Explorations of the Three Legged Performance Stool

Charlie Hunt
JVM & Performance Junkie

CREATE THE FUTURE

ORACLE®

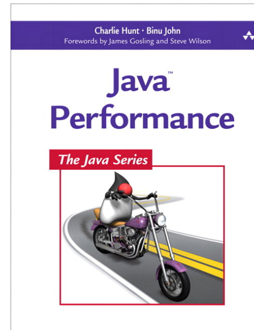
The following is intended to or may outline our [Oracle] general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Who is this guy?

Charlie Hunt

- Currently leading a several HotSpot JVM projects at Oracle
- Held various performance architect roles at Oracle, Salesforce.com & Sun Microsystems
- Lead author of ***Java Performance***, published Sept 2011



 | Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java ORACLE

Who is this guy?

And for those who enjoy reading in additional languages …

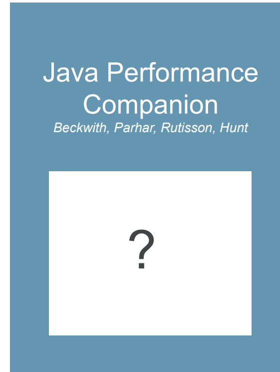
 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Who is this guy?

And coming soon to a book store near you …

Java Performance Companion – book cover in process

Authors: Monica Beckwith, Bengt Rutisson, Poonam Parhar & Charlie Hunt



Intended to compliment the material found ***Java Performance***

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

What to expect

This session is about understanding the relationship between throughput, latency and footprint along with, understanding where system capacity fits in.

And, reasoning about the benefit & impact of each when making changes, JVM configuration or application.

We will also look at a case study / example.

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Agenda

- The key performance attributes
- (Very) quick re-visit of the HotSpot JVM GCs
- Reason about trade-offs with the performance Attributes
- Case Study (a JDK 9 feature)

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

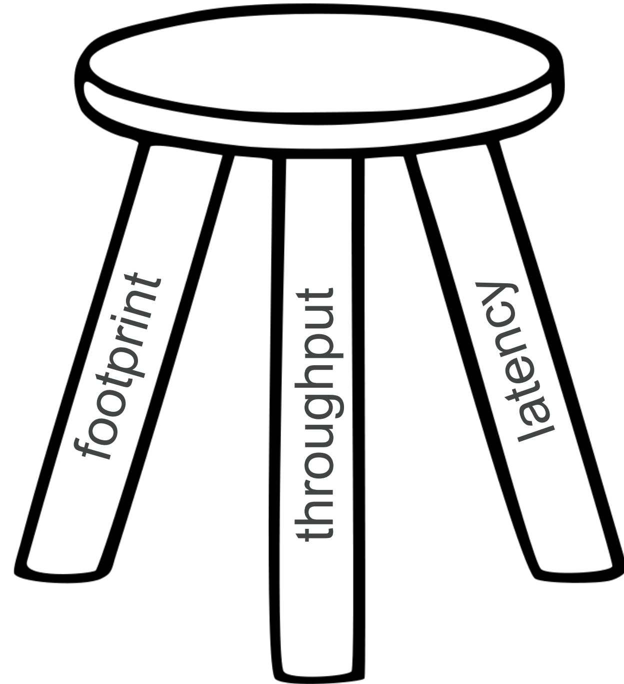
The Performance Attributes

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Three Legged Stool

- Throughput
- Latency
- (Memory) Footprint



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE

2 of 3 Principle

Improving one or two of these performance attributes, (throughput, latency or footprint) results in sacrificing some performance in the other.

Hunt, John. Java Performance. Upper Saddle River, NJ, Addison-Wesley, 2011

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

2 of 3 Principle (updated)

Improving all three performance attributes [usually] requires a lot of non-trivial [development] work.

