
JavaOne™

Permanent Generation Removal Overview

Coleen Phillimore
Jon Masamitsu
Oracle Corporation
Hotspot JVM Team

The permanent generation in Oracle's HotSpot JVM held metadata that HotSpot used to describe Java objects. The permanent generation was garbage collected along with the Java heap during full GC. The permanent generation has been removed in HotSpot for JDK8. This session briefly describes the permanent generation and the motivation for its removal. It also discusses side effects of the removal that may affect the execution of a Java application.

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.

It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

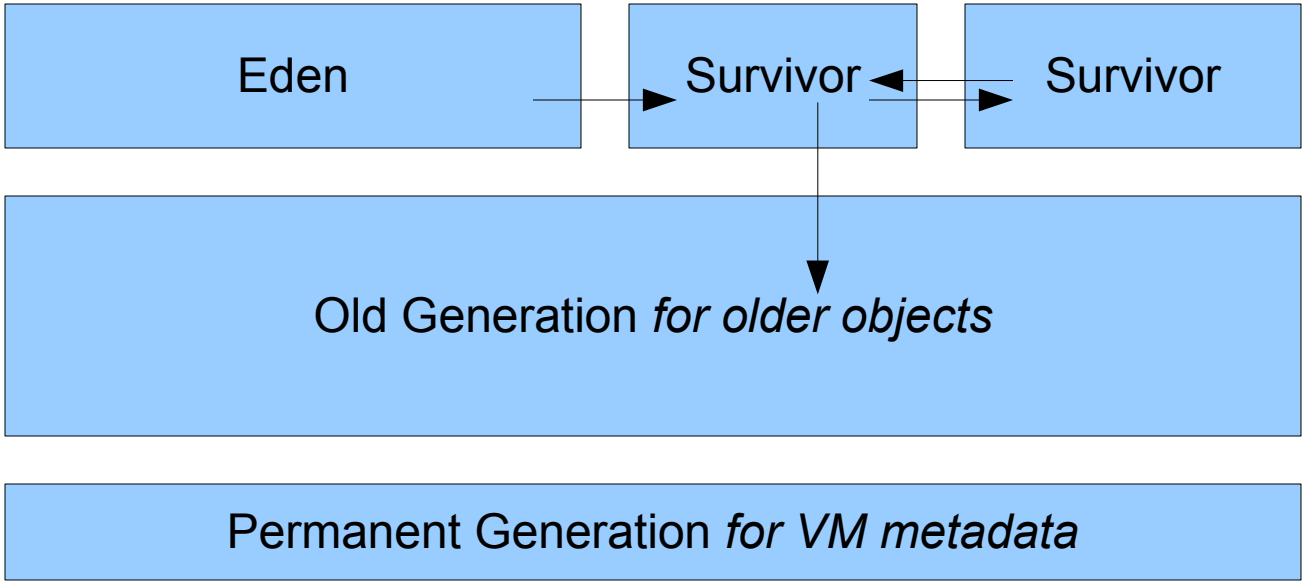
Program Agenda

- What was the Permanent Generation
- Where is Metadata Now
- Compressed Class Pointers
- New Tuning Flags
- Memory Pool MXBeans

What was PermGen?

- The “Permanent” Generation
- Region of Java Heap for JVM Class Metadata
- Hotspot’s internal representation of Java Classes
    - Class hierarchy information, fields, names
    - Method compilation information and bytecodes
    - Vtables
    - Constant pool and symbolic resolution

