Performance Beyond Throughput: An OpenJ9 Case Study

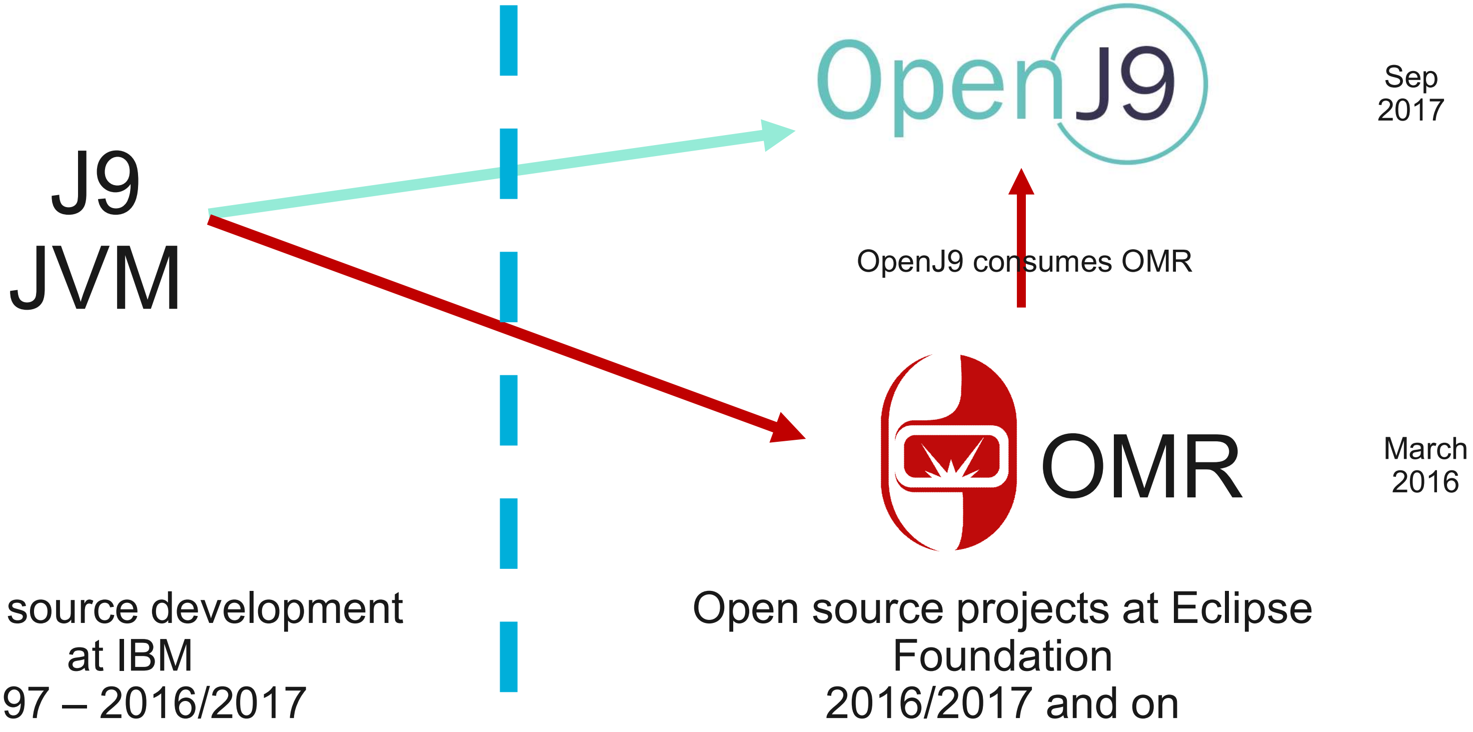
Marius Pirvu, IBM Runtime Technologies
Nov 13, 2017 - mpirvu@ca.ibm.com

IBM Runtime Technologies

Important disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
  - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Eclipse OpenJ9: an open source JVM



J9
JVM

OpenJ9

Sep
2017

OpenJ9 consumes OMR

OMR

March
2016

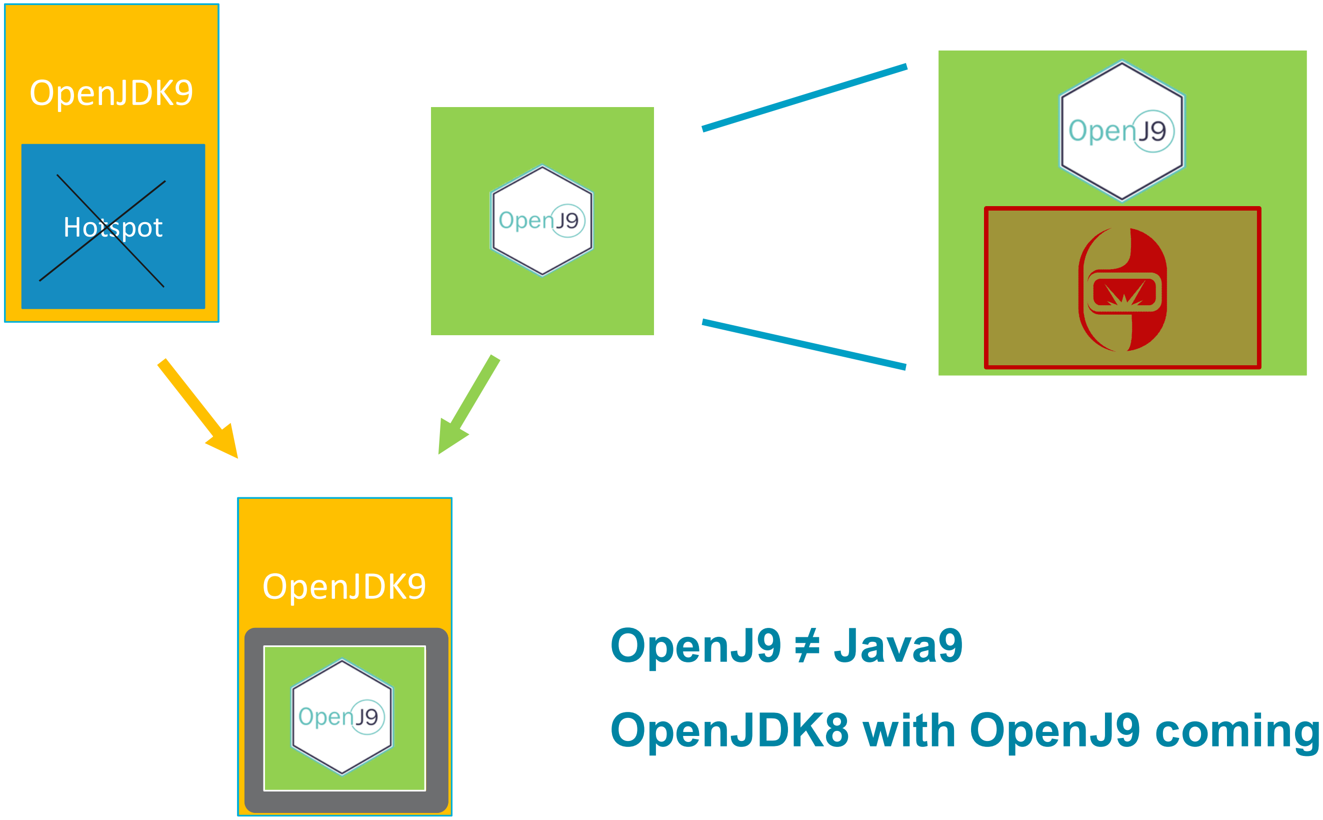
Closed source development
at IBM
1997 – 2016/2017

Open source projects at Eclipse
Foundation
2016/2017 and on

Why use Eclipse OpenJ9?

- Very open. Dual license: Eclipse Public License v2.0 and Apache 2.0
- Very easy for anyone to contribute
  - github repositories:
    - https://github.com/eclipse/openj9
    - https://github.com/eclipse/omr
  - Prebuilt binaries:
    - https://adoptopenjdk.net/nightly.html?variant=openjdk9-openj9
- Performance
  - Excellent performance for a wide variety of metrics important in the cloud
  - Hardware exploitation for x86, Power and Z mainframes
  - Focus on large applications rather than microbenchmarks

OpenJDK9 with OpenJ9



OpenJ9 ≠ Java9

OpenJDK8 with OpenJ9 coming soon!

Performance is about more than just throughput

- Performance means different things to different people
- OpenJ9 pays attention to many other metrics important to customers:
  - start-up time
  - footprint
  - ramp-up
  - response time
  - CPU
- Different goals → different design decisions
- Must keep a balance → make sensible trade-offs

Agenda

- Start-up time – 37% improvement
- Footprint – 44-60% improvement
- Behavior at idle – 55% improvement
- Ramp-up in a resource constrained environment
- Response time – 10x improvement
- Performance monitoring tools

Start-up time

- Start-up time == time needed for your server application to become operational
- Important for:
  - developers
  - scaling out operations
  - outages (planned or not)
- General characteristics of a start-up phase
  - A fair amount of class loading
  - A large amount of interpretation activity (jitting takes time!)
- OpenJ9 solutions
  - Shared class cache technology and dynamic Ahead-of-Time (AOT) compilation
  - Specialized running mode: -Xquickstart

Eclipse OpenJ9 shared class cache technology

- Memory mapped file used to cache:
  - ROM classes (pre-processed .class files)
  - AOT compiled code
  - Interpreter profiling data
- Population of the cache happens naturally and transparently at runtime
  - Distinction between ‘cold’ and ‘warm’ runs
- Enabled with **–Xshareclasses**
- Dynamic AOT compilation
  - Relocatable format
  - AOT loads are ~100 times faster than JIT compilations
  - More generic code → slightly less optimized
    - Generate AOT code only during start-up
    - Recompilation helps bridge the gap

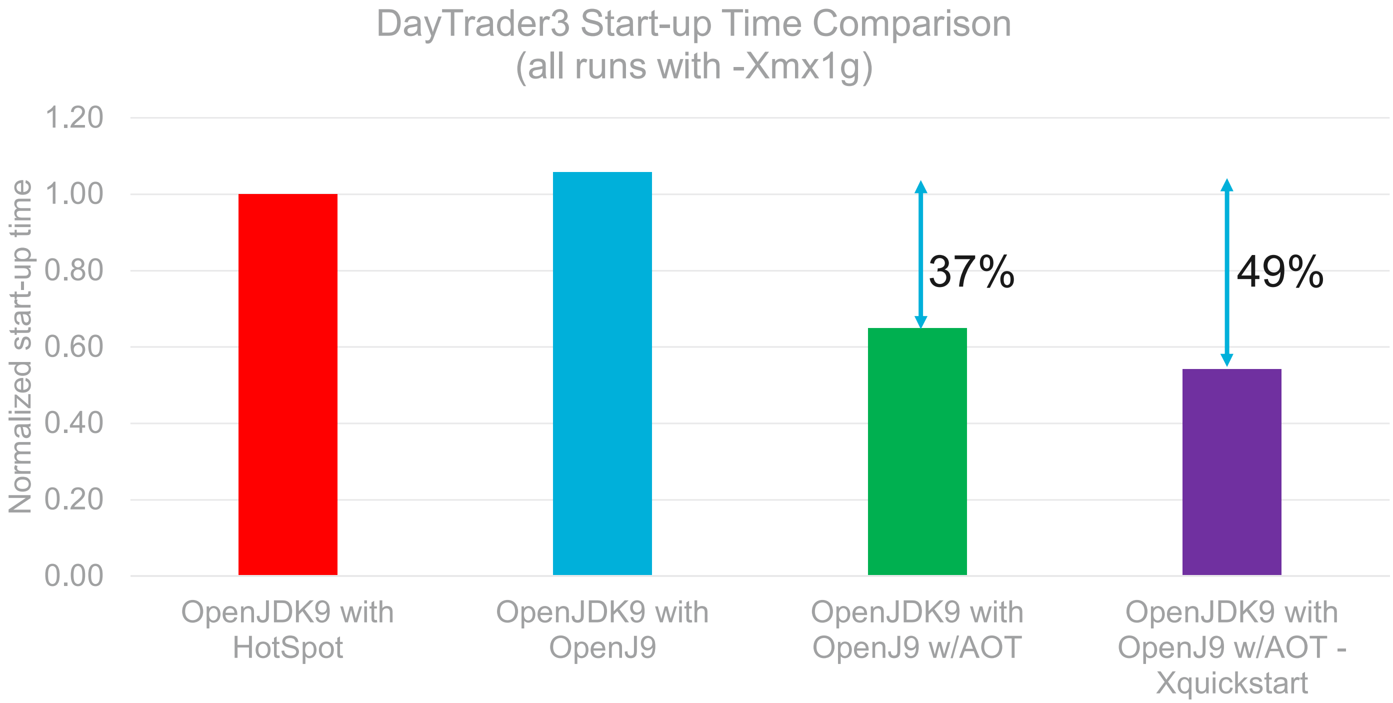
-Xquickstart mode

- Use cases
  - User cares a lot about start-up time
  - Very short running applications
  - Interactive, graphical applications