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

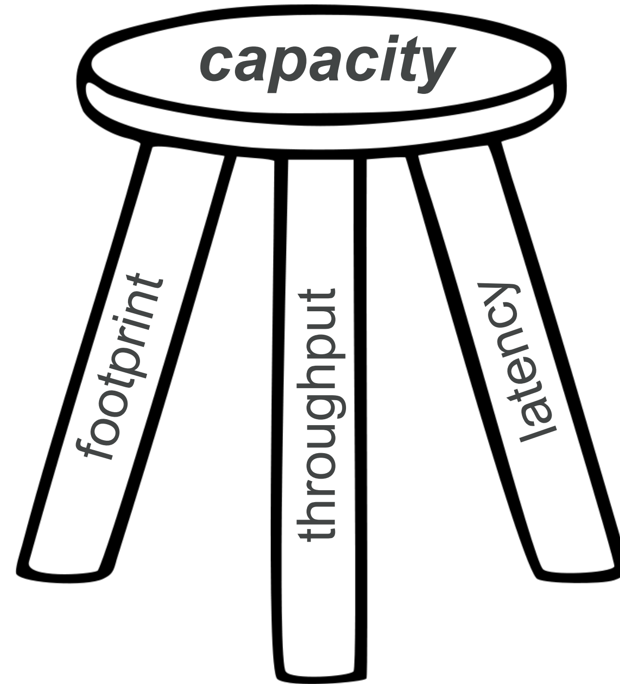
Another Principle - (yet to be named)

An improvement in throughput and/or latency may reduce or lower the amount of available CPU to the application, or other applications executing on the same system. Thus impacting the capacity of the system.

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Perhaps an Enhanced Three Legged Stool ?

- Throughput
- Latency
- (Memory) Footprint
- ***Capacity***



 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Quick (re)visit of HotSpot JVM GCs

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Generational GCs

- [Almost] all modern JVMs use a generational GC
  - Segregate objects by age into different spaces, and bias collection of younger objects
  - Typically two generations; young & old
- JVMs with generational GCs
  - HotSpot – all GCs supported by Oracle
  - Zing – C4 GC
  - J9 – all AFAIK … admit I'm not familiar with J9's GCs

