JDK Flight Recorder How to discover code hotspots



Liberica JDK Revision 1.0 October 17, 2023

be//soft

Copyright © BellSoft Corporation 2018-2025.

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

THIS INFORMATION MAY CHANGE WITHOUT NOTICE. TO THE EXTENT PERMITTED BY APPLICABLE LAW, BELLSOFT PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL BELLSOFT BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF BELLSOFT IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

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.

Alpaquita, Liberica and BellSoft are trademarks or registered trademarks of BellSoft Corporation. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. Other trademarks are the property of their respective owners and are used only for identification purposes.

Contents

1. Introduction	5
2. Starting profiling with Mission Control	6
Working with Local JVM	6
Working with remote JVM	6
3. Before starting the profiler	8
4. Mission Control JMX Console and Thread monitor	9
What if JMX is not an option?	10
5. Starting JDK Flight Recorder session	12
Making JFR recording without Mission Control	14
6. Method profiling report	16
Stack Trace view	17

Group Stacktraces From	17
Distinguish Frames By	18
Layout Options	18
Why CPU usage is important	19
Retrospective thread CPU usage in Mission Control	19
Limit scope to a subset of threads	21
The "Stack Trace" view of the "Threads" report	22
Caveats of JFR method profiling	24

7. "Cold" code hotspots

"File I/O" report	26
"Socket I/O" report	27
"Lock Instances" report	28
"Threads" report	29

26

31

8. Considerations

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 <u>Liberica JDK Download Center</u>, and Mission Control binaries — In <u>Mission Control Download Center</u>.

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 <u>official documentation</u>.
- 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 <u>configuration tweaks</u> 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 solution 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.

Live Thread Graph					+ &	A
ive Threads						t
ive Threads 19:43:01			-			
Search the table		V	CPU Profiling	Deadlock Detect	tion 🔲 Allocatio	or
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	
tack Traces for Selected Thre	ads					
tack traces for selected threads	10.43.01					

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.

Start Flight Reco	ording	
Start Flight Reco	ording tings and then click Finish to start the flight recording.	
Destination File: Name:	C:\LABDIR\rec1.jfr My Recording	Browse
Time fixed recording time	ording me: ⁹ 2 min	
Continuous re Maximum siz	cording ze:	
Event settings:	rofiling - on server	emplate Manager
Description:	opfiguration for profiling typically around 2 % everboad	
	onfiguration for profiling, typically around 2 % overhead.	Ţ
Note:Time fixed re	ecordings will be automatically dumped and opened.	
?	< Back Next > Finish	Cancel

- 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.

I	Start Flight Recording	
	Event Options for Profilin Change the event options fo	g r the flight recording.
	Low overhead configuration	for profiling, typically around 2 % overhead.
	Garbage Collector:	Normal
	Memory Profiling:	Object Allocation and Promotion 👻
	Compiler:	Detailed
	Method Sampling:	Maximum
	Thread Dump:	Every 60 s 🔹
	Exceptions:	Errors Only 🗸
	Memory Leak Detection:	Object Types + Allocation Stack Traces
	Synchronization Threshold:	10 ms
	File I/O Threshold:	10 ms
	Socket I/O Threshold:	10 ms
	Class Loading	
	?	< <u>B</u> ack <u>N</u> ext > <u>Finish</u> Cancel

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.