- Under the hood
  - Cheaper JIT compilations, but less optimized code
  - Interpreter profiler is disabled

- Somewhat similar to “-client” from HotSpot

Start-up performance with Eclipse OpenJ9



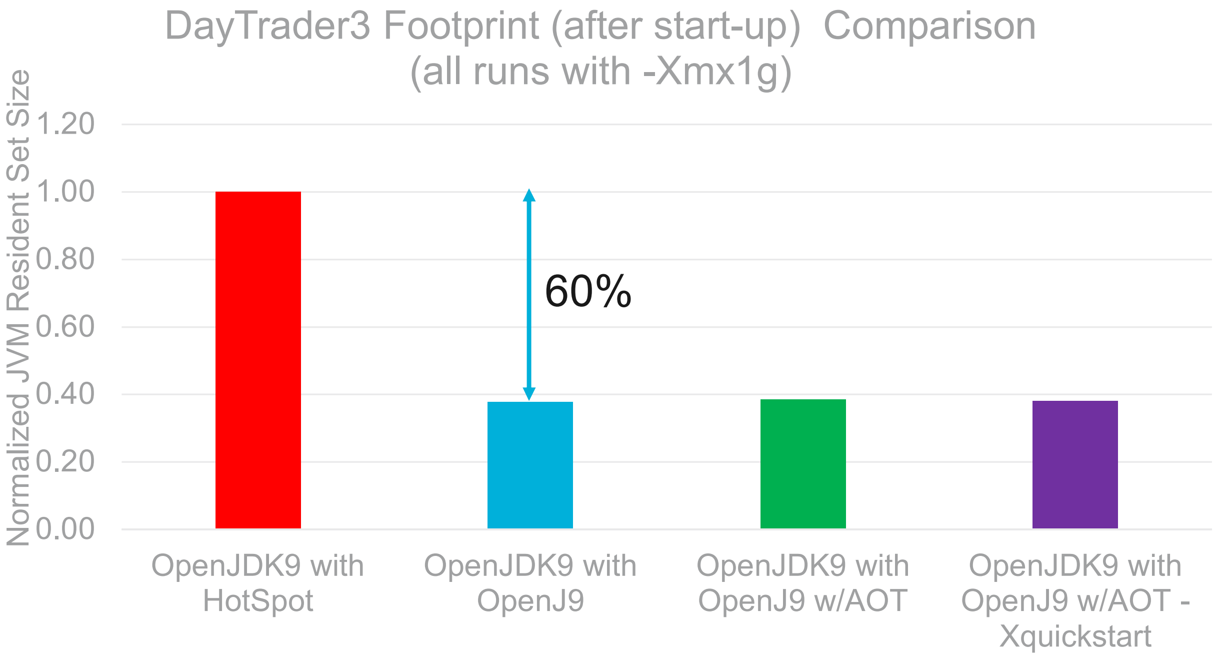
Benchmark: https://github.com/WASdev/sample.daytrader3
More details: https://github.com/eclipse/openj9-website/blob/master/benchmark/daytrader3.md

Footprint

- Myth: machines have plenty of RAM, so optimizing for footprint is not worthwhile
- Reality: application footprint is very important to:
  - Cloud users: pay for resources
  - Cloud providers: higher app density means lower operational costs
- Trends:
  - Virtualization → big machines partitioned into many smaller VM guests
  - Microservices → increased memory usage; native JVM footprint matters
- Distinction between:
  - On disk image size – relevant for Cloud Foundry
  - Virtual memory footprint – relevant for 32-bit applications
  - Physical memory footprint (RSS)

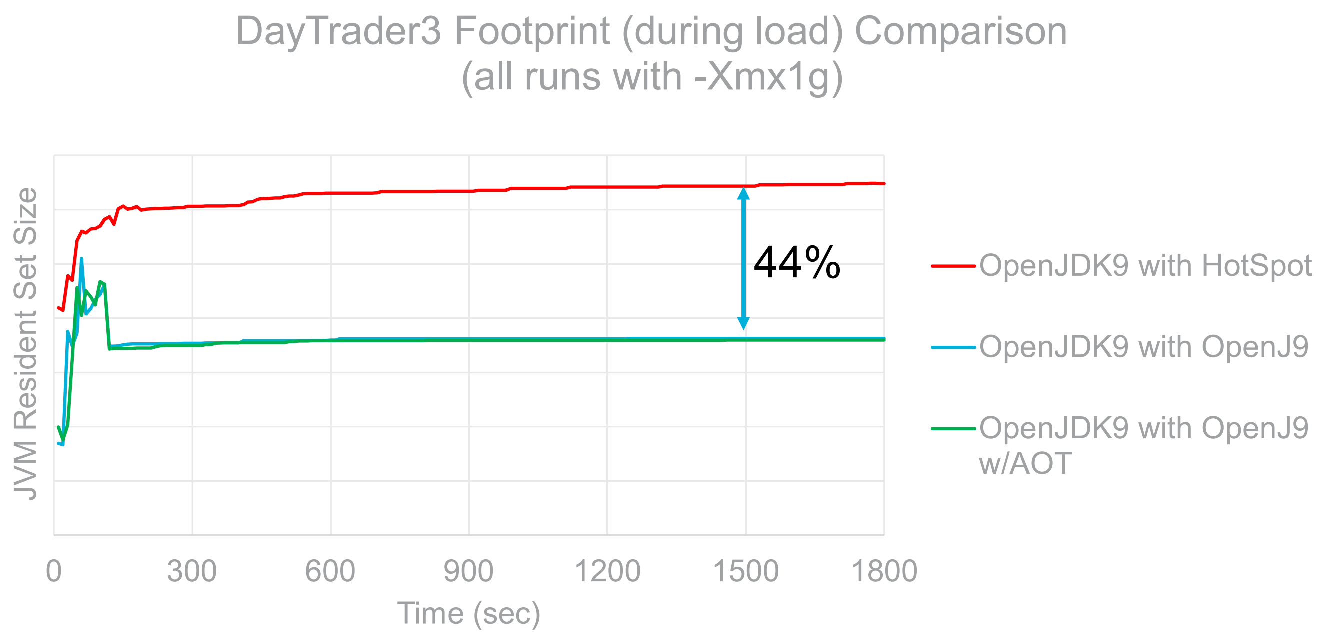
In the cloud footprint is king

Footprint after start-up comparison



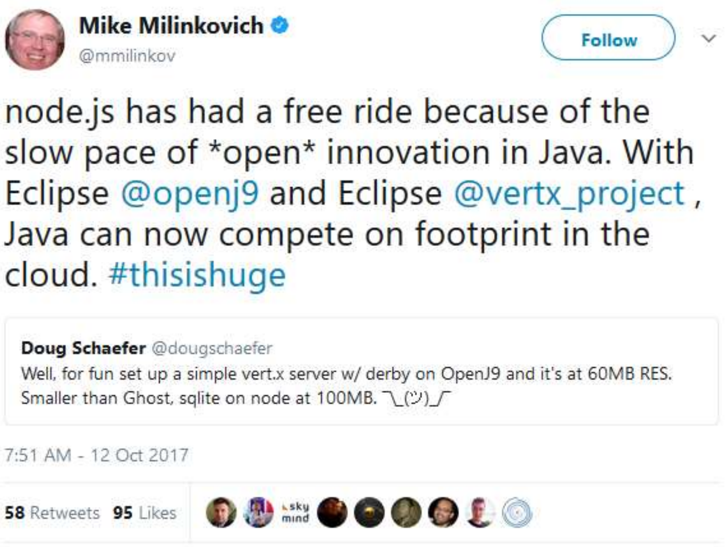
- After start-up, OpenJ9 uses 60% less physical memory than HotSpot

Footprint during load comparison



- During load, OpenJ9 uses 44% less physical memory than HotSpot
- Further savings when multiple JVMs connect to the same shared class cache

Footprint Testimonials



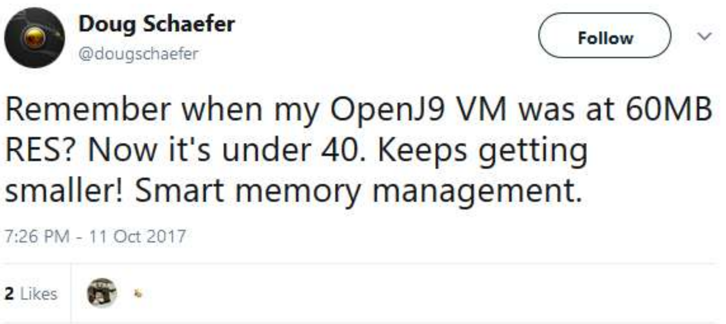
Mike Milinkovich @mmilinkov Follow

node.js has had a free ride because of the slow pace of *open* innovation in Java. With Eclipse @openj9 and Eclipse @vertx_project , Java can now compete on footprint in the cloud. #thisishuge

> **Doug Schaefer** @dougschaefer
> Well, for fun set up a simple vert.x server w/ derby on OpenJ9 and it's at 60MB RES. Smaller than Ghost, sqlite on node at 100MB. ¯_(ツ)_/¯

7:51 AM - 12 Oct 2017

58 Retweets **95** Likes

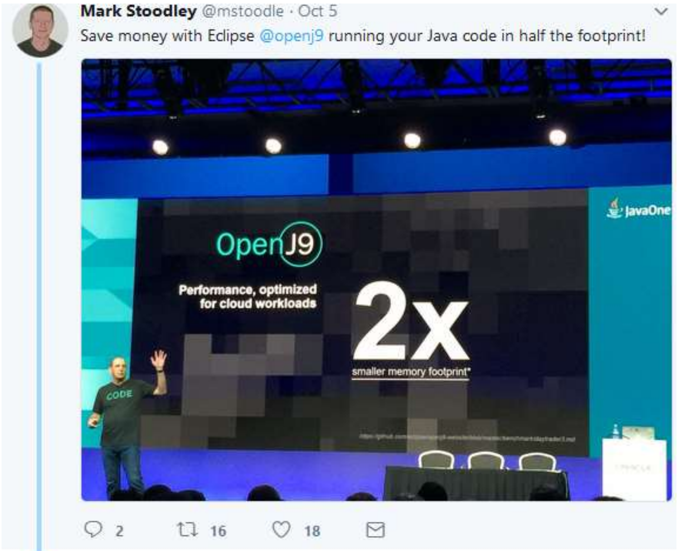


Doug Schaefer @dougschaefer Follow

Remember when my OpenJ9 VM was at 60MB RES? Now it's under 40. Keeps getting smaller! Smart memory management.

