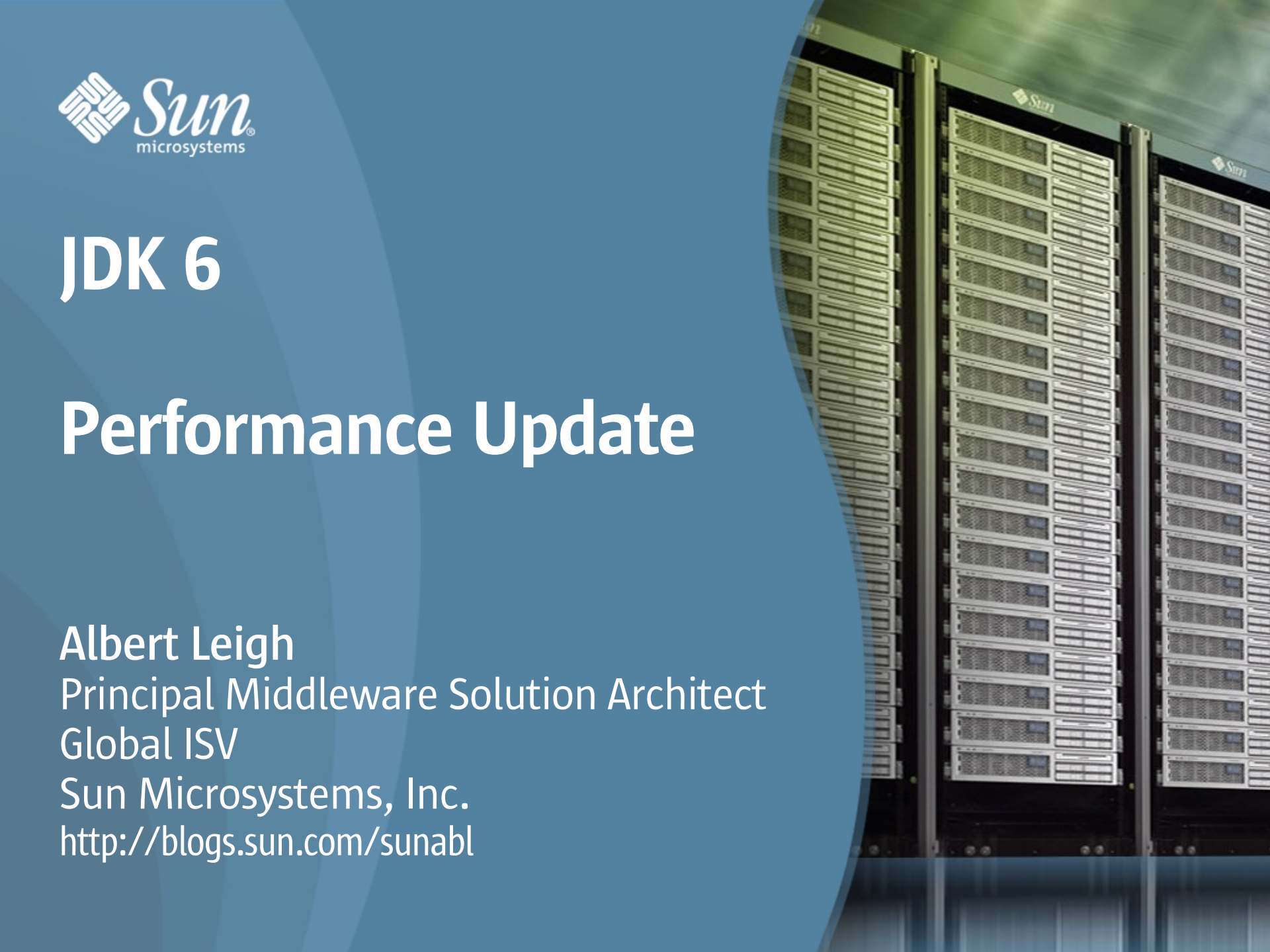


JDK 6

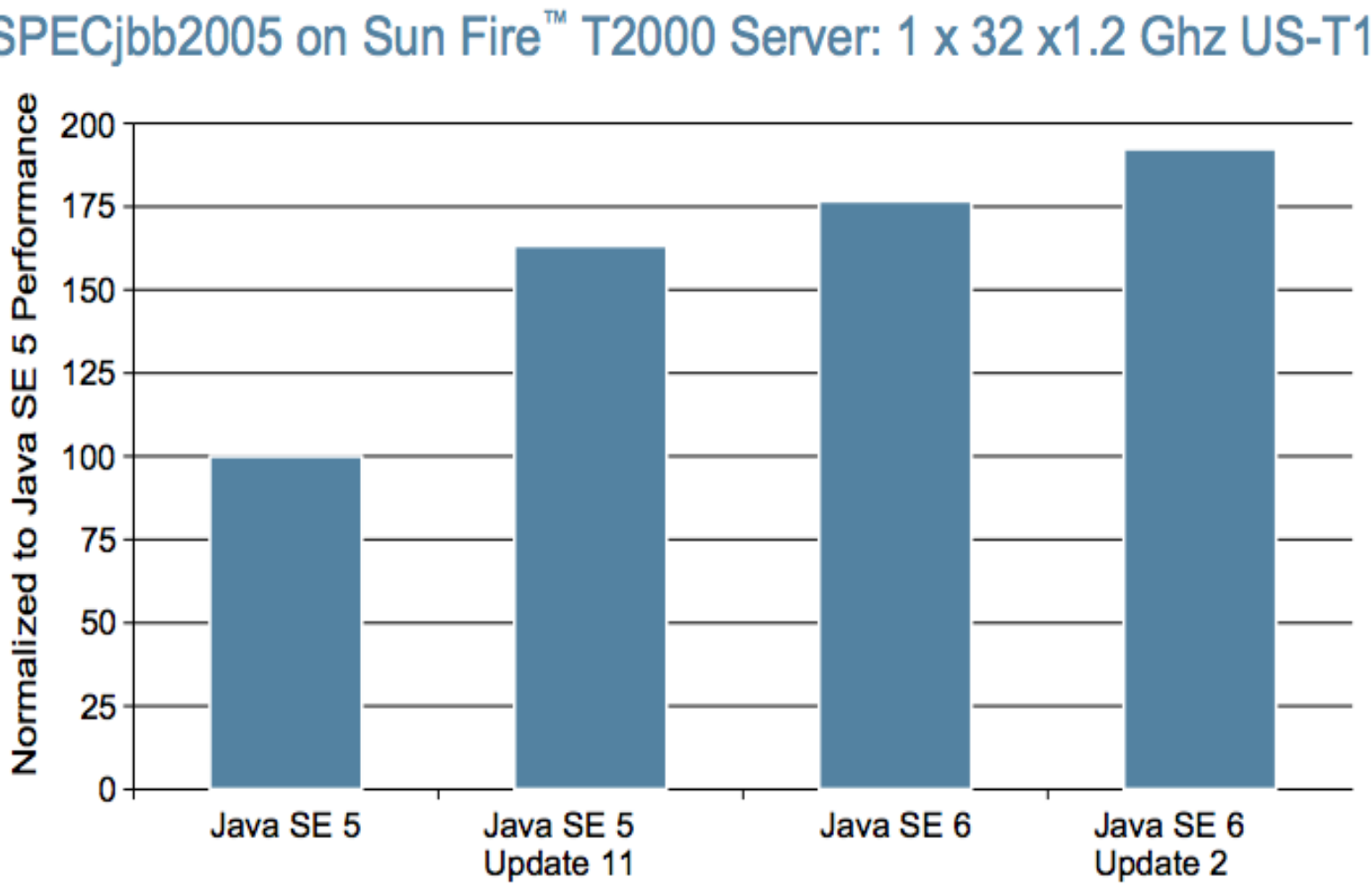
Performance Update

Albert Leigh
Principal Middleware Solution Architect
