Liberica JDK
Using perf to monitor Java performance

Liberica JDK
Revision 1.0
October 17, 2023

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Contents

1. Introduction

perf is a profiler built into the Linux kernel with features such as:

- Profiling kernel execution
- Profiling based on hardware performance counters

perf can analyze the execution of kernel code alongside user space application code. This is especially convenient for troubleshooting IO performance issues.

Another useful perf feature is hardware performance counters — special CPU registers that track various kinds of low-level events (e.g. CPU cache misses). You can configure perf to use an internal CPU event for sampling, thus building a specific view of program performance, e.g., cache miss or branch misprediction hotspots. Below you will find a Linux perf tutorial for JVM applications and the comparison of perf and JFR.

2. Using perf with JVM

Let's try profiling a Java program with perf. For that purpose, you require Linux, Liberica JDK 17, and a ready Java application.

Below is a simple Java program that benchmarks the performance of the MD5 hash.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class CryptoBench {
    private static final boolean trackTime = Boolean.getBoolean("trackTime");
    public static void main(String[] args) {
        CryptoBench test = new CryptoBench();
        while(true) {
            test.execute();
        }
    }
    public void execute() {
        long N = 5 * 1000 * 1000;
        RandomStringUtils randomStringUtils = new RandomStringUtils();
        long ts = 0,tf = 0;
        long timer1 = 0;
        long timer2 = 0;
        long bs = System.nanoTime();
        for (long i = 0; i < N; i++) {
            ts = trackTime ? System.nanoTime() : 0;
            String text = randomStringUtils.generate();
            tf = trackTime ? System.nanoTime() : 0;
            timer1 += tf - ts;
            ts = tf;
            crypt(text);
            tf = trackTime ? System.nanoTime() : 0;
            timer2 += tf - ts;
            ts = tf;
        }
        long bt = System.nanoTime() - bs;
        System.out.print(String.format("Hash rate: %.2f Mm/s", 0.01 * (N *
TimeUnit.SECONDS.toNanos(1) / bt / 10000)));
        if (trackTime) {
```

```
            System.out.print(String.format(" | Generation: %.1f %%",  0.1 *
(1000 * timer1 / (timer1 + timer2))));
            System.out.print(String.format(" | Hasing: %.1f %%", 0.1 * (1000 *
timer2 / (timer1 + timer2))));
        }
        System.out.println();
    }
    public String crypt(String str) {
        if (str == null || str.length() == 0) {
            throw new IllegalArgumentException("String to encrypt cannot be
null or zero length");
        }
        StringBuilder hexString = new StringBuilder();
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(str.getBytes());
            byte[] hash = md.digest();
            for (byte aHash : hash) {
                if ((0xff & aHash) < 0x10) {
                    hexString.append("0" + Integer.toHexString((0xFF &
aHash)));
                } else {
                    hexString.append(Integer.toHexString(0xFF & aHash));
                }
            }
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return hexString.toString();
    }
}

class RandomStringUtils {
    public String generate() {
        int leftLimit = 97; // letter 'a'
        int rightLimit = 122; // letter 'z'
        int targetStringLength = 10;
        Random random = new Random();
        StringBuilder buffer = new StringBuilder(targetStringLength);
        for (int i = 0; i < targetStringLength; i++) {
            int randomLimitedInt = leftLimit + (int)
                (random.nextFloat() * (rightLimit - leftLimit + 1));
            buffer.append((char) randomLimitedInt);
        }
```

```
        return buffer.toString();
    }
}
```

Compile the application, then run it with Liberica JDK.

