JDK Flight Recorder

How to discover memory issues

Liberica JDK
Revision 1.0
October 17, 2023

bell/soft

Contents

1. Introduction

While garbage collection (GC) and Stop-the-World (STW) pauses in Java are connected, they are not the same. The problem with GC is likely to manifest itself as prolonged STW pauses, but GC does not necessarily cause them.

General optimization of application code allocation patterns is another standard task, and Flight Recorder with Mission Control could be of great help here. Optimized memory allocation in application code is suitable both for GC and overall execution performance.

Finally, the application code may have memory leaks. Heap dumps are the most optimal tool for diagnosing memory leaks, JDK Flight Recorder offers an alternative approach with its pros and cons. In particular, heap dumps require Stop-the-World heap inspection to capture and tend to be extensive (depending on the heap size), while JFR files do not grow proportionally to the heap size.

2. Starting a memory profiling JDK Flight Recorder session

BellSoft provides Liberica JDK with JDK Flight Recorder support, as well as Mission Control. The Liberica JDK binaries are available in the Liberica JDK Download Center, and Mission Control binaries — in the Mission Control Download Center.

There are multiple ways to start a JDK Flight Recorder session. You can do it with Mission Control UI (locally or via JMX), with jcmd from the console, or even configure it to auto-start with JVM command-line arguments.

The configuration dialog of the Flight Recorder session in Mission Control has a few options relevant to the features explained later.

Garbage Collector

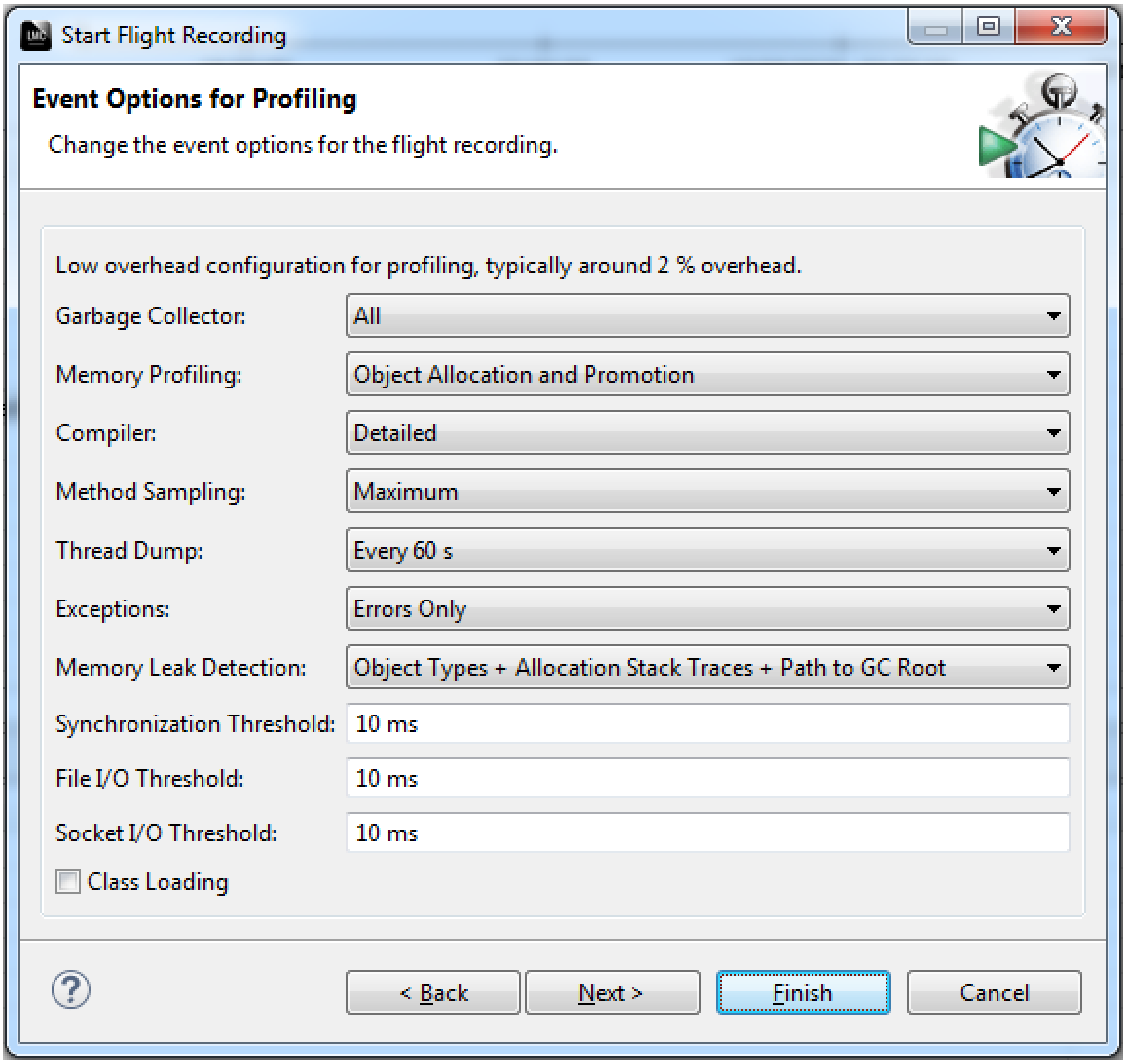
The default level is "Normal", and it should be enough. You may choose "All", which adds more details about GC Phases.

Memory Profiling

Here, you may choose one of three options:

- Off
- Object Allocation and Promotion
- All, including Heap Statistics

We recommend staying with "Object Allocation and Promotion". It provides valuable data points for allocation profiling with low overhead.



Heap Statistics included in the last option force regular Stop-the-World heap inspections which could be quite lengthy.

Memory Leak Detection

If you suspect memory leaks in your application, tweak the following options. You need to choose the "Object Types + Allocation Stack Traces + Path to GC Root" level of details to see the leaking path in the report afterward. Calculating "Path to GC Root" is a moderately expensive Stop-the-World operation, and we do not recommend enabling it just as a precaution.

3. Garbage collection in OpenJDK

OpenJDK JVM can use different implementations of garbage collectors.

