Java & Coherence

Simon Cook - Sales Consultant, FMW for Financial Services

with help from

Adrian Nakon - CMC Markets & Andrew Wilson - RBS

Presentation Agenda

- An Overview of the Java Memory Management
- Java Garbage Collectors
- Tuning Garbage Collection for Coherence
- The Next Generation Garbage Collector
- Questions and Answers

Tuning the Underlying Platform is Important

Efficiency is the key

- Coherence is extremely fast – even with restricted resources
- Operational efficiency has many advantages
  - Better run-time performance
  - Fewer resources to manage
  - Fewer Oracle Coherence licenses to buy
- Remember to take a holistic when tuning
  - The hardware and the operating system
  - The Java Virtual Machine
  - Coherence configuration
  - Application code
  - Database tuning and optimisation

Tuning the Java Virtual Machine

Lots and lots and lots of options

- Current generation JVMs have many tuning options
  - Some will give small efficiencies
  - Some will give massive efficiencies
- Tuning your GC to minimise pause time will be key
  - Reduce the number of Full GCs
  - Reduce the latency overhead of GCs
- Long GCs are disastrous for a distributed caches such as Coherence
- Understand your latency requirements and work towards them
- You will have to make compromises in some way

Generational Garbage Collection

Employed by all HotSpot GC algorithms

- The majority of JVMs use generational collectors
- The heap is split into “generations”
  - Young, newly created objects
  - Old, longer lived objects
- Weak generational hypothesis
  - Proved by observation and it’s extremely accurate for Java Applications
- Most objects are very short lived
  - 80-98% of all newly allocated objects die within a few million instructions
  - 80-98% of all newly allocated objects die before another megabyte has been allocated

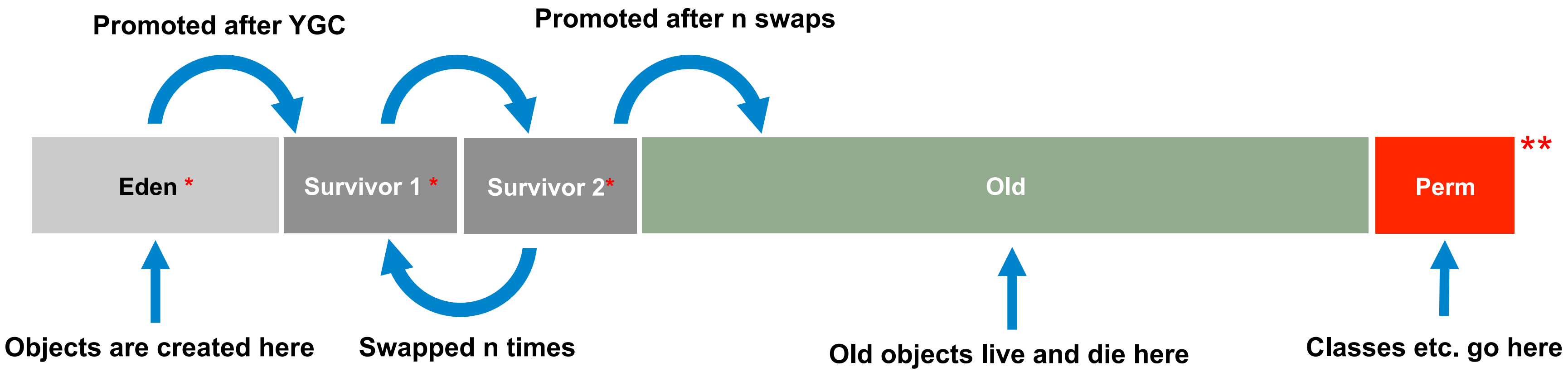
Some Quick Words on Garbage Collectors

Dealing with the rubbish

- There are two classes of GC algorithms in Java
  - The Throughput Collectors
  - The Low Pause (latency) Collectors
- Throughput collectors are the default
  - They reorganise the old heap during a collection
  - They are not suitable for Coherence
- CMS is a low pause collector
  - Aims to keep application pauses to a minimum
  - Is a suitable collector for Coherence
- G1 is still in development – do not use for Coherence today *