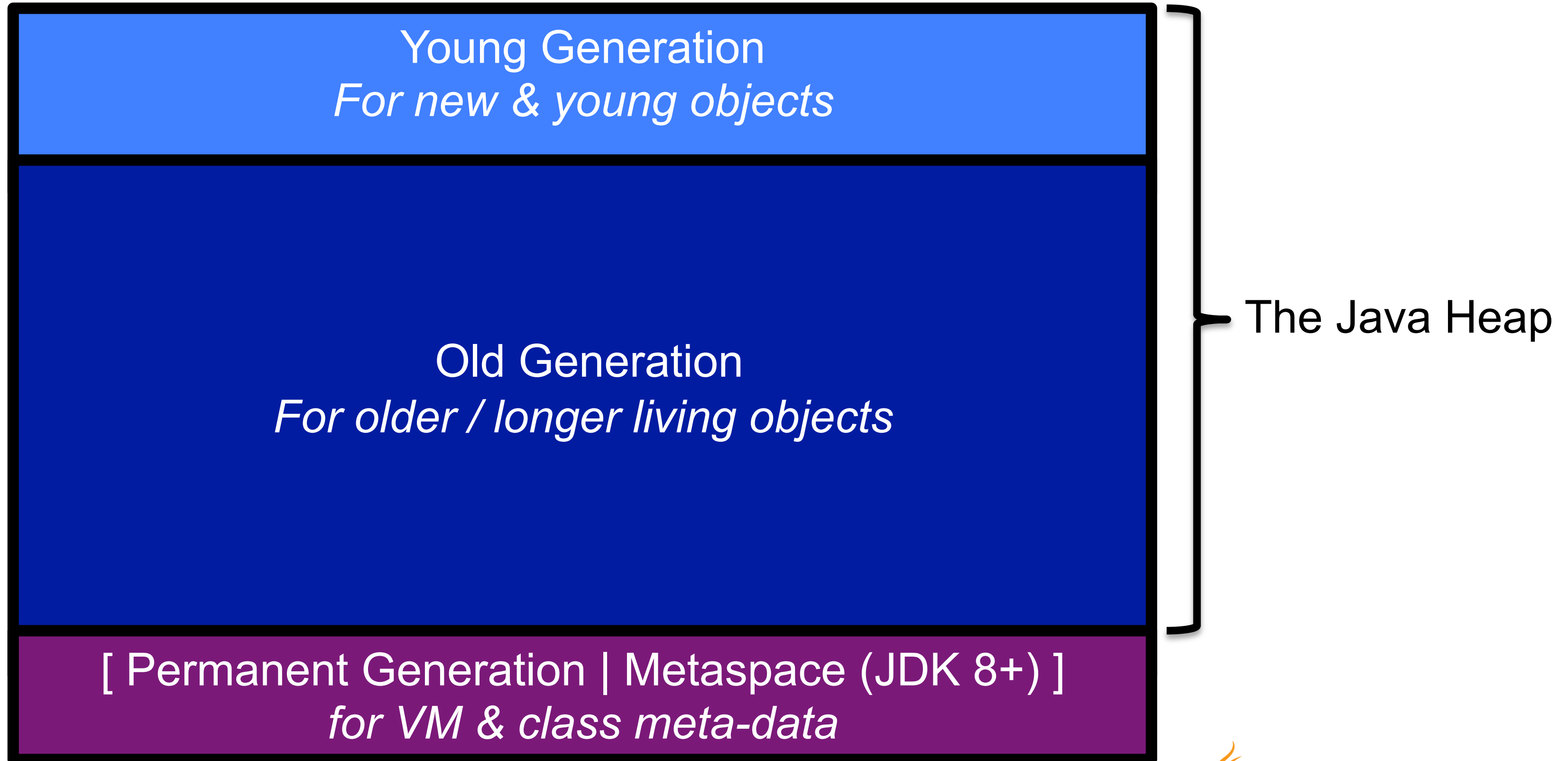
Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Why Generational GC ?

- Weak generational hypothesis
  - Most objects die young
- Generally accepted reason for generational GCs
  - Improved throughput, and scaling to large Java heaps

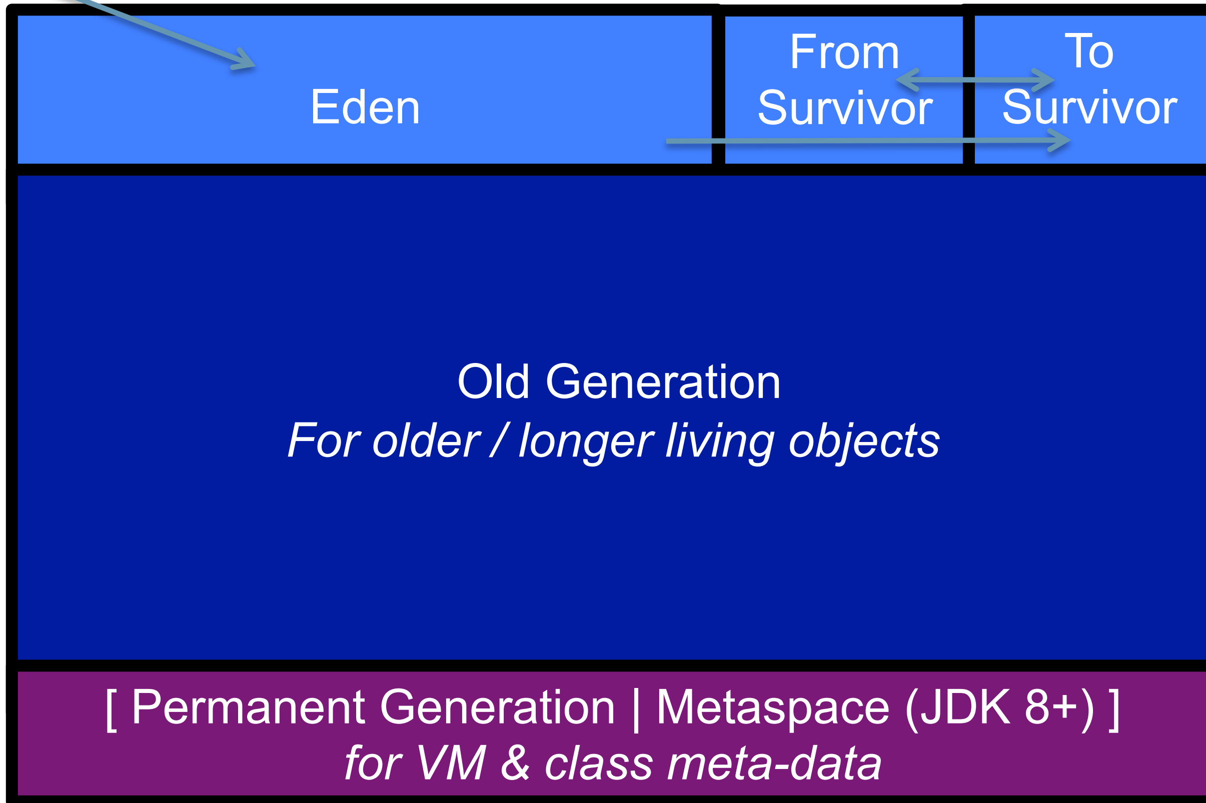
Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