Liberica Mission Control	CONTRACTOR DESIGNATION OF TAXABLE PARTY.	
<u>F</u> ile Edit <u>N</u> avigate <u>W</u> indow <u>H</u> elp		
🖉 🔤 JVM Br 📴 Outline 🖳 🗆	[[11.0.7] com.community.SiteApplication (16068)	- 8
 ■ a^t 2	O Method Profiling	⊘ ■
Java Application	No Selection> Aspect: Aspect: Show concurrent: Contain	ed 🛛 Same threads
Cock Instances	Top Package	Count
E Socket I/O	java.lang	150
Method Profiling	java.util	126
J. Exceptions	org.thymeleaf.engine	75
Thread Dumps	java.io	72 👻
⊿ internals IVM Internals	Top Class	Count
Galbage conections	🕞 java.lang.StringLatin1	86
	🕞 org.hsqldb.map.ValuePoolHashMap	36
G Class Loading	🕒 java.util.Arrays	30
M Operations		30
TLAB Allocations	G java.util.HashMap\$TreeNode	29
a 🚳 Environment	G iava.util.concurrent.ConcurrentHashMap	24 👻
Processes		
Environment Variables	≡ Stack Trace K 71 K] 🙆 🖉 🔻 🗖
System Properties	Stack Trace	Cou 🔺
Event Browser	t int java.lang.StringLatin1.hashCode(byte[]):0 (JIT Compiled)	
Exception Statistics	t String org.hsqldb.map.ValuePoolHashMap.getOrAddString(Object):0 (JIT Compiled)	
	t HashMap\$TreeNode java.util.HashMap\$TreeNode.getTreeNode(int, Object):0 (JIT Compiled)	
	t int java.lang.StringLatin1.compareToCI(byte[], byte[]):0 (JIT Compiled)	
	Diplot java.util.concurrent.ConcurrentHashMap.get(Object):0 (JIT Compiled)	
	boolean org.apache.catalina.connector.OutputBuffer.isBlocking():0 (JIT Compiled)	
	Iong java.io.ObjectOutputStream\$BlockDataOutputStream.getUTFLength(String):0 (JIT Complexity)	
	Void org.terracotta.statistics.jsr166e.LongAdder.add(long):0 (JIT Compiled)	
	StringBuilder java.lang.StringBuilder.append(String):0 (JII Compiled) StringBuilder intersteller.in	
	String org.nsqiab.lib.StringConverter.readUTF(byte[], int, int, char[]):0 (JTCompiled)	

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.

Group Stacktraces From	۱.		
Distinguish Frames By	×	<	Optimization Type
Layout Options Method Formatting Options	F F		Byte Code Index Line Number
Copy Clipboard Settings Store Selection	Ctrl+C	۲	Method Class Package
Store and Set As Focused Selection			

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.

Liberica Mission Control	and the second se	-	-	10 (annu 10)		
	[1107] com community SiteApplicatio	n (16068) 0 n rec1 202	20-06-22 10-11-10 -6244	511 ifr S?		П
	[11.0.7] conneonmanity.siteApplication	(10000) and rec1_202	20-00-22_10-11-10_80041			
	😼 Event Browser				🐋 🔝 🚍	
Automated Analysis Result						
⊿ 🛃 Java Application	<no selection=""></no>	Aspect: <no selection<="" td=""><td>Show concurrent</td><td>t: 🔄 Contained</td><td>d 📝 Same th</td><td>read</td></no>	Show concurrent	t: 🔄 Contained	d 📝 Same th	read
Cock Instances						
↓ File I/O	Event Types Tree	Start Time	Event Thread	System Mo	User Mod	Â
Socket I/O	CPU Load	22/06/2020, 10:10:09	C2 CompilerThread1	0 %	17.9 %	Ε
Method Profiling	A 🕞 Operating System 1 215	22/06/2020, 10:10:11	C2 CompilerThread1	0 %	13.5 %	
Issues a second seco	▲ Processor 317	22/06/2020, 10:10:34	C2 CompilerThread1	0 %	12.3 %	
🤌 Thread Dumps	CPU Load 60	22/06/2020, 10:10:19	C2 CompilerThread0	0.156 %	10.9 %	
🛛 🇑 JVM Internals	Thread CPU Load 245	22/06/2020, 10:10:39	C2 CompilerThread0	0.156 %	8.97 %	
前 Garbage Collections		22/06/2020, 10:10:49	C2 CompilerThread0	0.0389 %	6.38 %	
💼 GC Configuration		22/06/2020, 10:10:59	C2 CompilerThread0	0.117 %	6.24 %	
Compilations		22/06/2020, 10:10:10	C2 CompilerThread1	0.481 %	4.81 %	
Class Loading		22/06/2020, 10:10:59	http-nio-8080-exec	0.39 %	3.55 %	
🦃 VM Operations		22/06/2020, 10:10:49	http-nio-8080-exec	0.233 %	3.27 %	
TLAB Allocations		22/06/2020, 10:10:19	http-nio-8080-exec	0.156 %	2.96 %	
▲		22/06/2020, 10:10:49	http-nio-8080-exec	0.311 %	2.88 %	
Processes		22/06/2020, 10:10:29	http-nio-8080-exec	0.429 %	2.85 %	
Environment Variables		22/06/2020, 10:10:59	http-nio-8080-exec	0.234 %	2.85 %	
System Properties		22/06/2020, 10:10:19	http-nio-8080-exec	0.273 %	2.84 %	
Recording		22/06/2020, 10:10:29	http-nio-8080-exec	0.351 %	2.77 %	
Secontion Statistics		22/06/2020, 10:10:19	http-nio-8080-exec	0.117 %	2.69 %	
Exception statistics		22/06/2020, 10:10:19	http-nio-8080-exec	0.351 %	2.69 %	
		22/06/2020, 10:10:29	http-nio-8080-exec	0.195 %	2.65 %	
		22/06/2020, 10:10:29	http-nio-8080-exec	0.156 %	2.65 %	
		22/06/2020, 10:10:39	http-nio-8080-exec	0.156 %	2.65 %	
		22/06/2020, 10:10:59	http-nio-8080-exec	0.312 %	2.65 %	
		22/06/2020, 10:10:39	http-nio-8080-exec	0.429 %	2.61 %	
		22/06/2020, 10:10:29	http-nio-8080-exec	0.312 %	2.57 %	
	4	22/06/2020, 10:10:29	http-nio-8080-exec	0.312 %	2.57 %	-