The Java Heap Layout

For all collector algorithms

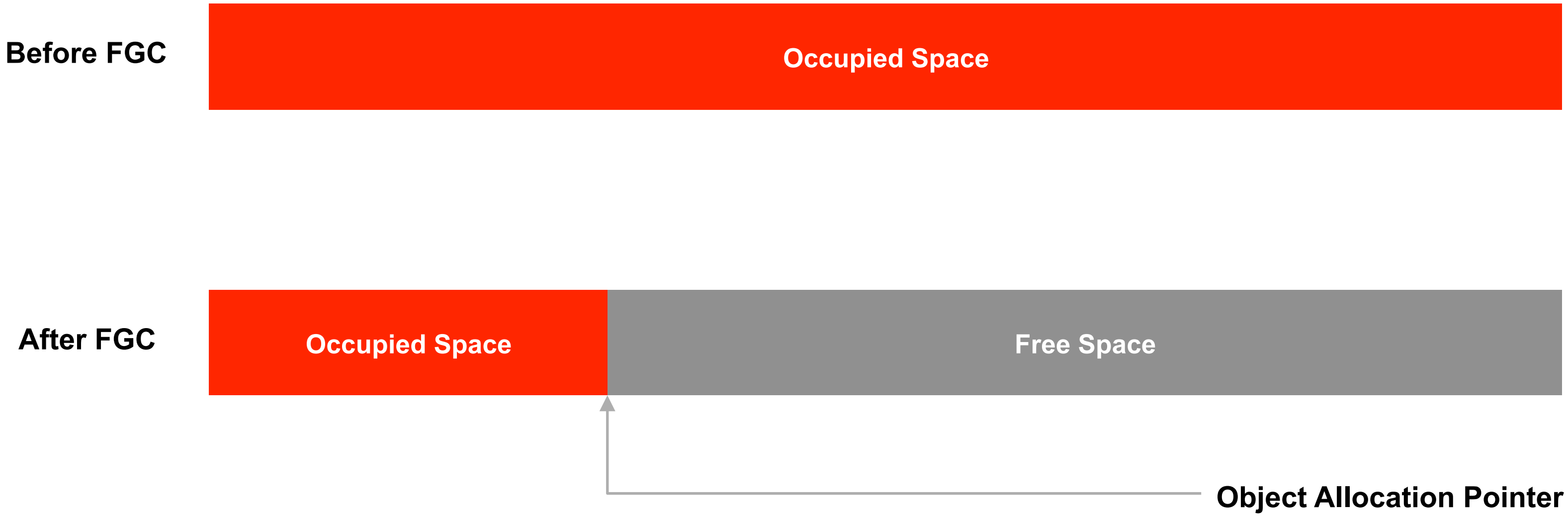


* Young Space is composed of Eden and the two Survivor Spaces.

** Perm Space is going away!

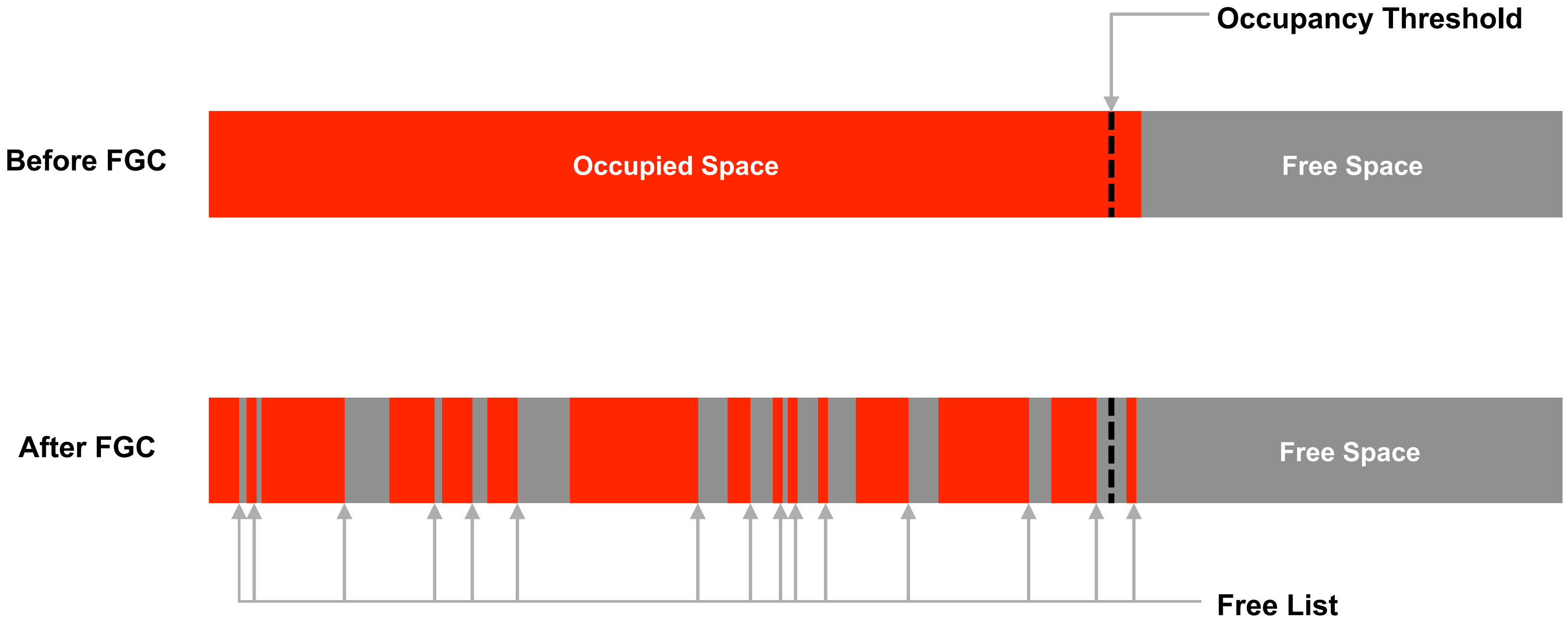
Compacted Old Space

Throughput Collectors – Serial and Parallel (and G1)



Fragmented Old Space

CMS Collector



"You may think it's impossible to run large heaps with CMS on restricted hardware. This is simply not true, it's very possible!"