HotSpot JVM Java Heap Layout



 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

HotSpot JVM Java Heap Layout



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Things to reason about wrt GC & your application

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE

GC Duration (pause time)

What impacts pause time duration?

- Duration of GC activity is mostly a function of the number of live objects
  - Other factors
    - Object graph structure
    - Use of, and number of Reference objects
    - Memory locality

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

GC Duration (pause time)

Strategies to reduce (young) GC duration

- Reduce young generation Eden size
  - Smaller space implies fewer live objects (may not always be true!)

Eden (1 GB)

- Concurrent young collector?
  - Eliminate pause by collecting concurrently

 | Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

GC Duration (pause time)

Consequences

- Reduce young generation Eden size
  - Likely lower throughput due to more frequent application pauses
  - More frequent GCs due to smaller size (more on that in a moment)
  - More object promotions to old generation due more rapid object aging
- Concurrent young collector
  - Available capacity impact due to higher CPU utilization from concurrent collection activity
  - Throughput and/or latency may be impacted due to CPU usage and CPU cache eviction
  - Footprint may be impacted due to floating garbage

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

GC Duration (pause time)

Other alternatives

- Lower application's object allocation rate
  - Lower injection rate or load on the application
    - Lower system throughput, & lower system capacity, maybe better latency
  - Capture memory profiles and reduce object allocations
    - Requires (development) work, and may be non-trivial
    - Might actually increase number of live objects at each young GC
      - Implies a potential increase in pause time
    - Throughput and latency may (or should) improve due to less frequent GC events

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Young GC Frequency

Often not thought of … and applicable to concurrent collection too

- How often Young GC's occur influenced by
  - Application object allocation rate – how fast it's allocating objects
  - Size of Eden space
    - This applies to the HotSpot JVM
    - For other generational JVMs it will be the size of the space where new objects are allocated
    - For STW collectors, once that space is exhausted, a GC event occurs
    - For concurrent collectors, there's usually an occupancy threshold that once it's surpassed, a concurrent GC event commences