The most commonly used ones in HotSpot JVM are G1 GC and Parallel GC.

There are several other algorithms available in various JDKs. Each GC has its unique features, but a few are the same.

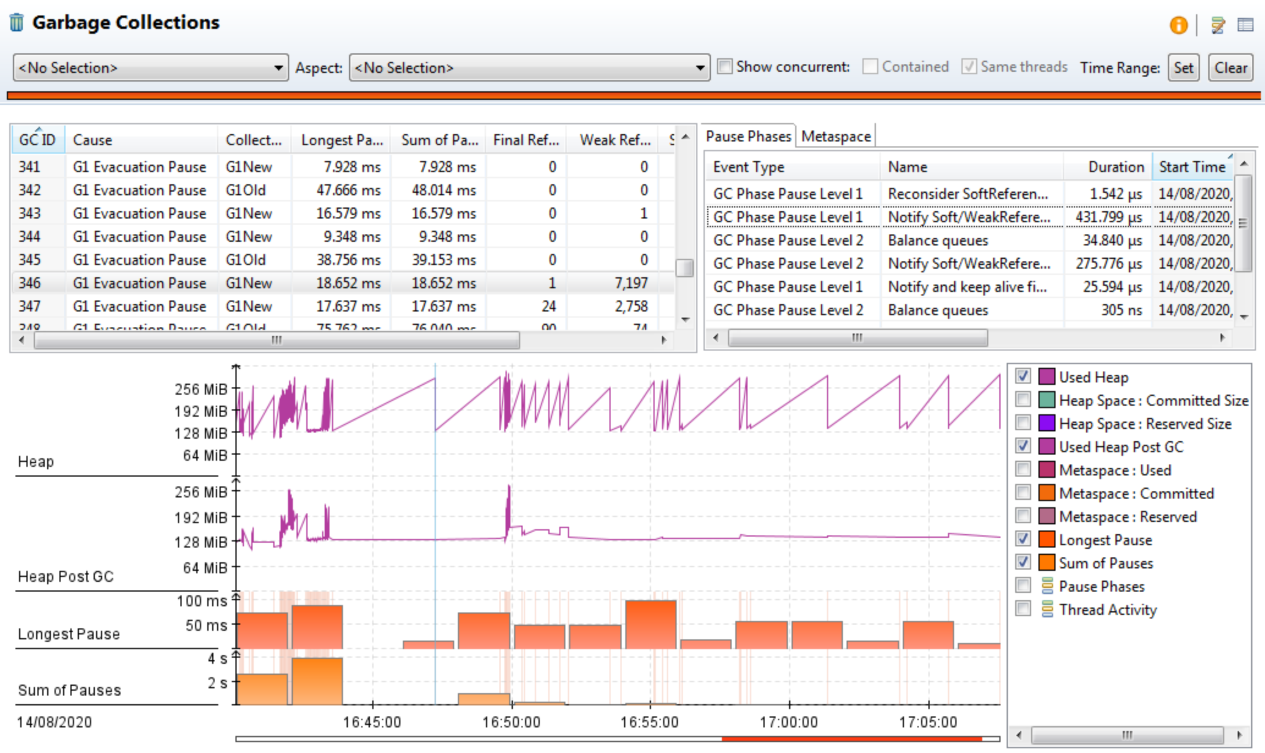
All the algorithms above are generational, meaning they split available heap space as separate young and old space. The result is three types of GC cycle:

- Minor (young) collections — recycle only young space
- Major (old) collections — recycle old and young space
- Last resort full collections — a big STW whole heap collection you want to avoid

Each GC cycle is split into a hierarchy of phases. Some could be concurrent (executing in parallel with application code), but most require a Stop-the-World pause.

4. Garbage collector report

Mission Control has a generic "Garbage Collections" report agnostic of the GC algorithm used by JVM.



This report shows a list of GC pauses with break down by phases (table on the right).

Information in this report could replace the typical usage of GC logs. Here, you can quickly spot the longest GC pause and dig a little deeper into its details. Notice GC ID assigned to each GC cycle: they are useful to pick additional events related to GC but not included in this report.

Two key metrics could be used to assess GC performance regardless of the algorithm:

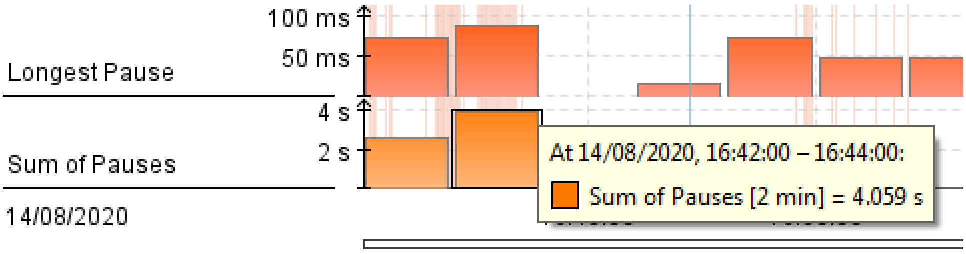
- Max duration of a Stop-the-World pause — it may add up to your application's request processing time, hence increasing response time.
- Percentage of time spent in a Stop-the-World pause during a certain period — such as, if we spent 6 seconds per minute in GC, the application can utilize 90% of available CPU resources at most.

The chart on the "Garbage Collections" report helps to spot both metrics easily. "Longest Pause" and "Sum of Pauses" visualize these metrics aggregated over regular time buckets (2 minutes in the screenshot).

"Longest Pause" shows the longest GC pause in the bucket, so you can observe your regular long pauses and outliers, if applicable.

"Sum of Pauses" shows the sum of all GC pauses in the bucket. Use this metric to derive the percentage of time spent on GC pauses. Hover your mouse over the bar, and you will see the exact numbers in the tooltip.

In the example below, the sum of pauses over 2 minutes equals 4 seconds. GC overhead is 4 / 120 ≈ 3.3%, which is reasonable.