Adrian Nakon

Coherence Architect, CMC Markets

A List of Stop the World Pauses

Know your enemy

1. Young space collections
2. Full GCs – All collectors
3. System GCs – Called via JMX or the application
4. CMS Initial Mark Phase
5. CMS Remark Phase
6. CMS Concurrent Mode Failure

ORACLE

Tuning the CMS Collector

The collector of choice for Coherence

- CMS will enable large heaps, even on restricted hardware
- CMS is not like the other collectors
  - Concurrent collections with multiple, small STW pauses
- Running with defaults can be fine for small heaps
- For larger heaps you need to consider tuning CMS for best results
- Should you use NIO or 64 bit JVMs?
  - CMS can perform very well will large heaps when correctly tuned
  - NIO has limitations and garbage collection a manual task

Tuning the CMS Collector

Good performance will take some thought

- A different collector is used for the Young Space
  - The ParNew Collector
- The aim is to minimise the STW pauses
  - The Young Space Collections
  - The Initial Mark
  - The Remark
- CMS is concurrent and will therefore require CPU
  - It will compete with your application during collections
- It fragments the Old Space
  - Object allocations are more complicated

Tuning the CMS Collector

The Weak Generational Hypothesis

- It is important to give objects the opportunity to die young
- Young Collections are fast and efficient
  - Only live objects are copied
  - Most objects will be dead (transient) so it is fast
  - Space is cleared quickly with minimal application pauses
- Sizing the Young ratio is key
  - Size the survivor spaces appropriately
  - Configure the Tenuring Threshold appropriately
  - Think about your cache expiry settings if appropriate (remember backups!)

ORACLE

Tuning the CMS Collector

Minimise the marking phases pause times

- Minimise your pause times
  - The Initial Mark Phase
  - The Remark Phase
- CMS has to scan Young Space to look for relationships
- If Young Space is not empty this will take time
- You can instruct CMS to wait for a Young GC before starting
- An empty Young Space will dramatically reduce marking times

Tuning the CMS Collector

Worst case scenario

- Concurrent Mode Failure
  - All bets are off
  - No new objects can be allocated into the Old Space
  - The heap will be compacted to recover fragmented space
  - This may take some time, grab a coffee
- Sizing your heap correctly is key to avoiding this
  - Undersized heaps will make CMS work overtime
- Allowing objects to die in the young space will help avoid this
  - Remember The Weak Generational Hypothesis
  - Most objects die young and can be collected easily

Recommended Settings

There may be some more, HotSpot has many

- Limited CPU resource results in ...
  - Fewer JVM's per server (less possible context switching)
  - Strive to keep as much garbage out of tenured space as possible
  - Maximum size of Young Space is limited by Young Gen collection time.
- Low latency requirements
  - Ensure Young gens and CMS operations (mark / remark phases) are tightly integrated.
- Large data heaps required
  - Use 64 bit JVM
- Every Application is different, do not just rely on the default JVM settings

Recommended Base Settings

Generic JVM settings

- `-verbose:gc`
- `-XX:+UseConcMarkSweepGC`
- `-XX:+UseParNewGC`
- `-XX:+HeapDumpOnOutOfMemoryError`
- `-XX:HeapDumpPath=coherence/logs/<filename>`
- `-XX:+UseNUMA`

ORACLE

Recommended Logging Settings

Logging related JVM settings

- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`
- `-XX:+PrintGCDateStamps`
- `-XX:+PrintTenuringDistribution`
- `-Xloggc:/opt/oracle/admin/coherence/logs/<filename>`

Recommended CMS Settings

CMS tuning JVM Settings

- `-XX:MaxTenuringThreshold=15`
- `-XX:CMSWaitDuration=300000`
- `-XX:+CMSScavengeBeforeRemark`
- `-XX:CMSInitiatingOccupancyFraction=65`
- `-XX:+UseCMSInitiatingOccupancyOnly`
- `-XX:SurvivorRatio=4`
- `-Xms<x>m -Xmx<x>m -Xmn<y>m`

Sizing your heap

The Goldilocks Heap

- You’re looking for The Goldilocks Heap
  - Not too small
  - Not too big
  - Just right
- Profile your applications and it’s object allocation and de-allocation
  - Coherence – Caches, expiry, processing, proxies, monitoring, etc.
- You have control over
  - Initial and maximum overall heap size
  - Perm space size
  - Young space/old space ratio
  - Survivor spaces/young space ratio