```
java -XX:+PreserveFramePointer -cp . CryptoBench
```

Additional option `-XX:+PreserveFramePointer` is required to allow perf to parse stack frames from Java code. Without this option, perf won't be able to reconstruct the call tree of JIT-compiled methods. Results could still be useful if you are interested in native library or kernel hot spots triggered by your Java code.

Let it run and use the second terminal to start profiling.

The exact way to install perf depends on your Linux distro, as perf is a part of linux-tools package family.

In some cases, such as examples in this document, root is required to use perf, although it is possible to set up perf to be used by non-root users.

Let's find PID of our test program:

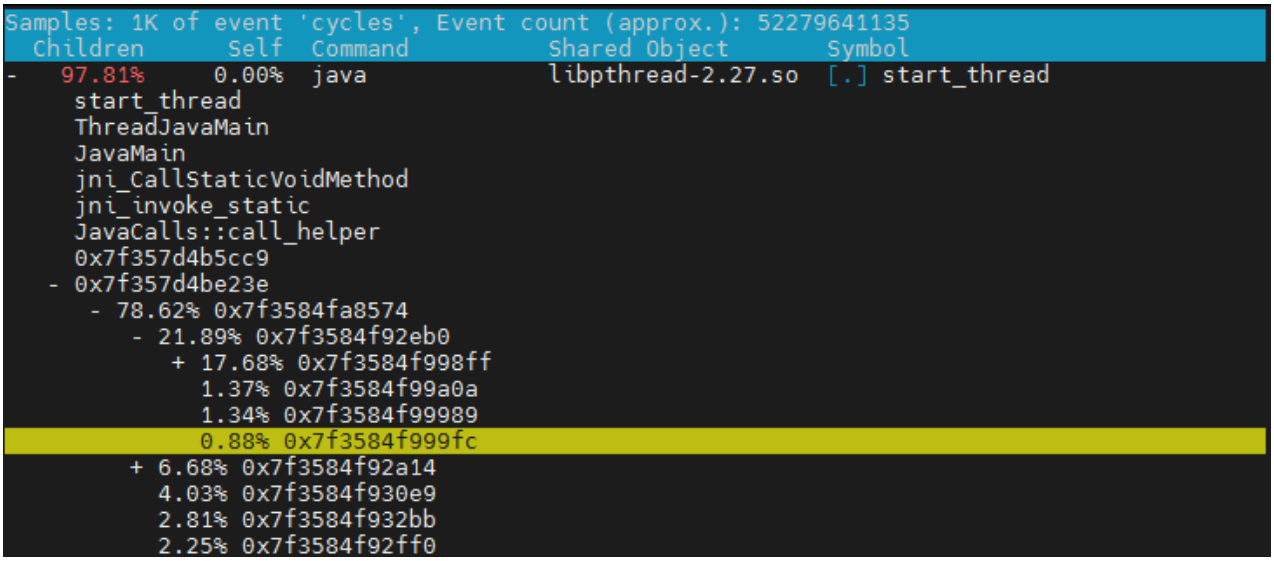
```
jcmd | grep CryptoBench
```

Now, we can run perf to collect stack trace samples (replace 1234 with PID you get after running the previous command):

```
sudo perf record -F 500 -p 1234 -g -o perf1.data – sleep 10
```

This command will sample all threads in the JVM process for 10 seconds using FP stackwalk to capture the traces 500 times per second. Data will be saved into the perf1.data file.

We can try to run `sudo perf report -i perf1.data`. This command calculates a frame histogram and allows you to browse it conveniently.



perf collected stack trace samples in a data file. By inspecting the above screenshot, we can see that something is definitely happening in threads started by JVM, but most of the code is presented with obscure memory addresses. This happens because perf records only the addresses of the call chain from the stack. In the case of JVM, these would be either part of JVM binary or JIT-compiled code.

To get the meaningful data, we have to utilize a symbol map to convert memory addresses into meaningful method names.

For statically compiled code, symbols can be distributed alongside the binary.

For JIT-compiled code, we have to create a code symbol map suitable for perf. It should be generated at runtime, preferably shortly after sampling is complete, as JIT-compiled code can be garbage collected over time, and addresses could be reused.

By default, perf is looking for /tmp/perf-PID.map for a file with symbols for JIT-produced machine code.

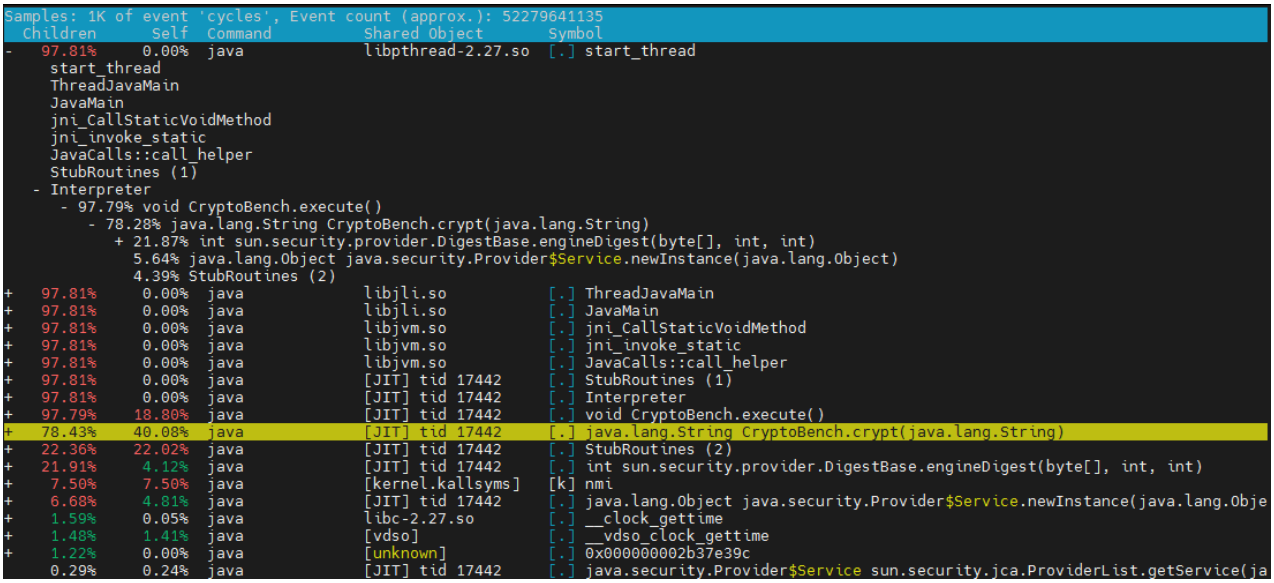
Several enhancements were introduced into JDK 16 concerning perf serviceability. For example, a new command line option, `-XX:+DumpPerfMapAtExit`, was added to write a perf map file on VM shutdown:

```
perf record – java -XX:+DumpPerfMapAtExit Workload
```

Furthermore, starting with OpenJDK 17, we can generate a map file with jcmd like this:

```
jcmd 1234 Compiler.perfmap
```

Now, the same command `sudo perf report -i perf1.data` provides a much more helpful report.



The report now displays Java method names.

Prior to OpenJDK 17, using perf with JVM was also an option, but it required an external tool (like perf-map-agent) to generate a symbol map.

We added the `-F 500` option to sample program execution with an approximate frequency of 500 Hz based on CPU cycles. perf utilizes the CPU interrupts, so only threads running on the CPU are sampled.

Default perf reporting mode is a frame histogram (similar to the hot method histogram in Java profilers). This way of presenting the data is convenient but usually requires some analysis to make sense.

Built-in perf visualization options are pretty basic, although you can export raw samples with the `perf script` command and use 3rd party tools and scripts to generate more sleek-looking reports.

3. Comparing JFR with perf using flame graph

A flame graph is a visual representation style for stack trace sampling data. It's a good starting point for analysis. By utilizing flame graphs, we can promptly compare data collected by perf with JFR-generated data, and a similar presentation style makes it easier.

4. Creating flame graph from perf data

We will use scripts from Brendan Greg to produce the flame graph with perf collected data.

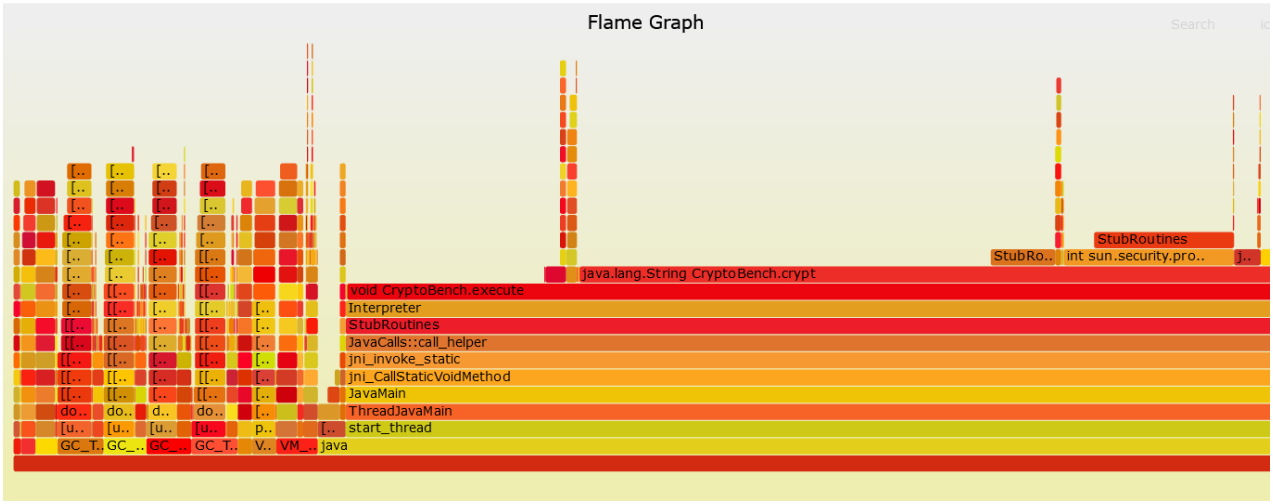
Let's check out some scripts.