Java Memory Layout with PermGen



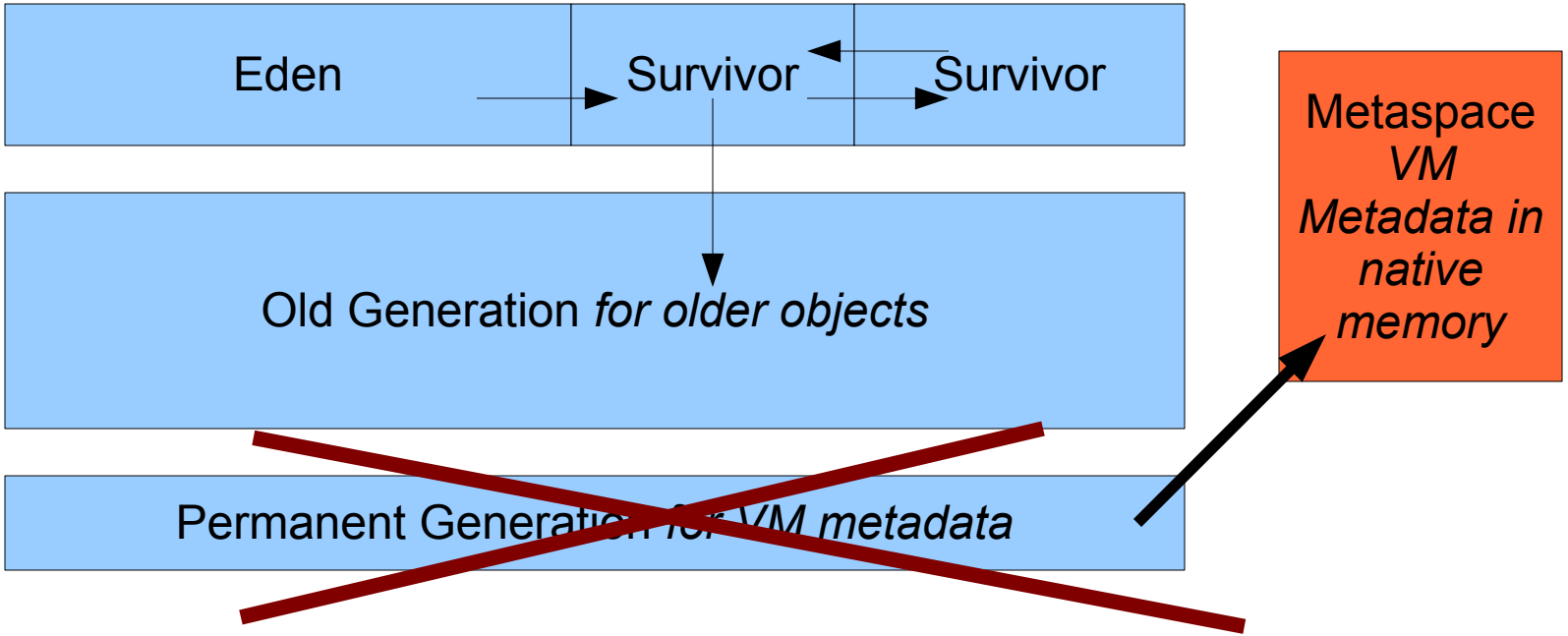
PermGen Size

- Limited to MaxPermSize – default ~64M - 85M
- Contiguous with Java Heap
  - Identifying young references from old gen and permgen would be more expensive and complicated with a non-contiguous heap – card table
- Once exhausted throws OutOfMemoryError “PermGen space”
  - Application could clear references to cause class unloading
  - Restart with larger MaxPermSize
- Size needed depends on number of classes, size of methods, size of constant pools

Why was PermGen Eliminated?

- Fixed size at startup – difficult to tune
  - -XX:MaxPermSize=?
- Internal Hotspot types were Java objects
  - Could move with full GC, opaque, not strongly typed and hard to debug, needed meta-metadata
- Simplify full collections
  - Special iterators for metadata for each collector
- Want to deallocate class data concurrently and not during GC pause
- Enable future improvements that were limited by PermGen

Where did JVM Metadata go?