ORACLE

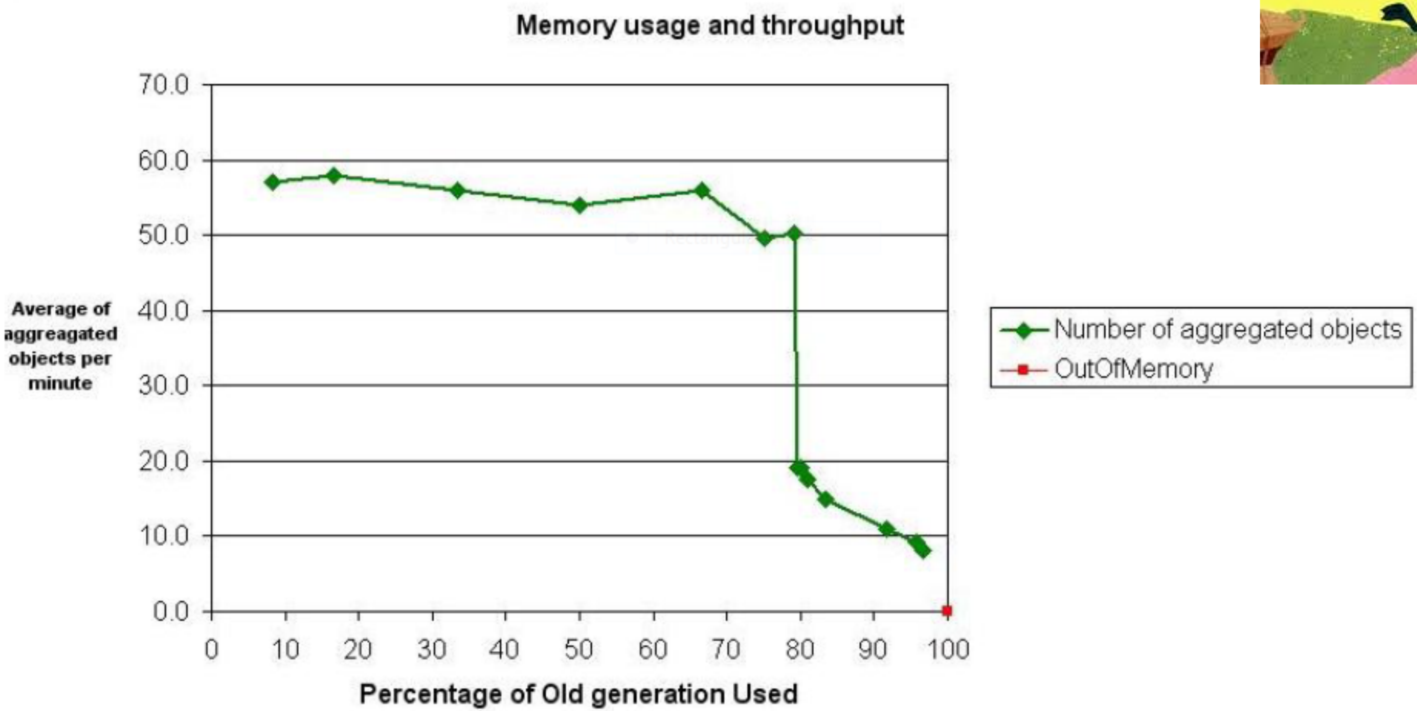
“If your heap is 80% full after a full GC then your application performance will drop off a cliff.”

Andrew Wilson

Coherence Architect, RBS

The Royal Bank of Scotland Group

Sizing Your Heap



Sizing the Young Space

If possible, allow objects to die young

- Remember the Weak Generational Hypothesis
  - The vast majority of objects die very young
- Young collections are cheaper than old
- You need to meet the following criteria
  - Make the young space large enough so objects die young
  - Do not make the young space too large – long GCs
- It's a balancing act
  - You need to understand your application's memory profile

About Survivor Spaces

Wait for short-lived objects to die

- Survivor spaces give objects more opportunity to die
- You have full control over this
- You can set the Survivor Space Ratio
  - `-XX:SurvivorRatio=<n>`
- You can set the Maximum Tenuring Threshold (number of swaps)
  - `-XX:MaxTenuringThreshold=15`
- If you get this right
  - Your young GCs will remain efficient
  - More objects will die in young

Tenuring Distributions

The flow of objects through the survivor spaces

```
[GC 526703.667: [ParNew
Desired survivor size 53673984 bytes, new threshold 8 (max 8)
- age   1:   19709184 bytes,   19709184 total
- age   2:     382384 bytes,   20091568 total
- age   3:     435072 bytes,   20526640 total
- age   4:     486544 bytes,   21013184 total
- age   5:     725872 bytes,   21739056 total
- age   6:     541144 bytes,   22280200 total
- age   7:     741464 bytes,   23021664 total
- age   8:     523912 bytes,   23545576 total
: 852844K->26740K(943744K), 0.1001560 secs] 2780990K->1959523K(8283776K),
  0.1003690 secs] [Times: user=0.26 sys=0.00, real=0.10 secs]
```

Tools

There are many tools, some free, some not.

OS Level Tools

- sar - ksar
- vmstat
- iostat
- Free
- nmon

Java Tools

- -verbose:gc
- gcstat
- jvisualvm
- jconsole

Log management tools

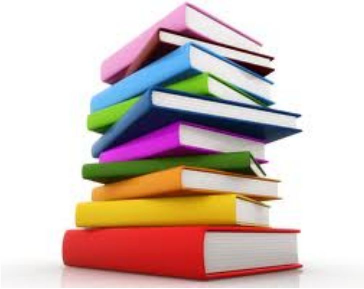
- vi, more, less, grep
- GCViewer
- Splunk
- Logscape
- LogMX

Payware Tools

- Oracle Enterprise Manager
- Wily Introscope (CA)
- ITRS Geneos
- SL RTView
- Evident Clearstone

