# 

[**Abstract**](#_wimjjc9oppm9) **1**

[**Caveats**](#_4shwpwiz83v) **1**

[**Benchmark Summary**](#_7ifj51wghvrd) **2**

[Summary of observations :](#_85qglwav48oi) 2

[**GC analysis using MissionControl**](#_idr1oxdtg7jw) **3**

[nopool-small](#_n52yc7wbobgu) 3

[pool-small](#_g5k1mwo1rwvx) 4

[nopool-large](#_b2rtlgofj2rw) 4

[Pool-large](#_gpzi2qn8nh59) 5

[**Code repo**](#_trpelnkd2alk) **6**

# Abstract

Java’s strength as a developer friendly, language managed memory is also its primary obstacle to ultra low latency applications (5-20 mics ingress to outgress).

Java’s default memory management and GC techniques are often the reason for unacceptable pause times.

Among the many techniques used, one key technique used is object pooling.

Benchmark test done below studies the impact of object pooling in 2 commodity grade machines - OSX and Windows8.1 using standard Oracle JDK 11.0.10 (LTS) and graalvm-ce-java11-21.0.0.2. The test covers two flavors of objects - small and large objects resembling a single ticker and basket of tickers respectively.

# Caveats

1. Below code is not thread-safe, but can be made so easily by changing the data structure to ArrayBlockingQueue and keeping a secondary data structure for additional capacity.
2. Benchmark results were captured after a single run after JVM shutdown for each flavor, smallobject-no pool, smallobject-pool, largeobject-nopool and largeobject-pool respectively
3. Averages of the 50th, 90th, 99th, 99.99th percentile were taken over a performance test done in 1 minute, with samples taken every second. The first 2 entries were discarded to avoid the usual JVM warmup related outliers. For production grade systems, we need to do more test runs to prove our hypothesis.
4. FlightRecorder was also run during the benchmark to verify runtime profiles.

# Benchmark Summary

All percentiles below in nanos

-XX:+FlightRecorder -XX:StartFlightRecording=duration=90s,filename=pool-small.jfr -Xms512M -Xmx1024M -XX:+UseG1GC

## Summary of observations :

1. Throughput (objects created/fetched per minute) was 1.3-2x times more in object pooled examples
2. Latency differences of the highest magnitude came from the 99.99th percentile, in many cases order of magnitude lesser latency was observed in the object pooled example
3. Interestingly, graalvm performed better than Oracle’s JDK in the 90th percentile bucket for both small and large objects, but performed worse in the throughput department in both cases. Behavior was reproducible in 3 runs.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Throughput in 1 min | 50th percentile avg | 90th percentile avg | 99th percentile | 99.99th percentile |
| 8 cores on Windows 8.1 OS on 11.0.10 JDK | Small object | without pool | 1009233034 | 0 | 302 | 302 | 4127 |
| with pool | 2145709136 | 0 | 302 | 302 | 302 |
| Large object | without pool | 277180921 | 0 | 302 | 322 | 10334 |
| with pool | 2101186082 | 0 | 302 | 302 | 302 |
| 8 cores on Windows 8.1 - GraalVM graalvm-ce-java11-windows-amd64-21.0.0.2\graalvm-ce-java11-21.0.0.2 | Small object | without pool | 902211936 | 0 | 302 | 302 | 4796 |
| with pool | 1821628252 | 0 | 0 | 302 | 302 |
| Large object | without pool | 319650671 | 301 | 302 | 302 | 9831 |
| with pool | 1846821561 | 0 | 0 | 302 | 302 |
| 4 cores on Mac OS X OS on 11.0.10 JDK | Small object | without pool | 513284230 | 70 | 73 | 92 | 5965 |
| with pool | 706066436 | 38 | 41 | 49 | 3238 |
| Large object | without pool | 267832802 | 102 | 171 | 389 | 11567 |
| with pool | 741270463 | 38 | 40 | 50 | 739 |