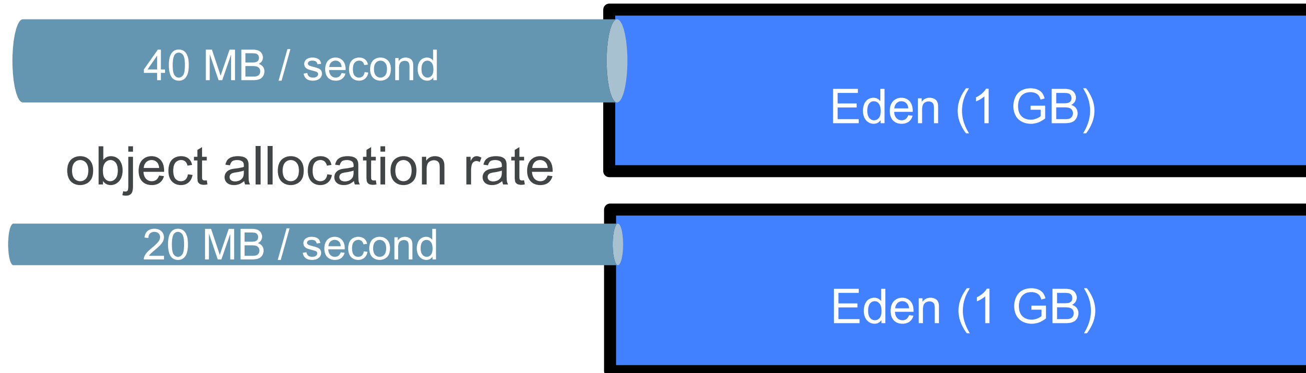
 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Influencing Minor GC Frequency

Reduce object allocation rate

- Eden space fills more slowly with reduced object allocation rate



- Obvious, right? 2x drop in allocation rate, time to fill Eden space increases 2x, i.e. minor GC frequency cut in ½

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Influencing Minor GC Frequency

Make Eden size bigger (assuming same object allocation rate)

- Same allocation rate, 2x increase in Eden space, time between GC increases 2x, i.e. minor GC frequency cut in ½

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Young GC Frequency

Consequences

- Reduce object allocation rate (saw this before, right?)
  - Lower injection rate or load on the application
    - Lower system throughput, lower system capacity, maybe better latency
  - Capture memory profiles and reduce object allocations
    - Requires (development) work, and may be non-trivial
    - Might actually increase number of live objects
      - Implies a potential increase in pause time, or GC time
    - Throughput and latency may (or should) improve

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Young GC Frequency

Consequences

- Increase young generation Eden size
  - Less frequent GCs due to larger size
  - Fewer object promotions due more effective object aging
  - Likely higher throughput due to less frequent application pauses or collection activity
    - All 3 probably sound attractive
  - Worse case latency may be higher due to larger space to GC
    - GC duration / pause time could be longer
    - Hmmm … may be not so attractive?

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Young GC Frequency

Consequences

- What about the potential impact with a concurrent young generation collector ?
  - Less impact on available capacity due to less frequent collection activity
  - May see higher throughput and/or lower latency due to lower CPU usage and lower CPU cache eviction due to less frequent concurrent collection activity
  - Larger footprint due to larger young generation size

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Old GC Duration

What influences Old GC duration?

- Number of live objects and objects to move for compaction
  - Other factors
    - Object graph structure
    - Use of, and number of Reference objects
    - Memory locality
- Differing GC algorithms; stop the world, mostly concurrent, concurrent, mark sweep, mark sweep compact, etc.
  - And, details within the GC algorithm, i.e. mostly concurrent write-barrier implementation, concurrent read & write barrier implementations

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Old GC Duration

Strategies to reduce (old) GC duration (pause time)

- Reduce old generation size
  - Smaller space implies fewer live objects (may not always be true!)

