Faster Java™ Applications: How To Tune The HotSpot™ Virtual Machine

Simon Ritter
Technology Evangelist
simon.ritter@sun.com

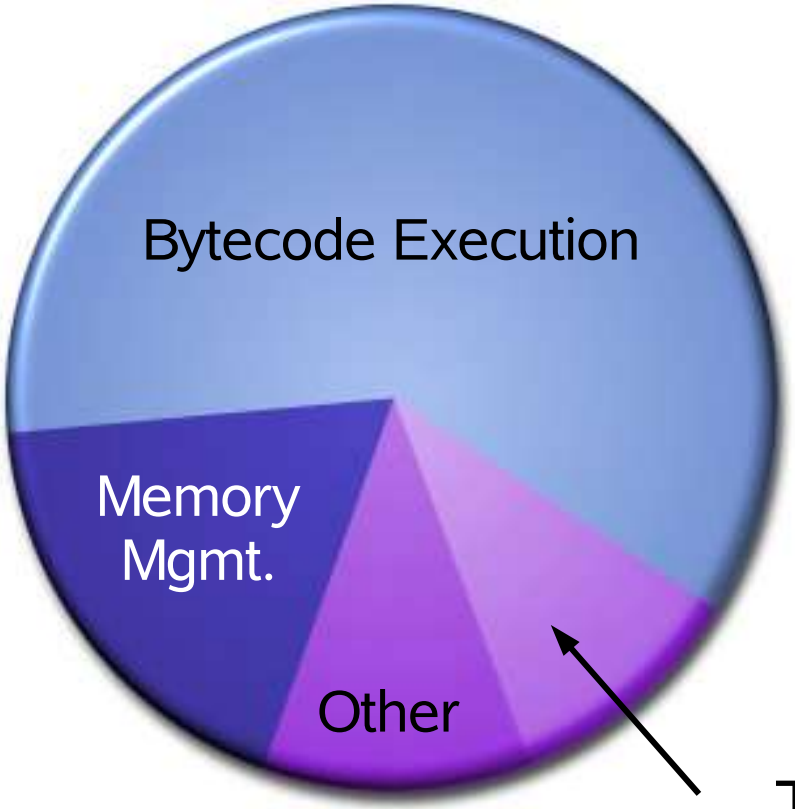
Sun™ Tech Days

Agenda

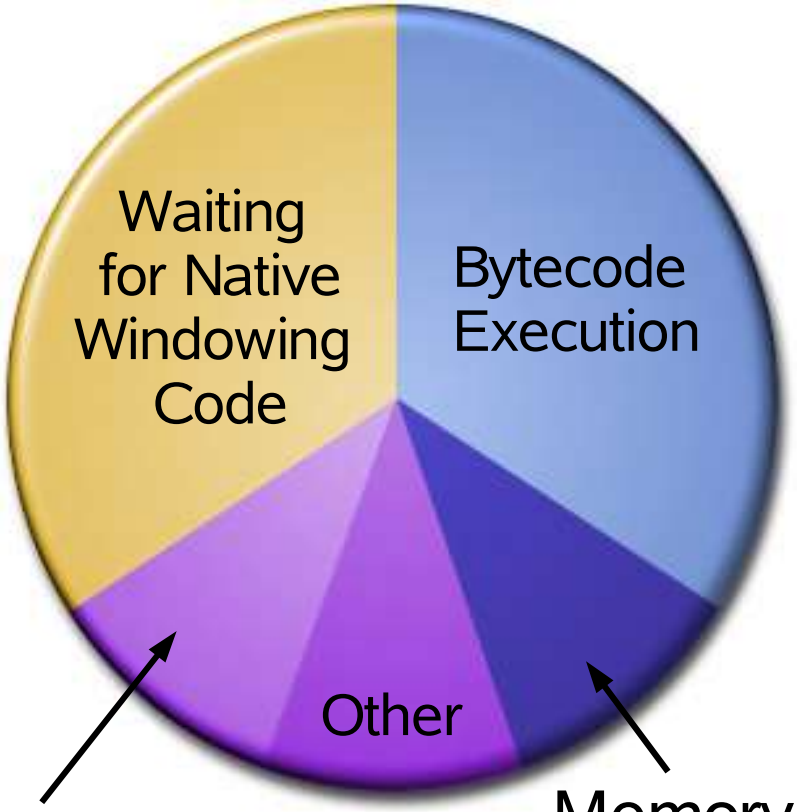
- Profile of JVM workload
- HotSpot™ VM internal architecture
- Garbage collection
- General HotSpot™ performance tuning
- Tuning HotSpot™ for application servers
- Further Information

JVM Workload

Server-side applications



Client-side applications



Thread Synchronization

Memory Mgmt.

