Java Performance Tuning
From A Garbage Collection Perspective

Nagendra Nagarajayya
MDE
Agenda

• Introduction To Garbage Collection
• Performance Problems Due To Garbage Collection
• Performance Tuning
  – Manual
  – Automatic
• Application Monitoring – Garbage Collection Perspective
• Summary
Introduction To Garbage Collection
What is Garbage Collection (GC)?

• GC enables automatic memory recycling
  – This part of JVM called the Garbage collector

• Most collectors are stop-the-world
  – Application is paused while the garbage collector runs

• Generational system
  – Young generation
  – Old generation
  – Permanent generation
**Generational Garbage Collection**

- **Efficient memory re-cycling**
  - **Aging of objects**

  - Program Allocation
    - Young Generation
      - Objects are allocated by the program in the young generation
      - Young Generation constantly filters out most objects (2/3), moves the rest to old generation (promotion)
    - Old Generation
      - Old generation reclaims objects once in a while

Two Types Of Collectors

- Low Pause Collectors
  - For applications where pause is a criteria, like Real Time Applications (telco), VoIP, Call Centers (Response time), Front Ends (GUI)

- Throughput Collectors
  - For enterprise applications where pause is not such a criteria, like Financial systems, Application Servers, Billing Systems, Health Care, Decision Support Systems
Low Pause Collectors
– to serve the needs of applications where pause is a criteria

- **Parallel Copy Collector**
  - Use many threads to process young generation collection
  - Still stop-the-world
  - Cost of collection is now dependent on live data and number of CPUs (single threaded pause / CPUs)

- **Concurrent Old Generation Collector**
  - Use one thread to process collection while application threads run on remaining CPUs
  - Cost is a percentage of CPU
Low Pause Collectors

Parallel Copy Collector (Young Generation)  Concurrent Mark-Sweep Collector (Old Generation)

Application Threads

E Eden  O Old Generation Heap  O1 Old Generation Heap After A Concurrent Collection
Throughput Collectors
- serve the needs of enterprise applications where pause is not such a criteria

• **Parallel Throughput Collector**
  - Uses many threads to process young generation collection
  - Works very well with large young generation heaps, and lots of CPUs
  - Still, stop-the-world collection
  - Cost of collection is dependent on live data and CPUs

• **Mark-Compact Old Generation Collector**
  - Uses one thread to process collection
  - Stop-the-world collection
Throughput Collectors

Parallel Throughput Collector
(Young Generation)

Mark-Sweep Collector
(Old Generation)

E Eden
S Survivor Space
O Old Generation Heap
O1 Old Generation Heap After Collection
Heap Layout For The Collectors

• Low Pause Collectors
  – Smaller young generations and larger older generations
  – Keep young generation pause low
  – Old generation concurrent collection pause should be low

• Throughput Collectors
  – Larger young generations, and smaller old generations
  – Young generation pause is dependent on number of CPUs
  – Old generation pause could be big, so keep old generation small, to hold only needed long term objects
Performance Problems Due To Garbage Collection
Problems of Garbage Collection

- Application is paused during GC
  - Adds latency
- High GC sequential overhead (serialization) leads to low throughput
  - Limits efficiency and scalability of most server applications
- GC pauses make application behavior non-deterministic
Client Side Problems
Applications like mobile and desktop GUIs (thick and thin client)

- **Requirements**
  - fast response times
  - very low pause time

- **Problems**
  - default collector used most of the time
  - heaps not sized
  - applications run with defaults
  - 1 or 2 CPUs, less memory

- **See**
  - GC frequency increase as heap fills up faster
  - GC takes over application time, leading to performance problems
Server Side Problems (1)

Serving Front end and Back end applications

• **Requirements**
  – Front end applications – application servers, call-processing, web applications
    - fast response time to interact with users
    - low pause time
  – Backend applications – MOMs, Billing, Financial, DSS
    - throughput based applications, no interaction with users
    - tolerate higher pauses, need more application time
  – Lots of CPUs, and memory
Server Side Problems (2)
Serving Front end and Back end applications

• **Problems**
  – Same as client side, default collector used most of the time
  – heaps are sized bigger
  – stop the world collection