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.

Edit <u>N</u> avigate <u>W</u> indow <u>H</u> elp						_
JVM B 📴 Outline 📃 🗆	[11.0.7] com.community.SiteApplicatio	n (16068) @@ rec1_202	0-06-22_10-11-10	_a634ba11.jfr 🖄		
	😼 Event Browser				o 🛤 💷	
🔤 Automated Analysis Result			_			
Java Application	<no selection=""></no>	Aspect: <no selection<="" td=""><td> Show cond </td><td>:urrent: 📃 Con</td><td>tained 🛛 📝 Same th</td><td>read</td></no>	 Show cond 	:urrent: 📃 Con	tained 🛛 📝 Same th	read
Threads						
⊳ 🚱 Memory						_
	Event Types Tree	Start Time	JVM System	JVM User	Machine Total	A
A Socket I/O	CPU Load	22/06/2020, 10:10:10	0.0381 %	0.384 %	16.1 %	
Method Profiling	· Concepting System 1 215	22/06/2020, 10:10:10	6.76 %	67.9 %	99.2 %	
I Exceptions	Operating System 1,215	22/06/2020, 10:10:12	5.39 %	56.4 %	99.2 %	
A Thread Dumps		22/06/2020, 10:10:13	9.67 %	55.4 %	100 %	
a 👸 JVM Internals	Thread CPU Load 245	22/06/2020, 10:10:14	8.62 %	64.9 %	100 %	Ε
Garbage Collections		22/06/2020, 10:10:15	5.71 %	60.7 %	100 %	
💼 GC Configuration		22/06/2020, 10:10:16	7.5 %	52.9 %	96.4 %	
General Compilations		22/06/2020, 10:10:17	6.51 %	61.7 %	100 %	
Class Loading		22/06/2020, 10:10:18	5.46 %	56.2 %	100 %	
🐓 VM Operations		22/06/2020, 10:10:19	7.77 %	53.6 %	100 %	-
TLAB Allocations		22/06/2020, 10:10:20	6.08 %	51.3 %	100 %	
a 🚳 Environment		22/06/2020, 10:10:21	5.91 %	55.9 %	100 %	
Processes		22/06/2020, 10:10:22	4.85 %	36 %	100 %	
Environment Variables		22/06/2020, 10:10:23	3.8 %	34.6 %	100 %	
System Properties		22/06/2020, 10:10:24	5.71 %	37.3 %	99.7 %	
For Kecording		22/06/2020, 10:10:25	10.2 %	51.7 %	100 %	
Event browser		22/06/2020, 10:10:26	5.15 %	62.6 %	100 %	
Exception statistics		22/06/2020, 10:10:27	7.71 %	51.9 %	99.1 %	
		22/06/2020, 10:10:28	6.22 %	51.3 %	100 %	
		22/06/2020, 10:10:29	7.4 %	55.1 %	100 %	
		22/06/2020, 10:10:30	8.91 %	55.4 %	100 %	
		22/06/2020, 10:10:31	3.29 %	37.4 %	100 %	
		22/06/2020, 10:10:32	5.41 %	38.2 %	99.2 %	
		22/06/2020, 10:10:33	5.3 %	45.4 %	90.5 %	
		22/06/2020, 10:10:34	8.51 %	51.1 %	87 %	-

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.

