JDK Flight Recorder
How to discover code hotspots

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

Hunting down code hotspots is probably the most common task for Java profilers. JDK Flight Recorder (JFR) and Mission Control (MC) are free and open source performance/profiling products available with Liberica JDK. They have a few powerful tools for code execution profiling.

Usually, you start a profiler when there is a performance problem, or you want to optimize code to meet specific performance goals. Performance is most commonly expressed as execution time (time needed to execute operation) or throughput (number of operations executed per time unit).

If you want to reduce the time spent on a request, it is obvious to focus on code that takes the longest to run. This is what we call "hot code" or "code hotspots." Profilers are optimal tools for identifying "code hotspots" and JFR + Mission Control can be used together for that purpose. In this document, we will overview the code profiling features they offer.

2. Starting profiling with Mission Control

We will work with Liberica JDK 11 and Mission Control 7.1. JDK Flight Recorder is integrated into Liberica JDK 11. The Liberica JDK binaries are available in Liberica JDK Download Center, and Mission Control binaries — In Mission Control Download Center.

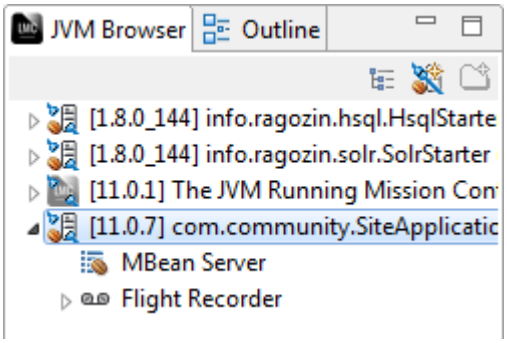
Flight Recorder support is also available with Liberica JDK 8u262 and later.

You can use Mission Control with either a locally running JVM or a remote one. In the latter case, you need a JMX port configured on a remote JVM.

You also need an application you will profile and some load to keep it busy.

Working with Local JVM

Locally running JVM processes are listed in the **JVM Browser** view. You need to identify the JVM you want to profile. Besides, you can either open the JMX console for that JVM or the control flight recorder.



Working with remote JVM

With OpenJDK, Flight Recorder is available remotely via JMX. You need a JMX socket configured on the JVM that you would like to profile.

There are two ways to make the JMX socket available:

- JMX can be enabled via JVM command-line arguments. You can find the required configuration options in the official documentation.
- Using the jcmd command, you can start the JMX socket on JVM with no up-front configuration and no restart necessary.

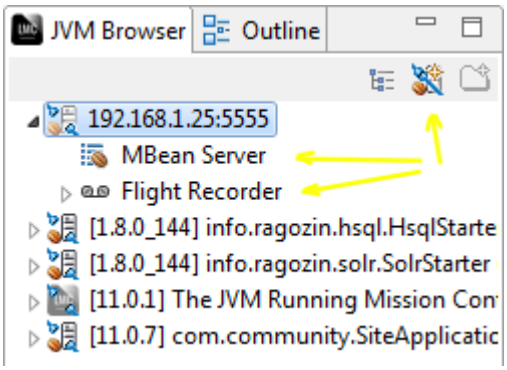
The following is an example of a command to start JMX with minimal configuration. You need to know the PID of a process and be logged under the same user as the target JVM.

```
jcmd PID ManagementAgent.start +
jmxremote.authenticate=false +
jmxremote.ssl=false +
jmxremote.port=5555
```

Once the command is executed, you can connect to your JVM via port 5555 using the instruction in the following pages.

If the JVM you want to profile is behind NAT/firewall (e.g., it is running within Kubernetes), you may need to use port forwarding and additional configuration tweaks to make JMX work.

Having configured the JMX socket, you need to add a remote JVM to it in the **JVM Browser** view. Click the button on the toolbar of that view. You will be prompted to enter JMX connection details. Once you finish JMX configuration, nodes for your remote JVM will appear in the view. Now, you can see either a JMX Console or control flight recorder.



3. Before starting the profiler

Before starting profiling and focusing on code, it is recommended to get an overall CPU usage picture on the host where your JVM is running.

- Is the CPU overutilized?
- Do other processes compete for CPU resources?
- How much CPU is the JVM process consuming?
- Which Java threads account for the highest CPU usage?

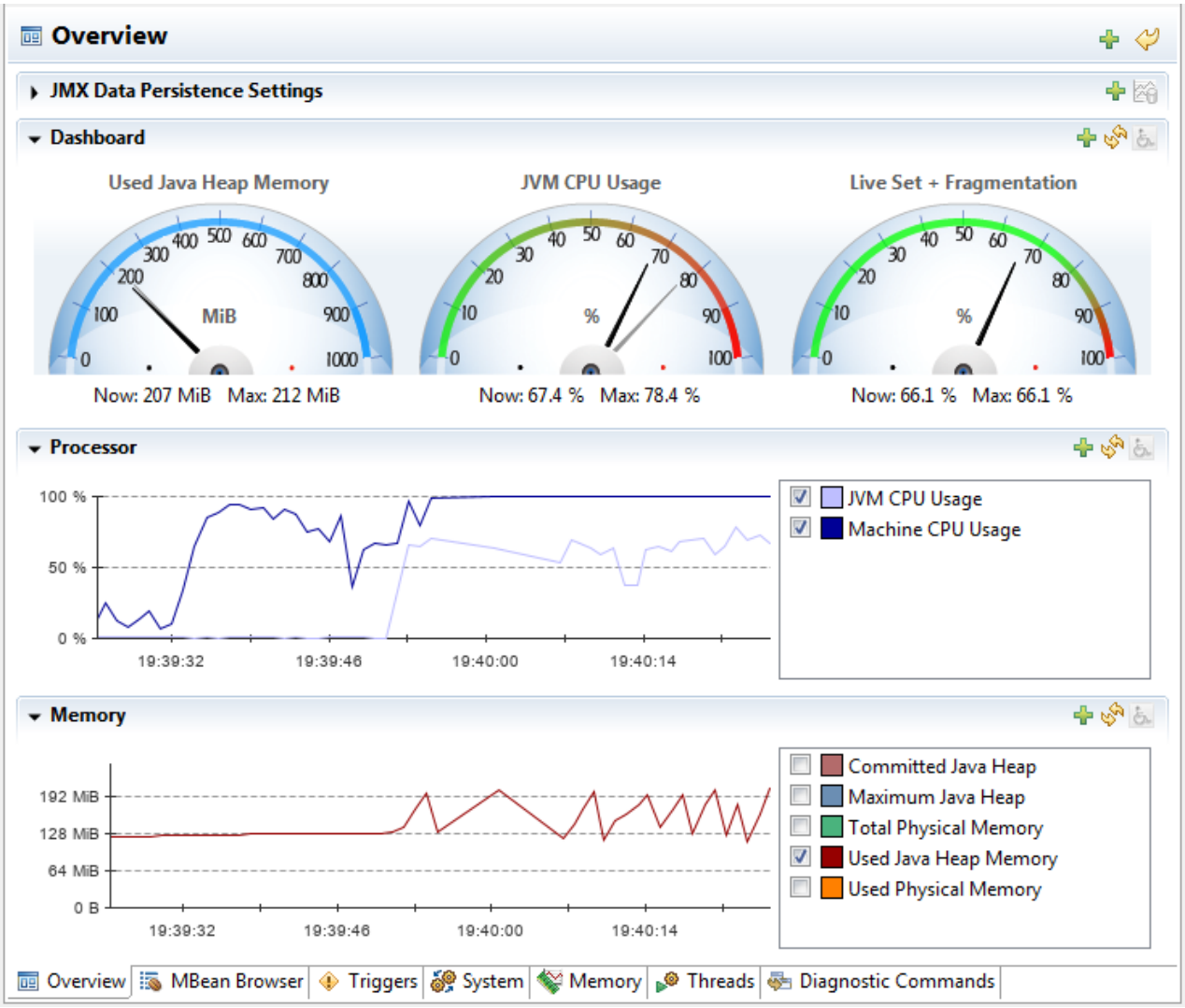
Answers to these questions will help you to choose the right reports in Mission Control later.

These questions can be resolved with standard system tools such as top and pidstats, but it is also possible to use the JMX console in Mission Control.

4. Mission Control JMX Console and Thread monitor

Before starting Flight Recorder, we recommend looking at the JMX console built into Mission Control.

With the JMX console, you can monitor the system and process CPU usage. These metrics are available on the **Overview** tab.



You can observe CPU usage per thread as well, which is useful for code execution profiling. Open the **Threads** tab and check the **CPU Profiling** box above the table of threads. After this, you will see values in the **Total CPU Usage** column.