Global ISV
Sun Microsystems, Inc.
http://blogs.sun.com/sunabl

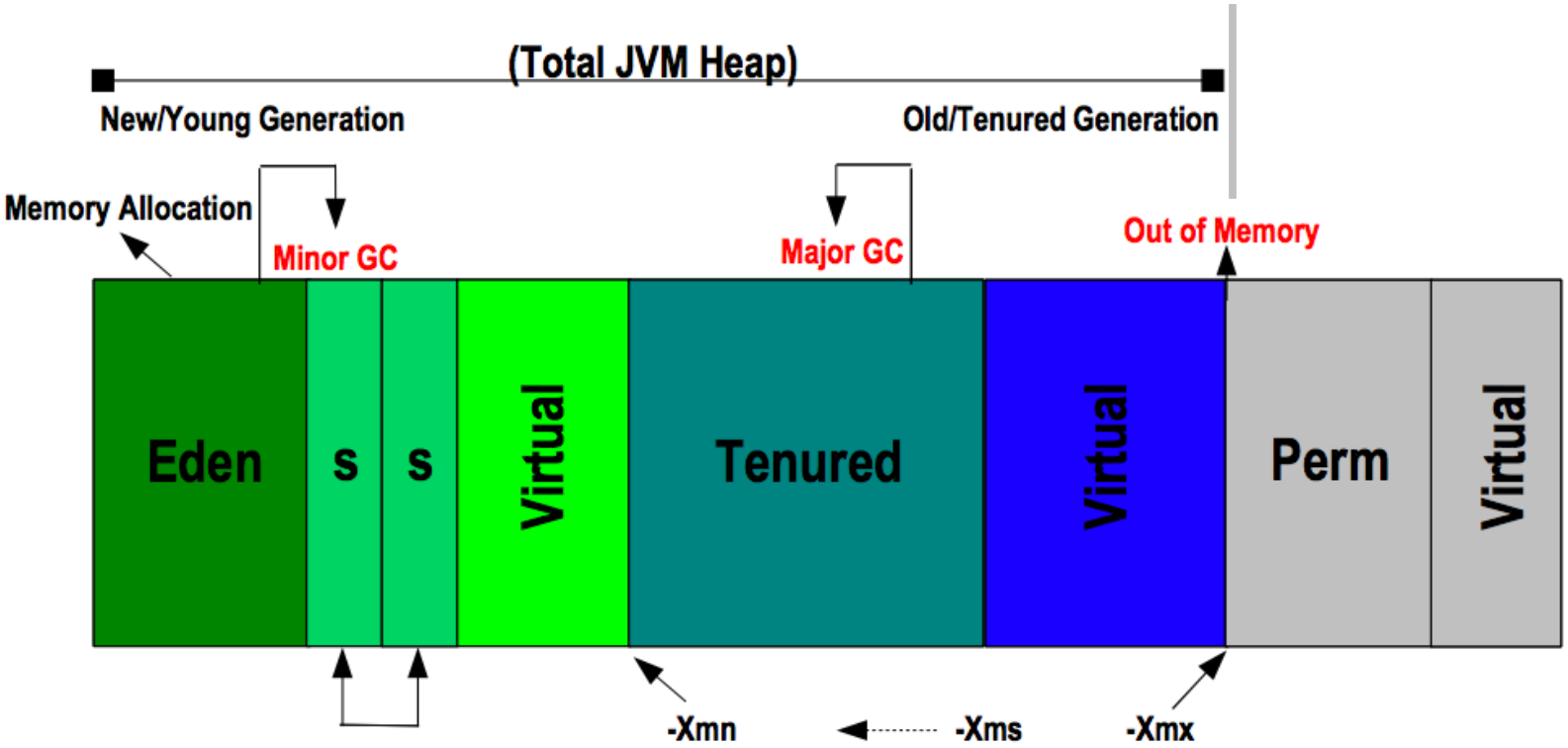
Java Platform, Standard Edition: Java SE 6

