Performance Optimisations

Angelo D' Agnano
John Kostaras

Performance activities

* Performance Monitoring
  * an activity of nonintrusively collecting or observing performance data from a running application
* Performance Profiling
  * an activity of collecting performance data from a running application that may be intrusive on application performance responsiveness or throughput
* Performance Tuning
  * an activity of changing tuneables, source code, or configuration attribute(s) for the purposes of improving application responsiveness or throughput

OS Performance Monitoring

* CPU
  * Graphical
    * gnome-system-monitor, xosview (Unix/Linux)
    * cpubar (Solaris)
    * TaskManager, Performance Monitor (Windows)
  * Text
    * vmstat, mpstat, pidstat, top
    * prstat (Solaris)
    * typeperf (Windows)

OS Performance Monitoring (cont.)

* Memory
  * vmstat
  * cpubar (Solaris)
  * Performance Monitor, typeperf (Windows)
* Network
  * nicstat, typeperf (Windows)
* I/O
  * iostat, iosnoop.d (Solaris)
* Other
  * sar (sysstat), kstat (Solaris), cpustat, cputrack

Performance Profiling - Types of Profilers

- Method profiler
  - Collects information about method execution times
  - Look for: internal/external method times, frequently called methods/classes etc.
- Memory profiler
  - Collects information about object creation and/or garbage collection
- Thread profiler
  - Looks for thread conflict situations

Performance Profiling - Profilers

- Method profilers
  - Oracle Solaris Studio Performance Analyzer, HPJMeter, JVisualVM
  - Free: java -agentlib:hprof=[help]|[<option>=<value>, ...], jvmstat, JRockit Runtime Analyzer
  - Commercial: JProbe, OptimizeIt!, JProfiler, YourKit
- Memory profilers
  - Oracle Solaris Studio Performance Analyzer
  - HPJMeter
  - JVisualVM
- Thread profilers
  - JVisualVM
  - Google's JChord, Java Race Finder, Mutability Detector, YourKit

A few things about the application to optimise

* A soft real-time Java application based on NetBeans RCP
* Displays plots and tracks on a digital map
* Requirement to run on SunBlade 1500 (1 CPU 1.062 GHz UltraSPARC, 1 GB RAM)

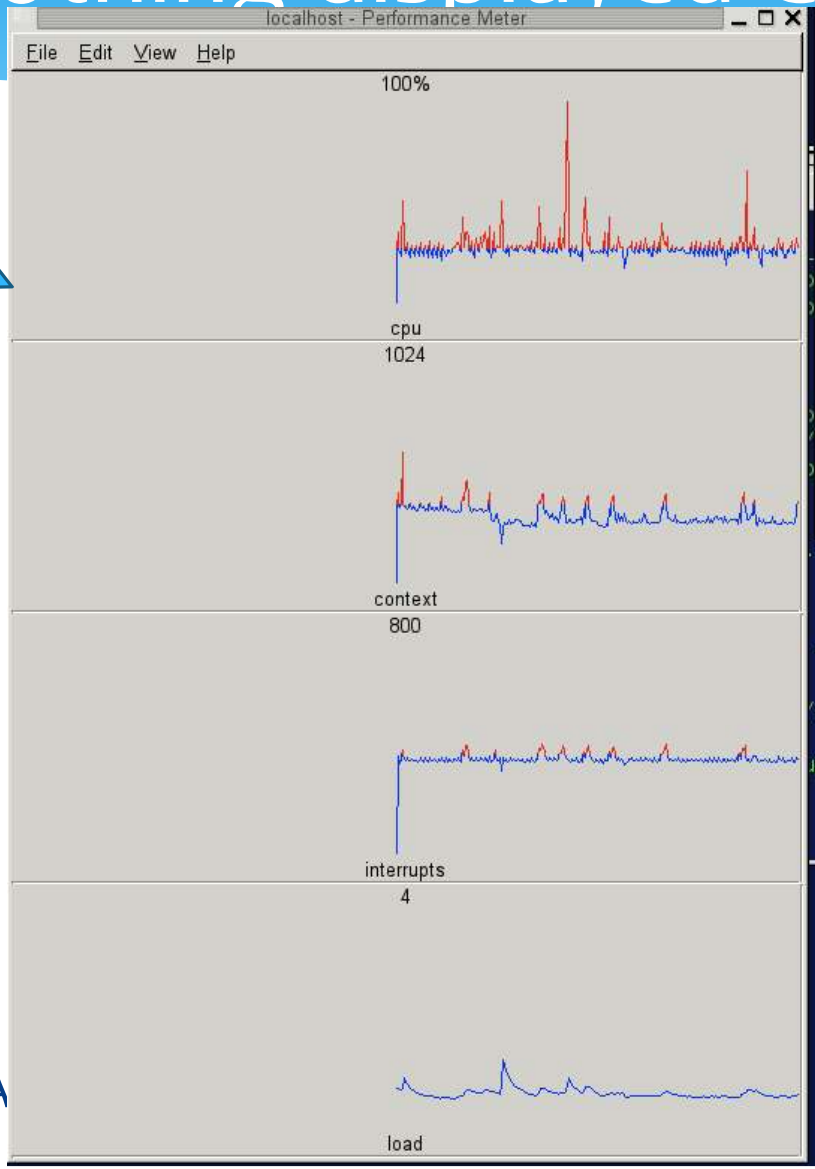
Steps executed

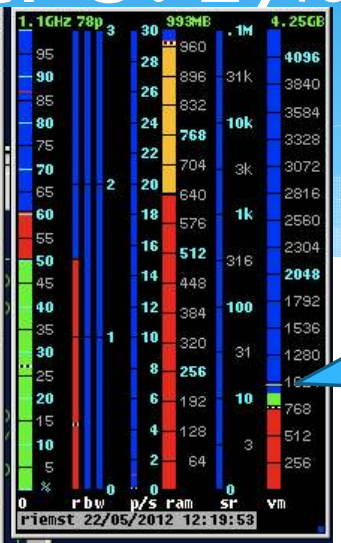
1) Research of root causes
   * Analysis of GC impact
   * Verification of lock contention
   * Hotspots in application code
2) Profile main branch application
3) Identification of hotspots
4) Optimisation of hotspots
5) Performance measurement with & without changes
6) Integration of changes
7) Restart from point 1