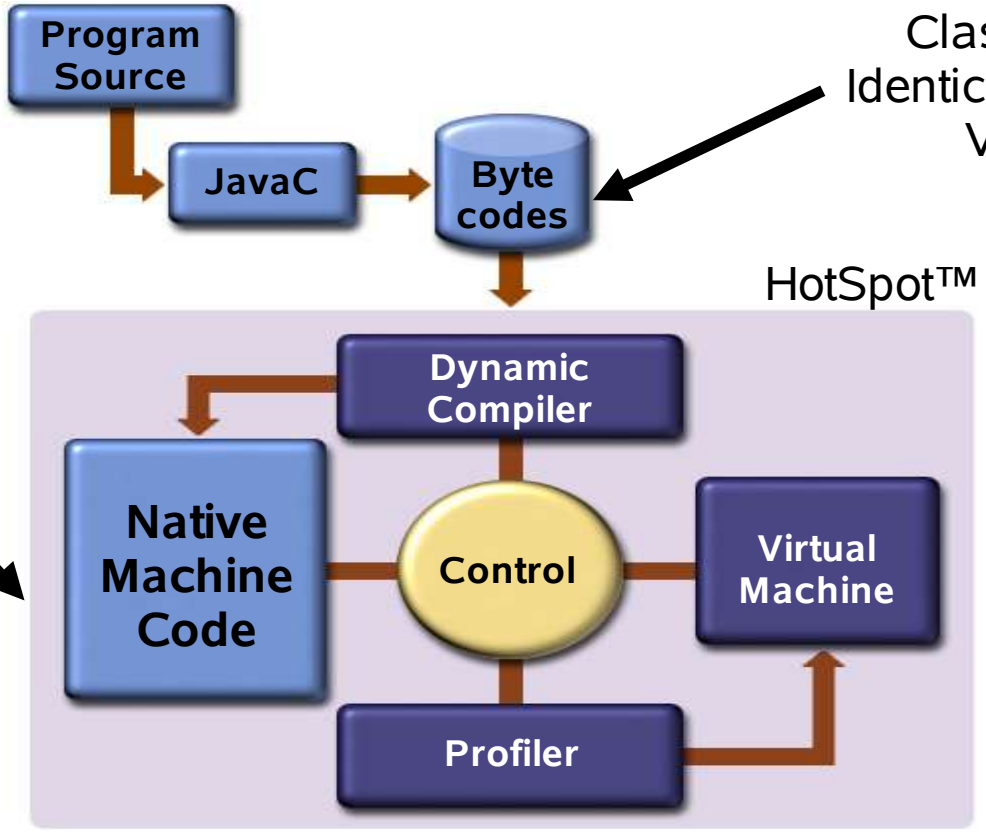
HotSpot™ Major Features

- Fast thread synchronization
- Adaptive compilation
- Generational garbage collector

Memory Model

- Handleless objects
- Two-word object headers
- Reflective data as objects
- Native thread support

Adaptive Compilation



Aggressive Inlining
Loop unrolling

Objects Need Storage Space

- Age old problems
  - How to allocate space efficiently
  - How to reclaim unused space (garbage) efficiently and reliably
- C (malloc and free)
- C++ (new and delete)
- Java™ (new and Garbage Collection)

GC Responsibilities

- Garbage detection
  - Distinguish live objects from garbage
  - Reference counting
  - Cyclic reference problem
- Garbage reclamation
  - Make space available to the running program again