Further Reading

Lots of good material out there



"A Generational Mostly-concurrent Garbage Collector" by Tony Printezis and David Detlefs – The guys who wrote CMS!

http://labs.oracle.com/techrep/2000/abstract-88.html

The Garbage First (G1) Collector

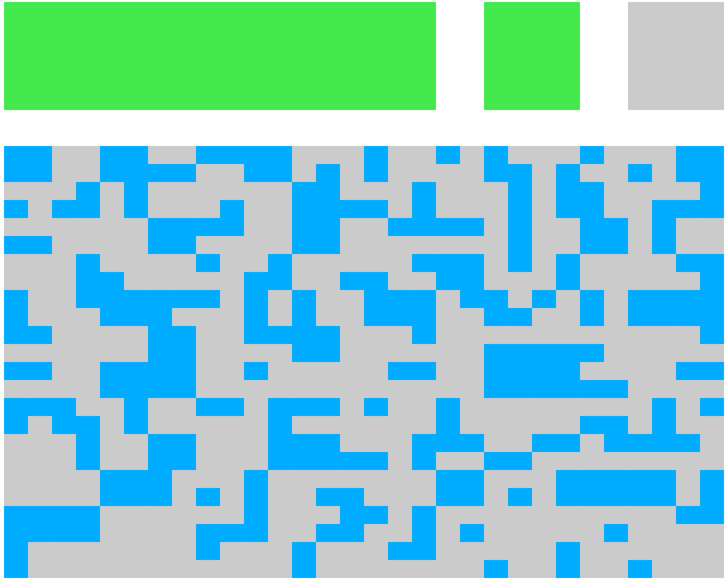
The next generation HotSpot Collector

- CMS Replacement (early access JRE 6 u14 onwards*)
- Server "Style" Garbage Collector
- Parallel
- Mostly Concurrent
- Generational
- Good Throughput
- *Compacting*
- *Improved ease-of-use*
- *Predictable (though not hard real-time)*

Colour Key for Heap Spaces

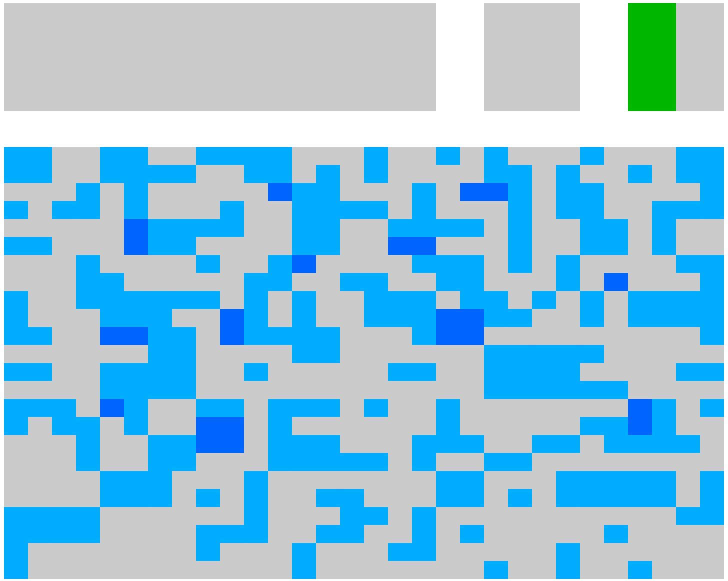
- Non-Allocated Space
- Young Generation
- Old Generation
- Recently Copied in Young Generation
- Recently Copied in Old Generation

Young GCs in CMS



- Young generation, split into
  - Eden
  - Survivor spaces
- Old generation
  - In-place de-allocation
  - Managed by free lists
  - Heap fragmentation