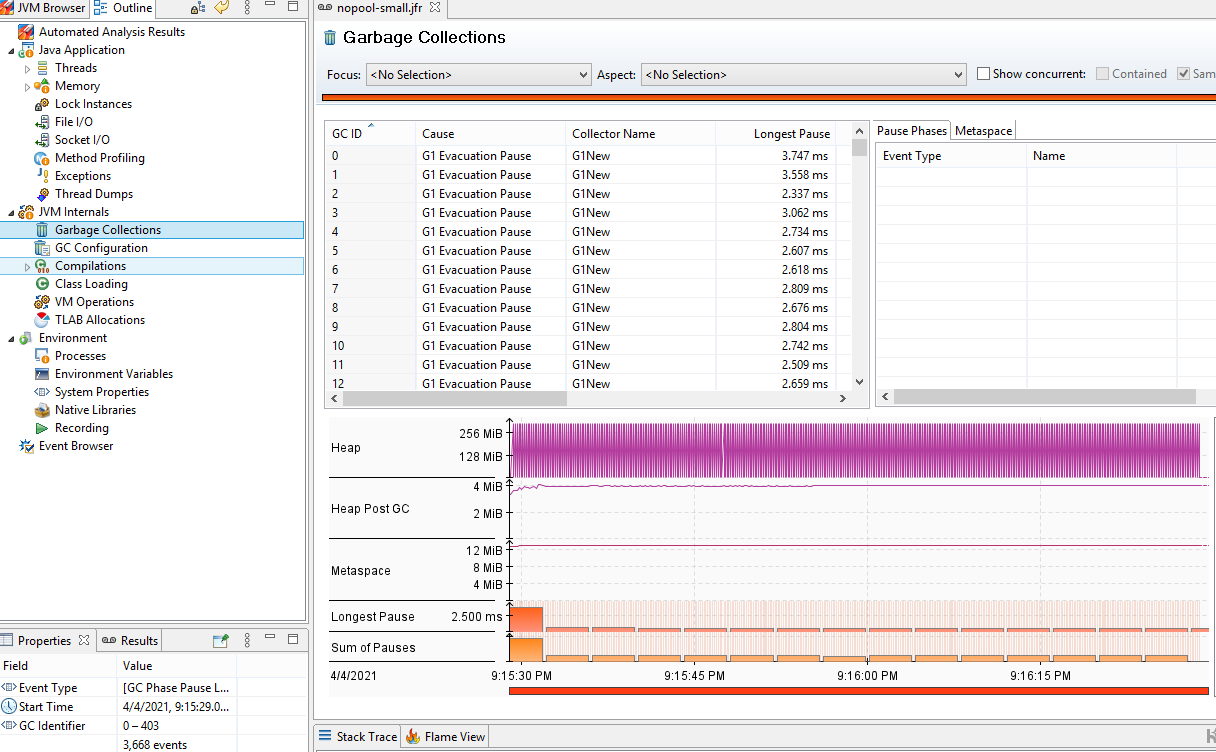
# 

# GC analysis using MissionControl

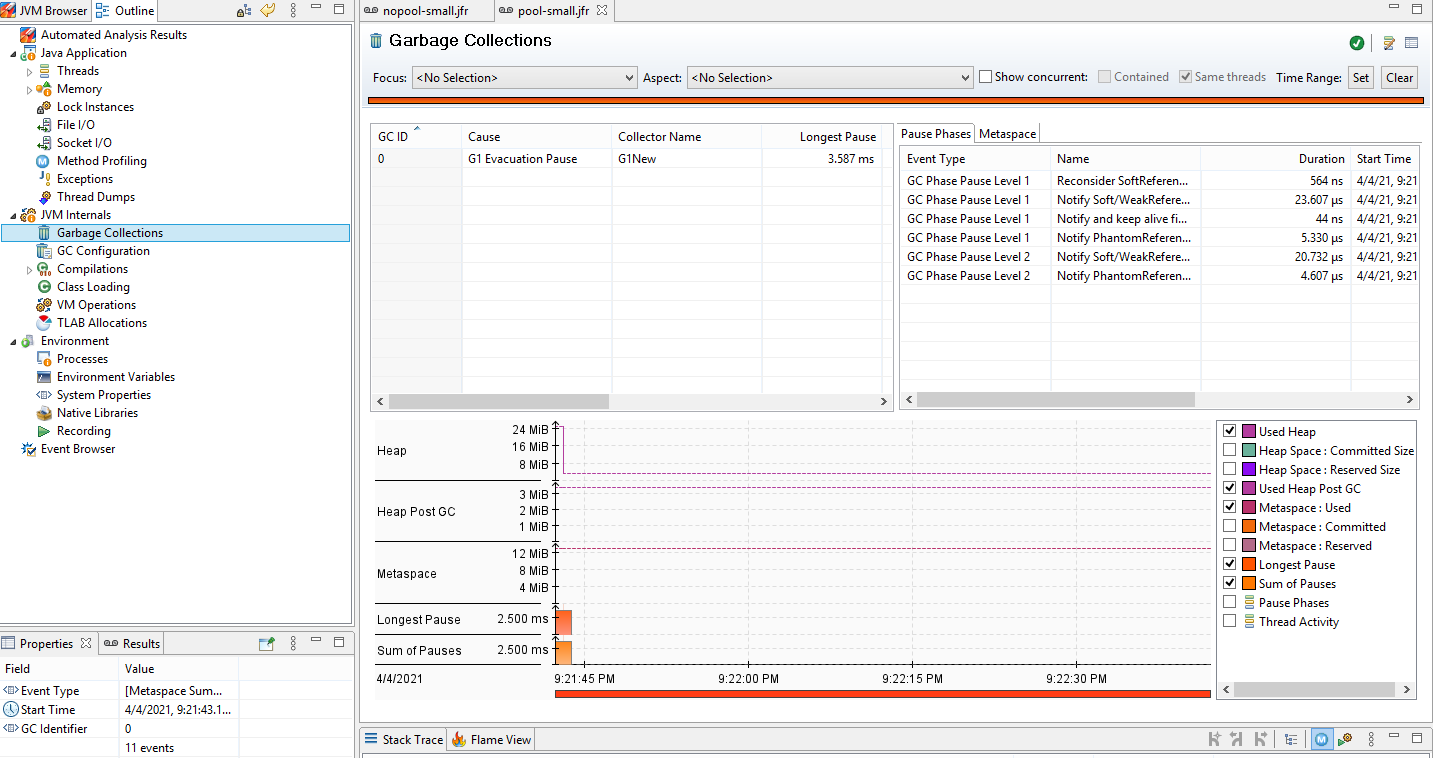
All examples are from the Windows sample, the OSX samples are similar except in the last example, hence added to the screenshots.

