying to do in those two executions is to invoke a remote REST service. The first execution uses an HTTP Client instance with `20` available connections, so no thread needs to wait for a connection from the pool:

```
// FirstApplicationConfiguration
@Bean("pool20RestTemplate")
RestTemplate pool20RestTemplate() {
    return createRestTemplate(20);
```

```
}
```

```
// WallService
void calculateAndExecuteSlow() {
    Random random = ThreadLocalRandom.current();
    CpuConsumer.mathConsumer(random.nextDouble(), CPU_MATH_ITERATIONS);

    invokeWithLogTime(() ->
        pool20RestTemplate.getForObject(SECOND_APPLICATION_URL +
            "/examples/wall/slow", String.class)
    );
}
```

The time spent on a socket is entirely spent waiting for the REST endpoint to respond. The second execution uses a different instance of `RestTemplate` that has just **3** connections in the pool. Since the load is generated from **4** threads by the `ab`, one thread must wait for a connection from the pool. You may think this is a stupid human error, that someone created a pool without enough connections. In the real world, the problem is with defaults that are quite low. In our testing application, the default settings for the thread pool are:

- `maxTotal` - **25** connections totally in the pool
- `defaultMaxPerRoute` - **5** connections to the same address

That number varies between versions. I remember one application with HTTP Client 4.x with defaults set to **2**.

There are plenty of tools that log the invocation time of external services. The common problem in those tools is that the time waiting on the pool for a connection is usually included in the invocation time, which is a lie. I saw this in the past when the caller had a line in the logs that gave an execution time of `X ms`; the callee had a similar log that presented `1/10 * X ms`. What were those teams doing to understand that? They tried to convince the network department that this was a network issue. Big waste of time.

I also saw plenty of custom logic that traced external execution time. In our application you can see such a pattern:

```
void calculateAndExecuteFast() {
    // ...
    invokeWithLogTime(() ->
            pool3RestTemplate.getForObject(SECOND_APPLICATION_URL + "/examples/v
    );
}

private <T> T invokeWithLogTime(Supplier<T> toInvoke) {
    StopWatch stopWatch = new StopWatch();
```

```
    stopWatch.start();
    T ret = toInvoke.get();
    stopWatch.stop();

    log.info("External WS invoked in: {}ms", stopWatch.getTotalTimeMillis());
    return ret;
}
```

The logs don't trace the time using an external service; the time also includes all the magic done by Spring, including waiting for a connection from the pool. You can easily see that for the second request, the logs look like this:

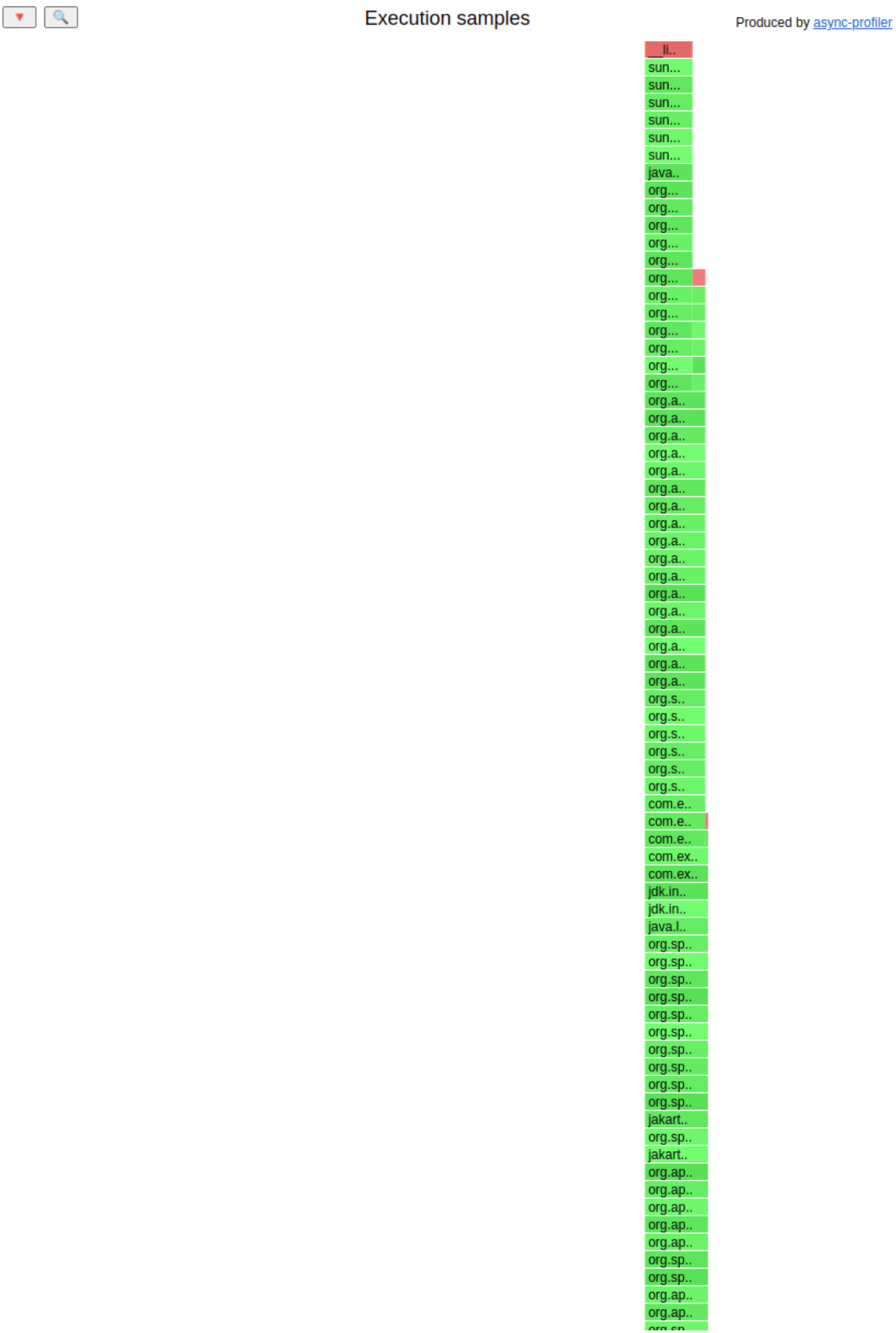
```
External WS invoked in: 937ms
```

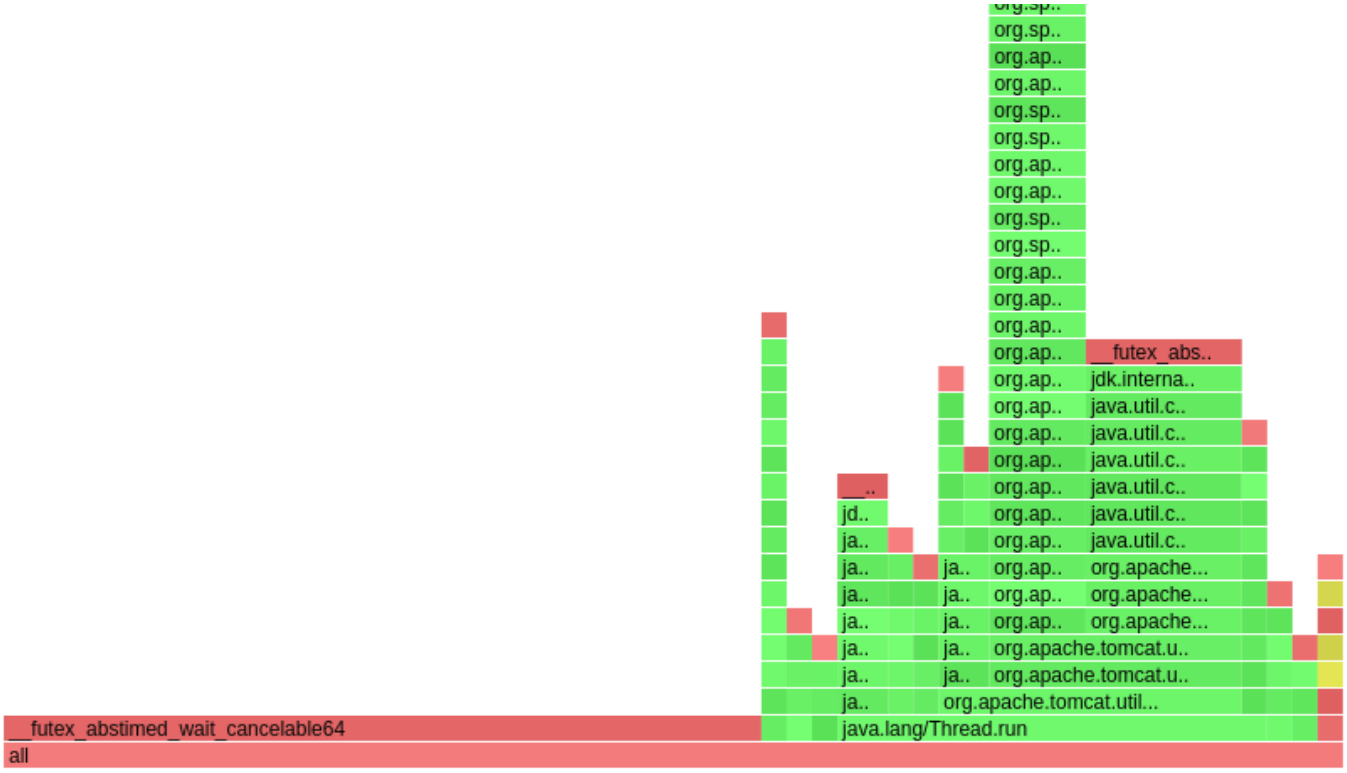
But the second service that is invoked is:

```
    @GetMapping("/fast")
    String fast() throws InterruptedException {
        Thread.sleep(500);
        return "OK";
    }
```

Wall-clock - filtering

If you open the wall-clock flame graph for the first time, you may be confused, since it usually looks like this:


Execution samples
Produced by async-profiler
li..
sun...
sun...
sun...
sun...
sun...
sun...
java..
org...
org...
org...
org...
org...
org...
org...
org...
org...
org...
org...
org...
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.a..
org.s..
org.s..
org.s..
org.s..
org.s..
org.s..
com.e..
com.e..
com.e..
com.ex..
com.ex..
jdk.in..
jdk.in..
java.l..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
org.sp..
jakart..
org.sp..
jakart..
org.ap..
org.ap..
org.ap..
org.ap..
org.ap..
org.ap..
org.sp..
org.sp..
org.ap..
org.ap..



You see all the threads, even if they are sleeping or waiting in some queue for a job. Wall-clock shows you all of them. Most of the time, you want to focus on the frames where your application is doing something, not when it is waiting. All you need to do is to filter the stack traces. If you are using Spring Boot with the embedded Tomcat, you can filter stack traces that contain the `SocketProcessorBase.run` method. In my viewer, you can just paste it to *stack trace filter*, and you are done. It's just a matter of proper filtering if you want to focus on one controller, class, method, etc.

CPU - easy-peasy

If you know that your application is CPU intensive, or you want to decrease CPU consumption, then the CPU mode is suitable.

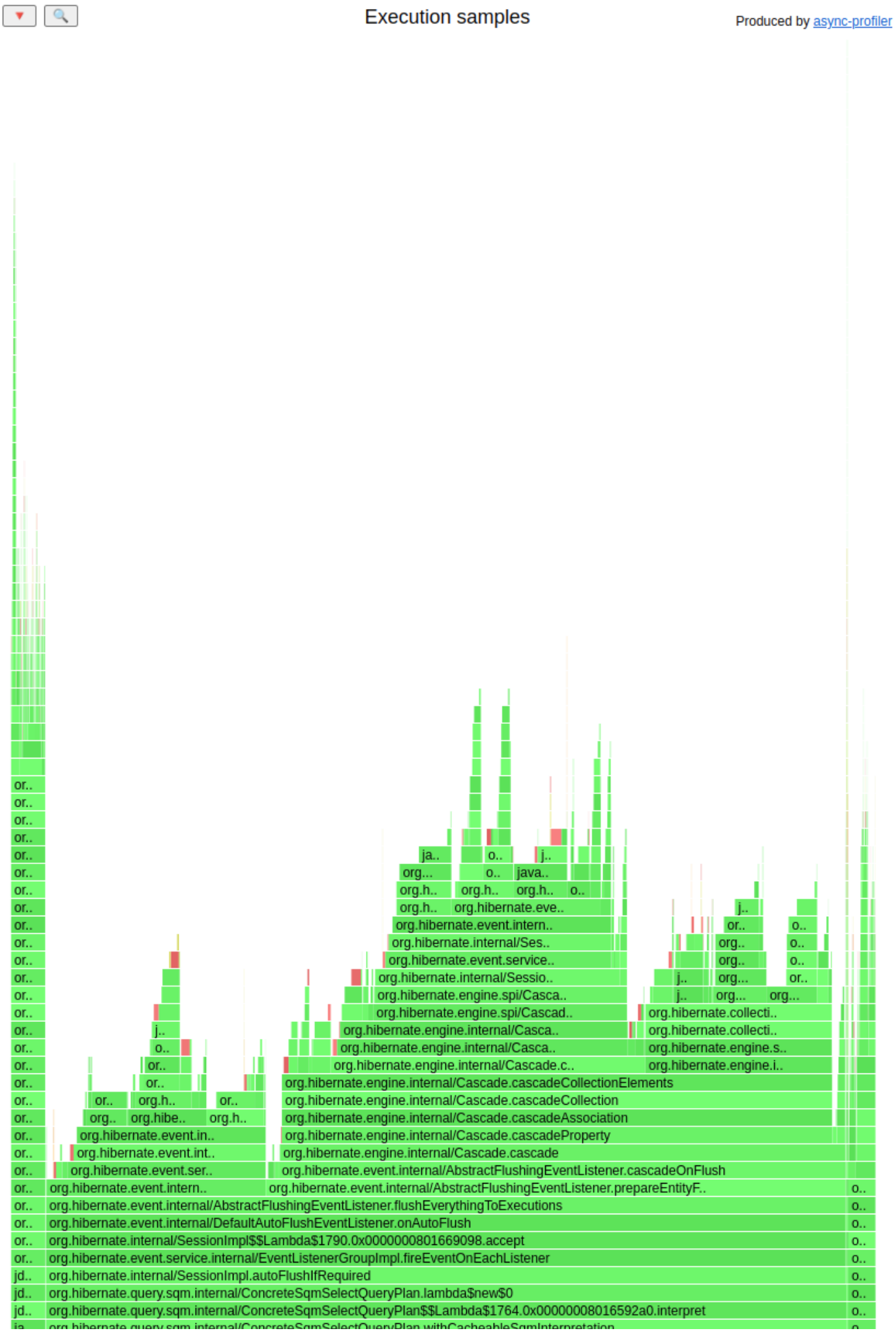
Let's prepare our application:

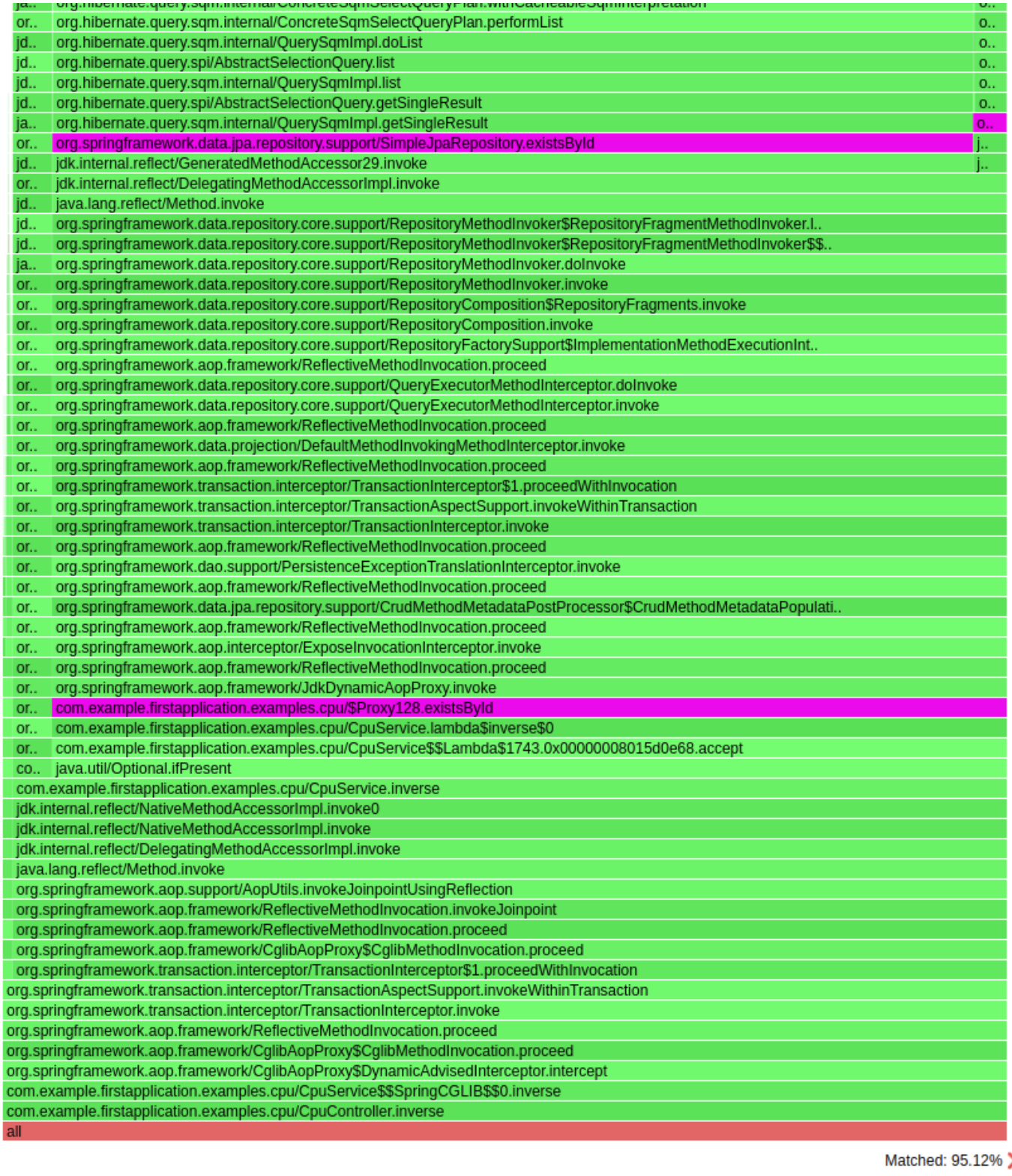
```
# execute it once
curl -v http://localhost:8081/examples/cpu/prepare

# little warmup
ab -n 5 -c 1 http://localhost:8081/examples/cpu/inverse

# profiling time:
./profiler.sh start -e cpu -f cpu.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 5 -c 1 http://localhost:8081/examples/cpu/inverse
./profiler.sh stop -f cpu.jfr first-application-0.0.1-SNAPSHOT.jar
```

You can check during the benchmark what the CPU utilization of our JVM is:

```
$ pidstat -p `pgrep -f first-application-0.0.1-SNAPSHOT.jar` 5

09:42:49      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Comma
09:42:54     1000     49813  115,40    0,40    0,00    0,00  115,80     1  java
09:42:59     1000     49813  111,40    0,60    0,00    0,00  112,00     1  java
09:43:04     1000     49813  106,80    0,20    0,00    0,00  107,00     1  java
09:43:09     1000     49813  113,00    0,20    0,00    0,00  113,20     1  java
```

We are using a bit more than one CPU core. Our load generator creates the load with a single thread so that the CPU usage is pretty high. Let's see what our CPU is doing while executing our spring controller: (HTML)


Execution samples
Produced by async-profiler
org.hibernate.event.intern..
org.hibernate.event.internal/AbstractFlushingEventListener.prepareEntityF..
org.hibernate.event.internal/AbstractFlushingEventListener.flushEverythingToExecutions
org.hibernate.event.internal/DefaultAutoFlushEventListener.onAutoFlush
org.hibernate.internal/SessionImpl$$Lambda$1790.0x0000000801669098.accept
org.hibernate.event.service.internal/EventListenerGroupImpl.fireEventOnEachListener
org.hibernate.internal/SessionImpl.autoFlushIfRequired
org.hibernate.query.sqm.internal/ConcreteSqmSelectQueryPlan.lambdanew0
org.hibernate.query.sqm.internal/ConcreteSqmSelectQueryPlan$$Lambda$1764.0x00000008016592a0.interpret
org.hibernate.event.ser..
org.hibernate.event.int..
org.hibernate.event.in..
org.hibernate.engine.internal/Cascade.cascadeCollectionElements
org.hibernate.engine.internal/Cascade.cascadeCollection
org.hibernate.engine.internal/Cascade.cascadeAssociation
org.hibernate.engine.internal/Cascade.cascadeProperty
org.hibernate.engine.internal/Cascade.cascade
org.hibernate.event.internal/AbstractFlushingEventListener.cascadeOnFlush
org.hibernate.engine.internal/Cascade.c..
org.hibernate.engine.internal/Casca..
org.hibernate.engine.spi/Cascad..
org.hibernate.engine.spi/Casca..
org.hibernate.internal/Sessio..
org.hibernate.event.service..
org.hibernate.internal/Ses..
org.hibernate.event.intern..
org.hibernate.eve..
org.hibernate.collecti..
org.hibernate.engine.s..
org.hibernate.engine.i..



I know that flame graph is pretty large, but hey, welcome to Spring and Hibernate. I highlighted the `existsById()` method. You can see that it consumes **95%** of the CPU time. But why? It doesn't look scary at all when looking at the code:

```
@Transactional
public void inverse(UUID uuid) {
    sampleEntityRepository.findById(uuid).ifPresent(sampleEntity -> {
        boolean allConfigPresent = true;
        for (int i = 0; i < 100; i++) {
            allConfigPresent = allConfigPresent && sampleConfigurationRepository
```

```
        }
        sampleEntity.setFlag(!sampleEntity.isFlag());
    });
}
```

We are just executing `existsById()` on the Spring Data JPA repository. The answer why that method is slow is at the beginning of the method:

```
sampleEntityRepository.findById(uuid)
```

and in JPA mapping:

```
public class SampleEntity {
    @Id
    private UUID id;

    @Fetch(FetchMode.JOIN)
    @JoinColumn(name = "fk_entity")
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, orphanRemoval
    private Set<SampleSubEntity> subEntities;

    private boolean flag;
}
```

This means that when we get one `SampleEntity` by id, we also extract the `subEntities` from the database because of `fetch = FetchType.EAGER`. This is not a problem yet. All JPA entities are loaded into the Hibernate session. That mechanism is pretty cool because it gives you the *dirty checking* functionality. The downside, however, is that the *dirty* entities need to be flushed by Hibernate to DB. You have different flush strategies in Hibernate. The default one is `AUTO`: you can read about them in the Javadocs. What you see in the flame graph is exactly Hibernate looking for dirty entities that should be flushed.

What can we do about this? Well, first of all, it should be forbidden to develop a large Hibernate application without reading Vlad Mihalcea's book. If you are developing such an application, buy that book, it's great. From my experience, some engineers tend to abuse Hibernate. Let's look at the code sample I pasted before. We are loading a huge `SampleEntity`. What are we doing with it?

```
@Transactional
public void inverse(UUID uuid) {
    sampleEntityRepository.findById(uuid).ifPresent(sampleEntity -> {
        // irrelevant
        sampleEntity.setFlag(!sampleEntity.isFlag());
    });
```

```
}
```

So we're basically changing one column in one row in the DB. We can do it more efficiently by using the `update` query, even with Spring Data JPA repository or simple JDBC. But do we really need to use Hibernate everywhere?