7:26 PM - 11 Oct 2017

2 Likes

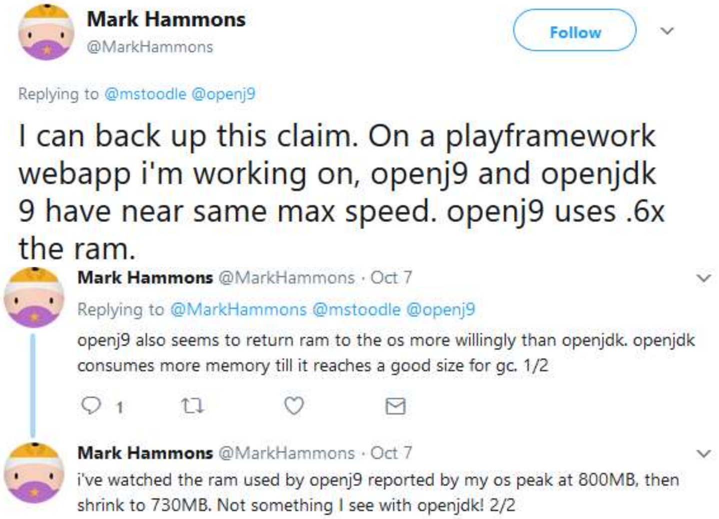


Mark Stoodley @mstoodle · Oct 5

Save money with Eclipse @openj9 running your Java code in half the footprint!

JavaOne
OpenJ9
Performance, optimized for cloud workloads
2x
smaller memory footprint*
CODE

2 16 18



Mark Hammons @MarkHammons Follow

Replying to @mstoodle @openj9

I can back up this claim. On a playframework webapp i'm working on, openj9 and openjdk 9 have near same max speed. openj9 uses .6x the ram.

Mark Hammons @MarkHammons · Oct 7

Replying to @MarkHammons @mstoodle @openj9

openj9 also seems to return ram to the os more willingly than openjdk. openjdk consumes more memory till it reaches a good size for gc. 1/2

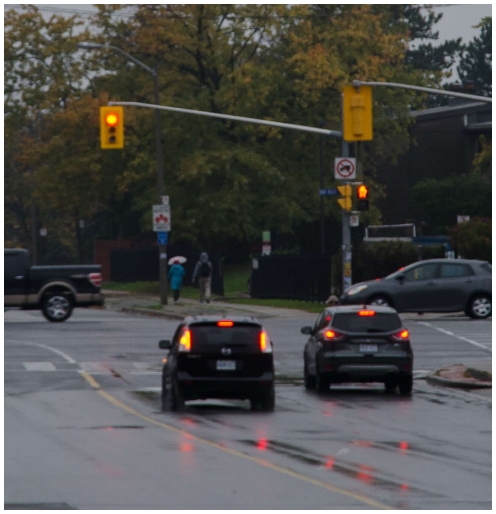
1

Mark Hammons @MarkHammons · Oct 7

i've watched the ram used by openj9 reported by my os peak at 800MB, then shrink to 730MB. Not something I see with openjdk! 2/2

Behavior at idle

- Important for cloud in high application density scenarios (over commit)
- anthesisgroup.com: “Some 30 percent of VMs are zombies”
  https://anthesisgroup.com/wp-content/uploads/2017/03/Comatsoe-Servers-Redux-2017.pdf



- Undesirable effects of idle JVMs:
  – May consume a small amount of CPU
  – May create some churn at the hypervisor level (swapping in/out guest VMs)
  – May take the CPU out of low power mode
  – May hold on to garbage memory that they don’t really need

Idle behavior in Eclipse OpenJ9

- Idle state detection mechanism

- Reduced frequency of sampling thread in idle state

- Reduced optimization level for JIT compiler during idle state

- Free the garbage in the heap and disclaim physical memory pages after some time in idle state

CPU and wakeups of idle JVM

- Analyze behavior of idle OpenLiberty server with powertop tool

OpenJDK9 with HotSpot – 0.168% CPU

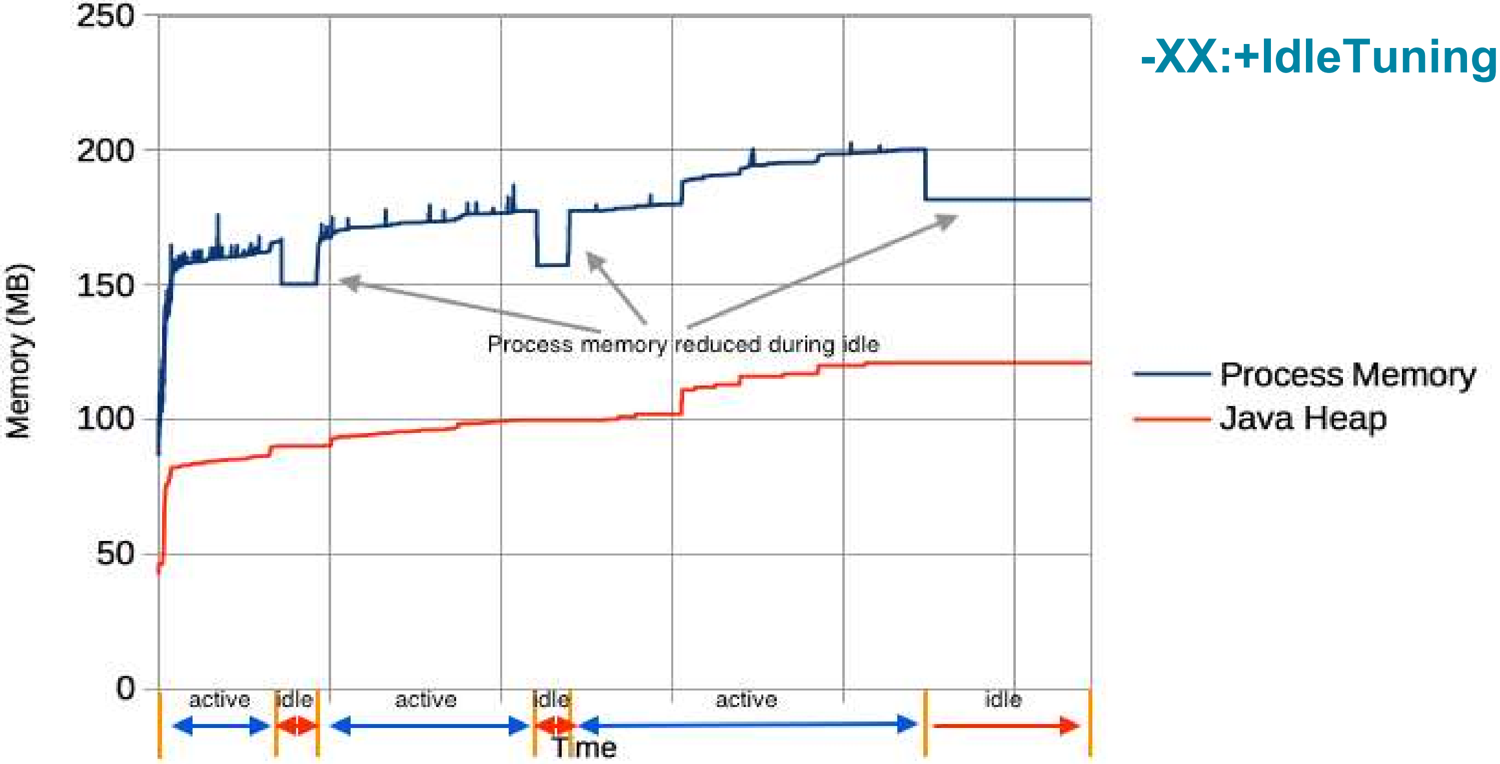
```
Summary: 84.7 wakeups/second,  0.0 GPU
ops/seconds, 0.0 VFS ops/sec and 0.3% CPU use.
Usage        Events/s    Category       Description
0.9 ms/s       44.2          Process       /sdks/OpenJDK9-
x64_Linux_20172509/jdk-9+181/bin/java
119.5 µs/s      20.0        Process       [xfsaild/dm-1]
138.6 µs/s       7.4        Timer         tick_sched_timer
10.5 µs/s        1.6        Process       [rcu_sched]
190.4 µs/s       1.5        Timer         hrtimer_wakeup
```

OpenJDK9 with OpenJ9 – 0.111% CPU

```
Summary: 38.5 wakeups/second,  0.1 GPU
ops/seconds, 0.0 VFS ops/sec and 0.2% CPU use
Usage        Events/s    Category       Description
681.2 µs/s    19.2        Process        /sdks/OpenJDK9-
OPENJ9_x64_Linux_20172509/jdk-9+181/bin/java
58.3 µs/s       5.2        Timer          tick_sched_timer
21.9 µs/s       3.6        Process       [rcu_sched]
39.3 µs/s       2.0        Timer          hrtimer_wakeup
157.1 µs/s      1.0        kWork          ixgbe_service_task
```

- OpenJ9 triggers ~55% fewer wakeups than HotSpot

Footprint of idle Eclipse OpenJ9



-XX:+IdleTuningGcOnIdle

Benchmark: https://github.com/blueperf/acmeair
More details: https://developer.ibm.com/javasdk/2017/09/25/still-paying-unused-memory-java-app-idle

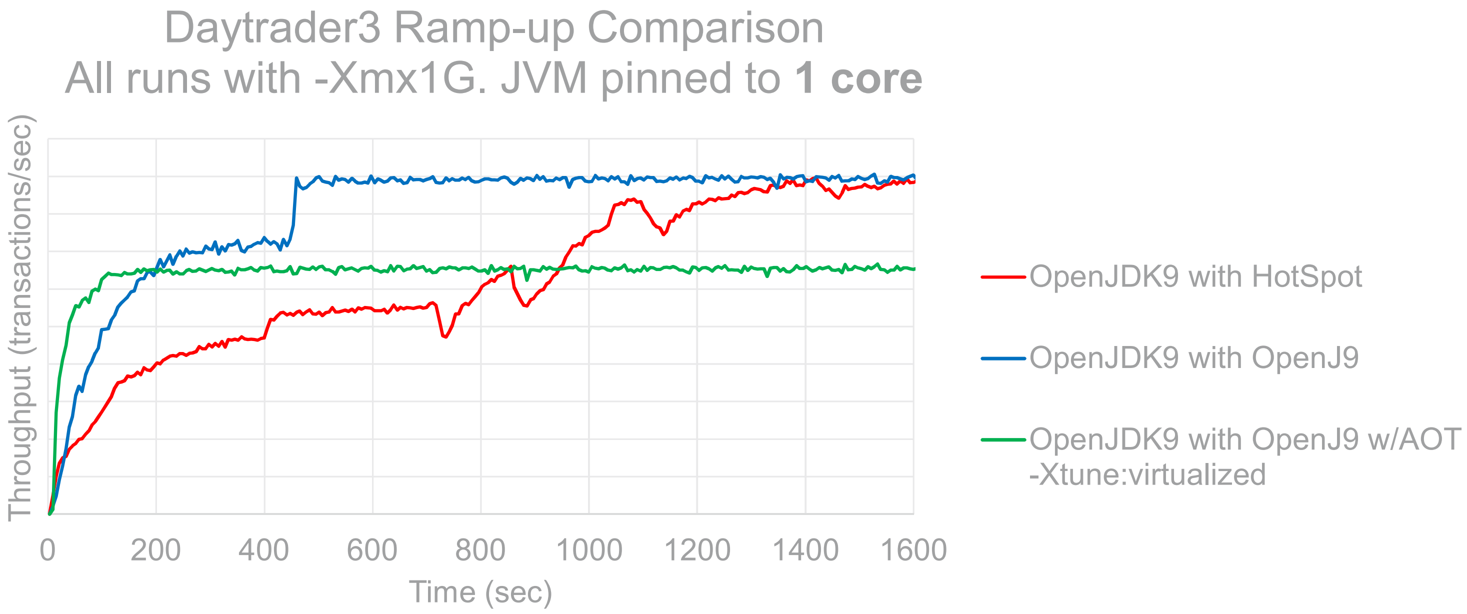
CPU constrained environments

- Virtual machines with 1 CPU are not that uncommon
- Compilation threads contending for CPU with application threads; side effects:
  - Slow ramp-up
  - Possible jitter in server response time