- Key Design Goal
  - Improve Performance and Scalability
- New features and Performance Enhancements
  - Garbage Collection
  - Ergonomics
  - Runtime Performance Optimization
  - Client-side Performance Features and Improvements
- Tools for Observability
- Java SE 6 Update – Performance Releases
  - More significant improvements in Java SE 6 Update 14 and newer

JDK Performance Improvement

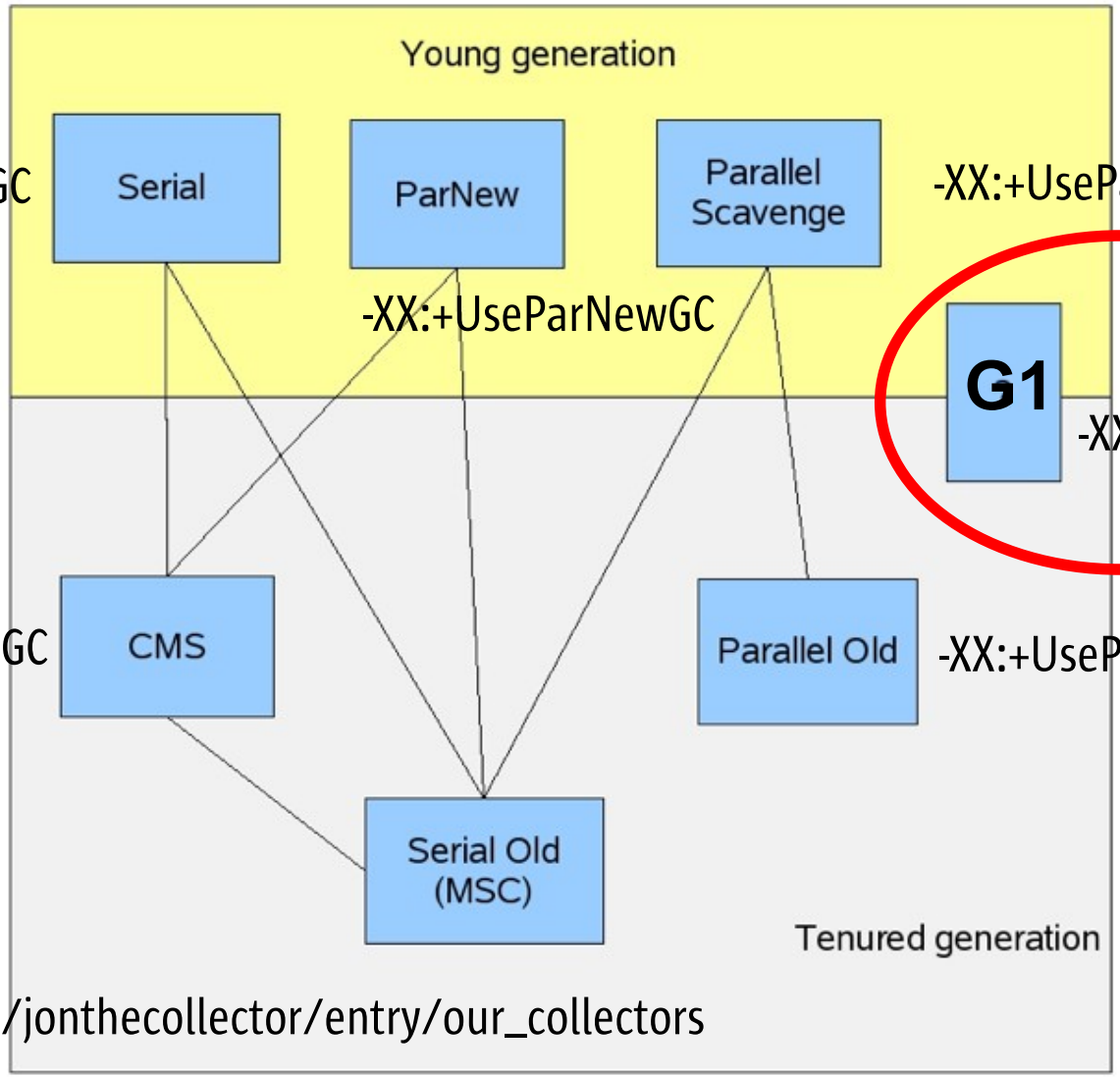


JVM Heap Layout & Generational Spaces



- Young Generation
  - Eden where new objects get instantiated
  - 2 Survivor Spaces to hold live objects during minor GC
- Old Generation
  - Tenured objects
- Permanent Generation
  - JVM class information, etc.

