開
放
的
열린
مفتوح
libre
मुक्त
ముక్త
livre
libero
ಮುಕ್ತ
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை


# JVM Internals, 0.75

Stefan Parvu
System Administrator

*http://www.nbl.fi/stefan.parvu*

# License, copyrights



This presentation includes graphics from **Open Clip Art Library**

The goal of the Open Clip Art Library is to provide the public with a huge collection of reusable art. Some highlights about this release can be seen in the RELEASE*.svg file. Also, look in the Examples directory for templates and tutorials on ways to make use of the clipart. For more information, including how you can contribute to this growing library, please see http://www.openclipart.org/ Public Domain Dedication Copyright-Only Dedication

# Agenda

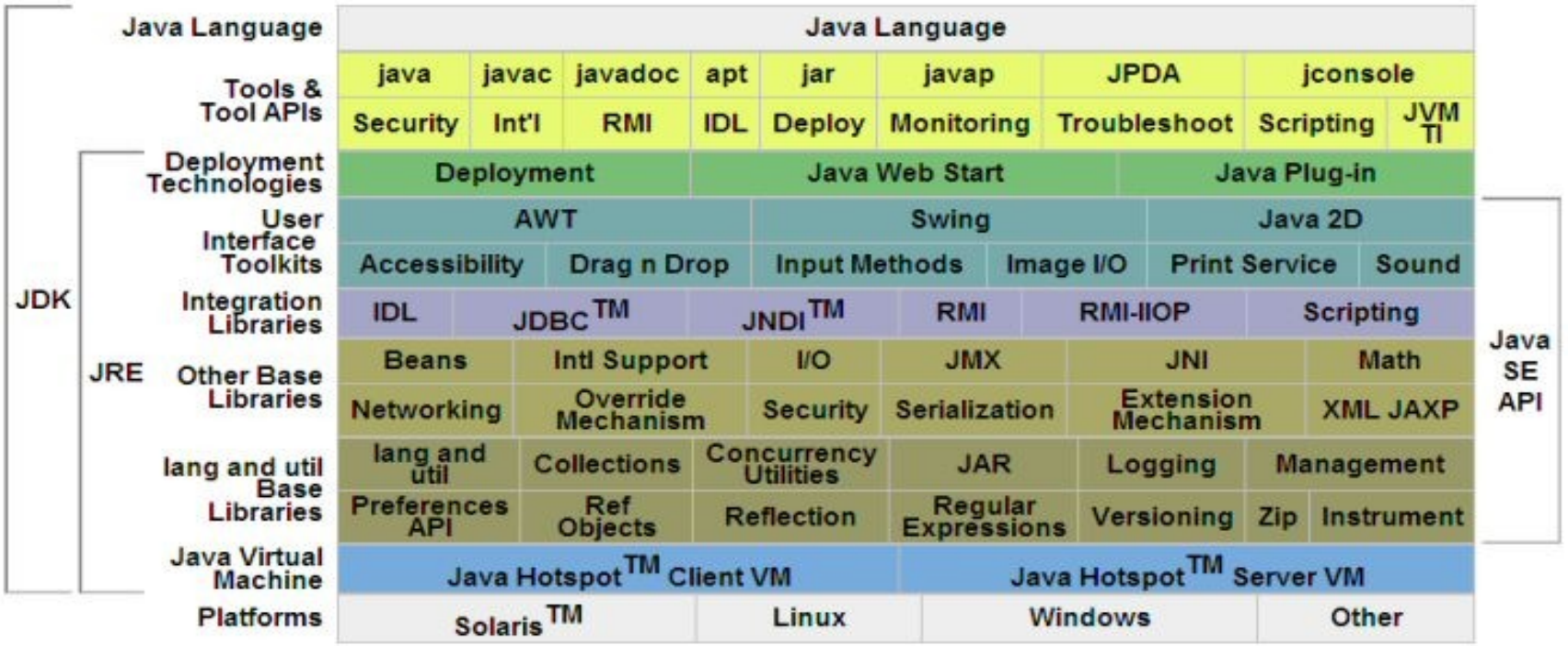
- Introduction to Java Virtual Machine
- HotSpot VM
  - Architecture
  - Memory management and Garbage Collection
  - Heap Sizing
- Java 5 and 6
  - What's new ?
- Troubleshooting

# Agenda

- **Introduction to Java Virtual Machine**
- HotSpot VM
  - Architecture
  - Memory management and Garbage Collection
  - Heap Sizing
- Java 5 and 6
  - What's new ?
- Troubleshooting

# Java Virtual Machine

- Part of the Java Platform
- Runs your Java application by implementing a virtual machine model, virtualized environment



# Java Virtual Machine, cont.

- Java Runtime Environment, JRE:
  - a virtual machine
  - a set of standard class libraries
  - usually used to run Java applications
  - not intended for development
- Java Development Kit, JDK:
  - intended for development
  - includes the JRE

# Java Virtual Machine, cont.

- Example JREs
  - Java Runtime Environment (JRE) 6 Update 4
  - Java Runtime Environment (JRE) 5.0 Update 14
- Example JDKs
  - JDK 6 Update 4
  - JDK 5.0 Update 14

- http://java.sun.com/javase/downloads/?intcmp=1281
- http://java.sun.com/javase/downloads/index_jdk5.jsp

# Java Virtual Machine, cont.

- Java Community Process
  - over 1000 corporate and individual participants
  - developer community
  - http://jcp.org/en/home/index
- Open Source: OpenJDK
  - source code available, anyone can participate
  - https://openjdk.dev.java.net/
  - http://www.sun.com/software/opensource/java/

# Java Virtual Machine, cont.

- Programs which run inside a JVM must be compiled in a portable binary format (your *.class* file)
- A set of *.class* files can be combines into a jar
- Safety: no user should be able to crash the host machine (Sandbox model, bytecode verification)
- The VM runtime subsystem executes the class or the jar file

# Java Virtual Machine, cont.

- Execution can be interpretation or just-in-time compile
- Interpretation is a slower approach
- Just-In-Time compilation
  - turns the bytecode into native code for that machine
  - is one of the most common methods of execution inside a VM
  - a short delay in the startup caused by the initial bytecode compilation

# Java Virtual Machine, cont.

- Examples:
  - Sun Java Virtual Machine, HotSpot
  - IBM Java Virtual Machine, J9 VM
  - BEA Java Virtual Machine, JRockit
  - Apache Java Virtual Machine, Harmony

# Agenda

- Introduction to Java Virtual Machine
- **HotSpot VM**
  - Architecture
  - Memory Management and Garbage Collection
  - Heap Sizing
- Java 5 and 6
  - What's new ?
- Troubleshooting

# HotSpot architecture

- HotSpot is Sun's Java Virtual Machine technology, the foundation for Java Standard Edition
- HotSpot technology: the VM continuously analyses the program performance for **hot spots,** frequent part of the program which are executed
- Hot spots are then highly optimized, excluding the less performance code

# HotSpot architecture, cont.

- **HotSpot Client VM:** used for light applications, reducing memory footprint and startup time
- **HotSpot Server VM:** used for enterprise applications, maximizing the speed
- Both options: *-server* or *-client* share the same HotSpot runtime, but use different compilers internally !

# HotSpot architecture, cont.

- ***-client***: does less computation, analyzing and compiling the code, resulting in a faster startup
- ***-server***: advanced static analysis and compilation techniques resulting in faster execution time for long lasting applications

**Note:**
Check your production environment VM options !
Are you running HotSpot VM using client or server compilers !?

# HotSpot architecture, cont.

- HotSpot Client VM:

```
java version "1.5.0_11"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)
Java HotSpot(TM) Client VM (build 1.5.0_11-b03, mixed mode, sharing)
```

- HotSpot Server VM:

```
java version "1.5.0_11"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)
Java HotSpot(TM) Server VM (build 1.5.0_11-b03, mixed mode)
```

# HotSpot architecture, cont.

- **Operation mode:** interpreted, compiled or mixed
- **Native thread support:** using the operating system's stack and threading model
- **Garbage Collection:** based on generational model, support several algorithms to manage the objects in different generations. Objects allocated on heap. Automatically freeing unused objects – *when the objects are not referenced anymore by the application*

# HotSpot architecture, cont.

- **64bit safe:** when needed large heaps, on server class machines with lots of RAM. This is selectable via *-d32* or *-d64* options and applicable for HotSpot Server VM
- **Adaptive Optimizations:** the code first run through an interpreter and then is analyzed to detect the hot spots, the most called parts of the code. Finally the hot spots are then processed by a native-code optimizer. This way the memory footprint is decreased

# HotSpot architecture, cont.

- **Method inlining:** the hot spots are native compiled and the Java VM performs method inlining for that code. Reduces the time of method invocations
- **Dynamic deoptimization:** new classes might be loaded anytime into the VM. The Java VM should be able to reorganize during this process: dynamically deoptimize and then apply the optimizations if needed. Server and Client VM supports deoptimization

# HotSpot architecture, cont.

- **Scalability:** the new ergonomics model found in Java 5. The Java VM automatically adjusts itself based on server configuration: a client or server VM will be started and a GC algorithm will be selected
- **Serviceability:** Integration for profiling tools, debuggers or monitors using JVM Tools Interface. Supports full speed debugging, error reporting from JVM.

# HotSpot architecture, cont.

- Memory Management in HotSpot
  - automatic memory management
  - objects allocated and deallocated from Heap
  - memory is managed in generations, pools of objects
  - the Garbage Collector, GC is managing the process when one pool of objects fills up
  - easy to control the VM memory settings using several JVM options

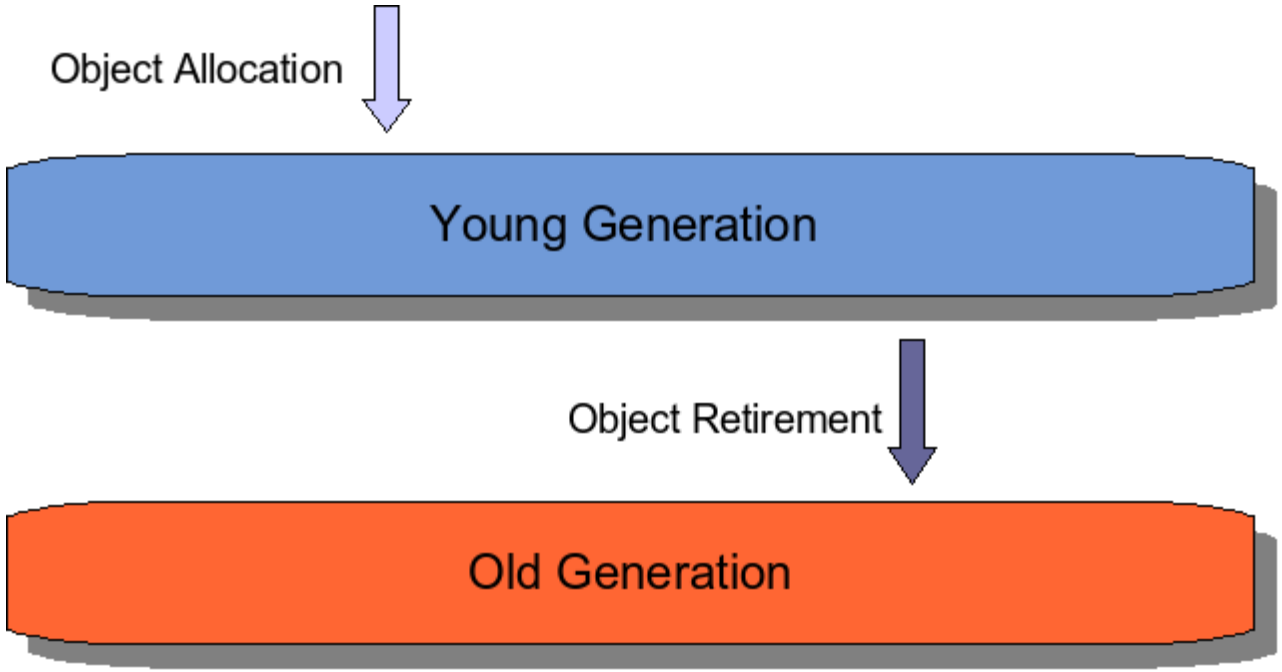
# HotSpot architecture, cont.

- The Garbage Collector
  - based on a generational model
  - responsible for: allocating memory, making sure all referenced objects are in memory and freeing the memory after the objects are no longer referenced
  - objects: live and dead
  - GC will try to garbage all dead objects
  - Several GC algorithms
  - easy control for each GC algorithm, using JVM options

# HotSpot architecture, cont.

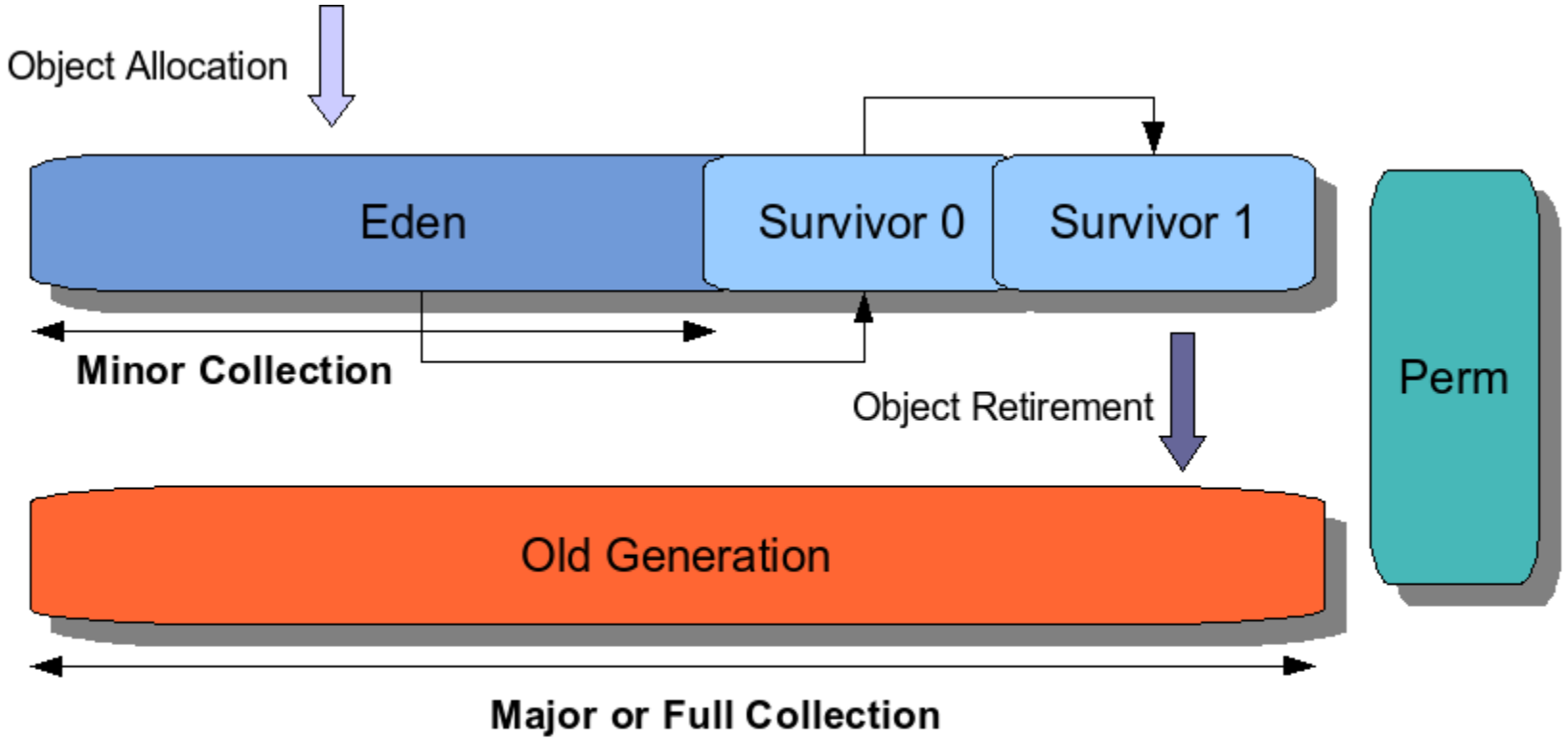
- The Garbage Collector
  - Safe and Efficient: all live objects must never be freed and the time for collection should not take very long
  - In real life it very much depends: time, space and frequency
  - Fragmentation: the memory which has been freed could be found in different places. This will make harder to allocate a large object. Compaction will help during GC activity
  - Scalability: for large CPU servers GC should be scalable and it should not turn into a bottleneck