- OpenJ9 solutions to reduce CPU consumption:
  - Dynamic AOT compilation (enabled with -Xshareclasses)

  -Xtune:virtualized
    - More conservative JIT optimization. Subdued recompilation.
    - Saves compilation CPU (20-30%) at the expense of a 2-3% throughput loss
    - Some reduction in footprint
    - Works well in conjunction of dynamic AOT (generate AOT code as much as possible - if enabled)

Ramping-up in a CPU constrained environment



- -Xtune:virtualized and AOT good for CPU constrained situations and short running applications

Response time

- Jitter in response time due to:
  - JIT compilation overhead (when JVM is CPU constrained)
  - GC operation – “stop the world”

- Addressing the GC pauses in OpenJ9
  - Metronome – soft real-time GC policy
    - GC pauses configurable to as low as 1ms
  - Pause-less GC feature for zOS
    - GC can run concurrently with application
    - Hardware support in z14 – Guarded Storage Facility
    - Enable with **-Xgc:concurrentScavenge**

z14: Pause-less Garbage Collection
Java Store Inventory and Point of Sale Application

Java GC-tuning made easier

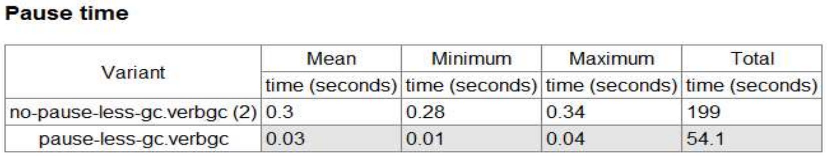
High scavenge pause times made this application a candidate for Pause-less GC

- Up to **3.4x** better throughput for **response-time** constrained Service Level Agreements (SLAs)
- Up to **10x** better average GC pause-times

Enable Pause-less GC with:

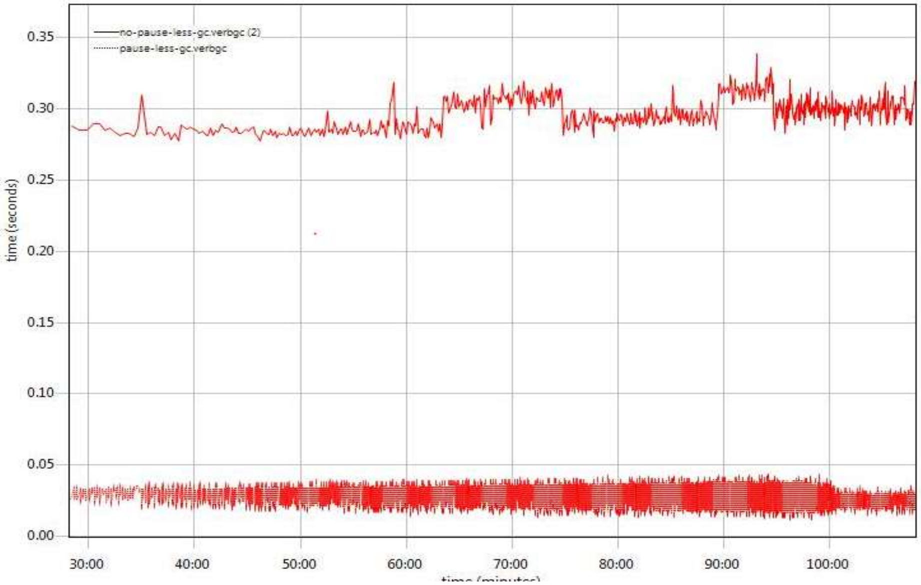
- IBM Java 8 SR5 or newer (OpenJ9 included)
- IBM z14’s Guarded Storage Facility
- z/OS 2.3 or z/OS 2.2 with APAR OA51643

JVM option: **-Xgc:concurrentScavenge**



Pause time

Variant	Mean	Minimum	Maximum	Total
	time (seconds)	time (seconds)	time (seconds)	time (seconds)
no-pause-less-gc.verbgc (2)	0.3	0.28	0.34	199
pause-less-gc.verbgc	0.03	0.01	0.04	54.1



IBM Monitoring and Diagnostic Tools - [Garbage Collection and Memory Visualizer]

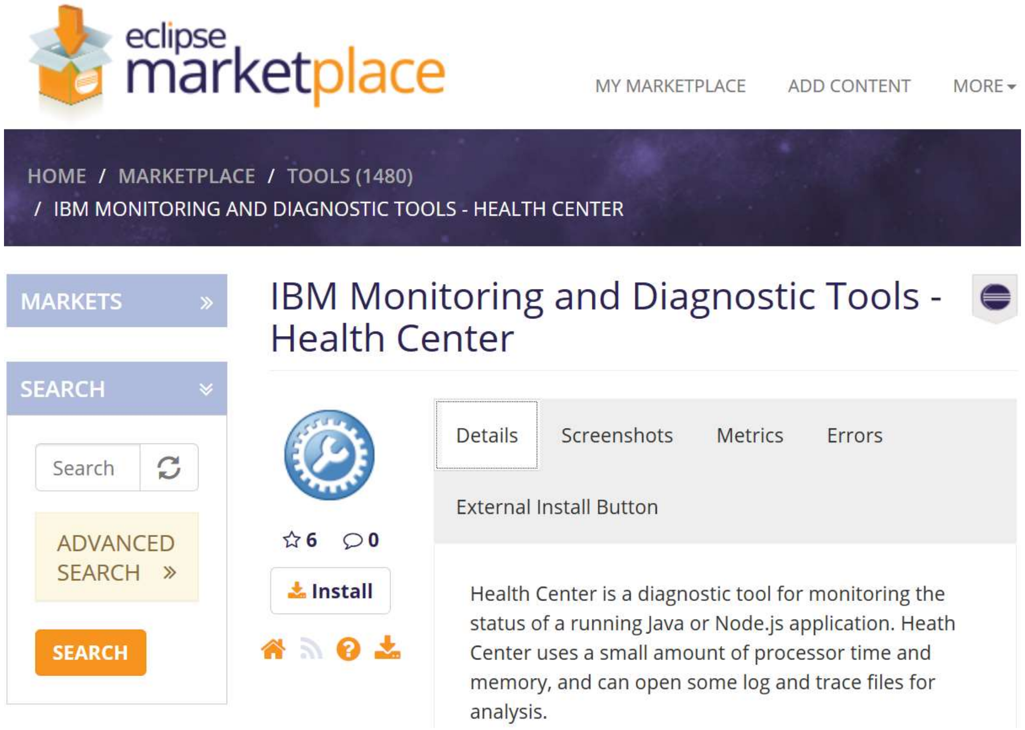
Performance monitoring tools

- Many low level performance tools exist
  - CPU: top, htop, vmstat, pidstat, mpstat, sar, nmon
  - Memory: sar, dstat, slabtop, free, nmon
  - Disk activity: iotop, iostat, sar, nmon
  - Network: ping, iftop, netstat, tcp, nicstat,
  - Profilers: perf, oprofile, tprof

- OpenJ9 performance tools
  - Health Center
  - Garbage Collector and Memory Visualizer (GCMV)

Health Center

- Live monitoring tool with low overhead (<1%)
- Provides insight into your application behavior with visualization
- Diagnoses potential problems and makes recommendations
- Powerful API allowing embedding of Health Center into other applications



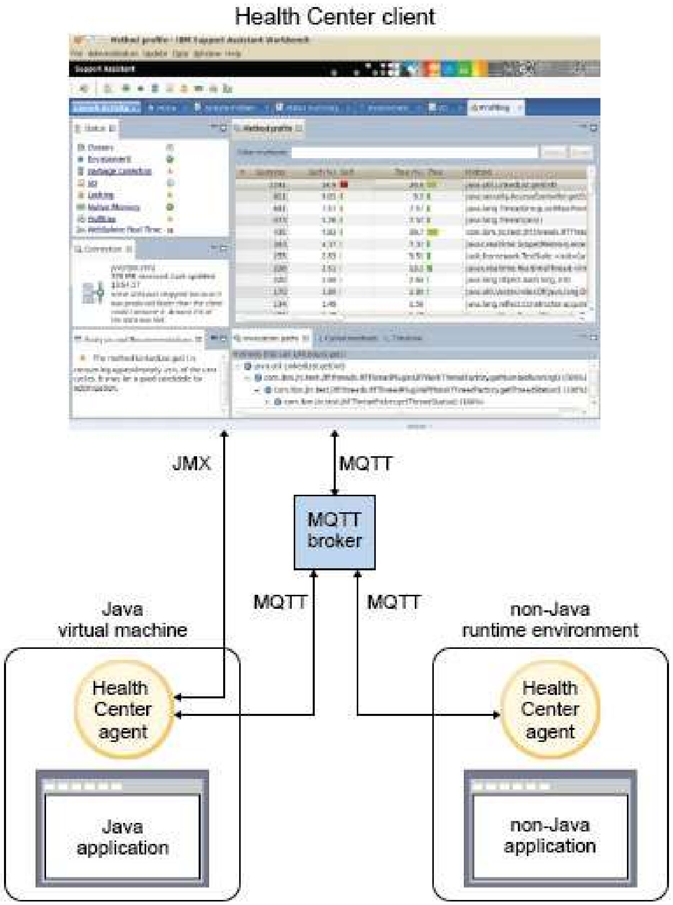
IBM Monitoring and Diagnostic Tools - Health Center

Health Center is a diagnostic tool for monitoring the status of a running Java or Node.js application. Heath Center uses a small amount of processor time and memory, and can open some log and trace files for analysis.