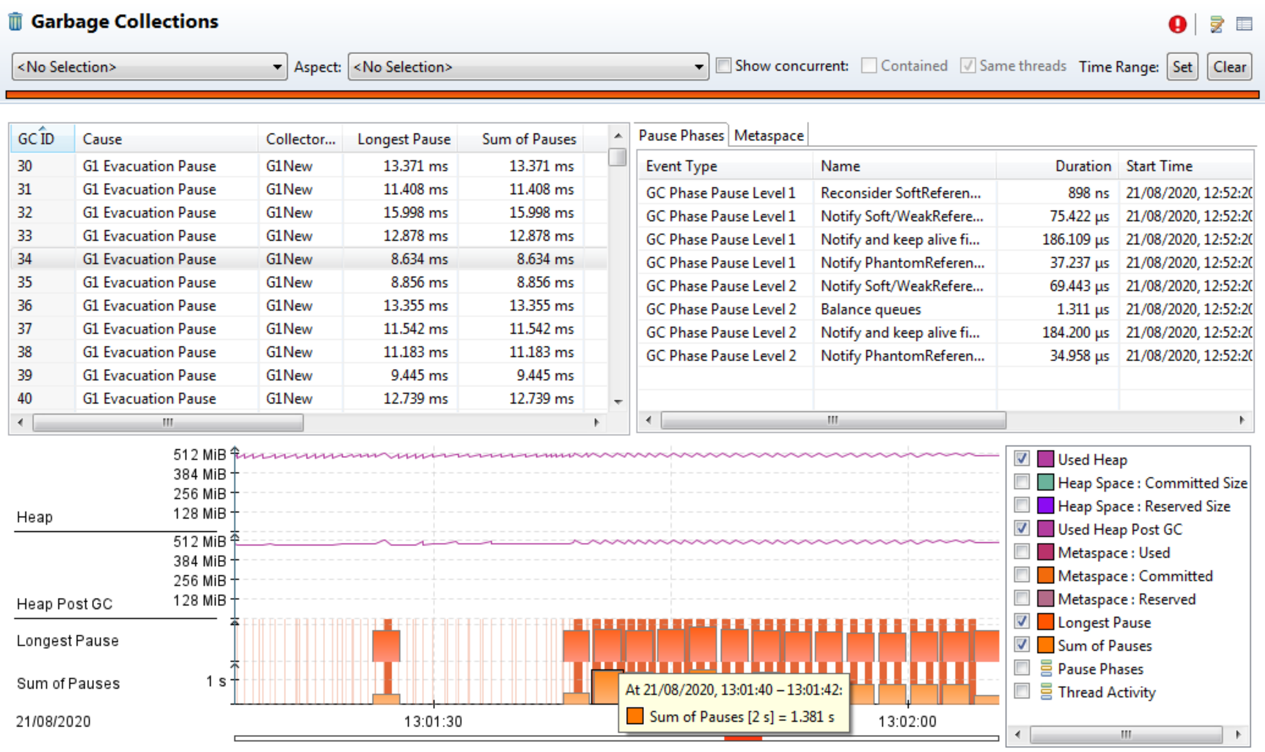
What if you are not happy with your GC performance?

GC tuning is a big topic out of scope of this document, but we will highlight a few typical cases.

"Heap is too small" problem

GC in modern JVM usually just works. A lot of engineering effort has gone into making GC adaptable for the application needs. But GC needs enough memory to accommodate application live data sets and some headroom to work efficiently.

If the heap size is not enough, you will see increased GC overhead (percentage of time spent on GC pauses) and, as a consequence, the application slowing down.



The screenshot above illustrates JVM running on low memory.

During the hovered time bucket, 1.4 seconds out of 2, JVM was in a GC pause, which is an absolutely unhealthy proportion.

Often increasing the max heap allowed to JVM solves the problem with high GC overhead. However, if an application has a memory leak, increasing heap size is no help. The leak needs to be fixed.

"Metaspace is too small" problem

Metaspace is a memory area to store class metadata. Starting from Java SE 8, metaspace has not been part of heap, even though the lack of available space in metaspace may cause GC there.

To free some memory in metaspace, JVM needs to unload unused classes. Class unloading, in turn, requires a major GC on the heap. The reason is that objects on the heap keep references to corresponding classes, and a class cannot be unloaded until any instance of it exists in the heap.

You will quickly spot the metaspace keyword in the **Cause** column in this case. You can also view the metaspace size in the timeline chart at the bottom of the report to check if you are hitting the limit.

Special reference abuse

Java offers utility classes that have very specific semantics related to garbage collection. These are weak, soft, and phantom references. Besides, JVM uses finalizer references internally to implement the semantics of finalizers.

All these references require particular processing within the bounds of a GC pause. Due to semantic special references, they can only be processed after all live objects are found.

Usually, this processing runs fast, but abusing these special references (especially finalizers as they are most expensive) can cause abnormally long pauses for otherwise healthy GC.