# HotSpot architecture, cont.




The generational model of Garbage Collection

# HotSpot architecture, cont.




The mechanics of Garbage Collection

# HotSpot architecture, cont.

- The Garbage Collector goals
  - **Throughput**: the total time **not** spent in garbage collection
  - **Pause Time**: the time during the application is paused which garbage collection takes place
  - **Frequency of collection**: how often the GC is working
  - **Footprint**: the Heap size

# HotSpot architecture, cont.

- **Serial Collection**
  - Only one thing happens at the time. If the Serial collector is used on a multi CPU machine only one CPU is utilized
- **Parallel Collection**
  - The garbage collector job is spread across multiple jobs and multiple CPUs if available
  - The execution is faster but at the expenses of fragmentation

# HotSpot architecture, cont.

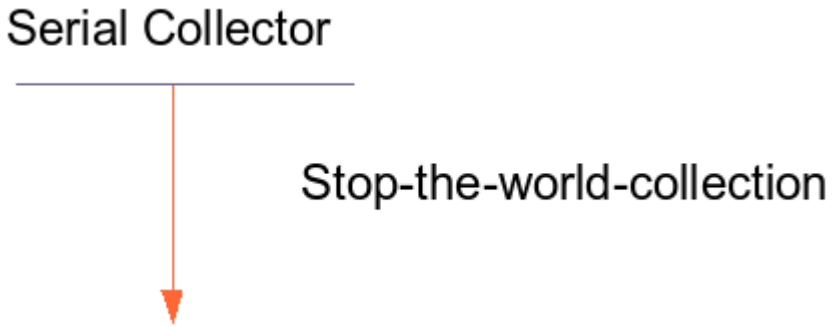
- **Stop-the-World Collection**
  - the execution of the application is totally suspended during this phase.
  - very simple, the entire Heap is frozen
- **Concurrent Collection**
  - many garbage collection jobs can be executed at the same time the application is running
  - some of the work is done concurrently some is done as a short Stop-of-World collections
  - operates on live and large heaps

# HotSpot architecture, cont.

- **Compacting Collection**
  - The memory is compacting, resulting in easy and fast allocations
  - Moves all live objects to a contiguous space
- **Non-Compacting Collection**
  - Does not move all live objects
  - Finishes faster
  - Expensive allocations in this mode. Searches for contiguous are of memory

# HotSpot architecture, cont.

- **Serial Collector**
  - Young and Old generations are collected serially



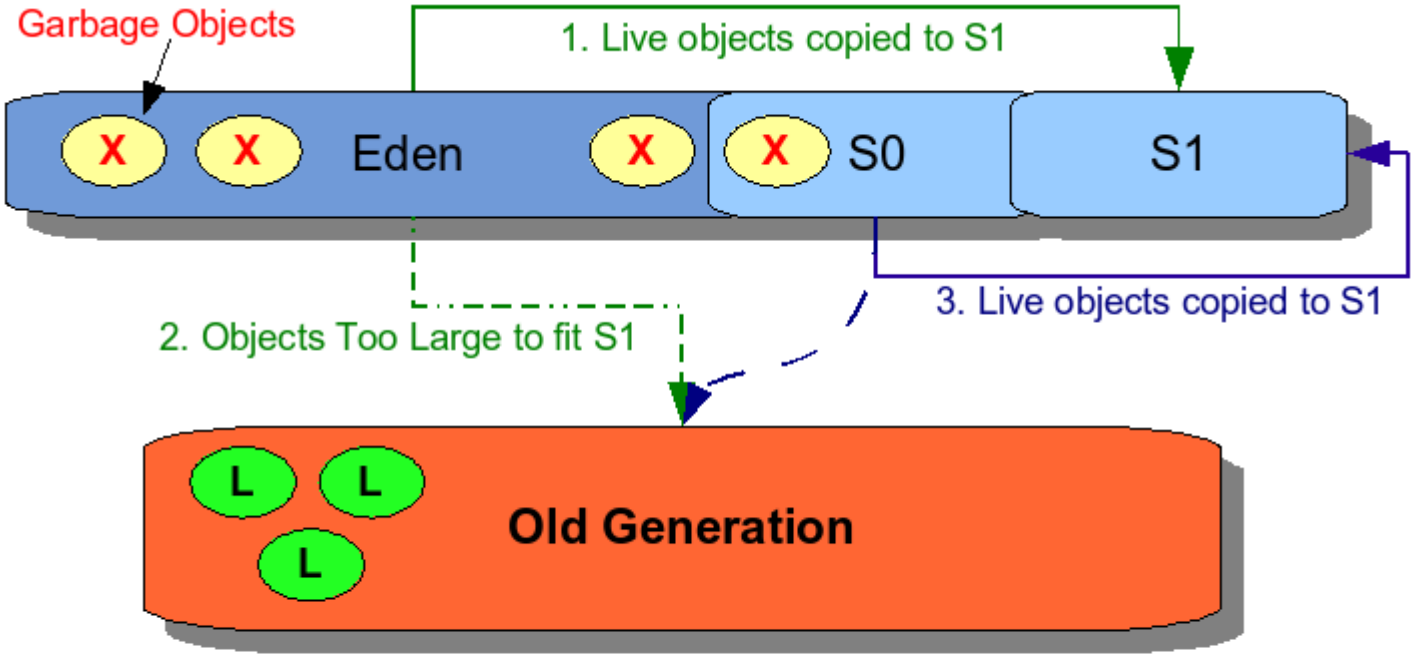

  - default on client type of hardware or manually using `-XX:+UseSerialGC` option
  - the application is paused during this operation , the collection is being done serially, using one CPU, stop-the-world method

# HotSpot architecture, cont.

- **Serial Collector**
  - usage: applications which run in a client type of hardware and don't have a requirement for low pause times
  - Young generation collection
    - live objects from Eden are copied to the initial free survivor space S1, the larger ones are moved directly to Old
    - The live objects from occupied survivor space S0 are copied to the S1, the old ones retire to Old

# HotSpot architecture, cont.

- **Serial Collector**
  - Objects marked with X will be garbaged and space freed, S1 and S0 swap positions at the end




Serial Collector in Young Generation

# HotSpot architecture, cont.

- **Serial Collector**
  - Old generation collection
    - Old and Permanent generations are done using mark-sweep-compact method
    - Mark phase: marks all live objects
    - Sweep phase: sweeps over the heap identifying the garbage
    - Slide phase: the GC performs a sliding compaction, sliding the live objects towards the start of the Heap and maintaining the rest compact. This help future allocations.

# HotSpot architecture, cont.

- **Serial Collector,** -verbose:gc

[GC 4692K->4278K(5220K), 0.0018318 secs]
[GC 4790K->4445K(5220K), 0.0022160 secs]
[GC 4957K->4597K(5220K), 0.0021238 secs]
[GC 5109K->4777K(5348K), 0.0020124 secs]
**[Full GC 4777K->3682K(5348K), 0.0907230 secs]**
[GC 4193K->3715K(6716K), 0.0013422 secs]
[GC 4672K->4652K(6716K), 0.0011858 secs]
**[Full GC 4652K->4509K(6716K), 0.0848523 secs]**
[GC 6484K->6467K(8160K), 0.0011248 secs]
**[Full GC 6467K->6467K(8160K), 0.0877565 secs]**
[GC 10803K->10474K(11676K), 0.0019494 secs]
**[Full GC 10366K->6250K(11676K), 0.0947010 secs]**

# HotSpot architecture, cont.

- **Serial Collector,** -verbose:gc -XX:+PrintGCDetails

[GC [DefNew: 576K->64K(576K), 0.0032166 secs] 3645K->3310K(3944K), 0.0034281 secs]

**[GC [DefNew: 576K->64K(576K), 0.0031091 secs][Tenured: 3423K->3061K(3496K), 0.0702154 secs] 3822K->3061K(4072K), 0.0737055 secs]**

[GC [DefNew: 512K->64K(576K), 0.0025877 secs] 3573K->3237K(5680K), 0.0027977 secs]

[GC [DefNew: 567K->64K(576K), 0.0049749 secs] 3740K->3407K(5680K), 0.0050272 secs]

[GC [DefNew: 576K->64K(576K), 0.0022290 secs] 4805K->4446K(5680K), 0.0022749 secs]

**[GC [DefNew: 576K->64K(576K), 0.0040543 secs][Tenured: 5209K->4170K(5232K), 0.1937438 secs] 5591K->4170K(5808K), 0.1980173 secs]**

[GC [DefNew: 512K->63K(576K), 0.0037735 secs] 4682K->4234K(7528K), 0.0038244 secs]

**[GC [DefNew: 177K->59K(576K), 0.0015195 secs][Tenured: 6530K->6582K(6952K), 0.0866866 secs] 6611K->6582K(7528K), 0.0884189 secs]**

**[GC [DefNew: 156K->55K(896K), 0.0013317 secs][Tenured: 10400K->6586K(10972K), 0.0854252 secs] 10525K->6586K(11868K), 0.0868328 secs]**

# HotSpot architecture, cont.

- **Serial Collector**, -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

**1.680**: [GC 1.680: [DefNew: 512K->64K(576K), 0.0023637 secs] 2746K->2342K(4304K), 0.0026758 secs]

1.772: [GC 1.772: [DefNew: 576K->64K(576K), 0.0053333 secs] 2854K->2436K(4304K), 0.0056777 secs]

1.819: [GC 1.819: [DefNew: 576K->63K(576K), 0.0023710 secs] 2948K->2580K(4304K), 0.0027122 secs]

2.085: [GC 2.086: [DefNew: 576K->64K(576K), 0.0032957 secs] 3550K->3153K(4304K), 0.0036396 secs]

2.117: [GC 2.117: [DefNew: 575K->64K(576K), 0.0037193 secs] 3665K->3337K(4304K), 0.0040632 secs]

2.144: [GC 2.144: [DefNew: 576K->64K(576K), 0.0033613 secs] 3849K->3512K(4304K), 0.0037018 secs]

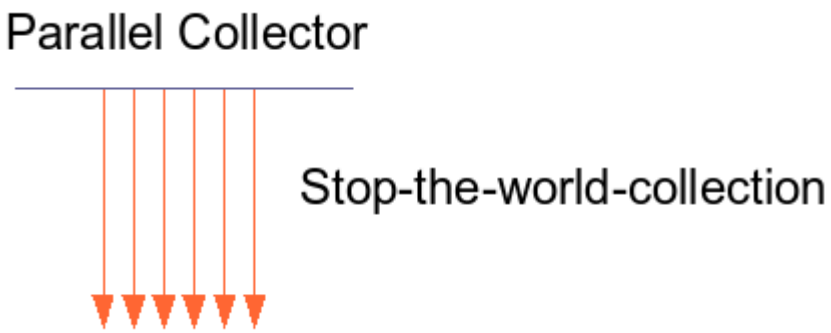
**2.235: [GC 2.235: [DefNew: 576K->64K(576K), 0.0050588 secs]2.240: [Tenured: 3815K->3152K(3856K), 0.0756689 secs] 4210K->3152K(4432K), 0.0809025 secs]**

# HotSpot architecture, cont.

-Xloggc:gc.log
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

# HotSpot architecture, cont.

- **Parallel Collector**




  - used to collect Young and the Old generation, using a stop-of-world type of collection
  - also known as the throughput collector

# HotSpot architecture, cont.

- **Parallel Collector**
  - default on server type of hardware or manually using `-XX:+UseParallelGC` option
  - the application is still paused during this operation, but the collection is performed in parallel on multiple CPUs
  - usage: applications which run on servers with more than one CPU and which don't have time constraints: batch processing, billing, payroll, scientific computing

# HotSpot architecture, cont.

- **Parallel Collector**
  - Young generation collection
    - a parallel version of the Serial collector
    - performs the collection in parallel, using many CPUs
    - increases application throughput
  - Old generation collection
    - similar as for Serial Collector

# HotSpot architecture, cont.

- **Parallel Collector**

  -Xloggc:gc.log -XX:+PrintGCDetails

  -XX:+PrintGCTimeStamps

1.369: [GC [PSYoungGen: 3072K->501K(3584K)] 3072K->622K(36352K), 0.0089261 secs]

1.918: [GC [PSYoungGen: 3573K->505K(3584K)] 3694K->1108K(36352K), 0.0068211 secs]

221.580: [GC [PSYoungGen: 9984K->3522K(19392K)] 26144K->22754K(56256K), 0.0270233 secs]

221.744: [GC [PSYoungGen: 19093K->3112K(20032K)] 38324K->28918K(56896K), 0.0176338 secs]

221.900: [GC [PSYoungGen: 18681K->3112K(20480K)] 44487K->37370K(57344K), 0.0202805 secs]

**221.921: [Full GC [PSYoungGen: 3112K->0K(20480K)] [PSOldGen: 34258K->23454K(57344K)] 37370K->23454K(77824K) [PSPermGen: 15535K->15535K(36864K)], 0.1227910 secs]**

# HotSpot architecture, cont.

- **Parallel Compacting Collector**
  - new GC algorithm with J2SE 5.0 Update 6
  - used to collect the Young or Old generations using a totally new algorithm, support compacting
  - enabled using `-XX:+UseParallelOldGC` option
  - usage: applications which run on machines with more than one CPU and have a requirement for low pause times. Not suitable for large machines where many applications are running at the same time – a single application which can use the entire CPU capacity

# HotSpot architecture, cont.

- **Parallel Compacting Collector**
  - Young generation collection
    - similar method as in Parallel collector
  - Old generation collection
    - each generation is given a certain fixed size region
    - Mark phase: all live objects are marked in parallel, and the data for the object's region is updated
    - Summary phase: operates on regions and not objects
    - Compaction phase:

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - The low-latency collector, important for fast response time
  - Application is paused for very short periods of time
  - It is not a compacting collector
  - enabled using `-XX:+UseConcMarkSweepGC` option
  - used usually with large heaps – the application is allowed to run during the collection phase, meaning the Old generation can grow in size

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - supports an incremental mode: concurrent phases are processed incremental
  - usage: applications which require short garbage collection pauses and which can share the CPU, the CMS collector takes CPU cycles from the application for its own GC activity, web servers, app servers – applications with large Old generations and long lived data objects

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - Young generation collection
    - same as the Parallel collector
  - Old generation collection
    - decreases dramatically the old generation pause time
    - starts a collection early enough to avoid a Full GC
    - **Initial mark phase** – identifies the initial set of live objects, STW (stop-the-world)
    - **Concurrent Marking phase** – marks all live objects; some of the objects might not be marked so a second cycle is needed to make sure all objects are marked

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - Old generation collection
    - **Precleaning phase** – tries to minimize the Remark phase time, by marking all objects which were previously modified by the application threads
    - **Remark phase** – STW, rescans all objects, by revisiting any object which was modified in the previous phase. Here the collector uses multiple threads to finish faster the remark phase
    - **Sweep phase** – sweeps over the heap to reclaim all the garbage which has been identified

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

14.505: [GC 14.505: [DefNew: 3968K->0K(4032K), 0.0147255 secs] 9382K->6234K(16320K), 0.0148194 secs]

14.523: [GC [1 **CMS-initial-mark**: 6234K(12288K)] 6247K(16320K), 0.0017829 secs]

14.524: [CMS-concurrent-mark-start]

14.587: [**CMS-concurrent-mark**: 0.063/0.063 secs]

14.587: [**CMS-concurrent-preclean-start**]