CPU Performance Monitoring

Monitoring Results of Main branch Nothing displayed CPU: 27%

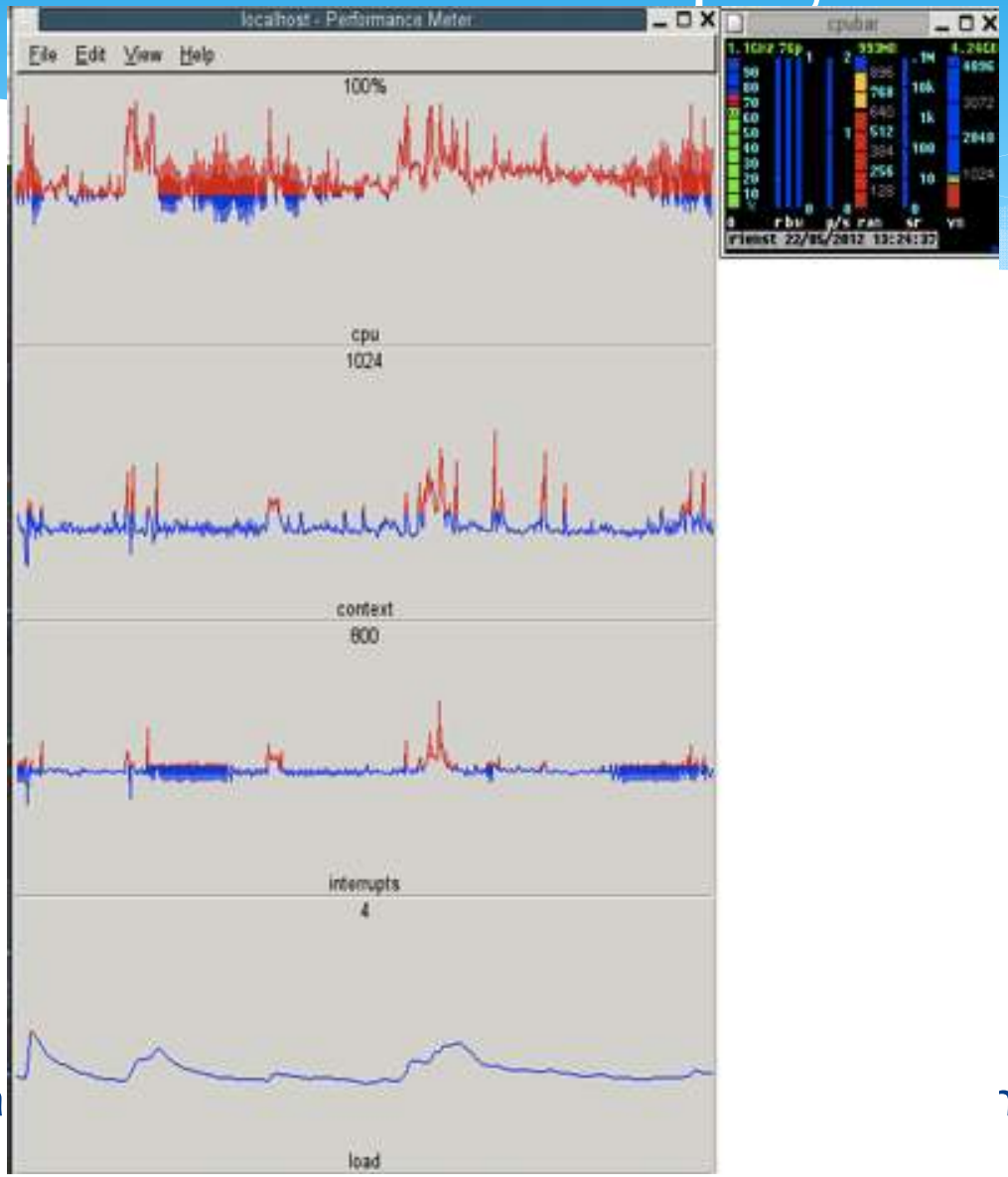
gnome-system-monitor





cpubar

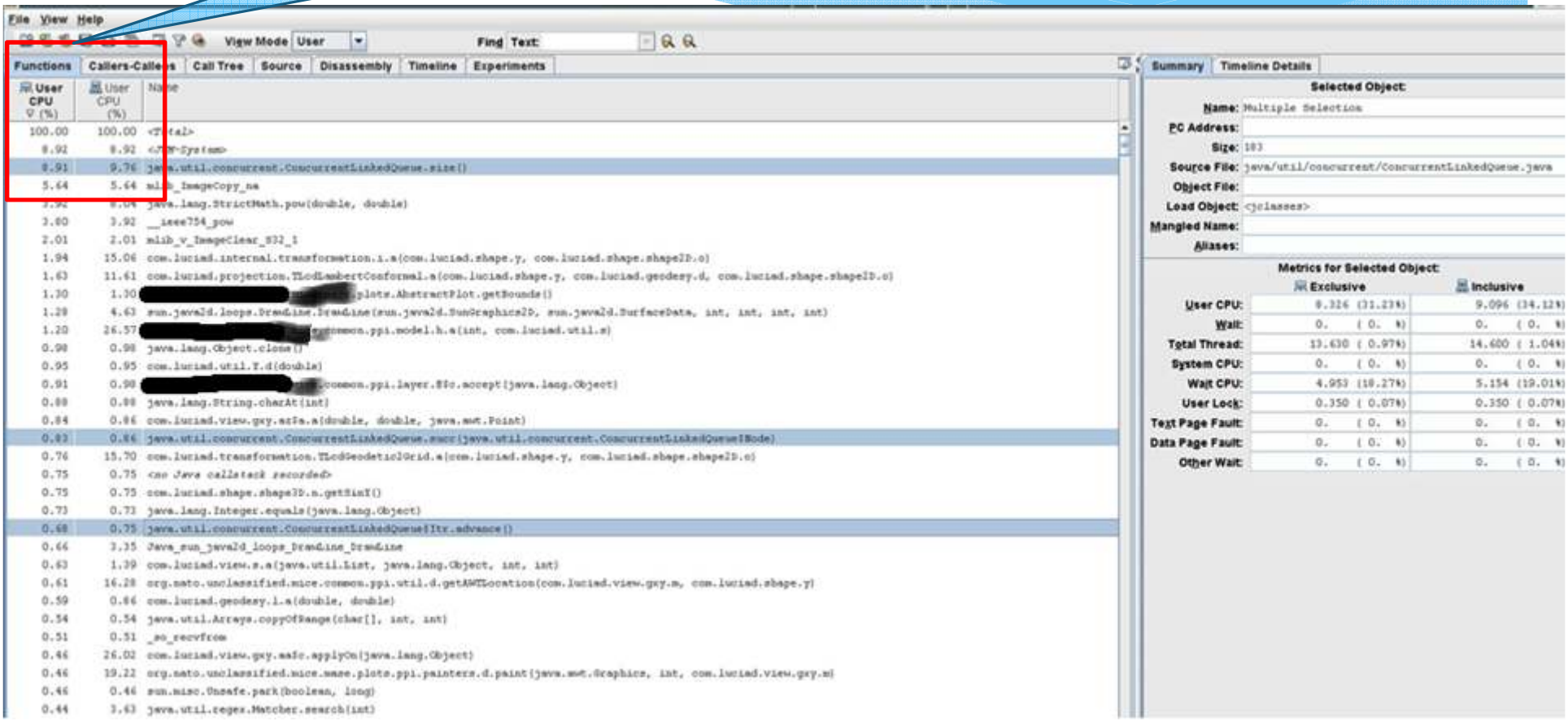
Monitoring Results of Main branch
Plots & tracks with labels displayed CPU: 64%