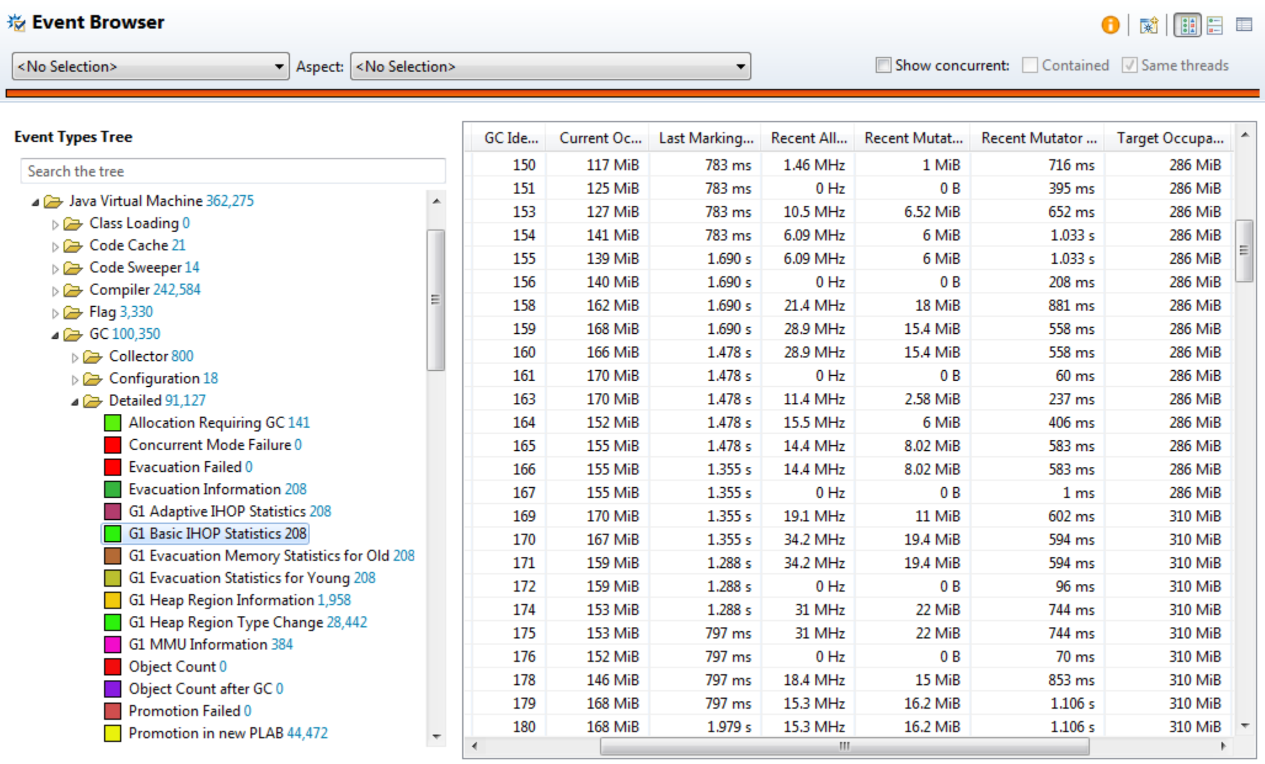
Columns with a count of each special reference type are on the right side in the GC event table. Numbers on the order of tens of thousands could indicate a problem.

You can also find the reference processing phase in the **Pause Phases** table and see how much time was spent on it.

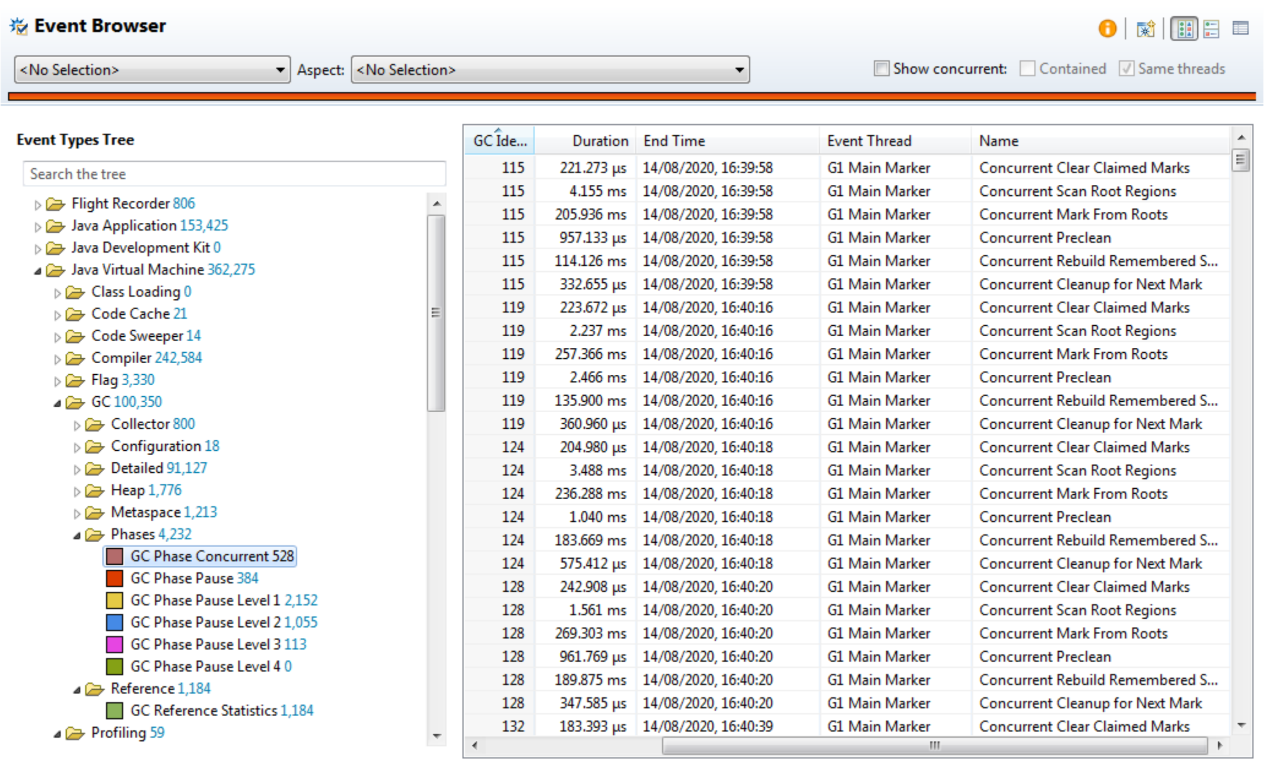
5. More GC related events

The "Garbage Collections" (**GC**) report is a handy tool to spot standard problems, but it does not display all GC details available from JDK Flight Recorder.

As usual, you can go to the event browser and look at specific events directly. Look under **Java Virtual Machine** > **GC** > **Details** for more diagnostic data related to garbage collection.



Besides work done under Stop-the-World, some collectors (G1 in particular) use background threads working in parallel with application threads to assist in memory reclamation. You may find details for such tasks if you look at the **GC Phase Concurrent** event type.