14.588: [CMS-concurrent-preclean: 0.000/0.000 secs]

14.588: [GC[YG occupancy: 13 K (4032 K)]14.588: [Rescan (non-parallel) 14.588: [grey object rescan, 0.0001824 secs]14.588: [root rescan, 0.0007562 secs], 0.0009850 secs]14.589: [weak refs processing, 0.0018470 secs] [1 CMS-remark: 6234K(12288K)] 6247K(16320K), 0.0029740 secs]

14.595: [**CMS-concurrent-sweep-start**]

14.603: [CMS-concurrent-sweep: 0.007/0.007 secs]

14.624: [**CMS-concurrent-reset-start**]

14.627: [CMS-concurrent-reset: 0.004/0.004 secs]

22.807: [GC 22.807: [DefNew: 3968K->0K(4032K), 0.0090208 secs] 9109K->5890K(16320K), 0.0091121 secs]

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

24.626: [GC [1 CMS-initial-mark: 8881K(12288K)] 10035K(16320K), 0.0007227 secs]

24.627: [CMS-concurrent-mark-start]

24.682: [CMS-concurrent-mark: 0.055/0.055 secs]

24.682: [CMS-concurrent-preclean-start]

24.682: [CMS-concurrent-preclean: 0.000/0.000 secs]

24.683: [GC[YG occupancy: 1154 K (4032 K)]24.683: [Rescan (non-parallel) 24.683: [grey object rescan, 0.0002292 secs]24.683: [root rescan, 0.0007530 secs], 0.0010290 secs]24.684: [weak refs processing, 0.0023396 secs] [1 CMS-remark: 8881K(12288K)] 10035K(16320K), 0.0034769 secs]

24.686: [CMS-concurrent-sweep-start]

24.695: [CMS-concurrent-sweep: 0.009/0.009 secs]

24.695: [CMS-concurrent-reset-start]

24.699: [CMS-concurrent-reset: 0.004/0.004 secs]

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

58.456: [GC 58.456: [DefNew: 3456K->0K(4032K), 0.0078289 secs] 40036K->38255K(45704K), 0.0079163 secs]

58.466: [GC [1 CMS-initial-mark: 38255K(41672K)] 38854K(45704K), 0.0006956 secs]

58.467: [CMS-concurrent-mark-start]

58.523: [CMS-concurrent-mark: 0.056/0.056 secs]

58.523: [CMS-concurrent-preclean-start]

58.523: [CMS-concurrent-preclean: 0.000/0.000 secs]

58.525: [GC[YG occupancy: 598 K (4032 K)]58.525: [Rescan (non-parallel) 58.525: [grey object rescan, 0.0003533 secs]58.525: [root rescan, 0.0006906 secs], 0.0010895 secs]58.526: [weak refs processing, 0.0024445 secs] [1 CMS-remark: 38255K(41672K)] 38854K(45704K), 0.0036423 secs]

58.530: [CMS-concurrent-sweep-start]

58.535: [CMS-concurrent-sweep: 0.005/0.005 secs]

58.535: [CMS-concurrent-reset-start]

58.538: [CMS-concurrent-reset: 0.003/0.003 secs]

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

**58.558: [Full GC 58.558: [CMS: 13971K->13842K(41672K), 0.1547712 secs] 14579K->13842K(45704K), [CMS Perm : 14788K->14748K(24648K)], 0.1551333 secs]**

58.789: [GC 58.789: [DefNew: 3965K->0K(4032K), 0.0051317 secs] 17807K->17395K(45704K), 0.0058756 secs]

58.856: [GC 58.856: [DefNew: 2562K->0K(4032K), 0.0016115 secs] 19958K->17396K(45704K), 0.0016866 secs]

58.869: [GC 58.869: [DefNew: 3908K->0K(4032K), 0.0049839 secs] 21304K->21303K(45704K), 0.0050737 secs]

58.957: [GC 58.957: [DefNew: 3968K->0K(4032K), 0.0063983 secs] 25271K->24592K(45704K), 0.0064776 secs]

**59.275: [Full GC 59.275: [CMS: 24592K->16145K(41672K), 0.1323715 secs] 26912K->16145K(45704K), [CMS Perm : 14832K->14832K(24648K)], 0.1361513 secs]**

59.752: [GC 59.752: [DefNew: 3690K->0K(4032K), 0.0059947 secs] 19836K->18431K(45704K), 0.0063151 secs]

**59.803: [Full GC 59.803: [CMS: 18431K->14558K(41672K), 0.1263113 secs] 20150K->14558K(45704K), [CMS Perm : 14833K->14833K(24776K)], 0.1268321 secs]**

60.018: [GC 60.019: [DefNew: 3608K->0K(4032K), 0.0038720 secs] 18166K->16791K(45704K), 0.0041940 secs]

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

  ***CMS-initial-mark*** – marks all living objects. A STW phase

  ***CMS-concurrent-mark-start*** – the application starts to run at the same time the GC is marking additional objects. Not very exact operation in marking all objects

  ***CMS-concurrent-preclean-start*** – a cleaning operation before the remark phase; similar as the remark but processed in parallel, with multiple threads. The goal is to check again all objects, which were changed in previous phase and mark them – minimizing the time for remark operation

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**

  ***CMS-remark*** – re-scanning the objects which where changed by application threads. It is the longest phase. A STW phase

  ***CMS-concurrent-sweep-start*** – sweeps over the heap for all dead objects. Performs in parallel with all the other threads running. Here might occur compaction problems, objects are not compacted properly since the CMS is a non-compacting collector

  ***CMS-concurrent-reset-start*** – prepare for the next cycle of collection

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - Concurrent-mode-Failure, collection of the Old generation did not finish before it was already full !
    - Long living objects take too much space in Old, there is not enough free space in Old
    - Signal the GC to start enough earlier a concurrent collection using:- `X:CMSInitiatingOccupancyFraction=XX`
  - Fragmentation
    - Space in Old generation can be scattered resulting in poor allocation rate Young/Old
    - Young GC will backout if there is not enough space in Old

# HotSpot architecture, cont.

- **Concurrent Mark-Sweep Collector**
  - Fragmentation
    - Make sure you run Java 5
    - Try to run with a bigger Heap size, -Xms*XXX* and -Xmx*XXX*
    - Smaller Young generation
    - During night time, try to schedule a System.gc() - this will compact all heap

# HotSpot architecture, cont.

- **Ergonomics in Java 5**
  - the JVM automatically setting the Heap, GC, and compiler VM: client or server
  - dynamically tune the Heap, for the parallel collector, no need to specify the Heap size
  - Maximum pause time goal
    - `-XX:MaxGCPauseMillis=N`, the Parallel GC will adjust the heap size in order to keep this requirement
  - Throughput goal
    - `-XX:GCTimeRatio=N`, where N is a ratio which defines the total time spent in GC

# HotSpot architecture, cont.

- **Ergonomics in Java 5**
  - Throughput goal
    - `-XX:GCTimeRatio=N`
  - Application time: time not spent in GC
  - GC time: time spent in GC activity
  - the ratio GC time to App time = 1 / ( 1 + N )
  - ratio, N=29 will set the goal to 3% as of the total time to be allowed in GC ! `-XX:GCTimeRatio=29`
  - if the target goal can't be met then all generations will be increased, in an effort to keep the goal

# HotSpot architecture, cont.

- **Ergonomics in Java 5**
- **Server class machine**
  - A computer with 2+ CPUs and 2GB+ of memory
  - Exception 32bit Windows platforms
  - Hotspot Server VM selected
  - Initial Heap size: 1/64$^{th}$ of the physical memory, up to 1GB
  - Maximum Heap Size: 1/4$^{th}$ of the physical memory, up to 1GB

# HotSpot architecture, cont.

- **Client class machine**
  - A computer which is not a server class machine
  - Hotspot Client VM selected
  - GC: the serial collector
  - Heap size: 4MB initial up to 64MB maximum

# Heap Sizing

- I've never heard of this. Java does automatically this !
- Why do I need to tune the JVM ? The vendor takes care of that.
- Who will do this operation ? I will take care of this later on !
- It is not important, the contracting company does these sort of things !
- Should I stop profiling my application ?
- Hmm..., do I **really** need to tune the JVM ?

# Heap Sizing, cont.

**STAY IN CONTROL !**

# Heap Sizing, cont.

- WHY ?
  - Default settings, usually, not suitable for real life applications
  - To understand – your application might work, but you have to clear see how does it work
  - To avoid downtime and wasting time understanding what's going on
  - To observe closely how your application works in different conditions
- Define a tuning procedure
- Goals: throughput and footprint

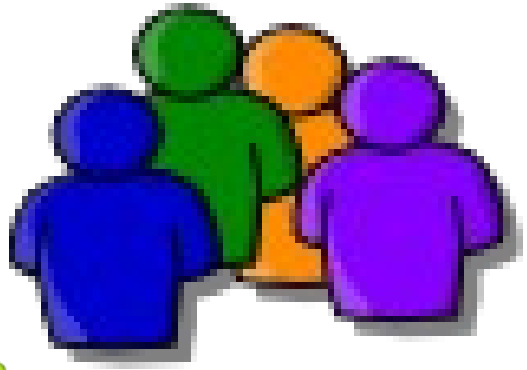
# Heap Sizing, cont.

- WHO ?
  - System Administrators or AppServer Administrators
  - Performance Testing Team
  - Java Development Team
- WHEN ?
  - Before releasing a new major version in Production
  - New features are needed to be developed
  - Business has communicated that the expected usage will triple during next 6 months

External Contractors

Java developers

Testing Team

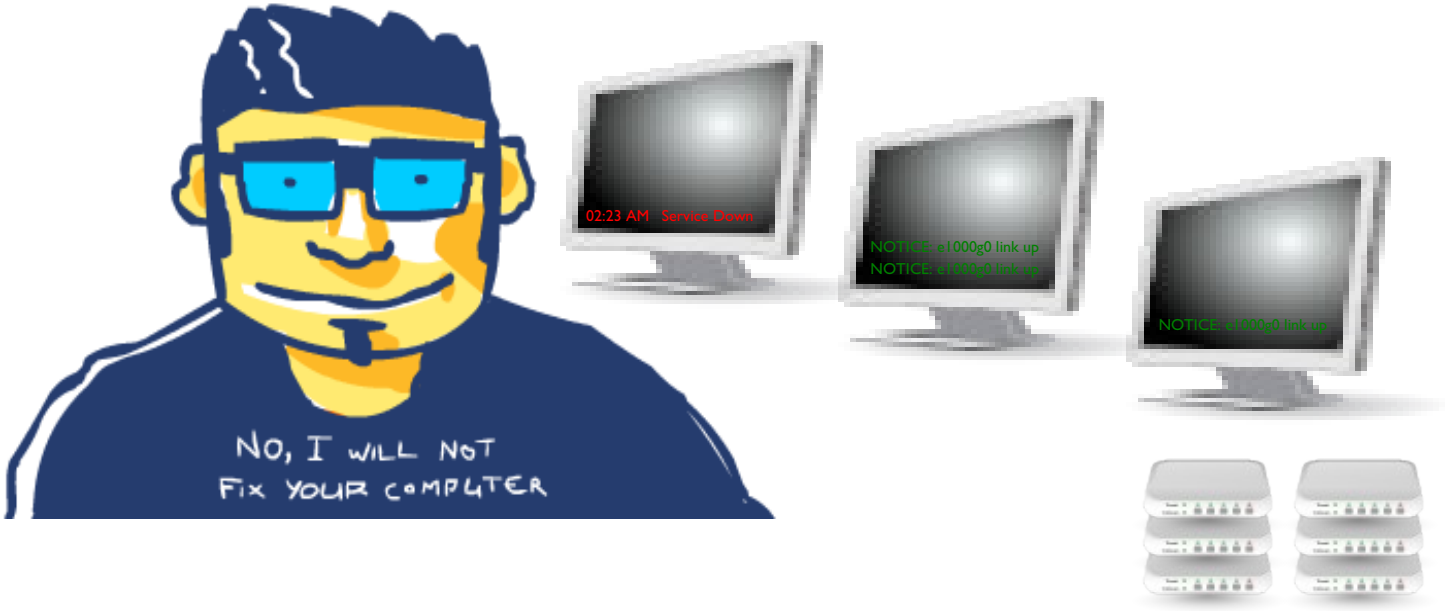


**JVM Tuning**

# System Administrator

**<Error> java.lang.OutOfMemoryError: PermGen space**

**02:23 AM Service Down**



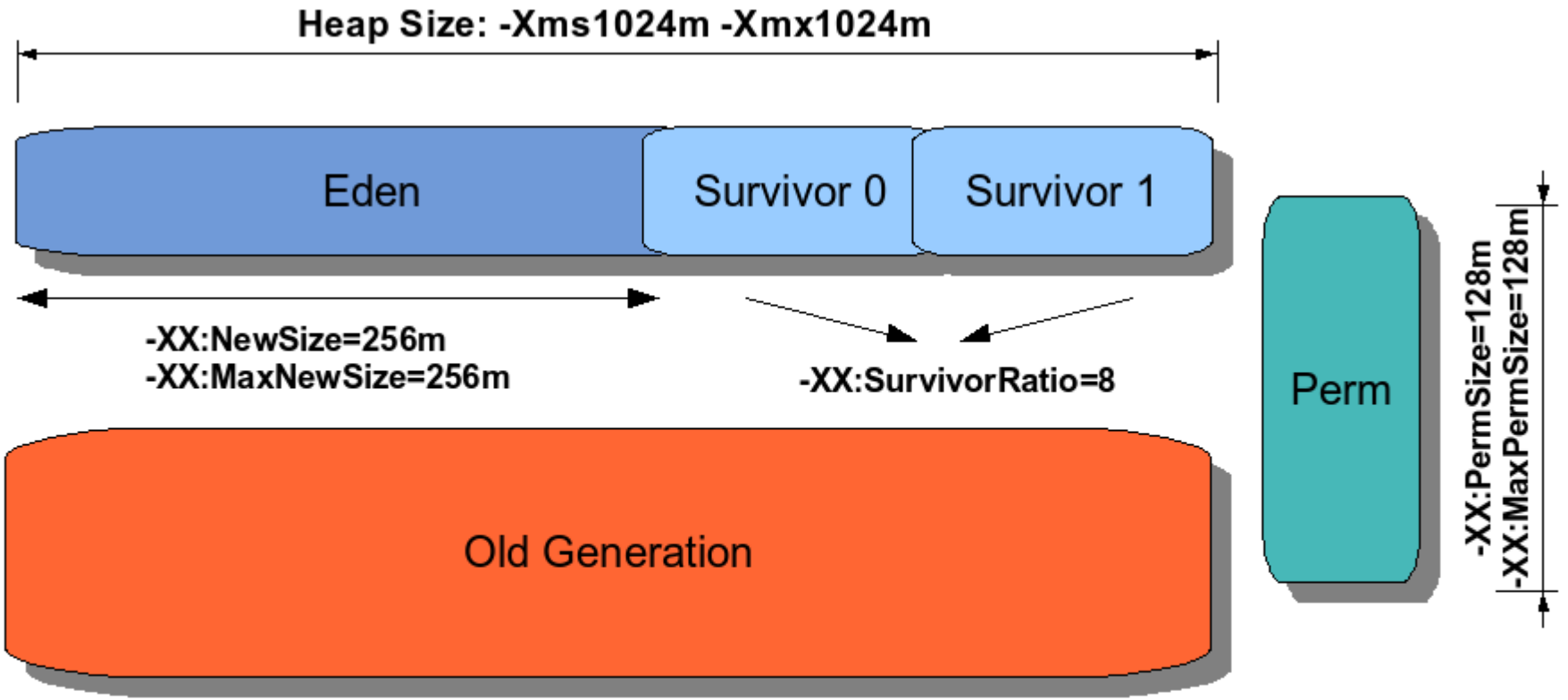
# Heap Sizing, cont.