• **See**
  – larger pauses as heap sizes are bigger
  – GC frequency dependent on load
  – CPUs are idle during collection
  – scalability problems
Solution
Performance Tuning

• Manual tuning by application modeling
  – What is application modeling
  – GC Portal
  – General Tuning Tips

• Automatic tuning
  – Ergonomics
Application Modeling From A GC Perspective
Application Modeling and Performance Analysis

- What is it?
- Recommendations based on the model
- Empirical modeling
- Theoretical projections
Application Modeling And Performance Tuning: GC Perspective

- Goal: remove the unpredictable behavior of an application
- Construct a mathematical model by mining the verbose GC log files
- The model takes into account
  - Incoming load information
  - Data in verbosegc log files
The Model
Incoming load information

● Transaction Rate (Allocation Rate)
● Active Transaction Duration
  – Lifetimes of short and long lived data
● Size of objects per transaction
The Model (1)
Data in the verbose GC log files

- GC pauses
  - Young and Old generation pauses
  - Time to start-stop application threads
  - Application time

- GC frequency
  - Young and old generation periodicity

- Rate of allocation/promotion of objects

- Direct allocation of objects in old generation
The Model (2)
Data in the verbosegc log files (Contd.)

- **Total**
  - GC time, Application time
  - Objects promoted
  - Garbage collected

- **Heap sizes**
  - Size of Young generation (Eden, Semi-Space)
  - Size of Old generation
  - Initial and Final Size of old generation
  - Size of Permanent generation
  - Average occupancy, and heap thresholds for GC
0.740905: [GC {Heap before GC invocations=8:
Heap
def new generation total 1536K, used 1055K [0xf2c00000, 0xf2e00000, 0xf2e00000]
eden space 1024K, 99% used [0xf2c00000, 0xf2cfef0, 0xf2d00000]
from space 512K, 7% used [0xf2d00000, 0xf2d09c50, 0xf2d80000]
to space 512K, 0% used [0xf2d80000, 0xf2e00000, 0xf2e00000]
concurrent mark-sweep generation total 59392K, used 540K [0xf2e00000, 0xf6800000, 0xf6800000]
concurrent-mark-sweep perm gen total 4096K, used 1158K [0xf6800000, 0xf6c00000, 0xfa800000]}
0.741773: [DefNew Desired survivor size 262144 bytes, new threshold 1 (max 31)
age 1: 280048 bytes, 280048 total
age 2: 40016 bytes, 320064 total
: 1055K->312K(1536K), 0.0048282 secs] 1595K->853K(60928K)

Heap after GC invocations=9:Heap
def new generation total 1536K, used 312K [0xf2c00000, 0xf2e00000, 0xf2e00000]
eden space 1024K, 0% used [0xf2c00000, 0xf2c00000, 0xf2d00000]
from space 512K, 61% used [0xf2d80000, 0xf2dce240, 0xf2e00000]
to space 512K, 0% used [0xf2d00000, 0xf2d80000, 0xf2e00000]
concurrent mark-sweep generation total 59392K, used 540K [0xf2e00000, 0xf6800000, 0xf6800000]
concurrent-mark-sweep perm gen total 4096K, used 1158K [0xf6800000, 0xf6c00000, 0xfa800000]
}, 0.0063803 secs]
Data Calculated

- GC sequential overhead (Directly related to application throughput)
- GC concurrent overhead
- Average size of objects
- Active data duration (long and short term objects)
- Actual throughput
- Application efficiency
- Speedup (Amdahl’s law)
- % CPU utilization
- Memory Leak detection
General Recommendations based on the model (1)

- General JVM Tuning and Sizing methodology
  - \textit{Size of old generation} = \textit{Call rate} \times \textit{active call duration} \times \textit{long lived data/call}
  - \textit{Size of young generation} = \textit{Call rate} \times \textit{expected periodicity of GC} \times \textit{short lived data/call}
    - for desired pause and frequency

- Reduce GC pauses
- Reduce GC sequential overhead
General Recommendations Based On The Model (2)

- Size the young and old generation heaps to handle a given load
- Detect memory leaks
- Choice of collector
- Choice of the different JVM options and switches
Empirical Modeling