```
git clone https://github.com/brendangregg/FlameGraph.git
```

Now we can generate a flame graph for the data file we already have.

```
perf script -i perf1.data | FlameGraph/stackcollapse-perf.pl |
FlameGraph/flamegraph.pl > perf_flame.svg
```

This command generates a report that looks like this:



5. Creating flame graph with JFR

Below is a sequence of commands to capture data with JFR.

```
jcmd 1234 JFR.start settings=profile
1234 :
Started recording 1. No limit specified, using maxsize=250MB as default.
Use jcmd 1234 JFR.dump name=1 filename=FILEPATH to copy recording data
to file.
```

Wait for half a minute and run the following command.

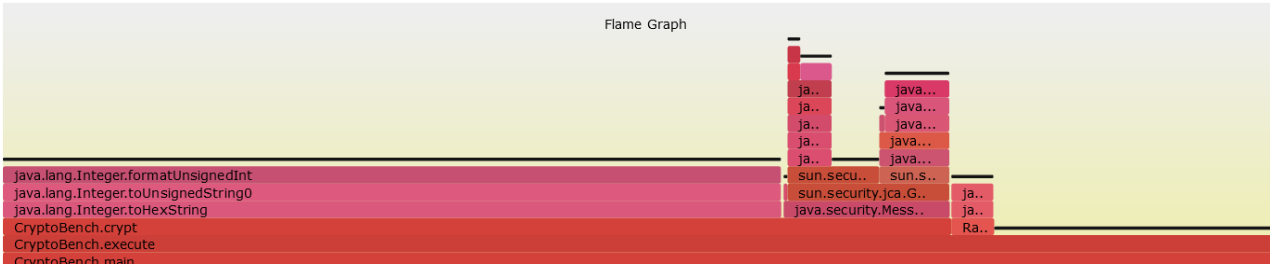
```
jcmd 1234 JFR.dump name=1 filename=perf1.jfr
```

There are multiple ways to convert JFR data to flame graphs, one of which is using SJK.