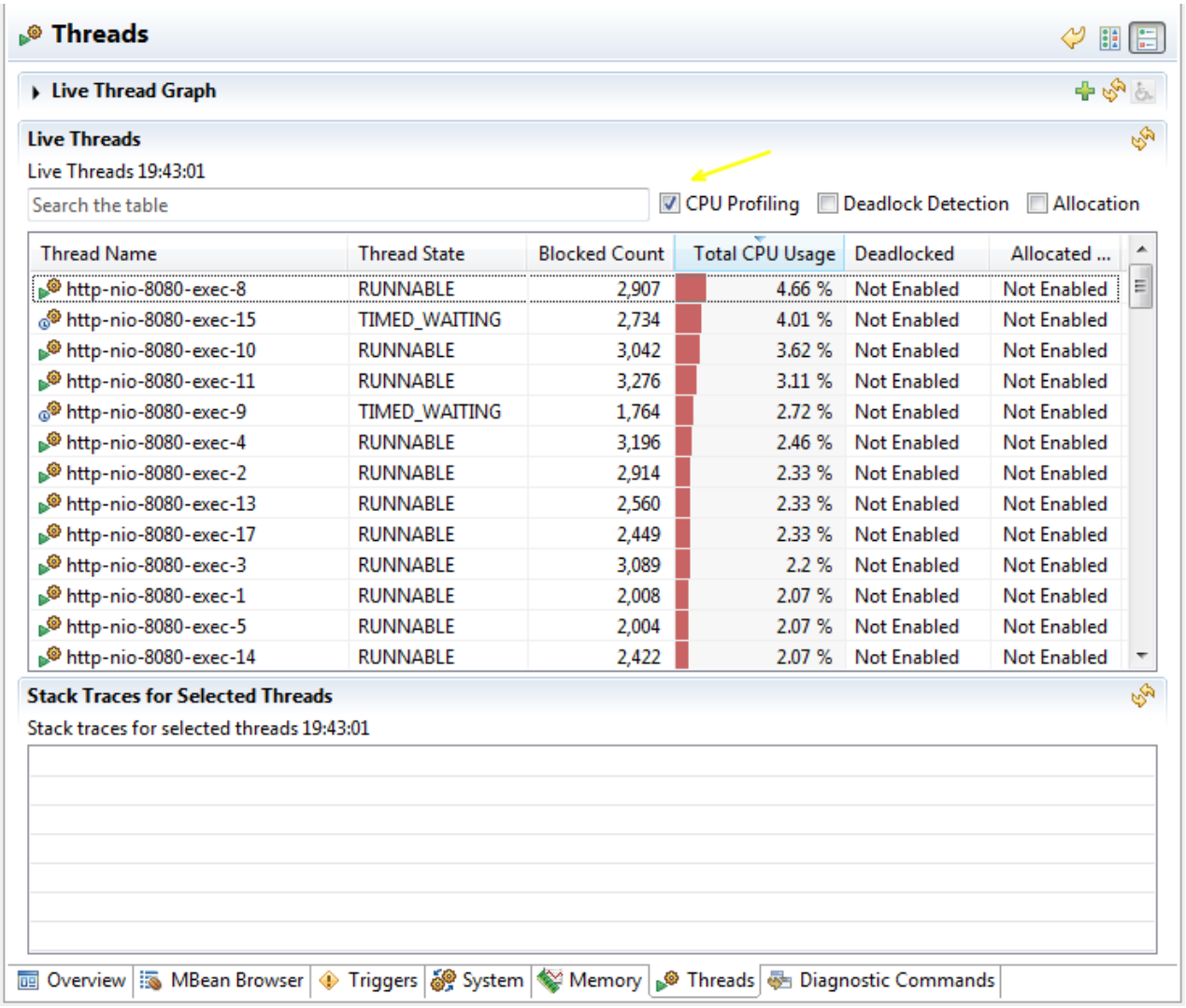
> **Note:**
>
> Mission Control shows the percent of CPU usage from all cores available on the host. A single thread cannot consume more than a single core, therefore 100% / N (where N is the number of cores) is the max value you can see in that table.

What if JMX is not an option?

What if you have console access to JVM, but no way to connect to it via JMX?

You can still record JFR files using jcmd, then copy them to your machine and open them in Mission Control.

The JMX Console provided by Mission Control won't be useful in this case, but you can use console tools such as pidstat or sjk to monitor per-thread CPU usage.


Threads
Live Thread Graph
Live Threads
Live Threads 19:43:01
Search the table
CPU Profiling
Deadlock Detection
Allocation
Thread Name | Thread State | Blocked Count | Total CPU Usage | Deadlocked | Allocated ...
http-nio-8080-exec-8 | RUNNABLE | 2,907 | 4.66 % | Not Enabled | Not Enabled
http-nio-8080-exec-15 | TIMED_WAITING | 2,734 | 4.01 % | Not Enabled | Not Enabled
http-nio-8080-exec-10 | RUNNABLE | 3,042 | 3.62 % | Not Enabled | Not Enabled
http-nio-8080-exec-11 | RUNNABLE | 3,276 | 3.11 % | Not Enabled | Not Enabled
http-nio-8080-exec-9 | TIMED_WAITING | 1,764 | 2.72 % | Not Enabled | Not Enabled
http-nio-8080-exec-4 | RUNNABLE | 3,196 | 2.46 % | Not Enabled | Not Enabled
http-nio-8080-exec-2 | RUNNABLE | 2,914 | 2.33 % | Not Enabled | Not Enabled
http-nio-8080-exec-13 | RUNNABLE | 2,560 | 2.33 % | Not Enabled | Not Enabled
http-nio-8080-exec-17 | RUNNABLE | 2,449 | 2.33 % | Not Enabled | Not Enabled
http-nio-8080-exec-3 | RUNNABLE | 3,089 | 2.2 % | Not Enabled | Not Enabled
http-nio-8080-exec-1 | RUNNABLE | 2,008 | 2.07 % | Not Enabled | Not Enabled
http-nio-8080-exec-5 | RUNNABLE | 2,004 | 2.07 % | Not Enabled | Not Enabled
http-nio-8080-exec-14 | RUNNABLE | 2,422 | 2.07 % | Not Enabled | Not Enabled
Stack Traces for Selected Threads
Stack traces for selected threads 19:43:01
Overview
MBean Browser
Triggers
System
Memory
Threads
Diagnostic Commands

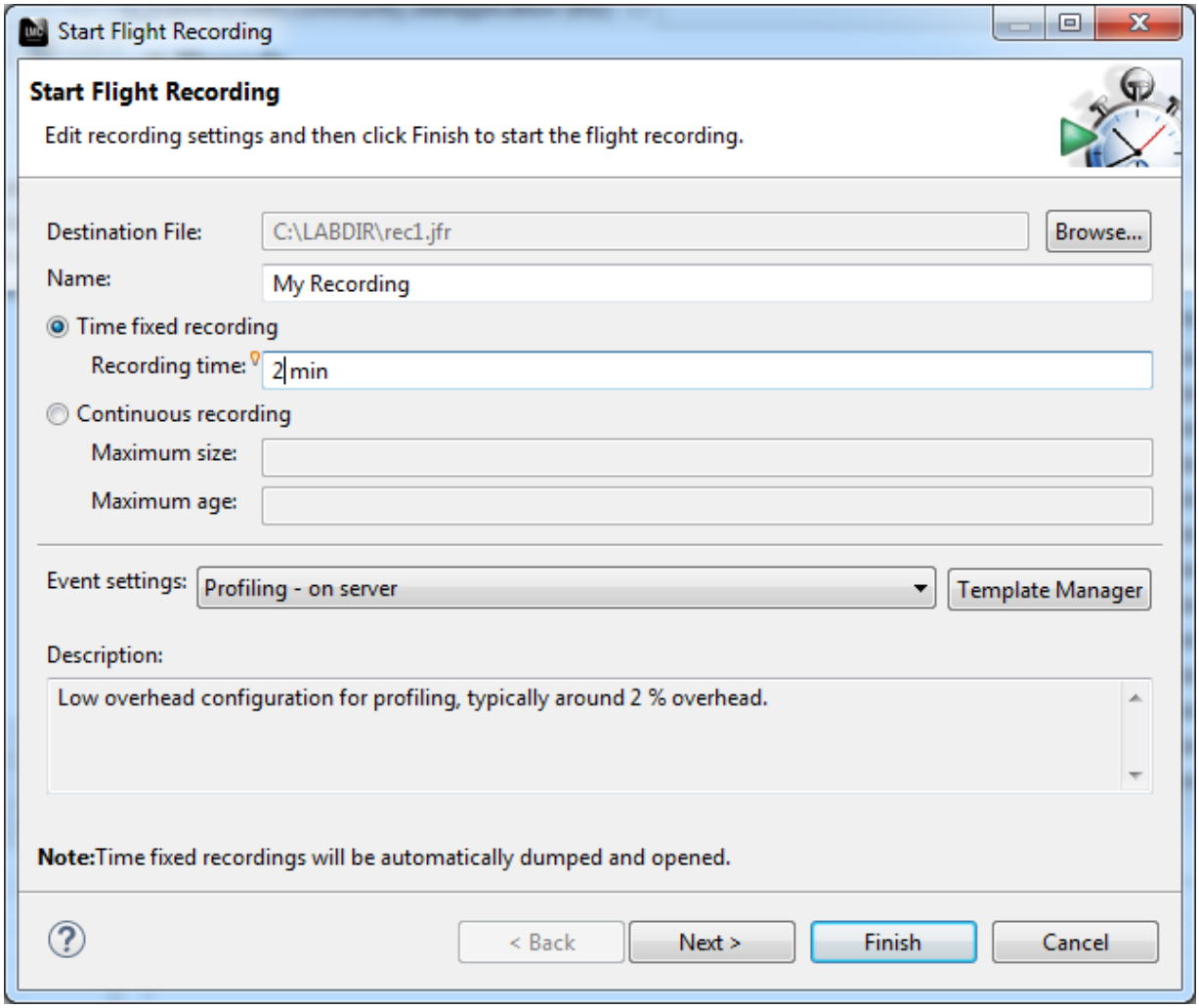
5. Starting JDK Flight Recorder session

Let’s start the Flight Recorder.