- Rank the Application runs based on data analyzed from the verbosegc logs
- Choose the optimum JVM environment based on criteria:
  - Heap sizes
  - No. of Processors
  - GC sequential overhead
  - GC concurrent overhead, etc.
  - Application efficiency
Theoretical Projections For Tuning Based On The Model

- "What-if" scenarios could be tried
  - How GC behavior changes with change in Application and JVM parameters

- "What-if" input parameters include:
  - Size of young generation
  - Size of old generation
  - Request rate/Load
  - Garbage/request
  - No. of processors
Theoretical Projections For Tuning

• Projection output shows:
  – what could be the
    • GC pause (latency)
    • GC frequency
    • GC sequential load (bandwidth)
    • % CPU utilization, Speedup
    • Application efficiency
    • Allocation rate, Promotion rate
    • Size and duration of Short lived data
    • Size and duration of Long lived data
GC Portal
GC Portal

- Enables as a service, Application Modeling and Performance Tuning from GC perspective.
- Implemented in J2EE
- Allows developers to submit log files, and analyze application behavior
- Portal can be used to performance tune, and size application to run optimally under lean, peak, and burst conditions
SnapShot from the GC Portal

GC Information Average Summary

Filename: gc.sum

Number of processors: 4

Load or Call Rate: 50 CPS

Actual Throughput: 44.59057 CPS

Short Term Data: 67 Kb

Long Term Data: 178 Kb

CPU Utilization Efficiency: 75.5287 %

Other GC Details:

<table>
<thead>
<tr>
<th>gcSeqLd</th>
<th>gcConcl</th>
<th>CRLd</th>
<th>totLd</th>
<th>gen[0] GC count</th>
<th>gen[0] GC Freq.</th>
<th>gen[0] GC Total Time</th>
<th>Avg. Pause</th>
<th>Max. Pause</th>
<th>Promotion Rate</th>
<th>Allocation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.8</td>
<td>5.4</td>
<td>0.0</td>
<td>16.2</td>
<td>1583</td>
<td>1.02 /sec</td>
<td>156800 ms</td>
<td>99 ms</td>
<td>313</td>
<td>3.371 mb/s</td>
<td>12.275 mb/s</td>
</tr>
<tr>
<td>Total Time</td>
<td>gcSeq ms time</td>
<td>gcConcl ms time</td>
<td>Application ms. time</td>
<td>kb allocated</td>
<td>Kb promoted</td>
<td>Kb collected</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1547510</td>
<td>167423</td>
<td>331691</td>
<td>1380087</td>
<td>1899606</td>
<td>5216550</td>
<td>44058506</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recommendations for Modeling and Performance Tuning

Detailed Analyzed Data
GC Portal

• Plots and displays graphically GC behavior over time. Parameters include:
  – GC pauses (Max. and Average)
  – GC frequency
  – GC sequential load
  – GC concurrent load
  – Garbage Allocation rate
  – Garbage Promotion rate
Snapshot from the GC portal
Graphical Engine

Variation of GC sequential load, GC[0] pauses, Successful throughput and response retries ratio with Young Generation size

- Sequential GC overhead
- GC[0] pauses
- Successful throughput
- Percentage of retries per successful transaction

Size of young generation

36 | JA-SIG 2004
GC Portal

- Provides General Recommendations
- Projections for sizing and tuning via "what-if" scenarios
- Empirical modeling
### Snapshot from the GC portal
What-if scenarios