Other \*jfr are added to the repo.

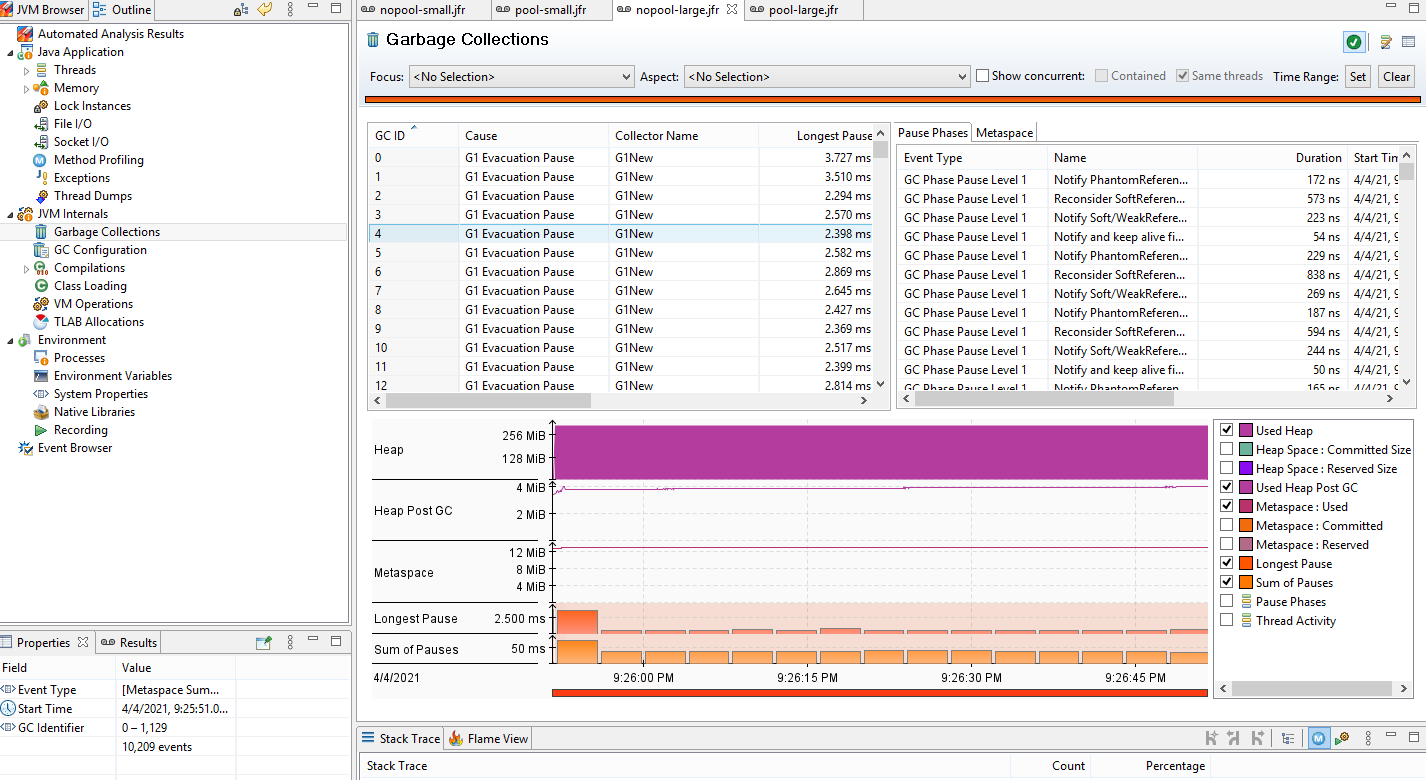
## nopool-small



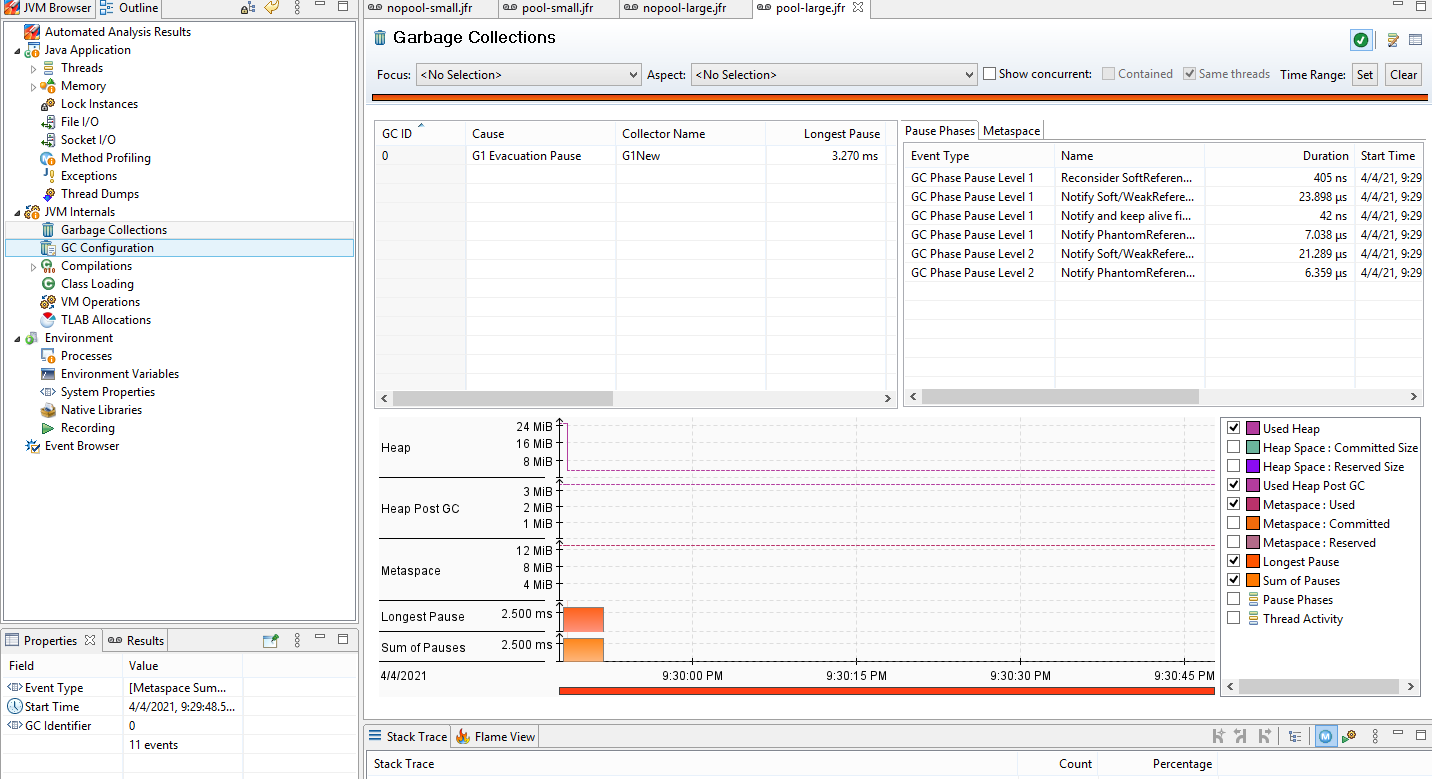