1. Right-click the **Flight Recorder** node in the tree on the **JVM Browser** view and select **Start Flight Recording**.

   You will see a dialog box with options to start the Flight Recorder session. The options on the following screenshot instruct to start Flight Recorder for 2 minutes, with **Profiling - on server** event settings.

   Press **Next**.

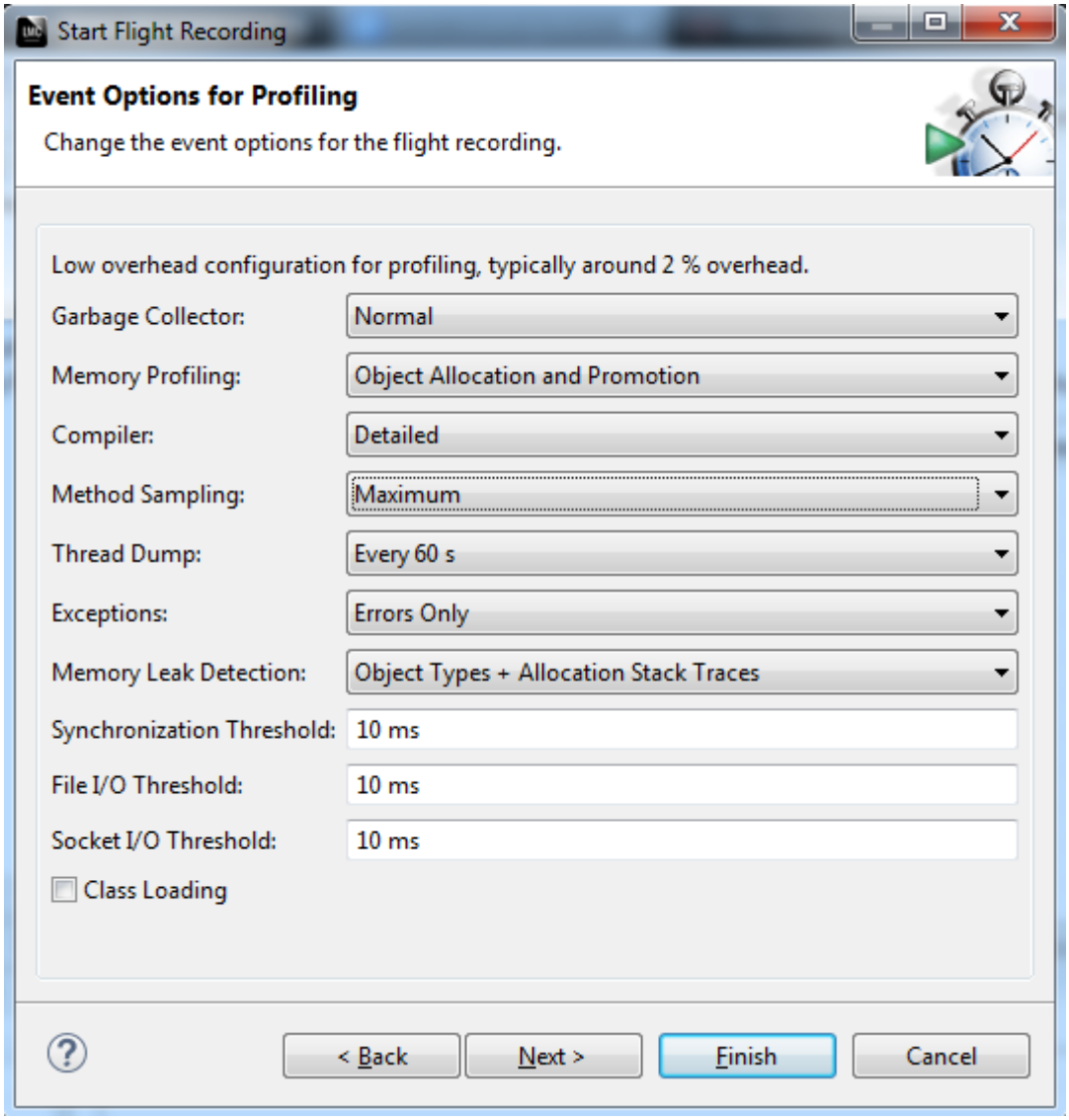


2. In the **Event Options for Profiling** dialog box tweak the most common Flight Recorder options for the next recording session. Some of the following options affect the reports explained later in this document.

    - **Method Sampling** — this option controls the frequency of stack trace sampling. **Maximum** means 100 samples per second, which is a good trade-off between performance and data quality.

    - **Exceptions** — you can choose whenever to record all `Throwables` or just `Errors`. Recording all `Throwables` may be expensive but useful to pinpoint excessive exception usage in an application.

    - **Synchronization Threshold**, **File I/O Threshold**, **Socket I/O Threshold** — for detailed analysis of "cold" hotspots, we recommend to set these thresholds to zero. They may increase overhead significantly, so enable them only if you need this information.



3. Press **Finish** to start the Flight Recording session. You can click **Next** to view and edit the Flight Recorder configuration at a lower level.

After starting the Flight Recording session, Mission Control will wait for the specified time, then stop Flight Recorder, dump the data file, open it, and present several reports.

Making JFR recording without Mission Control

JFR recording can be produced without Mission Control, too. You can use jcmd to create JFR files, then copy them to the desktop and open them in Mission Control.

Here is an instruction to capture JFR recording. You need to know the PID of the target JVM and

execute jcmd under the same user account.

1. Start JFR recording with the command below:

   ```
   jcmd JFR.start settings=profile
   ```

   The result of this command should be a message similar to the following. You need to remember the number of the recording for later.

   ```
   Started recording 1. No limit (duration/maxsize/maxage) in use.
   ```

2. Use `JFR.dump recording=1 filename=FILEPATH` to copy recording data to file.
3. Wait a little for data to be collected.
4. Use the command suggested in step 1 to dump the JFR data.

   ```
   jcmd JFR.dump recording= filename=
   ```

   This action creates a JFR file.

5. Stop the recording session with the following command.

   ```
   jcmd JFR.stop recording=
   ```

Now that you have a JFR file, you can open it in Mission Control later.

6. Method profiling report

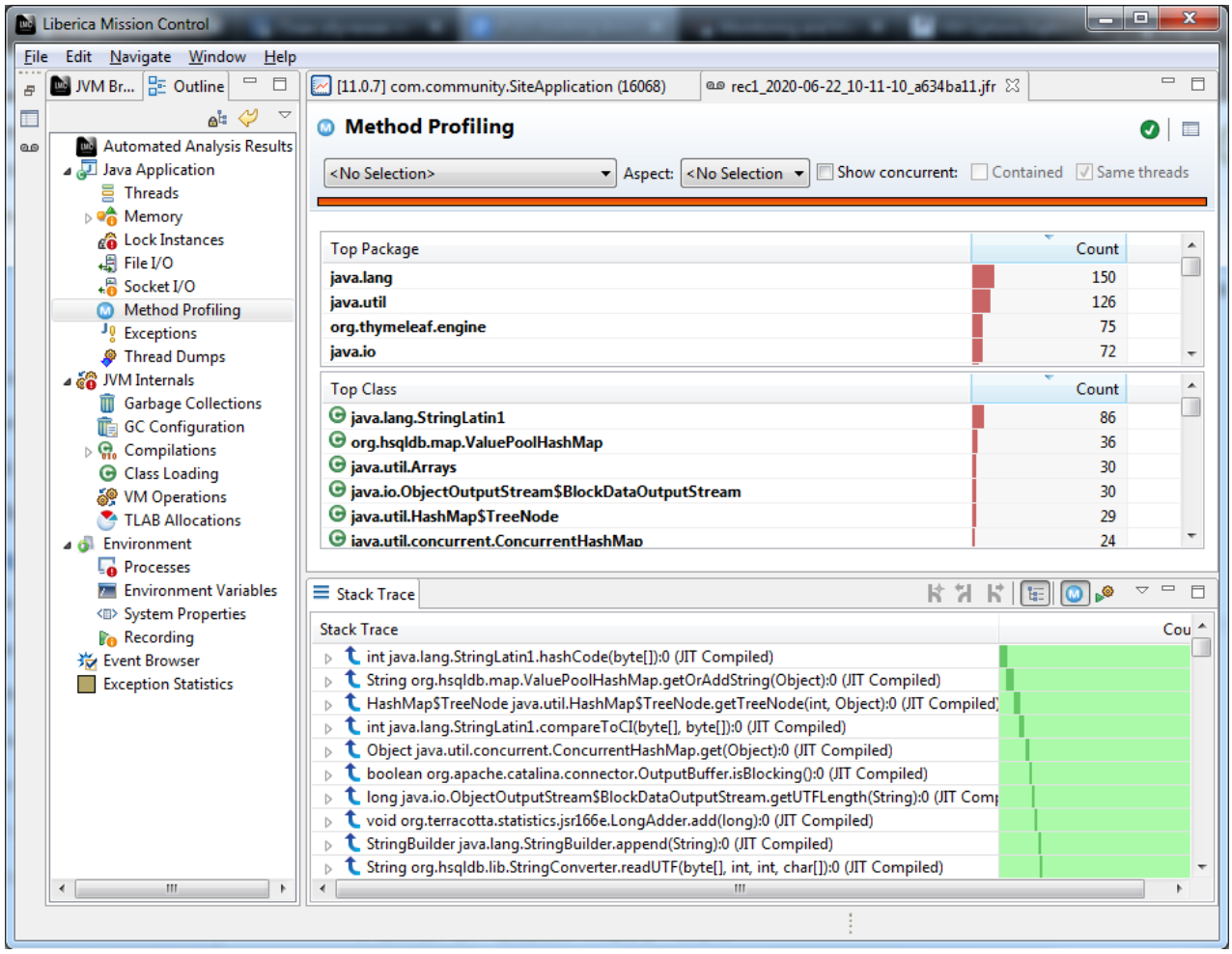
The Flight Recorder collects various types of events and Mission Control builds a number of reports from that data.