Method Profiling			<
4: Threads Histogram Selection Aspect: Thread Name Show concurrent:	Contai	ned 🗹 Sam	e threads
Top Package		Count	*
iava lang		55	
java.util		45	
org.attoparser	Γ	22	
org.thymeleaf.engine		22	-
Top Class		Count	^
€ java.lang.StringLatin1		34	
G org.hsqldb.map.ValuePoolHashMap	Γ	14	
G java.util.concurrent.ConcurrentHashMap	ĺ .	12	
G java.util.HashMap\$TreeNode		11	
G org.springframework.core.convert.TypeDescriptor		11	
G org.apache.catalina.connector.OutputBuffer		9	Ψ.
≡ Stack Trace k 1	K E	0	~
Stack Trace			Coι 🔺
t String org.hsgldb.map.ValuePoolHashMap.getOrAddString(Object) (JIT Compiled)			
t int java.lang.StringLatin1.hashCode(byte[]) (JIT Compiled)			
t int java.lang.StringLatin1.compareToCI(byte[], byte[]) (JIT Compiled)			
t HashMap\$TreeNode java.util.HashMap\$TreeNode.getTreeNode(int, Object) (JIT Compil	ed)		
b t Object java.util.concurrent.ConcurrentHashMap.get(Object) (JIT Compiled)			
boolean org.apache.catalina.connector.OutputBuffer.isBlocking() (JIT Compiled)			
void org.terracotta.statistics.jsr166e.LongAdder.add(long) (JIT Compiled)		7	stack traces (
TypeDescriptor org.springframework.core.convert.TypeDescriptor.forObject(Object) (JIT	Con		7 stack trace
Euffer java.nio.Buffer.limit(int) (JIT Compiled) Euffer java.nio.Buffer.l			, stack trace
CharBuffer java.nio.CharBuffer.put(String, int, int) (JIT Compiled)			•
< III			•

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.

🗏 Threads				2
8: Event Types Tree Selection	•	Aspect: Event Type' is 🔻 🗖	Show concurrent: Contained	✓ Same threads
Thread	Thread ^]		1
▶ [®] bl%0053tandard%0045le	main			
▶ [®] http-nio-8080-ClientPoll	main			
▶ [®] http-nio-8080-exec-11	main —			
<mark>▶[®] http-nio-8080-exec-13</mark>	main	At -∞ - 0	0:	
<mark>▶[©] http-nio-8080-exec-14</mark>	main 🗏	Three	ad Lifesnan of http-nio-8080-ever	-15: N/A
▶ [®] http-nio-8080-exec-15	main		ad Ellespan of http-mo-bood-exec	-15. N/A
▶ [®] http-nio-8080-exec-16	main			
▶ [®] http-nio-8080-exec-17	main			
▶ [®] http-nio-8080-exec-18	main	http-nio-8080-exec-15		
<mark>▶[©] http-nio-8080-exec-2</mark>	main 🖕	22/06/2020	10:10:30	10:11:00
▲		22/00/2020	10.10.50	10.11.00
E Stack Trace			ホント (m) (m) (m)	>
Stack Trace				Coi ^
String org.hsgldb.map.Valu	JePoolHash	Map.getOrAddString(Object) (J	IT Compiled)	
t int java.lang.StringLatin1.h	ashCode(by	/te[]) (JIT Compiled)		
t HashMap\$TreeNode java.u	itil.HashMa	p\$TreeNode.getTreeNode(int, (Object) (JIT Compiled)	
boolean org.apache.catalin	a.connecto	or.OutputBuffer.isBlocking() (JIT	Compiled)	
t int org.thymeleaf.engine.A	ttributeDefi	initions\$AttributeDefinitionRepo	ository.binarySearch(bo	
t int org.attoparser.ParsingN	larkupUtil.fi	indNextNonWhitespaceCharWil	ldcard(char[], int, int, in	
TypedValue org.springfram	nework.expr	ression.spel.ast.PropertyOrField	Reference.readProperty	
View Striptor org.springf	ramework.c	core.convert.TypeDescriptor.for	Object(Object) (JIT Con	
Object Java.util.HashMap.p	ut(Object,)	Object) (JIT Compiled)		
void java.utii.Arrays.soft(Ol	oject[], Con	nparator) (JTI Complied)		

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.