Old (2 GB)

- Use a concurrent, or mostly concurrent old collector?
  - Eliminate, or reduce lengthy pause(s) by collecting concurrently or mostly concurrently

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Old GC Duration (pause time)

Consequences

- Reduce old generation size
  - More frequent GCs due to smaller size, i.e. space fills faster
  - Throughput and/or latency may be impacted due to collections occurring more frequently
  - Available capacity impact due to higher CPU utilization from more frequent collection activity
  - Likely a smaller footprint due to smaller old generation size

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Old GC Duration (pause time)

Consequences of reduced old gen size

- What about using a concurrent, or mostly concurrent old collector
  - Available capacity impact due to higher CPU utilization from concurrent collection activity
  - Throughput and/or latency may be impacted due to CPU cache eviction from concurrent GC activity
  - Footprint may be impacted due to floating garbage

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Old GC Frequency

Applicable to mostly concurrent collectors too

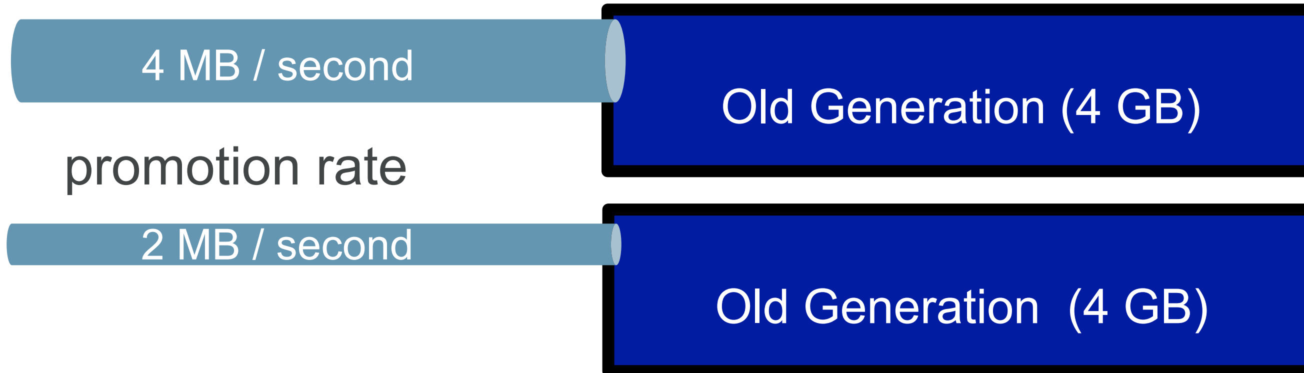
- How often Old GC's occur influenced by
  - Object promotion rate – how many objects promoted from young to old
  - Size of the Old space
    - This applies to the HotSpot JVM
    - For other generational JVMs it will be the size of the space where older objects are promoted / allocated
- For STW collectors, once the space is exhausted, a GC event occurs
- For (mostly) concurrent collectors, there's usually an occupancy threshold that once it's surpassed, a concurrent GC event commences

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE

Influencing Old GC Frequency

Reduce promotion rate

- Old space fills more slowly with reduced object allocation rate



- Obvious, right? 2x drop in promotion rate, time to fill Old Generation space increases 2x, i.e. old GC frequency cut in ½

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Influencing Old GC Frequency

Make Old Generation size bigger (assuming same promotion rate)

Old Generation (4 GB)

Old Generation (8 GB)

- Same promotion rate, 2x increase in Old Generation space, time between GC increases 2x, i.e. Old GC frequency cut in ½

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Old GC Frequency

Consequences

- Reduce promotion rate
  - Lower injection rate or load on the application
    - Lower system throughput & system capacity, maybe better latency
  - Capture memory profiles and reduce promotions
    - Focus on object retention as much as, or more than object allocations (Look at java.util.LinkedList element removal)
    - Requires (development) work, and may be non-trivial
    - Throughput and latency may (or should) improve

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Old GC Frequency

Consequences

- Increase old generation size
  - Less frequent old GCs due to larger size
  - Likely higher throughput, and possibly lower latency due to less frequent application pauses or collection activity
    - All 3 are probably attractive
  - Worse case latency likely impacted for non-concurrent, or non mostly concurrent collectors
    - Hmmm … may be not so attractive?