We want to identify areas of code that have contributed most to the execution time of our request. The usual profiling method for such tasks is a stack trace sampling.

During execution at regular intervals, Flight Recorder records a trace for each thread. Taking samples from a single thread and observing that method X is present on a stack in 50% of the sample, you can assume that the sum of all method X invocation time is 50% of profiling session time.

While the idea is simple, actually you have to deal with multiple threads and non-ideal sample distribution. Still, this approach remains extremely useful.

Let's open the **Method Profiling** report.



The report itself is pretty basic, and if you used a different Java profiler before, it feels lacking details, which are hidden in a **Stack Trace** view (panel below).

Stack Trace view

Stack view allows visualizing a set of stack traces as a tree.

How do stack traces become a tree?

We know how Java stack traces look like (be it exception stack trace or stack trace from thread dump). Data produced by sampling profiling is just a number of stack traces.

Here are steps on how this data is transformed into a tree.

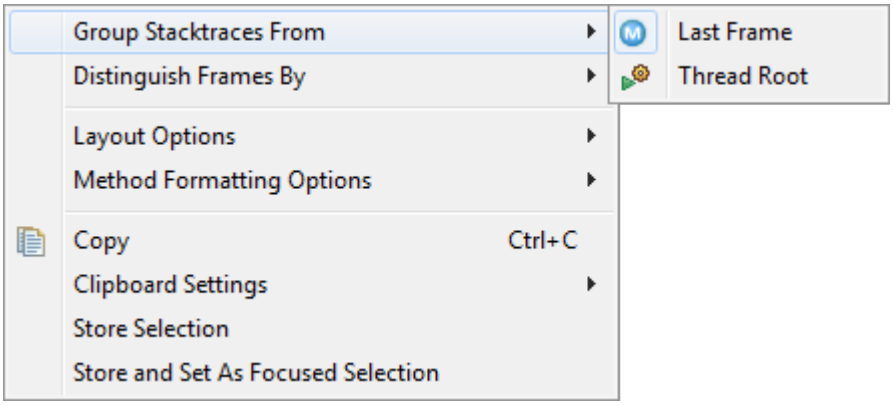
1. Convert each stack trace into a string by concatenating frames.
2. Now that you have a multiset of strings, calculate each one's occurrences and compile a histogram.
3. You've got a histogram, a table with two columns "trace" and "count." If you sort this table and group by common prefix, it will become a tree.

In the context menu, you have the following **Stack Trace** view customization options.

Group Stacktraces From

This option controls how frames are concatenated in step 1 of the algorithm above and what root nodes will be in the tree.

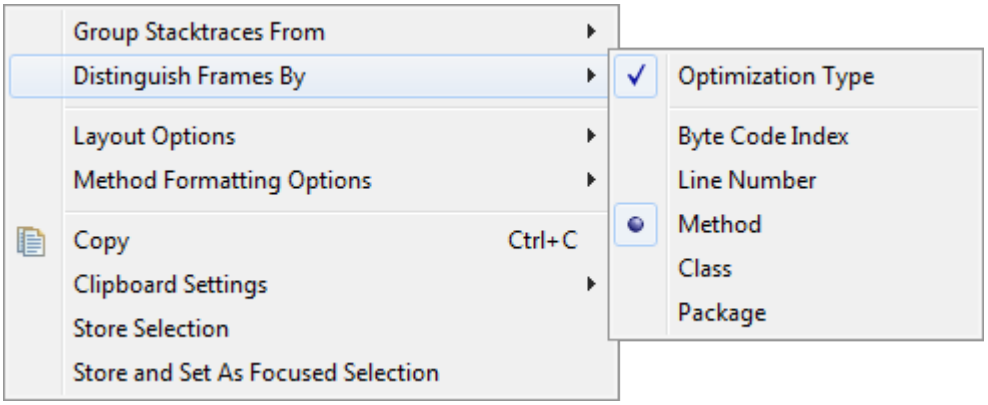
- Last Frame — aka the "hot methods" mode.
- Thread Root — aka the "call tree" mode.



Distinguish Frames By

This option controls which information is stripped from the frame description in step 1.

Take note of the **Line Number** option. Line Numbers add clutter to the tree, but sometimes you want them to be visible to get the exact reference at source code.

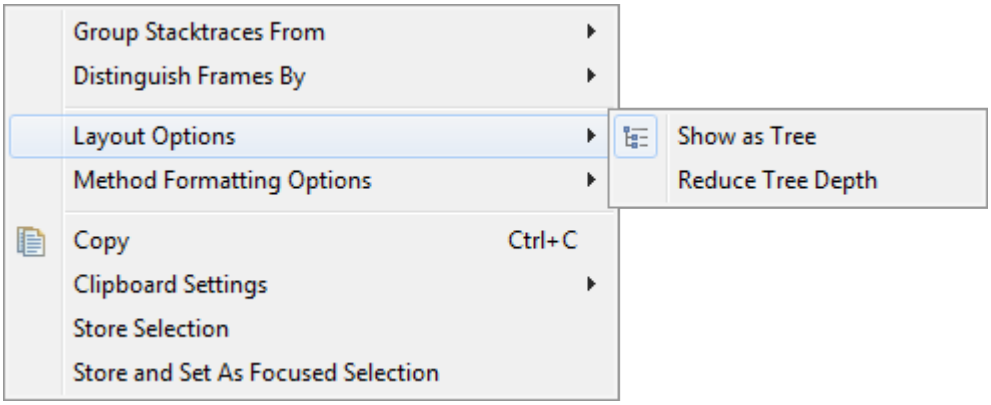


Layout Options

The default tree is horizontally compressed, which is good but can confuse. Here, you can switch back to a classic tree presentation.

Stack view will work for all reports based on events that incorporate stack trace (including method profiling).

Mission Control can even mix different types of events in the same stack view, but be aware that mixing events sampled by different rules, such as method profiling and IO traces, does not make much sense.



Why CPU usage is important

If we look at Java thread at any given time, it could be

1. running on CPU in Java code;
2. ready to run in Java code but waiting in OS queue to get CPU;
3. running on CPU in native (non-Java) code;
4. ready to run in native code but waiting in OS queue to get CPU;
5. waiting to do something in native non-Java code;
6. waiting/blocked at the Java level.

JFR samples only threads in categories 1 or 2. There is also separate sampling for 3, 4 and 5.

So, if your server is "starving" on CPU, JFR will show you a particular picture, but it will be skewed due to mixing 1 and 2 categories. The sampling picture will not accurately reflect real code computation cost in this situation.

If the server is not CPU "starved" and threads you are profiling are actively consuming CPU, method sampling is the right tool for you to start with.

If the server is not CPU "starved" and threads you are profiling are low on CPU, you are likely to have a "cold" hotspot, falling into categories 5 and 6.

Category 5 code can be analyzed with the "Native method sampling" event.