Health Center

- Tool is composed of two parts
  - Agent that collects data from running JVM
  - Eclipse based client that connects to the agent (typically running remotely)
- The agent ships with all IBM SDK for Java releases
- Latest version of agent available from within Health Center client
- Full usage instructions provided in the client Help topics
- Monitoring enabled with command line option

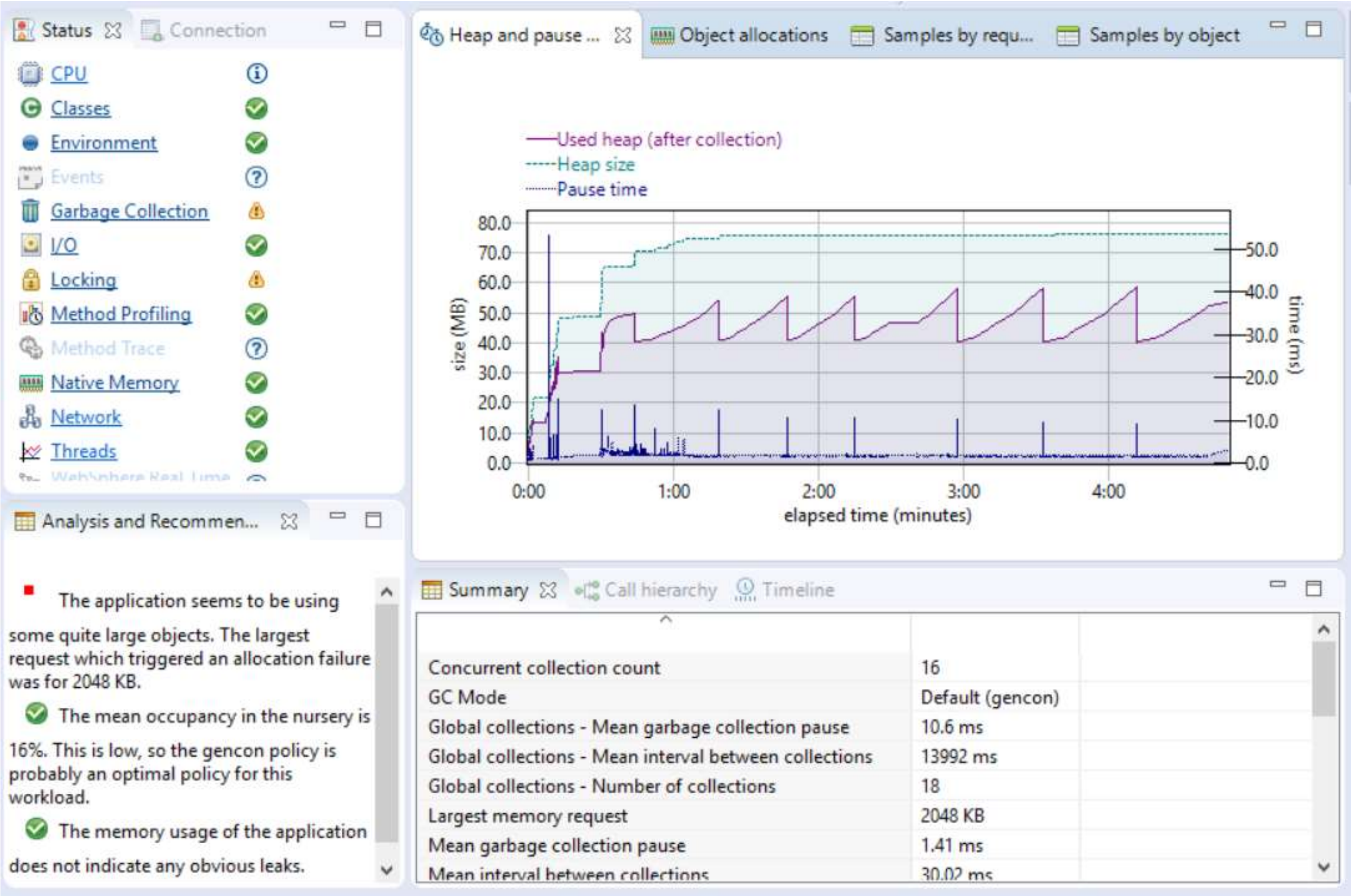
  java –Xhealthcenter HelloWorld
- Late attach possible
- Headless mode - collection without connecting the GUI



Health Center

- Provides visualization and monitoring in the following areas
  - Garbage collection
  - Method profiling
  - Lock analysis
  - Threading
  - Classes
  - Environment
  - Memory
  - CPU
  - I/O
  - Network

Health Center – Garbage collection perspective



CPU	ⓘ
Classes	✓
Environment	✓
Events	?
Garbage Collection	⚠
I/O	✓
Locking	⚠
Method Profiling	✓
Method Trace	?
Native Memory	✓
Network	✓
Threads	✓
WebSphere Real Time	

■ The application seems to be using some quite large objects. The largest request which triggered an allocation failure was for 2048 KB.

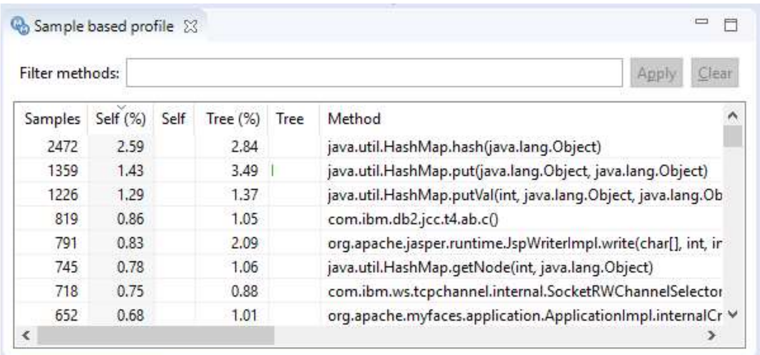
✓ The mean occupancy in the nursery is 16%. This is low, so the gencon policy is probably an optimal policy for this workload.

✓ The memory usage of the application does not indicate any obvious leaks.

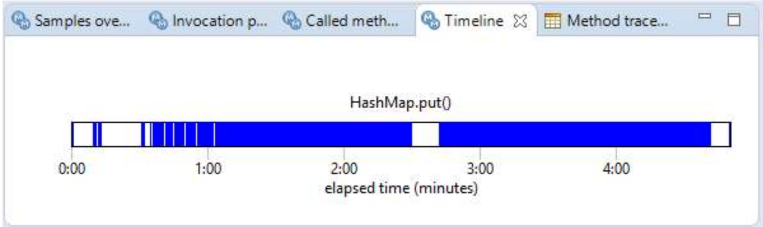
Concurrent collection count	16	
GC Mode	Default (gencon)	
Global collections - Mean garbage collection pause	10.6 ms	
Global collections - Mean interval between collections	13992 ms	
Global collections - Number of collections	18	
Largest memory request	2048 KB	
Mean garbage collection pause	1.41 ms	
Mean interval between collections	30.02 ms	

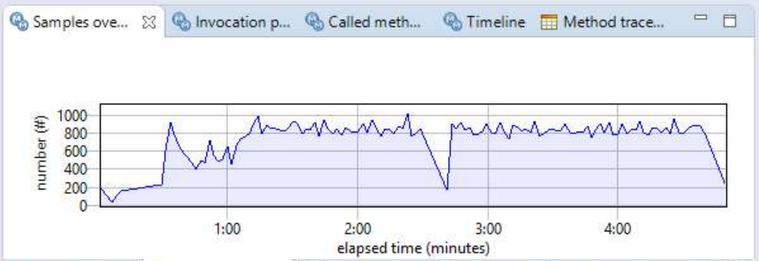
Health Center – Method Profiling perspective

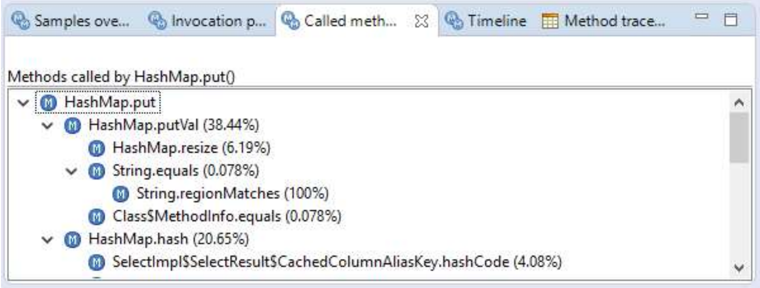
- Always-on profiling
  - No bytecode instrumentation, no recompilation
- Identifies hottest methods
- Full callstacks to identify callers and callees



Samples	Self (%)	Self	Tree (%)	Tree	Method
2472	2.59		2.84		java.util.HashMap.hash(java.lang.Object)
1359	1.43		3.49		java.util.HashMap.put(java.lang.Object, java.lang.Object)
1226	1.29		1.37		java.util.HashMap.putVal(int, java.lang.Object, java.lang.Ob
819	0.86		1.05		com.ibm.db2.jcc.t4.ab.c()
791	0.83		2.09		org.apache.jasper.runtime.JspWriterImpl.write(char[], int, ir
745	0.78		1.06		java.util.HashMap.getNode(int, java.lang.Object)
718	0.75		0.88		com.ibm.ws.tcpchannel.internal.SocketRWChannelSelector
652	0.68		1.01		org.apache.myfaces.application.ApplicationImpl.internalCr

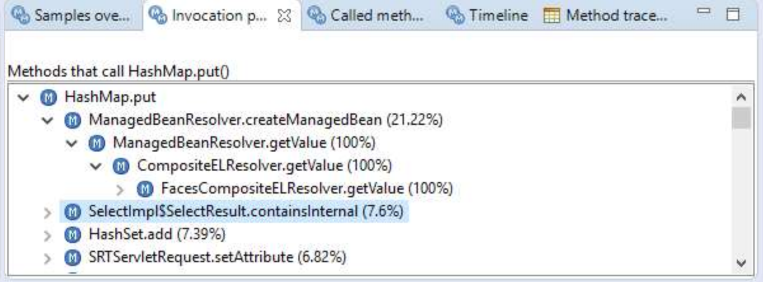






Methods called by HashMap.put()

- HashMap.put
  - HashMap.putVal (38.44%)
    - HashMap.resize (6.19%)
    - String.equals (0.078%)
      - String.regionMatches (100%)
    - Class$MethodInfo.equals (0.078%)
  - HashMap.hash (20.65%)
    - SelectImpl$SelectResult$CachedColumnAliasKey.hashCode (4.08%)