Sun HotSpot Garbage Collectors in Java SE 6



http://blogs.sun.com/jonthecollector/entry/our_collectors

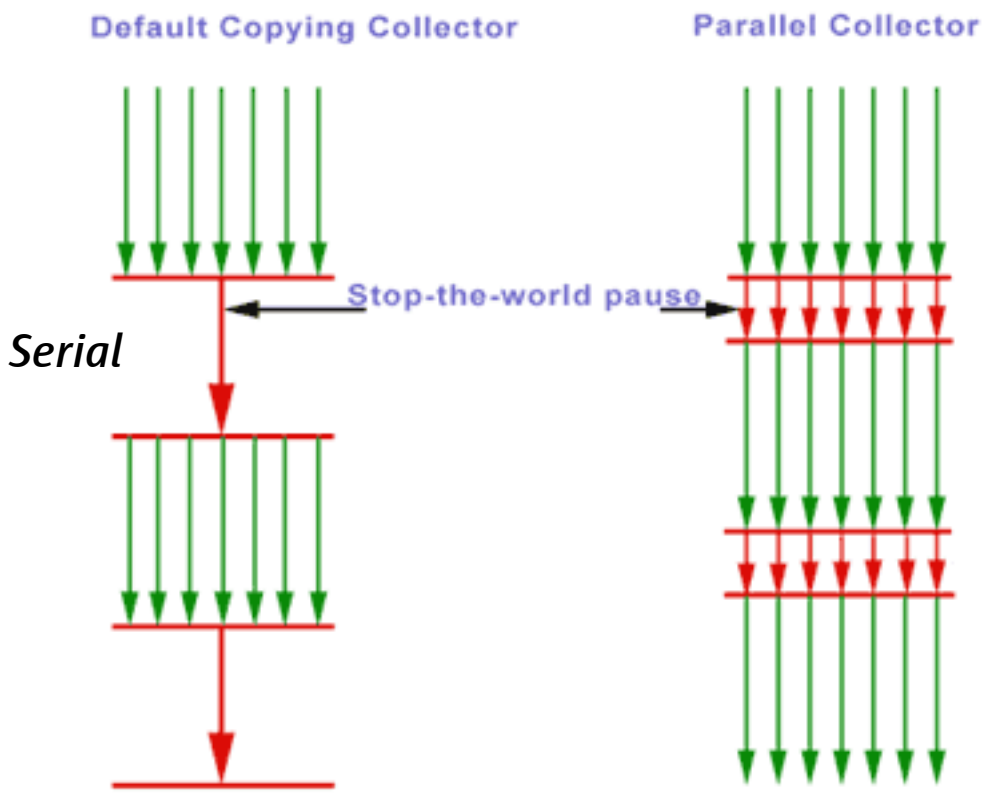
Types of GC Collector

- Serial Collector (-XX:+UseSerialGC)
- Throughput Collectors
  - Parallel Scavanging Collector for Young Gen
    - -XX:+UseParallelGC
  - Parallel Compacting Collector for Old Gen
    - -XX:+UseParallelOldGC (on by default with ParallelGC in JDK 6)
- Concurrent Collector
  - Concurrent Mark-Sweep (CMS) Collector
    - -XX:+UseConcMarkSweepGC
  - Concurrent (Old Gen) and Parallel (Young Gen) Collectors
    - -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
- The new G1 Collector as of Java SE 6 Update 14 (-XX:+UseG1GC)

Sample JVM Parameters

- Performance Goals and Exhibits
  - A) High Throughput (e.g. batch jobs, long transactions)
  - B) Low Pause and High Throughput (e.g. portal app)
- JDK 6
  - A) -server -Xms2048m -Xmx2048m -Xmn1024m -XX:+AggressiveOpts -XX:+UseParallelGC -XX:ParallelGCThreads=16
  - B) -server -Xms2048m -Xmx2048m -Xmn1024m -XX:+AggressiveOpts -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:ParallelGCThreads=16
- JDK 5
  - A) -server -Xms2048m -Xmx2048m -Xmn1024m -XX:+AggressiveOpts -XX:+UseParallelGC -XX:ParallelGCThreads=16 -XX:+UseParallelOldGC -XX:+UseBiasedLocking
  - B) -server -Xms2048m -Xmx2048m -Xmn1024m -XX:+AggressiveOpts -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:ParallelGCThreads=16 -XX:+UseBiasedLocking