Metaspace

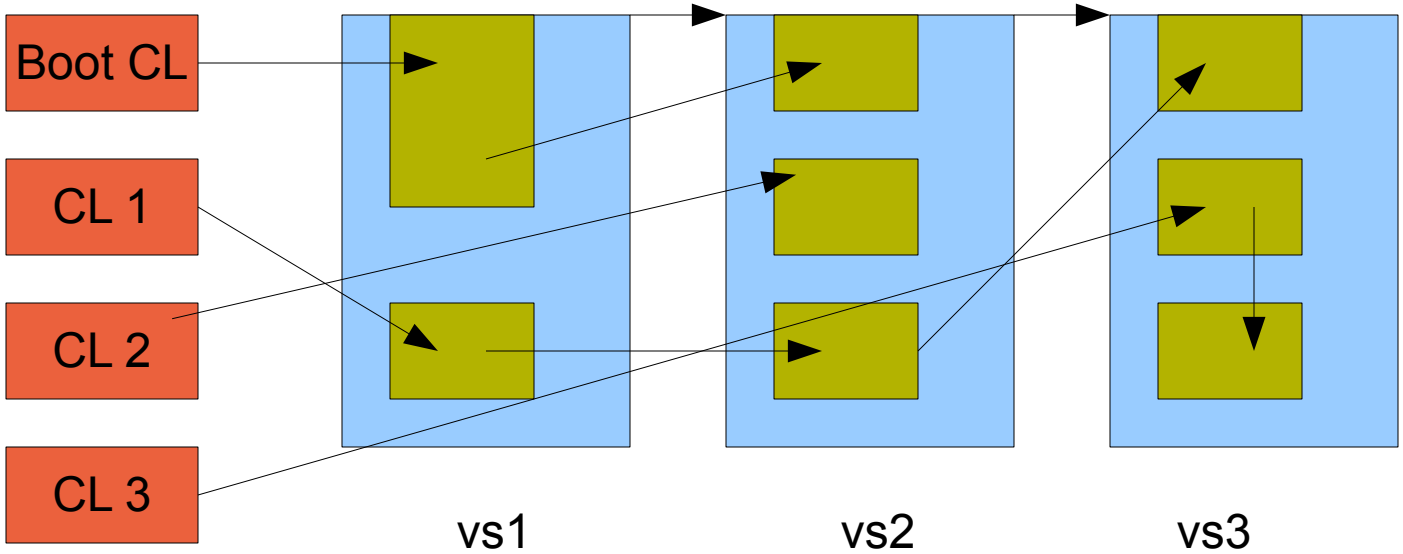
- Take advantage of Java Language Specification property
  - Classes and associated metadata lifetimes match class loader's
- Per loader storage area – Metaspace
  - Linear allocation only
  - No individual reclamation (except for RedefineClasses and class loading failure)
  - No GC scan or compaction
  - No relocation for metaspace objects
  - Reclamation *en-masse* when class loader found dead by GC
- **Collectively called Metaspace**

Metaspace Allocation

- Multiple mapped virtual memory spaces allocated for metadata
- Allocate per-class loader chunk lists
  - Chunk sizes depend on type of class loader
  - Smaller chunks for sun/reflect/DelegatingClassLoader, JSR292 anonymous classes
- Return chunks to free chunk lists
- Virtual memory spaces returned when emptied
- Strategies to minimize fragmentation

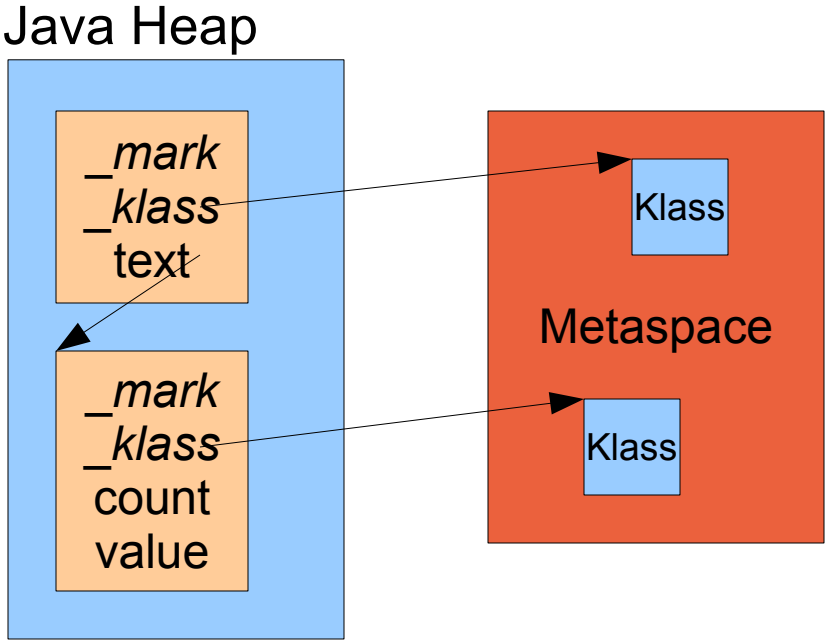
Metaspace Allocation

- Metachunks in virtual spaces (vs1, vs2, vs3...)