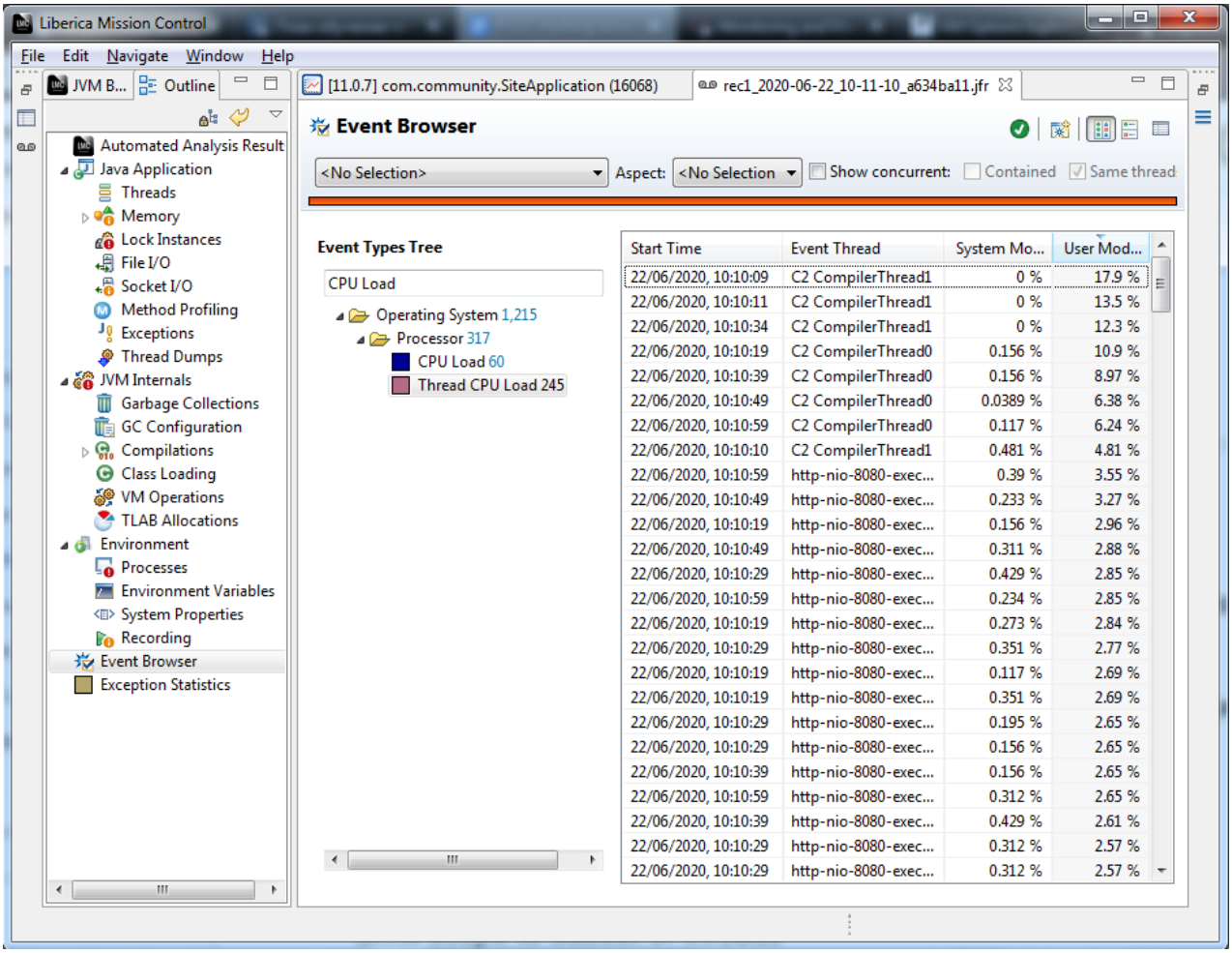
Category 6 is either contention or IO, examined with separate reports.

So by looking at the CPU, you get an idea which report will be most useful.

Retrospective thread CPU usage in Mission Control

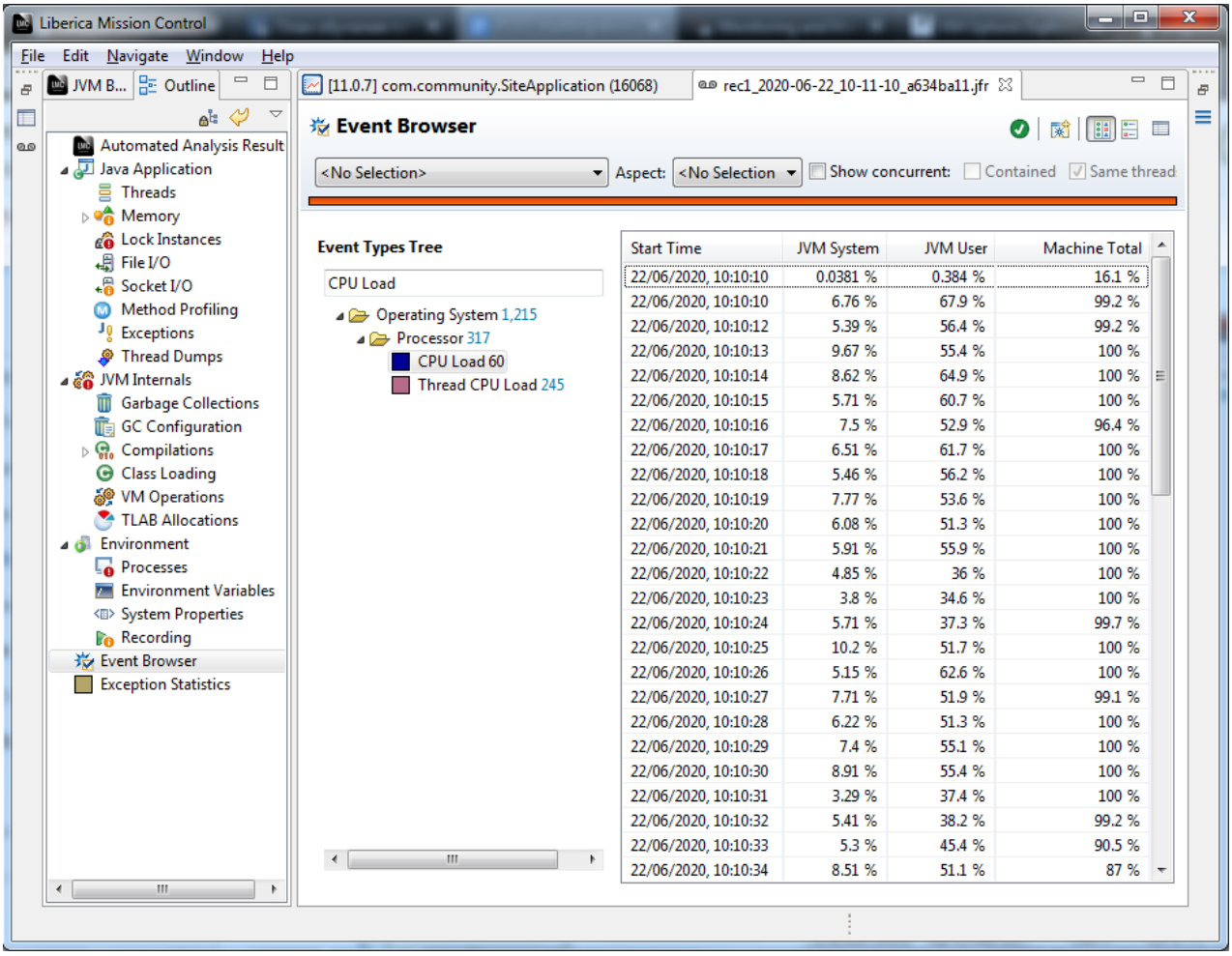
What if you have a JFR recording and have no idea of CPU usage by threads at runtime? That data is recorded in a JFR file as well.

Open the **Event Browser** and find the **Thread CPU Load** event under the **Operating System** > **Processor** category as in the following image.



CPU usage for each thread is recorded with 10-second intervals. Remember that the percentage is taken from the total number of cores. In the screenshot, 25% means 100% of a single CPU core.

JVM and host OS CPU usages are also recorded. Click the **CPU Load** event type in the same category.



Limit scope to a subset of threads

You can now find CPU hungry threads, but the **Method Profiling** report shows all threads. You can filter by a subset of threads if necessary.

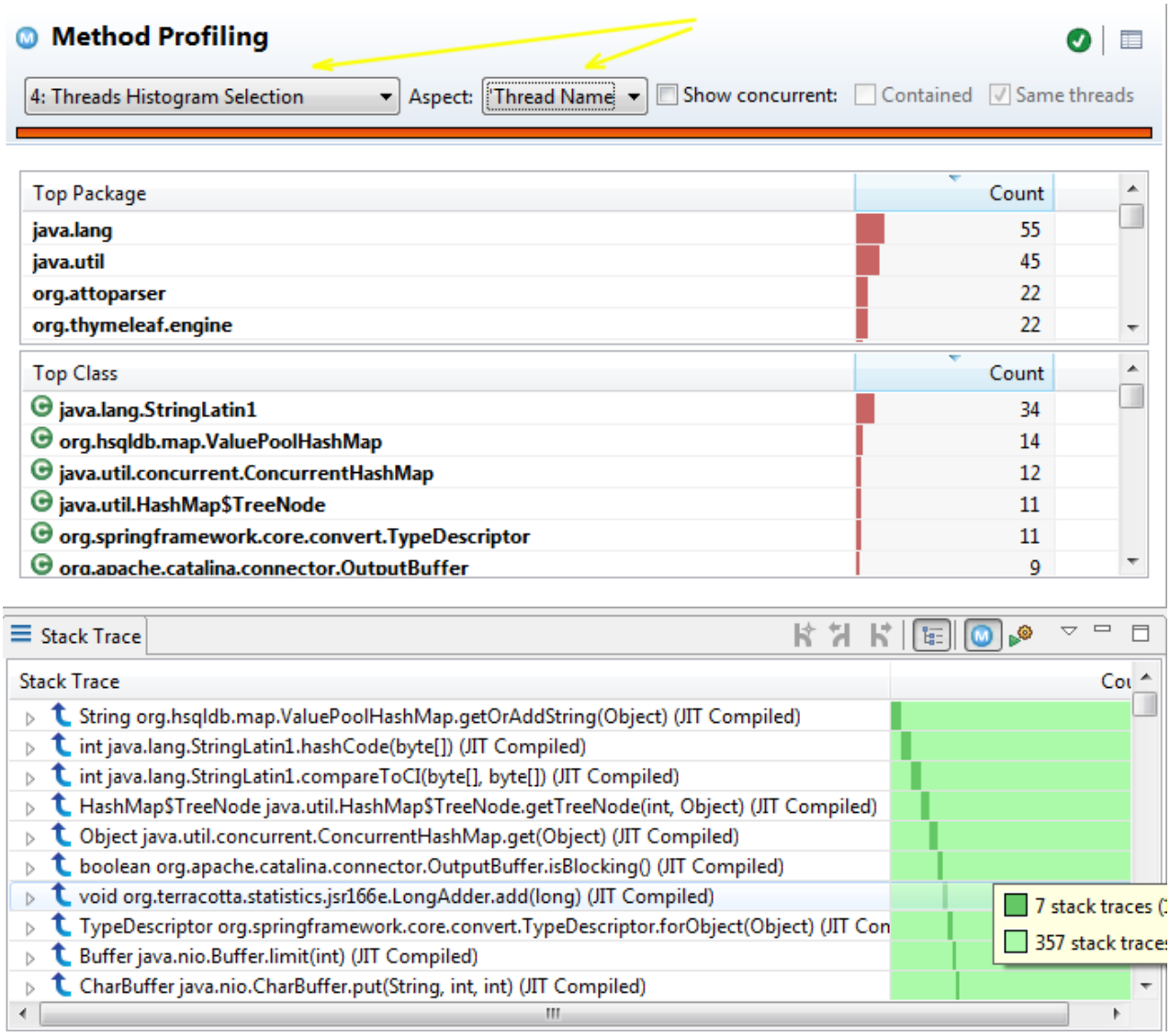
Mission Control has very flexible but non-obvious filtering features.