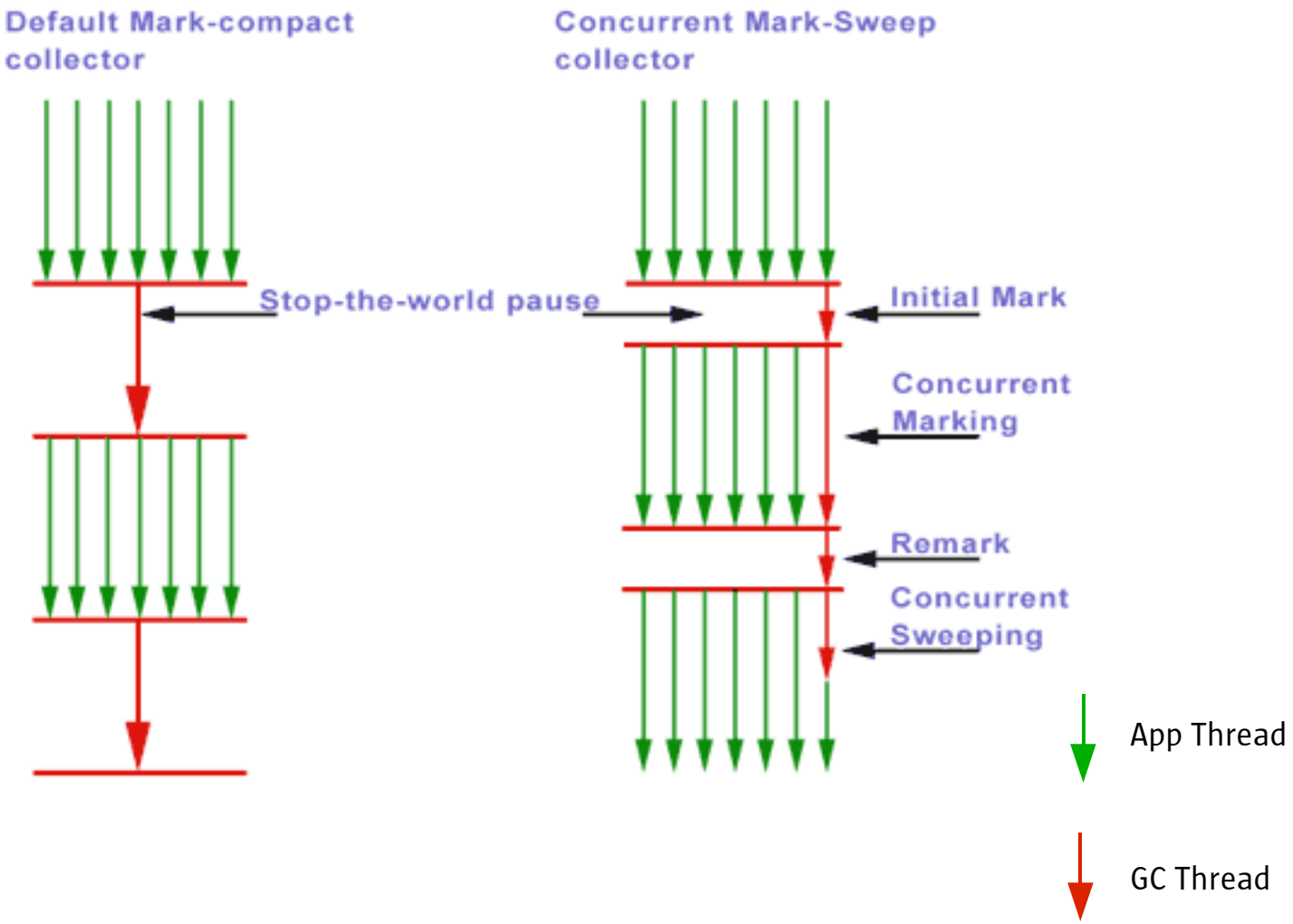
Serial vs Parallel Collector



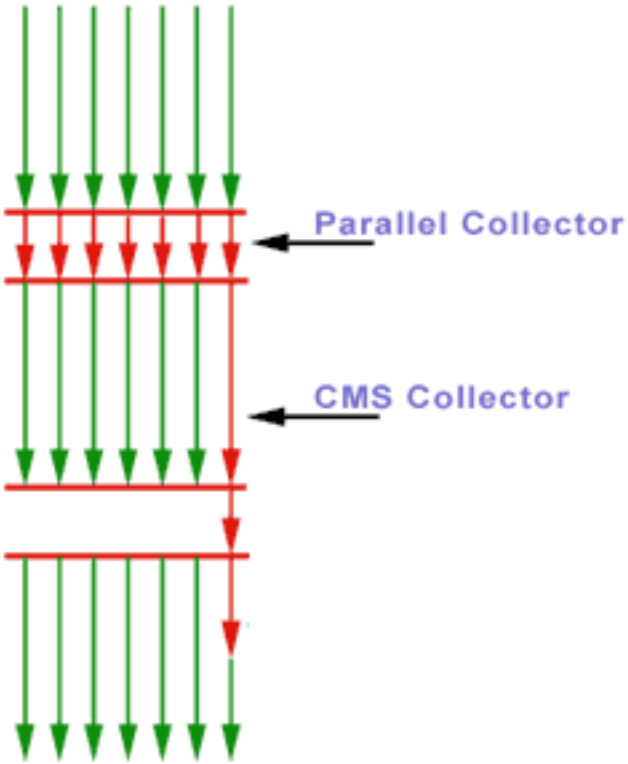
App Thread

GC Thread

CMS Collector



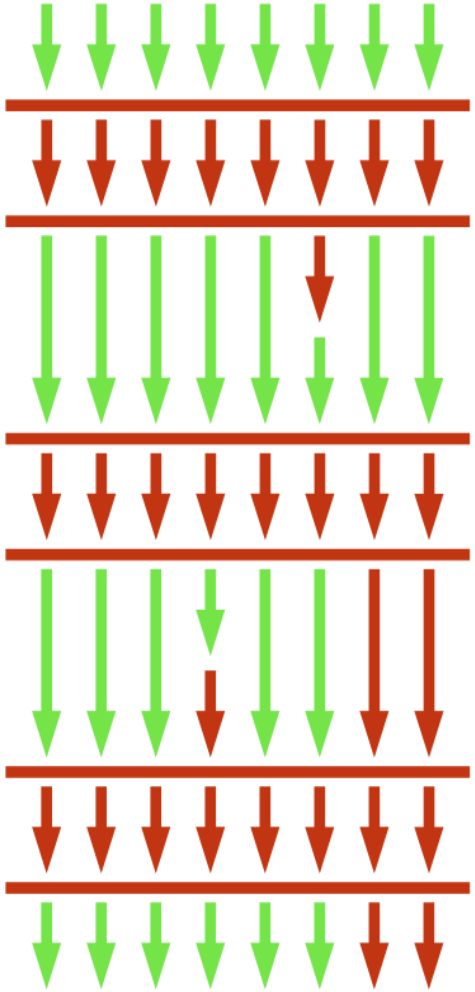
CMS Collector with ParNewGC



App Thread

GC Thread

G1 Collector: Parallelism & Concurrency



App Thread

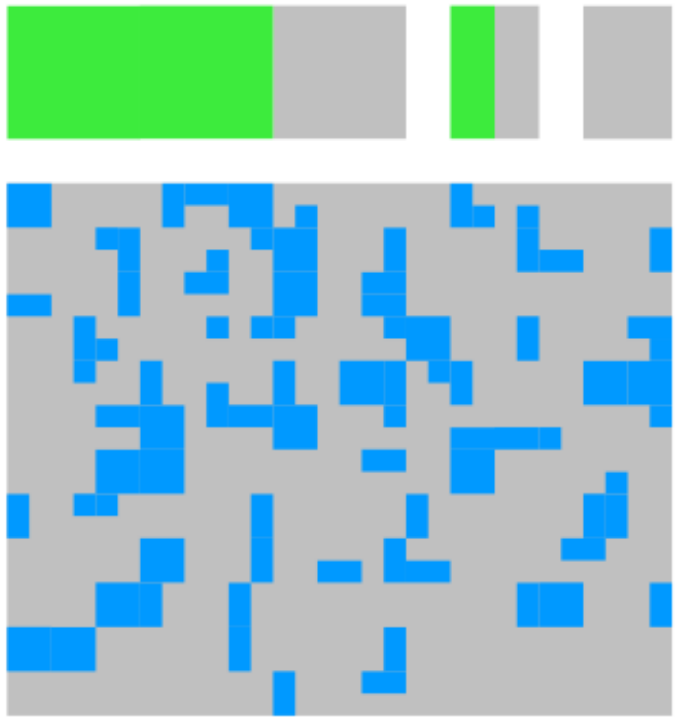
GC Thread

The G1 Collector

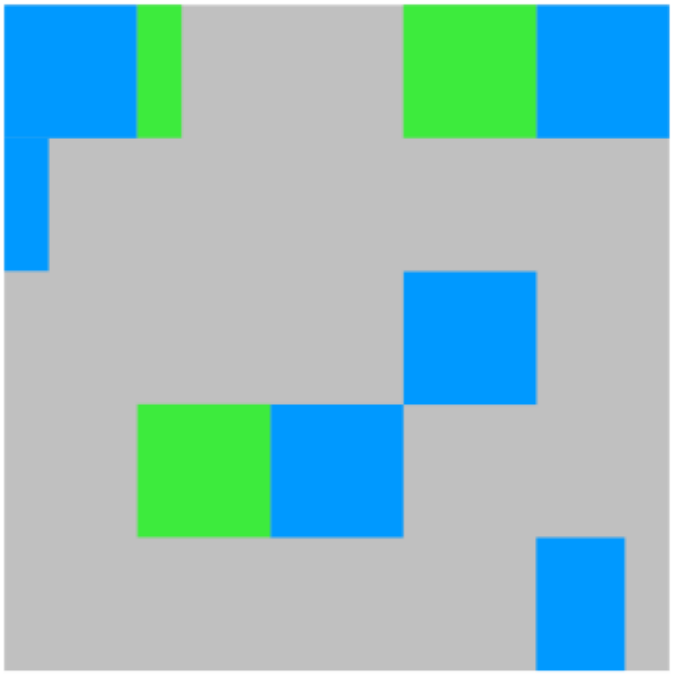
- Future CMS Replacement
- Server “Style” Garbage Collector
- Parallel
- Concurrent
- Generational
- Good Throughput
- Compacting
- Improved ease-of-use
- Predictable (though not hard real-time)
- JVM Options: -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC

CMS vs G1 Collectors

- Heap split into regions
- Young generation (A set of regions)
- Old generation (A set of regions)



CMS



G1

Non-Allocated Space

Young Generation

Old Generation

Young GC's in G1

- Single physical heap, split into regions
  - Set of contiguous regions allocated for large ("humongous") objects
- No physically separate young generation
  - A set of (non-contiguous) regions
  - Very easy to resize
- Young GCs
  - Done with "evacuation pauses"
  - Stop-the-world
  - Parallel
  - Evacuate surviving objects from one set of regions to another

Old GC's in G1

- Concurrent marking phase
  - Calculates liveness information per region
  - Identifies best regions for subsequent evacuation pauses
  - No corresponding sweeping phase
  - Different marking algorithm than CMS
  - Snapshot-at-the-beginning (SATB)
  - Achieves shorter remarks
- Old regions reclaimed by
  - Remark (when totally empty)
  - Evacuation pauses
- Most reclamation happens with evacuation pauses
  - Compaction

JVM Ergonomics

- Java SE 5.0 and newer – improved self tuning in Java 6
  - Provide good performance from the JVM with a minimum of command line tuning
- Best selection of
  - Garbage Collector (Make sure to override GCThreads)
  - Heap Size (-Xms == 1/64 Max Memory or Max Heap and -Xmx == ¼ Max Memory or Max Heap)
  - Runtime Compiler (-server vs -client)
- Desired Goals (This is a hint, not a guarantee)
  - Maximum Pause Time (-XX:MaxGCPauseMillis=<n>)
  - Application Throughput (-XX:GCTimeRatio=<n> where Application time = 1 / (1 + n))

Runtime Performance Enhancements

- Synchronization (Uncontended) improvements in Java SE 6