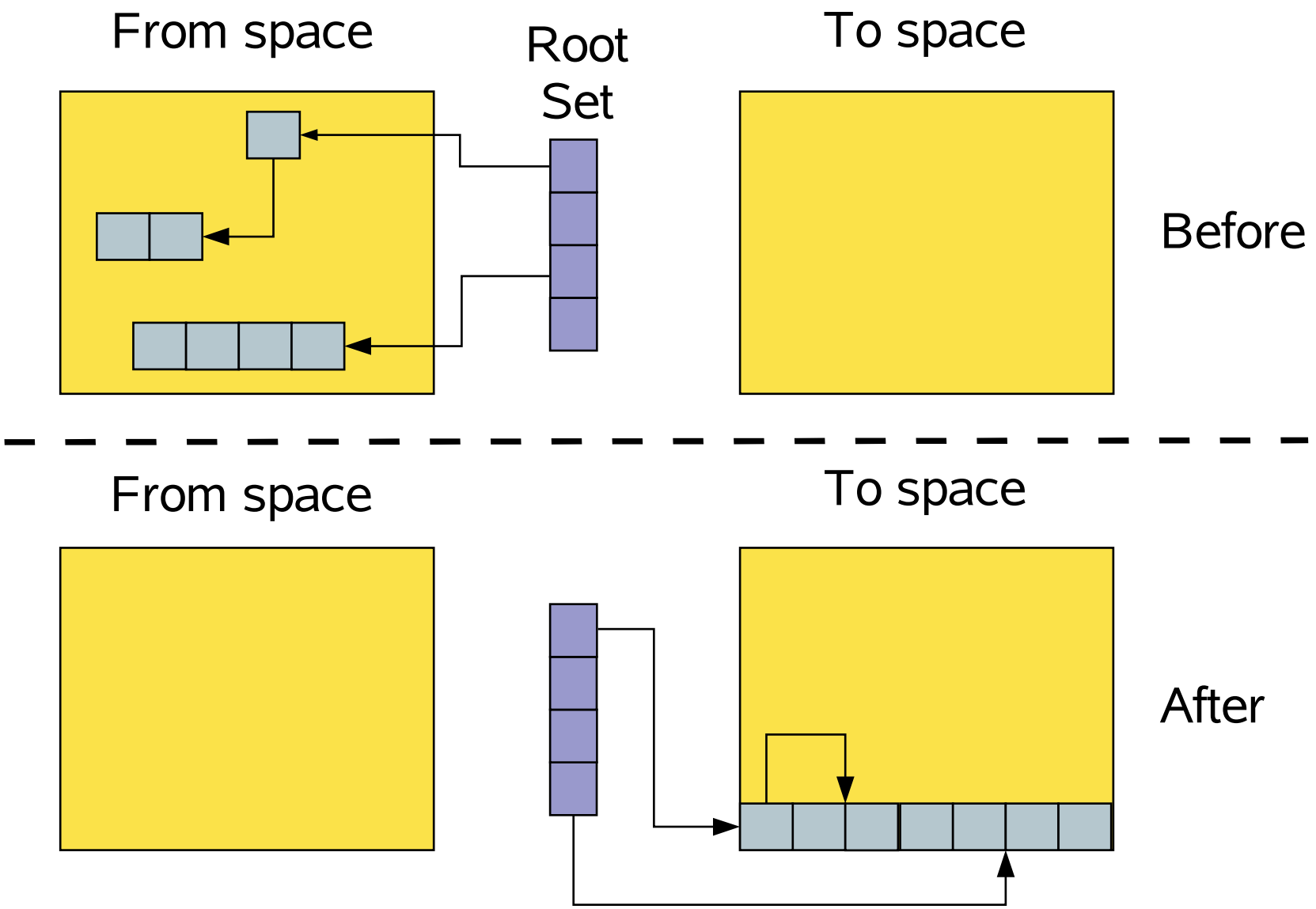
Object Lifetimes

- Most objects are very short lived
  - 80-98% of all newly allocated objects die within a few million instructions
  - 80-98% of all newly allocated objects die before another megabyte has been allocated
- This impacts heavily on choices for GC algorithms

Collector Algorithms

- Copying
- Mark - Sweep
- Mark - Compact
- Incremental
- Generational
- Parallel Copy
- Concurrent
- Parallel Scavenge

Copying GC


From space
Root Set
To space
Before
From space
To space
After

Copying GC

- Stop-the-world collector
- Very Efficient
  - Traverses object list and copies objects in a single cycle
  - Simultaneous detection and reclamation
- GC pause is directly proportional to total size of live objects
  - Bigger semi-spaces improve efficiency
  - Less frequent GC, more dead objects