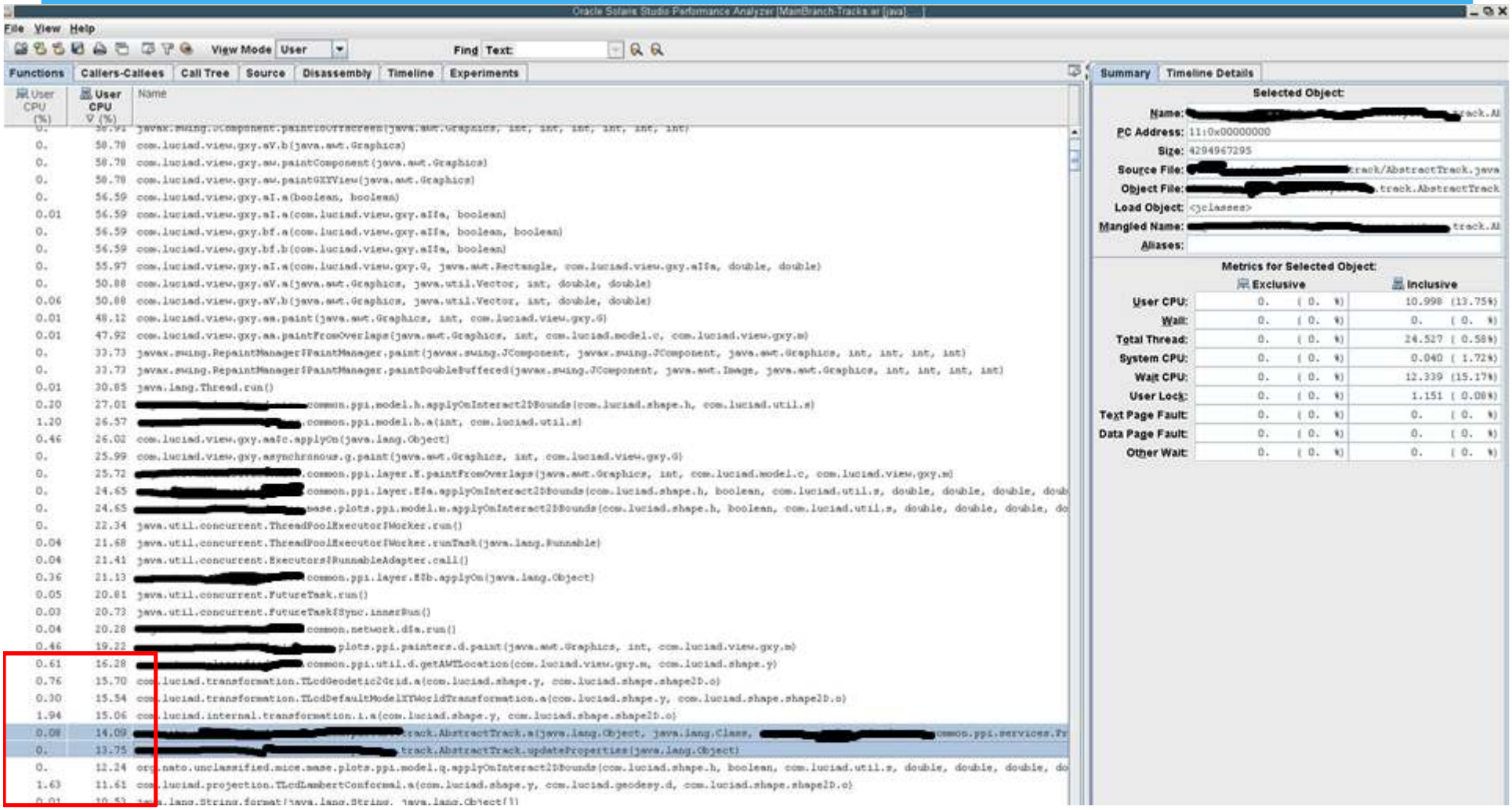
CPU Performance Profiling

Identification of Hotspot 1

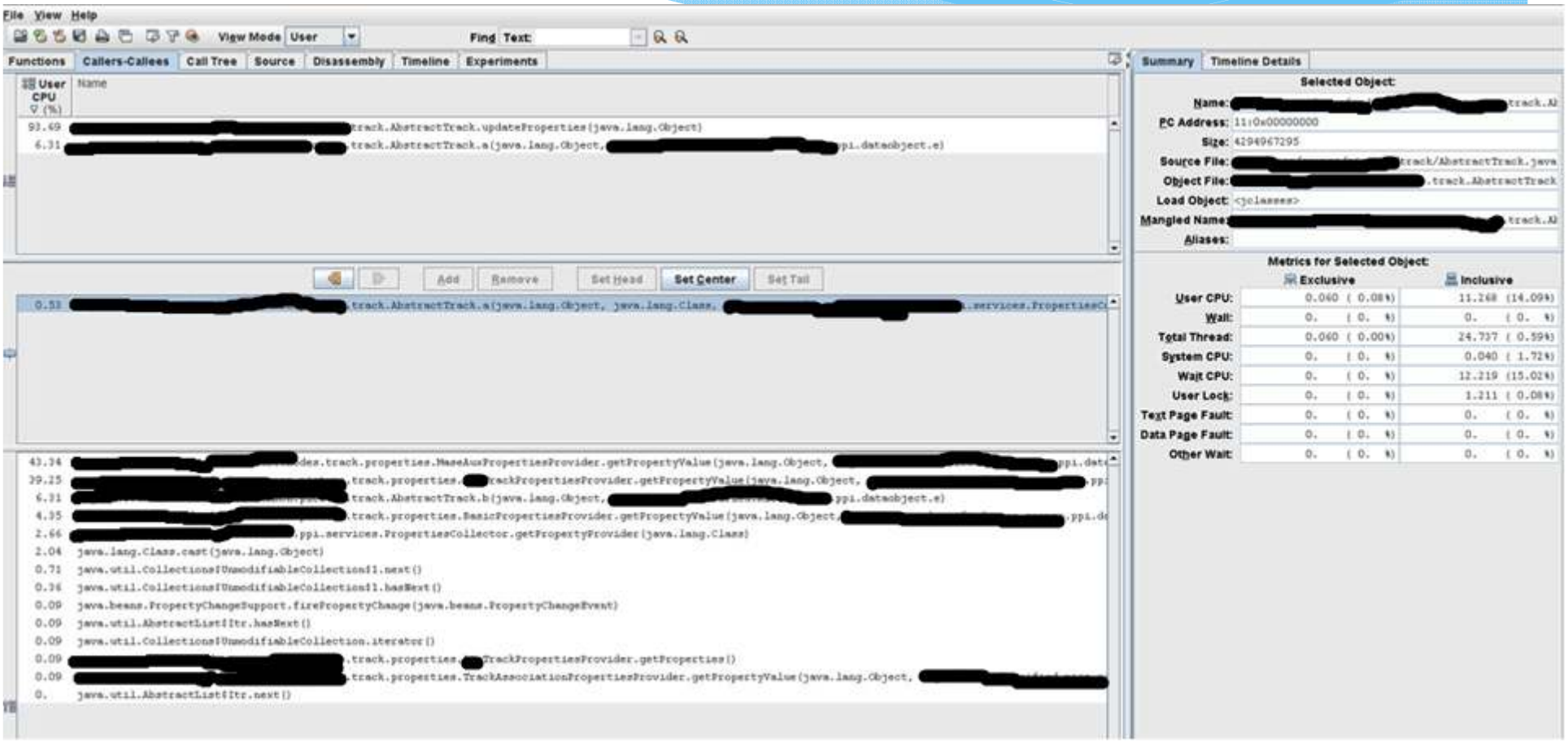
Exclusive & inclusive user CPU utilisation metrics