To filter by a subset of threads in the **Method Profiling** report, perform the following steps.

- Open the **Threads** view.
- Select one or more threads in a list of threads on the left.
- Right click and choose **Store Selection** on the context menu.
- Go back to **Method Profiling**.

- In the **Focus** list, select **Threads Histogram Selection** and in the **Aspect**, select **Thread Name is**.

Now, the report includes events only for threads you have chosen.



Any filter is shared across all reports, so do not forget to reset it when switching reports.

The "Stack Trace" view of the "Threads" report

While completing the steps above, you may notice that the **Stack Trace** view is available in **Threads**. Moreover, it was reacting to selection in the thread list.

How is the **Stack Trace** on the **Threads** report different from the **Method Profiling** report?

Each report has a scope: a set of events (usually certain types of events) used to calculate the report. When you work with report UI, a focused subset of events in scope is maintained behind the scene.

A filter can narrow the scope, and this is what we did in the previous section.

Any event may include the stack trace, and many do. The **Stack Trace** view visualizes all stack traces of the focused set of events in an active report.

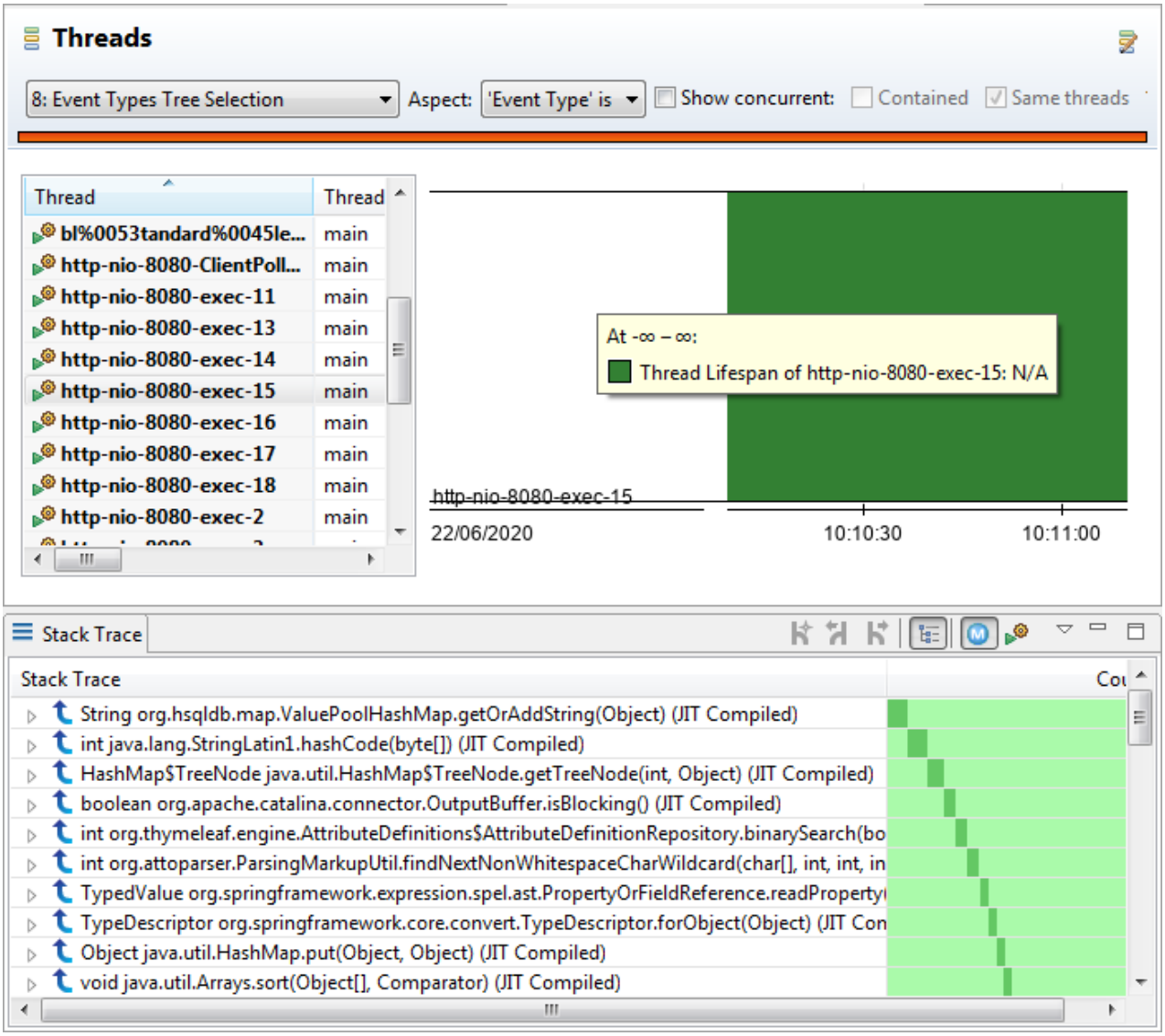
In **Method Profiling**, only method profiling samples are in scope.

The **Threads** report includes almost every event bound to some thread (both sampled and non-sampled). So, while technically they can be aggregated together, it makes little sense.

Nevertheless, you can use filters to fix this to make the **Threads** report show the same picture as **Method Profiling**.

1. Open the **Event Browser**.
2. Right-click the **Method Profiling Sample** in the event types tree and choose **Store Selection** in the context menu.
3. Go back to the **Threads** report.
4. In the **Focus** list, select source **Event Types Tree Selection** and aspect **Event Type is Method Profiling Sample**.

Now, your **Stack Trace** tab includes only method profiling data, and you can change selected threads without switching reports.



Caveats of JFR method profiling

The way JFR takes stack traces during profiling is precise. The majority of Java profilers use a JVM wide thread dump to get traces of all threads at once.

A thread dump is a Stop-the-World operation, and doing it 100 times per second can be expensive. JFR manages to avoid Stop-the-World. Instead of stopping all threads at once, JFR stops them individually (using OS facilities) and captures stack traces from individual threads.

Entering/leaving Stop-the-World state involves sophisticated protocol between application and system threads in JVM, and JFR skips this overhead altogether.

Yet, this optimization introduces certain nuances of data captured by JFR. Only stack traces ending up in Java code (not the native code) are recorded and visible in the **Method Profiling** report. Thus, unless threads you are looking at the active CPU consumers, the picture in **Method Profiling** will be skewed.

Besides the "Method Profiling Sample," there is the "Method Profiling Sample Native" event type that captures threads in states 4, 5 and 6 (see the list mentioned earlier). However, this event type is available only in the **Event Browser** not in the **Method Profiling** or the **Threads** reports. The good news is that the **Stack Trace** view also works in the **Event Browser** report.

Another important caveat for all types of Java sampling profilers is the "safepoint bias." In short, due to various effects of JVM runtime and JIT compiler, profilers can incorrectly identify the exact hot method or code line. In some edge cases, profilers can be very inaccurate.

Technically, JFR avoids the "safepoint" part of the problem, though it is still biased due to the way Java JIT compiler works.

While the edge case of the "safepoint bias" effect could make you lose your trust in the profiler, it is rarely a problem in actuality. A profiler’s job is to narrow the scope of search; it does not need to be 100% accurate to be useful. But do not trust it blindly either.

7. "Cold" code hotspots

A hotspot is a portion of code (e.g., method or line of the source file) responsible for a considerable part of execution time during request processing (or other kinds of workload) related to other code involved.

But how was that time spent? Was that code burning CPU cycles or sitting cold off the CPU waiting for something?

"Cold" hotspot is a kind of code that consumes time, but does not consume CPU resources. Most typical kinds of "cold" hotspots are

- blocking IO calls;
- contention points in multi-threaded applications;
- waiting for completions of async tasks.

The method sampling approach used by JFR is not suitable for identifying "cold" hotspots (it can see only the "CPU hot" code). There are separate reports for blocking IO and contention.