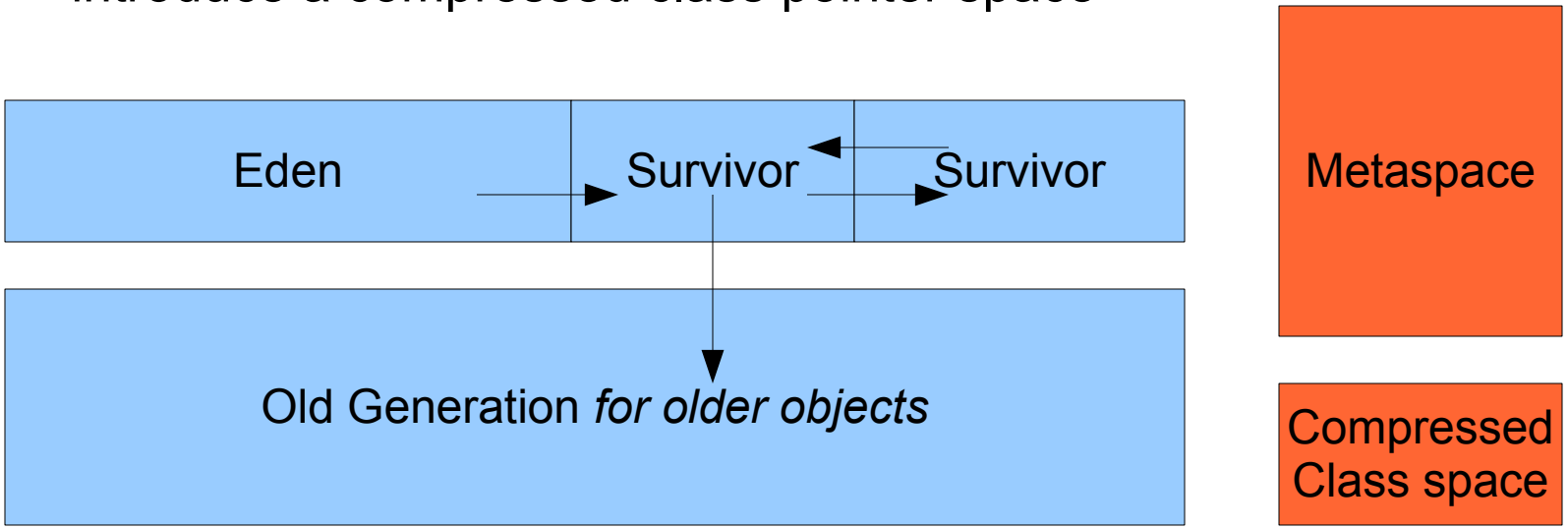
Java Object Memory Layout

```
class Message {
        // JVM adds _mark and
        //_klass pointer
        String text;
        void add(String s) { …}
        String get() { … }
}
```