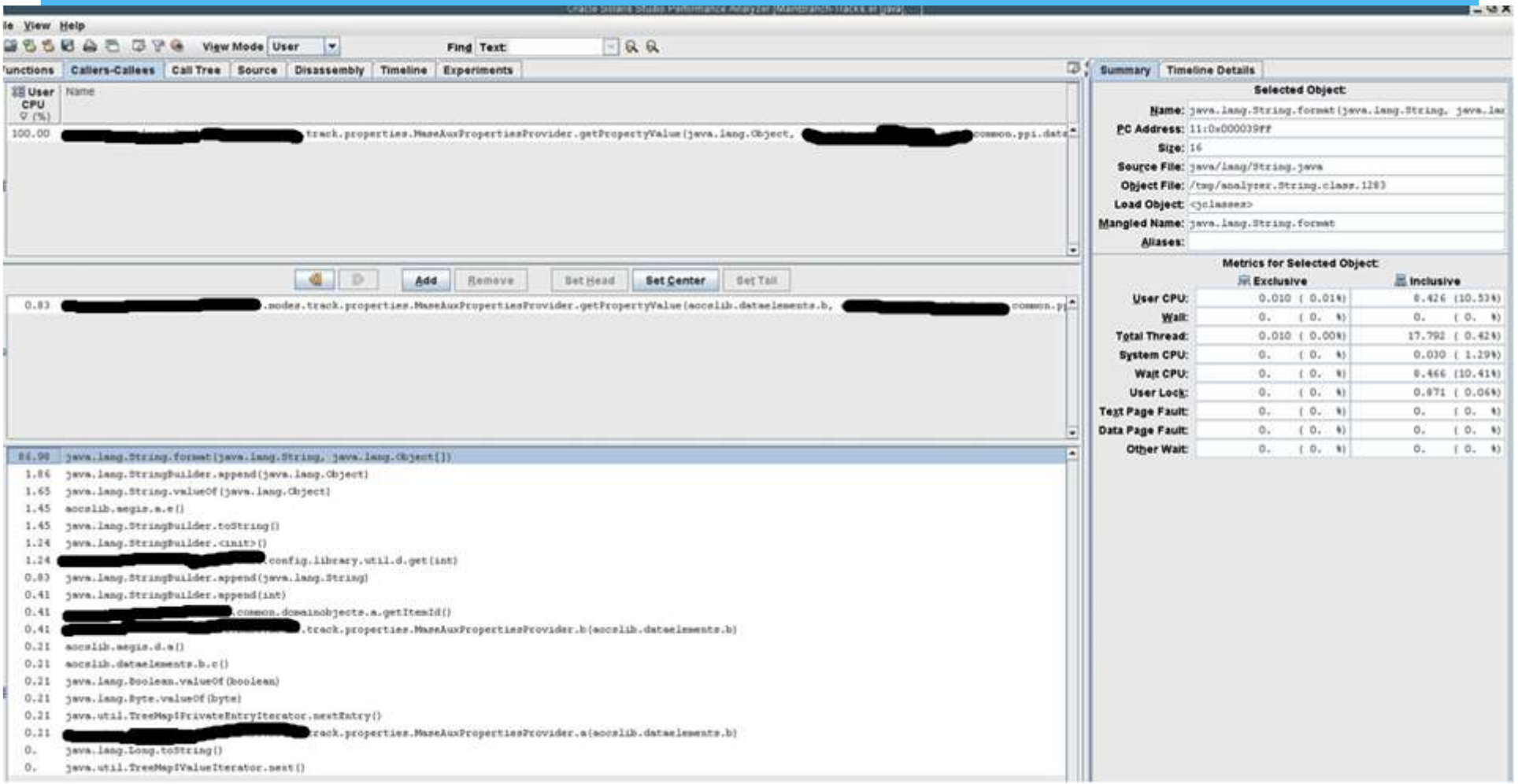
Identification of Hotspot 2



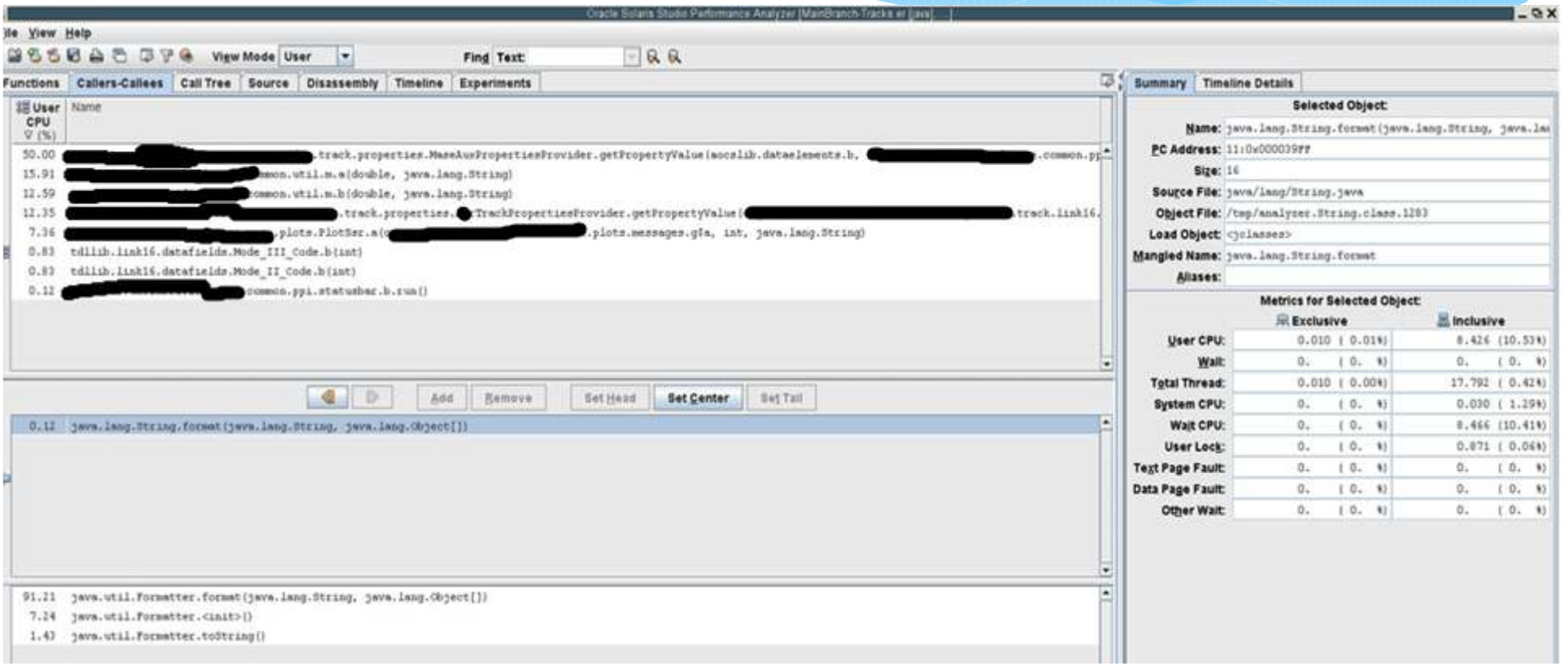
Details of Hotspot 2



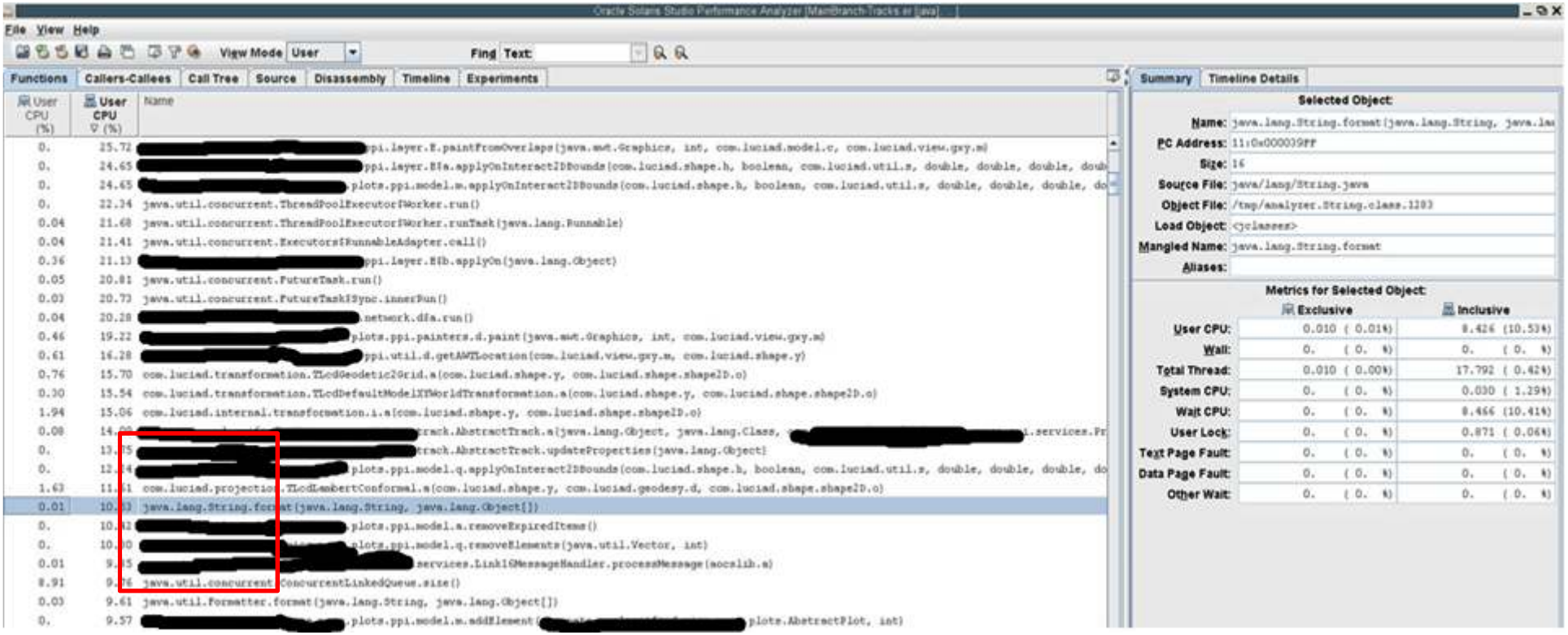
Details of Hotspot 2 (cont.)



Details of Hotspot 2 (cont.)

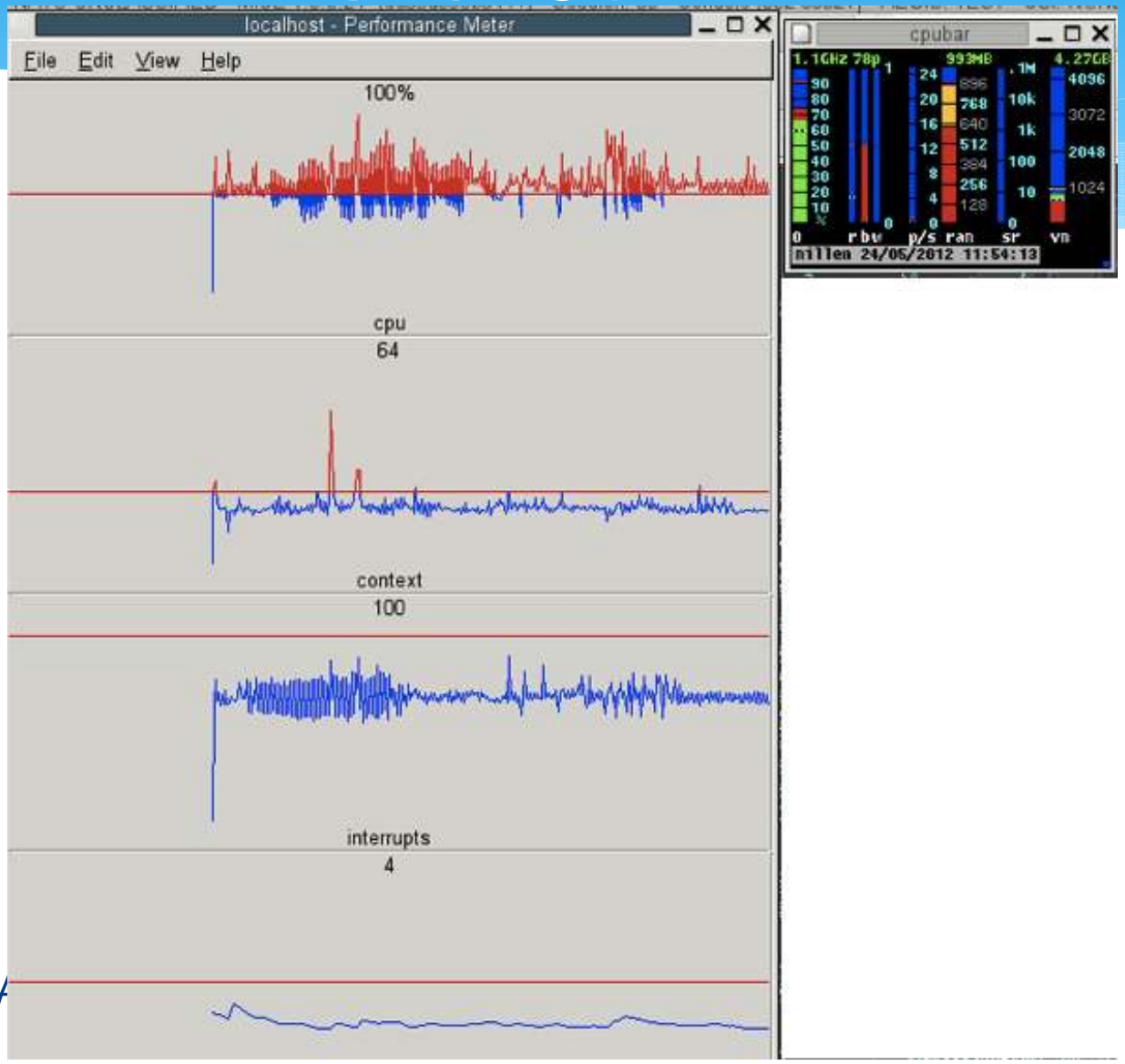


Identification of Hotspot 2 (cont.)