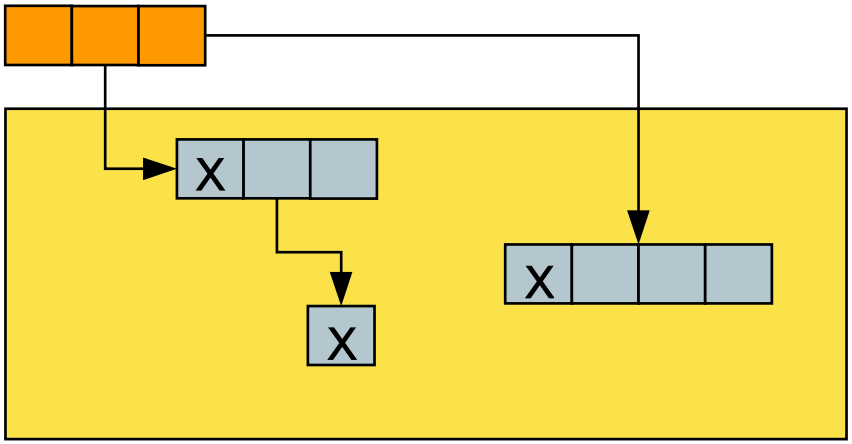
Mark – Sweep GC

- Stop-the-world collector
- Distinguish live objects from garbage
  - Traverse graph of pointer relationships
  - Mark objects that can be reached
- Reclaim the space
  - Heap space is “swept” for marked areas
  - Free space is added to a free list, ready for use

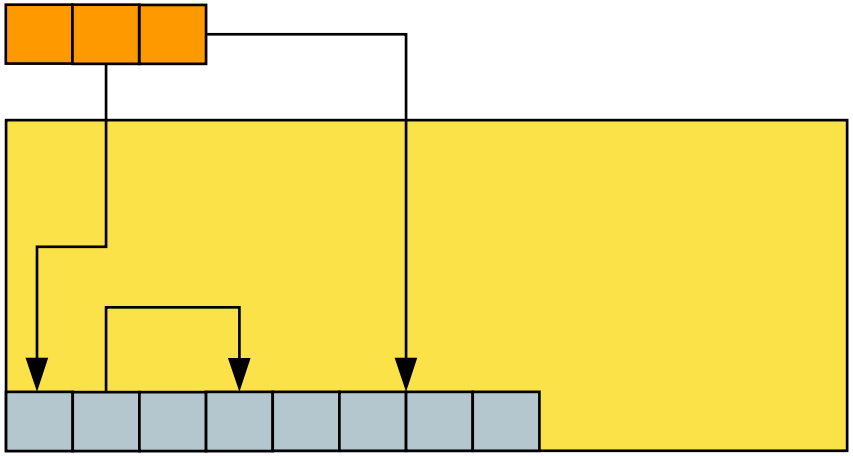
Mark – Sweep Problems

- Different-sized objects cause fragmentation
  - Multiple free lists for different-sized blocks
- Cost of collection proportional to size of heap
  - Not just live objects
- Locality of reference
  - New objects get interleaved with old objects
  - Bad for VM-based operating systems

Mark – Compact GC



Before



After

Mark – Compact GC

- Eliminates fragmentation issue of Mark-Sweep
- Allocation becomes stack-based
- Order of objects maintained
  - Locality of reference
- Requires multiple passes to complete
  - Mark live objects
  - Compute new location
  - Update pointers