CPU - a bit harder

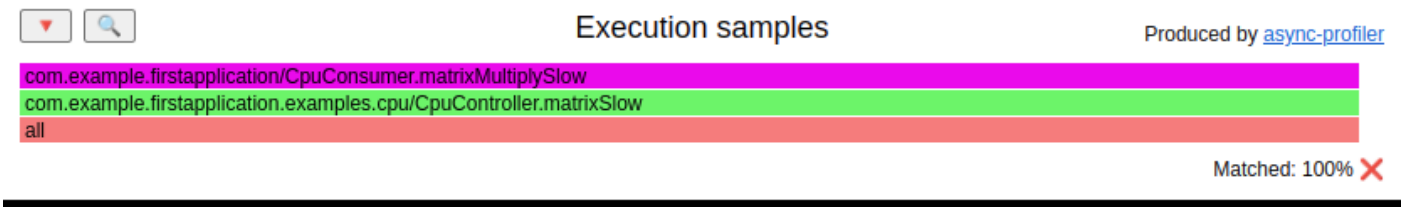
Sometimes the result of a CPU profiler is just the beginning of the fun. Let's consider the following example:

```
# little warmup
ab -n 10 -c 1 http://localhost:8081/examples/cpu/matrix-slow

# profiling time
./profiler.sh start -e cpu -f matrix-slow.jfr first-application-0.0.1-SNAPSHOT.j
ab -n 10 -c 1 http://localhost:8081/examples/cpu/matrix-slow
./profiler.sh stop -f matrix-slow.jfr first-application-0.0.1-SNAPSHOT.jar

# little warmup
ab -n 10 -c 1 http://localhost:8081/examples/cpu/matrix-fast

# profiling time
./profiler.sh start -e cpu -f matrix-fast.jfr first-application-0.0.1-SNAPSHOT.j
ab -n 10 -c 1 http://localhost:8081/examples/cpu/matrix-fast
./profiler.sh stop -f matrix-fast.jfr first-application-0.0.1-SNAPSHOT.jar
```

Let's see the run times of the `matrix-slow` request:

```
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:  1601 1795 181.5   1785    2078
Waiting:     1600 1794 181.5   1785    2077
Total:       1601 1795 181.5   1786    2078
```

The profile looks like the following: (HTML)



The whole CPU is wasted in the `matrixMultiplySlow` method:

```java
public static int[][] matrixMultiplySlow(int[][] a, int[][] b, int size) {
    int[][] result = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int sum = 0;
            for (int k = 0; k < size; k++) {
                sum += a[i][k] * b[k][j];
            }
            result[i][j] = sum;
        }
    }
    return result;
}
```

If we look at the run times of the `matrix-fast` request:

```
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:   107  114   6.5    114     128
Waiting:      106  113   6.5    113     128
Total:        107  114   6.5    114     128
```

That request is **18 times faster** than the `matrix-slow` request, but if we look at the profile (HTML)

We see that the whole CPU is wasted in the method `matrixMultiplyFaster`:

```java
public static int[][] matrixMultiplyFaster(int[][] a, int[][] b, int size) {
    int[][] result = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            int current = a[i][k];
            for (int j = 0; j < size; j++) {
                result[i][j] += current * b[k][j];
            }
        }
    }
    return result;
}
```

Both methods `matrixMultiplySlow` and `matrixMultiplyFaster` have the same complexity O(n^3). So why is one faster than the other? Well, if you want to understand exactly how a CPU-intensive algorithm works, you need to understand how the CPU works, which is far away from the topic of this post. Be aware that if you want to optimize such algorithms; you will probably need at least one of the following:

- Knowledge of CPU architecture
- Top-Down performance analysis methodology
- Looking at the ASM of generated methods

Many Java programmers forget that all the execution is done on the CPU. Java needs to use ASM to run on the CPU. That's basically what the JIT compiler does: It converts your hot methods and loops into optimized ASM. At the assembly level, you can check, for example, if the JIT used vectorized instruction for your loops. So yes, sometimes you need to get dirty with such low-level stuff. For now, async-profiler gives you a hint on which methods to focus.

We will return to this example in the Cache misses section.

Allocation

The most common use cases where the allocations are tracked are:

- decreasing GC run frequency
- finding allocations outside the TLAB, which are done in the slow path
- fighting with single/tens of milliseconds latency, where even the creation of heap objects matters

First, let's understand how new objects on a heap are created so we have a better understanding of what the async-profiler shows us.

A portion of our Java heap is called an **eden**. This is a place where new objects are born. Let's assume for simplicity that eden is a continuous slice of memory. The very efficient way of allocation in such a case is called **bumping pointer**. We keep a pointer to the first address of the free space:

When we do `new Object()`, we simply calculate its size, locate the next free address and bump the pointer by `sizeof(Object)`:

But there is one major problem with that technique: We have to synchronize the object allocation if we have more than one thread that can create new objects in parallel, but this is quite costly. We solve this by giving each thread a portion of eden dedicated to only that thread. This portion is called **TLAB** - thread-local allocation buffer. With this, each thread can use **bumping pointer** at its TLAB safely and in parallel.

Introducing TLABs creates two more issues that the JVM needs to deal with:

- a thread can allocate an object, but there is not enough space in its TLAB - the JVM creates a new TLAB if there is still space in eden
- a thread can allocate a big object, so it's not optimal to use the TLAB mechanism - the JVM will use the *slow path* of the allocation that allocates the object directly in eden or in the old generation

What is important to us is that in both these cases, the JVM emits an event that a profiler can capture. That's basically how async-profiler samples allocations:

- if the allocation of an object needed a new TLAB - we see an aqua frame for that
- if the allocation was done outside the TLAB - we see a brown frame

In real-world systems, the frequency of GC can be monitored by systems like Grafana or Zabbix. Here we have a synthetic application, so let's measure the allocation size differently:

```
# little warmup
ab -n 2 -c 1 http://localhost:8081/examples/alloc/

# measuring the heap allocations of a request
jcmd first-application-0.0.1-SNAPSHOT.jar GC.run
jcmd first-application-0.0.1-SNAPSHOT.jar GC.heap_info
ab -n 10 -c 1 http://localhost:8081/examples/alloc/
jcmd first-application-0.0.1-SNAPSHOT.jar GC.heap_info
```

```
# profiling time
./profiler.sh start -e alloc -f alloc.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 1000 -c 1 http://localhost:8081/examples/alloc/
./profiler.sh stop -f alloc.jfr first-application-0.0.1-SNAPSHOT.jar
```

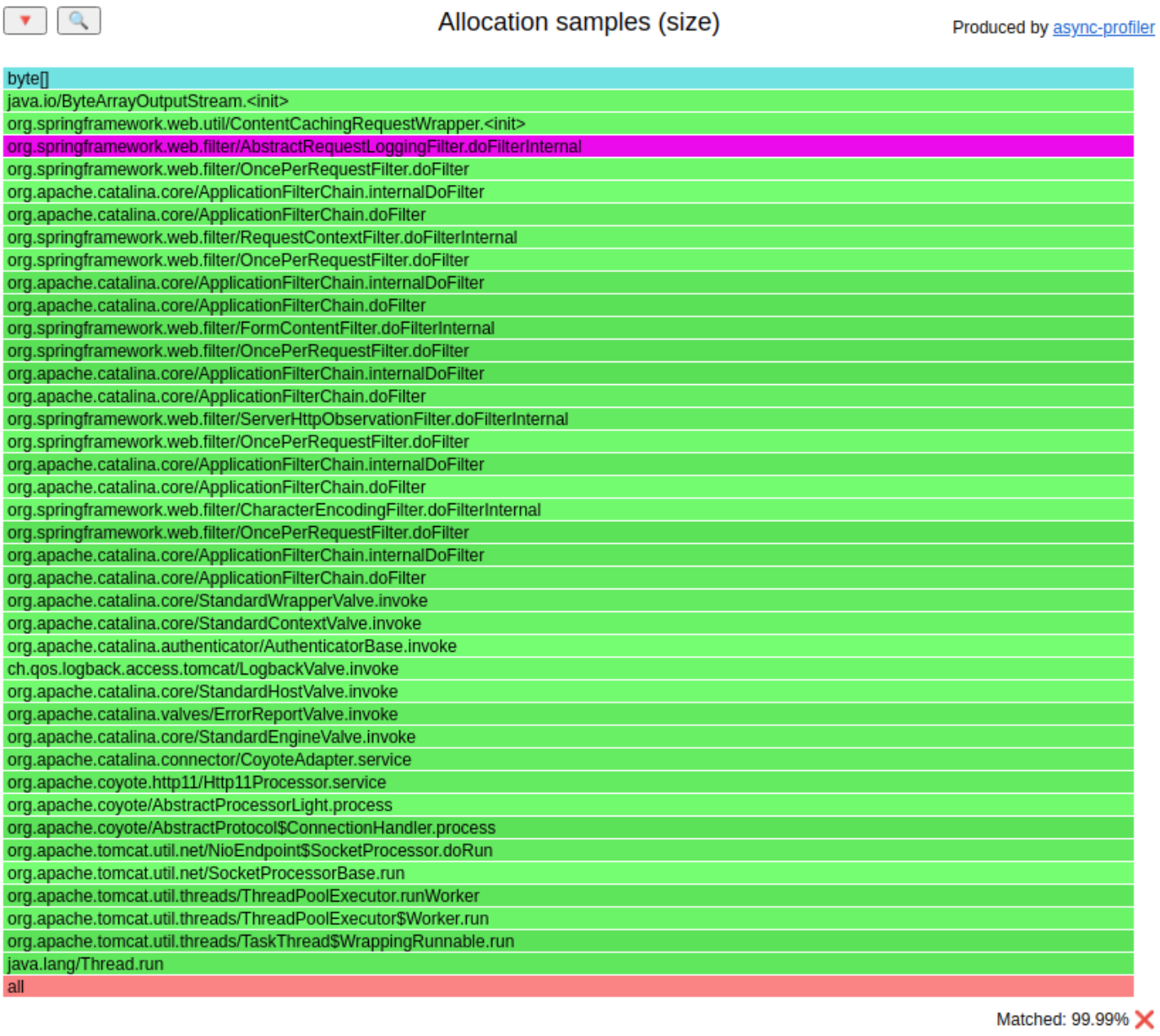
Let's look at the output of the `GC.heap_info` command:

```
 garbage-first heap   total 1048576K, used 41926K [0x00000000c0000000, 0x000000
  region size 1024K, 2 young (2048K), 0 survivors (0K)
 Metaspace       used 67317K, committed 67904K, reserved 1114112K
  class space    used 9868K, committed 10176K, reserved 1048576K

 garbage-first heap   total 1048576K, used 110018K [0x00000000c0000000, 0x000000
  region size 1024K, 8 young (8192K), 0 survivors (0K)
 Metaspace       used 67317K, committed 67904K, reserved 1114112K
  class space    used 9868K, committed 10176K, reserved 1048576K
```

We executed `alloc` requests ten times, and our heap usage has increased from **41926K** to **110018K**. So we are creating over **6MB** of objects per request on the heap. If we look at the controller source code it's hard to justify that:

```
@RestController
@RequestMapping("/examples/alloc")
@RequiredArgsConstructor
class AllocController {
    @GetMapping("/")
    String get() {
        return "OK";
    }
}
```

Let's look at the allocation flame graph: (HTML)



The class `AbstractRequestLoggingFilter` is responsible for over **99%** of recorder allocations. You can find its main creation sites using the techniques from the Methods profiling section; feel free to skip it for now. Here is the answer:

```
@Bean
CommonsRequestLoggingFilter requestLoggingFilter() {
    CommonsRequestLoggingFilter loggingFilter = new CommonsRequestLoggingFilter(
        @Override
        protected boolean shouldNotFilter(HttpServletRequest request) throws Se
            return !request.getRequestURI().contains("alloc");
        }
    };

    loggingFilter.setIncludeClientInfo(true);
    loggingFilter.setIncludeQueryString(true);
    loggingFilter.setIncludePayload(true);
    loggingFilter.setMaxPayloadLength(5 * 1024 * 1024);
```

```
    loggingFilter.setIncludeHeaders(true);
    return loggingFilter;
}
```

I have seen such code many times; this `CommonsRequestLoggingFilter` class helps you log the REST endpoint communication. The `setMaxPayloadLength()` method sets the maximum number of bytes of the payload which are logged. You can browse over the Spring source code to see that the implementation creates byte arrays of such size in the constructor. No matter how big the payload is, we always create a **5MB** array here.

The advice that I gave to users of that code was to create their own filter that would do the same job but allocate the array lazily.

Allocation - humongous objects

If you use the G1 garbage collector, JVM's default since JDK 9, your heap is divided into regions. The region sizes vary from **1 MB** to **32 MB** depending on the heap size. The goal is to have no more than **2048** regions. You can check the region size for different heap sizes with the following:

```
java -Xms1G -Xmx1G -Xlog:gc,exit*=debug -version
```

The output contains a line containing the `region size 1024K` information.

If you are trying to allocate an object larger or equal to half of the region size, you are doing a humongous allocation. Long story short: It has been, and it is, painful. It is allocated directly in the old generation but is cleared during minor GCs. I saw situations where G1 GC needed to invoke a FullGC phase because of the humongous allocation. If you do much of this, G1 will also invoke more concurrent collections, which can waste your CPU.

While running the previous example, you could spot in `first-application-0.0.1-SNAPSHOT.jar` logs:

```
...
[70,149s][debug][gc,humongous] GC(34) Reclaimed humongous region 436 (object si
[70,149s][debug][gc,humongous] GC(34) Reclaimed humongous region 442 (object si
[70,149s][debug][gc,humongous] GC(34) Reclaimed humongous region 448 (object si
[70,149s][debug][gc,humongous] GC(34) Reclaimed humongous region 454 (object si
...
```

These GC logs tell us that some humongous object of size `5242896` was reclaimed. The nice thing about JFR files is that they also keep the size of sampled allocations. Using this, we should be able to find out the stack trace that has created that object.

We don't need sophisticated JFR viewers for that. We get the `jfr` command with any JDK distribution. Let's use it:

```
$ jfr summary alloc.jfr
...
 Event Type                          Count  Size (bytes)
=============================================================
 jdk.ObjectAllocationOutsideTLAB      1013         19220
 jdk.ObjectAllocationInNewTLAB         359          6719
...
```

Let's focus on allocations outside the TLAB; it is unlikely to allocate humongous objects in the TLAB.

```
$ jfr print --events jdk.ObjectAllocationOutsideTLAB --stack-depth 10 alloc.jfr
...
jdk.ObjectAllocationOutsideTLAB {
  startTime = 2022-12-05T08:56:04.354766183Z
  objectClass = byte[] (classLoader = null)
  allocationSize = 5242896
  eventThread = "http-nio-8081-exec-9" (javaThreadId = 49)
  stackTrace = [
    java.io.ByteArrayOutputStream.<init>(int) line: 81
    org.springframework.web.util.ContentCachingRequestWrapper.<init>(HttpServle
    org.springframework.web.filter.AbstractRequestLoggingFilter.doFilterInterna
    org.springframework.web.filter.OncePerRequestFilter.doFilter(ServletRequest,
    org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ServletReq
    org.apache.catalina.core.ApplicationFilterChain.doFilter(ServletRequest, Se
    org.springframework.web.filter.RequestContextFilter.doFilterInternal(HttpSe
    org.springframework.web.filter.OncePerRequestFilter.doFilter(ServletRequest,
    org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ServletReq
    org.apache.catalina.core.ApplicationFilterChain.doFilter(ServletRequest, Se
    ...
  ]
}
...
```

We can easily match `allocationSize = 5242896` with the object size from the GC logs, so we can find and eliminate humongous allocations using that technique. You can filter the allocation JFR file for objects with a size larger or equal to half of our G1 region size. All of these allocations are humongous allocations.

Allocation - live objects

Now on to memory leaks, which form the reason for tracking live object allocations:

> A memory leak occurs when a *Garbage Collector* cannot collect Objects that are no longer needed by the Java application.

Memory leaks are one of the most common problems related to Java heaps; the other is

- **not enough space on a heap** - sometimes, a Java application may work fine with the heap it has, but there is the possibility to run a part of the application that needs more heap than specified via **-Xmx**
- **a gray area between** - these are cases when we allocate memory indefinitely, but our application needs these objects

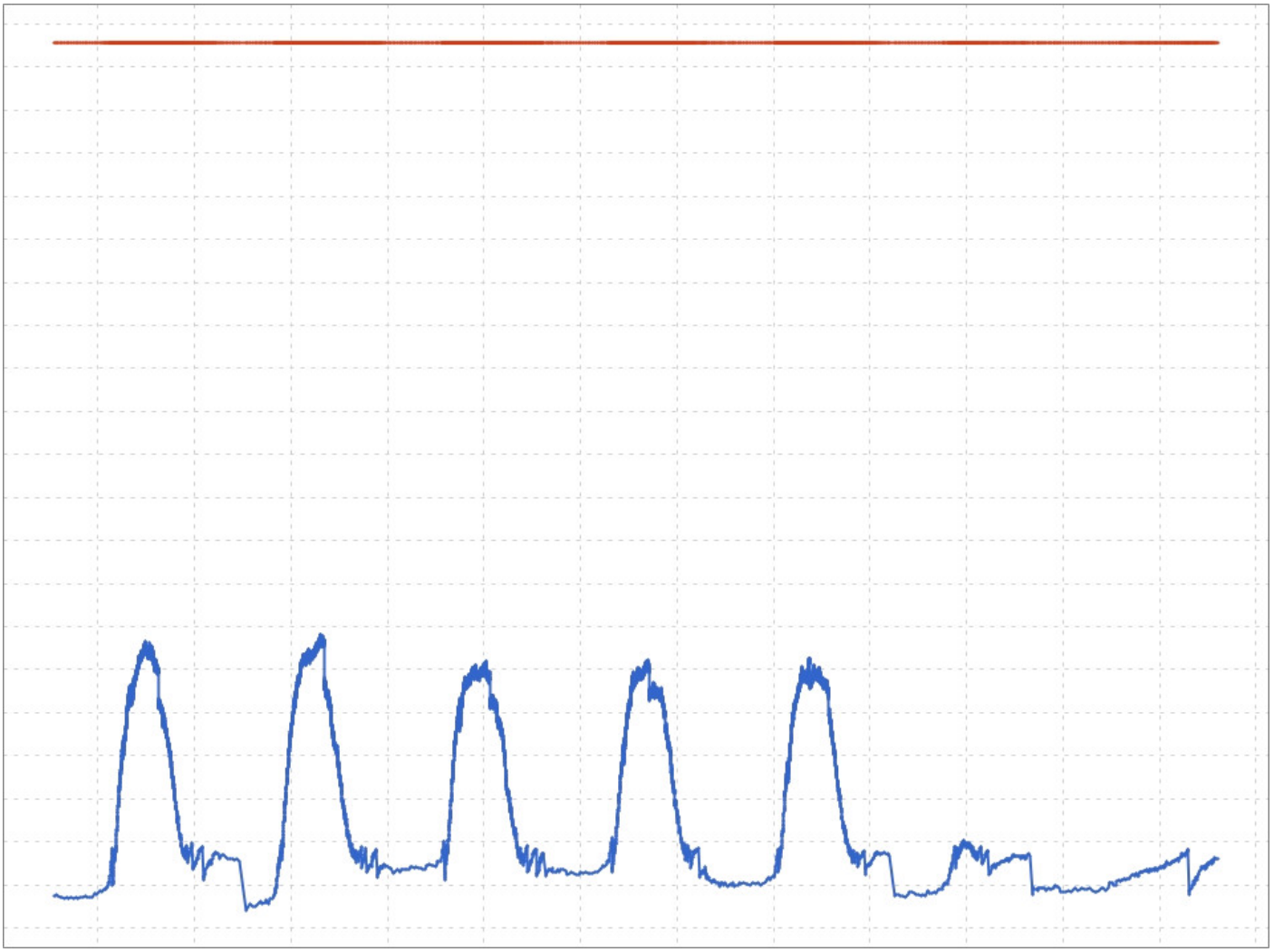
How can we detect memory leaks? The GC emits the following kind of entry at the end of each *GC cycle* into the GC logs at *info* level:

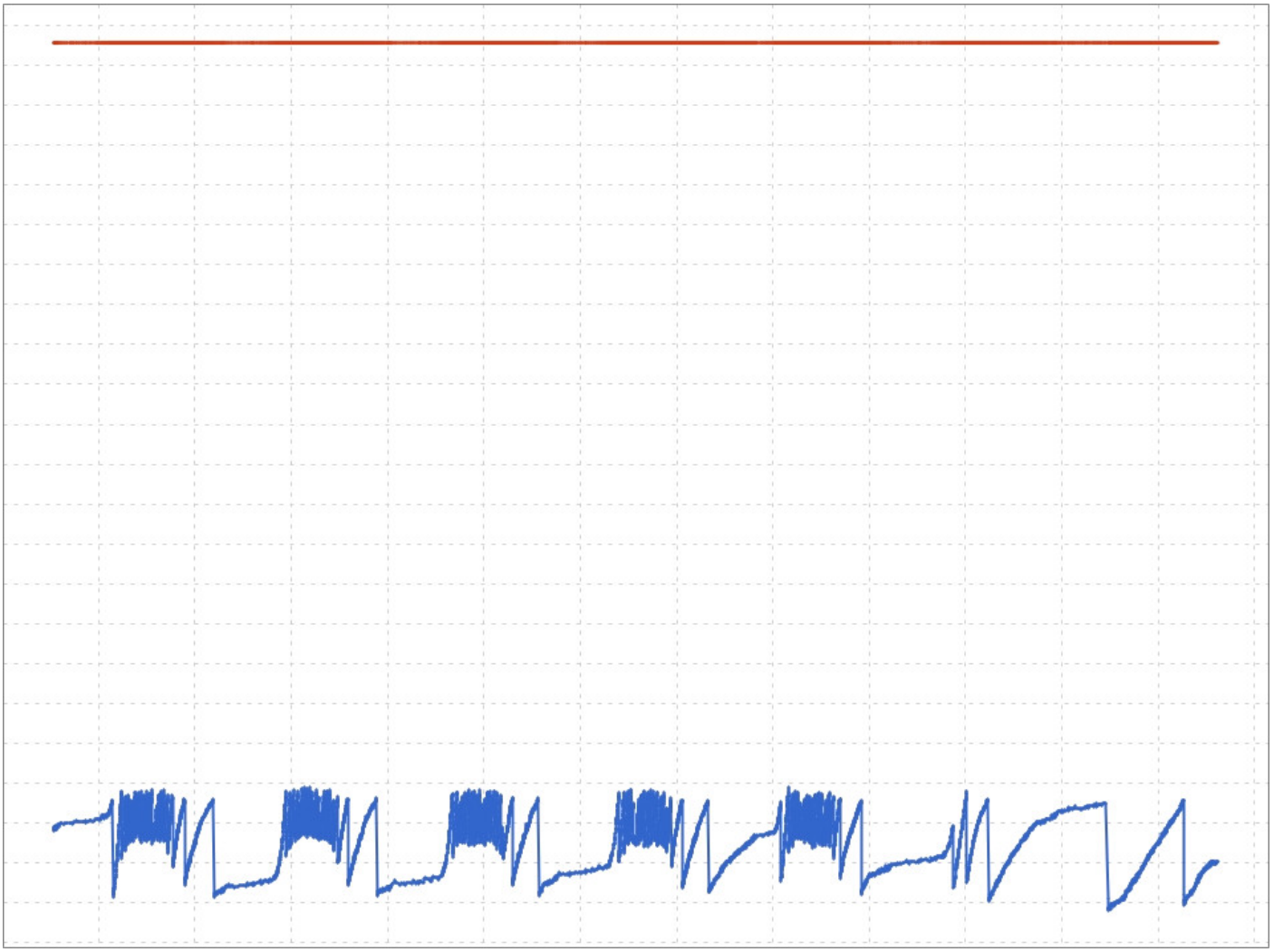
```
GC(11536) Pause Young (Normal) (G1 Evacuation Pause) 6746M->2016M(8192M) 40.514r
```

You can find **three** sizes in such an entry **A->B(C)** that are:

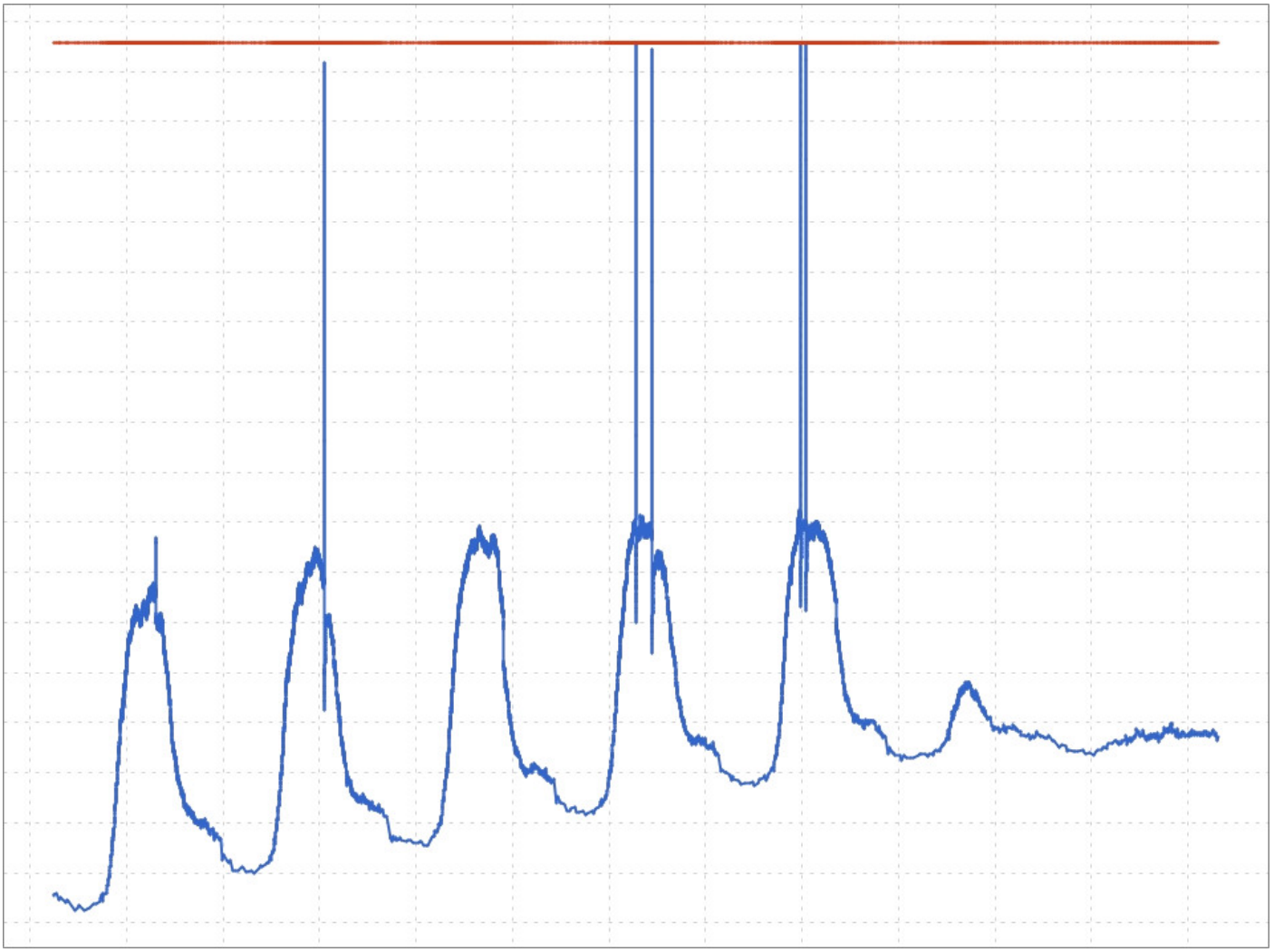
- **A** - used size of a heap before *GC cycle*
- **B** - used size of a heap after *GC cycle*
- **C** - the current size of a whole heap

If we take the **B** value from each collection and put it on a chart, we can generate the *Heap after GC* chart. We can use such a chart to then detect if we have a memory leak: If a chart looks like those (from a **7 days** period):



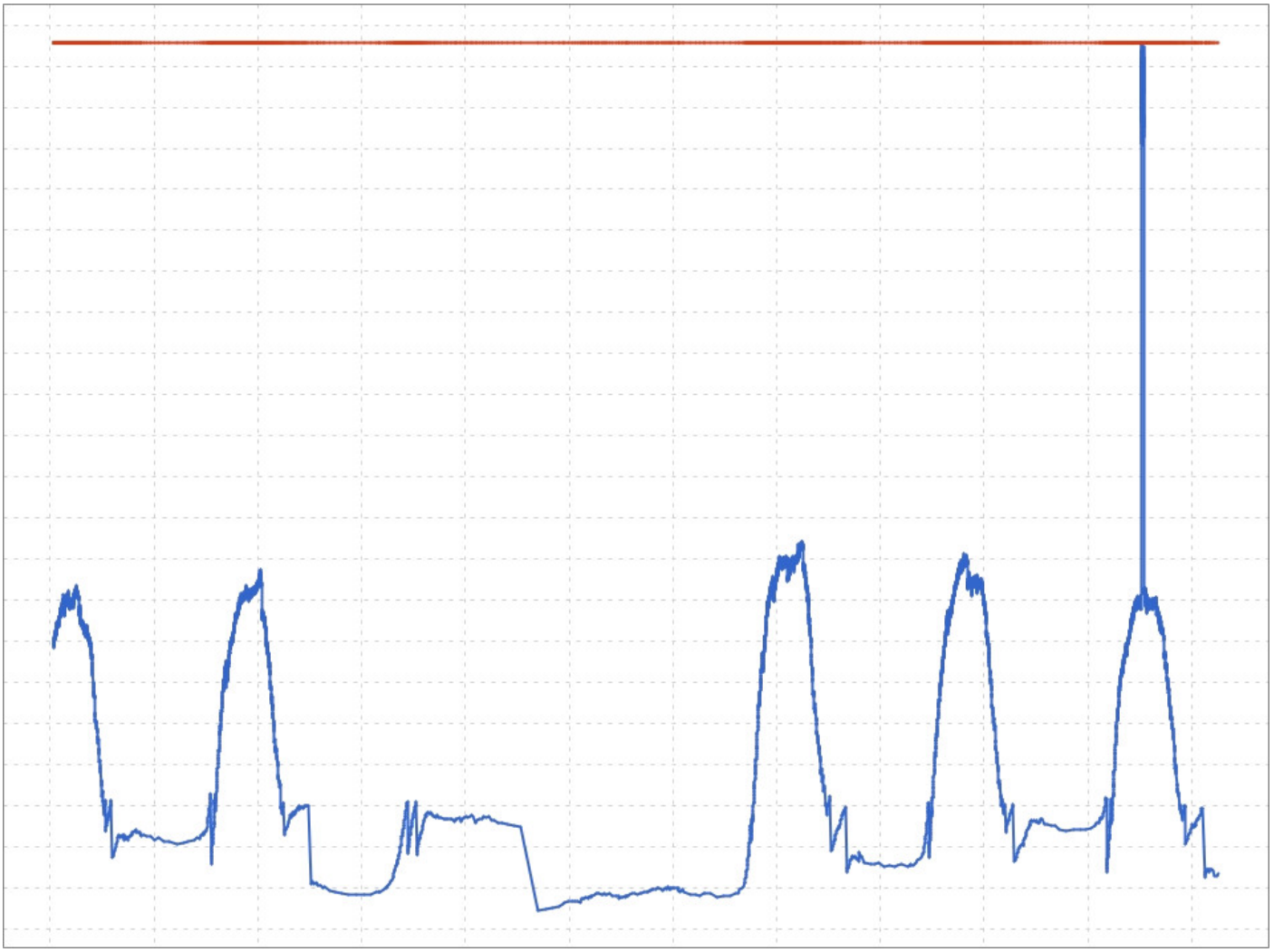


then there is **no memory leak**. The *garbage collector* can clean up the heap to the same level every day. The chart with a memory leak looks like the following one:



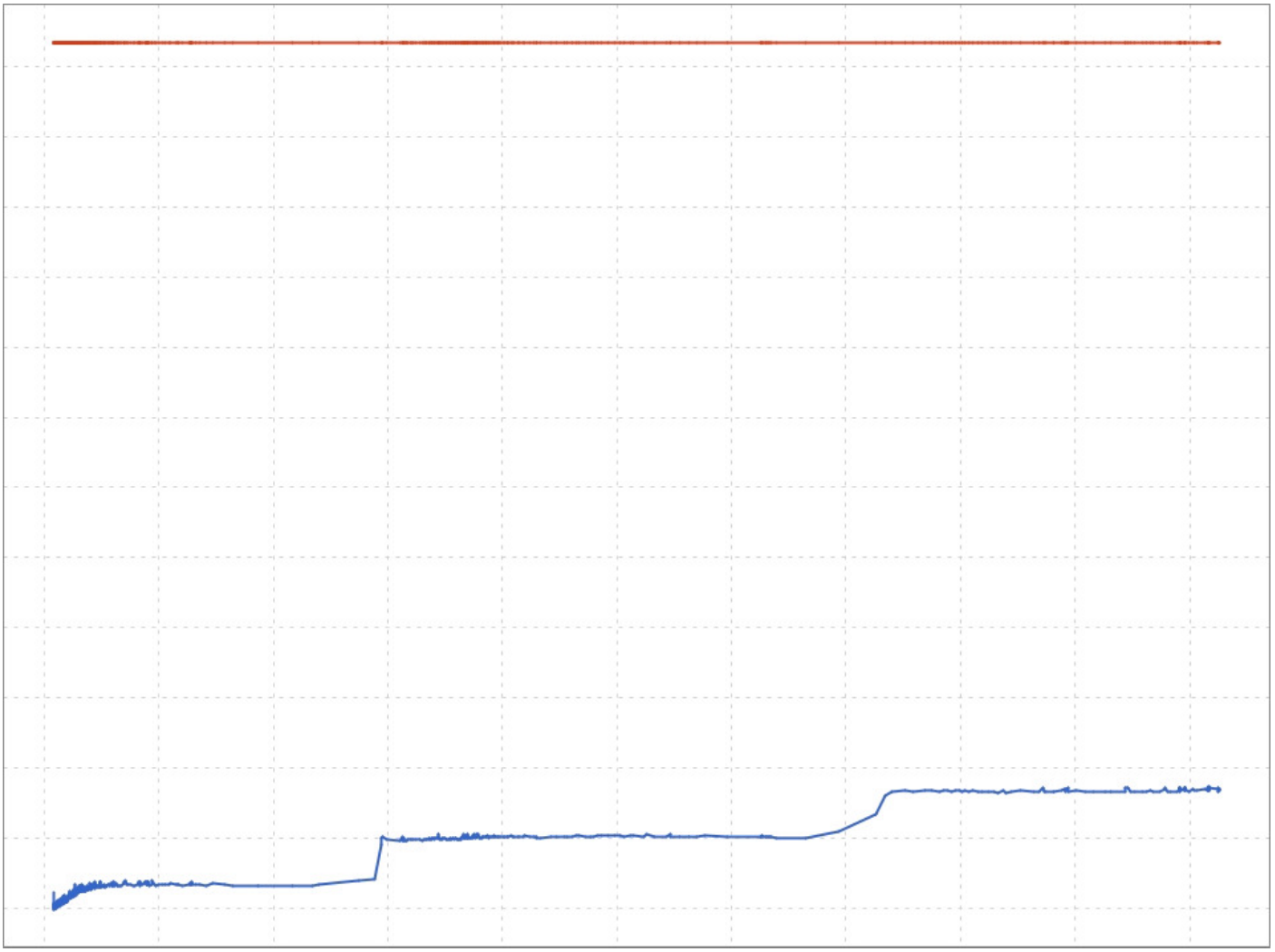
These spikes to the roof are *to-space exhausted* situations in the **G1** algorithm; those are not *OutOfMemoryErrors*. After each of those spikes, there was a **Full GC** phase that is a **failover** in that algorithm.

Here is an example of the **not enough space on a heap** problem:

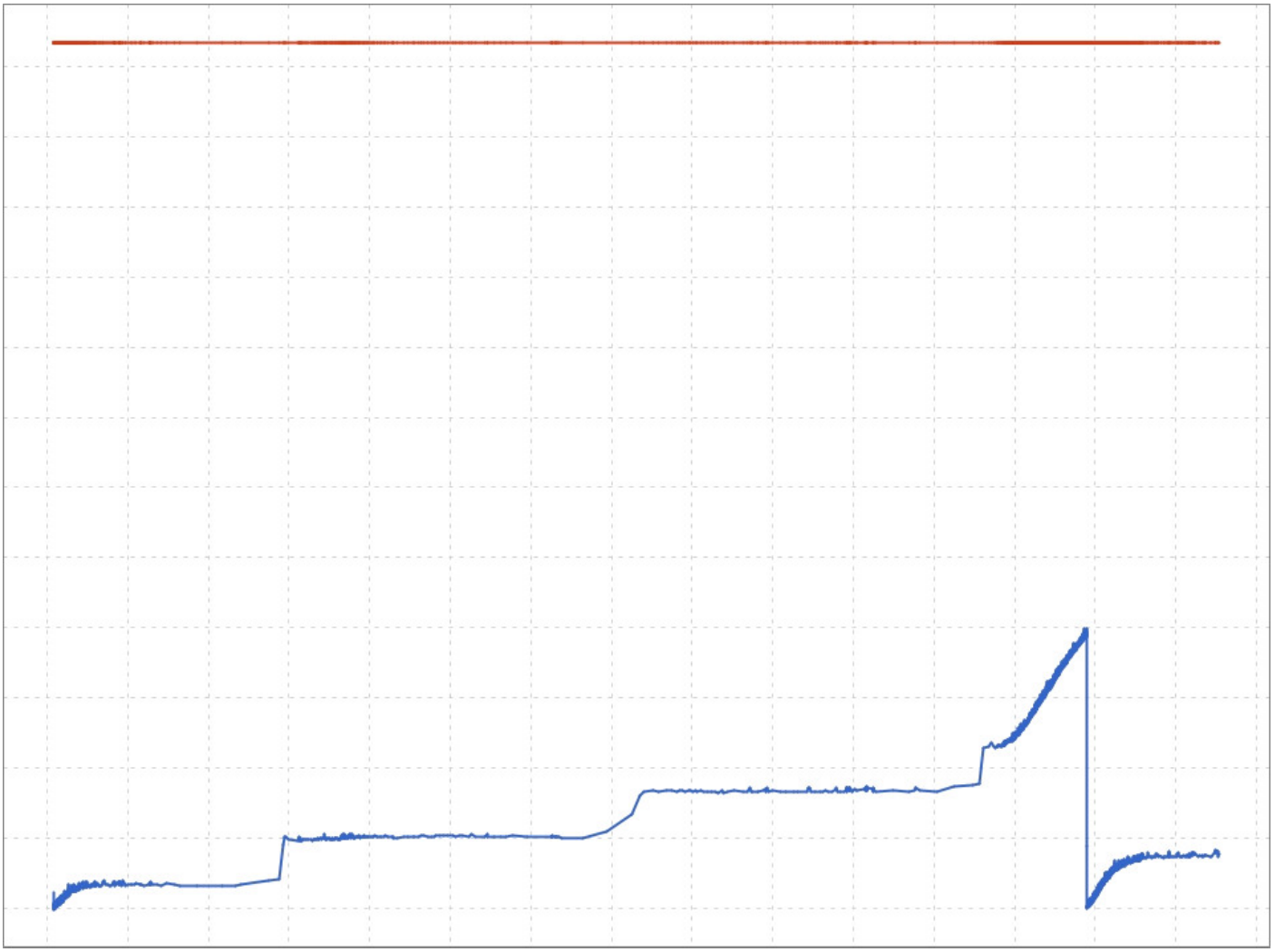


This one spike is an *OutOfMemoryError*. One service was run with arguments that needed **~16GB** on a heap to complete. Unfortunately **-Xmx** was set to **4GB**. **It is not a memory leak**.

We must be careful if our application is entirely stateless and we use GC with **young/old generations** (like G1, parallel, serial, and CMS). We must remember that objects from a **memory leak** live in the **old generation**. In stateless applications, that part of the heap can be cleared even once a week. Here is an example recording **3 days** of the stateless application:



It looks like a memory leak, the `min(heap after GC)` increasing every day, but if we look at the same chart with one additional day:



The GC cleared the heap to the previous level. This was done by an **old-generation** cleanup that didn't happen in the previous days.

The *Heap after GC* chart can be generated by probing through JMX. The JVM gives that information via mBeans:

- `java.lang:type=GarbageCollector,name=G1 Young Generation`
- `java.lang:type=GarbageCollector,name=G1 Old Generation`

Both mBeans provide attributes with the name `LastGcInfo` from which we can extract the needed information.

Most memory leaks I discovered in recent years in enterprise applications were either in frameworks/libraries or in some kind of bridge between them. Recreating such an issue in our example application would require introducing a lot of strange dependencies, so I chose to recreate one custom-made heap memory leak I discovered a few years ago.

```
# preparation
curl http://localhost:8081/examples/leak/prepare
ab -n 1000 -c 4 http://localhost:8081/examples/leak/do-leak
```

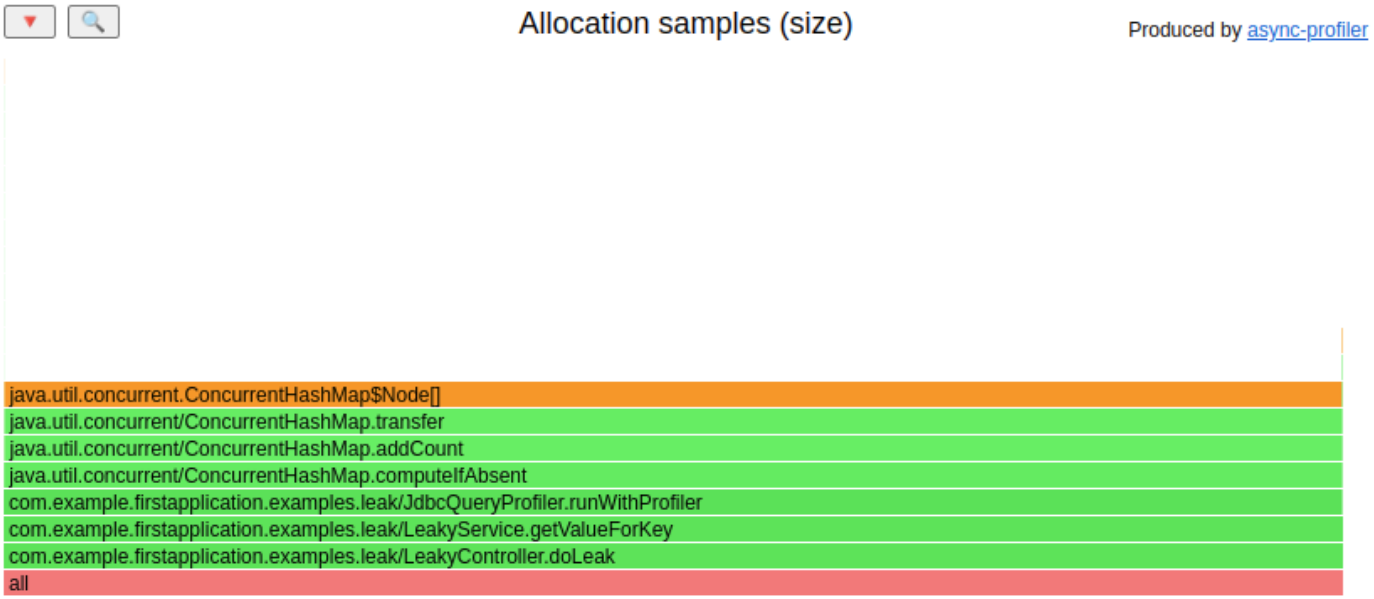
```
# profiling
jcmd first-application-0.0.1-SNAPSHOT.jar GC.run
jcmd first-application-0.0.1-SNAPSHOT.jar GC.heap_info
./profiler.sh start -e alloc --live -f live.jfr first-application-0.0.1-SNAPSHOT
ab -n 1000000 -c 4 http://localhost:8081/examples/leak/do-leak
jcmd first-application-0.0.1-SNAPSHOT.jar GC.run
jcmd first-application-0.0.1-SNAPSHOT.jar GC.heap_info
./profiler.sh stop -f live.jfr first-application-0.0.1-SNAPSHOT.jar
```

Let's look at the output of the `GC.heap_info` commands that were invoked soon after running the GC:

```
 garbage-first heap   total 1048576K, used 50144K [0x00000000c0000000, 0x000000
  region size 1024K, 1 young (1024K), 0 survivors (0K)
 Metaspace       used 68750K, committed 69376K, reserved 1114112K
  class space    used 10054K, committed 10368K, reserved 1048576K

 garbage-first heap   total 1048576K, used 237806K [0x00000000c0000000, 0x000000
  region size 1024K, 1 young (1024K), 0 survivors (0K)
 Metaspace       used 68841K, committed 69440K, reserved 1114112K
  class space    used 10063K, committed 10368K, reserved 1048576K
```

So invoking our `do-leak` request created **~183MB** of objects that GC couldn't free.

Let's look at the allocation flame graph with the `--live` option enabled: (HTML)



The largest part of the leak is created in the `JdbcQueryProfiler` class; let's look at the sources:

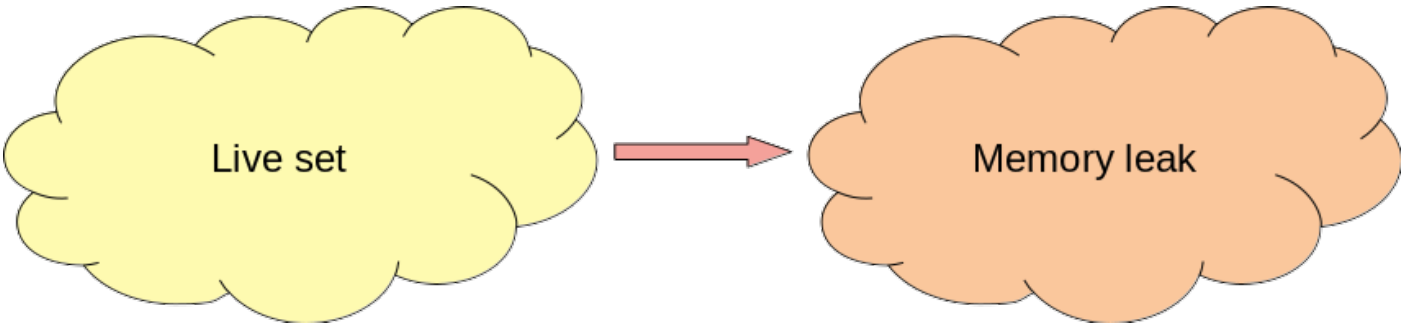
```
class JdbcQueryProfiler {
    private final Map<String, ProfilingData> profilingResults = new ConcurrentHa

    <T> T runWithProfiler(String queryStr, Supplier<T> query) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        T ret = query.get();
        stopWatch.stop();
        profilingResults.computeIfAbsent(queryStr, ProfilingData::new).nextInvoc
        return ret;
    }
    // ...
}
```

So that class calculates the execution time for each query and remembers it in some `ProfilingData` structure that is placed in `ConcurrentHashMap`. That doesn't look scary; as long as we use parametrized queries under our control, the map should have a finite size. Let's look at the usage of the `runWithProfiler()` method:

```
String getValueForKey(int key) {
    String sql = "select a_value from LEAKY_ENTITY where a_Key = " + key;

    return jdbcQueryProfiler.runWithProfiler(sql, () -> {
        try {
            return jdbcTemplate.queryForObject(sql, String.class);
        } catch (EmptyResultDataAccessException e) {
            return null;
        }
    });
}
```

So, well, so good; we are not using parameterized queries; we create new query strings for every `key`. This way mentioned `ConcurrentHashMap` is growing with every new `key` passed to the `getValueForKey` method.