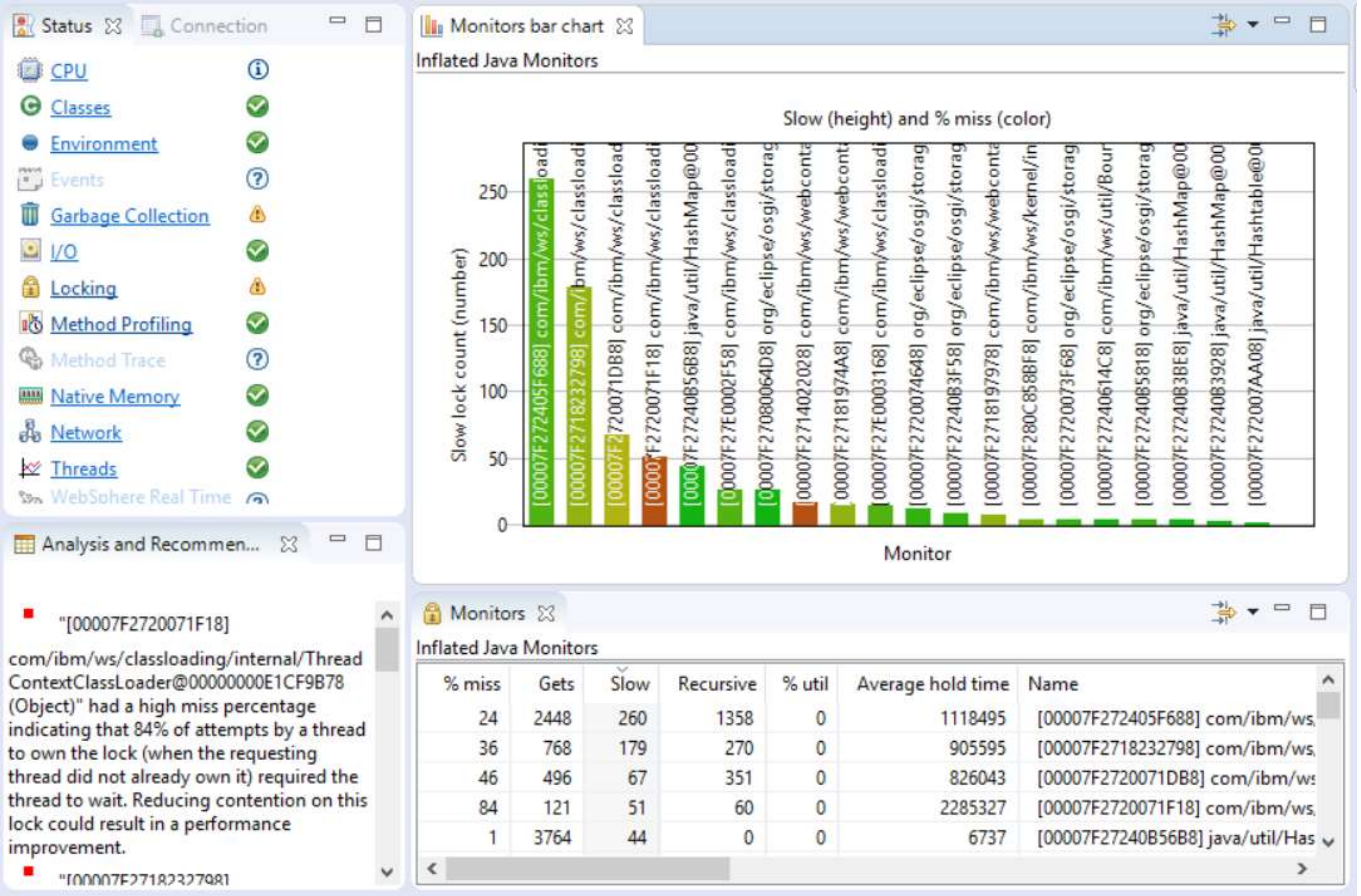


Methods that call HashMap.put()

- HashMap.put
  - ManagedBeanResolver.createManagedBean (21.22%)
    - ManagedBeanResolver.getValue (100%)
      - CompositeELResolver.getValue (100%)
        - FacesCompositeELResolver.getValue (100%)
  - SelectImpl$SelectResult.containsInternal (7.6%)
  - HashSet.add (7.39%)
  - SRTServletRequest.setAttribute (6.82%)

Health Center – Locking perspective

- Always-on lock monitoring
- Helps identify points of contention in the application


Status
Connection
CPU
Classes
Environment
Events
Garbage Collection
I/O
Locking
Method Profiling
Method Trace
Native Memory
Network
Threads
WebSphere Real Time
Analysis and Recommen...
Monitors bar chart
Inflated Java Monitors
Slow (height) and % miss (color)
Slow lock count (number)
250
200
150
100
50
0
Monitor
[00007F272405F688] com/ibm/ws/classloadi
[00007F2718232798] com/ibm/ws/classloadi
[00007F2720071DB8] com/ibm/ws/classload
[00007F2720071F18] com/ibm/ws/classloadi
[00007F27240B56B8] java/util/HashMap@00
[00007F27E0002F58] com/ibm/ws/classloadi
[00007F27080064D8] org/eclipse/osgi/storag
[00007F2714022028] com/ibm/ws/webconta
[00007F27181974A8] com/ibm/ws/webconta
[00007F27E0003168] com/ibm/ws/classloadi
[00007F2720074648] org/eclipse/osgi/storag
[00007F27240B3F58] org/eclipse/osgi/storag
[00007F2718197978] com/ibm/ws/webconta
[00007F280C858BF8] com/ibm/ws/kernel/in
[00007F2720073F68] org/eclipse/osgi/storag
[00007F2724061 4C8] com/ibm/ws/util/Bour
[00007F27240B5818] org/eclipse/osgi/storag
[00007F27240B3BE8] java/util/HashMap@00
[00007F27240B3928] java/util/HashMap@00
[00007F272007AA08] java/util/Hashtable@00
Monitors
Inflated Java Monitors

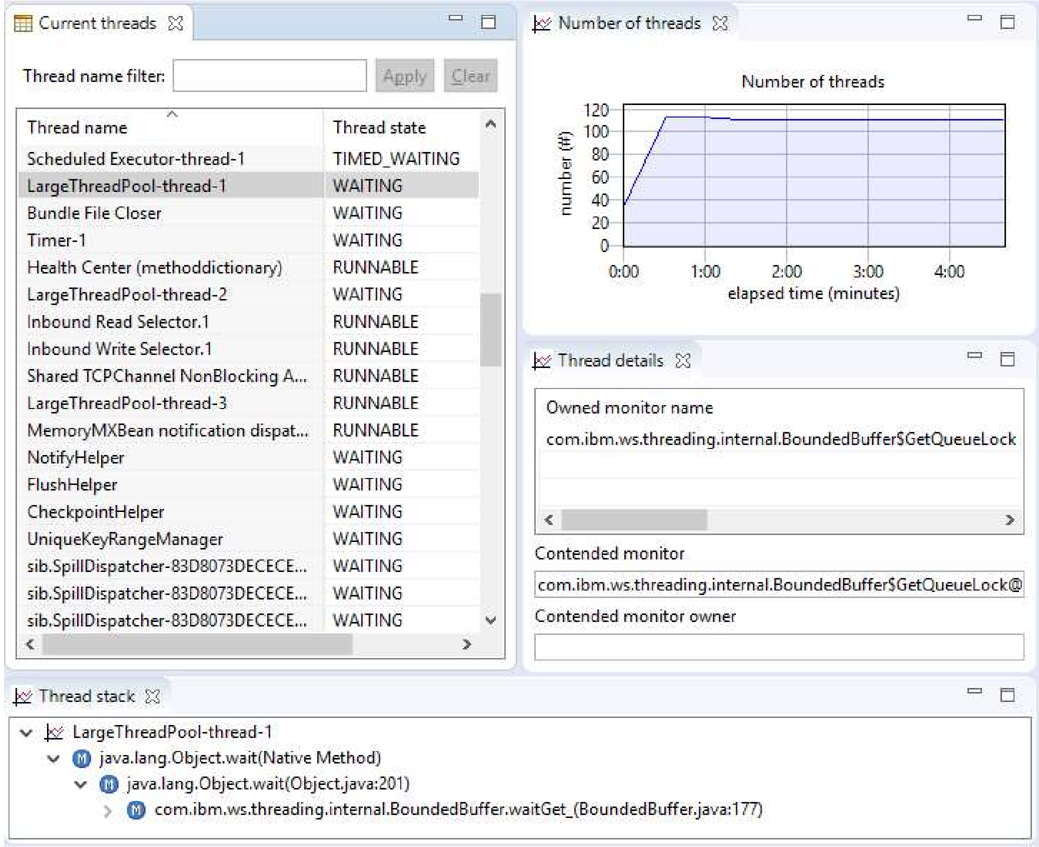
"[00007F2720071F18] com/ibm/ws/classloading/internal/ThreadContextClassLoader@00000000E1CF9B78 (Object)" had a high miss percentage indicating that 84% of attempts by a thread to own the lock (when the requesting thread did not already own it) required the thread to wait. Reducing contention on this lock could result in a performance improvement.

"[00007F2718232798]

% miss	Gets	Slow	Recursive	% util	Average hold time	Name
24	2448	260	1358	0	1118495	[00007F272405F688] com/ibm/ws,
36	768	179	270	0	905595	[00007F2718232798] com/ibm/ws,
46	496	67	351	0	826043	[00007F2720071DB8] com/ibm/ws
84	121	51	60	0	2285327	[00007F2720071F18] com/ibm/ws,
1	3764	44	0	0	6737	[00007F27240B56B8] java/util/Has

Health Center – Threads perspective

- List of current threads and states
- Number of threads over time
- Detection of contended monitors
- Deadlock detection and analysis



Thread name	Thread state
Scheduled Executor-thread-1	TIMED_WAITING
LargeThreadPool-thread-1	WAITING
Bundle File Closer	WAITING
Timer-1	WAITING
Health Center (methoddictionary)	RUNNABLE
LargeThreadPool-thread-2	WAITING
Inbound Read Selector.1	RUNNABLE
Inbound Write Selector.1	RUNNABLE
Shared TCPChannel NonBlocking A...	RUNNABLE
LargeThreadPool-thread-3	RUNNABLE
MemoryMXBean notification dispat...	RUNNABLE
NotifyHelper	WAITING
FlushHelper	WAITING
CheckpointHelper	WAITING
UniqueKeyRangeManager	WAITING
sib.SpillDispatcher-83D8073DECECE...	WAITING
sib.SpillDispatcher-83D8073DECECE...	WAITING
sib.SpillDispatcher-83D8073DECECE...	WAITING

Owned monitor name

com.ibm.ws.threading.internal.BoundedBuffer$GetQueueLock

Contended monitor

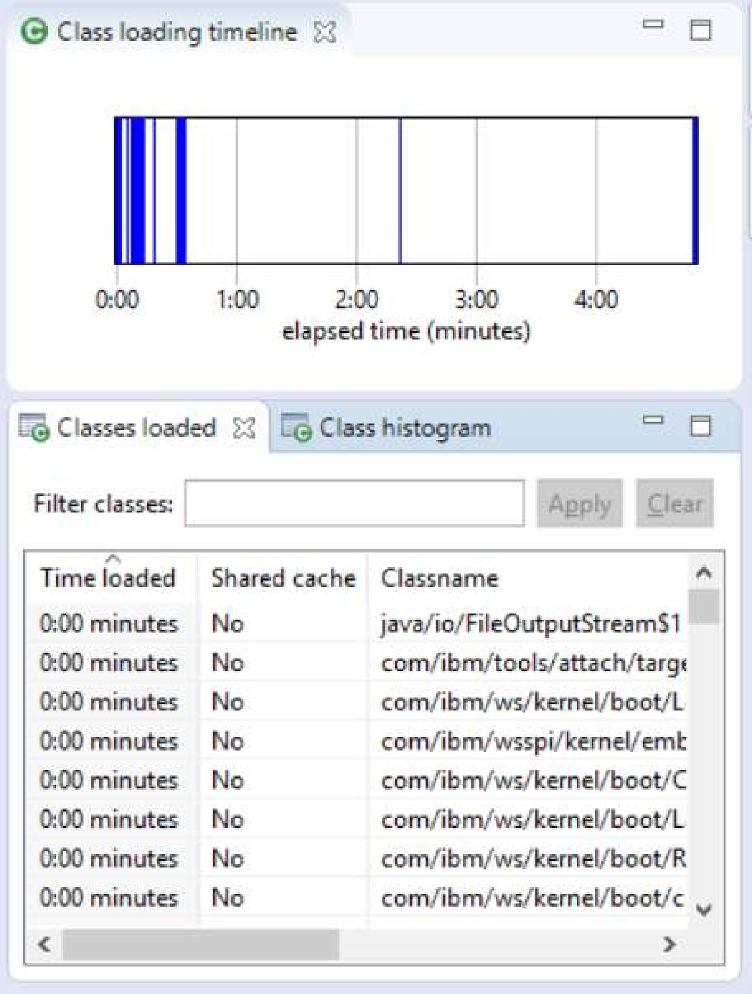
com.ibm.ws.threading.internal.BoundedBuffer$GetQueueLock@