  - Lock Coarsening (on by default. -XX:+EliminateLocks)
    - Reduce lock overhead by broadening an existing synchronized block
    - Eliminates the unlock and re-lock operations in where pattern exists
  - Lock Elision through Escape Analysis (-XX:+DoEscapeAnalysis)
    - Locally referenced object's synchronized block is thread local
    - Can omit the lock because it is uncontended
  - Biased Locking (on by default. -XX:+UseBiasedLocking)
    - Keep the lease of a lock to the thread that created it until another thread contends it
    - Available in JDK 5 (as of Update 6) as an option

http://work.tinou.com/2009/06/lock-coarsening-biased-locking-escape-analysis-for-dummies.html

Runtime Performance Enhancements

- Synchronization (Contended) improvement with Adaptive Spinning
  - A two-phase spin-then-block strategy is used by threads attempting a contended synchronized enter operation
  - Reduction of context switching and repopulation of Translation Look-aside Buffers (TLBs)
  - http://blogs.sun.com/dave/entry/java_util_concurrent_reentrantlock_vs
- Array Copy Performance Improvements
- Support for large page heap on AMD and Intel 64-bit platforms
  - On Solaris, large pages are on by default
- Other HotSpot™ Compiler improvements
  - Background Compilation in Client Compiler
  - Better implementation for performance

Tools for Observability: Java SE 5, 6

- jvmstat are now standard
  - http://java.sun.com/performance/jvmstat
  - jps, jstat, jinfo, etc.

```
$ jps
11072 Jps
9465 WSPreLauncher
```

- jps is equivalent to the Solaris proc tool ps
  - Different from doing: pgrep java OR ps -ef | grep java
  - Finds all JVM instances of the current user, even the ones that don't use the java executable (for example, custom launchers)
- jinfo provides info about the Java process such as JVM options, path, etc.
- jstat

```
$JAVA_HOME/bin/jstat -gcutil 9465 10000 5
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
0.00  15.29  71.80   0.00  71.02      1    0.123      0    0.000    0.123
0.00  15.29  71.80   0.00  71.02      1    0.123      0    0.000    0.123
0.00  15.29  71.80   0.00  71.02      1    0.123      0    0.000    0.123
0.00  15.29  71.80   0.00  71.02      1    0.123      0    0.000    0.123
0.00  15.29  72.05   0.00  71.02      1    0.123      0    0.000    0.123
```