**SO, IT REALLY MATTERS !**

# Heap Sizing, cont.

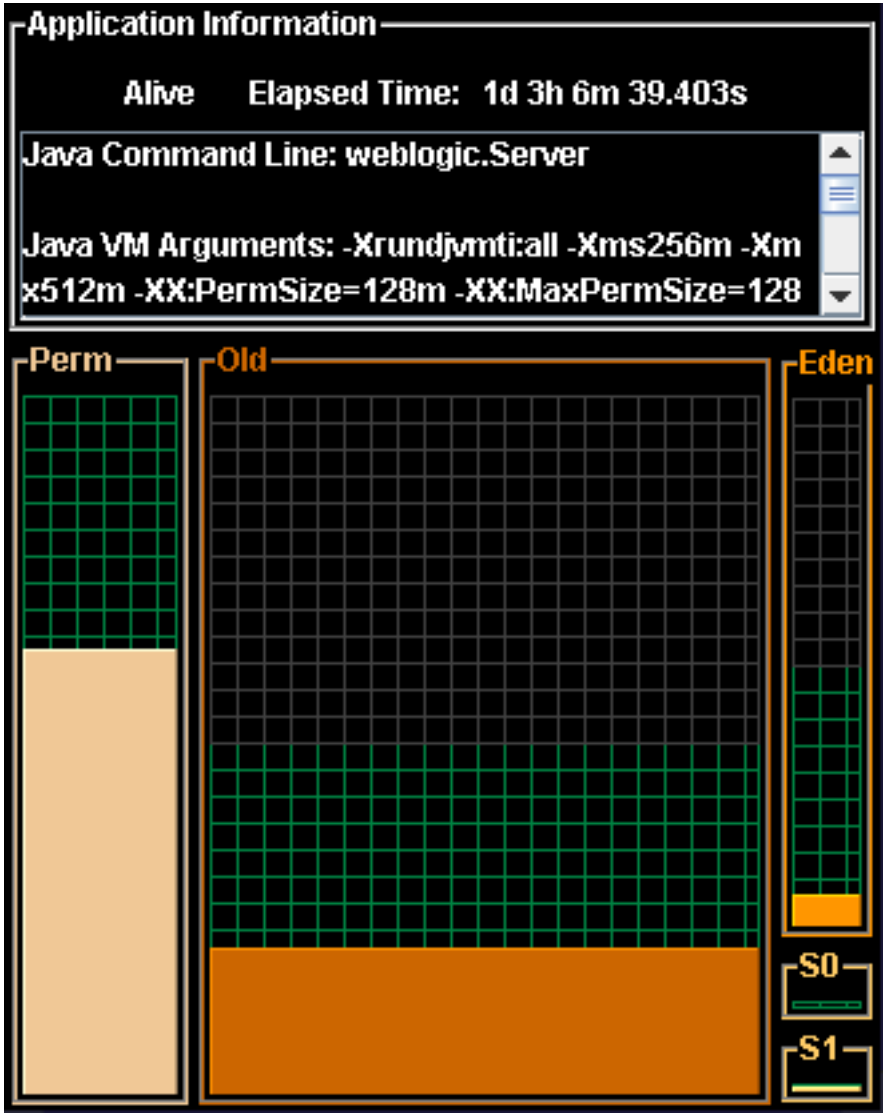
- Tools needed
  - Load injection: JMeter/LoadRunner
  - Monitoring JVM:
    - visualGC(jvmstat)
    - jmap
    - jstack, jstat
  - Monitoring OS:
    - netstat, vmstat, sysperfstat, nicstat
    - prstat
    - pstack/pmap
    - DTrace, DTraceToolkit

# Heap Sizing, cont.




The Java Heap

# Heap Sizing, cont.




The Java Heap

# Heap Sizing, cont.

- Prepare JMeter or LoadRunner performance scripts and run them against your application
- Start by allocating enough memory to the initial Heap, `-Xms`*XXX* `-Xmx`*XXX*
- Observe how many minor and major collections (Full GC) are performed
- Adjust the memory settings and change to a different GC algorithm if needed
- Repeat until you will minimize the number of major collections and the response time is ok

# Heap Sizing, cont.

- NewRatio
  - Ratio between Young/Old
  - Very important how your objects will be transitioned
  - NewRatio=2, Old 2/3 of Heap and Young 1/3
  - Flags: -Xmn`xxx` similar as -XX:NewSize=`xxx` and -XX:MaxNewSize=`xxx`
  - What value !?
  - It very much depends: if your application has a lot of long lived objects, Old > Young to hold all data

# Heap Sizing, cont.

- Young generation: define and experiment with different sizes for the Young generation using: `-XX:NewSize=`*`value`* and `-XX:MaxNewSize=`*`value`*
- Permanent generation: start with a big enough to hold all classes. Stores JVM's metadata. If the applications loads a large number of classes then this generation might require resizing using `-XX:PermSize=`*`value`* and `-XX:MaxPermSize=`*`value`*

# Heap Sizing, cont.

- For large applications set the Heap size for both options the same. This avoids extra allocations. Apply the same principle for the Young generation
- Finally: run several tests with different values for Heap and with different GC algorithms. Choose the ones they fit best your requirement: throughput, footprint, etc.
- Run several times the steps defined in the tuning procedure and review the results with your team

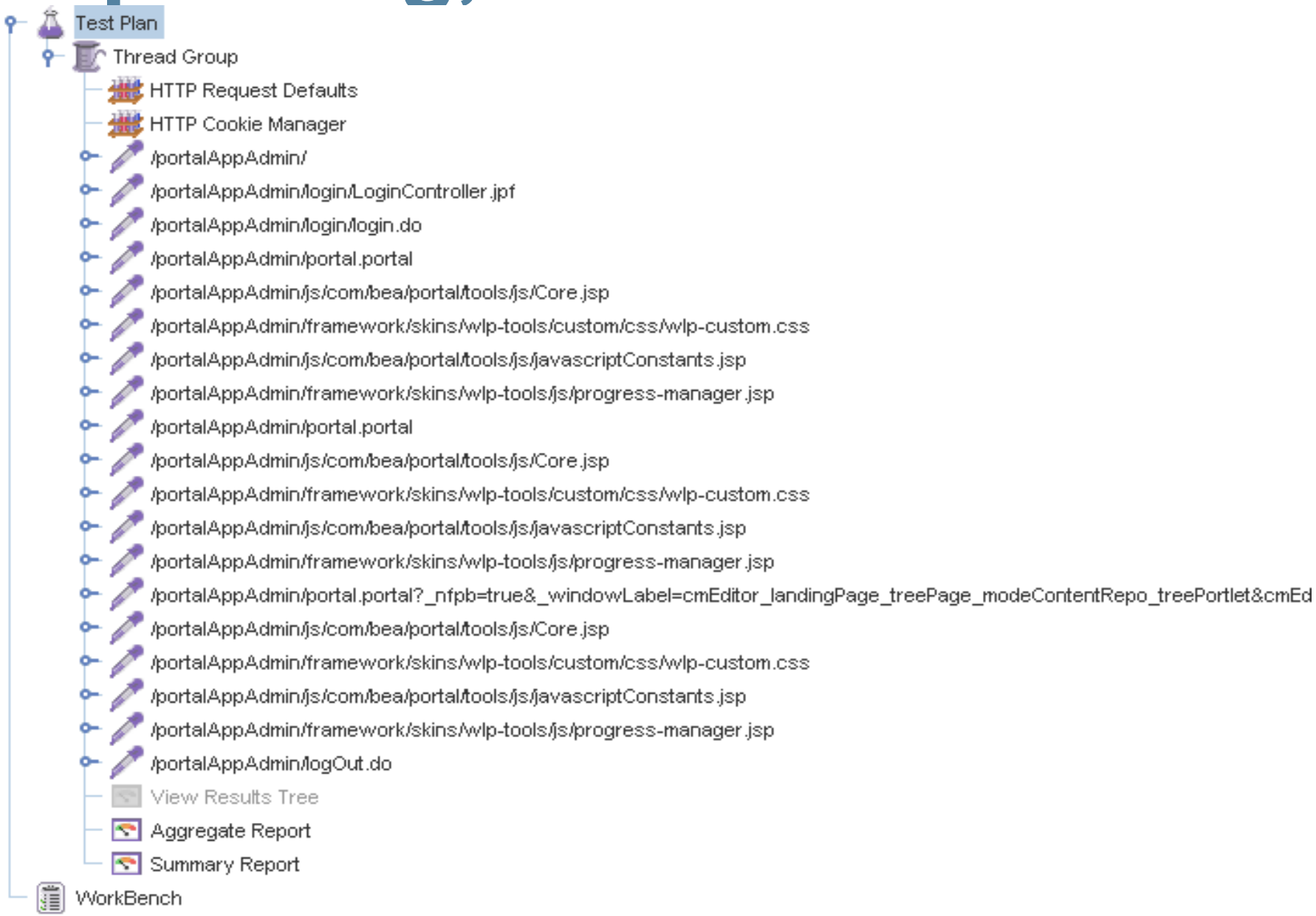
# Heap Sizing, cont.

- Example
  - BEA Weblogic Portal 10 sample application, portalAppAdmin
  - Jdk-1.5.0_14
  - Heap: 512 – 1024MB
  - Permanent generation: 256MB
  - Default GC
- Observe and Identify bottlenecks
- Tune

# Heap Sizing, cont.

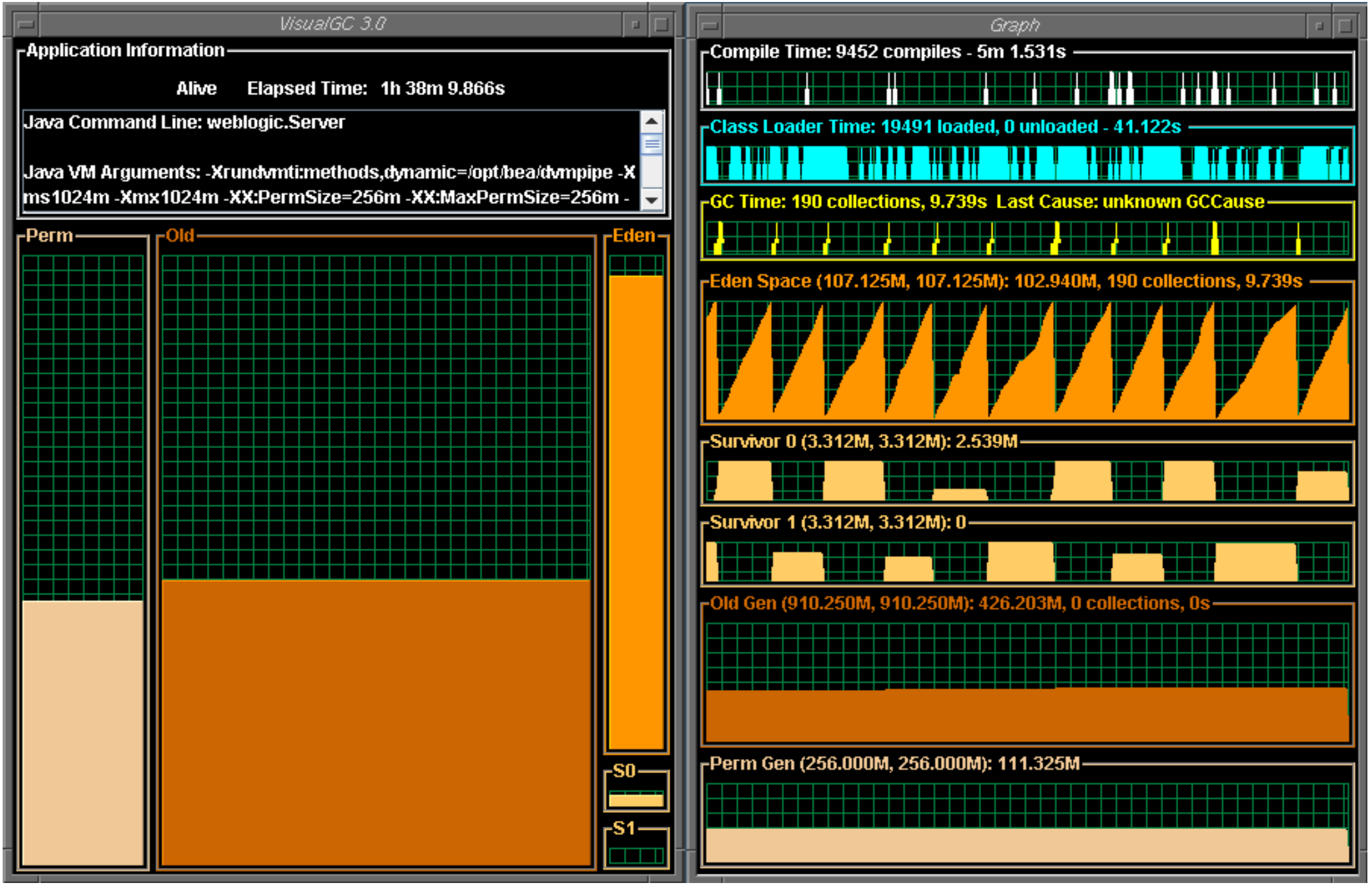
- JMeter script
  - define a simple script which will simulate concurrent users for portalAppAdmin application
  - for complex applications try to define a much realistic script
- Start with a minimal payload, 1-5 VUs
  - if you are confident about your application use a bigger payload from start, 10-50VUs
  - think time, ramp-up time
- Examine the GC logs and visualGC screen

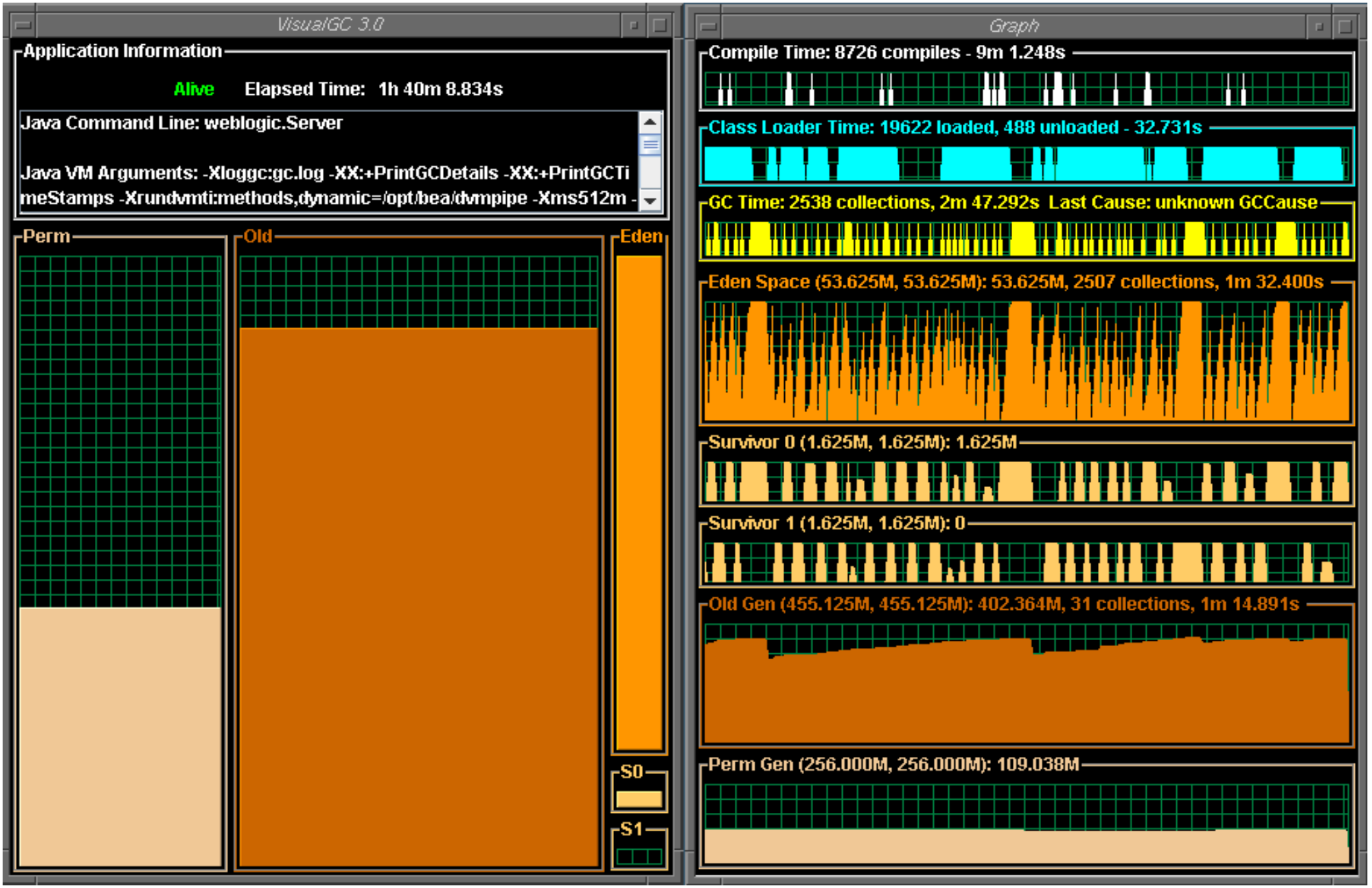
# Heap Sizing, cont.