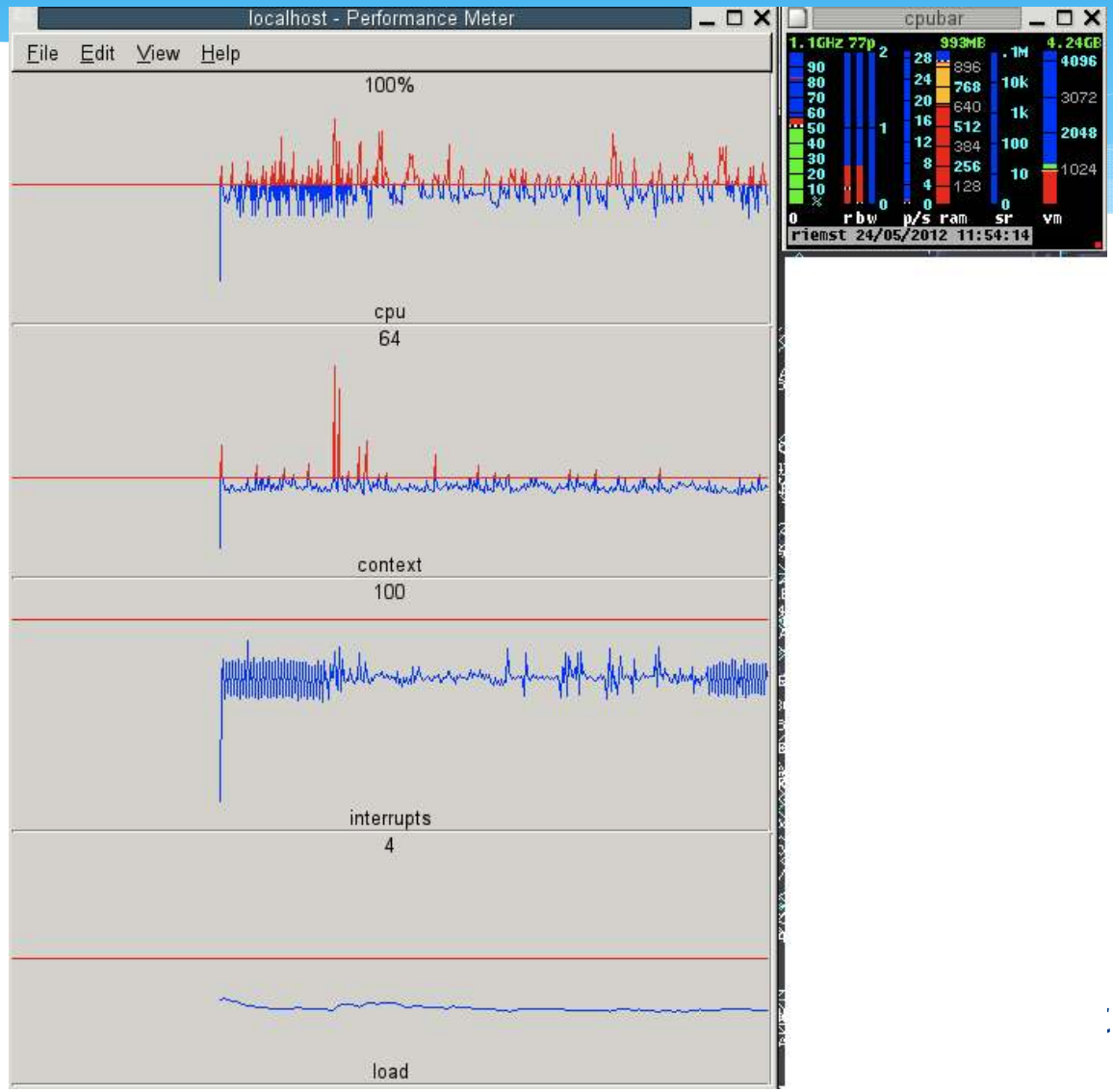


CPU Performance Tuning

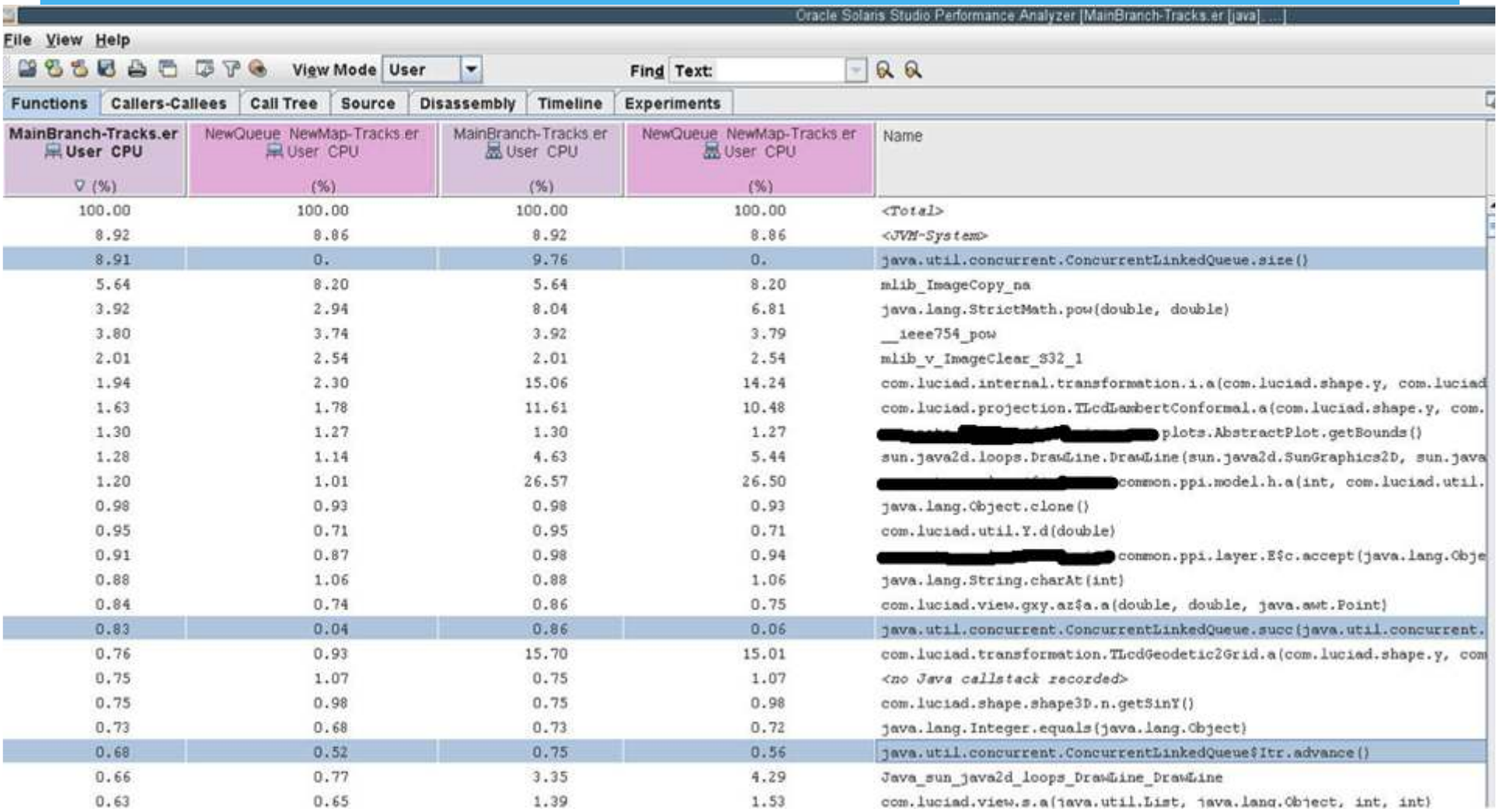
Performance before the changes
CPU: 60%



Performance after the changes
CPU: 50%

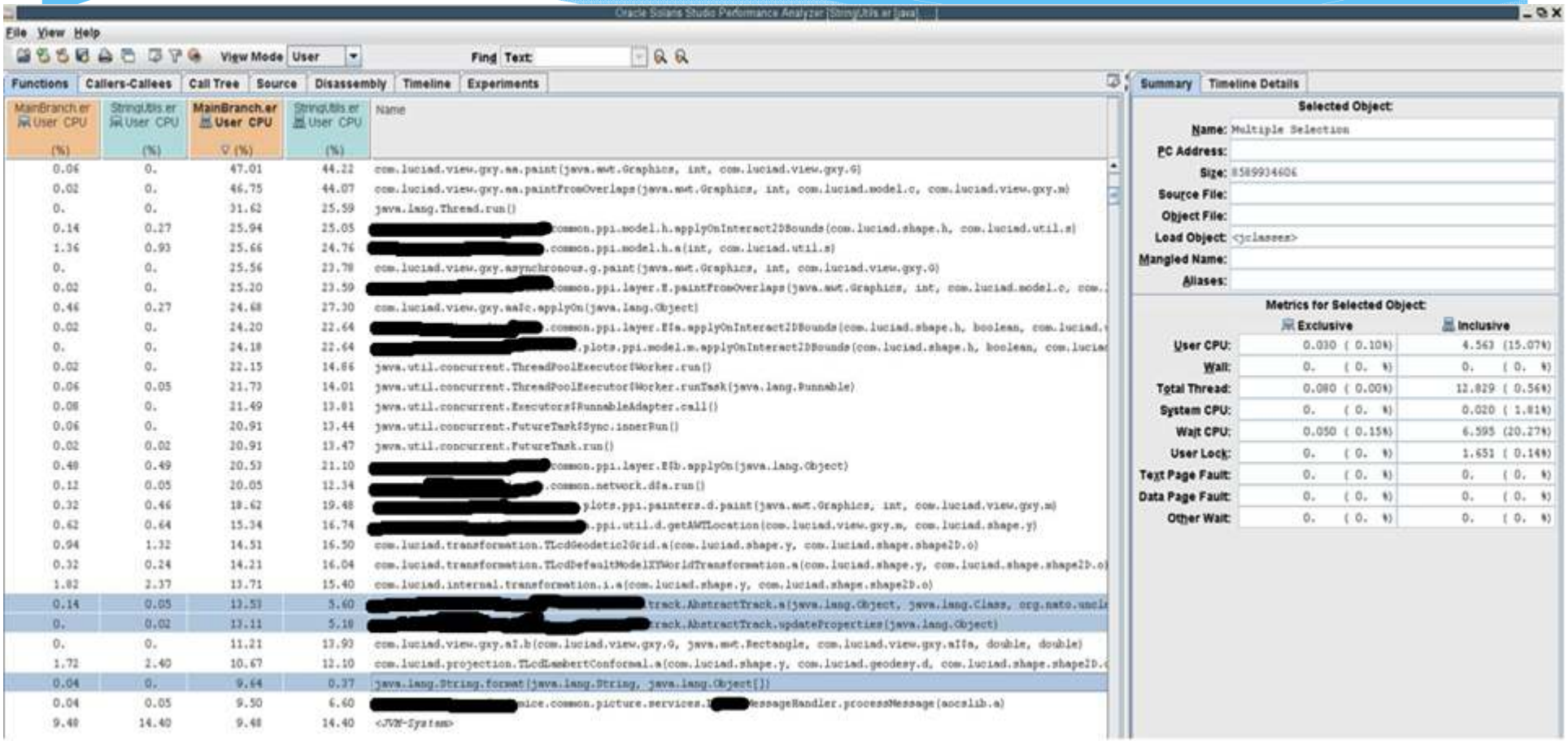


Profiling comparison 1



MainBranch-Tracks.er User CPU ▽ (%)	NewQueue NewMap-Tracks.er User CPU (%)	MainBranch-Tracks.er User CPU (%)	NewQueue NewMap-Tracks.er User CPU (%)	Name
100.00	100.00	100.00	100.00	*<Total>*
8.92	8.86	8.92	8.86	*<JVM-System>*
8.91	0.	9.76	0.	java.util.concurrent.ConcurrentLinkedQueue.size()
5.64	8.20	5.64	8.20	mlib_ImageCopy_na
3.92	2.94	8.04	6.81	java.lang.StrictMath.pow(double, double)
3.80	3.74	3.92	3.79	__ieee754_pow
2.01	2.54	2.01	2.54	mlib_v_ImageClear_S32_1
1.94	2.30	15.06	14.24	com.luciad.internal.transformation.i.a(com.luciad.shape.y, com.luciad
1.63	1.78	11.61	10.48	com.luciad.projection.TLcdLambertConformal.a(com.luciad.shape.y, com.
1.30	1.27	1.30	1.27	[redacted]plots.AbstractPlot.getBounds()
1.28	1.14	4.63	5.44	sun.java2d.loops.DrawLine.DrawLine(sun.java2d.SunGraphics2D, sun.java
1.20	1.01	26.57	26.50	[redacted]common.ppi.model.h.a(int, com.luciad.util.
0.98	0.93	0.98	0.93	java.lang.Object.clone()
0.95	0.71	0.95	0.71	com.luciad.util.Y.d(double)
0.91	0.87	0.98	0.94	[redacted]common.ppi.layer.E$c.accept(java.lang.Obje
0.88	1.06	0.88	1.06	java.lang.String.charAt(int)
0.84	0.74	0.86	0.75	com.luciad.view.gxy.az$a.a(double, double, java.awt.Point)
0.83	0.04	0.86	0.06	java.util.concurrent.ConcurrentLinkedQueue.succ(java.util.concurrent.
0.76	0.93	15.70	15.01	com.luciad.transformation.TLcdGeodetic2Grid.a(com.luciad.shape.y, com
0.75	1.07	0.75	1.07	*<no Java callstack recorded>*
0.75	0.98	0.75	0.98	com.luciad.shape.shape3D.n.getSinY()