Event Browser
<No Selection>
Aspect:
<No Selection>
Show concurrent:
Contained
Same threads
Event Types Tree
Search the tree
Flight Recorder 806
Java Application 153,425
Java Development Kit 0
Java Virtual Machine 362,275
Class Loading 0
Code Cache 21
Code Sweeper 14
Compiler 242,584
Flag 3,330
GC 100,350
Collector 800
Configuration 18
Detailed 91,127
Heap 1,776
Metaspace 1,213
Phases 4,232
GC Phase Concurrent 528
GC Phase Pause 384
GC Phase Pause Level 1 2,152
GC Phase Pause Level 2 1,055
GC Phase Pause Level 3 113
GC Phase Pause Level 4 0
Reference 1,184
GC Reference Statistics 1,184
Profiling 59
GC Ide... Duration End Time Event Thread Name
115 221.273 µs 14/08/2020, 16:39:58 G1 Main Marker Concurrent Clear Claimed Marks
115 4.155 ms 14/08/2020, 16:39:58 G1 Main Marker Concurrent Scan Root Regions
115 205.936 ms 14/08/2020, 16:39:58 G1 Main Marker Concurrent Mark From Roots
115 957.133 µs 14/08/2020, 16:39:58 G1 Main Marker Concurrent Preclean
115 114.126 ms 14/08/2020, 16:39:58 G1 Main Marker Concurrent Rebuild Remembered S...
115 332.655 µs 14/08/2020, 16:39:58 G1 Main Marker Concurrent Cleanup for Next Mark
119 223.672 µs 14/08/2020, 16:40:16 G1 Main Marker Concurrent Clear Claimed Marks
119 2.237 ms 14/08/2020, 16:40:16 G1 Main Marker Concurrent Scan Root Regions
119 257.366 ms 14/08/2020, 16:40:16 G1 Main Marker Concurrent Mark From Roots
119 2.466 ms 14/08/2020, 16:40:16 G1 Main Marker Concurrent Preclean
119 135.900 ms 14/08/2020, 16:40:16 G1 Main Marker Concurrent Rebuild Remembered S...
119 360.960 µs 14/08/2020, 16:40:16 G1 Main Marker Concurrent Cleanup for Next Mark
124 204.980 µs 14/08/2020, 16:40:18 G1 Main Marker Concurrent Clear Claimed Marks
124 3.488 ms 14/08/2020, 16:40:18 G1 Main Marker Concurrent Scan Root Regions
124 236.288 ms 14/08/2020, 16:40:18 G1 Main Marker Concurrent Mark From Roots
124 1.040 ms 14/08/2020, 16:40:18 G1 Main Marker Concurrent Preclean
124 183.669 ms 14/08/2020, 16:40:18 G1 Main Marker Concurrent Rebuild Remembered S...
124 575.412 µs 14/08/2020, 16:40:18 G1 Main Marker Concurrent Cleanup for Next Mark
128 242.908 µs 14/08/2020, 16:40:20 G1 Main Marker Concurrent Clear Claimed Marks
128 1.561 ms 14/08/2020, 16:40:20 G1 Main Marker Concurrent Scan Root Regions
128 269.303 ms 14/08/2020, 16:40:20 G1 Main Marker Concurrent Mark From Roots
128 961.769 µs 14/08/2020, 16:40:20 G1 Main Marker Concurrent Preclean
128 189.875 ms 14/08/2020, 16:40:20 G1 Main Marker Concurrent Rebuild Remembered S...
128 347.585 µs 14/08/2020, 16:40:20 G1 Main Marker Concurrent Cleanup for Next Mark
132 183.393 µs 14/08/2020, 16:40:39 G1 Main Marker Concurrent Clear Claimed Marks

6. Stop-the-World pauses in OpenJDK

A Stop-the-World (STW) pause is a state of JVM when all application threads are frozen, and internal JVM routines have exclusive access to the process memory.

Hotspot JVM has a protocol called safepoint to ensure proper Stop-the-World pauses.

STW pauses are mainly associated with GC activity, but it is only one possible reason. In Hotspot JVM, STW pauses get involved in other special JVM operations.

Namely:

- JIT compiler related operations (e.g. deoptimization or OSR)
- Bias lock revocation
- Thread dumps and other diagnostic operations, including JFR-specific ones

As a rule, STW pauses are unnoticeably short (less than a millisecond), but putting JVM on a safepoint is a sophisticated process. Things can go wrong here.

Safepoint protocol requires each thread executing Java code to explicitly put itself in a "safe" state, where JVM knows exactly how local variables are stored in the memory on the stack.

After it starts, each thread running Java code receives a signal to put itself on the safepoint. A JVM operation cannot begin until all threads have confirmed their safe state (threads executing native code via JNI are an exception; STW does not stop them unless they try to access heap to switch from native code to Java).

If one or more threads are slow at reaching the safe state, the rest of the JVM will wait for them to be frozen effectively.