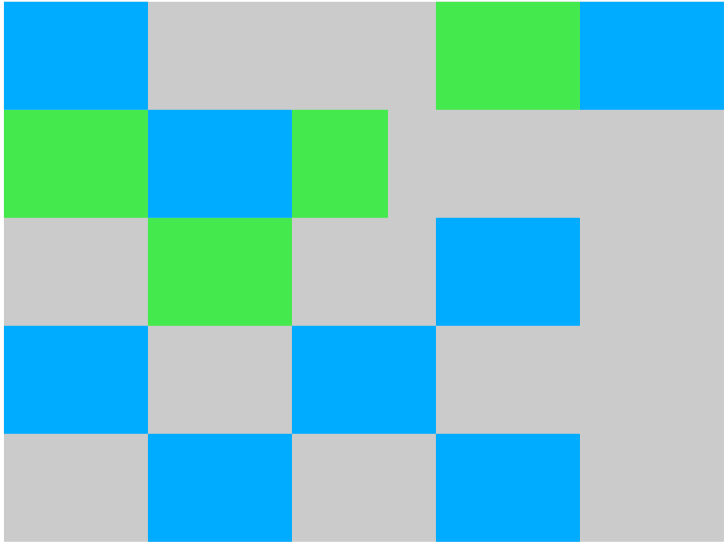
Young GCs in CMS



- End of young generation GC

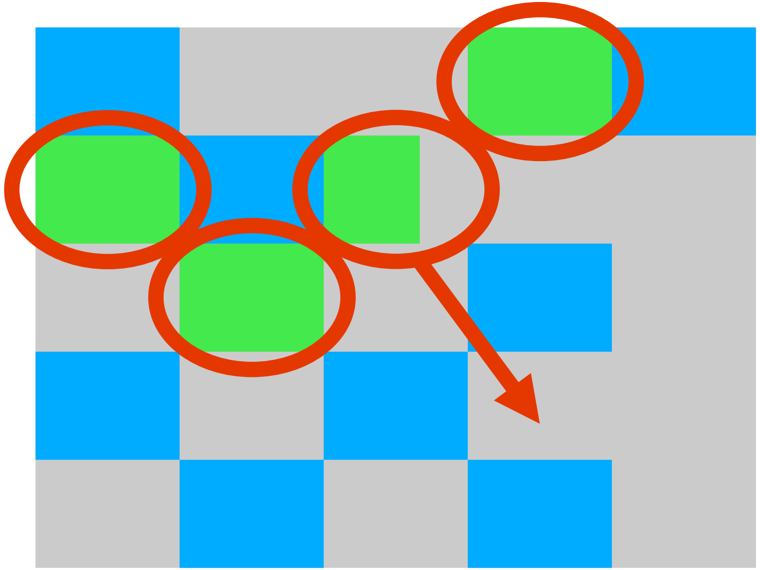
Young GCs in G1

- Heap split into regions
  - Currently 1MB regions
- Young generation
  - A set of regions
- Old generation
  - A set of regions



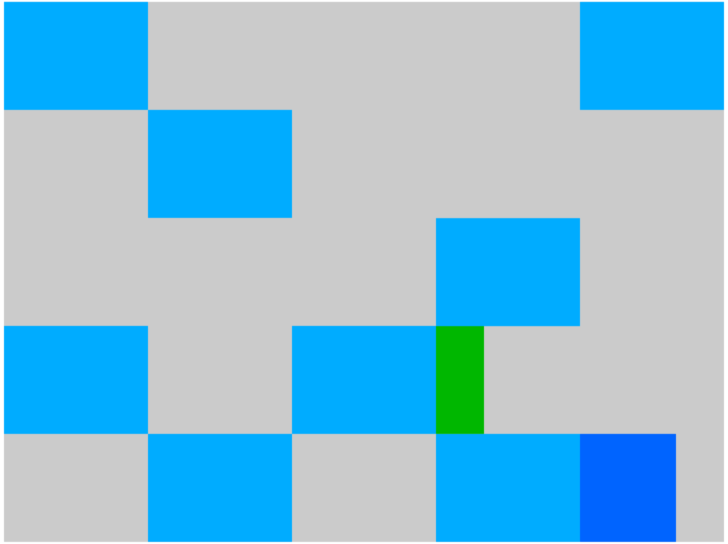
Young GCs in G1

- During a young generation GC
  - Survivors from the young regions are evacuated to:
    - Survivor regions
    - Old regions



Young GCs in G1

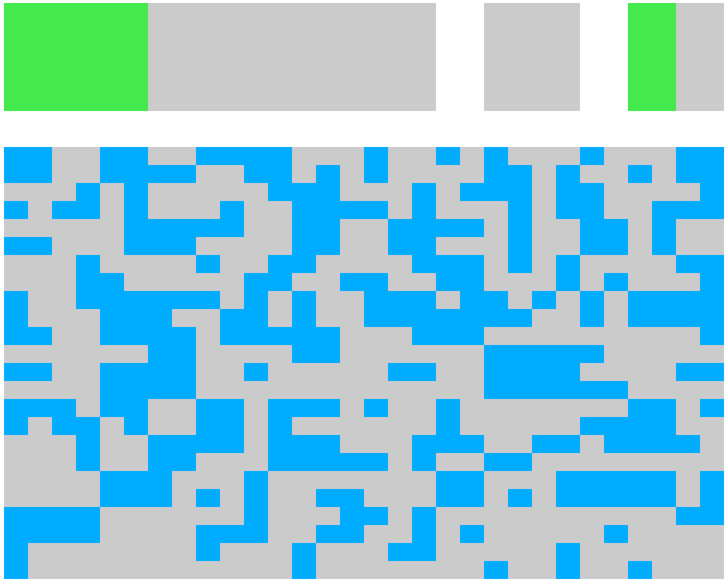
- End of young generation GC



Summary: Young GCs 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

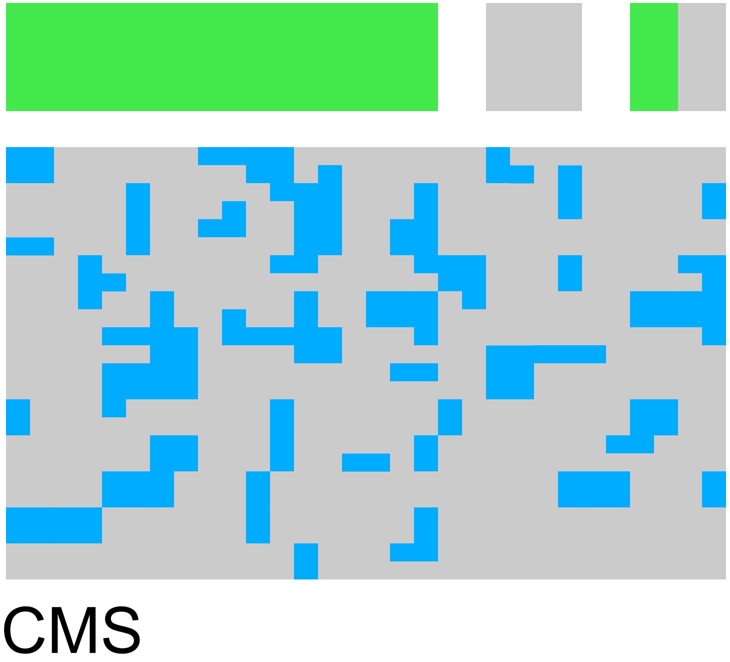
Old GCs in CMS (Sweeping After Marking)