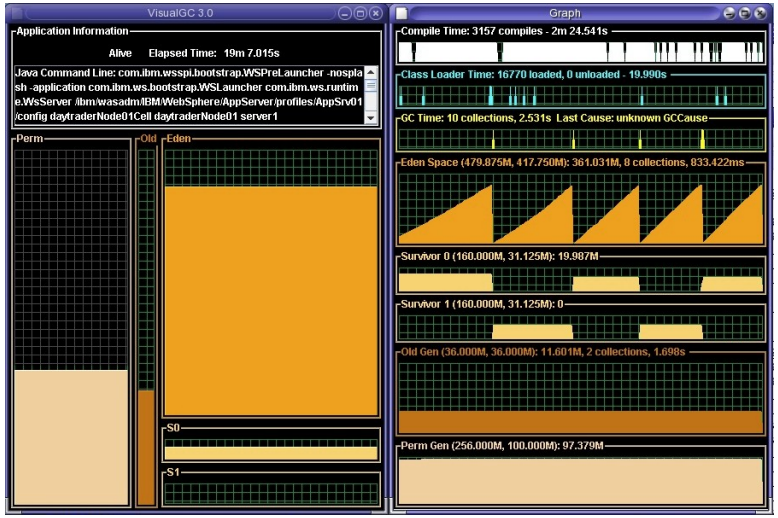
Tools for Observability: Java SE 5, 6

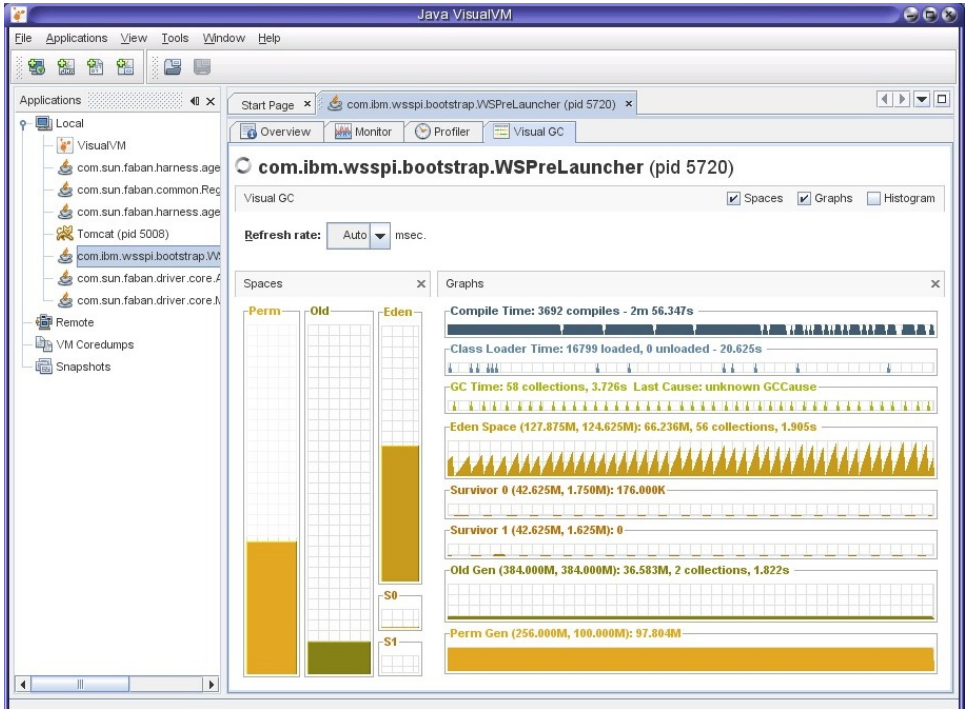
- Querying JVM Heap with jmap

```
$JAVA_HOME/bin/jmap -heap 9465

Attaching to process ID 2501, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.5.0_09

using thread-local object allocation.
Parallel GC with 24 thread(s)

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 1258291200 (1200.0MB)
   NewSize          = 471859200 (450.0MB)
   MaxNewSize       = 471859200 (450.0MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 2
   SurvivorRatio    = 8
   PermSize         = 16777216 (16.0MB)
   MaxPermSize      = 268435456 (256.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 355860480 (339.375MB)
   used     = 157075456 (149.798828125MB)
   free     = 198785024 (189.576171875MB)
   44.139617863720076% used
From Space:
   capacity = 11534336 (11.0MB)
   used     = 11534336 (11.0MB)
   free     = 0 (0.0MB)
   100.0% used
To Space:
   capacity = 61865984 (59.0MB)
   used     = 0 (0.0MB)
   free     = 61865984 (59.0MB)
   0.0% used

PS Old Generation
   capacity = 788529152 (752.0MB)
   used     = 720840944 (687.4474945068359MB)
   free     = 67688208 (64.55250549316406MB)
   91.41589022697286% used

PS Perm Generation
   capacity = 268435456 (256.0MB)
   used     = 176482696 (168.30701446533203MB)
   free     = 91952760 (87.69298553466797MB)
   65.74492752552032% used
```