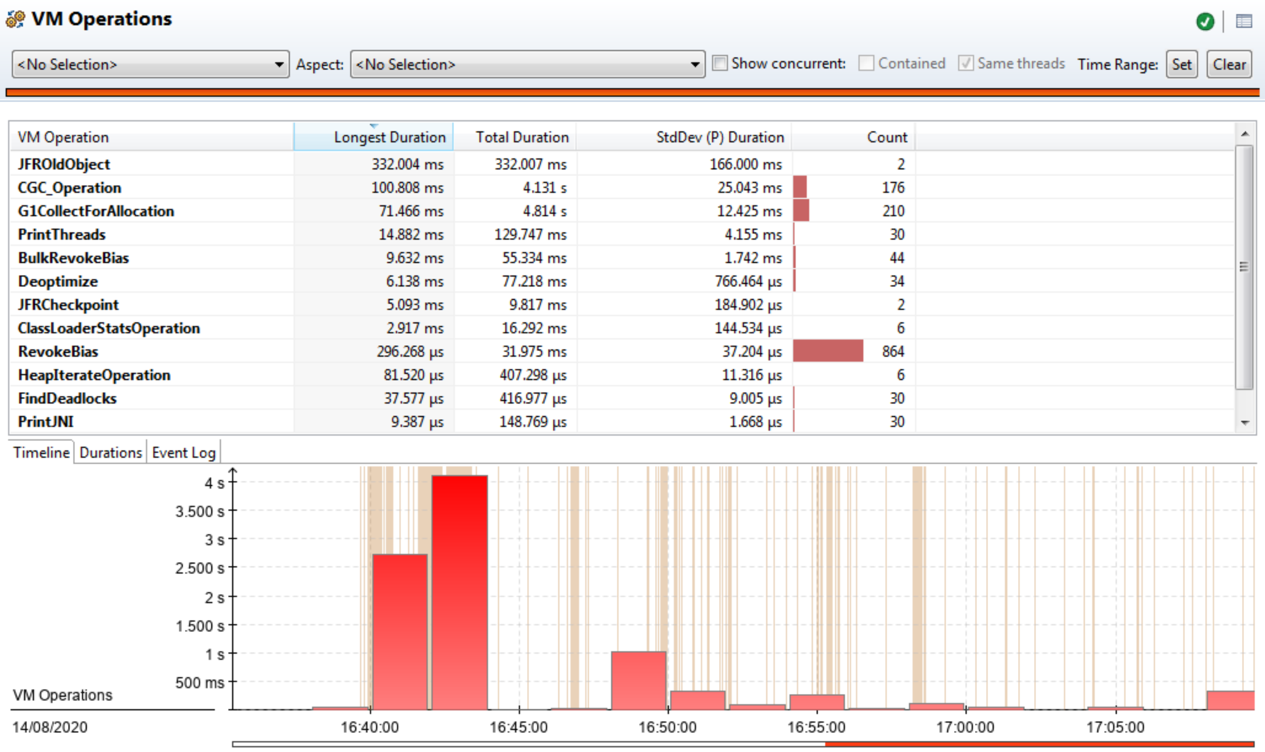
There are two main reasons for such misbehavior:

- A thread is blocked while accessing a memory page (d state in Linux) and cannot react.
- A thread is stuck in a hot loop, where the JIT compiler omitted a safepoint check as it was considered too fast of a loop.

The first situation may happen if the system is swapping or application code is using memory-mapped IO. The second one may occur in heavily optimized computation-intensive code (in Java runtime, java.lang.Sting and java.util.BigDecimal operations over large objects are common culprits).

7. VM Operations report

Mission Control has a report with a summary of all Stop-The-World VM operations.



VM Operation	Longest Duration	Total Duration	StdDev (P) Duration	Count
JFROldObject	332.004 ms	332.007 ms	166.000 ms	2
CGC_Operation	100.808 ms	4.131 s	25.043 ms	176
G1CollectForAllocation	71.466 ms	4.814 s	12.425 ms	210
PrintThreads	14.882 ms	129.747 ms	4.155 ms	30
BulkRevokeBias	9.632 ms	55.334 ms	1.742 ms	44
Deoptimize	6.138 ms	77.218 ms	766.464 µs	34
JFRCheckpoint	5.093 ms	9.817 ms	184.902 µs	2
ClassLoaderStatsOperation	2.917 ms	16.292 ms	144.534 µs	6
RevokeBias	296.268 µs	31.975 ms	37.204 µs	864
HeapIterateOperation	81.520 µs	407.298 µs	11.316 µs	6
FindDeadlocks	37.577 µs	416.977 µs	9.005 µs	30
PrintJNI	9.387 µs	148.769 µs	1.668 µs	30

This report shows a summary grouped by type of operations. In the screenshot above, **CGC_Operation** and **G1CollectForAllocation** are the only operations related to GC (G1 in particular).

You can see a fair number of other VM operations, but they are very quick.

Also, bear in mind that many operations are caused by JFR itself, such as **JFRCheckpoint**, **JFROldObject**, **FindDeadlocks**, etc.

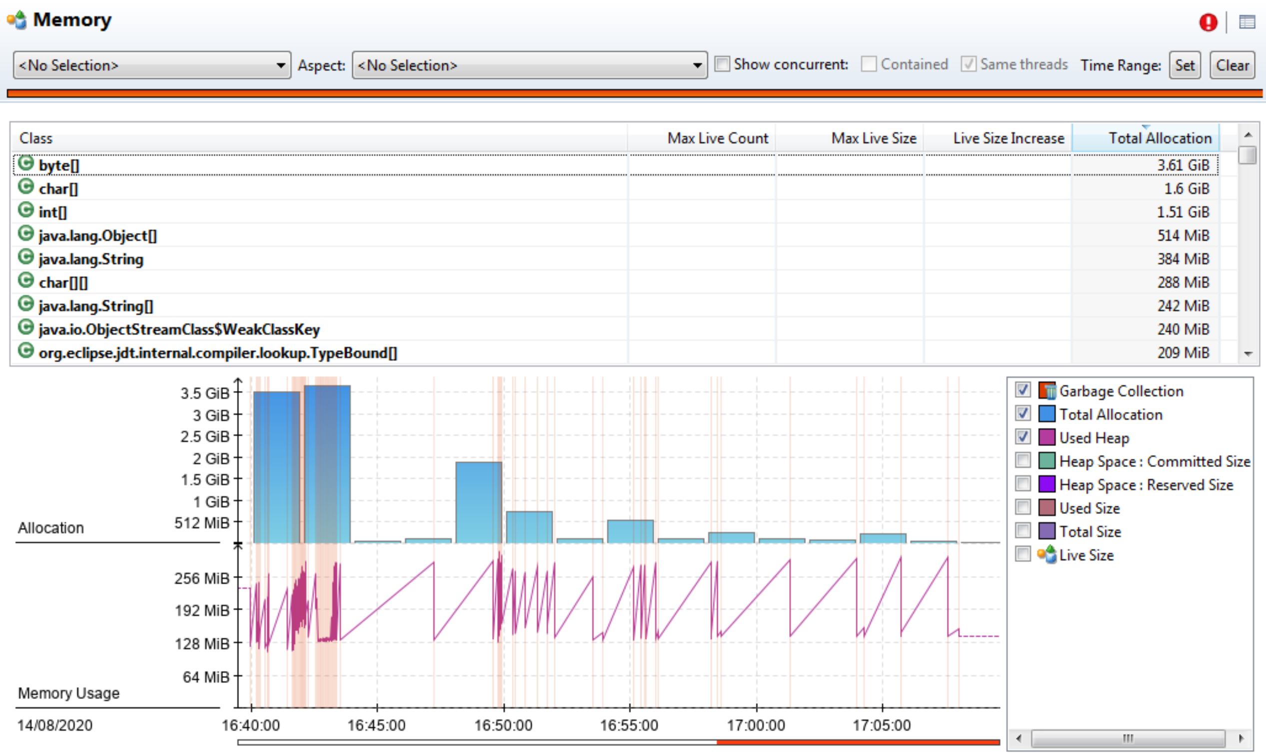
Problems with safepoints are relatively rare in practice, but spending a few seconds to quickly check the "VM Operations" report for abnormal pauses is a good habit.