CMS

- Concurrent marking phase
  - Two stop-the-world pauses
- Initial mark
- Remark
  - Marks reachable (live) objects
  - Unmarked objects are deduced to be unreachable (dead)

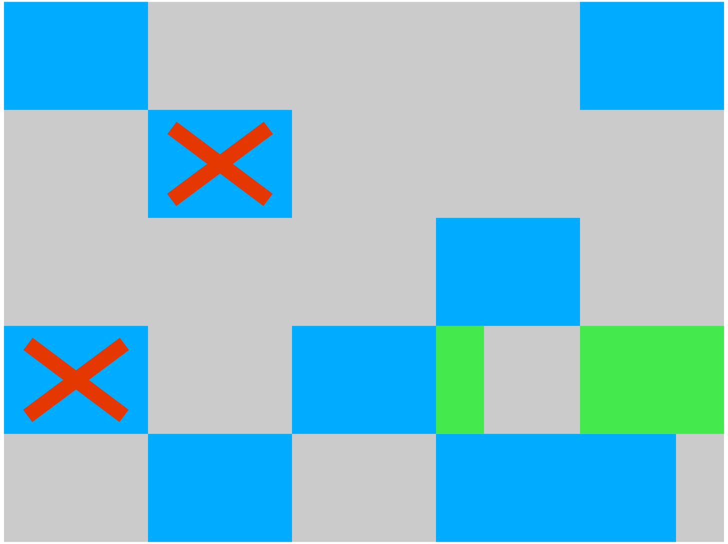
Old GCs in CMS (Sweeping After Marking)



- End of concurrent sweeping phase
- All unmarked objects are de-allocated

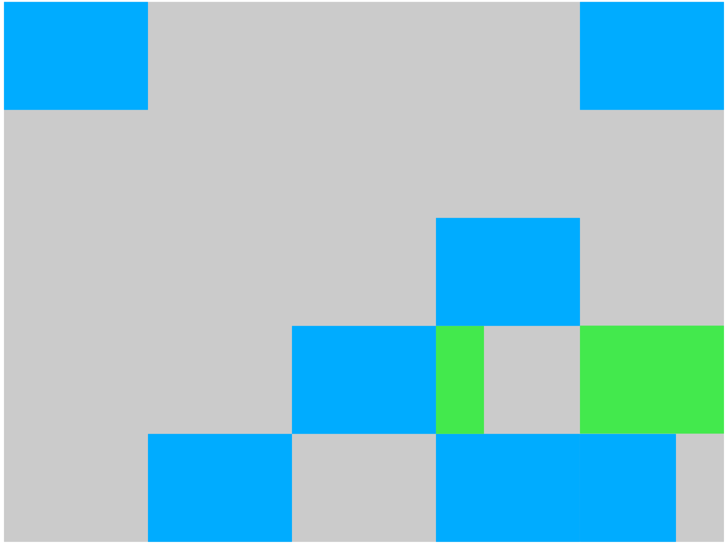
Old GCs in G1 (After Marking)

- Concurrent marking phase
  - One stop-the-world pause
- Remark
- (Initial mark piggybacked on an evacuation pause)
  - Calculates liveness information per region
- Empty regions can be reclaimed immediately



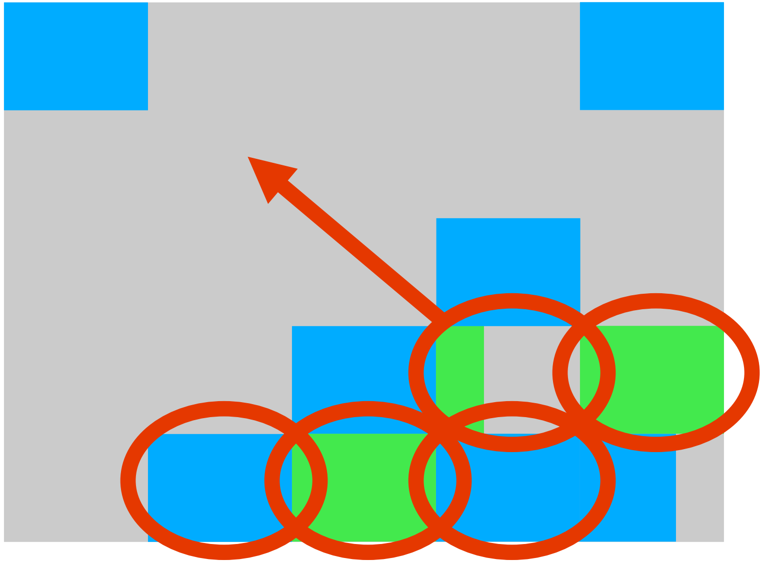
Old GCs in G1 (After Marking)