# Heap Sizing, cont.

- Examine the young generation and the number of minor collections
- Adjust the ratio of Young/Old such way to have a good throughput and have enough big old generation
- Check the permanent generation
- Monitor closely the number of Major collections


VisualGC 3.0
Application Information
Alive Elapsed Time: 1h 38m 9.866s
Java Command Line: weblogic.Server
Java VM Arguments: -Xrundvmti:methods,dynamic=/opt/bea/dvmpipe -X
ms1024m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=256m -
Perm
Old
Eden
S0
S1
Graph
Compile Time: 9452 compiles - 5m 1.531s
Class Loader Time: 19491 loaded, 0 unloaded - 41.122s
GC Time: 190 collections, 9.739s Last Cause: unknown GCCause
Eden Space (107.125M, 107.125M): 102.940M, 190 collections, 9.739s
Survivor 0 (3.312M, 3.312M): 2.539M
Survivor 1 (3.312M, 3.312M): 0
Old Gen (910.250M, 910.250M): 426.203M, 0 collections, 0s
Perm Gen (256.000M, 256.000M): 111.325M


VisualGC 3.0
Application Information
Alive
Elapsed Time: 1h 40m 8.834s
Java Command Line: weblogic.Server
Java VM Arguments: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTi
meStamps -Xrundvmti:methods,dynamic=/opt/bea/dvmpipe -Xms512m -
Perm
Old
Eden
S0
S1
Graph
Compile Time: 8726 compiles - 9m 1.248s
Class Loader Time: 19622 loaded, 488 unloaded - 32.731s
GC Time: 2538 collections, 2m 47.292s Last Cause: unknown GCCause
Eden Space (53.625M, 53.625M): 53.625M, 2507 collections, 1m 32.400s
Survivor 0 (1.625M, 1.625M): 1.625M
Survivor 1 (1.625M, 1.625M): 0
Old Gen (455.125M, 455.125M): 402.364M, 31 collections, 1m 14.891s
Perm Gen (256.000M, 256.000M): 109.038M

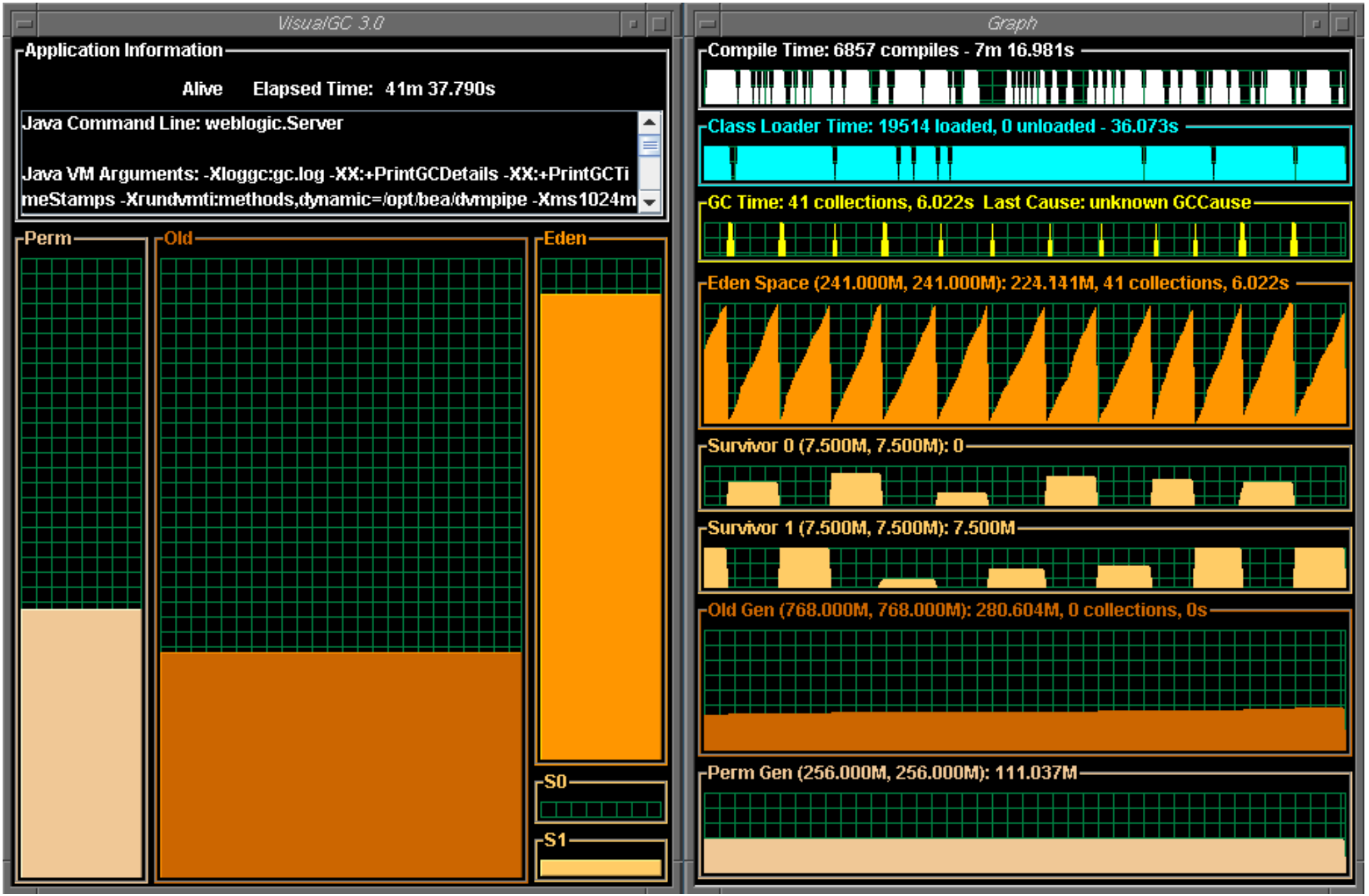

VisualGC 3.0
Application Information
Dead
Elapsed Time: 2h 6m 12.325s
Java Command Line: weblogic.Server
Java VM Arguments: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTi
meStamps -Xrundvmti:methods,dynamic=/opt/bea/dvmpipe -Xms512m -
Perm
Old
Eden
S0
S1
Graph
Compile Time: 8971 compiles - 11m 24.146s
Class Loader Time: 19789 loaded, 663 unloaded - 33.928s
GC Time: 3235 collections, 11m 41.011s Last Cause: unknown GCCause
Eden Space (53.625M, 53.625M): 53.625M, 3058 collections, 1m 50.845s
Survivor 0 (1.625M, 1.625M): 1.625M
Survivor 1 (1.625M, 1.625M): 0
Old Gen (455.125M, 455.125M): 455.125M, 177 collections, 9m 50.166s
Perm Gen (256.000M, 256.000M): 109.065M

# Heap Sizing, cont.

- Young generation too small
- Survivor spaces not big enough, what ratio !?
- Permanent generation ok
- Old generation too small
- A better GC ?
- After 1st tuning:

  -Xms1024m -Xmx1024m

  -XX:NewSize=256m -XX:MaxNewSize=256m


VisualGC 3.0
Application Information
Alive Elapsed Time: 41m 37.790s
Java Command Line: weblogic.Server
Java VM Arguments: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTi
meStamps -Xrundvmti:methods,dynamic=/opt/bea/dvmpipe -Xms1024m
Perm
Old
Eden
S0
S1
Graph
Compile Time: 6857 compiles - 7m 16.981s
Class Loader Time: 19514 loaded, 0 unloaded - 36.073s
GC Time: 41 collections, 6.022s Last Cause: unknown GCCause
Eden Space (241.000M, 241.000M): 224.141M, 41 collections, 6.022s
Survivor 0 (7.500M, 7.500M): 0
Survivor 1 (7.500M, 7.500M): 7.500M
Old Gen (768.000M, 768.000M): 280.604M, 0 collections, 0s
Perm Gen (256.000M, 256.000M): 111.037M

# Heap Sizing, cont.

Conclusions:

- Retest the previous JVM options
- Are these options ok ? What about more fine tuning ? A better young generation, survivor spaces, bigger heap !?
- GC: CMS or Parallel Compactor ?
- Repeat all over the test based on different payloads until you will match your desired results

# Heap Sizing, cont.

- Parallel GC tuning
  - Let the JVM chose the Heap sizes
  - Define a throughput goal
  - The heap will dynamically grow or shrink based on the application usage
  - Sometimes the heap will set to its max value, the throughput goal cannot be met
  - Throughput goal is met but still are application pauses: set a maximum pause time

# Heap Sizing, cont.

- For large applications, a tuning procedure might be needed
- Clear steps and test cases
- A cycle consists of one or more test cases
- Each cycle is reviewed and retested
- Usually 3-4 cycles are enough for sizing a generic JVM
- Adapt the procedure for your application and purpose

# Heap Sizing, cont.

- **Tuning procedure**
  - Cycle 1 - Initial observations
    - default settings for JVM: Heap size, Permanent generation, default garbage collector. Monitor the initial behavior of the application
  - Cycle 2 – Tuning
    - different settings for garbage collector and memory generations. Try to discover the maximum number of VUs one JVM can sustain
  - Cycle 3 - Fine tuning
    - apply a series of extra test cases to confirm C2
  - Cycle 4 - Plan to crash

# Heap Sizing, cont.

- **Tuning Cycle 1**

**Monitoring points**

System utilisation: **CPU**, Mem, Disk, Net, TCP: **ESTABLISHED, CLOSE_WAIT, TIME_WAIT**

**JVM utilisation: GC, Old, Young, Permanent generations, throughput**

JVM options: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

  - C1.1: Initial observations
  - C1.2: Minimal payload
  - C1.3: Long run test
  - C1.4: Stress test
  - C1.5: Final Exit Criteria

# Heap Sizing, cont.

- **Tuning Cycle 1**
  - C1.1: Initial observations
    - Start with default settings: 1-2GB Heap size and 128 MB for permanent generation.
    - Observe using visualGC the behavior of your application for an average number of VUs: ~25-100 during 1, 2hrs.
    - Examine closely the GC log and monitor the number of minor and major collections. Record the time spent in GC and the response time of the application. If you start to notice long time spent in GC try to decrease the number of VUs to 10-50 and repeat the test.
    - If you cant sustain for 1-2 hrs a minimal number of 5-10 VUs, recall the application to functional testing and escalate with your Java project development team.

# Heap Sizing, cont.

- **Tuning Cycle 1**
  - C1.2: Minimal payload
    - Ensure the application works ok with this minimal payload found in C1.1
    - Keep the application long enough, say 2-4hrs and examine closely the JVM data: GC, the number of minor/major collections and the time spent for these events.
    - If your application cant sustain for more than 2hrs this minimal payload return to functional testing and core development, opening bugs to the development team finding out why the application handles so bad such minimal payload.

# Heap Sizing, cont.

- **Tuning Cycle 1**
  - C1.3: Long run test
    - If everything goes fine in C1.2 plan a longer test for 10-12hrs with this payload.
    - Analyse the results with the Java development and support teams, together and write down the results found for this payload.
    - If you found problems in this phase return to 1.2 and decrease the payload. Easily adjust the payload trying to find out the for how long your application works fine.

# Heap Sizing, cont.

- **Tuning Cycle 1**
  - C1.4: Stress test
    - Plan a stress test at this point where you can analyse how long your application will survive for a different number of virtual users.
    - Keep this load for certain period of time: 2-6hrs. For instance: start with the minimal payload, slowly ramping up every 30minutes with 50VUs more. Keep a max load level, say 500 VUs and observe the System and JVM data. Decrease if necessarily or increase if the application seems to handle ok that load.
    - Try to find out the max number of VUs that your application can handle. Repeat the test to ensure the step is correct.

# Heap Sizing, cont.

- **Tuning Cycle 1**
  - C1.5: Final Exit Criteria
    - Based on C1.2-4 you should approximate the max number of VUs one JVM can sustain and if the initial VM options used, for Heap and Permanent generation are ok.
    - It is highly recommended to have a common meeting review after Cycle 1 and discuss the results with your Java Development Team and Support Unit.
    - Finally review the results and apply the changes to your JVM to be ready for your next cycle.

# Heap Sizing, cont.

- **Tuning Cycle 2**

**Monitoring points**

System utilisation: **CPU**, Mem, Disk, Net, TCP: **ESTABLISHED, CLOSE_WAIT, TIME_WAIT**

**JVM utilisation: GC, Old, Young, Permanent generations, throughput**

JVM options: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

  - C2.1: Sizing generations, chose a collector
  - C2.2: New settings, 10hrs
  - C2.3: Uplimit
  - C2.4: Endurance
  - C2.5: Final Exit Criteria

# Heap Sizing, cont.

- **Tuning Cycle 2**
  - C2.1: Sizing generations, chose a collector
    - Based on the previous cycle results, from C1, try to adjust the JVM options: enlarge the Heap size and dedicate enough memory to the Young generation and Permanent generation. Chose a Garbage collector, experimenting with the Parallel Compactor and CMS.
    - Try first the Parallel collector, using as well the new Parallel Compactor. Write down the settings and results.
    - Try at the end the CMS collector, write down the settings and results. Chose one which is best for your application.
    - Review once more the setting and pickup a final collector what you will use.

# Heap Sizing, cont.

- **Tuning Cycle 2**
  - C2.2: New settings, 10hrs
    - Use the max payload found in C1 and run a test for 10hrs using the new collector and memory settings from C2.1.
    - If you haven't changed anything in C2.1, simple check that one JVM can easily sustain this payload. Observe the JVM dynamics using visualGC and jstat.
    - Record all times and ensure you don't have a degradation of your service using the new VM settings.
    - Repeat the test as many times you think in order to ensure the new JVM settings are ok. Make final adjustments to your Heap, Permanent generation and garbage collector and freeze the options. Repeat the test with the final settings.

# Heap Sizing, cont.

- **Tuning Cycle 2**
  - C2.3: Uplimit
    - Increase the payload to 250-500-1000 VUs and run it for 2-4hrs with the new settings.
    - If problems decrease the payload to a decent and safe value where you dont find anymore errors. Write down the max value for VUs.

# Heap Sizing, cont.

- **Tuning Cycle 2**
  - C2.4: Endurance
    - Ensure the JVM can sustain for 10-12hrs the C2.3 payload. If you have troubles return to C2.1
    - If ok, plan a longer test for 48-72hrs with this load in Quality and Assurance environment (The QA should have similar network and configuration as the PROD environment)

# Heap Sizing, cont.

- **Tuning Cycle 2**
  - C2.5: Final Exit Criteria
    - At this moment you should have a set of the new settings for your JVM and a min and max value numbers of VUs.
    - Write down the average number of VUs for one JVM as well the max number of VUs you have recorded.