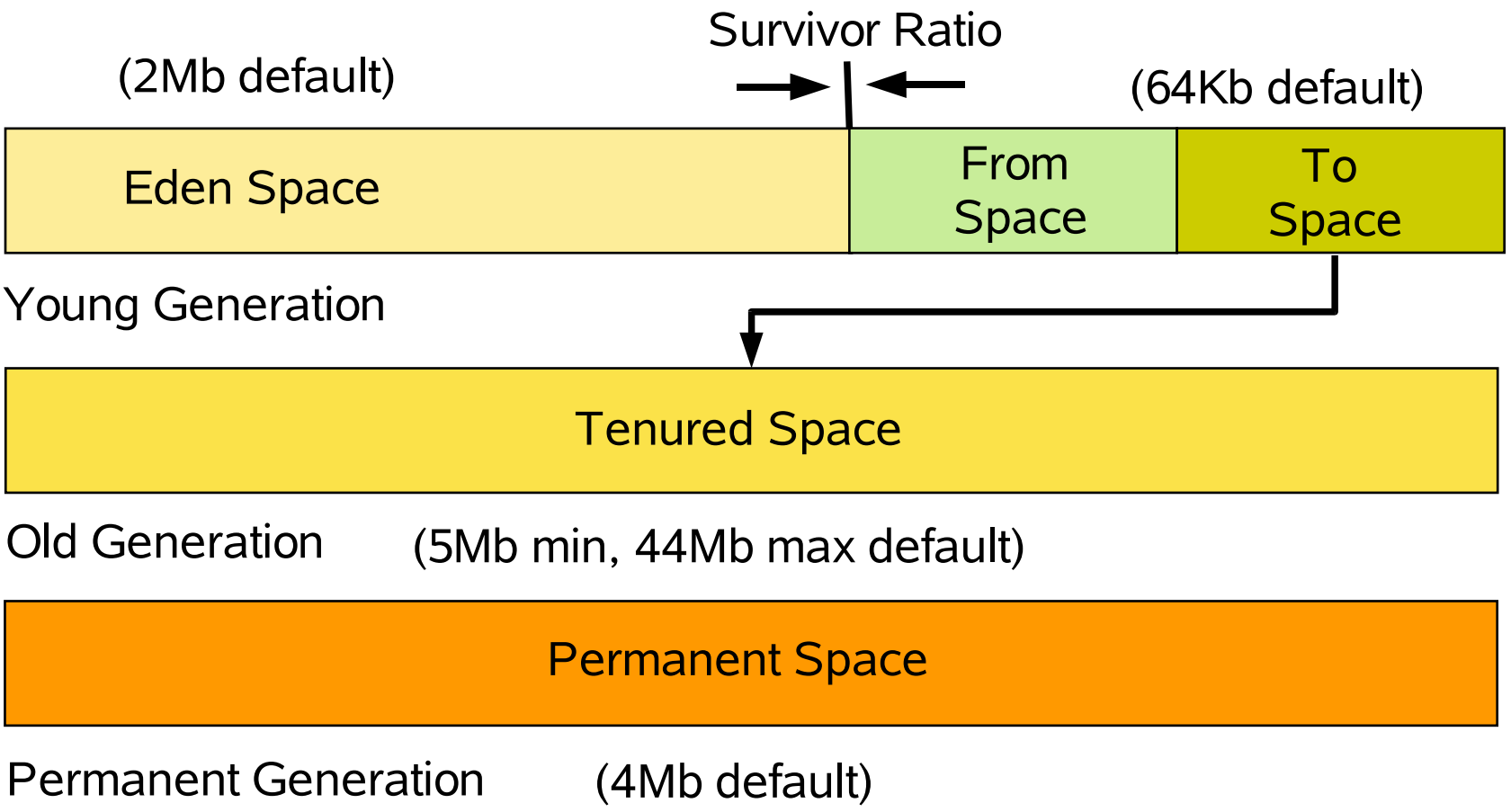
Incremental GC

- Stop-the-world impacts performance
  - Big heap, big pauses (00's – 000's ms)
- Interleave units of GC work with application work
- Problem is that references change while GC runs
  - Get floating garbage

Generational GC

- Old objects tend to live for a long time
  - GC can spend lots of time analysing and copying the same objects
- Generational GC divides heap into multiple areas (generations)
  - Objects segregated by age
  - New objects die more quickly, GC more frequent
  - Older generations collected less frequently
  - Different generations use different algorithms

HotSpot™ VM Heap Layout



Young Generation Heap Size

Sun™ Tech Days

Sun microsystems

Eden = NewSize –

((NewSize / (SurvivorRatio + 2)) * 2)

From Space = (NewSize – Eden / 2)

To Space = (NewSize – Eden) / 2)