```
wget https://bit.ly/2H3Uqck -O sjk.jar
```

```
java -jar sjk.jar ssa -f perf1.jfr -flame > jfr1.svg
```

The above code generates a flame graph based on JFR sampling.



6. Comparing flame graphs

We can now compare the results of JFR and perf. A closer look reveals that the graphs look sufficiently different.

- perf includes samples from GC threads occasionally running.
- perf includes frames from JVM routines.
- "CryptoBench.main" is replaced by "Interpreter" on the perf flame graph.
- JFR shows diverse calls from "CryptoBench.crypt" including string manipulations missing on perf's chart.

It is common for different profilers to display different results after inspecting the same application. The root cause of this discrepancy is a specific approach to sampling and data interpretation.

7. Under the hood of sampling machinery

perf approach to sampling

perf sampling is driven by events, in the example above we used cycles (CPU cycles) as a driver for sampling. Once the event counter reaches the crossing sampling period, a stack trace is captured and recorded.

perf walking stack, assuming that frames are linked via FP (frame pointer), registers and collects return addresses. These addresses are further translated into symbolic method names at the time of reporting.

Stack has to be immutable while it is being walked. Being a part of the Linux kernel, perf can execute its probe in the context of running a thread without freezing it (usually though interrupt). Yet, perf is still subject to `http://www.spinics.net/lists/linux-perf-users/msg02157.html[skid]', because even hardware interrupt cannot freeze thread execution in an instant.

Provided both stack walk and thread interrupts are quick and efficient, perf can do sampling at very high frequencies (thousands of samples per core every second).

When the program code is idle, such as waiting for semaphore or IO, it usually does not produce events and is omitted from the sample population. This issue can be solved by the following feature — perf can trace thread state changes and IO-specific events separately using scheduler events.

JFR approach to sampling

Like most Java profilers, JFR uses internal JVM structures to reconstruct stack traces. There are multiple reasons why simple stack walking is not good for JVM:

- Java code can be interpreted or JIT-compiled, in the case of interpreted code, the address of interpreter runtime will be on the stack.
- JIT compiler inlines method calls aggressively, so a single stack frame may represent multiple nested calls.

In the previous example, several calls have been included into the compiled body of the `CryptoBench.crypt` method. They become invisible to perf, but JFR shows them.

JVM has information to reconstruct idiomatic stack traces in its internal data structure, although it cannot be accessed concurrently. A thread has to be stopped by the profiler (or the profiler should run its code on behalf of the thread).

The easiest way to achieve that is the "Stop-the-World" pause — this is the way JVM thread dumps work, and many JVM profilers are using the same method for sampling.

Stop-the-World pause is a relatively heavy operation, it also introduces bias into samples (a safepoint bias).

JFR utilizes a more efficient approach. It stops each thread individually and does not rely on safepoints. It still requires a number of syscalls for each thread though.

JFR method sampling also excludes idle (sleeping/blocking) threads from sampling.

8. Safepoint bias

While neither perf nor JFR relies on JVM safepoints, both are affected by some form of safepoint bias. By default, JVM emits symbols only for instructions, which can yield a safepoint, assuming that the stack trace can be calculated only under Stop-the-World condition.

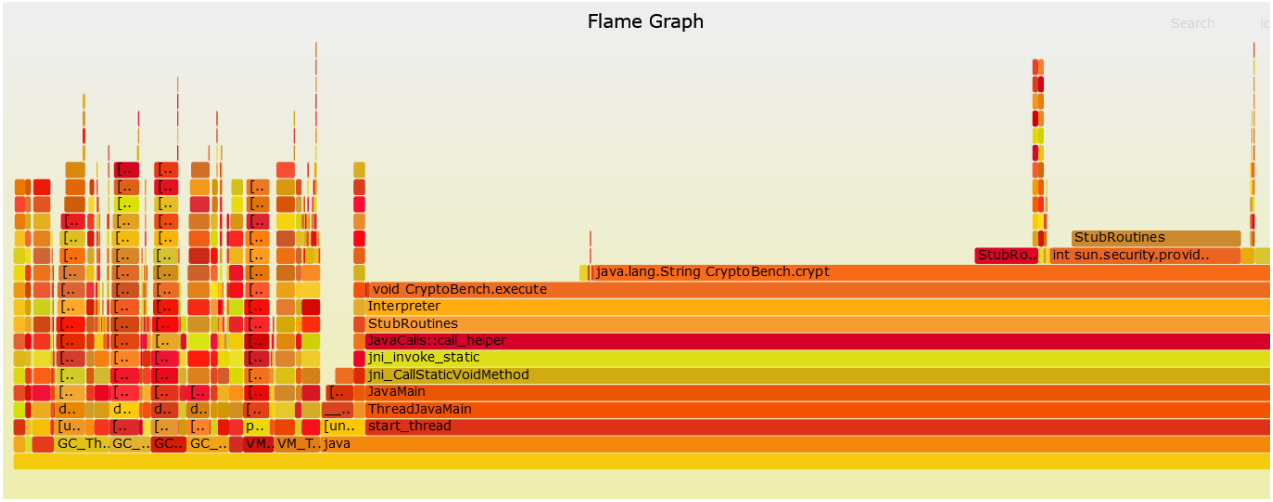
This behavior can be changed by adding the following options to the JVM start command