      VUavg/JVM

      VUmax/JVM

# Heap Sizing, cont.

- **Tuning Cycle 3**

**Monitoring points**

System utilisation: **CPU**, Mem, Disk, Net, TCP: **ESTABLISHED, CLOSE_WAIT, TIME_WAIT**

**JVM utilisation: GC, Old, Young, Permanent generations, throughput**

JVM options: -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps

  - C3.1: Endurance test
  - C3.2: Final test
  - C3.5: Final Exit Criteria

# Heap Sizing, cont.

- **Tuning Cycle 3**
  - C3.1: Endurance test
    - Plan with the LoadRunner/Jmeter team several test cases for QA where the application will use the new settings found in C2.
    - Use as a payload the C2 findings.

# Heap Sizing, cont.

- **Tuning Cycle 3**
  - C3.2: Final test
    - Run a final test for 72hrs with all managed servers up and running, no extra debug level on. Record the response time of the application and watch for major collections and how long they take. Ngs.
  - C3.5: Final Exit Criteria
    - If no troubles found in C3.2 or C3.1 push the new settings for PROD env. If you have experinced troubles in C3.1,2 it is good to review the results and return to Cycle 2

# Heap Sizing, cont.

- **Tuning Cycle 4**

**Monitoring points**

*Proactive step, in order to prevent possible failures inside JVM or from application itself*

JVM: jstack, jmap, jstat

OS: truss, dtrace, pstack, pmap

  - C4.1: JVM core dump
  - C4.2: Application not responsive, very slow
  - C4.3: Dynamic Tracing

# Heap Sizing, cont.

- **Tuning Cycle 4**
  - C4.1: JVM core dump
    - Plan a simple recovery plan in case your application core dumps. Keep up to date the JDK with your vendor, check periodically the Release Notes for any corrections on JDK.
    - Examine the core dump using dbx or mdb in Solaris.
  - C4.2: Application not responsive, very slow
    - One of the most common symptoms what you could experience: your application starts to answer very slow, or looks like is doing nothing. Check the CPU consumption and get two, three thread dumps using SIGQUIT signal. Use *prstat -mL* to get a lwp usage distribution. As well experiment this in QA env by developing a series of D scripts which can be used. Consider using here: jstack, pstack, dtrace

# Heap Sizing, cont.

- **Tuning Cycle 4**
  - C4.3: Dynamic Tracing
    - You should not wait until you have a real problem. Therefore it is good you develop in QA env a series of D scripts, based on DTrace to find out how your application really works. Try to understand the numbers you get and run these periodically communicating the findings to your Java development team and Support Unit. Check the Troubleshooting section for more details

# Heap Sizing, cont.

- Final: you should have a tuning JVM document where you have all results gathered
- Review this document with your Java Development and Support unit and chose the best values for your JVM
- Implement these in QA before PRODUCTION
- You are almost done: periodically review and re-tune if needed. Application changes can have an effect the way the JVM works

# Heap Sizing, cont.

- **Tuning Cycle 1, example**

| VUs | Test | JVM Settings |
|---|---|---|
| | | |
| 50 | 12:00 – 13:00: JVM 1,2 running | -verbose:gc -Xloggc:gc.log<br>-XX:+PrintGCDetails<br>-XX:+PrintGCTimeStamps<br>-Xms2048m<br>-Xmx2048m<br>-XX:MaxPermSize=512m |
| | | |
| 100 | 14:00 – 15:00: JVM 1,2 running | -verbose:gc -Xloggc:gc.log<br>-XX:+PrintGCDetails<br>-XX:+PrintGCTimeStamps<br>-Xms2048m<br>-Xmx2048m<br>-XX:MaxPermSize=512m |
| 200 | 20:00 – 20:30: JVM 1,2 running | -verbose:gc -Xloggc:gc.log<br>-XX:+PrintGCDetails<br>-XX:+PrintGCTimeStamps<br>-Xms2048m<br>-Xmx2048m<br>-XX:MaxPermSize=512m |
| 300 | 20:30 – 21:00: JVM 1,2 running | -verbose:gc -Xloggc:gc.log<br>-XX:+PrintGCDetails<br>-XX:+PrintGCTimeStamps<br>-Xms2048m<br>-Xmx2048m<br>-XX:MaxPermSize=512m |
| 500 | 21:00 – 23:00: JVM 1,2 running | -verbose:gc -Xloggc:gc.log<br>-XX:+PrintGCDetails<br>-XX:+PrintGCTimeStamps<br>-Xms2048m<br>-Xmx2048m<br>-XX:MaxPermSize=512m |

# Heap Sizing, cont.

- **Tuning Cycle 1, example**
  - Test 1: 50 VUs/10sec think time; 12:00 – 14:00

| CPU | Mem | Disk | Net | TCP Sessions | GC | Old | Young | Permanent |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| 1-10% | 20.00% | <3% | 15% | ~70 | Minimal activity | OK | OK | Requires tuning |
| | | | | | | | | |

  - Test 2: 100 VUs/10sec think time, 14:00 – 15:00

| CPU | Mem | Disk | Net | TCP Sessions | GC | Old | Young | Permanent |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| 15.00% | 20.00% | <3% | 15% | 110 | Minimal, minor GC | OK | OK | Requires tuning |

  - Test 3: 200-300 Vus/5sec think time 20:00 – 20:30 – 21:00

| CPU | Mem | Disk | Net | TCP Sessions | GC | Old | Young | Permanent |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| ~20% | 20.00% | <3% | 20% | 200 | minor GC | OK | OK | Requires tuning |
| | | | | | | | | |

# Heap Sizing, cont.

- **Tuning Cycle 1, example**
  - Test 4: 500 VUs/10sec think time; 21:00 - 23:00

| CPU | Mem | Disk | Net | TCP Sessions | GC | Old | Young | Permanent |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| 20.00% | 20.00% | <3% | 20% | 350 | Minor and major collection. Full GC 5sec | Needs tuning | OK | Requires tuning |

  - Review the results after each test case
  - Write down the results in such report and after all test cases in cycle 1 review the results with your support unit and Java development team

# Heap Sizing, cont.

- **Tuning Cycle 1, example**
  - Final Exit criteria to Cycle 2:

| Change | Description |
|---|---|
| -XX:PermSize=256m<br>-XX:MaxPermSize=256m | The permanent generation needs to be set to 256MB from the beginning. Currently the generation starts as 64MB and grows up to 512MB. 512MB is way too high for the current application. |
| CPU utilisation | Identified a CPU usage issue: sometimes one managed server is using more CPU than the other, even for longer periods of time. |
| -Xms2048m<br>-Xmx2048m | The heap might not be enough set for running more than 24hrs. Requires a bit more analyse in Cycle 2 to observe how often we will have FullGCs from one JVM. |
| Payload: 250VUs | Both JVM were able to sustain 500VUs for certain amount of time. For next Cycle, we will chose 250 VUs as the starting payload for one JVM and we will try to identify the safe payload for one JVM |

# Agenda

- Introduction to Java Virtual Machine
- HotSpot VM
  - Architecture
  - Memory management and Garbage Collection
  - Heap Sizing
- **Java 5 and 6**
  - What's new ?
- Troubleshooting

# Java 5 and 6

- Java releases over time
- JDK 1.1
  - JDK 1.1.4 Sparkler Sept 12, 1997
  - JDK 1.1.5 Pumpkin Dec 3, 1997
  - JDK 1.1.6 Abigail April 24, 1998
  - JDK 1.1.7 Brutus Sept 28, 1998
  - JDK 1.1.8 Chelsea April 8, 1999

# Java 5 and 6, cont.

- JDK 1.2 and 1.3
  - **J2SE 1.2 Playground Dec 4, 1998**
  - J2SE 1.2.1 (none) March 30, 1999
  - J2SE 1.2.2 Cricket July 8, 1999
  - **J2SE 1.3 Kestrel May 8, 2000**
  - J2SE 1.3.1 Ladybird May 17, 2001

# Java 5 and 6, cont.

- JDK 1.4, 5 and 6
  - **J2SE 1.4.0 Merlin Feb 13, 2002**
  - J2SE 1.4.1 Hopper Sept 16, 2002
  - J2SE 1.4.2 Mantis June 26, 2003
  - **J2SE 5.0 (1.5.0) Tiger Sept 29, 2004**
  - **J2SE 6.0 (1.6.0) Mustang Dec, 2006**

# Java 5 New Features

- JVM
  - Class Data Sharing
  - Garbage Collector Ergonomics
  - Thread Priority Changes
  - Fatal Error Handling: OOM

```
<Oct 28, 2007 8:35:21 PM EET> <Error> <netuix> <BEA-423167> <An exception
  or error occurred in the backing file [com.bea.jsptools.comm
on.PatDesktopBacking] while executing its preRender method. It was
  java.lang.OutOfMemoryError: PermGen space
java.lang.OutOfMemoryError: PermGen space
```

  - High-Precision Timing Support

# Java 5 New Features, cont.

- Java Language, User Interface, RMI, JDBC
- Base Libraries
  - Monitoring and Management
  - Monitoring and management API for the Java virtual machine
  - JMX instrumentation of the Java virtual machine
  - JavaTM Management Extensions
  - *jconsole, jstack, jmap*
  - *jmap* backported to JDK 1.4.2_09 branch helps you to diagnose problems where you don't have latest java

# Java 5 New Features, cont.

- Tools and Tool Architecture
  - Java Virtual Machine Tool Interface (JVMTI)
  - Java Platform Debugger Architecture (JPDA)
  - A new troubleshooting guide
- Hardware 64-Bit AMD Opteron support

# Java 6 New Features

- JVM
  - DTrace probes into the JVM, Hotspot provider
  - GC: Parallel Compaction and CMS enhancements
  - CMS: Parallel Marking, the CMS collector now uses multiple threads to perform the concurrent marking task. Minimizes the duration of the concurrent marking cycle, allowing the collector to support applications with larger numbers of threads and higher object allocation rates, particularly on large multiprocessor machines

# Java 6 New Features

- Scripting
  - integrated JSR 223: Scripting for the Java Platform API
  - Java Applications can "host" script engines, example javascript for Java
- Security
  - XML Digital Signature API and implementation
  - JSR 268, Smart Card I/O API
  - Elliptic Curve Cryptography (ECC)
  - java.io.Console class

# Java 6 New Features

- Observability
  - jconsole officially supported
  - `java.util.concurrent` locks has been added,
  - easy access to list all locks owned by a thread and to report which stack frame locked a monitor
  - `getSystemLoadAverage()`, has been added to OperatingSystemMXBean to return the system load average
  - Java Virtual Machine Tool Interface (JVM TI), enhanced heap walking

# Java 6 New Features

- Observability
  - HPROF
    - built-in heap dumper – easy to dump the Heap when an OOM occurs -XX:+HeapDumpOnOutOfMemoryError
    - postmortem analysis from a core file, generate a Heap dump
  - jhat, Java Heap Analysis Tool enhanced
    - Scripting interface to HAT
    - Object Query Language, OQL, is a SQL like language used to query the Java Heap !

# Java 6 New Features

- Performance
  - Runtime enhancements:
    - biased locking, lock coarsening, adaptive spinning
    - large page heap on x86/64
    - System.arraycopy()
  - Ergonomics
  - Garbage Collection:
    - Parallel Compaction Collector
    - Concurrent Mark Sweep Collector
  - Faster startup, boot class loader

# Agenda

- Introduction to Java Virtual Machine
- HotSpot VM
  - Architecture
  - Memory management and Garbage Collection
  - Heap Sizing
- Java 5 and 6
  - What's new ?
- **Troubleshooting**

# Troubleshooting

- Help, my Java application is slow !
- Again my application has crashed in PRODUCTION – whats going on ?
- We can't migrate our J2EE application to the new hardware platform – what can we do ?
- How can we understand and look what's going on with my application ?
- Which VM we should use: HotSpot, JRockit, J9 !? Why are so many virtual machines ?

# Troubleshooting, cont.

- For support:
  - Develop a number of Standard Operation Procedures (SOP) in case of problems
  - What do you do when the entire JVM has died, with a core dump ? How do you examine ?
  - What about the application is simple slow, what tools do you need to debug this ?
  - The JVM is wasting a lot of CPU resources, again what tools do you need to investigate
  - Or simple, the application does nothing but my users are complaining

# Troubleshooting, cont.

- For application development:
  - Do you have a plan for unit, integration testing ?
  - System Testing, Regression testing ?
  - Have you ever profiled your application ?
  - How well are you testing your application ? What code coverage criteria are you using ?
  - We have so many tools – if the profiler tool takes so much time to understand why don't you consider dynamic tracing ?

# Troubleshooting, cont.

- Almost in all real cases, Application development team does not work together with the Support team
- Both have same goal: keep production up and running
- How do you fix this ?
  - avoid internal politics
  - schedule common meetings
  - plan together certain activities: next QA system test, a debugging exercise, etc
  - review together the results of certain activities

# Troubleshooting, cont.

- Have you read the Troubleshooting Guide !?
  - Java 6:
    - http://java.sun.com/javase/6/docs/
    - http://java.sun.com/javase/6/webnotes/trouble/index.html
  - Java 5:
    - http://java.sun.com/j2se/1.5.0/docs/
    - http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#debug

# Troubleshooting, cont.

- Operating system:
  - **prstat,** process statistics
  - **pstack**, prints stack traces for a process
  - **pmap**, used to print the process address space and its segments
  - **truss**, general purpose syscall tracer
  - **DTrace**, new dynamic tracing facility in Solaris 10
  - **DTraceToolkit**, DTT, over 200 DTrace scripts ready to be used

# Troubleshooting, cont.

- DTraceToolkit, Java category:

```
/opt/DTraceToolkit-0.99/Java
# ls -lrt
total 108
-rw-r--r--   1 sparvu   eng          674 Sep 12 10:29 Readme
-rwxr-xr-x   1 sparvu   eng         1575 Sep 12 10:47 j_who.d
-rwxr-xr-x   1 sparvu   eng         1626 Sep 12 10:47 j_thread.d
-rwxr-xr-x   1 sparvu   eng         1789 Sep 12 10:47 j_syscalls.d
-rwxr-xr-x   1 sparvu   eng         2408 Sep 12 10:47 j_profile.d
-rwxr-xr-x   1 sparvu   eng         1418 Sep 12 10:47 j_package.d
-rwxr-xr-x   1 sparvu   eng         1688 Sep 12 10:47 j_objnew.d
-rwxr-xr-x   1 sparvu   eng         1657 Sep 12 10:47 j_methodcalls.d
-rwxr-xr-x   1 sparvu   eng         1504 Sep 12 10:47 j_events.d
-rwxr-xr-x   1 sparvu   eng         3045 Sep 12 10:47 j_calls.d
-rwxr-xr-x   1 sparvu   eng         2581 Sep 16 11:17 j_flow.d
-rwxr-xr-x   1 sparvu   eng         3008 Sep 17 05:20 j_flowtime.d
-rwxr-xr-x   1 sparvu   eng         4026 Oct  1 16:36 j_syscolors.d
-rwxr-xr-x   1 sparvu   eng         3845 Oct  3 11:21 j_cputime.d
-rwxr-xr-x   1 sparvu   eng         3161 Oct  3 11:21 j_cpudist.d
-rwxr-xr-x   1 sparvu   eng         3868 Oct  3 11:21 j_calltime.d
-rwxr-xr-x   1 sparvu   eng         3181 Oct  3 11:21 j_calldist.d
-rwxr-xr-x   1 sparvu   eng         2749 Oct  4 07:34 j_classflow.d
-rwxr-xr-x   1 sparvu   eng         3148 Oct  4 11:35 j_stat.d
```

# Troubleshooting, cont.

- JDK:
  - **jconsole**, provides information on performance and resource consumption of applications running. Based on Java Management Extension (JMX) technology:
    - Detect low memory situations
    - Enable or disable GC and class loading verbose tracing
    - Detect deadlocks
  - **jstack**, prints Java stack traces of each thread
  - **jmap**, memory statistics for a running JVM. Useful options here: -heap, -histo, -permstat

# Troubleshooting, cont.

- JDK:
  - **jstat**, performance and resources consumption. Used to identify bottlenecks with GC and Heap sizing
  - **visualGC**, a GUI used to monitor the activity of the Garbage Collection, GC
  - **jhat**, the heap analyser tool. Inspects a heap dump and displays object retention – objects which are not longer needed but still referenced

# Troubleshooting, cont.

- Java Post-Mortem Analysis

<work in progress>

# Troubleshooting, cont.

- Java Process Analysis

<work in progress>

# Troubleshooting, cont.

- Java Performance Analysis
  - Case 1: Java 5 and Weblogic Portal 10
  - Case 2: Java 6 and Sun Java System Webserver 7.0sp1

# Troubleshooting, cont.

- **Case study 1: Java 5 and BEA Weblogic Portal 10**
- Scope:
  - evaluate and debug the portalAppAdmin sample application, delivered with the BEA Weblogic Portal product
  - observe the application using different methods: DTrace, operating system utilities or the JDK tools
  - list all the inefficient parts of the application

# Troubleshooting, cont.

- **Notice:**
  - This applies only to BEA Weblogic Portal 10.0 and 10.1 software
  - The issues found are specific with Portal software only !