If you have a heap memory leak in your application, then you have two groups of objects:



- **Live set** - a group of objects that are still needed by your application
- **Memory leak** - a group of objects that are no longer needed

Garbage collectors cannot free the second group if there is at least one strong reference from the **live set** to the **memory leak**. The biggest problems with diagnosing memory leaks are:

- the fact that the object was created **is not an issue** - it was created because it was needed for something
- the fact that the mentioned reference was created **is not an issue** - it had some purpose too
- we need to understand why that reference was not removed from our application

The last one is not trivial. All the observability/profiling tools give us a great possibility to understand why some event has happened, but with memory leaks, we need to understand why something has not yet happened. Two additional tools come to our rescue:

- **heap dump** - shows us the current state of a heap - we can find out what kind of objects are there but shouldn't be
- **profiler** - shows us where these objects were created

In this simple example, any of those tools is enough. In more complicated ones, I needed both to find the root cause of the problem. It is nice to finally have a tool that can profile memory leaks on production systems.

It is worth mentioning that the `--live` option is available only since **async-profiler 2.9**, it needs **JDK >= 11** and might still contain bugs. I didn't have a chance to test it on any production system yet.

Locks

Async-profiler has a lock mode. This mode is useful when looking into look contention in our application. Let's try to use it and understand the internals of `ConcurrentHashMap`. The `get()` method is obviously lock-free, but what about `computeIfAbsent()`? Let's profile a code that uses it:

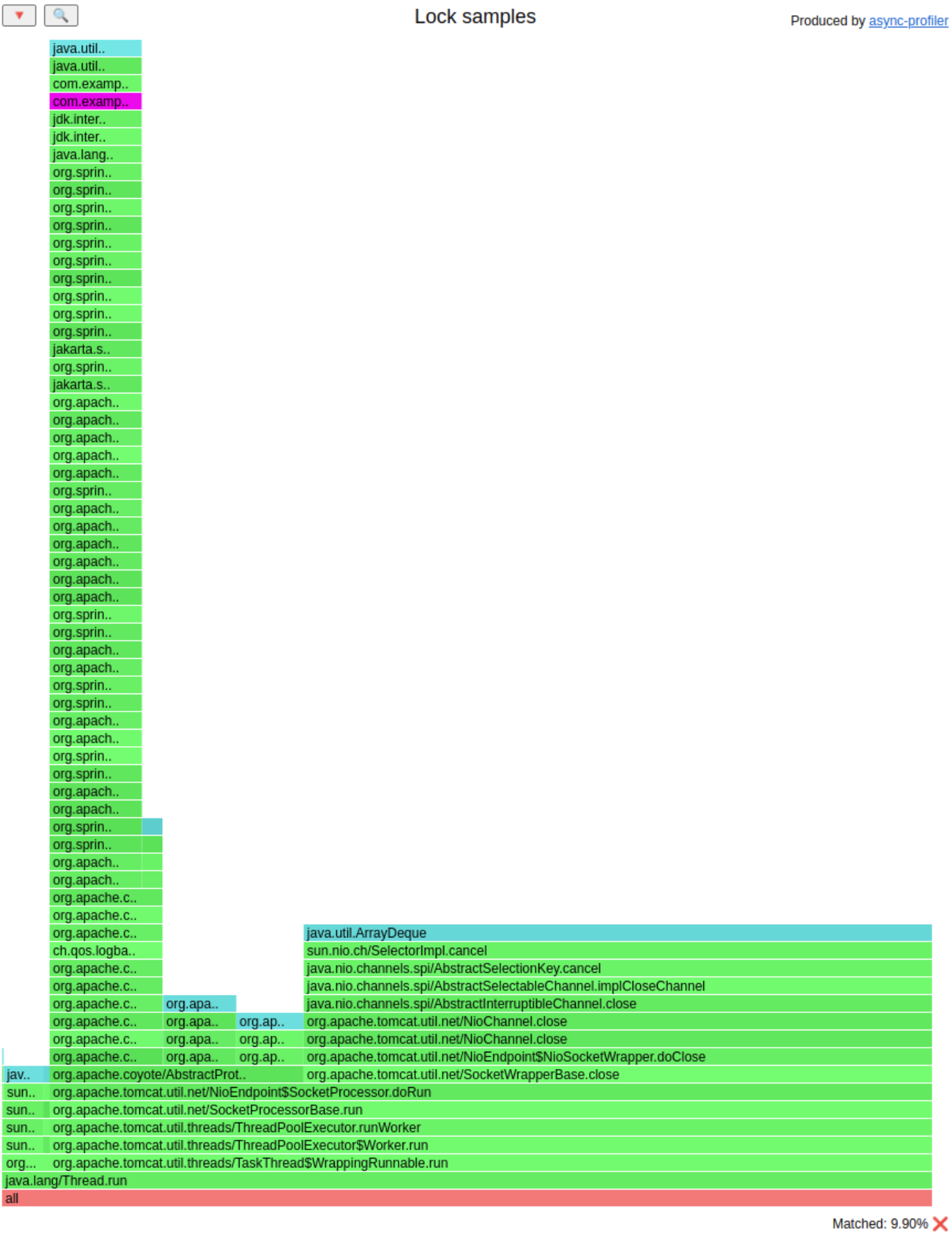
```
class LockService {
    private final Map<String, String> map = new ConcurrentHashMap<>();

    LockService() {
        String a = "AaAa";
        String b = "BBBB";
        log.info("Hashcode equals: {}", a.hashCode() == b.hashCode()); // true
        map.computeIfAbsent(a, s -> a);
        map.computeIfAbsent(b, s -> b);
    }
```

```
    void withLock(String key) {
        map.computeIfAbsent(key, s -> key);
    }

    void withoutLock(String key) {
        if (map.get(key) == null) {
            map.computeIfAbsent(key, s -> key);
        }
    }
}
```

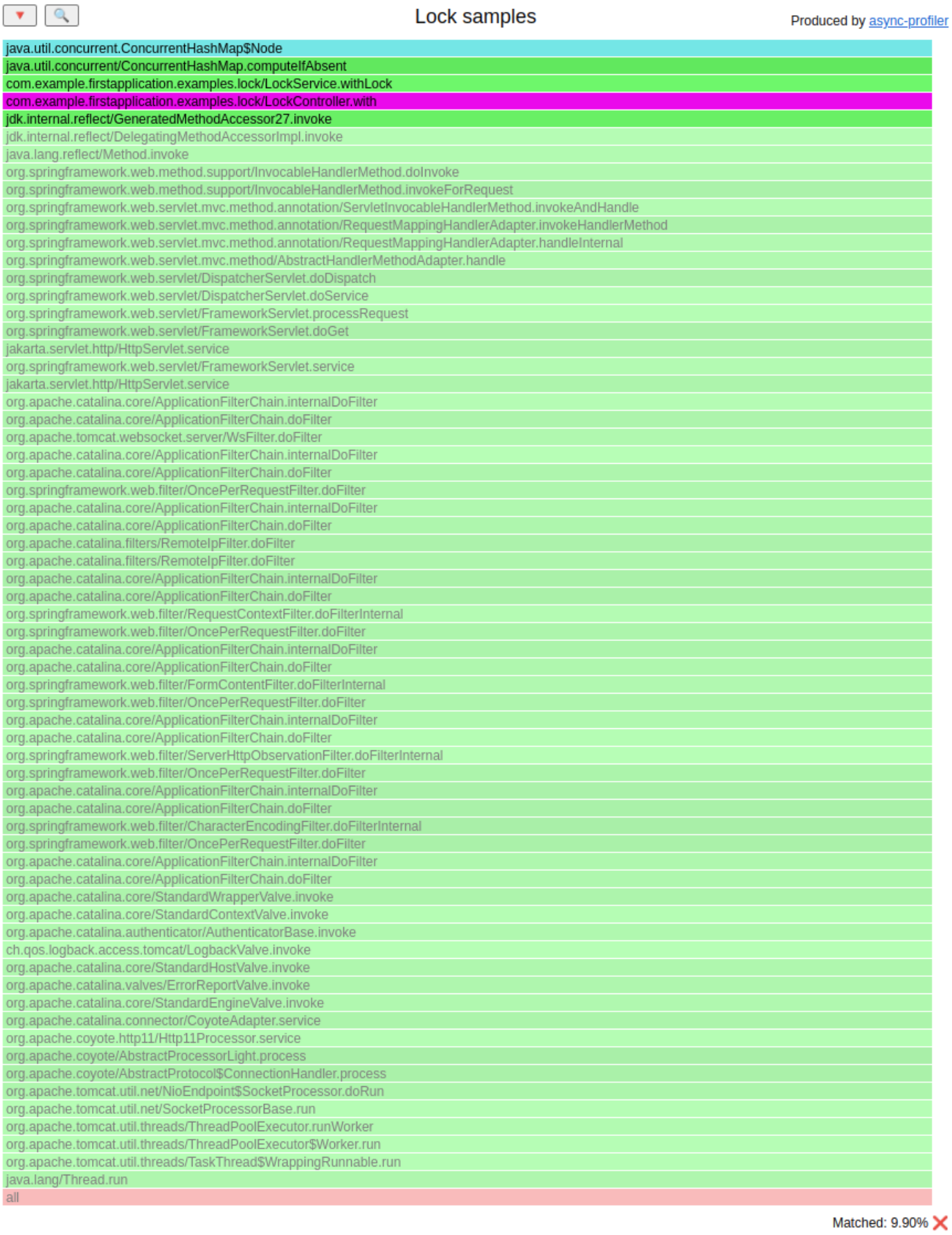
Let's use lock mode to profile that:

```
# preparation
ab -n 100 -c 1 http://localhost:8081/examples/lock/with-lock
ab -n 100 -c 1 http://localhost:8081/examples/lock/without-lock

# profiling
./profiler.sh start -e lock -f lock.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 100000 -c 100 http://localhost:8081/examples/lock/with-lock
ab -n 100000 -c 100 http://localhost:8081/examples/lock/without-lock
./profiler.sh stop -f lock.jfr first-application-0.0.1-SNAPSHOT.jar
```

The lock flame graph: (HTML)



I highlighted the `LockController` occurrence. Let's zoom it:



We see that only the `withLock()` method acquires locks. You can study the internals of the `computeIfAbsent()` method to see that it might lock on hash collisions. The easiest way to confirm this is by creating a small program that triggers hash collisions on purpose.

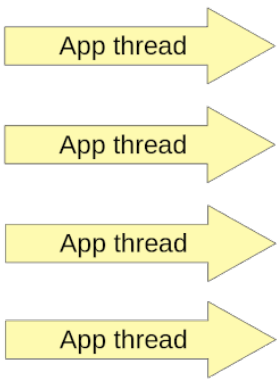
```
public static void main(String[] args) {
    Map<String, String> map = new ConcurrentHashMap<>();
    String a = "AaAa";
    String b = "BBBB";

    map.computeIfAbsent(a, s -> a);

    // it enters the synchronized section here
    map.computeIfAbsent(b, s -> b);

    // it enters the synchronized section here, and all the following
    // execution of computeIfAbsent with "BBBB" as a key.
    map.computeIfAbsent(b, s -> b);
    map.computeIfAbsent(b, s -> b);
    map.computeIfAbsent(b, s -> b);
    map.computeIfAbsent(b, s -> b);
}
```

If you observe a considerable lock contention in this method, and most of the time a key is already in the map; then you may consider the approach used in the `withoutLock()` method.

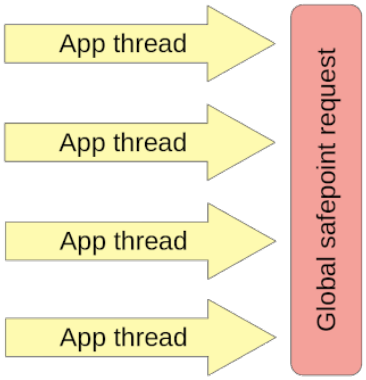
Time to safepoint

The common misconception in the Java world is that *garbage collectors* need a Stop-the-world (STW) phase to clean dead objects: But **not only GC needs it**. Other internal mechanisms require application threads to be paused. For example, the JVM needs an STW phase to *deoptimize* some compilations and to revoke *biased locks*. Let's get a closer look at how the STW phase works.

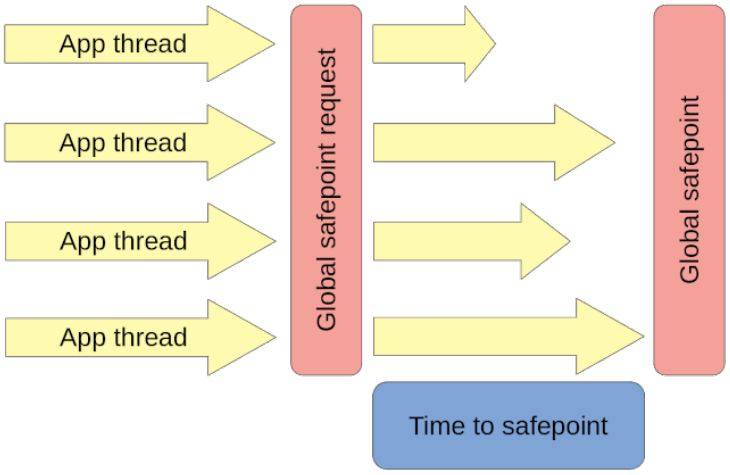
On our JVM, there are running some application threads:



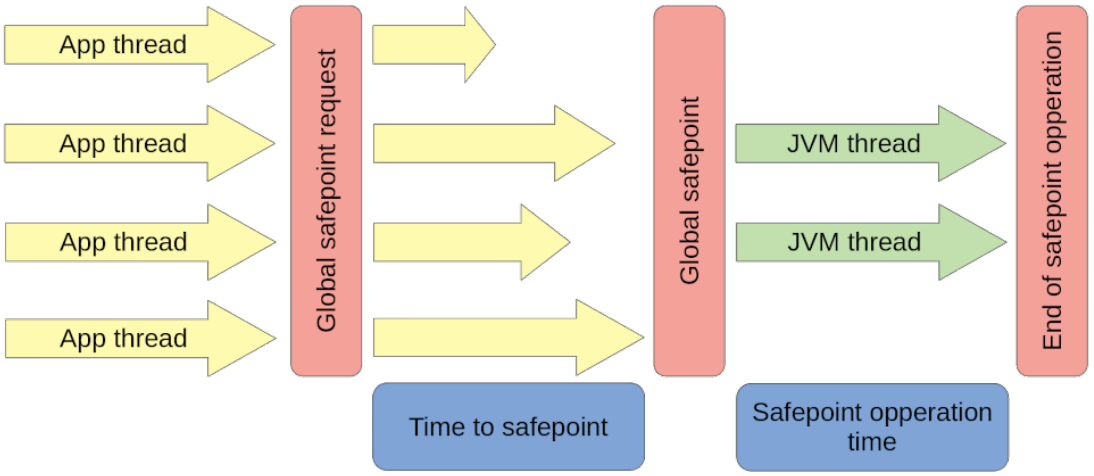
While running those threads from time to time, JVM needs to do some work in the STW phase. So it starts this phase, with a *global safepoint request*, which informs every thread to go to “sleep”:



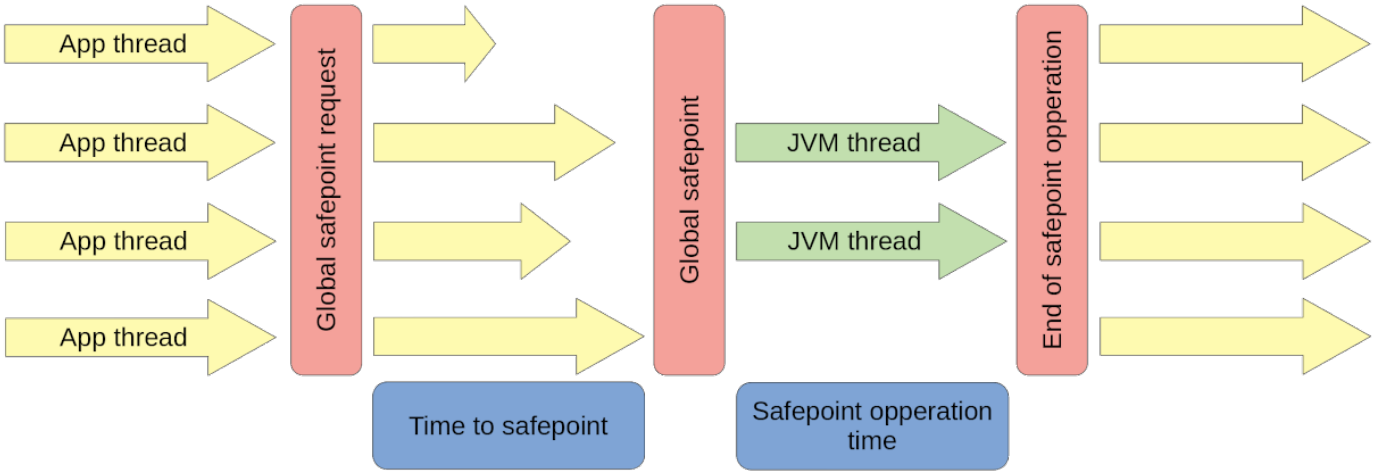
Every thread has to find out about this. Stopping at a safepoint is cooperative: Each thread checks at certain points in the code if it needs to suspend. The time in which threads will be aware of an STW phase is different for every thread. Every thread has to wait for the slowest one. The time between starting an STW phase, and the slowest thread suspension, is called *time to safepoint*:



JVM threads can do the work that needs the STW phase only after every thread is asleep. The time when all application threads sleep, is called *safepoint operation time*:



When the JVM finishes its work, application threads are wakened up:



If the application suffers from long STW phases, then most of the time, those are GC cycles, and that information can be found in the GC logs or JFR. But the situation is more tricky if the application has one thread that slows down every other from reaching the safepoint.

```
# preparation
curl http://localhost:8081/examples/tts/start
ab -n 100 -c 1 http://localhost:8081/examples/tts/execute

# profiling
./profiler.sh start --ttsp -f tts.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 100 -c 1 http://localhost:8081/examples/tts/execute
./profiler.sh stop -f tts.jfr first-application-0.0.1-SNAPSHOT.jar
```

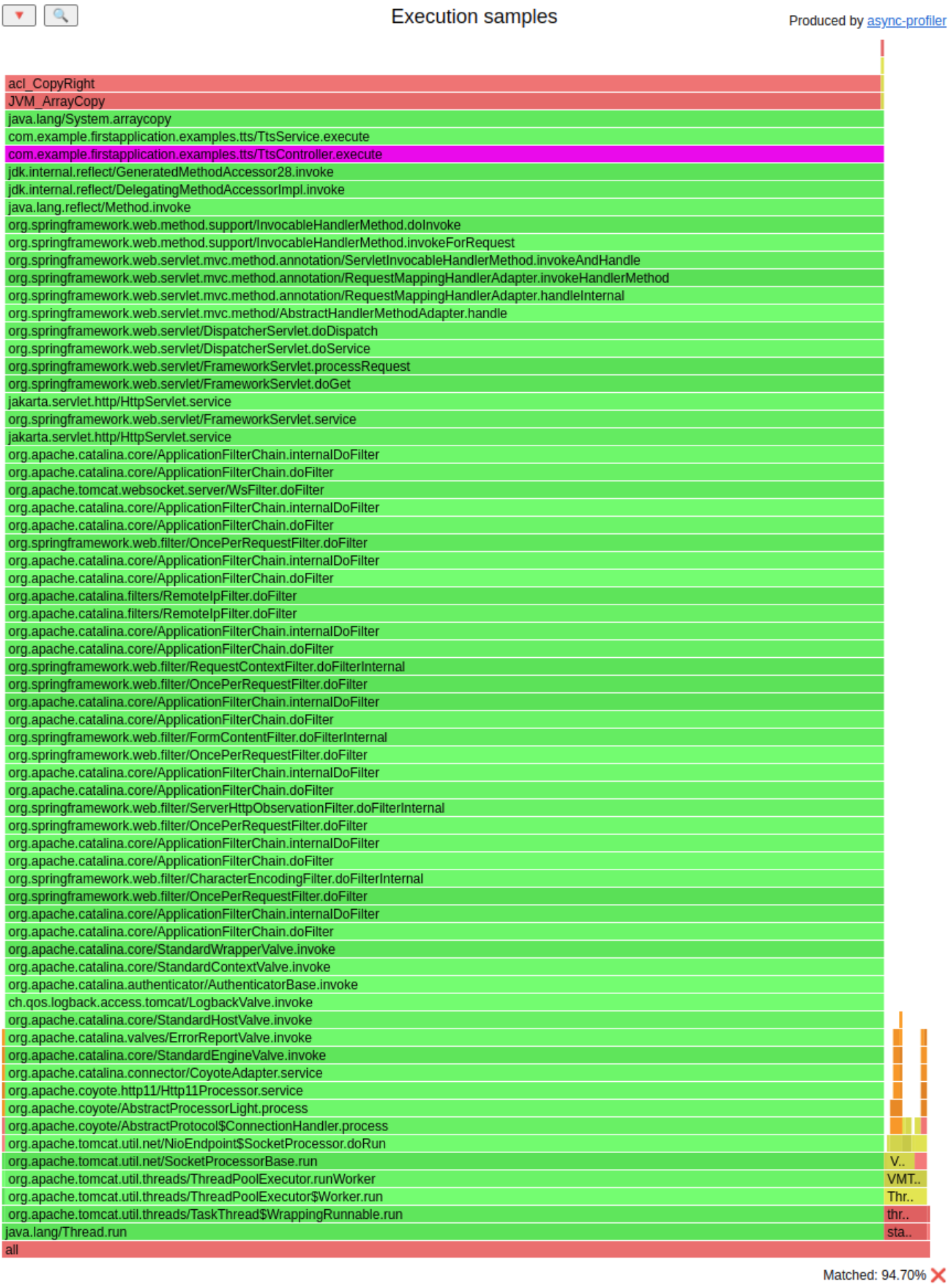
In safepoint logs (you need to run your JVM with the `-Xlog:safepoint` flag), we can see:

```
[105,372s][info ][safepoint    ] Safepoint "ThreadDump", Time since last: 156842
[105,372s][info ][safepoint    ] Safepoint "ThreadDump", Time since last: 157113
[105,373s][info ][safepoint    ] Safepoint "ThreadDump", Time since last: 157676
[105,402s][info ][safepoint    ] Safepoint "ThreadDump", Time since last: 159020
```

Reaching safepoint contains the time to safepoint. Most of the time, it is **<15 ms**, but we also see one outlier: **29 ms**. Async-profiler in `--ttsp` mode collects samples between:

- `SafepointSynchronize::begin`, and
- `RuntimeService::record_safepoint_synchronized`

During that time, our application threads are trying to reach a safepoint: (HTML)



We can see that most of the gathered samples are executing `arraycopy`, invoked from `TtsController`. The time to safepoint issues that I have approached so far are:

- **arraycopy** - as in our example
- **old JDK + loops** - since JDK 11u4 we have an *loop strip mining* optimization working correctly (after fixing JDK-8220374), before that if you had a counted loop, it could be compiled without any check for safepoint
- **swap** - when your application thread executes some work in *thread_in_vm* state (after calling some native method),
  and during that execution, it waits for some pages to be swapped in/out, which can slow down reaching the safepoint

The solution for the **arraycopy** issue is to copy larger arrays by some custom method, which might use **arraycopy** for smaller sub-arrays. It will be a bit slower, but it will not slow down the whole application when reaching a safepoint is required.

For the **swap** issue, just disable the swap.

Methods

Async-profiler can instrument a method so that we can see all the stack traces with this method on the top. To achieve that, async-profiler uses instrumentation.

Big warning: It's already pointed out in the README of the profiler that if you are not running the profiler from `agentpath`, then the first instrumentation of a Java method can result in a code cache flush. It's not the fault of the async-profiler; it's the nature of all instrumentation-based profilers combined with JVM's code. Here is a comment from the JVM sources:

// Deoptimize all compiled code that depends on the classes redefined. // // If the can_redefine_classes capability is obtained in the onload // phase then the compiler has recorded all dependencies from startup. // In that case we need only deoptimize and throw away all compiled code // that depends on the class. // // If can_redefine_classes is obtained sometime after the onload // phase then the dependency information may be incomplete. In that case // the first call to RedefineClasses causes all compiled code to be // thrown away. As can_redefine_classes has been obtained then // all future compilations will record dependencies so second and // subsequent calls to RedefineClasses need only throw away code // that depends on the class.