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Old GC Frequency

Consequences of larger old gen space

- What about the potential impact with a concurrent, or mostly concurrent old generation collector ?
  - Less impact on available capacity due to less frequent collection activity
  - May see higher throughput and/or lower latency due to lower CPU cache eviction due to less frequent concurrent GC activity
  - Larger footprint due to larger old generation size

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

Case Study: String Density

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE

String Density

JEP 254: Compact Strings

- JDK 9 feature
- More space-efficient internal representation for java.lang.String
- Terminology
  - String Density == Project name
  - Compact Strings == Feature name
- Goals
  - Lower memory footprint, yet no regression in throughput
    - Implied are no latency or CPU usage regressions

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

What about the 3 legged stool ?!?!

Lots of work!

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

What about the effort?

- 10 contributing engineers
  - 8 contributing from Oracle
  - 2 contributing from Intel
- Each spending about 1 year, ½ time on String Density
  - About 5 man years of effort

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

String Density

Additional requirements

- Improve space efficiency for String and related classes
- Preserve throughput and latency
- Preserve compatibility
  - Java supports UTF-16, yet many apps only use lower byte in Strings
  - No new Java SE APIs, no changes required for upstream applications
- Replacement for JDK 6’s Compressed Strings
- Platforms: X86/X64, SPARC, ARM 32/64
- OS: Linux, Solaris, Windows and Mac OS X

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

String Density

Analysis approach

- Repository of 950+ heap dumps from various Oracle FMW, Fusion Apps, and Java applications
  - How much can memory footprint be reduced?
- Java Object Layout tools
  - If fields added or removed from String class, what's the footprint impact per String instance?
- JVM model's analyzed; 32-bit, 64-bit w/ no compressed oops, 64-bit with compressed oops (8 byte aligned), and 64-bit with compressed oops (16 byte aligned)

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

What we learned

- Distribution of the live data size similar across the models
  - Roughly 300 MB to 2.5 GB with a long tail
- char[]'s consume about 10% - 45% of the live data size
- Most Strings contain single byte chars
- 75% of Strings are 35 chars or smaller
  - Long tail distribution as the String sizes get larger beyond 35 chars
- 35% to 40% reduction in char[] footprint, not 50% theoretical reduction
- 5% - 15% reduction in application footprint

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE

String Density

Proposed solution

- Store chars in String as either UTF-16, or ISO-8859-1/Latin1
  - Stripping off leading zero byte of two byte UTF-16 char
- Use byte[] instead of char[] to store String's characters
  - 1 byte per char for ISO-8850-1/Latin1
  - 2 bytes per char for UTF-16
- Add an encoding byte field to indicate the encoding in use
  - Ability to extend to support additional character encodings

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java ORACLE

String Density

Proposed solution continued …

- Strings with all leading byte bytes as 0
  - Candidate for ISO-8850-1/Latin1 encoding
  - Leading 0 bytes stripped off, and trailing bytes stored, i.e. single byte per char
- String with any leading byte in incoming char as non 0
  - Cannot be encoded as ISO-8859-1/Latin1, stored as UTF-16 encoded, i.e. two bytes per char
  - Compress (deflate) incoming characters, inflate to UTF-16 when returning sequence of chars via String API(s)

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

String Density

Proposed solution continued …

- Why not UTF-8?

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

String Density

Proposed solution continued …

- Why not UTF-8?
  - Cause we're stupid!

