Fundamentals of GC Tuning
Charlie Hunt
JVM & Performance Junkie
Who is this guy?

Charlie Hunt

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

This is not a session about JVM command line options, i.e. -Xms, -Xmx, -Xmn, -XX:InsertYourFavorite

It’s about understanding the basic concepts to enable you to reason about tuning a JVM’s GC, and the advantages, consequences, tradeoffs of those decisions.
The Goal

Ideally, understanding the basic concepts will enable you to tune any garbage collector.

At a minimum, make better decisions about what actions to consider taking.
Agenda

- The Key Performance Attributes
- High Level Look at Modern JVM GC Architectures
- Basic Concepts
The Performance Attributes
Three Legged Stool

- Throughput
- Latency
- (Memory) Footprint
2 of 3 Principle

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

Improving all three performance attributes [usually] requires a lot of non-trivial [development] work.
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.
Perhaps an Enhanced Three Legged Stool?

- Throughput
- Latency
- (Memory) Footprint
- Capacity
High Level Look at Modern JVM GC Architectures
Generational GC

- [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
  - JRockit – all, except JRockit Real-Time
  - Zing – C4 GC
  - J9 – all AFAIK … admit I’m not familiar with J9’s GCs
Why Generational GC?

- Weak generational hypothesis
  - Most objects die young
- #1 reason for generational GCs
  - Improved throughput
HotSpot JVM Java Heap Layout

**Young Generation**
*For new & young objects*

**Old Generation**
*For older / longer living objects*

[Permanent Generation | Metaspace (JDK 8+)]
*for VM & class meta-data*
To Survivor

From Survivor

Survivor

Old Generation

For older / longer living objects

New object allocations

Retention / aging of young objects during young / minor GCs

The Java Heap

[ Permanent Generation | Metaspace (JDK 8+) ]

for VM & class meta-data
When tuning GC

General Approach

- Think in terms of
  - Frequency
    - How frequent can I live with GC events occurring?
  - Duration
    - For STW GC, how long of a GC pause can I tolerate?
    - For concurrent GC, how much capacity (CPU consumption), throughput and additional memory am I willing to sacrifice?

- Note: answers to these will differ depending on the application
  - And, they can also differ between application stakeholders
First Important Concept
Important Concept #1

Frequency of Minor / Young GCs

- Frequency of a minor GC event is dictated 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.
Important Concept #1

Changing the frequency of a minor GC event

- Less frequent GC
  - Make Eden size bigger (assuming same object allocation rate)
    - Eden (512 MB)
    - Eden (1024 MB)

- Same allocation rate, 2x increase in Eden space, time between GC increases 2x, i.e. minor GC frequency cut in \( \frac{1}{2} \)
Important Concept #1

Choices for making Eden size larger

- HotSpot JVM’s CMS GC, Parallel GC and Serial GC
  - Increase -Xmn, -XX:NewSize / -XX:MaxNewSize
  - -XX:SurvivorRatio if you want preserve young gen size but change Eden and Survivor sizes
- HotSpot JVM’s G1 GC
  - Increase -XX:MaxGCPauseMillis
    - G1’s heuristics generally assume greater pause time implies larger Eden space
Important Concept #1

How to change the frequency of a minor GC event

- 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. time between GC increases 2x
Important Concept #1

Strategies for reducing object allocation rate

- Profile application and reduce object allocations
  - Focus on unnecessary object allocations!
- Throttle application
  - Reduce the injection rate or load on the application
Second Important Concept

1. Use existing picture box, DO NOT delete and create new picture box.
2. Right click on the shape.
3. At the bottom of the submenu select "Format Shape".
4. Select "Fill" at the top of the "Format Shape" dialog box.
5. Select "Picture or Texture fill" from the options.
6. And select "File" under the "Insert from" option.
7. Navigate to the file you want to use and select "Insert".
8. On the "Format" tab, in the Size group, click on "Crop to Fill" in the Crop tool and drag the image bounding box to the desired size.