- -XX:NewSize
- -XX:MaxNewSize
- -XX:NewRatio
- -XX:SurvivorRatio

Old Generation Heap Size

- Tenured generation
  - Objects with long lifetime

- -Xms
- -Xmx
- -XX:MinHeapFreeRatio
- -XX:MaxHeapFreeRatio

Permanent Heap Size

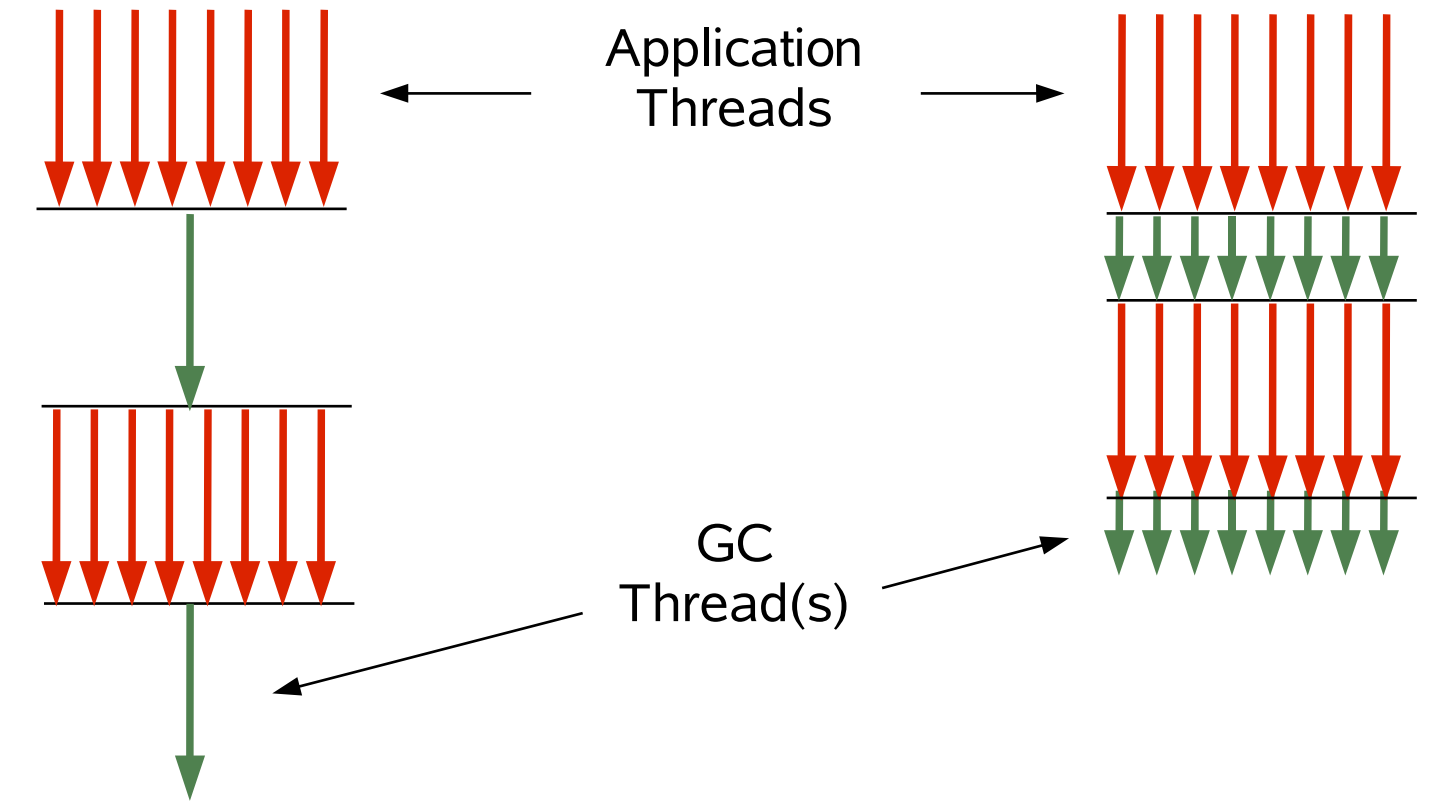
- Used to hold class files
- Default size is 4Mb

- -XX:PermSize
- -XX:MaxPermSize
- -Xnoclassgc

Parallel Copy GC

- Similar to copy-collector
  - Still stop-the-world
- Allocates as many threads as CPUs
  - Algorithm optimized to minimize contention
- Maximize work throughput
  - Work stealing

Parallel Copy GC



Single Threaded
Stop-the-world
collector

Parallel, multi-threaded
Stop-the-world
young generation collector

Parallel Copy Collector

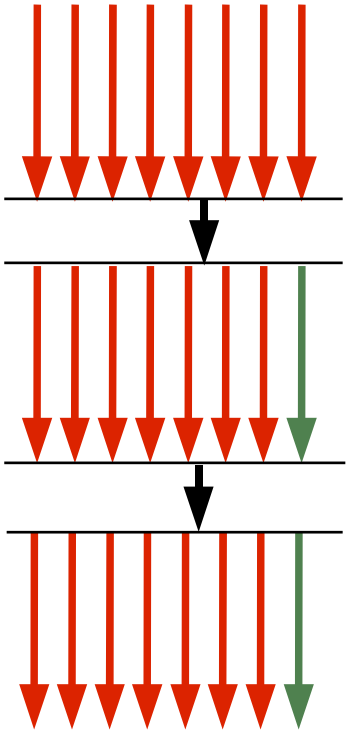
- -XX:+UseParNewGC
  - Default copy collector will be used on single CPU machines
- -XX:ParallelGCThreads=<*num*>
  - Default is number of CPUs
  - Can be used to force the parallel copy collector to be used on single a CPU machine

Concurrent GC

Sun™ Tech Days

Sun microsystems

-XX:+UseConcMarkSweepGC



ApplicationThreads

Stop-the-world initial mark phase

Concurrent mark phase

Stop-the-world re-mark phase

Concurrent sweep phase

Parallel Scavenge GC

- Stop-the-world
- Similar to parallel-copy collector
- Aimed at large young spaces (12-80Gb)
- Scales well with more CPUs
- Adaptive tuning policy
  - Survivor ratio
- Promotion undo to prevent out of memory