<table>
<thead>
<tr>
<th>Parameter</th>
<th>New Projected Value</th>
<th>Old Value from Logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Young Generation Size (MB)</td>
<td>6.04</td>
<td>4.03</td>
</tr>
<tr>
<td>Young GC pause (ms)</td>
<td>3.81</td>
<td>8.79</td>
</tr>
<tr>
<td>Total No of Promotions GCs</td>
<td>303</td>
<td>454</td>
</tr>
<tr>
<td>% GC Sequential Load</td>
<td>1.64</td>
<td>5.76</td>
</tr>
<tr>
<td>Throughput/Sec</td>
<td>49.18</td>
<td>47.12</td>
</tr>
<tr>
<td>% CPU Utilization</td>
<td>95.30</td>
<td>85.26</td>
</tr>
<tr>
<td>% Throughput Efficiency</td>
<td>98.36</td>
<td>94.24</td>
</tr>
<tr>
<td>% Degradation</td>
<td>1.67</td>
<td>5.79</td>
</tr>
<tr>
<td>Scalability</td>
<td>3.81</td>
<td>Not Available</td>
</tr>
<tr>
<td>Long Term Data / Call (MB)</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Short Term Data / Call (MB)</td>
<td>8.33</td>
<td>8.16</td>
</tr>
<tr>
<td>Calls processed / GC</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Pause / Call (ms)</td>
<td>0.33</td>
<td>1.16</td>
</tr>
<tr>
<td>Periodicity of Promotions (ms)</td>
<td>227.74</td>
<td>151.83</td>
</tr>
<tr>
<td>Application Run Time (ms)</td>
<td>70171.41</td>
<td>73144.00</td>
</tr>
</tbody>
</table>
GC Portal

• GC portal can be downloaded and installed

• It can also be accessed on the internet
  – http://sdc.sun.com/gcportal/
GC General Tuning Tips
Reducing Collection Times

- Use `-Xcongcgc` for low pause applications
- Use `-XX:+AggressiveHeap` for throughput applications
  - Use `-XX:+PrintCommandLine` to see AggressiveOptions, and use this to tune further

- Size Permanent Generation
- Reduce pooled objects
- Using NIO
- Avoid `System.gc()` and distributed RMI GC
  - Use `-XX:+DisableExplicitGC`

- Making immutables, mutables
  - String -> String Buffer for String manipulation, and maybe storage

- Avoiding old generation undersized heaps
  - Reduces collection time, but leads to lot of other problems like fragmentation, triggers Full GC
Reducing Frequency Of GC

- Frequency of a collection is dependent on
  - Size of young and old generations
  - Incoming load
  - Object life time

- Increase young generation to decrease frequency of collection but this will increase pause
  - Choose a size where pause is tolerable

- Increase in load will fill up the heap faster so increases collection frequency
  - Increase heap to reduce frequency

- Increase in lifetime of objects increases frequency as live objects take space
  - Keep live objects to the needed minimum
Sizing The Heap

• **Heap size influences the following**
  – GC frequency and collection times
  – Number of short and long term objects
  – Fragmentation and locality problems

• **Undersized heap with concurrent collector**
  – leads to Full GCs with increase in load
  – Fragmentation problems

• **Oversized heap**
  – leads to increased collection times
  – locality problems (smear problem)
  – Use ISM and variable page sizes to reduce smear problem

• **Size heap to handle peak and burst loads**
Improving Execution Efficiency

- GC Portal computes execution efficiency
- Efficiency calculated using Amdahl's law
- Translates to CPU utilization
- Higher this value the better
- Increase efficiency by reducing serial parts
  - Reducing GC pause & frequency
  - Reducing long term objects and increasing short term objects
  - Creating only needed objects like using NIO, mutables
  - Avoiding Full GC, For e.g. RMI DGC, undersized heaps
  - Choosing optimum heap size to reduce smear effect
Other Ways To Improve Performance On Solaris

- Using the Solaris RT (real-time) scheduling class
- Using the alternate thread library (/usr/lib/lwp)
  - default thread library on Solaris 9
- Using hires_tick to change clock resolution
- Using processor sets
- Binding process to a CPU
- Turning off interrupts
- Modifying the dispatch table
- Use large page sizes
- Use multi-threaded malloc library
Automatic Tuning In J2SE 1.5
Ergonomics

- What is ergonomics?
- Why do it?
- When is it used?
- What does it do?
- How does it work?
What Is Ergonomics?

- JVM™ automatically selects
  - Compiler
  - Garbage collector
  - Heap size

- User specifies behavior

- GC dynamically does tuning
  - AKA GC ergonomics
Why Do Ergonomics?

- Better Performance
  - Hand tuned performance is good

- Ease of Use
  - Hand tuning is hard

- Better Resource Usage
  - Use what you need
When Is Ergonomics Used?