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

String Density

Proposed solution continued …

- Why not UTF-8?
  - Cause we’re stupid! Just joking of course!!!
  - UTF-8 supports variable width characters
  - Many String operations require random access into sequence of chars
    - Their throughput performance would suffer!
  - UTF-8 encoding is great for character transmission
    - It’s not performant for String operations

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

JDK 8 String class versus new JDK 9 String class

Old String Class (JDK 8)

```
{
    private final char value[];
    private int hash;
    ...
}
```

New String Class (JDK 9)

```
{
    private final byte[] value;
    private final byte coder;
    private int hash;
    ...
}
```


Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

What about memory footprint per String instance?

```
***** 32-bit VM: *****************************************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0     8        (object header)                N/A
      8     4 char[] String.value                   N/A
     12     4    int String.hash                    N/A
Instance size: 16 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

***** 64-bit VM: *****************************************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    16        (object header)                N/A
     16     8 char[] String.value                   N/A
     24     4    int String.hash                    N/A
     28     4        (loss due to the next object alignment)
Instance size: 32 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```


Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java ORACLE

String Density

What about memory footprint per String instance?

```
***** 64-bit VM, compressed references enabled: ***************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    12        (object header)                N/A
     12     4 char[] String.value                   N/A
     16     4    int String.hash                    N/A
     20     4        (loss due to the next object alignment)
Instance size: 24 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

***** 64-bit VM, compressed references enabled, 16-byte align: ************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    12        (object header)                N/A
     12     4 char[] String.value                   N/A
     16     4    int String.hash                    N/A
     20    12        (loss due to the next object alignment)
Instance size: 32 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 12 bytes external = 12 bytes total
```

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java ORACLE

String Density

What about throughput?

- Developed **400** JMH micro-benchmarks for String APIs
  - Results
    - http://cr.openjdk.java.net/~thartmann/compact_strings/microbenchmarks
  - Micro-benchmarks (and various other String Density artifacts)
    - http://cr.openjdk.java.net/~shade/density/
- String is highly optimized using SIMD instructions
  - About 55 specific JIT compiler optimizations for String related operations
    - Non-trivial amount of work
    - Why not fly your own single byte String class?

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

Macro-level performance

- SPECjbb2015 (Intel x64)
  - Live data size reduction of about 7% *(footprint reduction)*
  - Critical-jOps increased by about 11% *(latency maintained or improved)*
    - At critical-jOps, CPU utilization remained the same as baseline
  - Max-jOps increased by about 3% *(throughput improved)*
- SPECjbb2005 (Intel x64)
  - Live data size reduction of about 21%
  - Throughput increase of about 5%
  - 23% reduction in GC frequency

*SPECjbb2015 & SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. See http://www.spec.org for more information.

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Java™ ORACLE®

String Density

Three legged stool revisited

- With a lot of effort ….
- We realized a reduction in memory footprint, yet able to maintain or improve throughput, and maintain latency or reduce latency
- And, able to maintain or improve system capacity

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Call to Action

Interested or Intrigued?

- Can’t wait to try it on your application(s) ?
  - Again, a JDK 9 feature
- Implementation:
  - Repository: http://hg.openjdk.java.net/jdk9/sandbox/
  - Branch: JDK-8054307-branch

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Call to Action

Interested or Intrigued?

- Build Steps:

```
$ hg clone http://hg.openjdk.java.net/jdk9/sandbox/
$ cd sandbox
$ sh ./get_source.sh
$ sh ./common/bin/hgforest.sh up -r JDK-8054307-branch
$ make configure
$ make images
```

- Command line option to enable/disable feature:

  -XX:+CompactStrings / -XX:-CompactStrings

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
Java™ ORACLE®

Main Takeaways

Raising all three legs of the performance stool is hard, but possible with a lot of non-trivial effort.

Reason about the tradeoffs in realizing improvements in one or two of the performance attributes, i.e. understand the alternatives, and realize system capacity may be a criteria that fits in here too.

 Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Acknowledgements

- String Density Team
  - Oracle
    - Aleksey Shipilev, Sherman Shen, Brent Christian, Roger Riggs, Tobias Hartmann, Vladimir Kozlov, Guy Delemarter
  - Intel
    - Sandhya Viswanathan, Vivek Deshpande
    - Special thanks to Sandhya – joint JavaOne 2015 presentation
- HotSpot GC Engineering Team
  - Past and present members

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

ORACLE®

CREATE THE FUTURE

Java™