# Troubleshooting, cont.

- A sample portal application

```
$ cd /opt/bea/wlp10/wlserver_10.0/samples/domains/portal/bin

$ cp /opt/bea/wlp10/wlserver_10.0/common/bin/commEnv.sh .

$ ./startWebLogic.sh

...

JAVA Memory arguments: -Xms256m -Xmx768m -XX:CompileThreshold=8000
   -XX:PermSize=48m  -XX:MaxPermSize=128m

<Oct 27, 2007 3:51:53 PM EEST> <Notice> <WebLogicServer> <BEA-000365>
   <Server state changed to RUNNING>

<Oct 27, 2007 3:51:53 PM EEST> <Notice> <WebLogicServer> <BEA-000360>
   <Server started in RUNNING mode
```

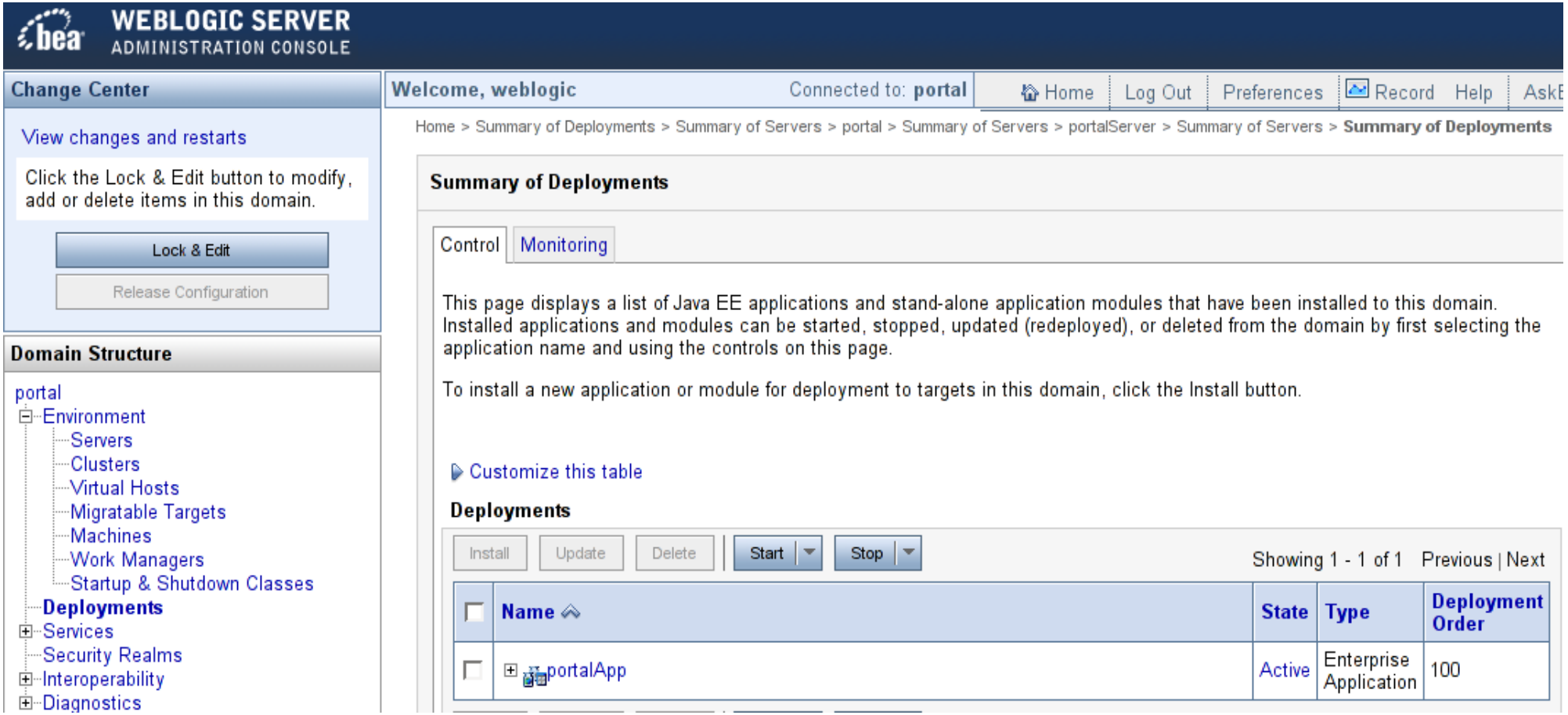
http://localhost:7041/console/

http://localhost:7041/portalAppAdmin/

# Troubleshooting, cont.

- Couple of configuration changes:
  - Switch the example domain from development to production mode
  - Switch off the debugging options
  - Experiment with different JVM options:
    - Client or server compiler: - server, - client
    - Heap size: 512, 1024MB
    - Permanent generation: 128, 256MB
  - Switch off the Autonomy module

# Troubleshooting, cont.

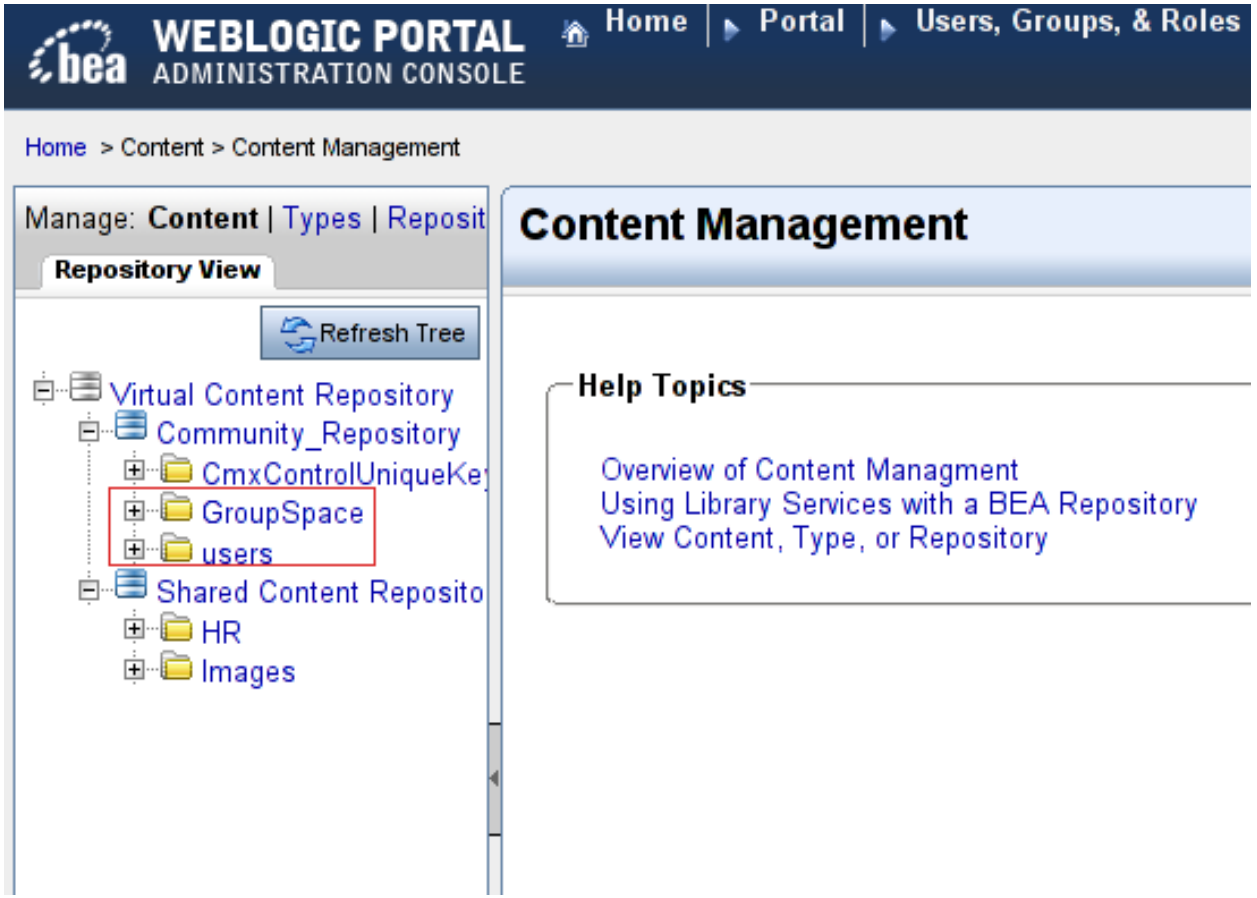


# Troubleshooting, cont.

- Scope:
  - What the portal sample application does when I click 'users ' ? Usually this should not take long time, however under some systems we can see it takes ~5-10 seconds
  - Check what Java methods are called.
  - Is there any way to count how long each of these calls are taking ?
  - How about object allocation when we access this part of the application ?

# Troubleshooting, cont.

- Select 'users' or 'GroupSpace'



# Troubleshooting, cont.

- Check the JVM target process, *pargs pid*

```
1007:   /opt/java/jdk1.5.0_13/bin/java -server -Xrundvmti:methods,dynamic=/opt/b
ea/dvmp
argv[0]: /opt/java/jdk1.5.0_13/bin/java
argv[1]: -server
argv[2]: -Xrundvmti:methods,dynamic=/opt/bea/dvmpipe
argv[3]: -Xms512m
argv[4]: -Xmx512m
argv[5]: -XX:CompileThreshold=8000
argv[6]: -XX:PermSize=128m
argv[7]: -XX:MaxPermSize=128m
argv[8]: -Xverify:none
argv[9]: -da
argv[10]: -Dplatform.home=/opt/bea/wlp10/wlserver_10.0
argv[11]: -Dwls.home=/opt/bea/wlp10/wlserver_10.0/server
argv[12]: -Dweblogic.home=/opt/bea/wlp10/wlserver_10.0/server
argv[13]: -Dwli.home=/opt/bea/wlp10/wlserver_10.0/integration
argv[14]: -Dweblogic.wsee.bind.suppressDeployErrorMessage=true
argv[15]: -Dweblogic.wsee.skip.async.response=true
argv[16]: -Dweblogic.management.discover=true
argv[17]: -Dwlw.iterativeDev=true
argv[18]: -Dwlw.testConsole=true
argv[19]: -Dwlw.logErrorsToConsole=true
argv[20]: -Dweblogic.ext.dirs=/opt/bea/wlp10/patch_wls1000/profiles/default/syse
xt_manifest_classpath:/opt/bea/wlp10/patch_wlp1000/profiles/default/sysext_manif
est_classpath:/opt/bea/wlp10/patch_wlw1000/profiles/default/sysext_manifest_clas
spath
argv[21]: -Dweblogic.Name=portalServer
argv[22]: -Djava.security.policy=/opt/bea/wlp10/wlserver_10.0/server/lib/weblogi
c.policy
argv[23]: weblogic.Server
```

# Troubleshooting, cont.

- Check the nlwp distribution, *prstat -mL -u weblogic*

| PID | USERNAME | USR | SYS | TRP | TFL | DFL | LCK | SLP | LAT | VCX | ICX | SCL | SIG | PROCESS/LWPID |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1007 | weblogic | 22 | 0.8 | 0.0 | 0.0 | 1.0 | 20 | 9.1 | 47 | 294 | 829 | 5K | 2 | java/14 |
| 1007 | weblogic | 12 | 0.0 | 0.0 | 0.0 | 0.0 | 76 | 0.0 | 12 | 12 | 67 | 14 | 0 | java/9 |
| 1004 | weblogic | 5.1 | 0.3 | 0.0 | 0.0 | 0.0 | 0.0 | 90 | 5.0 | 125 | 415 | 1K | 0 | java/21 |
| 1007 | weblogic | 5.1 | 0.0 | 0.0 | 0.0 | 0.0 | 93 | 0.0 | 1.8 | 17 | 23 | 17 | 0 | java/8 |
| 1007 | weblogic | 2.8 | 0.0 | 0.0 | 0.0 | 0.0 | 96 | 0.4 | 0.4 | 15 | 12 | 33 | 0 | java/2 |
| 1004 | weblogic | 0.9 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 98 | 1.0 | 530 | 38 | 1K | 0 | java/41 |
| 1004 | weblogic | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 99 | 0.0 | 0.4 | 113 | 6 | 223 | 0 | java/2 |
| 1007 | weblogic | 0.3 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.1 | 37 | 8 | 100 | 0 | java/18 |
| 1004 | weblogic | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 15 | 2 | 17 | 0 | java/6 |
| 1007 | weblogic | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 9 | 0 | 52 | 0 | java/20 |
| 1004 | weblogic | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 33 | 1 | 68 | 0 | java/20 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 94 | 6.4 | 136 | 0 | 85 | 0 | java/8 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 13 | 0 | 29 | 0 | java/15 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 94 | 6.5 | 136 | 0 | 85 | 0 | java/11 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 1 | 0 | 1 | 0 | java/4 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.2 | 15 | 0 | 32 | 0 | java/16 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 10 | 0 | 21 | 0 | java/49 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 12 | 0 | 26 | 0 | java/43 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 1 | 0 | 2 | 0 | java/3 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 2 | 0 | 2 | 0 | java/7 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 3 | 0 | 3 | 0 | java/13 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.1 | 1 | 0 | 3 | 0 | java/37 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 5 | 0 | 10 | 0 | java/3 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 5 | 0 | 5 | 0 | java/4 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 2 | 0 | 2 | 0 | java/30 |
| 1006 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 97 | 2.6 | 11 | 0 | 8 | 0 | tail/1 |
| 1007 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 99 | 0.0 | 1.4 | 1 | 0 | 1 | 0 | java/26 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/25 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/24 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/23 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/22 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/19 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/18 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/17 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 0 | 0 | 0 | 0 | java/16 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/14 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0 | 0 | 0 | 0 | java/9 |
| 1004 | weblogic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 100 | 0.0 | 0.0 | 0 | 0 | 0 | 0 | java/7 |