- **Server class machines**
  - 2 CPUs, 2 Gbytes
- **Exceptions**
  - Microsoft Windows ia32
What Does Ergonomics Do?

- Server compiler
- Parallel GC collector
- Maximum heap
  - Smaller of
    - $\frac{1}{4}$ physical memory
    - 1 Gbyte
- Initial heap
  - Smaller of
    - $\frac{1}{64}$ physical memory
    - 1 Gbyte
What is GC Ergonomics?

- **User specifies**
  - Maximum pause time goal
  - Throughput goal
  - Assumes minimum footprint goal

- **GC tunes**
  - Young generation size
  - Old generation size
  - Survivor space sizes
  - Tenuring threshold
Why Do GC Ergonomics?

- Common complaints
  - Pauses are too long
  - GC is too frequent

- Solution
  - Hand tune the GC
What Does GC Ergonomics Do?

- **Goals, not guarantees**
- **User specified behavior**
  - Maximum pause time goal
    - Reduce size of generation
  - Throughput goal
    - Increase size of generations
  - Minimum footprint
    - Reduce size of generations
- **Again, goals, not guarantees**
Throughput Example

Throughput Only

Old Generation Size vs. Time (sec)
Throughput Example

Throughput Only

Time (sec)

Major Pause (ms)
Throughput Example

Throughput Only

Major GC cost (%)

Time (sec)
Pause Example

Pause Goal - 1250ms
Ergonomics Usage

• Use `-XX:+UseParallelGC` with the below options

• Throughput Goal
  – `-XX:GCTimeRatio=nnn`
    • The ratio of GC time to application time
    • \(1 / (1 + nnn)\) where nnn is a value to obtain the percentage GC time vs application time. E.g. Nnn = 19, GC time 5% of application time

• Pause Time Goal
  – `-XX:MaxGCPauseMillis=nnn`
    • An hint to the JVM to keep the pauses below this value
Ergonomics Strategy

- Use throughput strategy, and set desired throughput
- Change maximum heap size if throughput cannot be achieved
- If throughput goal is achieved, set pause time goal, if pauses are high
JVM Monitoring & Management in J2SE 1.5
Java Monitoring and Management API

• Provides a way to manage and monitor a JVM
  – Information about loaded classes and threads
  – Memory usage
  – Garbage collection statistics
  – Low memory detection & thresholds

• Provides monitoring utilities
  – jconsole
  – jstat
Java Monitoring and Management API

- Provides MBeans
  - GarbageCollectorMXBean
  - MemoryManagerMXBean
  - MemoryMXBean
  - MemoryPoolMXBean
  - Other MBeans

- MBeans can be accessed through
  - jconsole
    - jconsole jvmpid
jstat

• An utility to obtain JVM statistics dynamically
  – Compiler statistics
  – Class loader statistics
  – GC statistics

• GC statistics include
  – cause of GC
  – generation information
    • capacity
    • utilization
jstat Usage

- **jstat -gc jvmid**
  - Provides statistics on the behavior of the garbage collected heap

- **jstat -gcutil jvmid**
  - Provides a concise summary of garbage collection statistics.

- **jstat -gcutil 21891**
  - S0  S1  E  O  P  YGC  YGCT  FGC  FGCT  GCT
    - 12.44  0.00  27.20  9.49  96.70  78  0.176  5  0.495  0.672
    - 12.44  0.00  62.16  9.49  96.70  78  0.176  5  0.495  0.672
    - 12.44  0.00  83.97  9.49  96.70  78  0.176  5  0.495  0.672
    - 0.00  7.74  0.00  9.51  96.70  79  0.177  5  0.495  0.673
Summary

- Introduced low pause and throughput collectors
- Performance problems seen with garbage collection
- Improving performance using manual and automatic tuning
- Introduction to the new monitoring & management API
Resources

- http://java.sun.com/docs/hotspot/gc1.4.2/
- http://java.sun.com/developer/technicalArticles/Programming/turbo/
- http://sdc.sun.com/gcportal/

Contact:

Nagendra.Nagarajayya@Sun.Com
gc-portal-team@sun.com
Java Performance Tuning
From A Garbage Collection Perspective

Nagendra Nagarajayya
nagendra.nagarajayya@sun.com