## pool-small



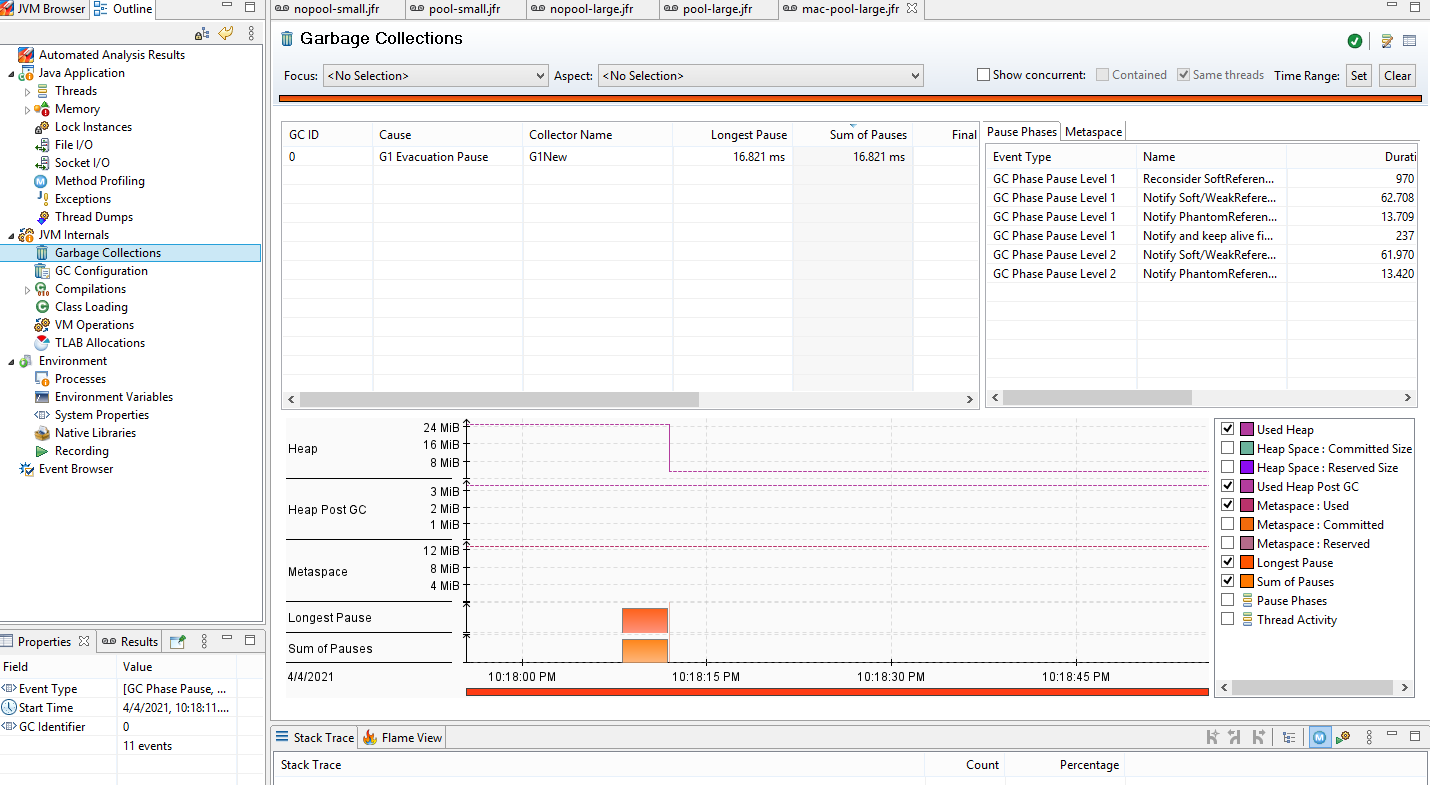
## nopool-large



## Pool-large



This was an interesting use-case as the number of objects created was larger than the number created using the small-objects usecase.



# Code repo

<https://github.com/anirbanchowdhury/SimpleJavaObjectPool>