8. Memory allocation profiling

While JVM is heavily optimized for allocating a lot of short-lived objects, excessive memory allocation may cause various problems.

Allocation profiling helps to identify which code is responsible for the most intense allocation of new objects.

Mission Control has a "Memory" report giving an overview of object allocation in application code.



This report shows a timeline of the application's heap allocation rate and a histogram of allocation size by object type.

At first glance, the report looks too ascetic and done in broad strokes, notably if you used Mission Control 5.5.

There are many more details to squeeze out from this report with Mission Control, but we need to get an idea of the source for these numbers.

How are allocation profiling data collected?

Allocation profiling in JVM was a rather challenging task. While previously many profilers had allocation profiling features, the performance impact on an application with allocation profiling turned on was inconvenient and often prohibitive.

Allocation profiling is based on sampling. A runtime allocation profiler collects a sample of allocation events to get the whole picture. Before JEP-331 (available since OpenJDK 11), profilers had to instrument all Java code and inject extra logic at every allocation site. The performance impact from such code mangling was dramatic.

JDK Flight Recorder, on the contrary, was always able to use low overhead allocation profiling. The key is to record TLAB allocation events instead of sampling normal ones.

Almost all new Java objects are allocated in the so-called Eden (a part of young object heap space). Not to compete for shared memory management structures, each Java thread reserves a thread local allocation buffer (TLAB) in Eden and allocates new objects there. It is an essential performance optimization for multicore hardware.

Eventually, the buffer dries up, and the thread has to allocate a new one. This type of event is the one recorded by Flight Recorder. The average TLAB size is roughly 500KiB (size is dynamic and changes per thread), so one allocation per approximately 500 KiB of allocated memory is recorded. This way, Flight Recorder does not add any overhead to the hot path of object allocation, yet it can get enough samples for further analysis.

Each sample has a reference to the Java thread, stack trace, type, and size of the object being allocated, as well as the size of the new TLAB. You can find raw events in the event browser under **Allocation in new TLAB**.

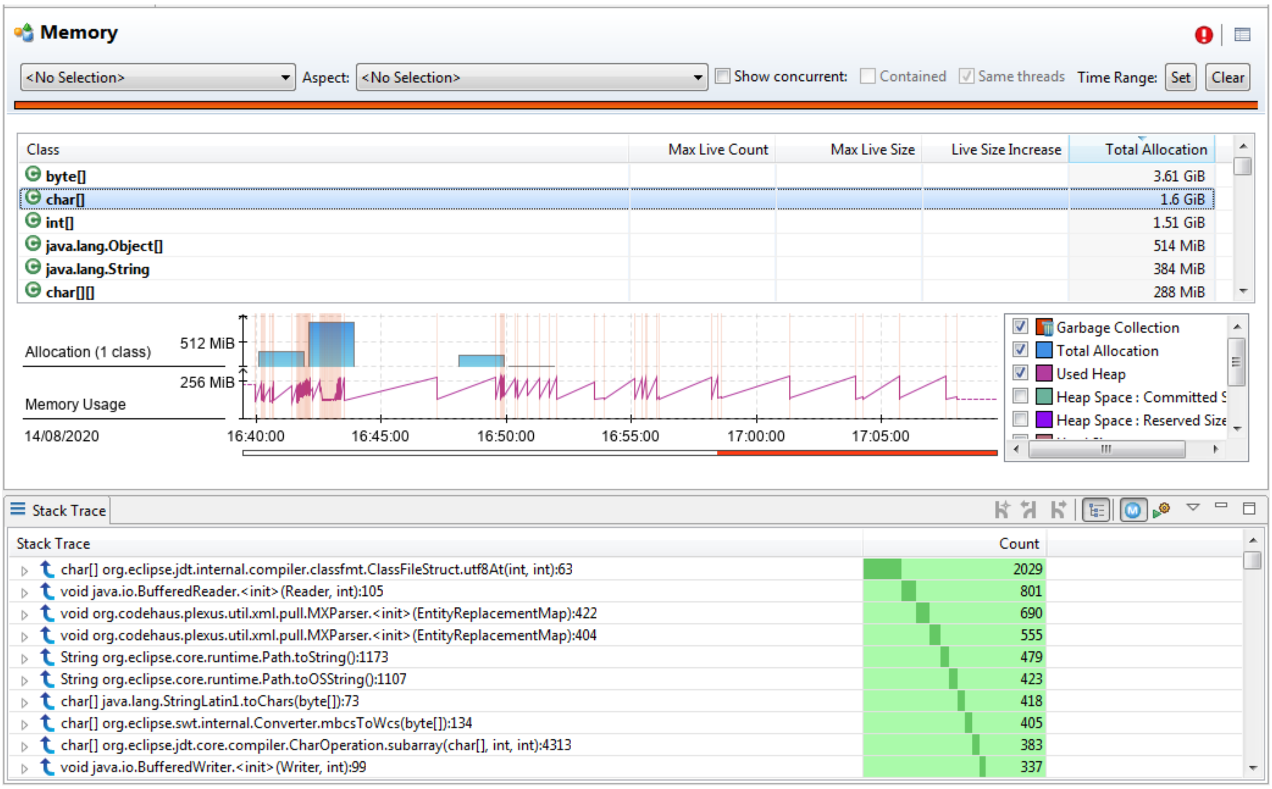
How to find out which code is allocating the most

The **Memory** report based on the "Allocation in new TLAB" events has associated thread and stack trace. It means we can use the **Stack Trace** view and filter to extract much more details from this report.