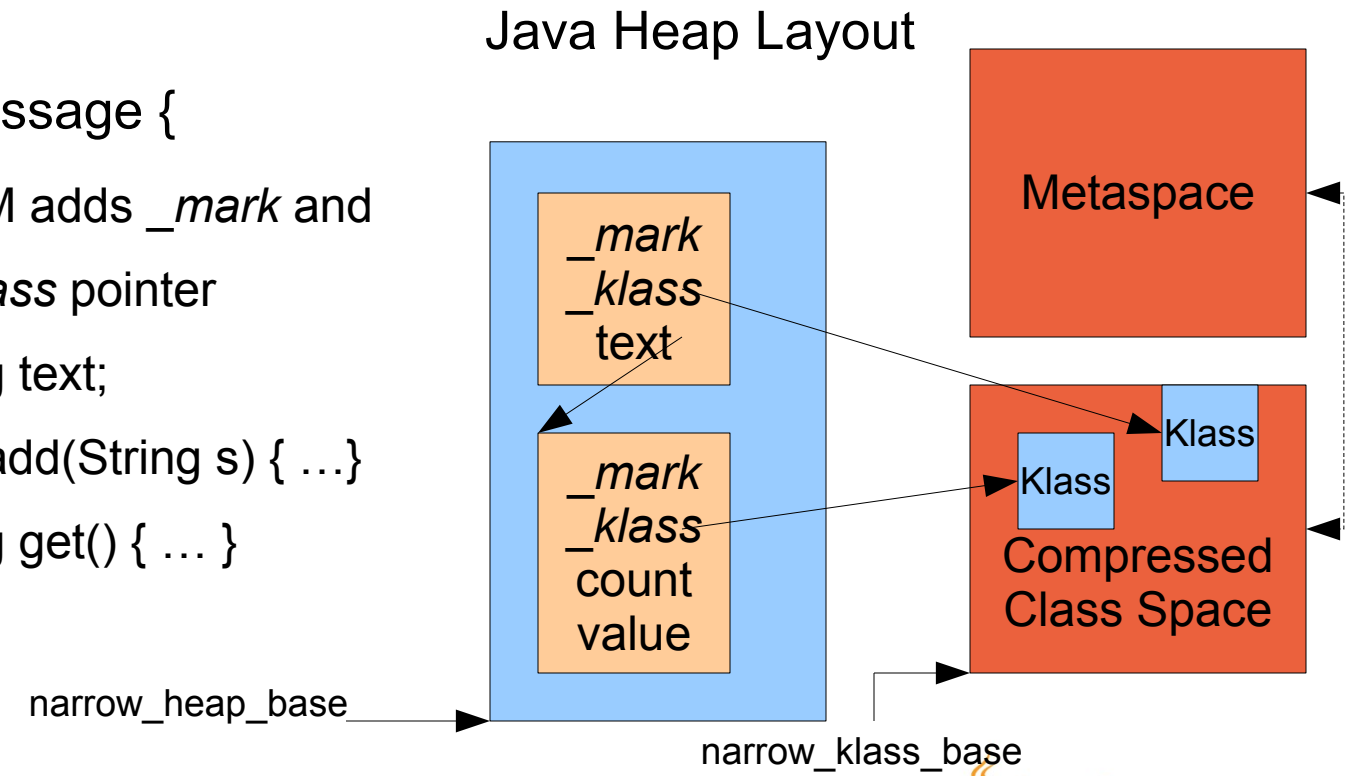
Compressed Class Pointer Space

- For 64 bit platforms, to compress JVM _*klass* pointers in objects, introduce a compressed class pointer space



Java Object Memory Layout with Compressed Pointers

```
class Message {
    // JVM adds _mark and
    // _klass pointer
    String text;
    void add(String s) { …}
    String get() { … }
}
```



Compressed Pointers

- Default for 64 bit platforms
- Compressed object pointers -XX:+UseCompressedOops
  - “oops” are “ordinary” object pointers
  - Object pointers are compressed to 32 bits in objects in Java Heap
  - Using a heap base (or zero if Java Heap is in lower 26G memory)
- Compressed Class Pointers -XX:+UseCompressedClassPointers
  - Objects have a pointer to VM Metadata class (2nd word) compressed to 32 bits
  - Using a base to the compressed class pointer space

Metaspace vs. Compressed Class Pointer Space

- Compressed Class Pointer Space contains only class metadata

  InstanceKlass, ArrayKlass

  - Only when UseCompressedClassPointers true
  - These include Java virtual tables for performance reasons
  - We are still shrinking this metadata type
- Metaspace contains all other class metadata that can be large

  Methods, Bytecodes, ConstantPool ...

Improved GC Performance

- During full collection, metadata to metadata pointers are not scanned
  - A lot of complex code (particularly for CMS) for metadata scanning was removed
- Metaspace contains few pointers into the Java heap
  - Pointer to java/lang/Class instance in class metadata
  - Pointer to component java/lang/Class in array class metadata
  - We should move these

Improved GC Performance

- No compaction costs for metadata
- Reduces root scanning (no scanning of VM dictionary of loaded classes and other internal hashtables)
- Improvements in full collection times – mileage will vary
- Working on class unloading in G1 after concurrent marking cycle
- Still finding optimizations and tuning

Tuning Flags - MaxMetaspaceSize

- -XX:MaxMetaspaceSize={unlimited}
- Metaspace is limited by the amount of memory on your machine.
- Limit the memory used by class metadata before excess swapping and native allocation failure occur
  - Use if suspected class loader memory leaks
  - Use on 32 bit if address space could be exhausted
- Sum of committed space in Metaspace and CompressedClassSpace