You can check the README PR discussion for more information on this topic. But let's focus on the usage of the mode for our purposes. In this case, we could easily do it with a plain IDE debugger, but there are situations where something is happening only in one environment, or we are tracing some issues we do not know how to reproduce.

Since Spring beans are usually created during applications startup, let's run our application that way:

```
java \
```

```
-agentpath:/path/to/libasyncProfiler.so=start,event="org.springframework.web.fi
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar
```

The `AbstractRequestLoggingFilter.<init>` is simply a constructor. We are trying to find out where such an object is created. After our application is started, we can execute such a command in the profiler directory:

```
./profiler.sh stop first-application-0.0.1-SNAPSHOT.jar
```

It will print us to one stack trace:

```
--- Execution profile ---
Total samples       : 1

--- 1 calls (100.00%), 1 sample
  [ 0] org.springframework.web.filter.AbstractRequestLoggingFilter.<init>
  [ 1] org.springframework.web.filter.CommonsRequestLoggingFilter.<init>
  [ 2] com.example.firstapplication.examples.alloc.AllocConfiguration$1.<init>
  [ 3] com.example.firstapplication.examples.alloc.AllocConfiguration.requestLog
  [ 4] com.example.firstapplication.examples.alloc.AllocConfiguration$$SpringCGI
  [ 5] com.example.firstapplication.examples.alloc.AllocConfiguration$$SpringCGI
...
  [24] org.springframework.beans.factory.support.AbstractBeanFactory.getBean
...
  [33] org.springframework.boot.web.embedded.tomcat.TomcatStarter.onStartup
...
  [58] org.springframework.boot.web.embedded.tomcat.TomcatWebServer.<init>
...
  [78] org.springframework.boot.loader.JarLauncher.main

       calls  percent  samples  top
  ----------  -------  -------  ---
           1  100.00%        1  org.springframework.web.filter.AbstractRequestLo
```

We have all the information that we need. The object is created in `AllocConfiguration` during creation of `CommonsRequestLoggingFilter` bean.

One of the other use cases where I used to use method profiling was finding memory leaks. I knew which types were leaking from the heap dump, and with method profiling, I could see where objects of these types were created. Consider this a fallback when the dedicated mode does not work.

Native functions

Not only can you trace Java code with the async-profiler but also a native one. That way of profiling doesn't cause deoptimizations.

Some native functions are worth a better look; let's cover them quickly.

Exceptions

How many exceptions should be thrown if your application works without any outage/ downtime and everything is stable? Exceptions should be thrown if something unexpected happens. Unfortunately, I saw an application that used the exception-control-flow approach more common in languages like Python. Creating a new exception is a CPU-intensive operation since, by default, it fills the stack trace. I once saw an application that consumed **~15%** of its CPU time on just creating new exceptions. You can use async-profiler in method mode with event `Java_java_lang_Throwable_fillInStackTrace` if you want to see where exceptions are created.

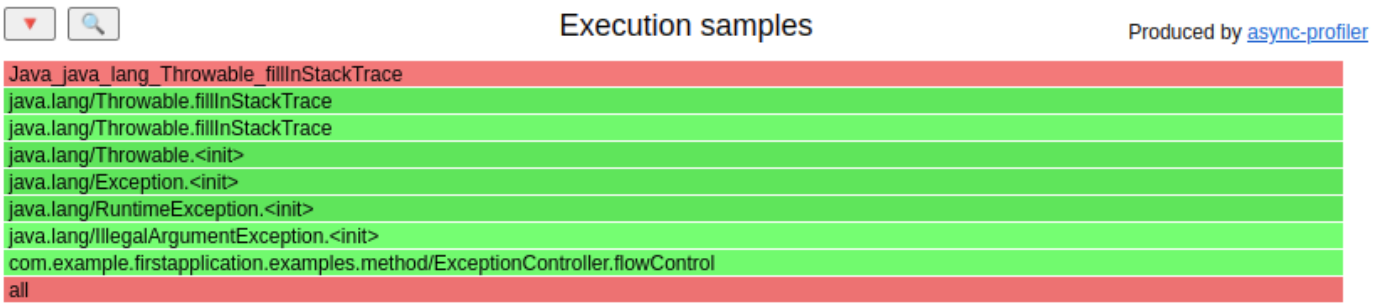
Let's start our application with profiler enabled from the start to see also how many exceptions are thrown during the startup of a Spring Boot application, just for fun:

```
java \
-agentpath:/path/to/libasyncProfiler.so=start,jfr,file=exceptions.jfr,event="Jav
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar
```

After the startup, let's run:

```
ab -n 100 -c 1 http://localhost:8081/examples/exc/
./profiler.sh stop -f exceptions.jfr first-application-0.0.1-SNAPSHOT.jar
```

The flame graph is too large to post it here as an image, sorry. Spring Boot, in that case, threw **12478** exceptions. You can play with HTML. Let's focus on our synthetic controller:



Source code of the controller:

```
@GetMapping("/")
```

```
String flowControl() {
    ThreadLocalRandom random = ThreadLocalRandom.current();
    try {
        if (!random.nextBoolean()) {
            throw new IllegalArgumentException("Random returned false");
        }
    } catch (IllegalArgumentException e) {
        return "EXC";
    }

    return "OK";
}
```

If you care about performance, don't use the exception-control-flow approach. If you really need such a code, reuse exception options like ANTLR or create an exception constructor that doesn't fill the stack trace:

```
/**
 * Constructs a new exception with the specified detail message,
 * cause, suppression enabled or disabled, and writable stack
 * trace enabled or disabled.
 *
 * @param  message the detail message.
 * @param cause the cause.  (A {@code null} value is permitted,
 * and indicates that the cause is nonexistent or unknown.)
 * @param enableSuppression whether or not suppression is enabled
 *                          or disabled
 * @param writableStackTrace whether or not the stack trace should
 *                           be writable
 * @since 1.7
 */
protected Exception(String message, Throwable cause,
                    boolean enableSuppression,
                    boolean writableStackTrace) {
    super(message, cause, enableSuppression, writableStackTrace);
}
```

Just set `writableStackTrace` to `false`. It will be rather ugly but faster.

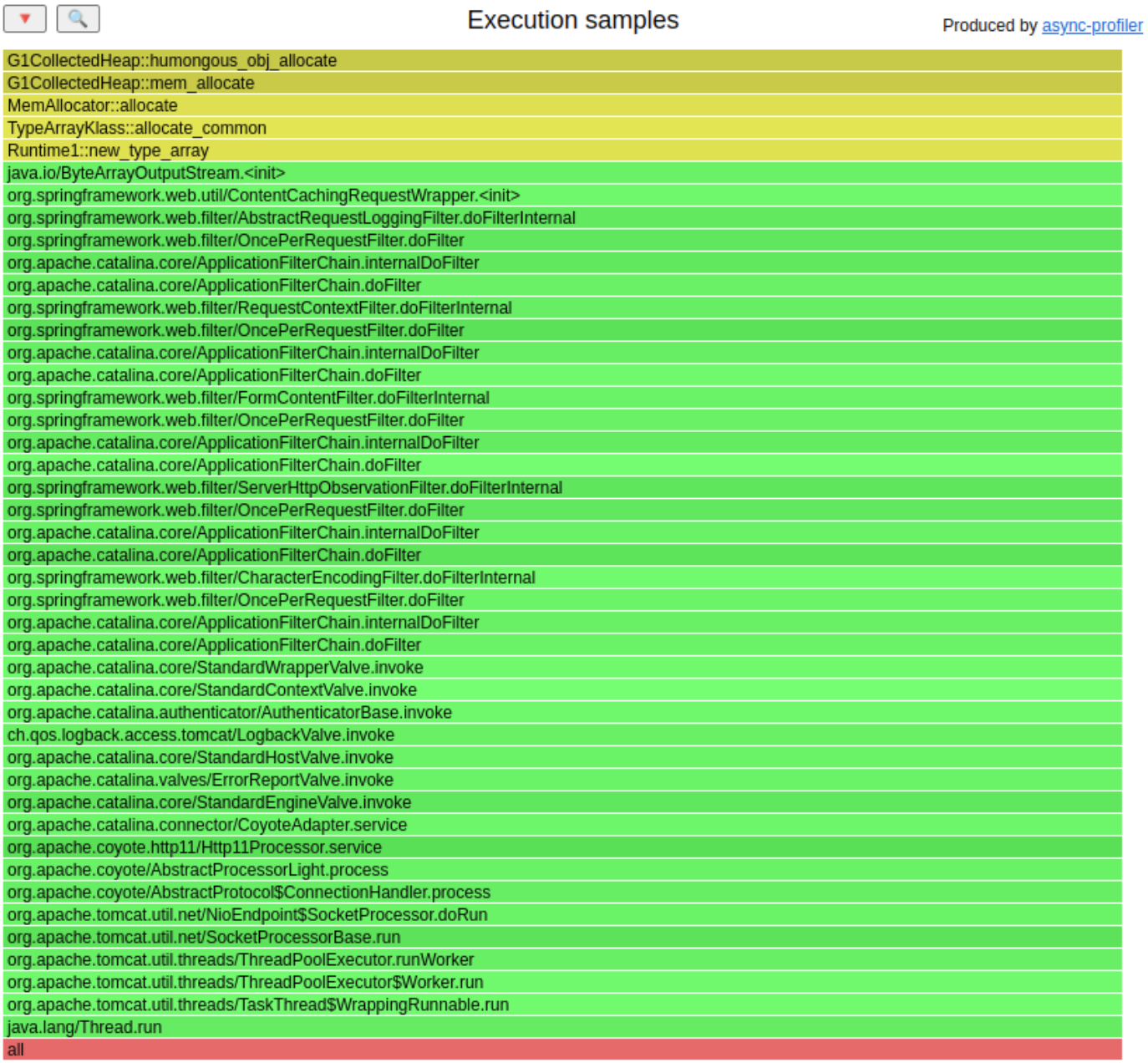
G1GC humongous allocation

We already saw how to detect humongous objects with allocation mode. Since async-profiler can also instrument JVM code, and allocations of a humongous objects are nothing else than invocations of C++ code, we can take advantage of that. If you want to check where humongous objects are allocated, you can use native functions mode with event

`G1CollectedHeap::humongous_obj_allocate`. This approach may have lower overhead but won't give you sizes of allocated objects.

```
# little warmup
ab -n 2 -c 1 http://localhost:8081/examples/alloc/

# profiling time
./profiler.sh start -e "G1CollectedHeap::humongous_obj_allocate" -f humongous.j
ab -n 1000 -c 1 http://localhost:8081/examples/alloc/
./profiler.sh stop -f humongous.jfr first-application-0.0.1-SNAPSHOT.jar
```

The flame graph is almost the same as in `alloc` mode; we can see some JVM yellow frames this time too: (HTML)

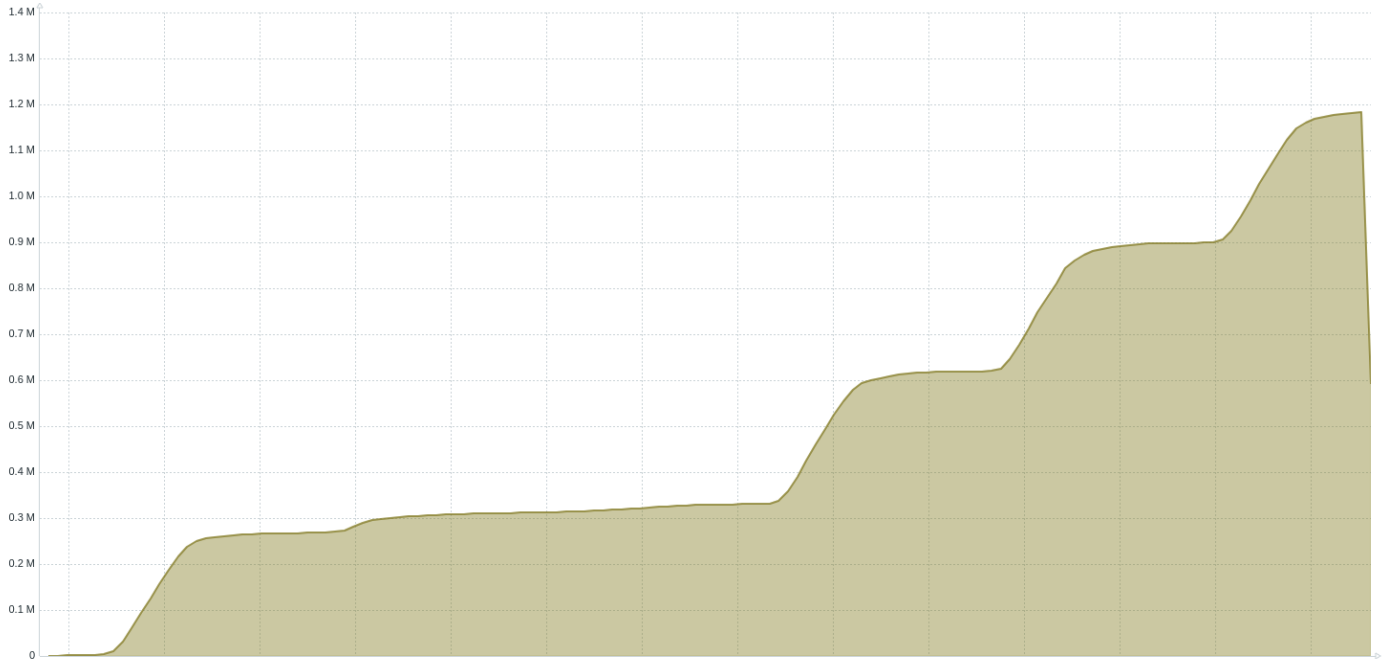


Thread start

Starting a platform thread is an expensive operation too. The number of started threads can be easily monitored with any JMX-based monitoring tool like JMC. Here is the MBean with the value of all the created threads:

```
java.lang:type=Threading
attribute=TotalStartedThreadCount
```

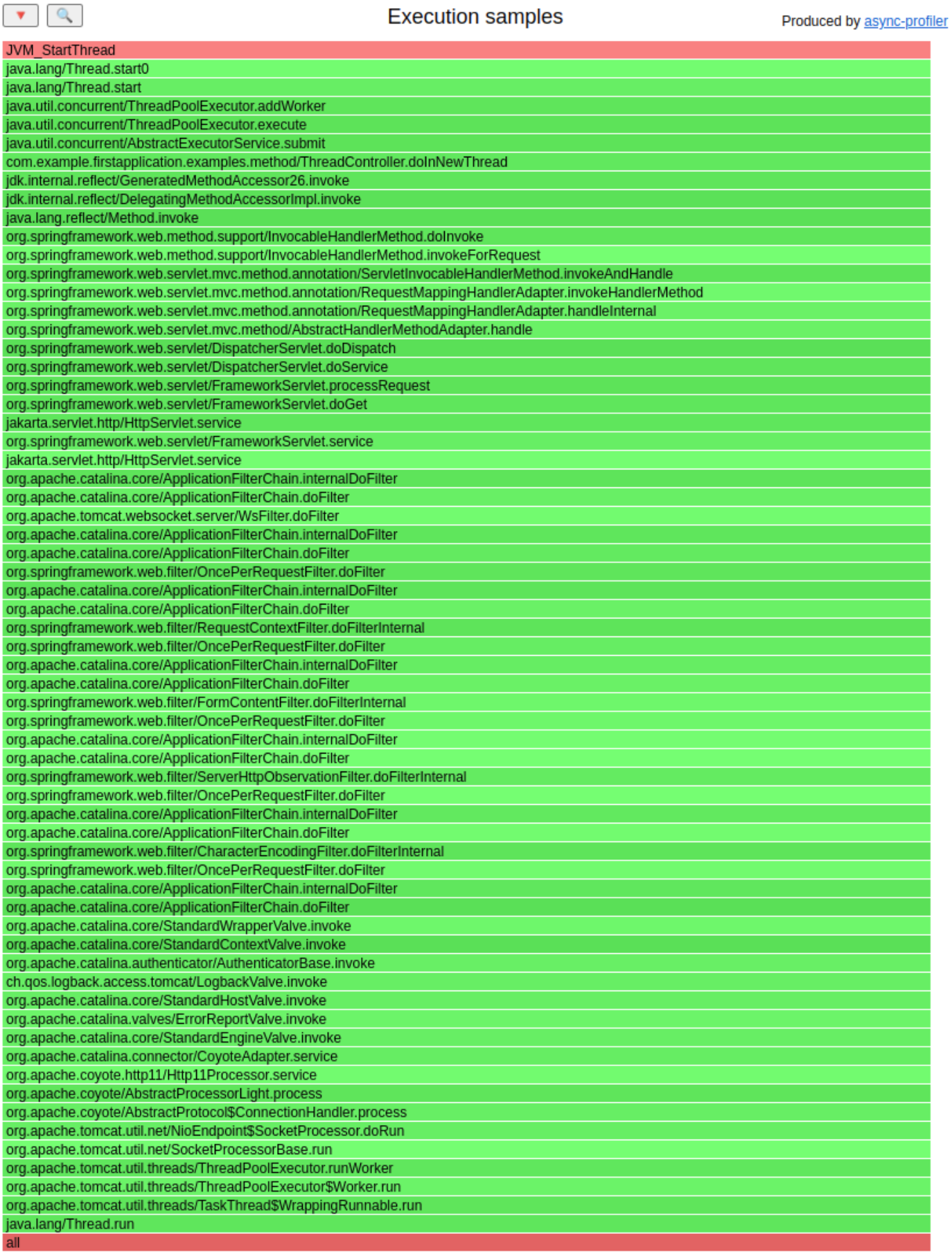
If you monitor that value and the chart like that:



Then you might want to check who is creating those short-living threads: We use async-profiler with the `JVM_StartThread` event in native functions mode for this purpose:

```
# little warmup
ab -n 100 -c 1 http://localhost:8081/examples/thread/

# profiling time
./profiler.sh start -e "JVM_StartThread" -f threads.jfr first-application-0.0.1
ab -n 1000 -c 1 http://localhost:8081/examples/thread/
./profiler.sh stop -f threads.jfr first-application-0.0.1-SNAPSHOT.jar
```

The flame graph: (HTML)



This flame graph is not really complicated. But it is only a small example. In real life, such flame graphs are larger.

The code responsible for the thread creation observed in the flame graph is the following:

```
@SneakyThrows
@GetMapping("/")
String doInNewThread() {
    ExecutorService threadPool = Executors.newFixedThreadPool(1);
    return threadPool.submit(() -> {
        return "OK";
    }).get();
}
```

And yes, I saw such a pattern in a real production application. The intention was to have a fixed thread pool and delegate tasks to it, but by mistake, someone created that pool for every request.

Class loading

Similar to creating short-living threads, I saw an application that created plenty of short-living class definitions. I know there are some use cases for such behavior, But it has been an accident in this case. You can monitor the number of loaded classes with JMX:

```
java.lang:type=ClassLoading
attribute=TotalLoadedClassCount
```

There are some internals of the JVM, like reflection or debugging, which are used in a variety of frameworks, that can generate new class definitions during runtime: So increasing that number (even after warmup) doesn't mean that we have a problem already. But if `TotalLoadedClassCount` is much higher than `LoadedClassCount`, then we might have a problem. You can find the creator of those classes with method mode and the event: `Java_java_lang_ClassLoader_defineClass1`.

To be honest, I saw such an issue only once and cannot reproduce it now. Making a synthetic example for this use-case seems wrong, so I will just keep you with the knowledge that there is such a possibility, especially if you purposefully create classes dynamically.

Perf events

Async-profiler can also help you with low-level diagnosis where you want to correlate perf events with Java code:

- `context-switches` - to find out which parts of your Java code do context switching
- `cache-misses` - which part of your code can stall due to cache misses - this information

is harder to analyze if you have many context switches
- `LLC-load-misses` - which part of your code needs a lot of data directly from RAM which is not cached
- ...

I want to describe the three in more detail in the following.

Cache misses

Let's return to the example with matrix multiplication from the CPU - a bit harder section. I usually start by looking at basic CPU performance counters to see what our CPU is doing in the slow and the fast multiplication. This is the textbook example of cache misses and their importance for performance. I like to start with the JMH test to profile the specific code properly.

I've prepared such a benchmark in the `jmh-suite` module. Let's run it with the perf profiler:

```
java -jar jmh-suite/target/benchmarks.jar -prof perf
```

The fast algorithm (I've cut the output to the most interesting metrics):

```
         20 544,42 msec task-clock                       #    1,008 CPUs utilize
    49 510 157 799      L1-dcache-loads                  #    2,410 G/sec
     9 300 675 824      L1-dcache-load-misses            #   18,79% of all L1-d
     1 635 877 333      LLC-loads                        #   79,626 M/sec
        27 833 149      LLC-load-misses                  #    1,70% of all LL-ca
```

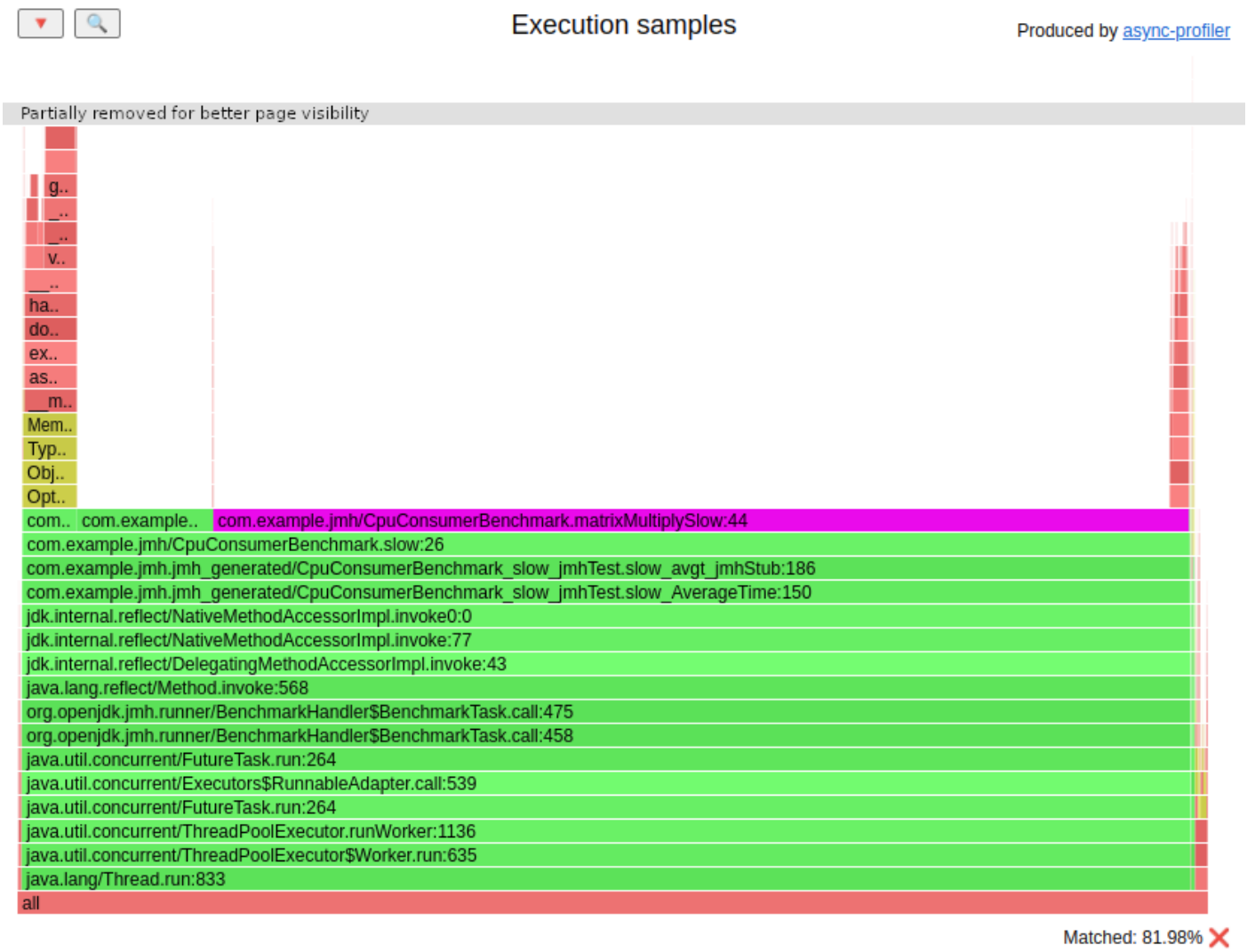
The slow one:

```
         22 291,74 msec task-clock                       #    1,008 CPUs utilize
    71 632 332 204      L1-dcache-loads                  #    3,213 G/sec
    29 718 804 848      L1-dcache-load-misses            #   41,49% of all L1-d
     6 909 042 687      LLC-loads                        #  309,937 M/sec
        10 043 405      LLC-load-misses                  #    0,15% of all LL-ca
```