Typical workflow for identifying memory hot spot looks as follows:

- Open the **Memory** report.
- Open the **Stack Trace** view (enabled via **Window** > **Show View** > **Stack Trace** if hidden).
- Enable **Show as Tree** and **Group traces from last method** on the **Stack Trace** view. This is the most convenient configuration for this kind of sampling.

- Switch **Distinguish Frames By** to **Line Number** in the context menu of the **Stack Trace** view if you want to see line numbers.
- By this point, you can already see a top hot spot of allocation in your application. You may also select a specific object type in the **Memory** report to only examine these type-related allocation traces.



Further, you can select a range of timelines to zoom in on a definite period. Another functionality here is applying filters to narrow the report to individual threads.

9. Live object sampling

Memory leaks are another well-known problem for Java applications. Traditional approaches to memory leak diagnostics rely on heap dumps.

Heap dumps are a vital tool for dealing with memory troubleshooting optimizations, but they have their drawbacks. More specifically, heap dumps can be quite large, slow to process, and require Stop-the-World heap inspection to capture.

Flight Recorder offers an alternative approach, live object sampling.

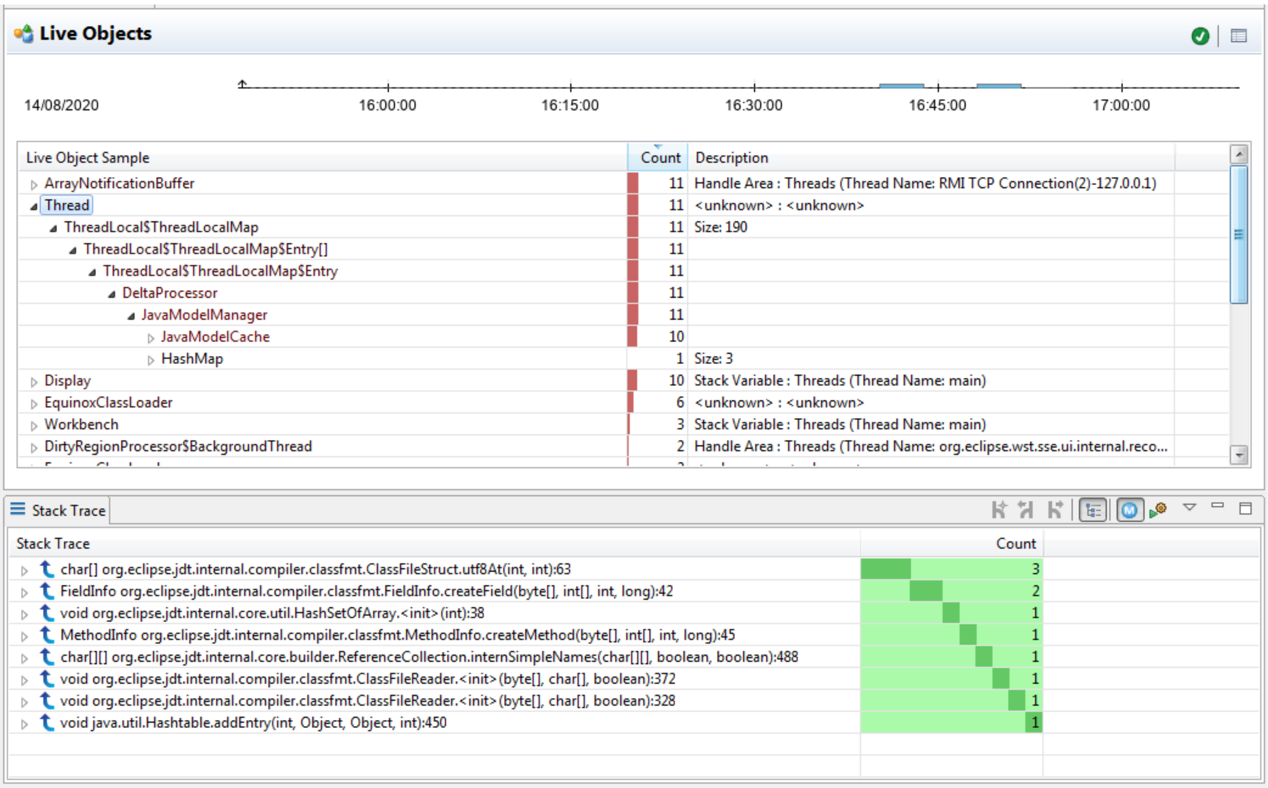
The idea is simple: we already have allocation sampling. Now, from the newly allocated objects, Flight Recorder picks few to trace across their lifetime. If an object dies, it is excluded from the sample (and no event is produced).

Then events are dumped into a file at the end of recording. Flight Recorder may optionally calculate and record the path to the nearest GC root for each object in a live object sample (a configuration option mentioned at the beginning of this document). This operation is expensive — requiring an STW heap inspection — but necessary to understand how an object is leaking.

But these are just random objects. How will sampling help to find a leak?

Normal objects are short-lived, but leaked objects can survive for long. If you sample JVM for a long enough time, you will likely catch a few leaked objects in the sample. Once these objects get to the final report, we can identify where they are leaking via the path to the nearest GC root, also recorded.

Mission Control has a **Live Objects** report to visualize this kind of event. You will also need to enable the **Stack Trace** view.



In the table, you can see a sampled object grouped by GC root. You can unfold a reference path from a GC root down to individual objects from the sample.

You can see the allocation stack trace of each sampled object in the ``Stack Trace'' view as well. This information is unique to Flight Recorder and cannot be reconstructed from a regular heap dump.

This feature is relatively new in Flight Recorder, whereas heap dump analysis is a reliable, time-tested technique.

A live object sample in Flight Recorder is rather small, and it is a matter of luck whether it catches a problematic object or not. On the other hand, Flight Recorder files several orders smaller in magnitude compared to heap dumps.

While Old Object Sampling is definitely not a silver bullet, it offers a viable alternative to the traditional heap dump wrangling approach for memory leak investigations.

JDK Flight Recorder

How to discover memory issues

bell/soft