Tuning Flags - MetaspaceSize

- Initial MetaspaceSize 21 mb – GC initial high water mark for doing a full GC to collect classes
  - GC's are done to detect dead classloaders and unload classes
- Set to a higher limit if doing too many GC's at startup
- Possibly use same value set by PermSize to delay initial GC
- High water mark increases with subsequent collections for a reasonable amount of head room before next Metaspace GC
  - See MinMetaspaceFreeRatio and MaxMetaspaceFreeRatio
  - Interpreted similarly to analogous GC FreeRatio parameters

Tuning Flags - CompressedClassSpaceSize

- Only valid if -XX:+UseCompressedClassPointers (default on 64 bit)
- -XX:CompressedClassSpaceSize=1G
- Since this space is fixed at startup time currently, start out with large reservation
- Not committed until used
- Future work is to make this space growable
  - Doesn’t need to be contiguous, only reachable from the base address
  - Would rather shift more class metadata to Metaspace instead

Tuning Flags - CompressedClassSpaceSize

- In future might set ergonomically based on PredictedLoadedClassCount (experimental flag now)
  - Sets size of other internal JVM data structures, like dictionary of loaded classes
- Developer preview has different names for CompressedClassSpaceSize and UseCompressedClassPointers
  - Fixed in JDK-8015107: NPG: Use consistent naming for metaspace concepts

Tuning Flags – GC flags

- -XX:+PrintGCDetails and -XX:+PrintHeapAtGC

314.096: [Full GC (Ergonomics) [PSYoungGen: 68132K->0K(2511872K)] [ParOldGen: 547726K->475916K(768000K)] 615859K->475916K(3279872K), [**Metaspace: 35786K->31737K(1085440K)**], 0.5446800 secs] [Times: user=3.25 sys=0.01, real=0.54 secs]

Heap after GC invocations=222 (full 4):
PSYoungGen total 2511872K, used 0K [0x0000000755500000, 0x00000007feb00000, 0x0000000800000000)
eden space 2248704K, 0% used [0x0000000755500000,0x0000000755500000,0x00000007de900000)
from space 263168K, 0% used [0x00000007eea00000,0x00000007eea00000,0x00000007feb00000)
to space 263168K, 0% used [0x00000007de900000,0x00000007de900000,0x00000007eea00000)
ParOldGen total 768000K, used 475916K [0x0000000600000000, 0x000000062ee00000, 0x0000000755500000)
object space 768000K, 61% used [0x0000000600000000,0x000000061d0c3068,0x000000062ee00000)

Metaspace total 32188K, used 31737K, reserved 1085440K
data space 28504K, used 28189K, reserved 36864K
class space 3684K, used 3547K, reserved 1048576K

Metaspace Monitoring and Management

- MemoryManagerMXBean with name MetaspaceManager
- Manages new Metaspace and CompressedClassSpace memory pool MXBeans
- Both are MemoryType.NON_HEAP
- PermGen memory pool was on the list of HEAP memory pools in MemoryManager, now removed