0.73	0.68	0.73	0.72	java.lang.Integer.equals(java.lang.Object)
0.68	0.52	0.75	0.56	java.util.concurrent.ConcurrentLinkedQueue$Itr.advance()
0.66	0.77	3.35	4.29	Java_sun_java2d_loops_DrawLine_DrawLine
0.63	0.65	1.39	1.53	com.luciad.view.s.a(java.util.List, java.lang.Object, int, int)

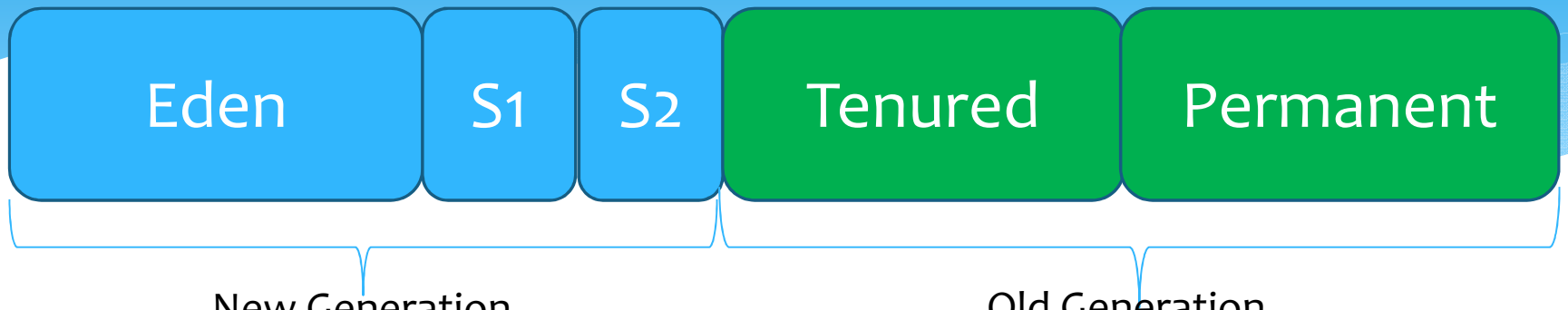
Profiling comparison 2



Memory Performance Profiling/Tuning

Goal is not to improve the CPU usage but to mitigate the impact of the GC on the real-time behaviour of the application

Generational Spaces & Garbage Collectors



New Generation GCs	Old Generation GCs
Copying collector (< Java 5) -XX:+UseSerialGC	Mark-Sweep collector (< Java 5) -XX:+UseSerialGC
Parallel copying collector (≥ Java 5) -XX:+UseParNewGC	Parallel Scavenge MarkSweep collector -XX:+UseParallelOldGC
Parallel scavenge collector (≥ Java 5, >10GB heap space) -XX:+UseParallelGC	Concurrent Mark Sweep collector (≥ Java 6) -XX:+UseConcMarkSweepGC
G1 young generation (≥ Java 7) -XX:+UseG1GC	G1 mixed generation (≥ Java 7) -XX:+UseG1GC

Recommended Settings

http://randomlyrr.blogspot.be/2012/03/java-tuning-in-nutshell-part-1.html

- `-Xmx = -Xms`
- `-XX:PermSize = -XX:MaxPermSize`
- Use `-Xmn` instead of `-XX:NewSize` and `-XX:MaxNewSize`
- Disable adaptive sizing of generations `-XX:-UseAdaptiveSizePolicy` if `-XX:SurvivorRatio` is specified
- Use `-XX:+UseConcMarkSweepGC`
- `40<-XX:CMSInitiatingOccupancyFraction<70` and always use `-XX:+UseCMSInitiatingOccupancyOnly` with it

Application's Runtime arguments

```
run.args.extra =
-Xms256m -Xmx768m -Xincgc
-XX:+UseConcMarkSweepGC
-XX:SurvivorRatio=3
-XX:-UseAdaptiveSizePolicy
-XX:CMSInitiatingOccupancyFraction=60
-XX:+UseCMSInitiatingOccupancyOnly
```