Tools for Observability

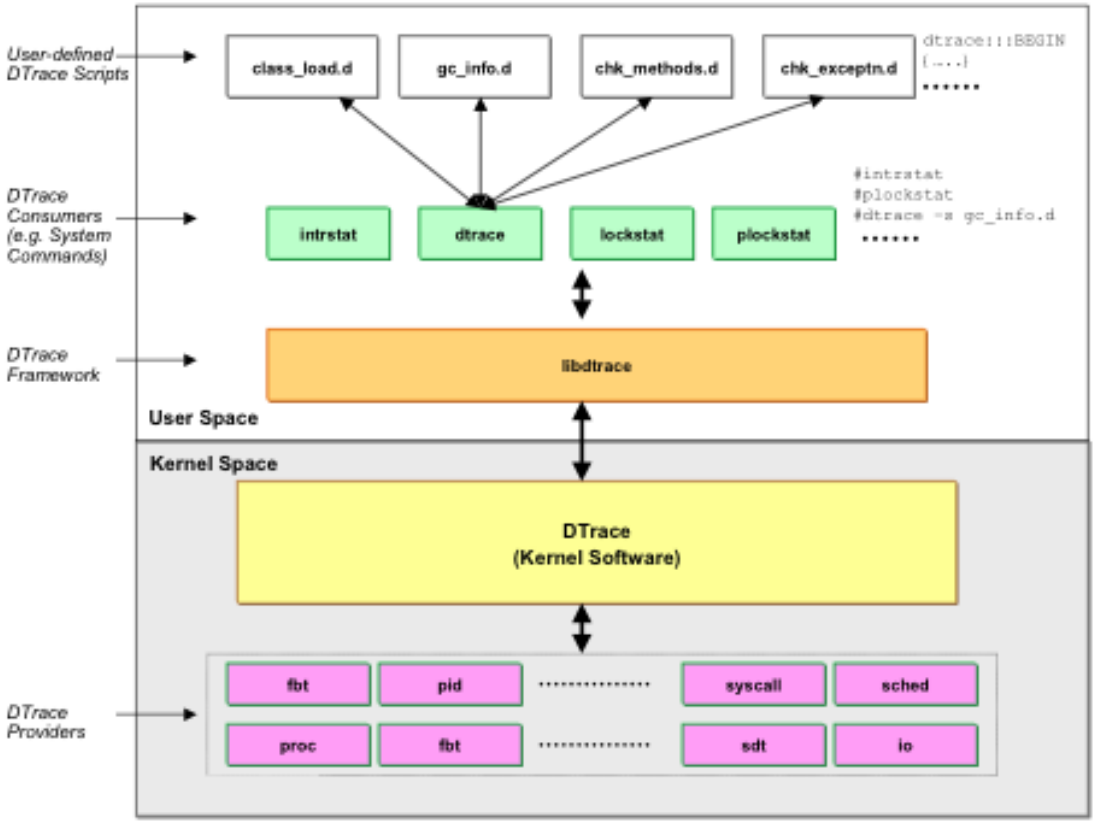
- VisualGC
  - https://java.sun.com/performance/jvmstat/visualgc.html
- VisualVM (VisualGC Plug-in must be downloaded)
  - Available as of JDK 6 Update 7 in $JAVA_HOME/bin/jvisualvm
  - https://visualvm.dev.java.net





Performance Tuning – Tools DTrace

- Available as of Solaris 10
- Is a comprehensive dynamic tracing facility
- Concise Answers to Arbitrary Questions



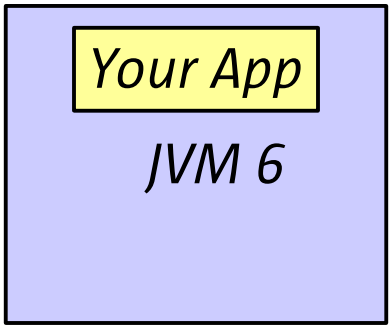
- Integrated hotspot provider in JDK 6
- DVM Providers download available
  - > Use JVMTI for JDK 5
  - > Use JVMPI for JDK 1.4.2
- Many other DTrace samples and DTrace Toolkit on the internet

Tools for Observability: BTrace

- BTrace (Byte-code Instrumentation) based on DTrace
  - VisualVM BTrace Plug-in available
  - Many examples on https://btrace.dev.java.net

```
1  import com.sun.btrace.annotations.*;
2  import static com.sun.btrace.BTraceUtils.*;
3
4
5  @BTrace
6  public class GC {
7
8      // @OnMethod annotation tells w probe.
9      // In this example, we probe into entry
10     // into the System.gc() method
11     @OnMethod(
12         clazz="java.lang.System",
13         method="gc"
14     )
15     public static void func() {
16         // jstack() and exit() is in BTraceUtils
17         // you can only call the methods of BTraceUtils
18         jstack();
19         exit(0);
20     }
21 }
```

Byte Code Instrumentation




```
bash-3.00$ btrace 19579 GC.java
btrace.GcWorkingObject.loopWork(Unknown Source)
btrace.DefaultWorkingObject.execute(Unknown Source)
btrace.FreeLockWorkingThread.run(Unknown Source)
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:8
86)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
java.lang.Thread.run(Thread.java:619)
```

Improvements for 64-bit JVM

- With 64-bit address space, Java heap size can be greater than ~3.5GB limitation that 32-bit JVM has
- Performance improvement depends
  - For apps that require larger heap, reduce GC cycles
  - Adverse effect can be in GC pause time
  - All address references are 64-bit wide, twice the size of address references in 32-bit deployments
- Ordinary Object Pointer a managed pointer to an object
  - Enable 64-bit JVM to be able to address with compressed object references to 32-bit managed pointer values
  - JVM Option: **-XX:+UseCompressedOops**
  - http://wikis.sun.com/display/HotSpotInternals/CompressedOops

References

- JDK 6 Performance Performance Enhancements
  - > http://java.sun.com/performance/reference/whitepapers/6_performance.html
  - > http://java.sun.com/performance/reference/whitepapers/tuning.html
  - > http://blog.xebia.com/2007/12/21/did-escape-analysis-escape-from-java-6/
- JDK 6 Monitoring and Obervability Tools
  - http://www.sun.com/bigadmin/features/articles/java_se6_observability.jsp
- A Collection of JVM Options
  - > http://blogs.sun.com/watt/resource/jvm-options-list.html
- HotSpot GC and G1 (Garbage First) Collector
  - > http://blogs.sun.com/jonthecollector/entry/our_collectors
  - > http://www.ddj.com/java/219401061
  - > http://research.sun.com/jtech/pubs/04-g1-paper-ismm.pdf
- CompressedOops to 64-bit JVM for better performance
  - > http://wikis.sun.com/display/HotSpotInternals/CompressedOops

Q & A

Thank You!

Albert Leigh
Sun Microsystems, Inc.

http://linkedin.com/in/albertleigh