The slower algorithm has **three times** more L1 data cache misses and over **four times** more last-level cache loads. We can now use async-profiler in three different modes:

```
java -jar jmh-suite/target/benchmarks.jar -prof async:libPath=/path/to/libasyncP
java -jar jmh-suite/target/benchmarks.jar -prof async:libPath==/path/to/libasync
java -jar jmh-suite/target/benchmarks.jar -prof async:libPath==/path/to/libasync
```

All three flame graphs are very similar; let's take a look at `cache-misses` : (HTML)



I added the line numbers this time, so we could see exactly where the problem was. **~82%** of cache misses occurred in the same line:

```
sum += a[i][k] * b[k][j];
```

This line is nested inside three loops. The order of loops is `i, j, k`. If we unroll the last loop four times we would get the following:

```
sum += a[i][k + 0] * b[k + 0][j];
sum += a[i][k + 1] * b[k + 1][j];
sum += a[i][k + 2] * b[k + 2][j];
sum += a[i][k + 3] * b[k + 3][j];
```

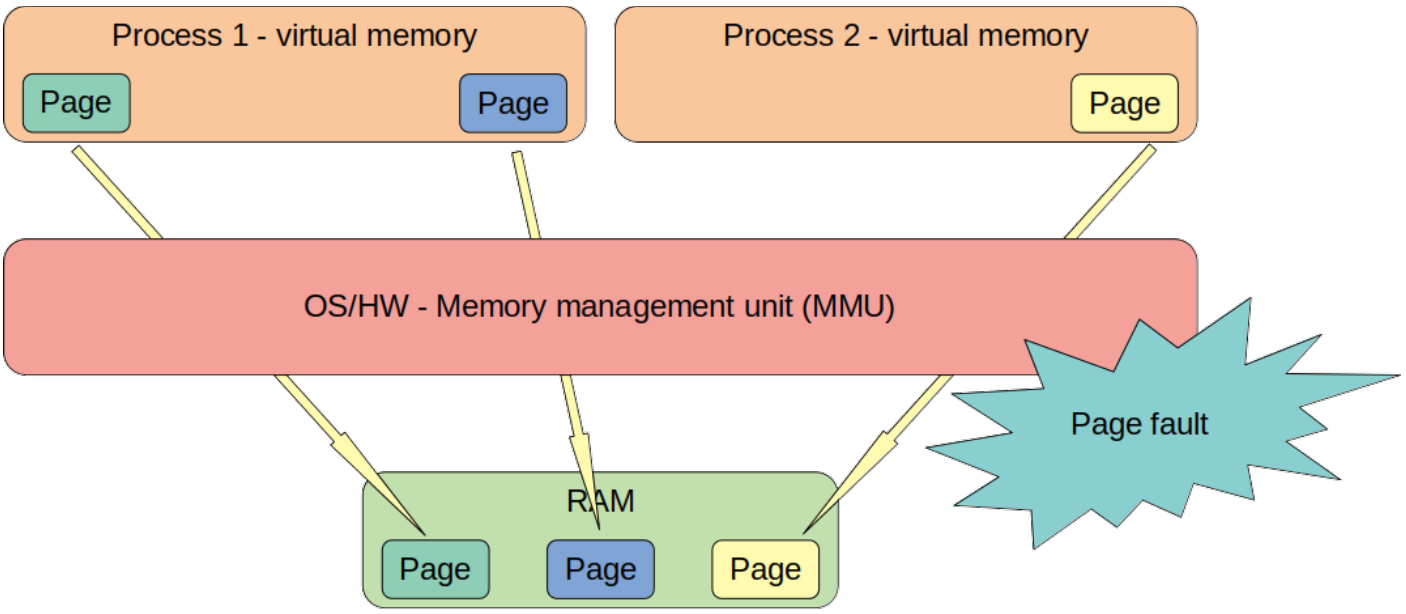
Let's look at this code from a memory layout perspective. The array `a[i]` is a contiguous part of memory. That's how Java allocates arrays. Elements `a[i][k + 0]` ... `a[i][k + 3]` are very close to each other and are loaded sequentially. The CPU loads small blocks of memory from RAM into the cache if the block is not already there. Accessing data in a sequential pattern is, therefore, far less expensive.

The access pattern to table `b` is completely different. The `b[k + x]` is just a pointer to a table. It is somewhere on the heap, but where exactly? Well, we cannot control that. Element `b[k + 0][j]` may be in a completely different place than `b[k + 1][j]`. That's unfortunate for the CPU. This is why the speed difference between both matrix multiplications is not as large as expected.

Memory access patterns are the key here. The `matrixMultiplyFaster` algorithm accesses the table `a` mostly sequentially, which is why it's faster.

I don't want to go into detail about what is happening in the CPU with these algorithms. This post aims to teach the usage of async-profiler, not CPU architecture and algorithm engineering. If you want to go deeper with that knowledge, a very good book for a start is Denis Bakhvalov - Performance Analysis and Tuning on Modern CPUs. It's not about Java, but I cannot recommend any Java-centric book related to CPU architecture, as it's still a relatively niche topic. I know that two very good performance engineers are writing one now. When it is published, I will paste a link here.

Page faults



Every process running on Linux contains its own virtual memory. If a process needs more memory, it invokes functions like `malloc` or `mmap`. The OS guarantees the returned memory to be readable/writable by the current process. But this does not mean that any block of physical RAM has been reserved for the process.

The OS is smart enough to decide whether that fault should be converted into a SEGFAULT or should trigger the kernel to map RAM to the process's virtual memory because it was previously promised to the process.

Java is a process from an OS perspective, nothing less, nothing more. Knowing that we can

trace `page fault` events to detect why our application consumes more RAM. It may be a native memory leak or some framework/library/JVM bug.

But this is not perfect for tracing leaks since it shows every request for additional RAM, including ones that may be freed in the future. I know that Andrei Pangin is working on a native memory leak detector that will trace allocations that haven't been freed, but for now, that feature is not in the latest release.

As an example, let's run our application with and without `-XX:+AlwaysPreTouch`, forcing the JVM to access all allocated memory after requesting it from the OS. This allows us to find where Java needs more RAM after startup. We will use the heap memory leak that we used before:

```
java -Xmx1G -Xms1G -XX:+AlwaysPreTouch \
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar
```

In the other console, let's do the following:

```
ab -n 10000 -c 4 http://localhost:8081/examples/leak/do-leak

./profiler.sh start -e page-faults -f page-faults-apt-on.jfr first-application-(
ab -n 1000000 -c 4 http://localhost:8081/examples/leak/do-leak
./profiler.sh stop -f page-faults-apt-on.jfr first-application-0.0.1-SNAPSHOT.ja
```

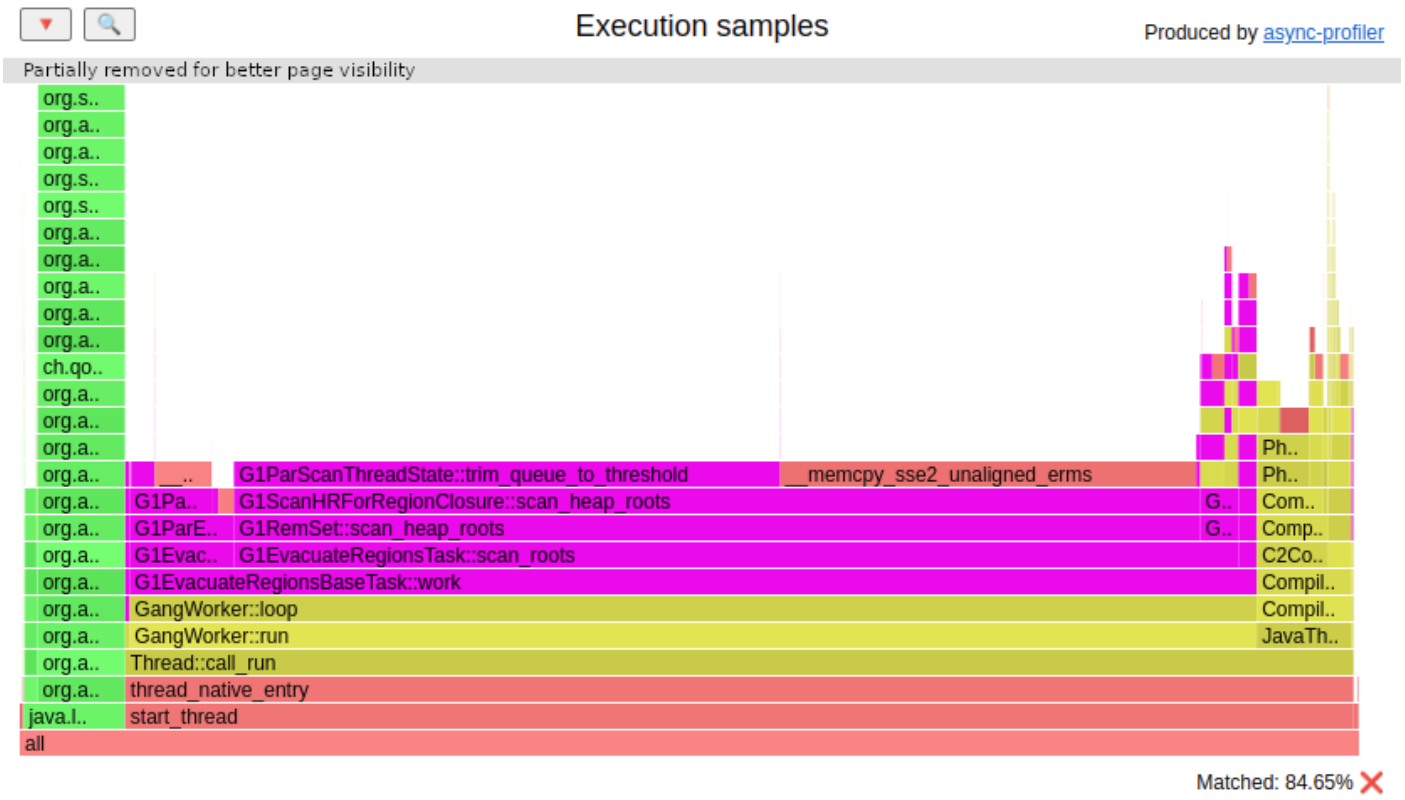
Now let's do the same without `-XX:+AlwaysPreTouch`:

```
java -Xmx1G -Xms1G \
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar
```

In the other console, let's execute the following:

```
ab -n 10000 -c 4 http://localhost:8081/examples/leak/do-leak

./profiler.sh start -e page-faults -f page-faults-apt-off.jfr first-application
ab -n 1000000 -c 4 http://localhost:8081/examples/leak/do-leak
./profiler.sh stop -f page-faults-apt-off.jfr first-application-0.0.1-SNAPSHOT.j
```

Flame graph without `-XX:+AlwaysPreTouch`: (HTML)



Most of the need for additional RAM is acquired in GC threads, but there are some page faults in our Java code (big green flame). These page faults can hurt your performance and make your latency less predictable.

Flame graph with `-XX:+AlwaysPreTouch` : (HTML)

Almost all the frames needing additional RAM now belong to compiler threads. This is due the

code heap growing with the compilation of new methods.

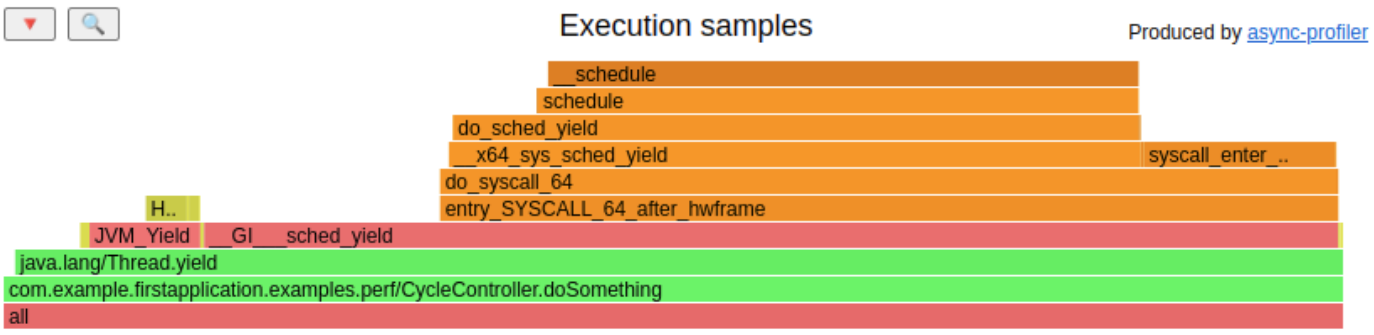
I was able to isolate and recreate the memory leak that I described in JDK-8240723 with that mode.

Cycles

If you need better visibility of what your kernel is doing, then you may consider choosing the `cycles` event instead of `cpu`. This may be useful for low-latency applications or while chasing bugs in the kernel (those also exist). Let's see the difference:

```
# warmup
curl -v http://localhost:8081/examples/cycles/

# profiling
./profiler.sh start -e cpu -f cycles-cpu.jfr first-application-0.0.1-SNAPSHOT.ja
curl -v http://localhost:8081/examples/cycles/
./profiler.sh stop -f cycles-cpu.jfr first-application-0.0.1-SNAPSHOT.jar
./profiler.sh start -e cycles -f cycles-cycles.jfr first-application-0.0.1-SNAPS
curl -v http://localhost:8081/examples/cycles/
./profiler.sh stop -f cycles-cycles.jfr first-application-0.0.1-SNAPSHOT.jar
```

The flame graph for `cpu` profiling: (HTML)



Corresponding profile for `cycles` event: (HTML)

As we can see, the `cycles` profile is more detailed.

Filtering single request

Why aggregated results are not enough

So far, we have been looking at the profile of a whole application. But what if the app works well but there are some slower requests from time to time, and we want to know why? In such an application, where one request is handled by one thread, we can extract the profile of a single request. The JFR file contains all the information needed; we just need to filter them out. To do it, we need to have a log that will tell us which thread was responsible for the execution of the request at the time of the execution. Tomcat, embedded into Spring Boot, has access logs with all that information.

I configured our example application with access logs in the format:

```
[%t] [%r] [%s] [%D ms] [%I]
```

Here is a short explanation of that magic:

- `%t` - time of finishing handling of the request
- `%r` - requested URI
- `%s` - response status code

- `%D` - duration time in milliseconds
- `%I` - thread that handled request

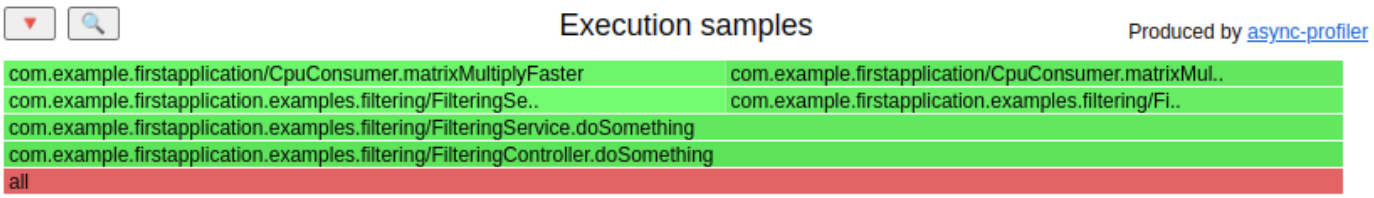
Let's see it in action.

```
# warmup
ab -n 20 -c 4 http://localhost:8081/examples/filtering/

# profiling of the first request
./profiler.sh start -e wall -f filtering.jfr first-application-0.0.1-SNAPSHOT.ja
ab -n 50 -c 4 http://localhost:8081/examples/filtering/
./profiler.sh stop -f filtering.jfr first-application-0.0.1-SNAPSHOT.jar
```

In the access logs, we can spot faster and slower requests:

```
[05/Dec/2022:18:41:57 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [1044 ms]
[05/Dec/2022:18:41:57 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [2779 ms]
[05/Dec/2022:18:41:58 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [1048 ms]
[05/Dec/2022:18:41:58 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [1052 ms]
[05/Dec/2022:18:41:59 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [2829 ms]
[05/Dec/2022:18:41:59 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [1058 ms]
```

Let's load the JFR into my viewer and look at the flame graph of the whole application: (HTML)



It's hard to guess why some requests are slower than others. We can see two different methods executed at the top of the flame graph:

- `matrixMultiplySlow()`
- `matrixMultiplyFaster()`

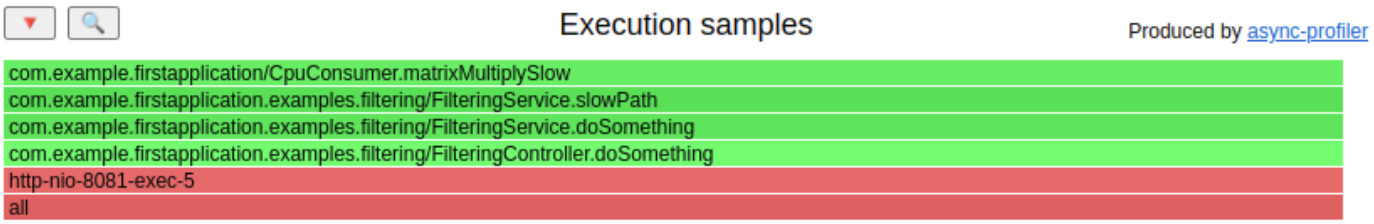
We cannot conclude from that which one is responsible for worse latency. Let's add filters to that graph to understand the latency of the second request from pasted access log:

```
[05/Dec/2022:18:41:57 +0100] [GET /examples/filtering/ HTTP/1.0] [200] [2779 ms]
```

- *Access log filter*:
  - *End date* - `05/Dec/2022:18:41:57 +0100`

    - *End date format* - let's keep the default one
    - *Duration* - `2779`
    - *Locale language* - `EN`
- *Thread filter* - `http-nio-8081-exec-5`

Now the flame graph is obvious: (HTML)



We can check this for a few more requests and figure out the following:

- In every slow request, we executed `matrixMultiplySlow()`
- In every fast request, we executed `matrixMultiplyFaster()`

This technique is great for dealing with the tail of the latency: We can focus our work on the longest operations. That may lead us to some nice fixes.

Real-life example - DNS

The point of the previous example was to show you why aggregated results can be useless for tracing a single request. Now I want to show you a widespread issue I have diagnosed a few times.

Spring Boot has a commonly used addition called Actuator. One of the features of the Actuator is the health check endpoint. Under URI `/actuator/health`, you can get JSON with information about the health of your application. That endpoint is sometimes used as a load balancer probe. Let's consider a multi-node cluster of our example application with a load balancer in front of the cluster, which:

- probes the actuator if the application is alive, expecting `"status" : "UP"` in the response JSON
- timeouts the probe after **1 second**

Now, I will do one hack in my local configuration to make this example work. It will be explained at the end of this example.

Let's find out what our IP is:

```
$ ifconfig
```

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 172.31.36.53  netmask 255.255.240.0  broadcast 172.31.47.255
```

Let's probe an actuator by this IP, not a `localhost` (without the hack described later, you cannot get the same results):

```
./profiler.sh start -e wall -f actuator.jfr first-application-0.0.1-SNAPSHOT.jar
ab -n 1000 http://172.31.36.53:8081/actuator/health # check your IP
./profiler.sh stop -f actuator.jfr first-application-0.0.1-SNAPSHOT.jar
```

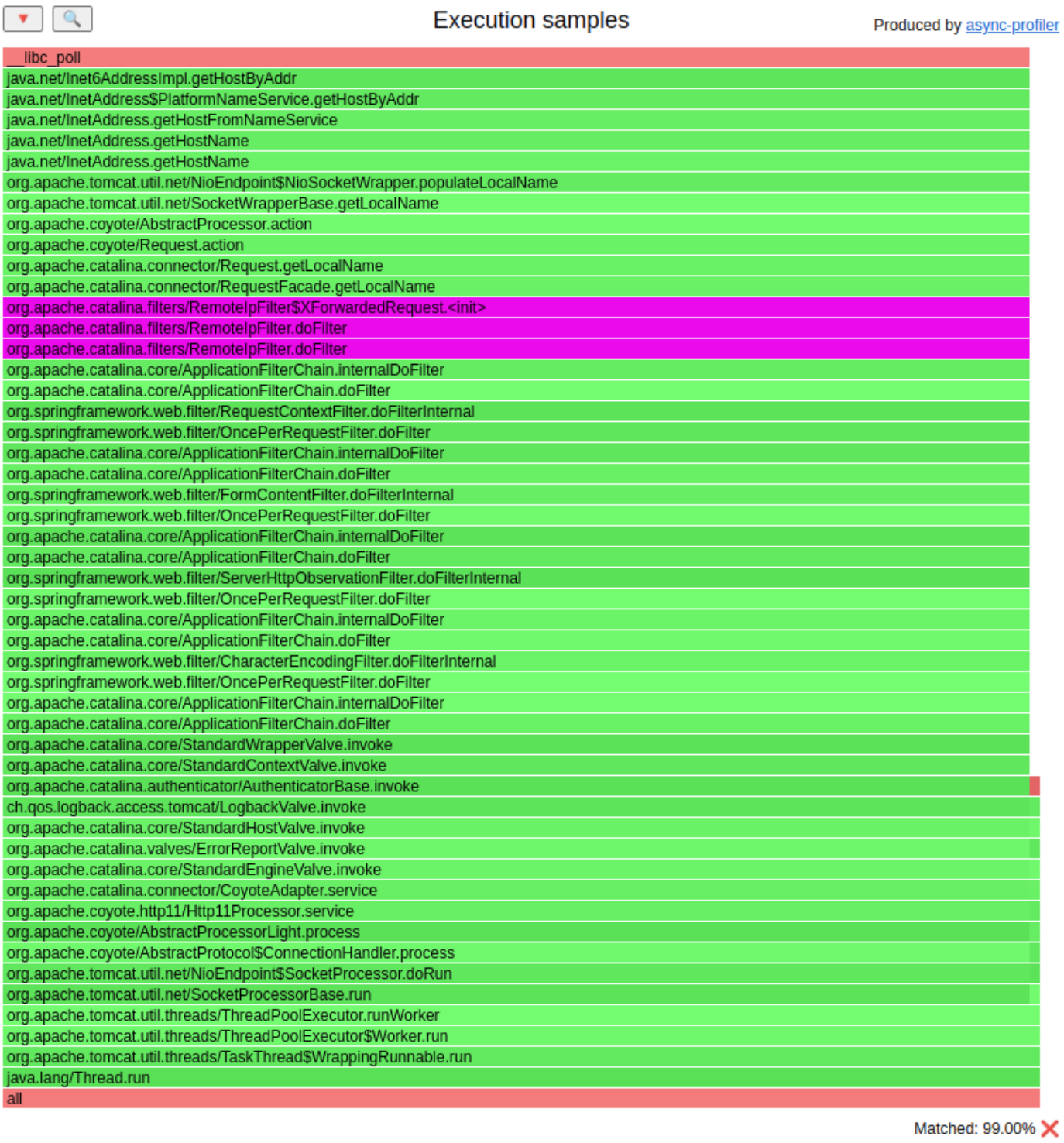
The result of `ab`:

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     0    5 158.1      0    5000
Waiting:        0    5 158.1      0    5000
Total:          0    5 158.1      0    5000

Percentage of the requests served within a specific time (ms)
  50%      0
  66%      0
  75%      0
  80%      0
  90%      0
  95%      0
  98%      0
  99%      1
 100%   5000 (longest request)
```

So almost all the actuator endpoints returned in **0ms**, but at least one lasted **5s**. We can see one longer request in the access logs:

```
[08/DEC/2022:08:56:24 +0000] [GET /actuator/health HTTP/1.0] [200] [0 ms] [http
[08/DEC/2022:08:56:24 +0000] [GET /actuator/health HTTP/1.0] [200] [0 ms] [http
[08/DEC/2022:08:56:24 +0000] [GET /actuator/health HTTP/1.0] [200] [0 ms] [http
[08/DEC/2022:08:56:24 +0000] [GET /actuator/health HTTP/1.0] [200] [0 ms] [http
[08/DEC/2022:08:56:29 +0000] [GET /actuator/health HTTP/1.0] [200] [4999 ms] [ht
[08/DEC/2022:08:56:29 +0000] [GET /actuator/health HTTP/1.0] [200] [0 ms] [http
```

That **5s** response would make our load balancer remove that node (for some time) from a cluster. Let's use the same technique to find out what was the reason for that latency: (HTML)



I highlighted the usage of the `RemoteIpFilter` class. Time for some explanations: When your requests are hitting your application, you can check the IP of the requester with the basic `HttpServletRequest` API. But if you have a load balancer before your application, well, you get an IP of the load balancer, not the original requester. Load balancers usually add HTTP headers to the request to avoid such confusion. The original IP is sent in the `X-Forwarded-For` header. The `RemoteIpFilter` is a tool that makes our lives easier and makes the `HttpServletRequest` API returns proper IP and so on.