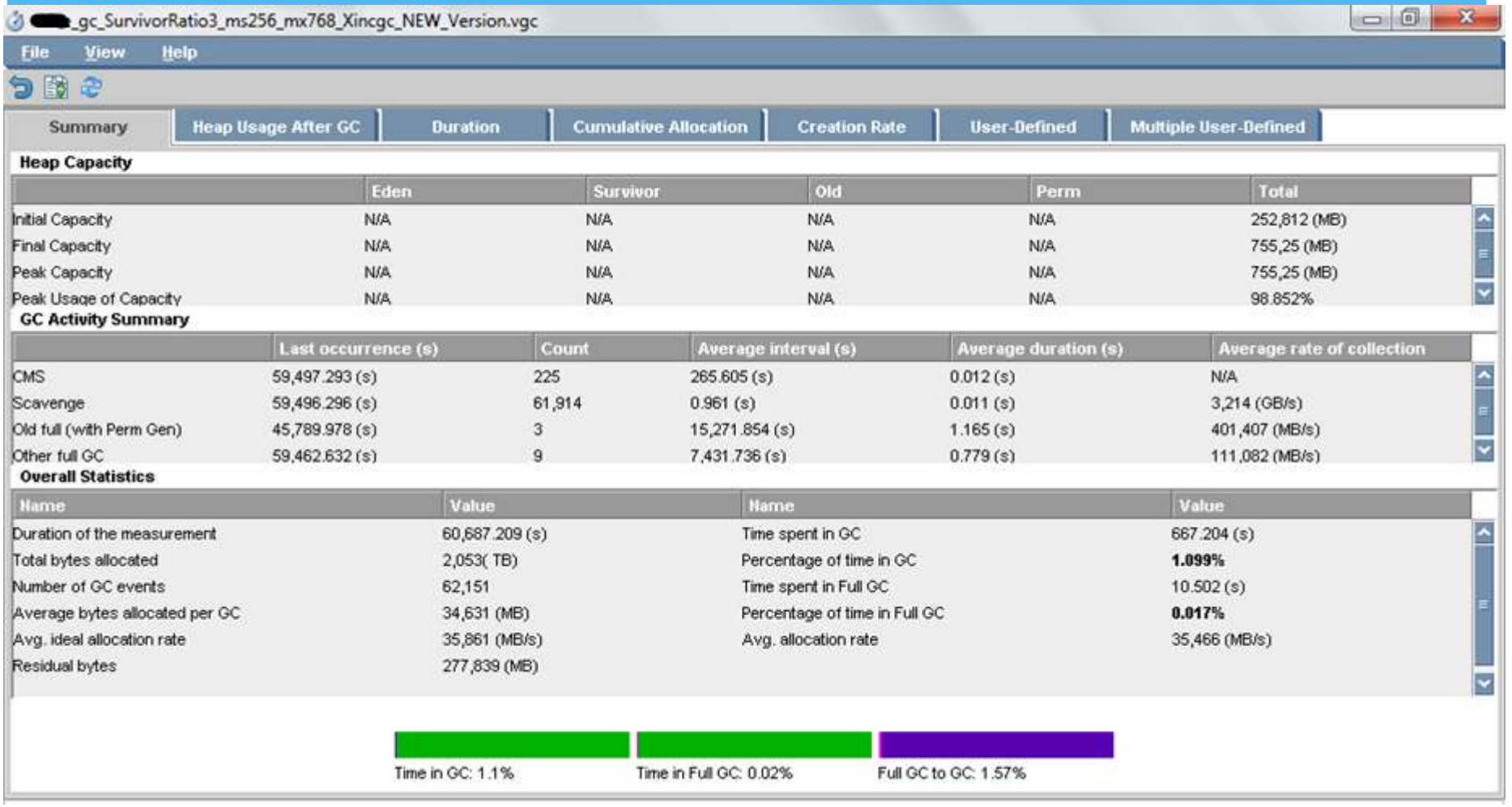
Init heap size

Max heap size

Eden=76,8 Mb, S1,S2= 25,6 Mb

CMS: Concurrent MarkSweep

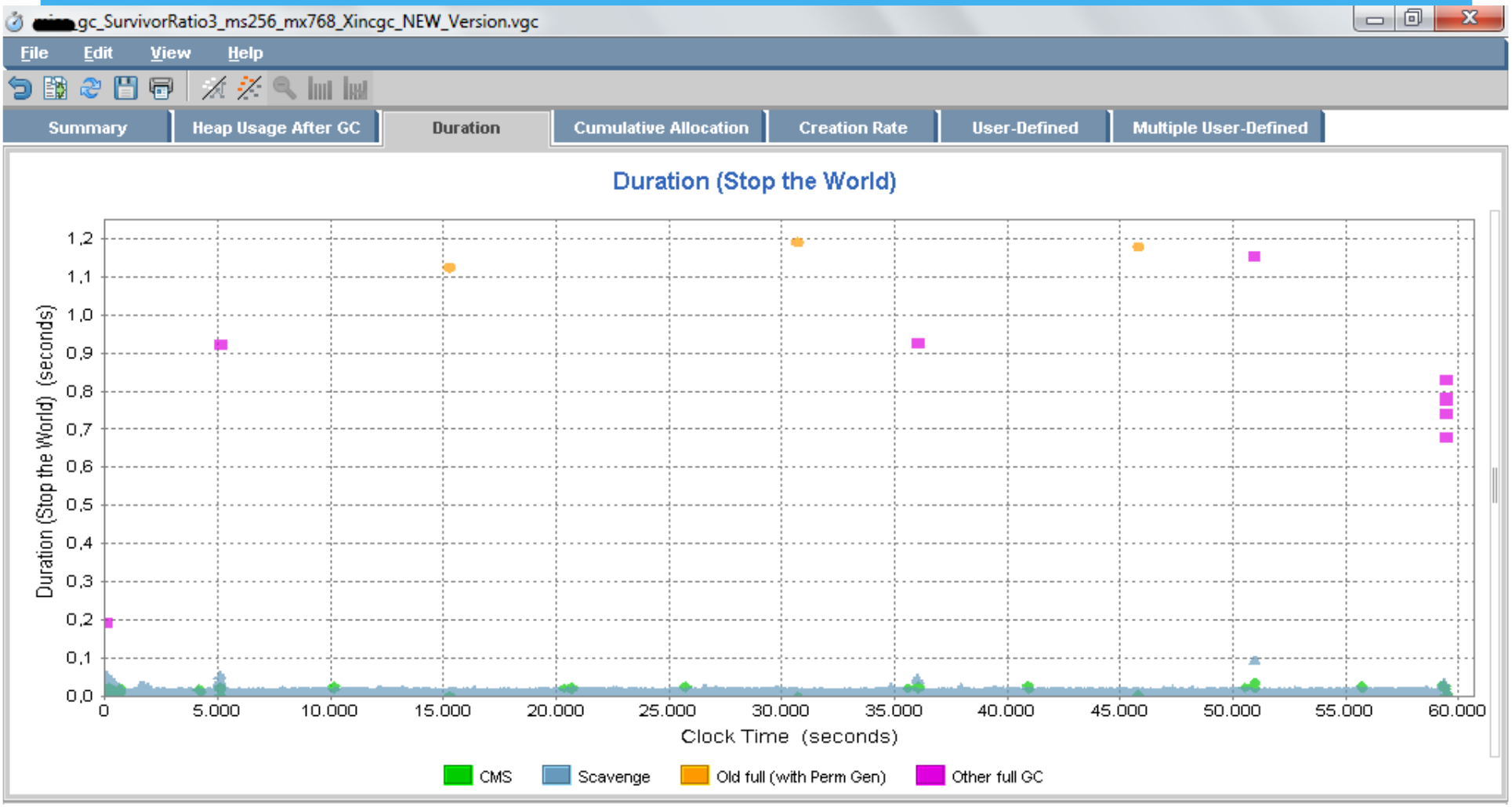
HPJMeter


[redacted]_gc_SurvivorRatio3_ms256_mx768_Xincgc_NEW_Version.vgc
File View Help
Summary
Heap Usage After GC
Duration
Cumulative Allocation
Creation Rate
User-Defined
Multiple User-Defined
Heap Capacity
Eden Survivor Old Perm Total
Initial Capacity N/A N/A N/A N/A 252,812 (MB)
Final Capacity N/A N/A N/A N/A 755,25 (MB)
Peak Capacity N/A N/A N/A N/A 755,25 (MB)
Peak Usage of Capacity N/A N/A N/A N/A 98.852%
GC Activity Summary
Last occurrence (s) Count Average interval (s) Average duration (s) Average rate of collection
CMS 59,497.293 (s) 225 265.605 (s) 0.012 (s) N/A
Scavenge 59,496.296 (s) 61,914 0.961 (s) 0.011 (s) 3,214 (GB/s)
Old full (with Perm Gen) 45,789.978 (s) 3 15,271.854 (s) 1.165 (s) 401,407 (MB/s)
Other full GC 59,462.632 (s) 9 7,431.736 (s) 0.779 (s) 111,082 (MB/s)
Overall Statistics
Name Value Name Value
Duration of the measurement 60,687.209 (s) Time spent in GC 667.204 (s)
Total bytes allocated 2,053(TB) Percentage of time in GC 1.099%
Number of GC events 62,151 Time spent in Full GC 10.502 (s)
Average bytes allocated per GC 34,631 (MB) Percentage of time in Full GC 0.017%
Avg. ideal allocation rate 35,861 (MB/s) Avg. allocation rate 35,466 (MB/s)
Residual bytes 277,839 (MB)
Time in GC: 1.1%
Time in Full GC: 0.02%
Full GC to GC: 1.57%