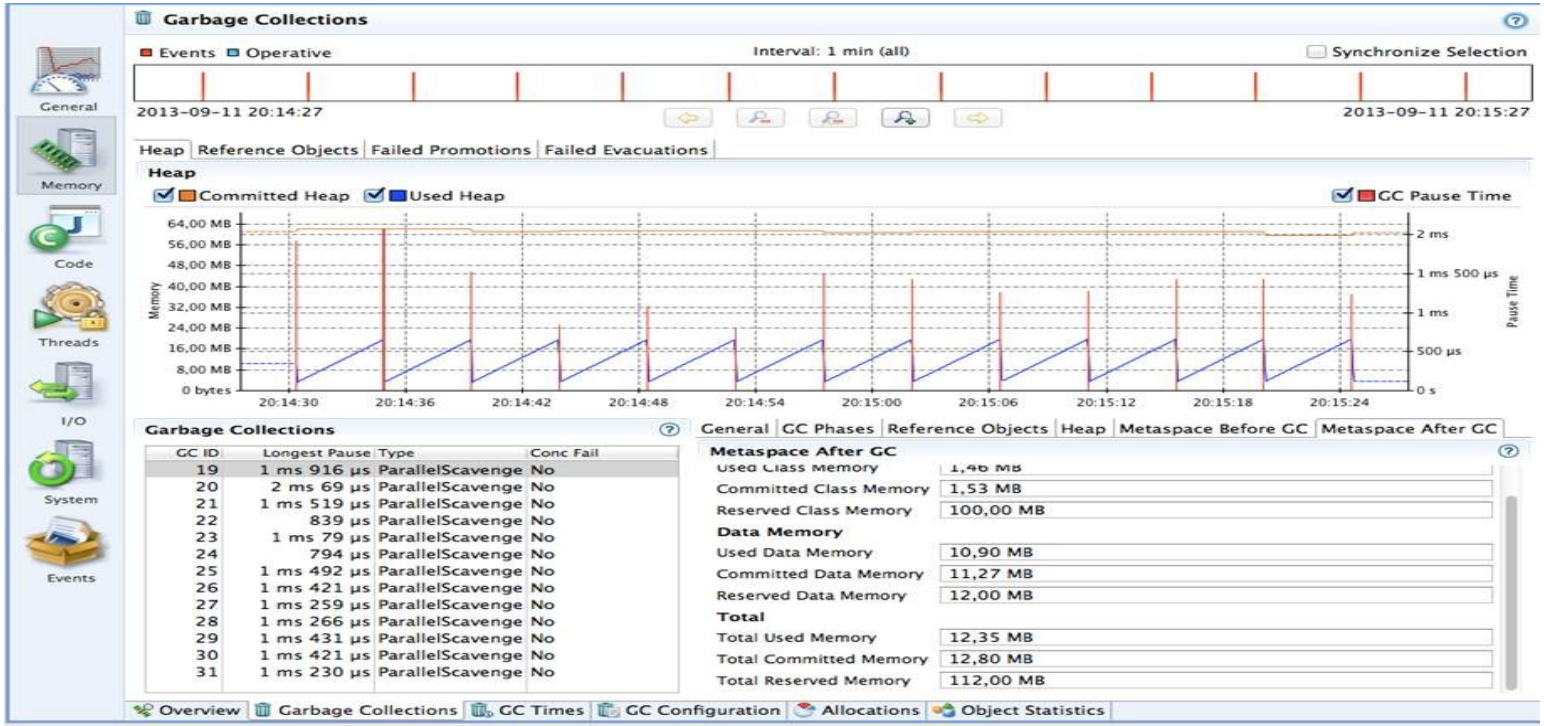
MemoryPoolMXBeans

- MemoryPoolMXBean with name “Metaspace”
  - Reports MemoryUsage for Metaspace plus Compressed Class Space
  - Max is MaxMetaspaceSize
- MemoryPoolMXBean with name “Compressed Class Space”
  - Only when -XX:+UseCompressedClassPointers is true
  - Reports MemoryUsage for CompressedClassSpace
  - Max is CompressedClassSpaceSize
  - Might be useful to set notifications because of fixed size

Other Tools for Metaspace

- jmap -permstat option renamed jmap -clstats
  - Prints class loader statistics of Java heap. For each class loader, its name, liveness, address, parent class loader, and the number and size of classes it has loaded are printed. In addition, the number and size of interned Strings are printed.
- jstat -gc option shows Metaspace instead of PermGen
- jcmd <pid> GC.class_stats
  - Gives detailed histogram of class metadata sizes
  - Start java with -XX:+UnlockDiagnosticVMOptions

Mission Control



Summary

- Hotspot metadata is now allocated in Metaspace
  - Chunks in mmap spaces based on liveness of class loader
- Compressed class pointer space is still fixed size but large
- Tuning flags available but not required
- Change enables other optimizations and features in the future
  - Application class data sharing
  - Young collection optimizations, G1 class unloading
  - Metadata size reductions and internal JVM footprint projects

Feedback and Questions?

- Suggestions and feedback welcome on mailing list

  hotspot-dev@openjdk.java.net

- Credits: Hotspot Permgen Removal Project Members

  Jon Masamitsu - Project Lead, Hotspot GC team
  Coleen Philimore - Hotspot Runtime team
  Stefan Karlsson - Hotspot GC team
  Mikael Gerdin - Hotspot GC team
  Erik Helin - Hotspot GC team
  Tom Rodriguez - formerly of the Hotspot Compiler team
  Roland Westrelin - Hotspot Compiler team
  Entire Hotspot team and OpenJDK community – Integrated 1 yr ago

MAKE THE FUTURE JAVA

JavaOne™

ORACLE


JavaOne™