DELETE THIS INSTRUCTION NOTE WHEN NOT IN USE
Important Concept #2

Minor GC Duration (pause time)

- Number of live objects has strongest influence of minor GC duration
  - GC only visit (i.e. marks) and evacuates live objects
- There are other things that can influence minor GC duration
  - Overuse (or abuse) of Reference objects; Weak, Soft, Phantom and Final
  - Memory locality of the marked objects, and evacuation location
  - Young generation object kept live by an old generation reference
  - There can be other reasons
- In short, ideally minor GC duration should be dominated by marking and evacuation time
Important Concept #2

Choices for reducing minor GC pause time

- Faster memory and CPU (in general a faster system)
- Faster STW GC algorithm
- Concurrent GC
  - May introduce more CPU consumption which can reduce available capacity
  - May require more memory (footprint) to handle floating garbage
- Fewer live objects
Important Concept #2

Options for realizing fewer live objects

- Strategies for achieving fewer live objects per minor GC
  - Smaller Eden space
    - Fewer objects in general implies fewer live objects
      - But may not always hold … smaller Eden space implies Eden space fills faster … implies more frequent minor GCs … less time for an object to die … higher live objects to objects allocated ratio
  - Common approach with a concurrent collector in old generation
    - Push “collection work” off til later
      - Leads to more frequent old generation [concurrent] collections, higher CPU consumption … can impact capacity
Important Concept #2

Options for realizing fewer live objects

- Strategies for achieving fewer live objects per minor GC
  - Remove unnecessary object retention
    - Not to be confused with unnecessary object allocation
  - Profile the application
    - Look for potential assignment between two objects that spans multiple generations
      - Objects in young gen may be presumed alive
      - Look at java.util.LinkedList.clear() as an example
Third Important Concept

To fill a shape with an image:

1. Use existing picture box, DO NOT delete and create new picture box.
2. Right click on the shape.
3. At the bottom of the submenu select “Format Shape”.
4. Select “Fill” at the top of the “Format Shape” dialog box.
5. Select “Picture or Texture fill” from the options.
6. And select “File” under the “Insert from” option.
7. Navigate to the file you want to use and select “Insert”.
8. On the “Format” tab, in the Size group, click on “Crop to Fill” in the Crop tool and drag the image bounding box to the desired size.

DELETE THIS INSTRUCTION NOTE WHEN NOT IN USE.
Important Concept #3

Old Generation Collection Frequency

- First some background info
  - Lots of variation with Old Generation collectors, STW & concurrent
    - HotSpot’s Parallel GC, ParallelOld GC and Serial GC are STW
    - HotSpot’s CMS GC and G1 GC are mostly concurrent
    - JRockit has both STW and mostly concurrent GCs
    - Zing C4 is a concurrent GC
    - J9 (AFAIK) has both STW and mostly concurrent
HotSpot JVM Java Heap Layout

New object allocations

Retention / aging of young objects during young / minor GCs

Eden

From Survivor

To Survivor

Old Generation

Promotions of longer lived objects during minor GCs

For older / longer living objects

[ Permanent Generation | Metaspace (JDK 8+) ]

for VM & class meta-data

The Java Heap
Important Concept #3

Old Generation Collection Frequency

- Old generation collection frequency is dictated by
  - the rate at which objects are promoted from young generation to old generation
    - HotSpot promotes objects based on an age tenuring threshold
      - Age is # of minor GCs an object survives
      - Age tenuring threshold computed at each minor GC
      - Size of the old generation space
    - STW GCs collect when space is exhausted, concurrent GCs use an occupancy threshold to initiate concurrent collection cycle
Important Concept #3

How to reduce Old Generation collection frequency

- Strategies for reducing the promotion rate
  - More effective object aging
    - In HotSpot, tune survivor space sizes using -XX:SurvivorRatio – strive to promote objects at max tenuring threshold
    - Increase Eden space size using -Xmn, -XX:[Max]NewSize
      - Minor GCs occur less frequently, object age increments slower
  - Reduce object allocation rate
    - Minor GCs occur less frequently, object age increments slower
Important Concept #3

How to reduce Old Generation collection frequency continued …

- Strategies for reducing the promotion rate
  - Reduce object retention
    - Fewer objects to promote per minor GC
    - Also reduces the required survivor space size to avoid early promotion
    - May also reduce the frequency of minor GCs which allows objects to age longer in young generation survivor spaces
Important Concept #3

How to reduce Old Generation collection frequency continued …

- Make old generation size bigger (assuming same promotion rate)

Old Generation (3 GB)

Old Generation (6 GB)

- Same promotion rate, 2x increase in old gen space, time between old generation collection increases 2x
Important Concept #3

Choices for increasing the size of old generation

- For the HotSpot JVM
  - For Parallel GC, CMS GC and Serial GC
    - Increase -Xms and -Xmx, and keep young generation sizing the same i.e. -Xmn, -XX:[Max]NewSize, -XX:SurvivorRatio
  - For G1 GC
    - May require an increase in both -Xms / -Xmx and -XX:InitiatingHeapOccupancyPercent
Important Concept #3

How to reduce Old Generation collection frequency continued …

- Lower old generation occupancy after collection
  - A form of lowering object retention
  - More available space to fill in old generation after collection
- Approach similar as described earlier for reducing object retention
  - Profile the application and look for objects unnecessarily kept live longer than need be
Important Concept #3

What about frequency of concurrent collection cycles?