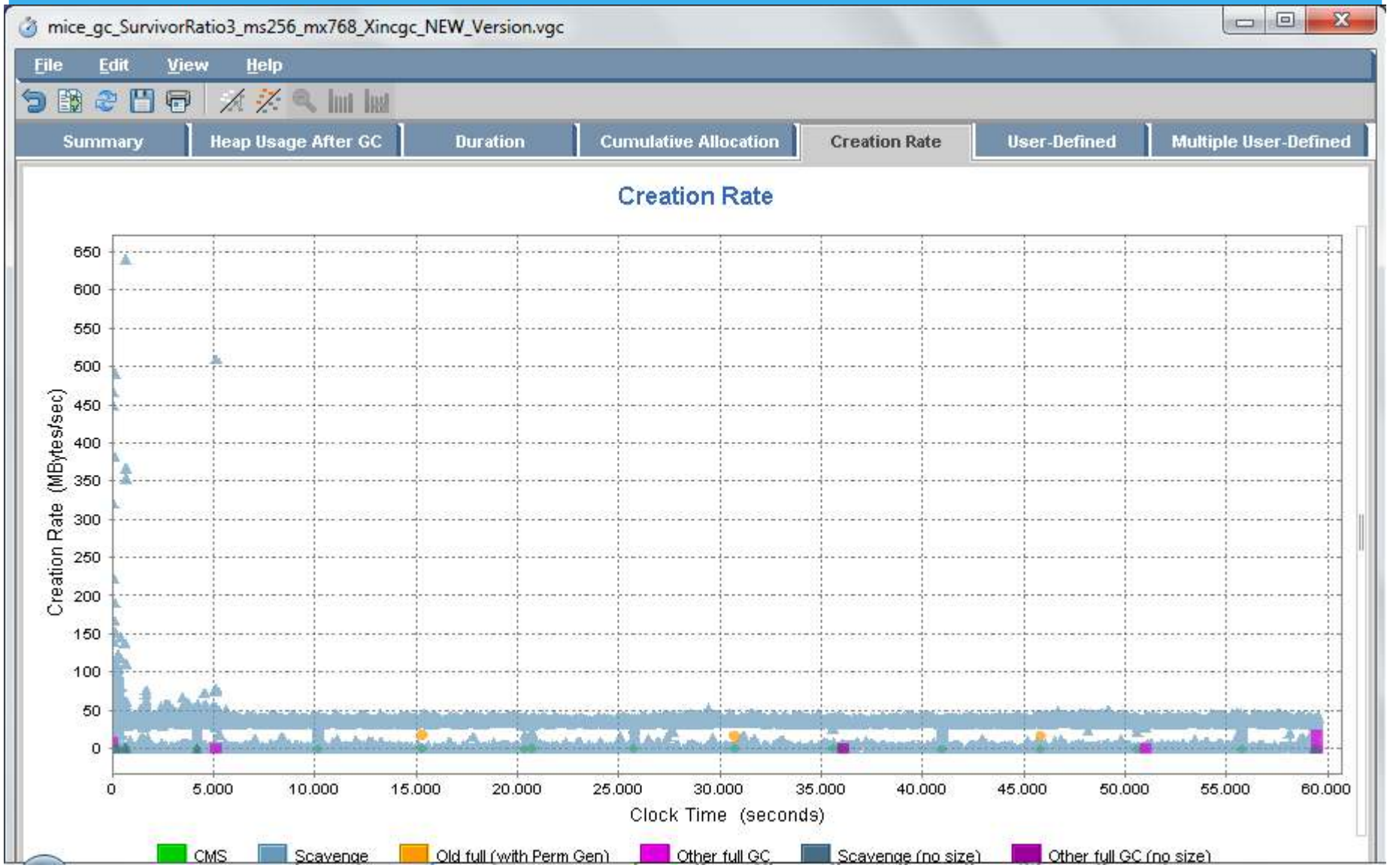
HPJMeter


_gc_SurvivorRatio3_ms256_mx768_Xincgc_NEW_Version.vgc
File
Edit
View
Help
Summary
Heap Usage After GC
Duration
Cumulative Allocation
Creation Rate
User-Defined
Multiple User-Defined
Heap Usage After GC
Heap Usage After GC (MBytes)
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
50.000
55.000
60.000
Clock Time (seconds)
CMS
Scavenge
Old full (with Perm Gen)
Other full GC

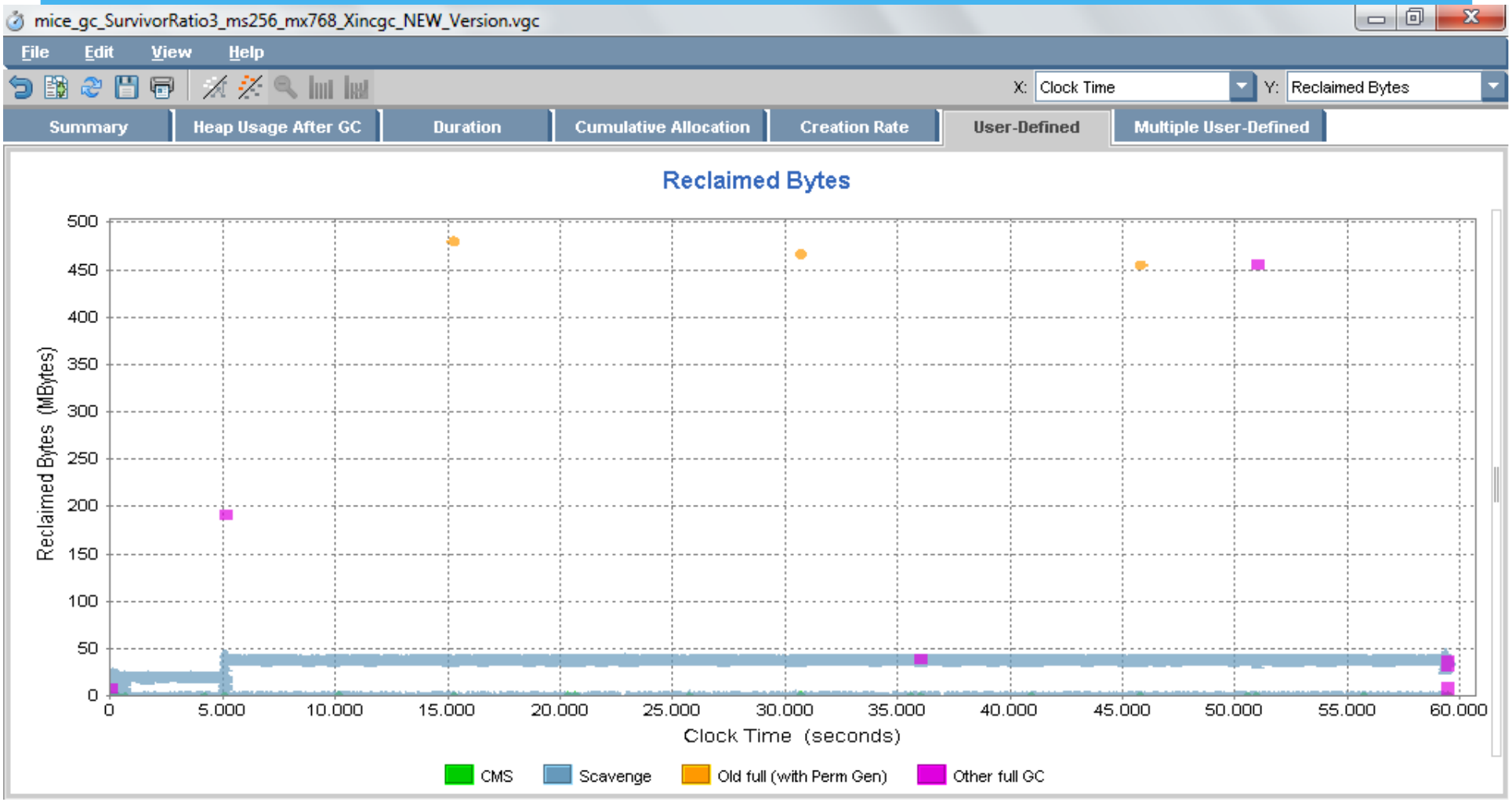
HPJMeter


[redacted]gc_SurvivorRatio3_ms256_mx768_Xincgc_NEW_Version.vgc
File
Edit
View
Help
Summary
Heap Usage After GC
Duration
Cumulative Allocation
Creation Rate
User-Defined
Multiple User-Defined
Duration (Stop the World)
Duration (Stop the World) (seconds)
1,2
1,1
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
50.000
55.000
60.000
Clock Time (seconds)
CMS
Scavenge
Old full (with Perm Gen)
Other full GC

HPJMeter


mice_gc_SurvivorRatio3_ms256_mx768_Xincgc_NEW_Version.vgc
File
Edit
View
Help
Summary
Heap Usage After GC
Duration
Cumulative Allocation
Creation Rate
User-Defined
Multiple User-Defined
Creation Rate
Creation Rate (MBytes/sec)
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
50.000
55.000
60.000
Clock Time (seconds)
CMS
Scavenge
Old full (with Perm Gen)
Other full GC
Scavenge (no size)
Other full GC (no size)

HPJMeter


mice_gc_SurvivorRatio3_ms256_mx768_Xincgc_NEW_Version.vgc
File
Edit
View
Help
X: Clock Time
Y: Reclaimed Bytes
Summary
Heap Usage After GC
Duration
Cumulative Allocation
Creation Rate
User-Defined
Multiple User-Defined
Reclaimed Bytes
Reclaimed Bytes (MBytes)
500
450
400
350
300
250
200
150
100
50
0
0
5.000
10.000
15.000
20.000
25.000
30.000
35.000
40.000
45.000
50.000
55.000
60.000
Clock Time (seconds)
CMS
Scavenge
Old full (with Perm Gen)
Other full GC

Thread Profiling/Tuning ?

References

* Hunt C. & John B. (2011), *Java Performance*, Prentice Hall
* Kabutz H. (2011), *Java Master's course slides*
* http://www.youtube.com/watch?v=VGQAL9aUKfs
* http://randomlyrr.blogspot.be/2012/03/java-tuning-in-nutshell-part-1.html
* http://www.fasterj.com/articles/oraclecollectors1.shtml
* http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

Questions