```
-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints
```

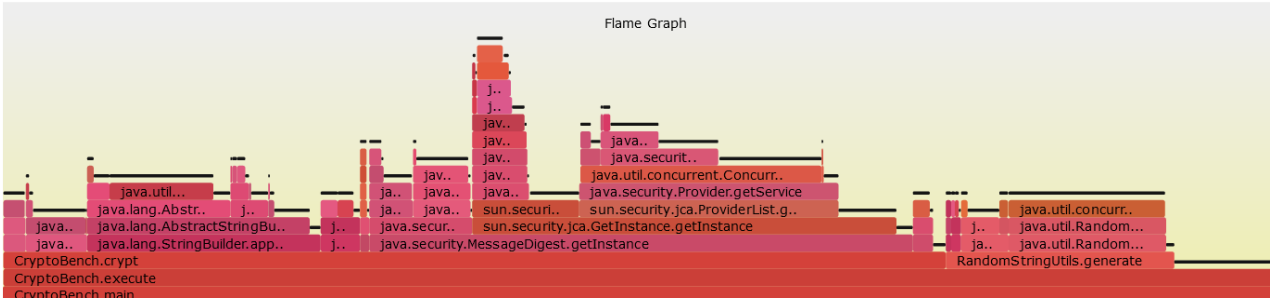
This affects both profilers, although JFR can still get a better picture thanks to virtual call sites decoding.

See the flame graphs for the example we used previously, but with the `-XX:+DebugNonSafepoints` option added.

Perf results



JFR results



As for JFR, the picture changes dramatically and provides more accurate symbol data, but this is not

the case for perf results.

9. Revealing JFR blind spots

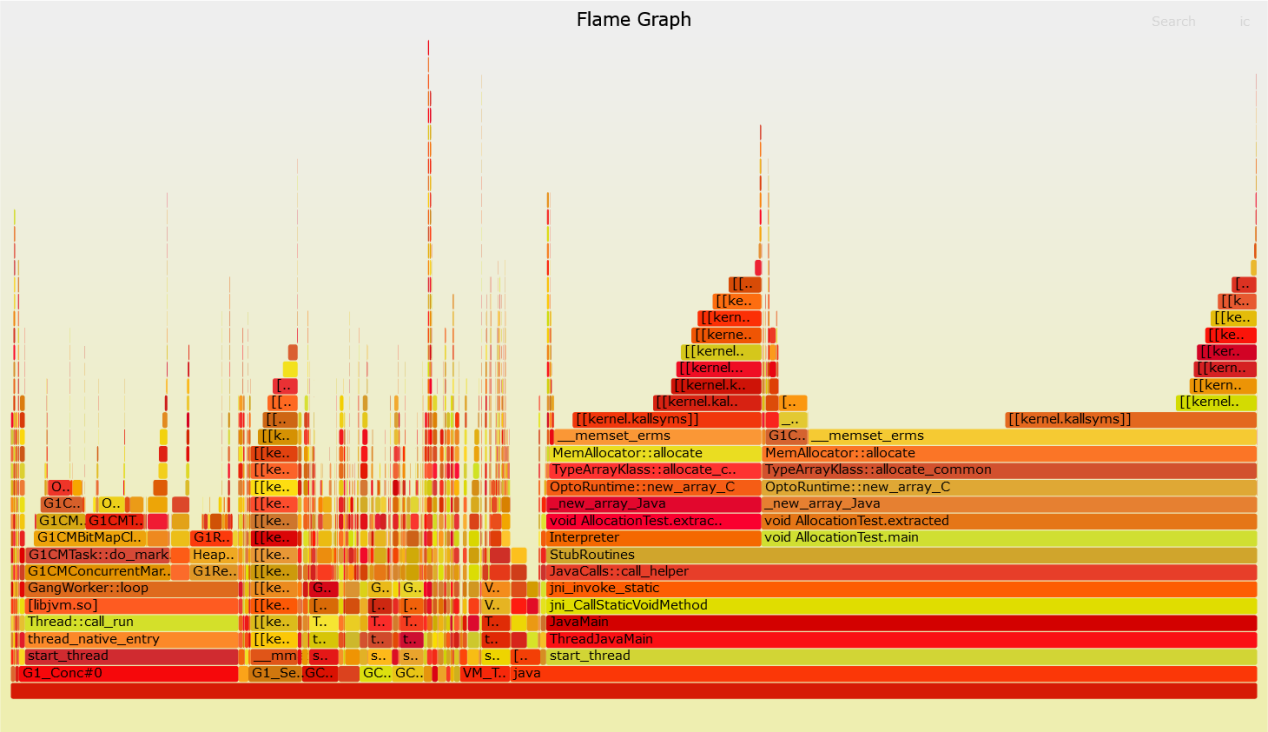
To further explore the topic of profiling inaccuracy, let's analyze another example.

Here is the code.

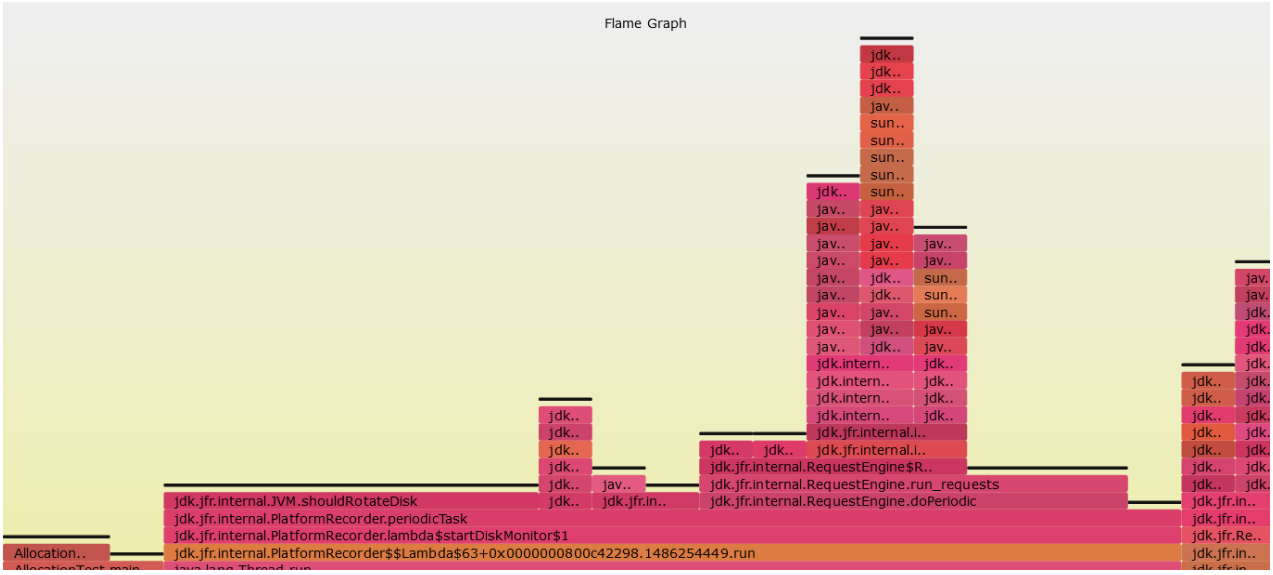
```
import java.util.Random;
public class AllocatioNTest {
    private static int totalSize = 0;
    public static void main(String[] args) {
        Random random = new Random();
        while (true) {
            extracted(random);
        }
    }