I liberica Mission Control		_			□ X			
Elle Edit Navinete Window Hale			_					
		1	11 10 -6245-11 %	~				
	🚓 Socket I/O			> 🚹 🚑	h 🗆			
an Automated Analysis Results								
Java Application								
Memory								
Cock Instances		T. 11/0 T	<u> </u>					
File I/O	Remote Address	Total I/O Time	Count	Read Count	Write (
 Socket I/Ο	127.0.0.1	8 min 49 s	610,471	319,307	29			
Method Profiling	192.168.100.1	1 min 38 s	3,019	1,387				
Exceptions	۲. III. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲. ۲.							
	Remote Port	Total I/O Time	Count	Read Count	Wri 🔦			
	9.005	8 min 42 s	588,702	314.448				
💼 GC Configuration	61,364	55.920 s	1,684	766				
Compilations	61.458	/2 188 e	1 003	155	-			
G Class Loading								
TLAB Allocations	Timeline Durations Event Log				:::::			
⊿ 🗿 Environment	Socket Read 16 MiB							
Processes	Socket Write							
Environment Variables	22/06/2020	10:10:15 10:10:20	10:10:45	10:11:00				
System Properties	22/00/2020	10.10.15	10.10.45	10.11.00				
French Browser								
Exception Statistics	E Stack Trace		K 7 K	E 💿 🔎	~			
	Stack Trace				Count			
	t int java.net.SocketInputStream.	read(byte[], int, int, int) (JIT Compiled)			316834			
	t void java.net.SocketOutputStream.socketWrite(byte[], int, int, int, int, int, int, int, int							
			16245					
			3738					
	↓ ↓ int sun.nio.ch.SocketChannelImpl.read(ByteBuffer) (Interpreted)							
	Int sun.nio.cn.socketChannelimpi.write(bytebuffer) (interpreted)				66			
4								

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.

Liberica Mission Control	Statement Statements	Manhoos and Park						
<u>F</u> ile Edit <u>N</u> avigate <u>W</u> indow <u>H</u> elp								
🔐 🔤 JVM Br 📴 Outline 🖳 🗖	[I1.0.7] com.community.SiteApplication (16068)	une rec1_2020-06-22_10-1	1-10_a634ba11.jfr 8	3				
🗖 🕹 💝 🗸	Lock Instances							
💩 🚵 Automated Analysis Results								
a 🛃 Java Application	Show concurrent: Contained Same threads							
Threads								
⊳ ♥ in Memory								
Cock Instances	Monitor Class	Total Blocked Time	Distinct Threads	Count 🔺				
+↓ File I/O	G java.io.RandomAccessFile	53.333 s	24	8,965				
• O Method Profiling	G net.sf.ehcache.store.disk.ods.FileAllocationTree	2.086 s	19	549				
J. Exceptions	G org.hsqldb.map.ValuePoolHashMap	1.188 s	17	5,389				
A Thread Dumps	Genet.sf.ehcache.store.cachingtier.OnHeapCachin	953.154 ms	17	211 -				
a 👸 JVM Internals	Monitor Address	Total Blocked Time	Distinct Threads	Count 🔺				
Garbage Collections	🗷 0x6B367E08	4.495 s	18	859				
GC Configuration	🗷 0x6BB5B108	2.914 s	18	179				
Compliations	🗷 0x639C2708	2.896 s	18	470				
W VM Operations TLAB Allocations	🗷 0x64F05988	2.472 s	18	422 👻				
	Thread	Total Blocked Time		Count 🔺				
	bl%004frder%0045lements.data	8.202 s		1,456				
	▶ [®] http-nio-8080-exec-2	3.709 s		937				
Environment Variables	▶ [®] http-nio-8080-exec-56	3.363 s		869				
Recording	http-nio-8080-exec-11	3.357 s		826 👻				
Event Browser								
Exception Statistics	E Stack Trace		K 7 K	E 🙆 🖉 🗸 🗖				
		Cou 🔺						
	C Element net.sf.ehcache.store.disk.DiskStorageFactory.read(DiskStorageFactory\$DiskMarker) 70							
	tring org.hsqldb.map.ValuePool.getString(String) (JIT Compiled) 52							
	void net.sf.ehcache.store.disk.ods.FileAllocationTree.free(Region) (JIT Compiled)							

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 socalled "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

be//soft