- Reducing frequency of concurrent collection cycles in old generation
  - Increase heap occupancy threshold that initiates a concurrent cycle
    - Reduces total CPU cycles used for GC over life of application execution
    - But, higher risk of running out of available old generation space
      - Don’t want to “lose the race”
  - In HotSpot CMS GC, -XX:CMSInitiatingHeapOccupancyFraction
    - Also suggest -XX:+UseCMSInitiatingOccupancyOnly
  - In HotSpot G1 GC, -XX:InitiatingHeapOccupancyPercent
Forth Important Concept
Important Concept #4

Duration of old generation collection

- I’m making a distinction between a Full GC, and an old generation collection
  - Full GC may involve collecting and compacting both young generation and old generation – (this is the default behavior for the HotSpot JVM)
  - Collecting old generation means the collecting of old generation only, not necessarily compacting old generation, and (obviously) not collecting young generation
Important Concept #4

- Duration of old generation collection
  - Similar to minor GC, number of live objects has strongest influence of old generation collection duration
  - Other things that can influence old generation collection duration
    - Overuse (or abuse) of Reference objects; Weak, Soft, Phantom and Final
      - Note: SoftReferences can also influence old generation collection frequency
    - Memory locality, etc.
Important Concept #4

- For old generation collection, desire duration to be dominated by time to mark live objects
  - And, if the GC also does compaction, then also live object compaction
  - HotSpot Parallel GC and Serial GC mark live objects and compacts them, aka a mark-compact collector
- These GCs also happen to be STW collectors, both young & old generation collectors
Important Concept #4

- Note, for [mostly] concurrent old generation collection, also desire duration to be dominated by time to mark live objects
  - HotSpot CMS GC does not do compaction - it is a mark-sweep collector
    - No compaction unless promotion failure – resorts to a Full GC
  - HotSpot G1 GC achieves compaction by evacuating live objects from old generation regions to available (empty) old generation regions
    - Evacuation (copy) time is where you want G1 to spend most of its time outside of marking
    - Evacuation in G1 is a STW part of a G1 concurrent old gen collection
Important Concept #4

What can be done to reduce old generation collection pause time

- Faster memory and CPU (in general a faster system)
- Faster STW GC algorithm
- Concurrent old generation collection
  - May introduce more CPU consumption which can reduce throughput and available capacity
  - May require more memory (footprint) to handle floating garbage
- Fewer live objects in old generation
Important Concept #4

Summary of concurrent old generation collection trade-offs

- What are you gaining with [mostly] concurrent collector
  - Lower GC pause times / improved latency
- What are you sacrificing / giving up
  - Usually more Java heap
  - More CPU consumption
- Translation, for improving latency, you’re sacrificing something in at least one, or possibly all three of
  - footprint, throughput and capacity
Important Concept #4

Strategies for Fewer Live Objects

- Achieving fewer live objects per old generation collection
  - Larger old generation space?
    - Possibly
      - In general, if the increase in size allows more objects to die
      - Example, if the length of a typical application transaction is greater than the time between old generation collection, then increasing old generation could reduce the number of live objects marked during an old generation collection
Summary

To fill a shape with an image.

1. Use existing picture box, DO NOT delete and create new picture box.
2. Right click on the shape.
3. At the bottom of the submenu select "Format Shape".
4. Select "Fill" at the top of the "Format Shape" dialog box.
5. Select "Picture or Texture fill" from the options.
6. And select "File" under the "Insert from" option.
7. Navigate to the file you want to use and select "Insert".
8. On the "Format" tab, in the Size group, click on "Crop to Fill" in the Crop tool and drag the image bounding box to the desired size.
9. DELETE THIS INSTRUCTION NOTE WHEN NOT IN USE.
Take away

If you think about GC tuning in terms of garbage collection frequency and duration, for both young generation, and old generation collection you can reason about the action to take to meet the throughput, latency, footprint and available capacity goals for your application.
Credits

- Huge thank you to all HotSpot GC team members, past and present
- Countless others, past and present from the HotSpot JVM team
Additional Reading (Books)

- Java Performance [1]
- The Garbage Collection Handbook [3]

CREATE THE FUTURE