Parallel Scavenge Collector

- -XX:+UseParallelGC
- -XX:ParallelGCThreads=<*num*>
  - Control number of threads
- -XX:+UseAdaptiveSizePolicy
  - Automatically sizes the young generation and selects optimum survivor ratio

Factors Affecting GC

- Rate of object creation
- Object life spans
  - Temporary, intermediate, long
- Types of object
  - Size, complexity
- Relationships between objects
  - Difficulty of determining and tracking object references

Basic Approach To Tuning

- Profile, profile, profile!
- Use profile data to determine factors affecting performance
- Modify parameters to optimize performance
- Repeat

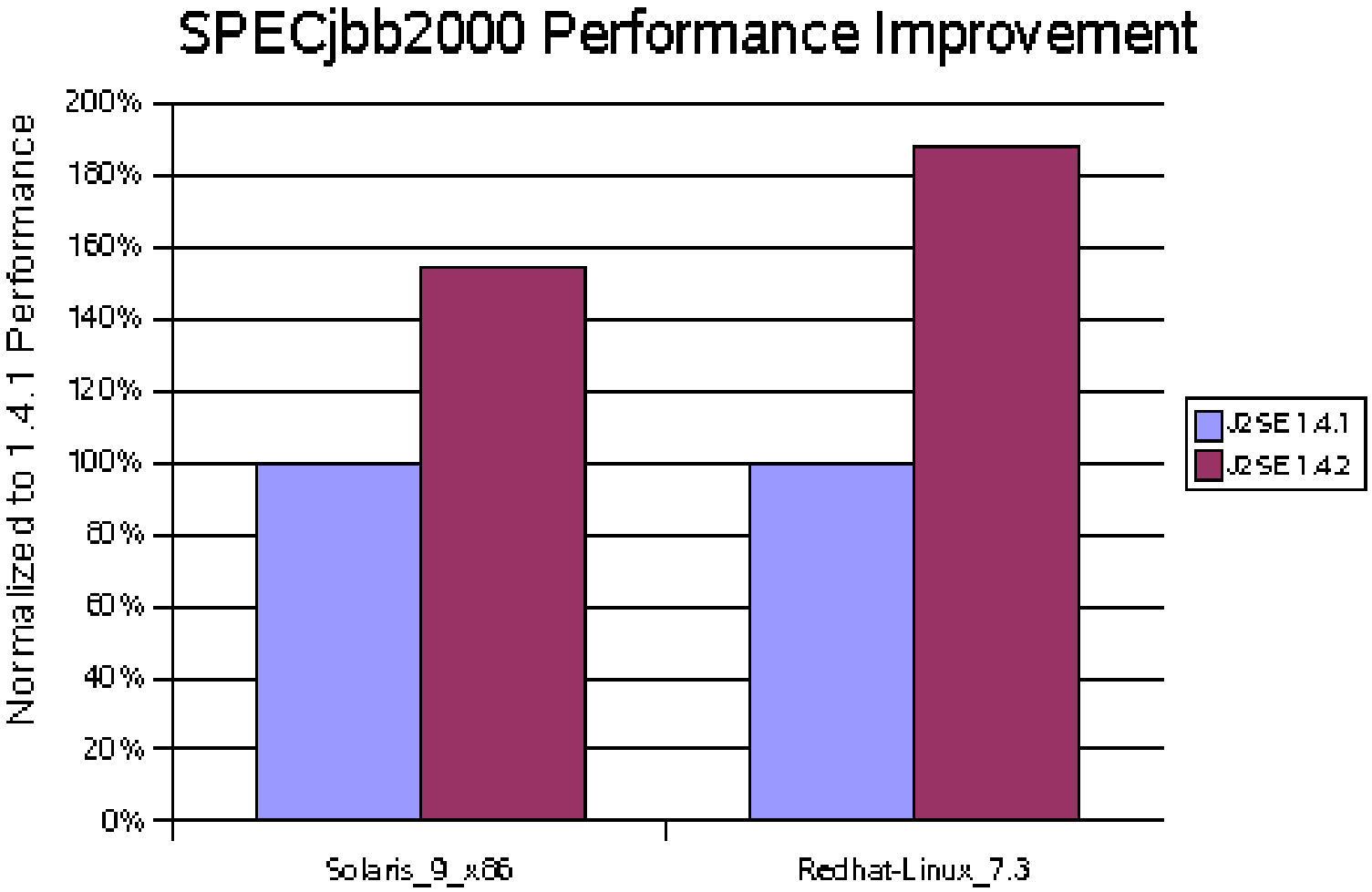
Profiling GC

- Simplest approach
  - -verbose:gc
  - -Xrunhprof
  - -XX:+PrintGCDetails
  - -XX:+PrintGCTimeStamps
  - -XX:+PrintHeapAtGC
    - Warning: very verbose