    private static void extracted(Random random) {
        int size = 8 * 1000 * 1000;
        int max = random.nextInt(10) + 1;
        int realsize = size * max;
        byte[] allocated = new byte[realsize];
        totalSize += allocated.length;
    }
}
```

Below are flame graphs produced by profiling time code with perf and JFR.

Perf results



JFR results



This example is one of the morbid cases for JFR. Perf shows us that a huge chunk of the main thread time is spent on memory allocation, which is an expected result for provided example code.

The allocation happens in large chunks, so the _new_array_Java routine is called directly without inlining. JFR can decode only stack traces where Java code is executed, either compiled or interpreted. In this example, the CPU is caught in JVM routines or even the kernel most of the time. JFR discards such samples, thus creating gaps in the sample population and dramatically skewing the final

distribution.

10. Considerations

OpenJDK 17 removed one more hassle of using perf with Java by introducing a simple way to generate debug symbol maps of JIT-produced code under Linux. While it was previously possible to achieve the same results by utilizing additional tools, with OpenJDK 17, the overall process became more reliable and convenient.

Perf is not the best tool for Java code profiling. Even with JIT symbol maps, method inlining and interpreted frames are too alien for perf to be handled accurately. And yet, perf is indispensable if you want to peek at the way your Java code interacts with native libraries or the kernel itself. It can also reveal the inner workings of JVM itself, for example GC or memory allocation, which most Java profilers cannot inspect.

As a general principle, we recommend using perf as a complementary tool to JFR or one of the other Java profilers:

- A Java profiler can give you an accurate Java call tree.
- Perf is more accurate for calculating a real on-stack percentage of frame, it has much higher sampling frequencies, and as such — higher accuracy.
- Perf can give additional insights on how the CPU is utilized in JVM, native libraries, and kernel. This data can prove to be very useful if your bottleneck is IO.

Liberica JDK
Using perf to monitor Java performance

bell//soft