Contended monitor owner

```
LargeThreadPool-thread-1
  java.lang.Object.wait(Native Method)
    java.lang.Object.wait(Object.java:201)
      com.ibm.ws.threading.internal.BoundedBuffer.waitGet_(BoundedBuffer.java:177)
```

Health Center – Class loading perspective

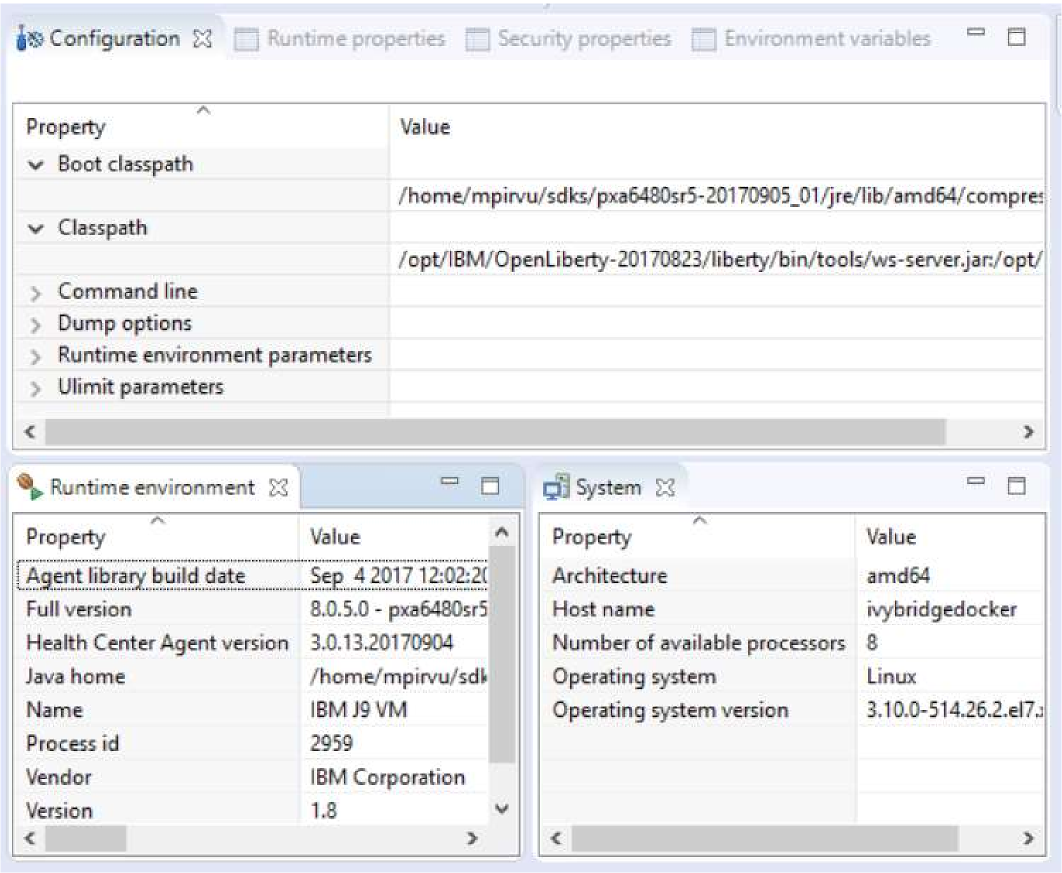
- Shows all loaded classes
- Shows timeline of loading events
- Identifies shared classes
- Shows number of unloaded classes



Time loaded	Shared cache	Classname
0:00 minutes	No	java/io/FileOutputStream$1
0:00 minutes	No	com/ibm/tools/attach/targe
0:00 minutes	No	com/ibm/ws/kernel/boot/L
0:00 minutes	No	com/ibm/wsspi/kernel/emb
0:00 minutes	No	com/ibm/ws/kernel/boot/C
0:00 minutes	No	com/ibm/ws/kernel/boot/L
0:00 minutes	No	com/ibm/ws/kernel/boot/R
0:00 minutes	No	com/ibm/ws/kernel/boot/c

Health Center – Environment reporting

- Detects invalid Java options
- Detects options which may hurt performance
- Useful for detecting configuration-related problems

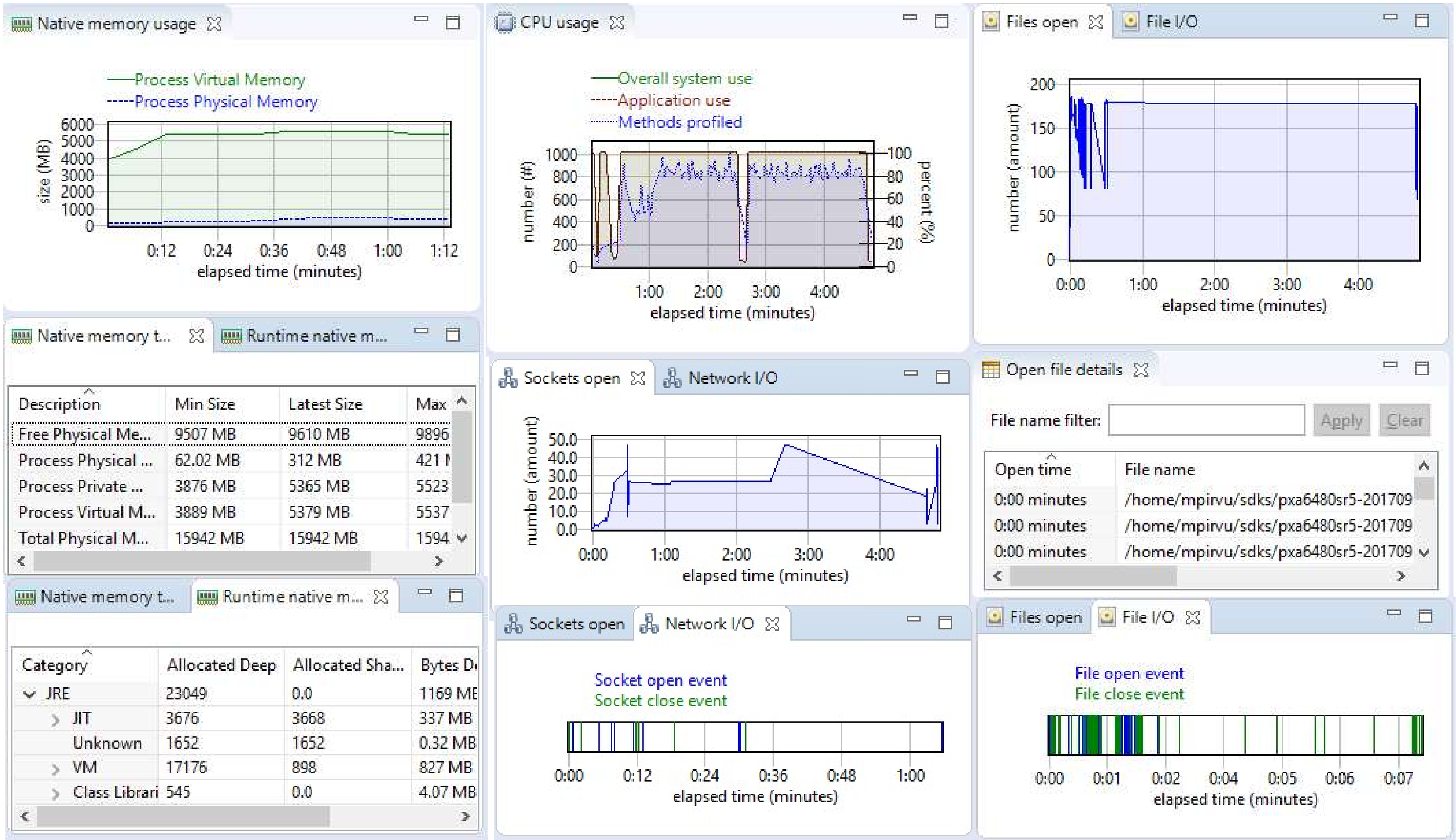


Property	Value
Boot classpath	
	/home/mpirvu/sdks/pxa6480sr5-20170905_01/jre/lib/amd64/compres
Classpath	
	/opt/IBM/OpenLiberty-20170823/liberty/bin/tools/ws-server.jar:/opt/
Command line	
Dump options	
Runtime environment parameters	
Ulimit parameters	

Property	Value
Agent library build date	Sep 4 2017 12:02:2(
Full version	8.0.5.0 - pxa6480sr5
Health Center Agent version	3.0.13.20170904
Java home	/home/mpirvu/sdk
Name	IBM J9 VM
Process id	2959
Vendor	IBM Corporation
Version	1.8

Property	Value
Architecture	amd64
Host name	ivybridgedocker
Number of available processors	8
Operating system	Linux
Operating system version	3.10.0-514.26.2.el7.

Health Center – Other perspectives



Description	Min Size	Latest Size	Max
Free Physical Me...	9507 MB	9610 MB	9896
Process Physical ...	62.02 MB	312 MB	421 I
Process Private ...	3876 MB	5365 MB	5523
Process Virtual M...	3889 MB	5379 MB	5537
Total Physical M...	15942 MB	15942 MB	1594

Open time	File name
0:00 minutes	/home/mpirvu/sdks/pxa6480sr5-201709
0:00 minutes	/home/mpirvu/sdks/pxa6480sr5-201709
0:00 minutes	/home/mpirvu/sdks/pxa6480sr5-201709

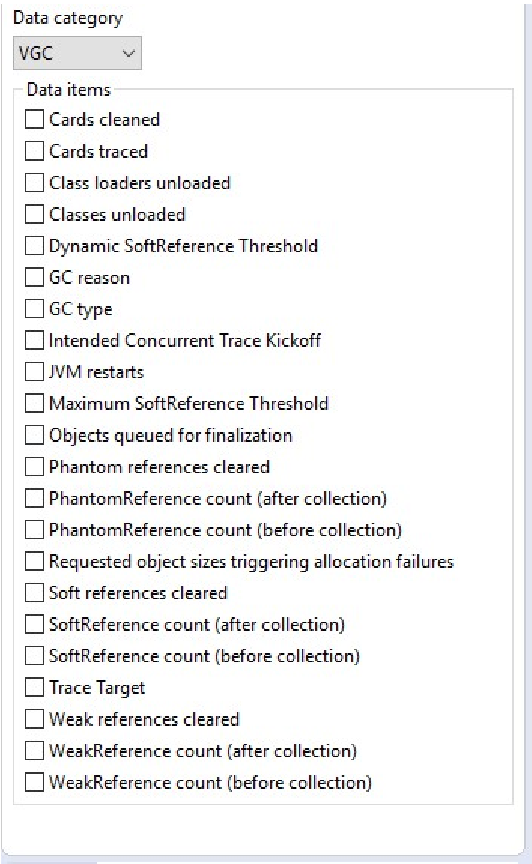
Category	Allocated Deep	Allocated Sha...	Bytes D
JRE	23049	0.0	1169 MB
JIT	3676	3668	337 MB
Unknown	1652	1652	0.32 MB
VM	17176	898	827 MB
Class Librari	545	0.0	4.07 MB

Garbage Collector and Memory Visualizer (GCMV)

- Visualize a wide range of GC data and Java heap statistics over time
- Recommendations for optimizing GC
- Detect memory leaks
- Visualize physical and virtual memory of the JVM

- Extracts information from:
  - GC verbose logs – for Java heap
  - ps (linux, z/OS), svmon (AIX) or perfmon (Windows) tools – for native footprint