Let's get back to the flame graph. We can see that this filter creates an instance of `XForwardedRequest` that executes `RequestFacade.getLocalName()`, that in the end

executes `Inet6AddressImpl.getHostByAddr()`. The last method is trying to identify the hostname by the IP address. How can it be done? Well, we just need a request to DNS, nothing more. In that case, the DNS protocol uses UDP, not TCP. UDP is a protocol that, by design, can lose packets. In Linux, the `resolv.conf` is responsible for configuring DNS and the related tools deal with all the retransmissions and other problems. Here is an excerpt of the manual:

```
timeout:n
       Sets  the amount of time the resolver will wait for a re-
       sponse from a remote  name  server  before  retrying  the
       query  via  a different name server.  This may not be the
       total time taken by any resolver API call and there is no
       guarantee  that a single resolver API call maps to a sin-
       gle  timeout.   Measured  in  seconds,  the  default   is
       RES_TIMEOUT (currently 5, see <resolv.h>).  The value for
       this option is silently capped to 30.
```

Long story short - the default timeout is **5s**. If your DNS request is lost, the tools related to `resolv.conf` will probe the next *nameserver* after **5s**. That's what is happening in our example and what I observed in quite a few Java applications. DNS is commonly used for DDoS attacks. Therefore you can easily have a firewall in your infrastructure that can drop some DNS packets by design.

The funny thing about `RemoteIpFilter` is that the result of that DNS probing is stored in the field `localName` which is not used later. So we are just making DNS requests for nothing. To avoid that problem, write a filter that won't fire DNS requests. `RemoteIpFilter` is open-source, so you can easily use it. There is also `RemoteIpValve` that can be enabled by just an entry in the Spring Boot properties. It used to have the same issue. I didn't check if the issue is still present in Spring Boot 3; it might be fixed accidentally with this bug fix which introduced the `changeLocalName` property. If you want to be sure, you need to check it yourself.

This is not the only DNS request that can be done by the Actuator health check. That endpoint also probes your databases, queues, and many other things. The same may happen if that probing is done by DNS name.

About the hack. If you want to simulate not responsive DNS, you can add 72.66.115.13 (blackhole.webpagetest.org) as your nameserver. That one is designed to drop all the packets. On different Linux distributions, it is done differently. I just use an AWS instance with Amazon Linux distribution, and there I could just edit the `/etc/resolv.conf` file, but in distros like Ubuntu, that file is generated by other services; see ubuntu.com for more information.

Continuous profiling

Let's now focus on a different problem: We had some performance degradation/outage in our system one hour ago. What can we do? The problem is gone, so attaching a profiler now won't help us much. We can start profiling and wait for the problem to occur again, but maybe we can inspire ourselves with a concept used in the aviation business.

In case of an airplane disaster, what is the plane owner doing? Are they adding logs or attaching instruments to the airplane and waiting for the next crash? No, the aviation business has a flight recorder on every plane.

This box records all available data it can during the flight. After any disaster, the data are ready to be analyzed. Can we apply a similar approach to Java profiling? Yes, we can. We can have a profiler attached 24/7 dumping the data every fixed interval of time. If anything

detrimental happens to our application, we have the data that we can analyze.

From my personal experience, continuous profiling is the best technique to diagnose degradations and outages efficiently. It is also handy to understand why the performance differs between two versions of the same application. You only need to get profiles of the previous version from your archives and compare them to the current one.

Here are the ways of enabling async-profiler in continuous mode:

Command line

Here is the simplest way to run async-profiler in continuous mode (dump a profile every **60 seconds** in **wall** mode):

```
while true
do
        CURRENT_DATE=`date +%F_%T`
        ./profiler.sh -e wall -f out-$CURRENT_DATE.jfr -d 60 <pid or applicatio
done
```

It looks dirt simple because it is, but I used that loop many times. Async-profiler includes this with the `--loop` option since version 2.6 in its `profiler.sh` script:

```
./profiler.sh -e wall --loop 1m -f profile-%t.jfr <pid or application name>
```

Java

I already introduced the Java API of AsyncProfiler here. To do it continuously, you can create a thread that is executing:

```
AsyncProfiler asyncProfiler = AsyncProfiler.getInstance();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd_HH:mm:ss")

while (true) {
    String date = formatter.format(LocalDateTime.now());
    asyncProfiler.execute(
        String.format("start,jfr,event=wall,file=out-%s.jfr", date)
    );
    Thread.sleep(60 * 1000);
    asyncProfiler.execute(
        String.format("stop,file=out-%s.jfr", date)
    );
}
```

Spring boot

If you have a Spring/SpringBoot application, you can use a starter written by Michał Rowicki and me:

```
<dependency>
    <groupId>com.github.krzysztofslusarski</groupId>
    <artifactId>continuous-async-profiler-spring-starter</artifactId>
    <version>2.1</version>
</dependency>
```

Read the **README** to get more details.

Contextual profiling

Continuous profiling, together with the possibility to extract a profile of a single request, is very powerful. Unfortunately, there are applications where that is not enough. Some examples:

- Any work that is delegated to a different thread will be missed in that profile
- If one request is computed by multiple threads/JVMs, we need to combine multiple profiles
- Applications in a distributed architecture, usually with microservices:
  There are usually remote calls to other services, even if every single request is processed by a single thread.

We can extract a profile for each microservice to understand the request processing behavior in such a distributed architecture. This is doable but consumes a lot of time.

All the problems mentioned above can be covered by **contextual profiling**. The concept is pretty simple: Whenever any thread is executing any work, that work is done in some context, usually in the context of a single request. Instead of just doing that work, we do the following:

```
asyncProfiler.setContextId(contextId);
actualWork();
asyncProfiler.clearContextId();
```

If any sample is gathered during `actualWork()`, the profiler can add `contextId` to the sample in the JFR file. Such functionality is introduced in Context ID PR. For now that PR is not merged into master, but it's a matter of paperwork; I hope it will be merged soon.

Spring Boot microservices

Let's try to join Spring Boot microservices with contextual profiling. In Spring Boot 3.0 we have included **Micrometer Tracing**. One of its functionalities is generating a **context ID** (called `traceId`) for every request. That `traceId` is passed during the execution to other Spring Boot microservices. We just need to pass that `traceId` to the async-profiler and we are done.

Since that PR is not merged into master, you need to compile the async-profiler from sources. I compiled it on my Ubuntu x86 with glibc here. It may not work on every Linux on every machine. If this is your case, just compile the profiler from sources. It's straightforward.

Ok, let's integrate it with the async-profiler. This time I will use the Java API:

```
public abstract class AsyncProfilerUtils {
    private static volatile AsyncProfiler asyncProfiler;
    // ...

    public static AsyncProfiler load() {
        // Lazy load with double-checked locking
        return asyncProfiler;
    }

    public static void start(String filename) throws IOException {
        load().execute("start,jfr,event=wall,file=" + filename);
    }

    public static void stop(String filename) throws IOException {
        load().execute("stop,jfr,event=wall,file=" + filename);
    }
}
```

I load the profiler from `/tmp/libasyncProfiler.so` and use **wall-clock** mode; I believe it is the most suitable mode for most enterprise applications.

To integrate the profiler with the Micrometer Tracing, we need to implement `ObservationHandler`:

```
public class AsyncProfilerObservationHandler implements ObservationHandler<Obse
    private static final ThreadLocal<TraceContext> LOCAL_TRACE_CONTEXT = new Th

    @Override
    public boolean supportsContext(Observation.Context context) { return true; }

    @Override
    public void onStart(Observation.Context context) {
        TracingContext tracingContext = context.get(TracingContext.class);
        TraceContext traceContext = tracingContext.getSpan().context();
        TraceContext currentTraceContext = LOCAL_TRACE_CONTEXT.get();

        if (currentTraceContext == null ||
                !currentTraceContext.traceId().equals(traceContext.traceId())) {
            LOCAL_TRACE_CONTEXT.set(traceContext);
            AsyncProfilerUtils.load().setContextId(lowerHexToUnsignedLong(trace
        }
    }

    @Override
    public void onError(Observation.Context context) { }

    @Override
    public void onEvent(Observation.Event event, Observation.Context context) {

    @Override
    public void onStop(Observation.Context context) {
        TracingContext tracingContext = context.get(TracingContext.class);
        TraceContext traceContext = tracingContext.getSpan().context();
        TraceContext currentTraceContext = LOCAL_TRACE_CONTEXT.get();

        if (currentTraceContext != null &&
                currentTraceContext.spanId().equals(traceContext.spanId())) {
            LOCAL_TRACE_CONTEXT.remove();
            AsyncProfilerUtils.load().clearContextId();
        }
    }
}
```

Big warning: Don't treat that class as production ready. It's suitable for that example but will not work with any asynchronous/reactive calls. Mind that `onStart/onStop` can be called

multiple times with the same `traceId` and different `spanId`.

Now we need to register that implementation:

```
@Bean
@Profile("context")
ObservedAspect observedAspect(ObservationRegistry observationRegistry) {
    observationRegistry.observationConfig().observationHandler(new AsyncProfile
    return new ObservedAspect(observationRegistry);
}
```

That will also register us using the `@Observed` aspect.

And that's it. Let's try it out. We need to rerun the Spring Boot applications with active profiling:

```
java -Xms1G -Xmx1G \
-Dspring.profiles.active=context \
-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints \
-jar first-application/target/first-application-0.0.1-SNAPSHOT.jar

java -Xms1G -Xmx1G \
-Dspring.profiles.active=context \
-jar second-application/target/second-application-0.0.1-SNAPSHOT.jar

java -Xms1G -Xmx1G \
-Dspring.profiles.active=context \
-jar third-application/target/third-application-0.0.1-SNAPSHOT.jar
```

Now the applications will look for an async-profiler in `/tmp/libasyncProfiler.so`.

```
# little warmup
ab -n 24 -c 1 http://localhost:8081/examples/context/observe

# profiling time - this time, we start profiler from Java
curl -v http://localhost:8081/examples/context/start
curl -v http://localhost:8082/examples/context/start
curl -v http://localhost:8083/examples/context/start

ab -n 24 -c 1 http://localhost:8081/examples/context/observe

# stopping the profiler
curl -v http://localhost:8081/examples/context/stop
curl -v http://localhost:8082/examples/context/stop
curl -v http://localhost:8083/examples/context/stop
```

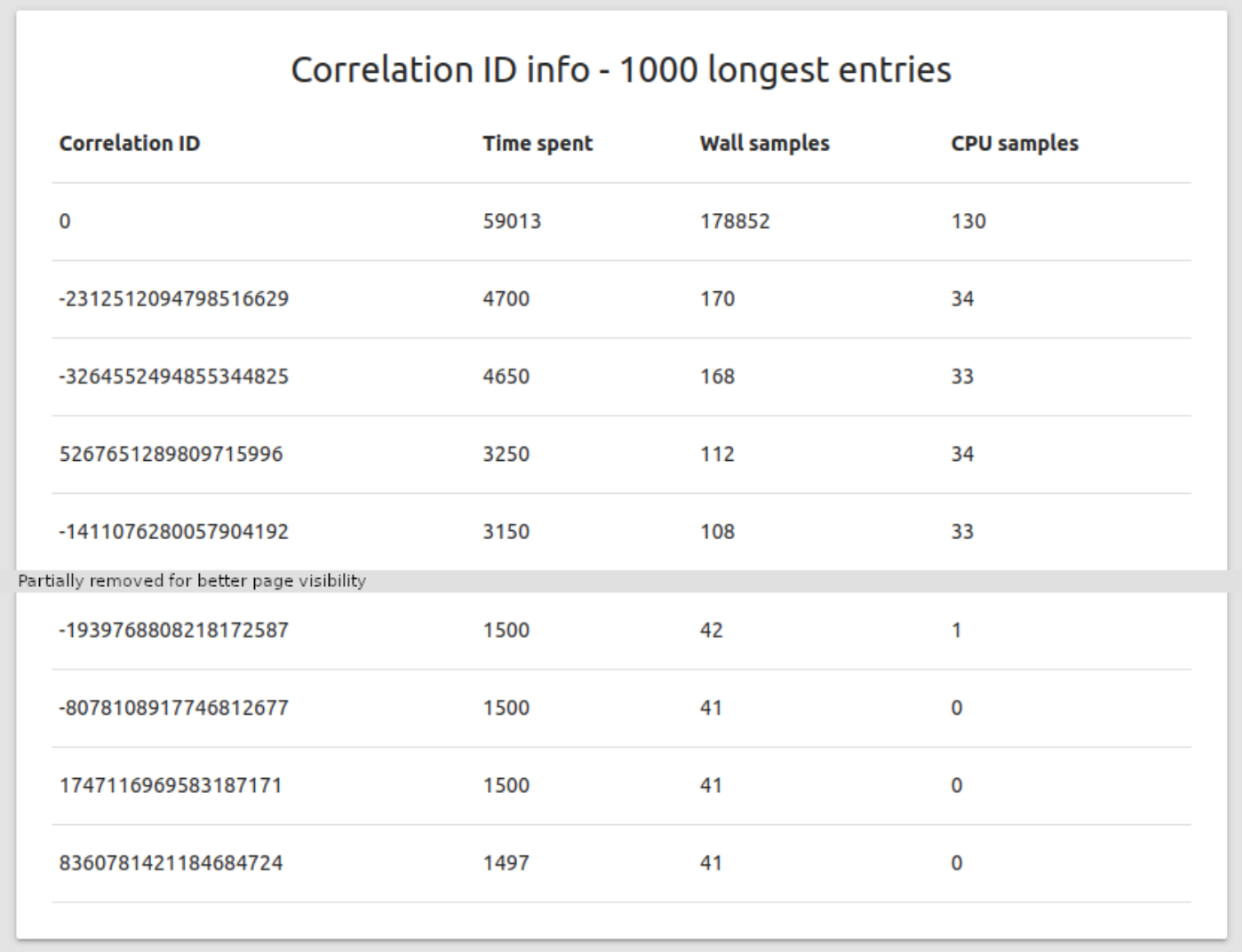
Let’s look at the timings during profiling. I’ve cut the output to 12 rows:

```
04/gru/2022:20:54:25 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1538
04/gru/2022:20:54:26 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1537
04/gru/2022:20:54:29 +0100 [GET /examples/context/observe HTTP/1.0] [200] [3039
04/gru/2022:20:54:33 +0100 [GET /examples/context/observe HTTP/1.0] [200] [3218
04/gru/2022:20:54:34 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1538
04/gru/2022:20:54:37 +0100 [GET /examples/context/observe HTTP/1.0] [200] [3038
04/gru/2022:20:54:39 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1538
04/gru/2022:20:54:42 +0100 [GET /examples/context/observe HTTP/1.0] [200] [3270
04/gru/2022:20:54:45 +0100 [GET /examples/context/observe HTTP/1.0] [200] [3038
04/gru/2022:20:54:47 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1536
04/gru/2022:20:54:48 +0100 [GET /examples/context/observe HTTP/1.0] [200] [1536
04/gru/2022:20:54:53 +0100 [GET /examples/context/observe HTTP/1.0] [200] [4756
```

We can see that we have three groups of timings:

- `~1500ms`
- `~3000ms`
- `~4500ms`

After we executed the script above, we had three JFR files in the `/tmp` directory. When we load all three files together to my viewer and check the *Correlation ID stats* section, we can see:



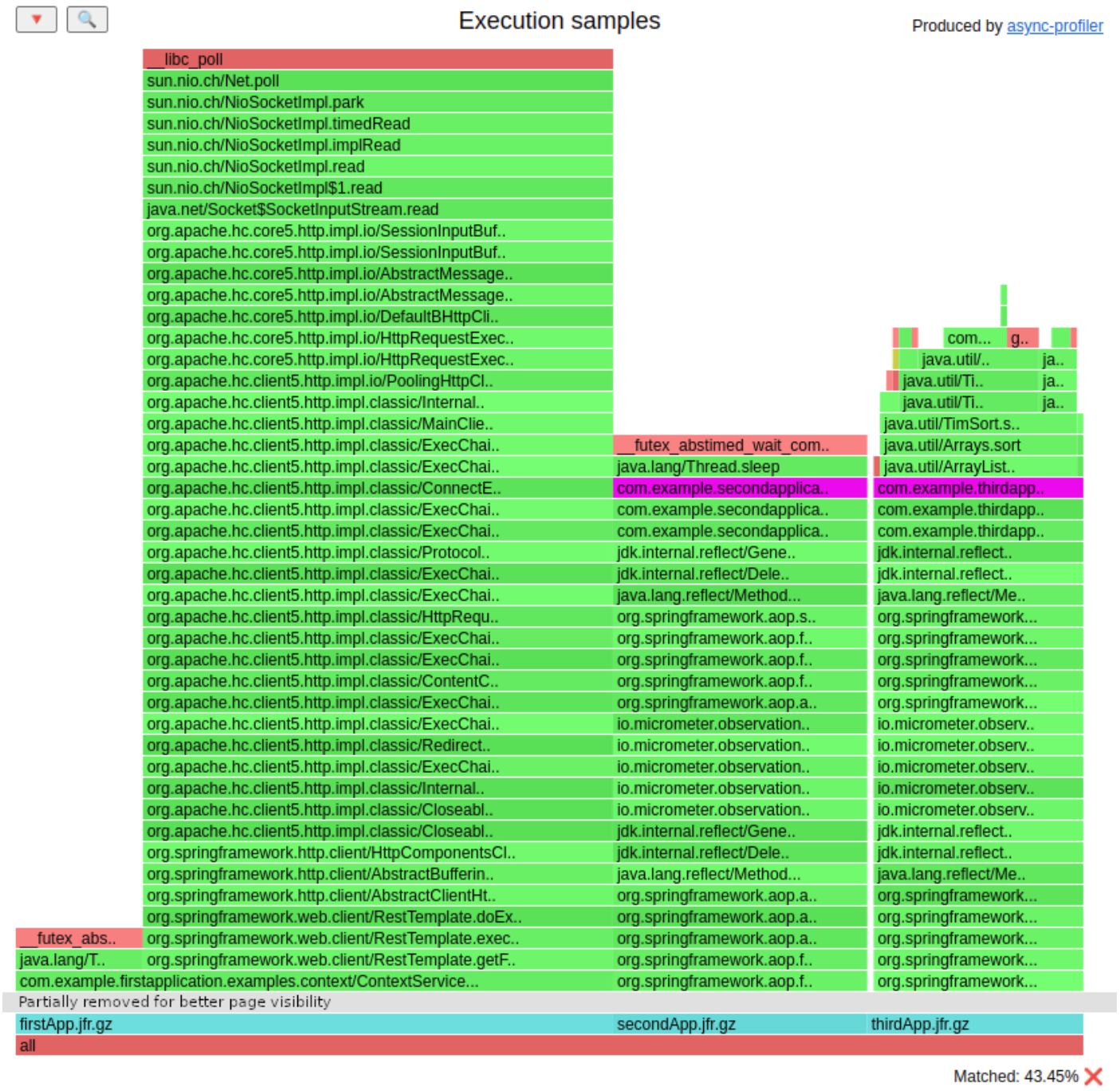
Correlation ID info - 1000 longest entries

Correlation ID	Time spent	Wall samples	CPU samples
0	59013	178852	130
-2312512094798516629	4700	170	34
-3264552494855344825	4650	168	33
5267651289809715996	3250	112	34
-1411076280057904192	3150	108	33

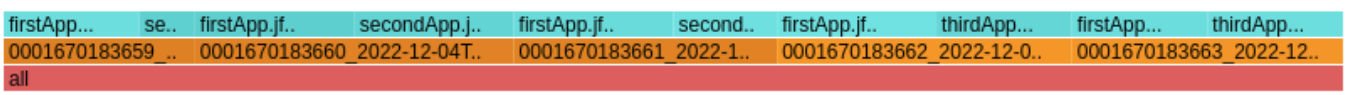
Partially removed for better page visibility

-1939768808218172587	1500	42	1
-8078108917746812677	1500	41	0
1747116969583187171	1500	41	0
8360781421184684724	1497	41	0

So we have similar timings from our JFR files. Looking good. Let's filter all the samples by context ID. It's called a *Correlation ID filter* in my viewer, let's use a value `-3264552494855344825` which took `4650ms` according to records in the JFR. Let's also add an additional *filename level*. The filename is correlated to the application name. Here comes the flame graph: (HTML)

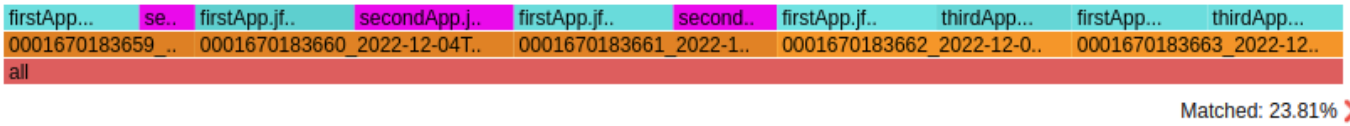


All three applications are on the same flame graph. This is beautiful. Just a reminder: it's not a whole application. It's a **single request** presented here. I highlighted the `slowPath()` method executed in the second and third app, which causes higher latency. You can play with the HTML flame graph to see what is happening there or jump into the code. I want to focus on what insights the context ID functionality gives us. Because there is more. We've already added an additional *filename level*. We can also add timestamps as another level. Let's do that. I will present you only the bottom of the graph since that's what is important here: (HTML)

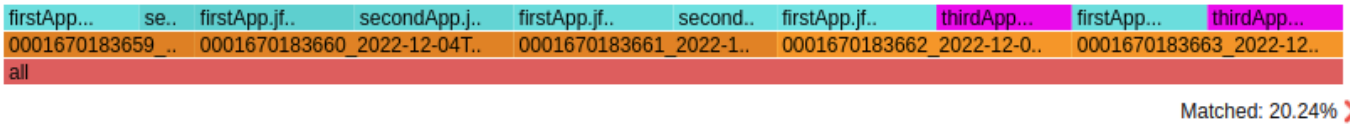


The image may look blurred, but you can check the HTML version for clarity. At the bottom,

you can see five brown rectangles. Those are timestamps truncated to seconds. So from left to right, we can see what was happening to our request second by second. Let's highlight when the second application was running during that request:



And the third:



The first application is always running since it's the entry point to our distributed architecture. The important code in the second application is the following:

```
@RequiredArgsConstructor
class ContextService {
    // ...
    void doSomething() {
        if (counter.incrementAndGet() % 3 == 0) {
            slowPath();
            return;
        }

        fastPath();
    }

    private void fastPath() {
        // ...
    }

    private void slowPath() {
        // ...
    }
}
```

And the third application:

```
class ContextService {
    // ...
    void doSomething() {
        if (counter.incrementAndGet() % 4 == 0) {
            blackhole = slowPath();
            return;
        }
```

```
        blackhole = fastPath();
    }

    private int slowPath() {
        // ...
    }

    private int fastPath() {
        // ...
    }
}
```

So the second application is slower every third request, and the third application is slower every fourth request. That means that for every twelfth request, we are slower in both of them.