How to find out that you are dealing with a "cold" hot spot?

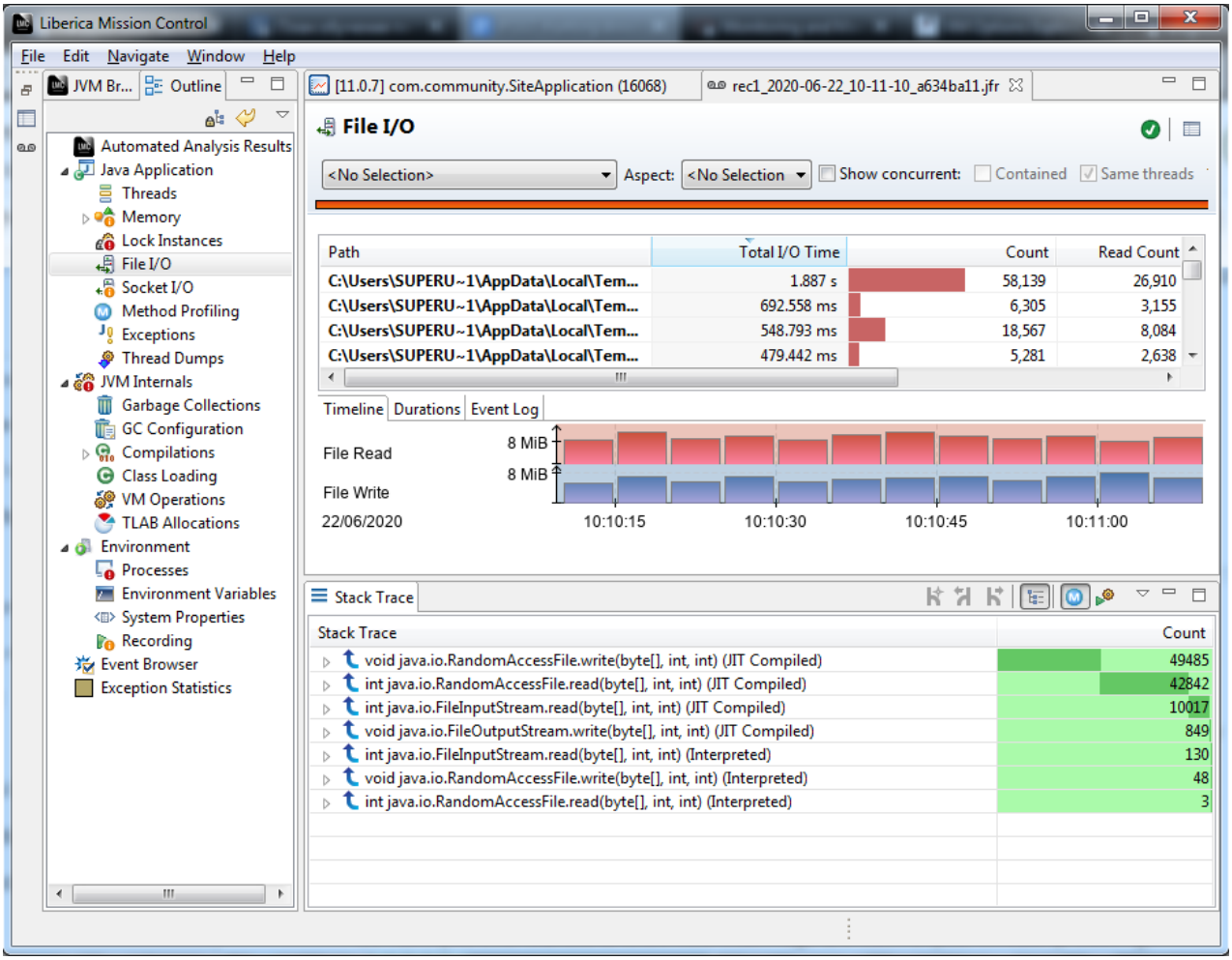
Low thread CPU usage for a thread supposed to process requests is a symptom that should trigger your attention.

"File I/O" report

This report is composed of "File Read" and "File Write" events, capturing blocking file IO operations. Since events include file names, this report shows at the file level how many bytes were read/written to each file and how much time it has taken.

An important caveat here is to consider the JFR recording session’s threshold. Only operations exceeding it will be visible here.

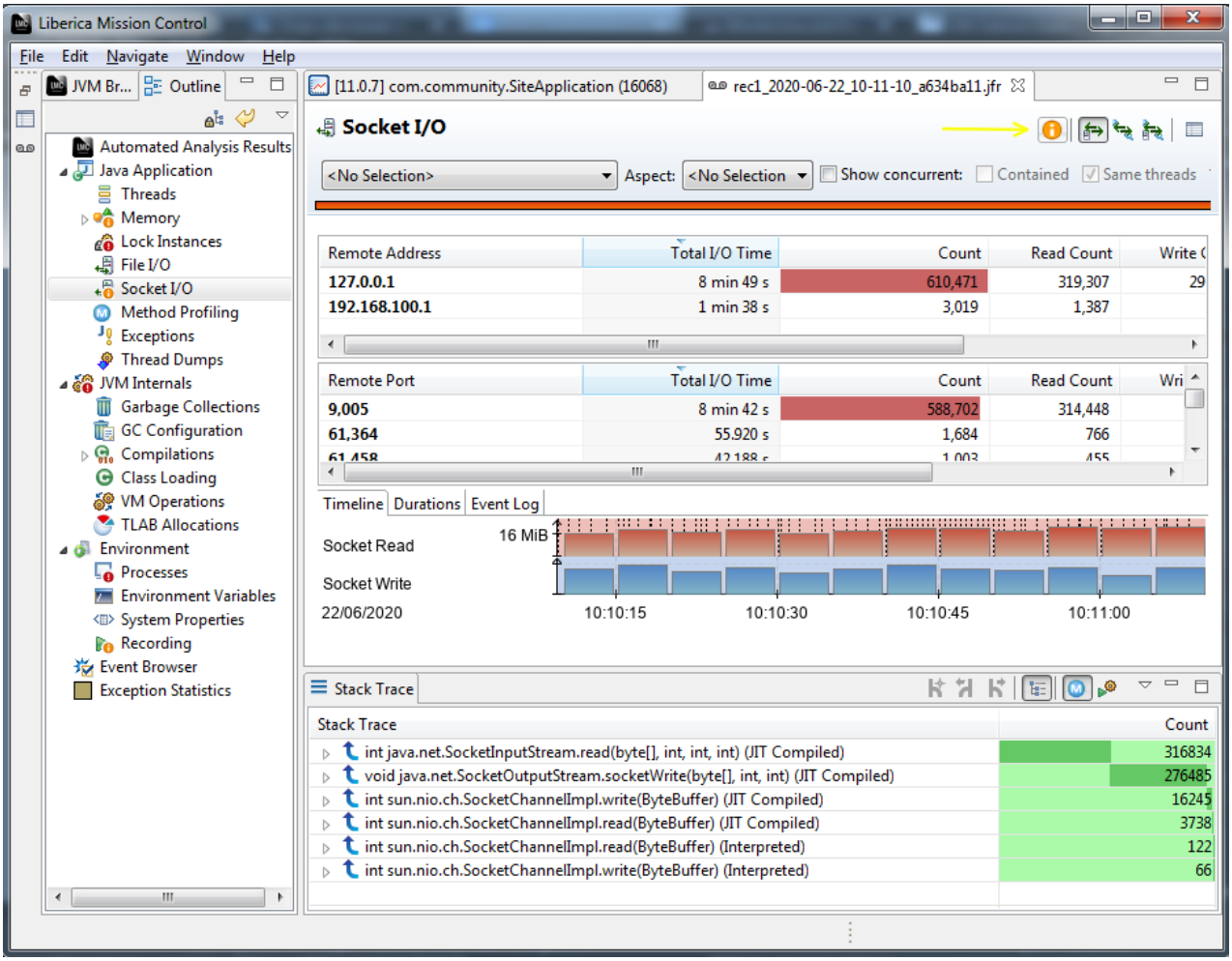
The report is relatively simple but can be quite powerful if combined with the filtering capabilities of Mission Control.



You can filter by focus and aspect. The **Stack Trace** view is also available here.

"Socket I/O" report

The "Socket I/O" report is quite similar to File I/O, but aggregation is shown by remote address and port.