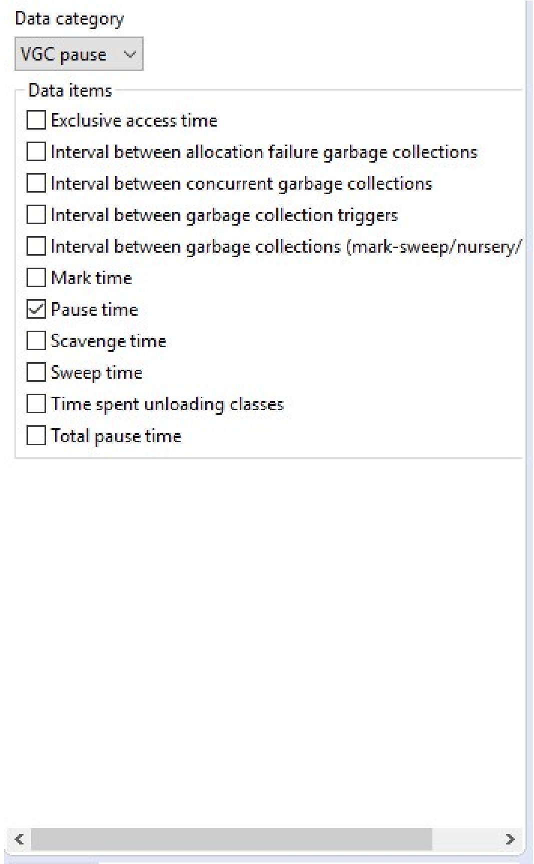
GCMV data categories



Data category: VGC

Data items

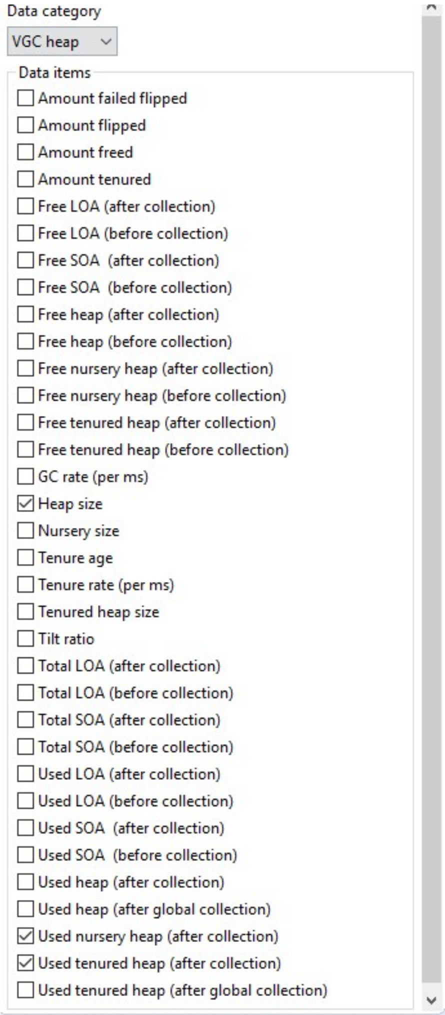
- ☐ Cards cleaned
- ☐ Cards traced
- ☐ Class loaders unloaded
- ☐ Classes unloaded
- ☐ Dynamic SoftReference Threshold
- ☐ GC reason
- ☐ GC type
- ☐ Intended Concurrent Trace Kickoff
- ☐ JVM restarts
- ☐ Maximum SoftReference Threshold
- ☐ Objects queued for finalization
- ☐ Phantom references cleared
- ☐ PhantomReference count (after collection)
- ☐ PhantomReference count (before collection)
- ☐ Requested object sizes triggering allocation failures
- ☐ Soft references cleared
- ☐ SoftReference count (after collection)
- ☐ SoftReference count (before collection)
- ☐ Trace Target
- ☐ Weak references cleared
- ☐ WeakReference count (after collection)
- ☐ WeakReference count (before collection)



Data category: VGC pause

Data items

- ☐ Exclusive access time
- ☐ Interval between allocation failure garbage collections
- ☐ Interval between concurrent garbage collections
- ☐ Interval between garbage collection triggers
- ☐ Interval between garbage collections (mark-sweep/nursery/
- ☐ Mark time
- ☑ Pause time
- ☐ Scavenge time
- ☐ Sweep time
- ☐ Time spent unloading classes
- ☐ Total pause time



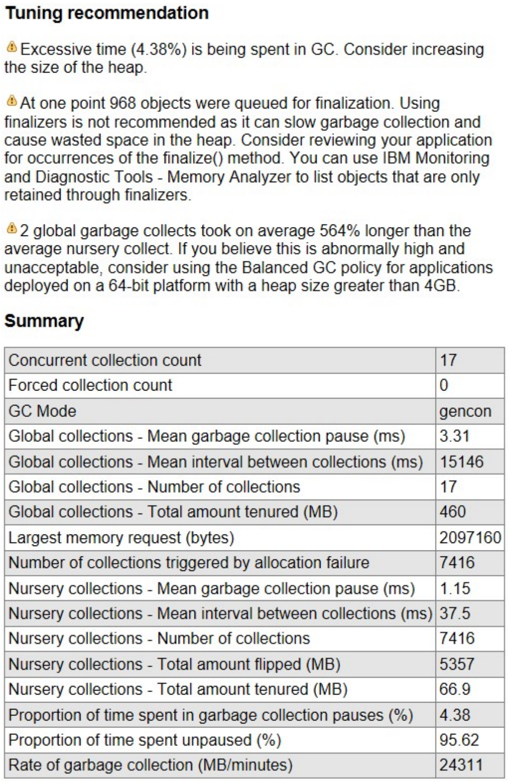
Data category: VGC heap

Data items

- ☐ Amount failed flipped
- ☐ Amount flipped
- ☐ Amount freed
- ☐ Amount tenured
- ☐ Free LOA (after collection)
- ☐ Free LOA (before collection)
- ☐ Free SOA (after collection)
- ☐ Free SOA (before collection)
- ☐ Free heap (after collection)
- ☐ Free heap (before collection)
- ☐ Free nursery heap (after collection)
- ☐ Free nursery heap (before collection)
- ☐ Free tenured heap (after collection)
- ☐ Free tenured heap (before collection)
- ☐ GC rate (per ms)
- ☑ Heap size
- ☐ Nursery size
- ☐ Tenure age
- ☐ Tenure rate (per ms)
- ☐ Tenured heap size
- ☐ Tilt ratio
- ☐ Total LOA (after collection)
- ☐ Total LOA (before collection)
- ☐ Total SOA (after collection)
- ☐ Total SOA (before collection)
- ☐ Used LOA (after collection)
- ☐ Used LOA (before collection)
- ☐ Used SOA (after collection)
- ☐ Used SOA (before collection)
- ☐ Used heap (after collection)
- ☐ Used heap (after global collection)
- ☑ Used nursery heap (after collection)
- ☑ Used tenured heap (after collection)
- ☐ Used tenured heap (after global collection)

GCMV snapshots

- **Analysis and recommendations**
  - Analysis can be limited using cropping



Tuning recommendation

⚠ Excessive time (4.38%) is being spent in GC. Consider increasing the size of the heap.

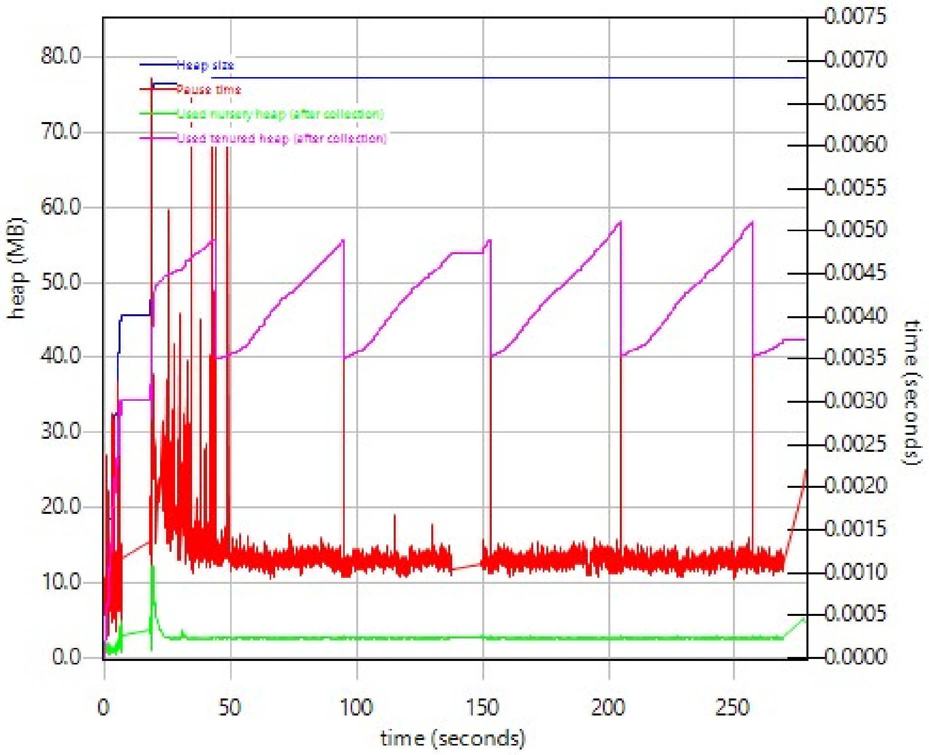
⚠ At one point 968 objects were queued for finalization. Using finalizers is not recommended as it can slow garbage collection and cause wasted space in the heap. Consider reviewing your application for occurrences of the finalize() method. You can use IBM Monitoring and Diagnostic Tools - Memory Analyzer to list objects that are only retained through finalizers.

⚠ 2 global garbage collects took on average 564% longer than the average nursery collect. If you believe this is abnormally high and unacceptable, consider using the Balanced GC policy for applications deployed on a 64-bit platform with a heap size greater than 4GB.

Summary

Concurrent collection count	17
Forced collection count	0
GC Mode	gencon
Global collections - Mean garbage collection pause (ms)	3.31
Global collections - Mean interval between collections (ms)	15146
Global collections - Number of collections	17
Global collections - Total amount tenured (MB)	460
Largest memory request (bytes)	2097160
Number of collections triggered by allocation failure	7416
Nursery collections - Mean garbage collection pause (ms)	1.15
Nursery collections - Mean interval between collections (ms)	37.5
Nursery collections - Number of collections	7416
Nursery collections - Total amount flipped (MB)	5357
Nursery collections - Total amount tenured (MB)	66.9
Proportion of time spent in garbage collection pauses (%)	4.38
Proportion of time spent unpaused (%)	95.62
Rate of garbage collection (MB/minutes)	24311

- **Graphical display of data**
  - Many metrics to choose from
  - Allows zoom, cropping and change of units



Conclusion

Eclipse OpenJ9 == The better JVM for the cloud

Questions?

Marius Pirvu
mpirvu@ca.ibm.com

Resources

- Description: https://www.eclipse.org/openj9
- Get involved: https://github.com/eclipse/openj9
  https://github.com/eclipse/omr
- Build your own: https://www.eclipse.org/openj9/oj9_build.html
- Download OpenJ9 binaries: https://adoptopenjdk.net/?variant=openjdk9-openj9
- Performance: https://github.com/eclipse/openj9-website/blob/master/benchmark/daytrader3.md
- Links to benchmarks:
  - Daytrader3: https://github.com/WASdev/sample.daytrader3
  - AcmeAir: https://github.com/blueperf/acmeair