Quick Performance Fix

- Always upgrade to the latest version of the JDK/JRE
  - Sun is always working to improve performance
  - Sun is always working to reduce the number of 'undocumented features'

Performance Example



Changed implementation of **AggressiveHeap** option

Object Lifetimes

- Temporary
  - Die before encountering a young GC
- Intermediate
  - Die before being tenured to old space
- Long
  - Get promoted to old heap space
- Ratio of these has big impact on heap layout

Reducing Object Lifetimes

- Code inspection
  - Remove references when not required
  - Can do this explicitly with

```
objectRef = null;
```

- Avoid creating objects
  - Intermediate objects silently created when immutable object values change

Object Pooling

- Can be good for heavy weight objects
  - Database connections/threads
  - Reduce frequency of young GC
- Can also be bad
  - Pooling can be more expensive than creation/collection
  - Can violate good OO design principles

Disabling Tenuring

- Promote all live objects
  - No tenuring of objects in survivor spaces
  - Good for apps with few intermediate objects
- -XX:MaxTenuringThreshold=0
  - Number of times an object is copied in the survivor spaces
- -XX:SurvivorRatio=100
  - Ensures all of young generation is allocated to the eden space

Helping The GC

- Reduce state
  - Objects die before leaving eden
- Avoid references that span heaps
  - More work required to trace links between young and old spaces
- Flatten objects
  - Complex structures require additional work to determine live objects

Heap Sizing

- Extremely important to GC performance
- Factors to consider
  - Young GC frequency/collection time
  - Ratio and number of short, intermediate and long life objects
  - Promotion size
  - Old GC frequency/collection times
  - Old heap fragmentation/locality problems

Sizing The Young Heap

- Fragmentation is not an issue
  - Locality of reference could be
- Maximize collection of temporary objects
  - Reduces promotion & tenuring
- Minimize frequency of GC
- Rule of thumb: make it as large as possible
  - Given acceptable collection times

Sizing the Old Heap

- Ensure heap fits in physical memory
  - Paging and locality of reference issues
- larger young heap, smaller old heap
- Undersized heap can lead to fragmentation
- Oversized heap increases collection times
  - Locality of reference problems
  - Use ISM and Variable page sizes to alleviate

Intimate Shared Memory

- Designed for use on big memory Solaris machines
  - Don't use if memory requirements will cause paging
  - JDK1.3.1 introduced support for heaps > 2Gb
  - ISM uses larger page sizes (4Mb rather than 8Kb)
  - Locks pages into memory (no paging to disk)
  - -XX:+UseISM (Solaris Only)
  - -XX:+UsePermISM (Solaris Only)
  - -XX:+UseMPSS (Solaris 9 Only)
  - Need to change shm parameters in /etc/system

Aggressive Heap

- -XX:+UseAgressiveHeap
  - Must have min of 256MB RAM
  - Overall heap will be around 3850Mb
  - Thread allocation area 256MB
  - GC deferred as long as possible
  - Do not use -Xms or -Xmx with this
  - May cause stack space to run out
    - Use -Xss to compensate
  - Not suited to multi-app servers

HotSpot™ Thread Options

- -XboundThreads *
- -XX:+UseThreadPriorities
- -XX:+UseLWPSynchronisation **
- -XX:+AdjustConcurrency *

* Solaris Only

** SPARC Only

General Tuning Advice

- Allocate more memory to the JVM
  - 64Mb default is often too small
- Set -Xms and -Xmx to be the same
  - Increases predictability, improves startup time
- Set Eden/Tenured space ratio
  - Eden >50% is bad
  - Eden = 33%, Tenured = 66% seems to be good

Conclusions

- Understanding the virtual machine will help you tune performance
- Use profiling tools to find bottlenecks
- Adapt HotSpot™ parameters to your application
- Always use the latest JRE
- Sun is always improving Java™ performance

Further Information

- java.sun.com/blueprints/performance
- java.sun.com/products/hotspot
- research.sun.com/projects/jfluid
- developers.sun.com/dev/coolstuff/jvmstat
- Developer.java.sun.com/developer/technicalArticles/Programming/GCPortal

Q&A

Sun microsystems

Push your development further

Simon Ritter
Technology Evangelist

simon.ritter@sun.com

Sun™ Tech Days