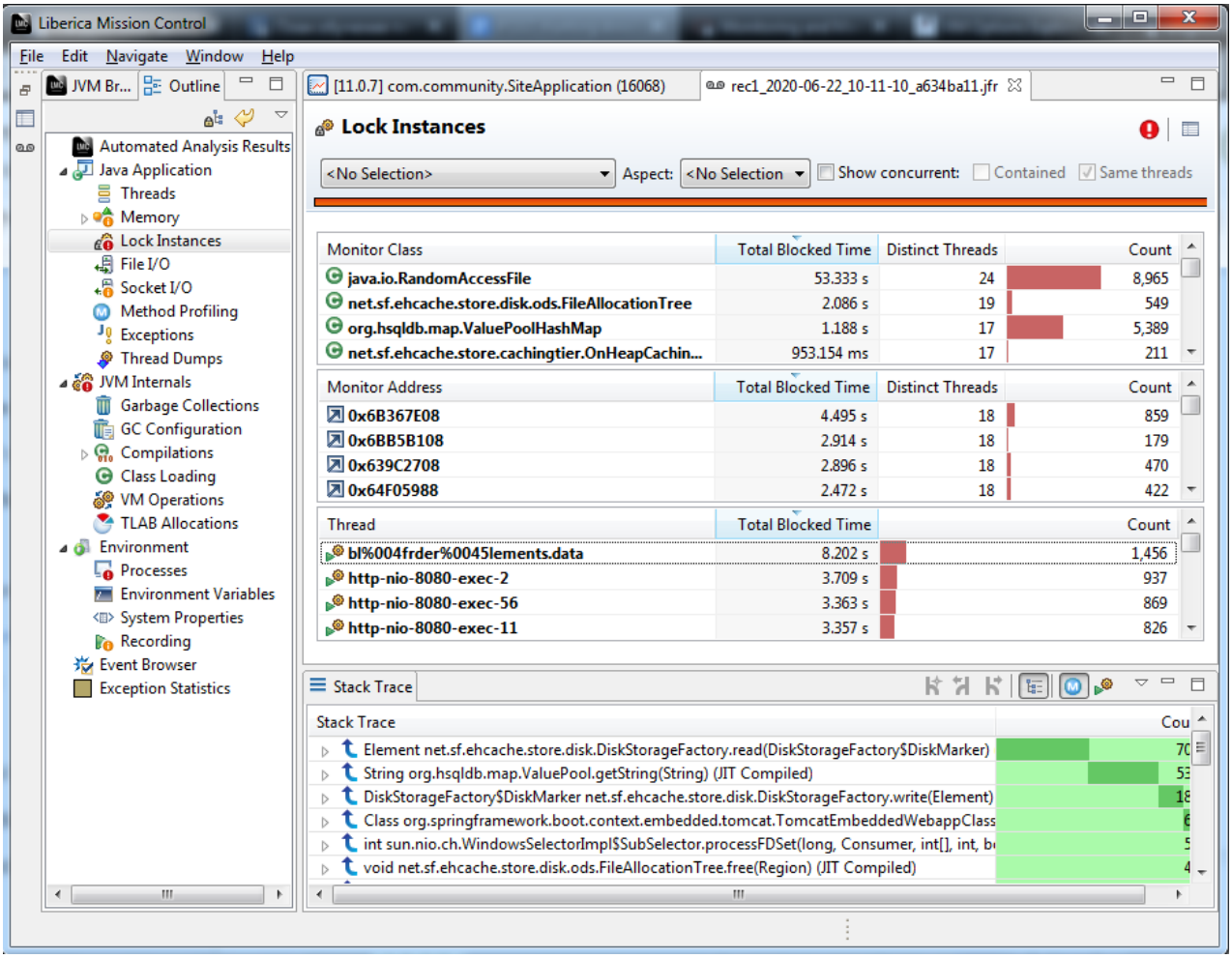
The taskbar lists three aggregation options available.

- By Host;
- By Port;
- By Host and Port.

"Lock Instances" report

Lock Instances reports are based on the "Java Monitor Blocked" event type. These events are produced when a Java thread is blocking, trying to acquire the semaphore.

This report is only useful with synchronized keyword-based thread coordination.

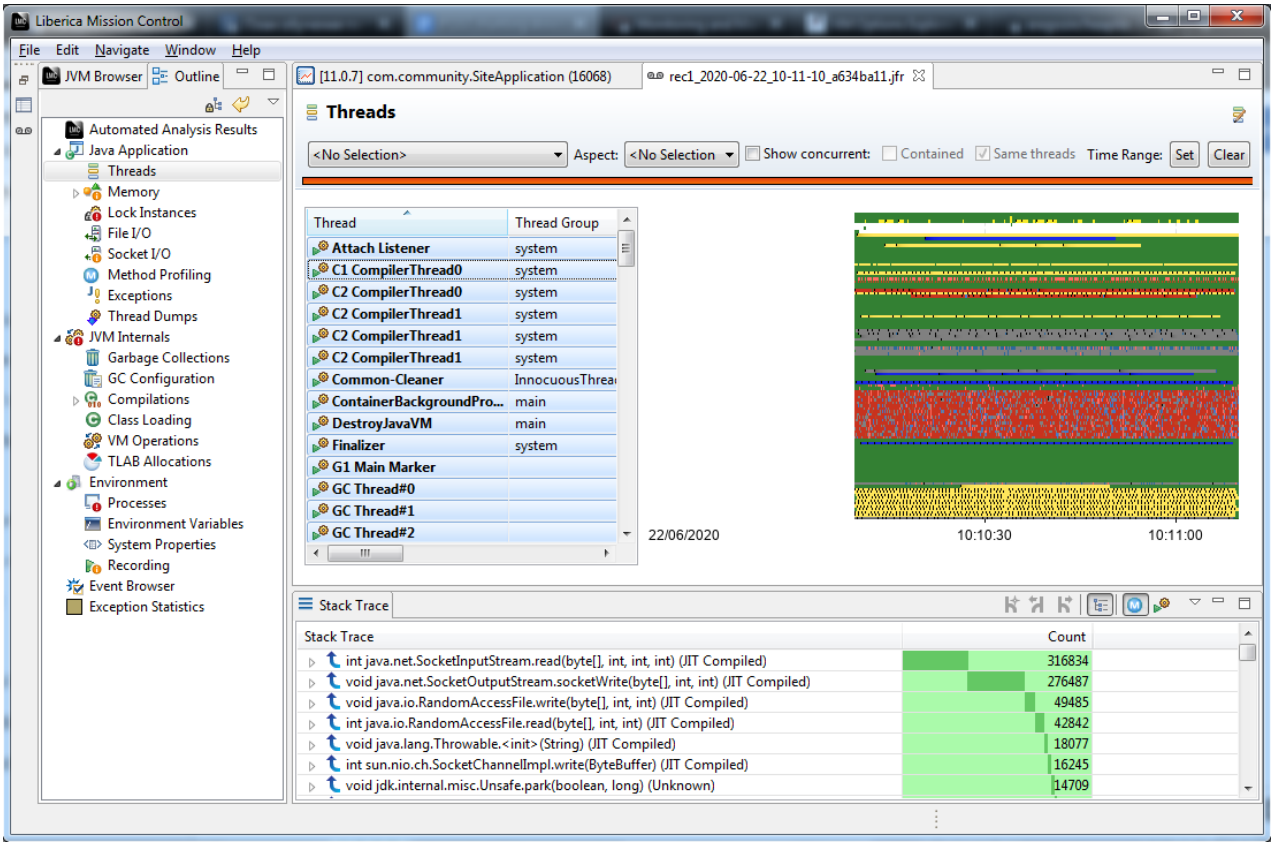


What about java.util.concurrent-based contention?

JFR also has events related to "new style" synchronization "Java Thread Park," but there are no dedicated reports in Mission Control yet.

"Threads" report

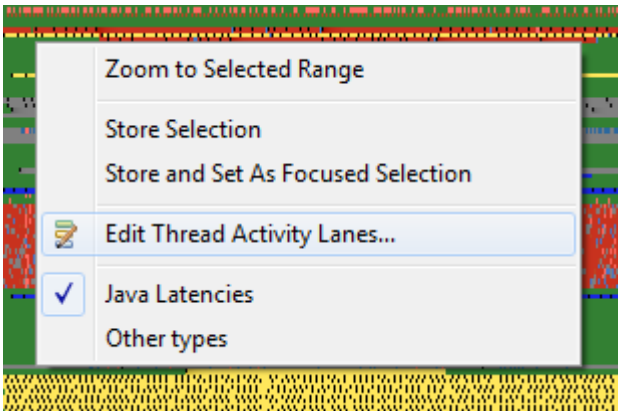
We've already had a brief interaction with the **Threads** report. It consists of a thread list and a timeline area.



The thread list is simply a list of threads, but the timeline is more interesting.

It has lanes for each thread and could visualize a wide range of JFR events. By default, they are so-called "Java Latency" events, but you can customize that via the context menu (see the **Edit Thread Activity Lanes** option).

Typically, individual events on the timeline will be tiny. You can hover to get the details under the mouse pointer, but it is not very useful if events are subpixel sized.



To zoom in, drag the rectangle over the timeline and hit **Zoom to Selected Range** on the context menu.

You can use the time range selected in the **Threads** report as a filter on other reports, too.

8. Considerations

One should keep in mind a few fundamental principles to use Mission Control efficiently.

- Flight Recorder can be started from the command line, which is very helpful in environments where JMX access is impossible or complicated. In addition, it does not require any upfront JVM configuration.
- CPU usage is a crucial metric. It is essential to know what kind of hotspot you are looking for. If you cannot monitor CPU usage in real time, you can find this information in the Flight Recorder file.
- CPU consuming "hot" hotspots are very different from "cold" ones caused by code spending time in idle state. For a former, traditional stack trace sampling (the **Method Profiling** report) works fairly well. "Cold" hotspots are more sophisticated. You may need to enable zero thresholds for I/O and contention events to fully picture idle state events in Mission Control.
- **Stack Trace** view is a potent tool that works for a wide range of events. For sampled events, you are most likely to use "hot methods" mode, whereas "call tree" is more informative for traced ones.

JDK Flight Recorder
How to discover code hotspots

bellsoft