- End of remark phase



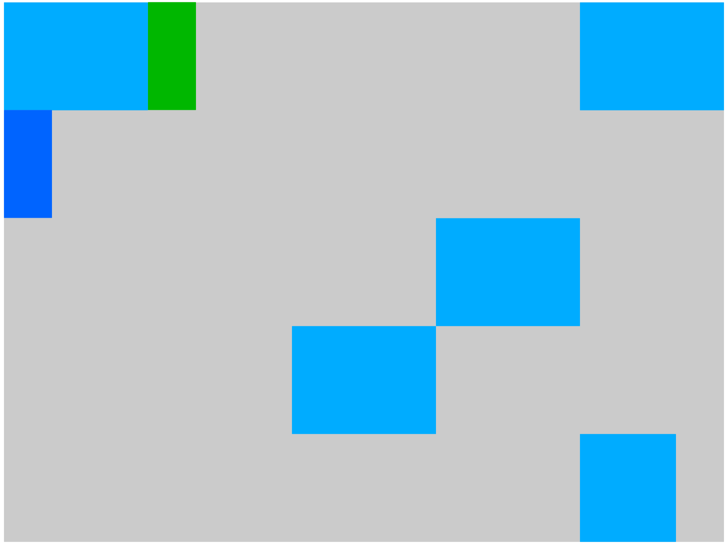
Old GCs in G1 (After Marking)

- Reclaiming old regions
  - Pick regions with low live ratio
  - Collect
- Only a few old regions collected per such GC

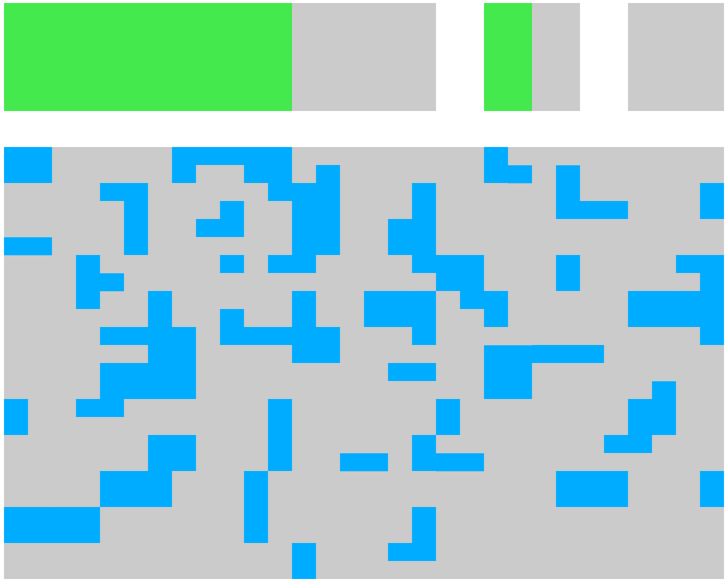


Old GCs in G1 (After Marking)

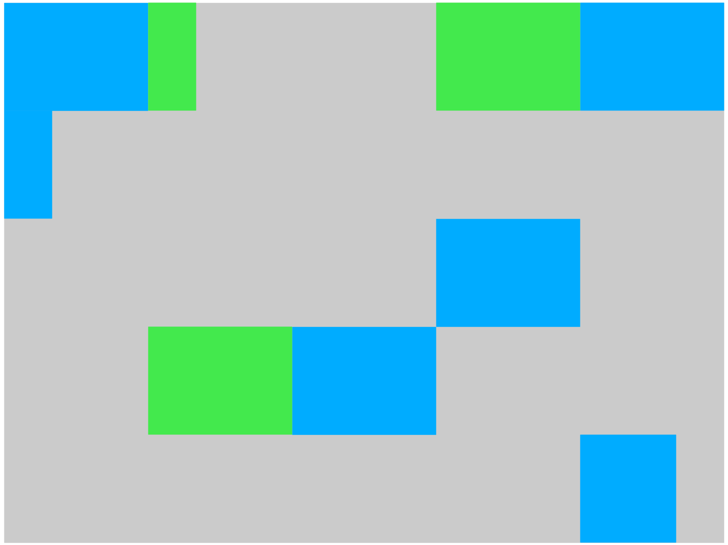
- We might leave some garbage objects in the heap
  - In regions with very high live ratio
  - We might collect them later



CMS vs. G1 Comparison



CMS



G1

In Summary

Don’t just accept the defaults, every application differs

- Strive to keep garbage out of Tenured/Old Space
- Size young accordingly
  – Too big and your pauses will be too long
  – Too small and too much garbage will be tenured
- Think about your Survivor Spaces
  – Allow objects to die young
  – Look at the object distributions
- Synchronise young and old collections with CMS
- Overall Heap size is important
  – Don’t give the GCs too much work to do

Q&A

Hardware and Software

ORACLE

Engineered to Work Together

ORACLE®