I firmly believe that contextual profiling is the future in that area. Everything you see here should be taken with a grain of salt. My Spring integration and my JFR viewer are far away from something professional. I want to inspire you to search for new possibilities like that. If you find any, share it with the rest of the Java performance community.

Distributed systems

First, let's differentiate distributed architecture from distributed systems. For this post, let's assume the following:

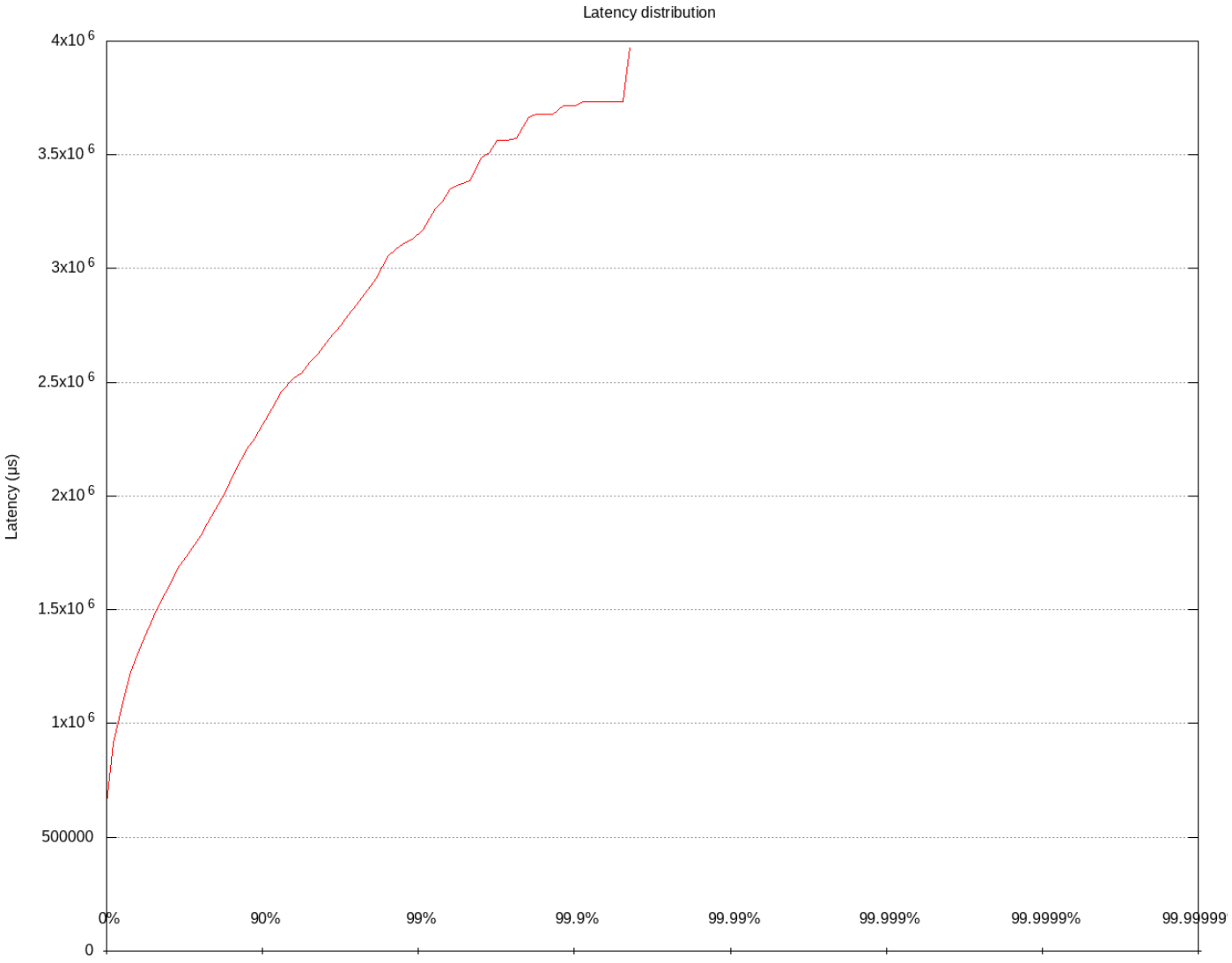
- a distributed architecture is a set of applications that works together - like microservices
- a distributed system is one application that is deployed on more than one JVM to service some request it distributes the work to more than one instance

One example of a distributed system may be Hazelcast. I've applied the context ID functionality to trace the tail of the latency in SQL queries.

Sample benchmark details:

- Hazelcast cluster size: **4**
- Servers with **Intel Xeon CPU E5-2687W**
- Heap size: **10 GB**
- **JDK17**
- SQL query that is benchmarked: `select count(*) from iMap`
- iMap size – **1 million** serialized Java objects
- Benchmark duration: **8 minutes**

The latency distribution for that benchmark is:

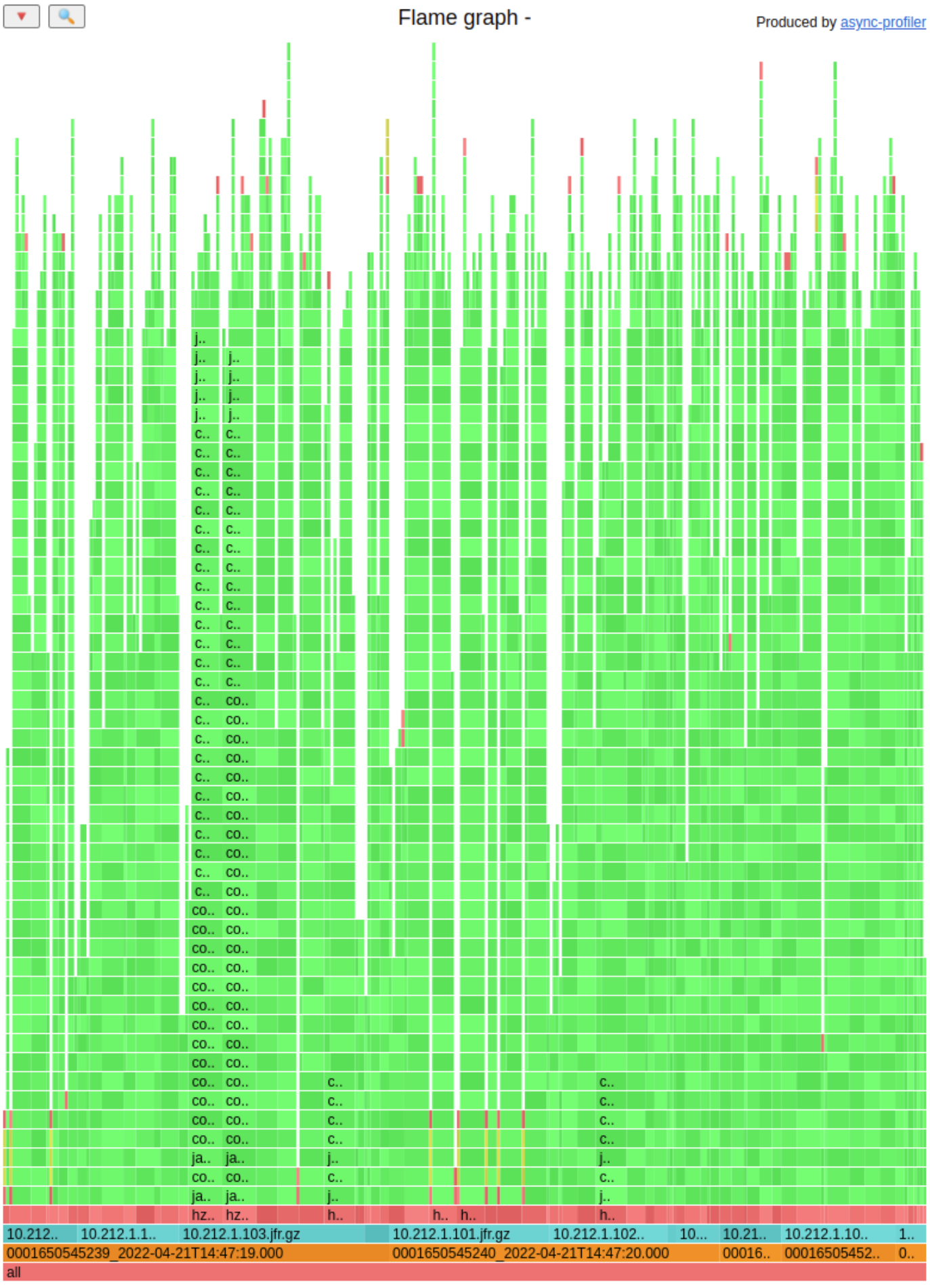


The **50th** percentile is **1470 ms**, whereas the **99.9th** is **3718 ms**. Let's now analyze the **JFR** file with my tool. I've created a table with the longest queries in the files:

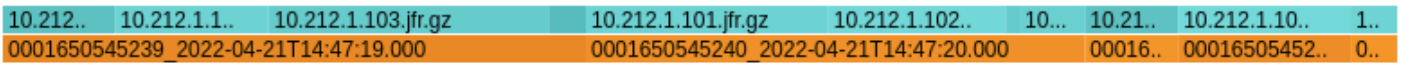
ECID info - 1000 longest ECIDs

ECID	Time spent	Wall samples	CPU samples
-1810992221072681376	3781	299	289
876899767738934713	3452	272	272
5310803793989643812	3446	268	264
8536260265930420694	3431	265	259
6719901203764079127	3348	256	255

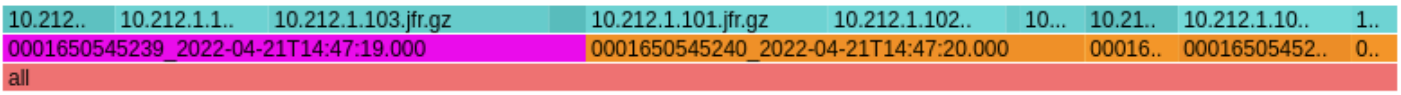
Let's analyze a single query using the context ID functionality. The full flame graph:


Flame graph -
Produced by async-profiler
10.212..
10.212.1.1..
10.212.1.103.jfr.gz
10.212.1.101.jfr.gz
10.212.1.102..
10...
10.21..
10.212.1.10..
1..
0001650545239_2022-04-21T14:47:19.000
0001650545240_2022-04-21T14:47:20.000
00016..
00016505452..
0..
all

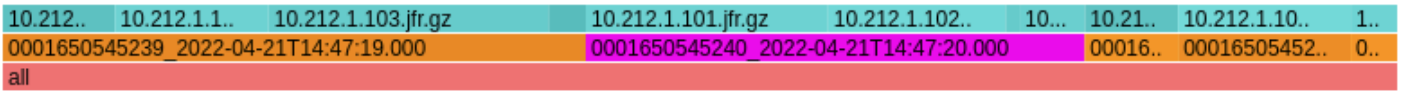
Let's focus on the bottom rows:



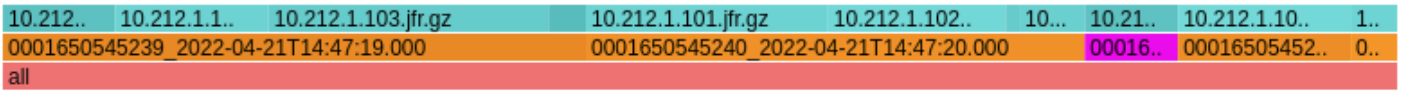
Let's start with timestamps and highlight them one by one:



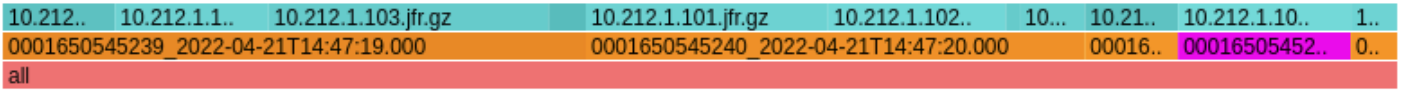
Matched: 41.81%



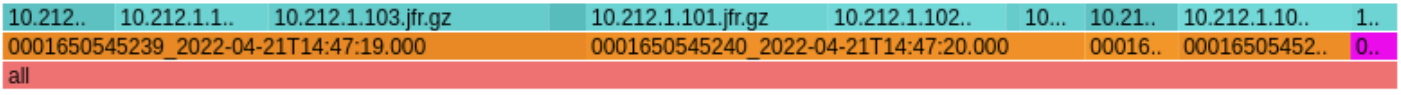
Matched: 35.79%



Matched: 6.69%



Matched: 12.37%

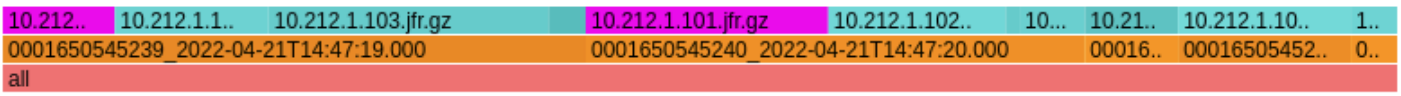


Matched: 3.34%

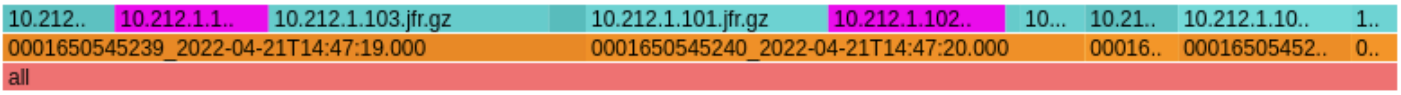
Summing that up:

- **41.91%** of samples were gathered between 14:47:19 and 14:47:20
- **35.79%** of samples were gathered between 14:47:20 and 14:47:21
- **6.68%** of samples were gathered between 14:47:21 and 14:47:22
- **12.37%** of samples were gathered between 14:47:22 and 14:47:23
- **3.34%** of samples were gathered between 14:47:23 and 14:47:24

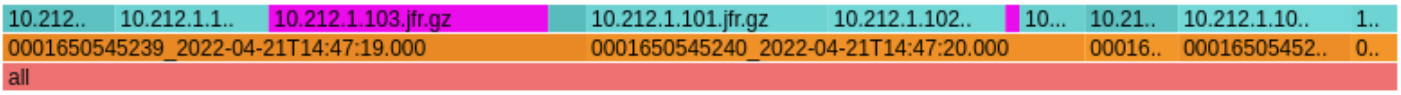
Let's highlight the filenames (which are named with the **IP** of the server) one by one:



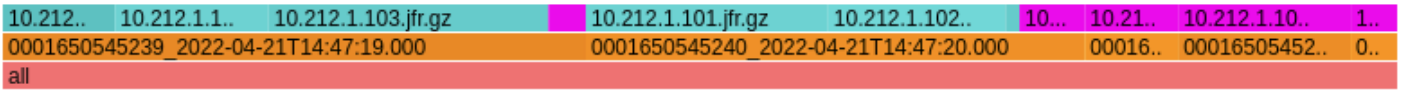
Matched: 25.42%



Matched: 23.75%



Matched: 21.07%



Matched: 29.77%

A little summary:

- **25.42%** of samples are from node 10.212.1.101
- **23,75%** of samples are from node 10.212.1.102
- **21.07%** of samples are from node 10.212.1.103
- **29.77%** of samples are from node 10.212.1.104

Since brown bars are sorted alphabetically by timestamps, we can conclude that the **10.212.1.104** server is doing any work in the last seconds of processing.

I checked five more long latency requests and the results were the same, confirming that the **10.212.1.104** server is the problem. I spent a lot of time trying to figure out what was wrong with that machine. My biggest suspect was a difference in meltdown/spectre patches in the kernel. In the end, we reinstalled Linux on those machines, which solved the problem with the **10.212.1.104** server.

Stability

Attaching a profiler to a JVM can potentially cause it to crash. It's essential to be aware of this risk, as bugs can occur not only in profilers but also in the JVM itself. The OpenJDK developers are working on improving the stability of the API used by profilers to minimize this risk. You can learn more about their work at:

- Johannes Bechberger
- Jaroslav Bachorik

In my opinion, async-profiler is a very mature product already. I know a few companies that are running async-profiler in continuous mode 24/7. Over two years, I only heard about one production crash caused by a profiler. It is working with **40** production JVMs, at least in wall-clock mode there. I have used async-profiler on multiple systems without crashes this year. While there is always some risk when attaching a profiler to a JVM, I believe the risk is minimal and can be safely ignored.

However, if you experience a profiler-related crash, I encourage you to file a GitHub issue to help improve the OpenJDK. During the crash, the `hs_err.<pid>` file is generated. It may be beneficial for finding the root cause of a problem.

The main problem, according to Johannes Bechberger, stated in a recent JEP proposal, is that async-profiler uses the internal AsyncGetCallTrace API for stack walking. This API was introduced in November 2002 for Sun Studio but was removed in January 2003 and demoted to an internal API (JVMTI). It is neither exported in any header nor standardized. To this date, there is only one tiny test in the whole OpenJDK: The API might be broken with any version, Johannes Bechberger caught such an issue with PR 7559 before the release. Be aware of this risk and test every JDK with your profiling setup before using it in production.

There is an ongoing effort by Johannes Bechberger, with the help of Jaroslav Bachorik and others, to improve this situation by proposing the new AsyncGetStackTrace API, that will hopefully be integrated into the OpenJDK. This API will be official, well-tested with stability and stress tests in the official OpenJDK test suite, and therefore more stable than AsyncGetCallTrace. It will also give the users of tools like async-profiler more information, like C/C++ frames between Java frames and inlining information for all Java frames. If you want to learn more, consider reading the JEP or visit the demo repository to see it in action.

Furthermore, many bugs have been found by both OpenJDK developers by using the JDK Profiling Tester to find and fix many stability issues. There are currently no known real-world stability issues.

Overhead

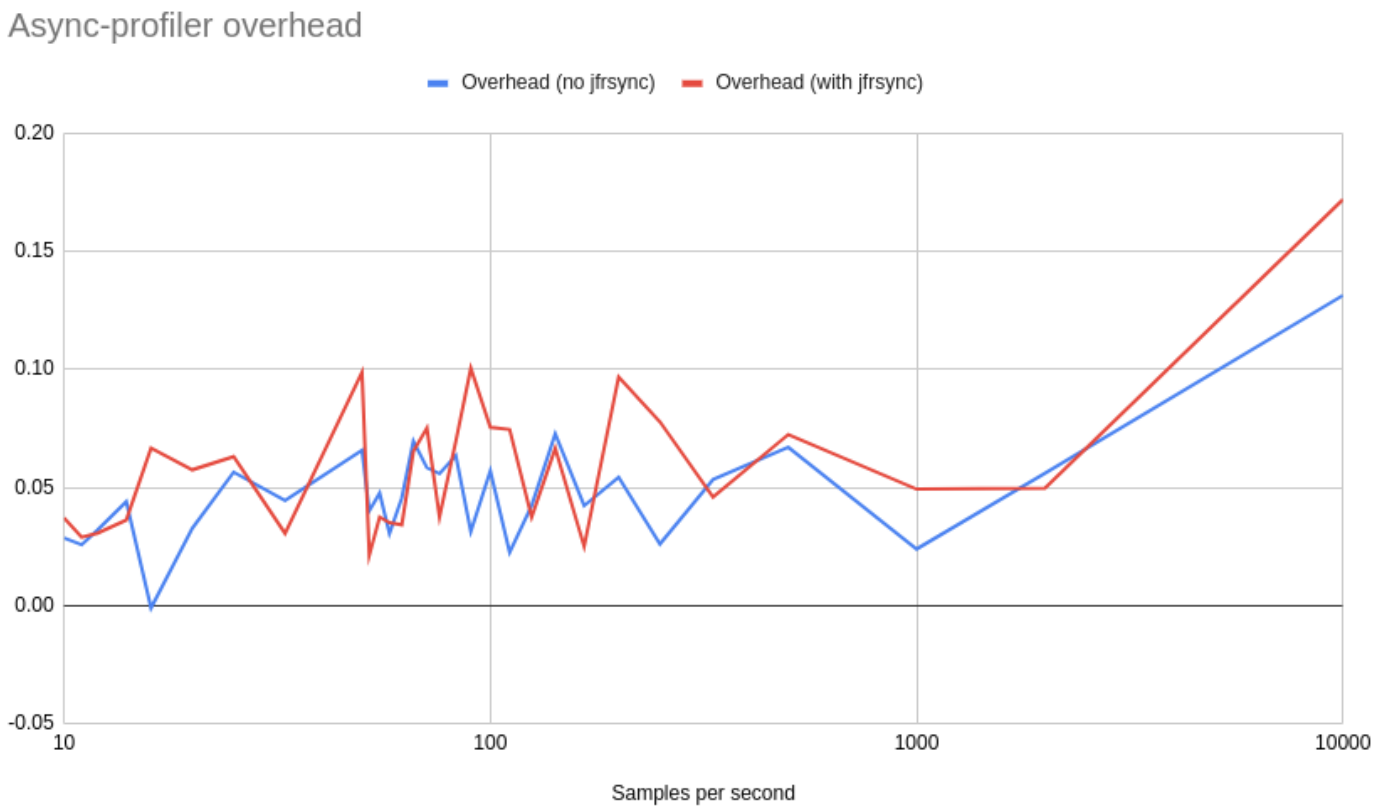
In the application where the profiler is running in continuous mode on production the overhead (in terms of response time) is typically between **0%** and **2%**. That number is a comparison of response times before and after introducing continuous profiling there. A bit of context:

- Spring and Spring Boot applications
- Mostly services that handle HTTP requests
- Not really CPU intensive - I would say that on average **60%** of the request time was spent off-CPU (waiting for DB/other service)
- JDK 11 and 17 - HotSpot from various vendors
- Wall-clock event, dump of JFR every minute from Java API
- Environment provided by VMware, both VMs and Tanzu clusters
- Async-profiler 1.8.x, later 2.8.x

Johannes Bechberger shared his benchmark results with me. He used the DaCapo Benchmark Suite:

- ThreadRipper 3995WX with 128GB RAM
- Async-profiler 2.8.3
- `dacapo benchmarks avrora fop h2 jython lusearch pmd -t 8 -n 3`
- CPU event

Johannes's results shows **~6%** overhead on default sampling interval without `jfrsync` flag, and **~7.5%** with `jfrsync`. The chart for his results:



The logarithmic-scaled X-axis is the number of samples per second, and the Y-axis is the additional overhead.

Remember: **You should always measure the overhead in your application by yourself and configure the profiling interval and captured events according to your specific needs.**.

Random thoughts

1. You need to remember that EVERY profiler lies in some way. The async-profiler is vulnerable to JDK-8281677. There is nothing that the profiler can do; JVM is lying to the profiler, so that lie is passed to the end user. You can change the mechanism used by a profiler, but you will be lied to, maybe differently.
2. You can run an async-profiler to collect more than one event. It is allowed to gather `lock` and `alloc` together with one of the modes that gathers execution samples, like `cpu`, `method`, ...
3. You can run an async-profiler with the `jfrsync` option that will gather more information exposed by the JVM, but be aware to use the `alloc` option for information on allocations. This way, you can also capture GC information and more.

If you want to know more on this topic, consider the curated collection of blogs and other resources you find here and the YouTube playlist with in-depth talks on profiling. Consider contacting Johannes Bechberger, who curates both, if you have any suggestions.

This page was generated by GitHub Pages.