# Troubleshooting, cont.

- Object histogram: *jmap -histo*:

```
Object Histogram:

Size    Count   Class description
-------------------------------------------------------
50406320        426389  char[]
33113656        244287  * ConstMethodKlass
27010904        156914  byte[]
17593272        244287  * MethodKlass
14085760        290326  * SymbolKlass
13028808        542867  java.lang.String
12339416        19018   * ConstantPoolKlass
9357168 193587  java.lang.Object[]
8560960 19018   * InstanceKlassKlass
7637024 17033   * ConstantPoolCacheKlass
7128936 81899   java.util.HashMap$Entry[]
5613256 63787   com.bea.xbean.store.Xobj$AttrXobj
5467056 227794  java.util.HashMap$Entry
3499360 57211   int[]
3250280 36935   com.bea.xbean.store.Cur
3173856 33061   com.bea.xbean.store.Xobj$ElementXobj
3106800 38835   java.lang.reflect.Method
2592768 108032  com.octetstring.vde.syntax.DirectoryString
2436408 7809    com.bea.xbean.schema.SchemaTypeImpl
2398016 74938   java.util.LinkedHashMap$Entry
2359328 2       workshop.util.WeakHashListSet$Entry[]
2043320 51083   java.util.HashMap
2021376 21056   java.lang.Class
1919016 79959   java.util.Vector
1844688 30208   short[]
1811016 75459   java.util.ArrayList
1744840 35721   java.lang.Object[]
1690096 6349    * MethodDataKlass
1654480 20681   java.net.URI
1481200 18515   com.bea.xbean.schema.SchemaPropertyImpl
1476960 46155   weblogic.xml.util.TernarySearchTree$Node
1412064 29418   java.util.LinkedHashMap
1382208 14398   com.bea.xbean.schema.SchemaLocalElementImpl
1301664 54236   javax.xml.namespace.QName
1250056 24286   java.lang.String[]
1150432 35951   com.bea.xbean.store.Locale$Ref
1001184 41716   weblogic.utils.collections.ConcurrentHashMap$Entry
929520  58095   com.octetstring.vde.Attribute
900080  49318   java.lang.Class[]
868104  36171   com.bea.xbean.store.Cursor
816288  25509   com.bea.common.ldap.ParserImpl$LDAPSingleAttributeBuilder
795808  49738   java.util.jar.Attributes$Name
726240  18156   com.bea.common.ldap.ParserImpl$DNFieldBuilder
725776  7731    weblogic.utils.collections.ConcurrentHashMap$Entry[]
```

# Troubleshooting, cont.

- When running *jmap -histo* your JVM target is stopped, the application is not running

```
  PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 1007 weblogic 1089M  597M stop    60    4   0:03:39 2.6% java/40
 1004 weblogic  232M   64M sleep   57    4   0:00:18 0.1% java/45
 1006 weblogic 1264K  780K sleep   59    0   0:00:00 0.0% tail/1
  924 weblogic 1404K 1076K sleep   59    0   0:00:00 0.0% ksh/1
  920 weblogic 1404K 1084K sleep   49    0   0:00:00 0.0% ksh/1
  940 weblogic 1372K 1048K sleep   57    4   0:00:00 0.0% startWebLogic.s/1
```

# Troubleshooting, cont.

- List of all syscalls when clicking 'users'

```
dtrace -n 'syscall:::entry/pid==xxxx/{ @[probefunc] =
   count(); }'
```

```
dtrace: description 'syscall::entry' matched 234 probes
^C

  accept                                                2
  mmap                                                  2
  setsockopt                                            2
  lwp_sigmask                                           3
  setcontext                                            3
  lwp_cond_broadcast                                    8
  ioctl                                                18
  write                                                19
  mprotect                                             36
  lwp_mutex_wakeup                                     65
  lwp_mutex_timedlock                                  69
  lwp_cond_signal                                     124
  lwp_cond_wait                                       206
  pollsys                                             273
  read                                               2388
  send                                               2409
  stat64                                            12003
```

# Troubleshooting, cont.

- Easier, using dtruss in DTT:

```
# /opt/DTT/dtruss -c -p xxxx
```

```
CALL                                        COUNT
accept                                          2
mmap                                            2
resolvepath                                     2
setsockopt                                      2
lwp_sigmask                                     6
setcontext                                      6
close                                          11
open64                                         11
ioctl                                          18
write                                          19
fstat64                                        26
lwp_cond_broadcast                             36
llseek                                         45
mprotect                                       96
lwp_mutex_wakeup                              131
lwp_mutex_timedlock                           146
lwp_cond_signal                               206
lwp_cond_wait                                 370
pollsys                                       480
send                                         3266
read                                         3273
stat64                                      26147
```

```
Elapsed Times for PID 1751,

         SYSCALL          TIME (ns)
      setsockopt              9661
lwp_cond_broadcast            26255
            mmap              28852
lwp_mutex_wakeup             102572
           write             147374
        mprotect             217394
 lwp_cond_signal             538935
            read           12121847
            send           13961864
lwp_mutex_timedlock        17091175
           stat64          65007514
          accept         2293867567
           ioctl         5554610572
         pollsys        13090531784
   lwp_cond_wait        66260832347
           TOTAL:       87309095713

CPU Times for PID 1751,

         SYSCALL          TIME (ns)
      setsockopt              3692
lwp_cond_broadcast            13544
            mmap              24411
          accept              31763
lwp_mutex_wakeup              49750
           write             110763
           ioctl             144829
        mprotect             160169
lwp_mutex_timedlock           199168
 lwp_cond_signal             338367
   lwp_cond_wait            1240415
         pollsys            1888676
            read            6040483
            send            9581449
           stat64          50841963
           TOTAL:          70669442
```

- procsystime

*# /opt/DTT/procsystime -a -p xxxx*

```
Syscall Counts for PID 1751,

         SYSCALL               COUNT
            mmap                   1
          accept                   2
       setsockopt                  2
lwp_cond_broadcast                    6
           ioctl                  18
           write                  19
        mprotect                  24
lwp_mutex_wakeup                  28
lwp_mutex_timedlock                   30
 lwp_cond_signal                  78
   lwp_cond_wait                 141
         pollsys                 232
            read                2355
            send                2376
           stat64               7705
           TOTAL:              13017
```

# Troubleshooting, cont.

- What are we stat-ing so much ?

```
dtrace -qn '
syscall::stat64:entry
/pid==xxxx/
{
    @[strjoin(basename(copyinstr(arg0)), "/")] = count();
}


END
{
    printa ("%-70s %8@d\n", @);
}'
```

# Troubleshooting, cont.

- What are we stat-ing so much ?

```
WLS_DIAGNOSTICS000000.DAT/                                      1
diagnostics/                                                    2
ear-classes.jar/                                                4
wlp-custom.css/                                                10
war-classes.jar/                                               12
cm_content_browse_folder.xml/                                  13
javascriptConstants.properties/                                13
NodeBeanInfo.class/                                            23
AddNodeFormBeanInfo.class/                                     24
ItemGroupFormBeanInfo.class/                                   24
en/                                                            39
ActionFormBeanInfo.class/                                      48
FormBeanInfo.class/                                            48
default.chromosome/                                            90
JavaControlFactory.class/                                     112
en_US/                                                        114
ObjectClassBeanInfo.class/                                    138
ContentEntityBeanInfo.class/                                  161
structure.xml/                                                180
DefaultControlBeanContextFactory.class/                       344
org.apache.beehive.controls.spi.bean.ControlFactory/         1008
ContentNodeControlBean.ser/                                  1064
ContentRepositoryControlBean.ser/                            1140
org.apache.beehive.controls.spi.context.ControlBeanContextFactory/  3096
```

# Troubleshooting, cont.

- jstack()

```
libc.so.1`stat64+0x15
java/io/UnixFileSystem.getLength*
nmethod
0xbf505b50
  1

libc.so.1`stat64+0x15
java/io/UnixFileSystem.getLastModifiedTime*
nmethod
0xbf505b50
  1

libc.so.1`stat64+0x15
java/io/UnixFileSystem.getLastModifiedTime
java/io/File.lastModified
0xc7852c30
  8

libc.so.1`stat64+0x15
java/io/UnixFileSystem.getBooleanAttributes0*
nmethod
0xbf505b50
  9
```

```
libc.so.1`stat64+0x15
java/io/UnixFileSystem.getBooleanAttributes0*
nmethod
0xbf505b50
7686
```

# Troubleshooting, cont.

- Easy to drill down and aggregate on each method, using dvm agent:

```
#pragma D option quiet

dvm$target:::method-entry
{
    @[strjoin(strjoin(basename(copyinstr(arg0)), "."),
            copyinstr(arg1))] = count();
    /* @[copyinstr(arg0),copyinstr(arg1)] = count(); */
}


dtrace:::END
{
    printa("%-73s %7@d\n",@);
}
```

| | |
|---|---|
| HashMap.get | 95720 |
| NullClassFinder.getSource | 103439 |
| SignatureParser.current | 107520 |
| AbstractStringBuilder.expandCapacity | 111105 |
| AbstractStringBuilder.<init> | 111804 |
| CharacterDataLatin1.getProperties | 124956 |
| StringCharBuffer.get | 129564 |
| Buffer.nextPutIndex | 129722 |
| Buffer.nextGetIndex | 134871 |
| HashMap.oldHash | 135067 |
| HashMap.hash | 136584 |
| HeapByteBuffer.put | 137580 |
| AbstractList$Itr.next | 138050 |
| AbstractList$Itr.checkForComodification | 138074 |
| HashMap.indexFor | 139095 |
| HeapByteBuffer.ix | 143217 |
| String.startsWith | 166006 |
| String.hashCode | 170089 |
| CopyOnWriteArrayList$COWIterator.<init> | 172580 |
| AbstractList$Itr.hasNext | 175566 |
| String.<init> | 182507 |
| StringBuilder.append | 195670 |
| String.getChars | 195919 |
| AbstractStringBuilder.appendCodePoint | 201627 |
| Character.isValidCodePoint | 201627 |
| ResourceIDBuilder$ParseContext.append | 201627 |
| ResourceIDBuilder$ParseContext.getTokenBufferInternal | 201627 |
| StringBuilder.appendCodePoint | 201627 |
| String.equals | 208242 |
| ResourceIDBuilder$ParseContext.read | 218951 |
| CopyOnWriteArrayList$COWIterator.next | 225603 |
| ResourceIDBuilder$ParseContext.isAtEnd | 227613 |
| URLClassPath$JarLoader.findResource | 278208 |
| Buffer.hasRemaining | 279972 |
| AbstractStringBuilder.append | 285408 |
| JarFile.getEntry | 297208 |
| JarFile.getJarEntry | 297208 |
| URLClassPath.getLoader | 305216 |
| URLClassPath$JarLoader.getResource | 306604 |
| CopyOnWriteArrayList$COWIterator.hasNext | 311865 |
| ZipFile.getEntry | 348179 |
| ZipFile.ensureOpen | 348187 |
| ArrayList.get | 494029 |
| ArrayList.RangeCheck | 494502 |
| ArrayList.size | 524805 |
| String.indexOf | 543499 |
| String.length | 4550897 |
| String.charAt | 5861122 |

For one click you can see how many calls from URLClassPath. As well a lot of String operations.

# Troubleshooting, cont.

- How about execution time ?

```
dvm$target:::method-entry
{
    self->start=vtimestamp;
}
dvm$target:::method-return
/self -> start/
{
    this->time = vtimestamp - self->start;
    @Time[copyinstr(arg0)] = sum(this->time);
    @Time["TOTAL:"] = sum(this->time);
    self->time = 0;
}
dtrace:::END
{ printa("%-69s %11@d\n", @Time); }
```

```
weblogic/transaction/internal/ServerTransactionImpl                    99393351
com/pointbase/net/netJDBCPrimitives                                   104046512
java/lang/IndexOutOfBoundsException                                   106599191
com/bea/portal/tools/resource/framework/TaggedResourceIDBuilder       106631804
java/lang/RuntimeException                                            107567424
java/lang/ArrayIndexOutOfBoundsException                              108302961
weblogic/ejb/container/internal/MethodDescriptor                      108380668
java/lang/Integer                                                     109012325
weblogic/transaction/internal/TransactionManagerImpl                  113939188
weblogic/utils/classloaders/MultiClassFinder                          117168908
com/bea/portal/tools/resource/ResourceIDBuilder                       126668488
java/lang/Throwable                                                   127877648
weblogic/transaction/internal/TransactionImpl                         128852529
java/lang/Exception                                                   130209564
sun/reflect/generics/parser/SignatureParser                           131895650
java/io/BufferedInputStream                                           149673077
weblogic/utils/collections/CopyOnWriteArrayList                       174292972
java/nio/StringCharBuffer                                             181152121
weblogic/utils/io/FilenameEncoder                                     181489874
java/util/AbstractList                                                188220758
java/util/regex/Pattern                                               194935739
java/lang/Class                                                       202016251
com/bea/netuix/nf/UIControl                                           205851384
java/lang/StringBuffer                                                215402224
java/lang/ClassLoader                                                 216144397
java/lang/CharacterDataLatin1                                         236419417
weblogic/utils/classloaders/ZipClassFinder                            266166243
java/nio/HeapByteBuffer                                               298576858
java/util/LinkedList                                                  319247501
com/bea/content/repo/internal/client/common/PropertyDefinitionData    328648451
java/nio/Buffer                                                       460468952
weblogic/utils/collections/CopyOnWriteArrayList$COWIterator           531601999
java/util/AbstractList$Itr                                            548096537
weblogic/utils/enumerations/SequencingEnumerator                      583190280
java/util/HashMap                                                     715050024
sun/misc/URLClassPath                                                 775178668
java/lang/AbstractStringBuilder                                       785866662
java/lang/Character                                                   805863598
java/lang/StringBuilder                                              1041245034
com/bea/portal/tools/resource/ResourceIDBuilder$ParseContext         1158201051
java/util/ArrayList                                                  1489611798
java/util/zip/ZipFile                                                1615423076
java/util/jar/JarFile                                                2901572805
sun/misc/URLClassPath$JarLoader                                      3555506306
java/lang/String                                                     8445270186
TOTAL:                                                              41453153913
```

The time spent calling these methods, interesting here to notice the URLClassPath$JarLoader. Time is recorded in ns.

# Troubleshooting, cont.

- **Conclusions**
  - The portalAppAdmin application is doing a lot of activity on classloader, making the application a bit slow on some configurations. Same thing is visible in the commercial Portal Administrative Tool, PAT
  - Possible the way the application has been developed. The case has been escalated with BEA
  - A nice example: how easily can you use free tools to debug and analyse your application

# Troubleshooting, cont.

- Possible cause of this high number of stat64() calls

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

*Paper detailing the "Best Practices for Developing High Performance Web and Enterprise Applications" using IBM's WebSphere. All the tips are generally applicable to servlet/EJB development, as well as other types of server development. (Page last updated September 2000, Added 2001-01-19, Author Harvey W. Gunther, Publisher IBM)*

  - Access entity beans from session beans, not from client or servlet code.
  - Reuse EJB homes
  - Use Read-Only methods where appropriate in entity-beans to avoid unnecessary invocations to store.

# Troubleshooting, cont.

- The EJB "remote programming" model always assumes EJB calls are remote, even where this is not so. Where calls are actually local to the same JVM, try to use calling mechanisms that avoid the remote call.
- Remove stateful session beans (and any other unneeded objects) when finished with, to avoid extra overheads in case the container needs to be passivated.
- **Beans.instantiate() incurs a filesystem check to create new bean instances. Use "new" to avoid this overhead.**

# Credits

Sun Microsystems:

- Jon Masamitsu, Chee-Weng Chea
- Sundar Athijegannathan, Alan Bateman
- Peter B. Kessler, Martin Steinkopf
- Jeff Savit, Stefan Hinker
- Richard Smith, Brian Doherty, Jeff Taylor

DTrace community:

- http://www.opensolaris.org/os/community/dtrace/
- Jarod Jenson, Brendan Gregg

JMeter Performance Testing:

- Alex Stoenescu

# Appendix

Solaris 10

- http://www.sun.com/software/solaris/

Java SE

- http://java.sun.com/javase/

DTraceToolkit and DTrace

- http://www.opensolaris.org/os/community/dtrace/

jvmstat

- http://java.sun.com/performance/jvmstat/

JMeter

- http://jakarta.apache.org/jmeter/

# Appendix, License

## Public Documentation License Notice

開
放
的
열린
مفتوح
libre
मुक्त
ముక్త
livre
libero
ಮುಕ್ತ
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை


# Thank you!

Stefan Parvu
System Administrator
stefan.parvu@sun.com
stefanparvu14@yahoo.com

http://www.nbl.fi/stefan.parvu

“open” artwork and icons by chandan:
http://blogs.sun.com/chandan