(Really) Understanding Garbage Collection

Gil Tene, CTO & co-Founder, Azul Systems
@giltene
This Talk’s Purpose / Goals

✎ This talk is focused on GC education

✎ This is not a “how to use flags to tune a collector” talk

✎ This is a talk about how the “GC machine” works

✎ Purpose: Once you understand how it works, you can use your own brain...

✎ You’ll learn just enough to be dangerous...

✎ The “Azul makes the world’s greatest GC” stuff will only come at the end, I promise...
High level agenda

- Some GC fundamentals, terminology & mechanisms
- Classifying currently available production collectors
- Why Stop-The-World is a problem
- Concurrent Approaches
- The C4 collector: What a solution to STW looks like...
About me: Gil Tene

- co-founder, CTO @Azul Systems
- Have been working on “think different” GC and runtime approaches since 2002
- Created Pauseless & C4 core GC algorithms (Tene, Wolf)
- A Long history building Virtual & Physical Machines, Operating Systems, Enterprise apps, etc...
- I also depress people by demonstrating how terribly wrong their latency measurements are...

* working on real-world trash compaction issues, circa 2004
Memory use

How many of you use heap sizes of:

- more than $\frac{1}{2}$ GB?
- more than 1 GB?
- more than 2 GB?
- more than 4 GB?
- more than 10 GB?
- more than 20 GB?
- more than 50 GB?
- more than 100 GB?
Why should you understand (at least a little) how GC works?
The story of the good little architect

- A good architect must, first and foremost, be able to impose their architectural choices on the project...

- Early in Azul’s concurrent collector days, we encountered an application exhibiting 18 second pauses
  
  - Upon investigation, we found the collector was performing 10s of millions of object finalizations per GC cycle
    
    *We have (long) since made reference processing fully concurrent...

- Every single class written in the project had a finalizer
  
  - The only work the finalizers did was nulling every reference field

- The right discipline for a C++ ref-counting environment
  
  - The wrong discipline for a precise garbage collected environment
Much of what People seem to “know” about Garbage Collection is wrong

In many cases, it’s much better than you may think
- GC is extremely efficient. Much more so that malloc()
- Dead objects cost nothing to collect
- GC will find all the dead objects (including cyclic graphs)
  ...

In many cases, it’s much worse than you may think
- Yes, it really will stop for ~1 sec per live GB (except with Zing)
- No, GC does not mean you can’t have memory leaks
- No, those pauses you eliminated from your 20 minute test are not gone
  ...

©2017 Azul Systems, Inc.
Some GC Terminology
A Concurrent Collector performs garbage collection work concurrently with the application's own execution.

A Parallel Collector uses multiple CPUs to perform garbage collection.
Classifying a collector’s operation

- A **Concurrent** Collector performs garbage collection work concurrently with the application’s own execution.
- A **Parallel** Collector uses multiple CPUs to perform garbage collection.
- A **Stop-the-World** collector performs garbage collection while the application is completely stopped.
- A **Monolithic** Collector performs the entire collection work in a single, indivisible step.
- An **Incremental** collector performs a garbage collection operation or phase as a series of smaller discrete operations with (potentially long) gaps in between.
- Mostly means sometimes it isn’t (usually means a different fall back mechanism exists).
Classifying a collector's operation

- **Concurrent** Collector performs garbage collection work concurrently with the application's own execution.
- **Parallel** Collector uses multiple CPUs to perform garbage collection.
- **Stop-the-World** collector performs garbage collection while the application is completely stopped.
- **Monolithic** Collector performs the entire collection work in a single, indivisible step.
- **Incremental** collector performs a garbage collection operation or phase as a series of smaller discrete operations with (potentially long) gaps in between.
- **Mostly** means sometimes it isn’t (usually means a different fall back mechanism exists).
- **Experimental** means “If you are lucky, it will only crash your application.”
Precise vs. Conservative Collection

A Collector is **Conservative** if it is unaware of some object references at collection time, or is unsure about whether a field is a reference or not.

A Collector is **Precise** if it can fully identify and process all object references at the time of collection.

A collector MUST be precise in order to move objects.

- The COMPILERS need to produce a lot of information (OopMaps)

All commercial server JVMs use precise collectors.

- All commercial server JVMs use some form of a moving collector
Safepoints

A **GC Safepoint** is a point or range in a thread's execution where the collector can identify all the references in that thread's execution stack.

- "Safepoint" and "GC Safepoint" are often used interchangeably.
- But there are other types of safepoints, including ones that require more information than a GC safepoint does (e.g. deoptimization).

"**Bringing a thread to a safepoint**" is the act of getting a thread to reach a safepoint and not execute past it.

- Close to, but not exactly the same as "stop at a safepoint".
  - e.g. JNI: you can keep running in, but not past the safepoint.
- Safepoint opportunities are (or should be) frequent.

In a **Global Safepoint** all threads are at a Safepoint.
Behavior that is common to ALL precise GC mechanisms

- Identify the live objects in the memory heap
- Reclaim resources held by dead objects
- Periodically relocate live objects

Examples:
- Mark/Sweep/Compact (common for Old Generations)
- Copying collector (common for Young Generations)
Mark (aka “Trace”)

- Start from “roots” (thread stacks, statics, etc.)
- “Paint” anything you can reach as “live”
- At the end of a mark pass:
  - all reachable objects will be marked “live”
  - all non-reachable objects will be marked “dead” (aka “non-live”).
- Note: work is generally linear to “live set”
Sweep

- Scan through the heap, identify “dead” objects and track them somehow
  - (usually in some form of free list)

- Note: work is generally linear to heap size
Compact

Over time, heap will get “swiss cheesed”: contiguous dead space between objects may not be large enough to fit new objects (aka “fragmentation”)

Compaction moves live objects together to reclaim contiguous empty space (aka “relocate”)

Compaction has to correct all object references to point to new object locations (aka “remap” or “fixup”)

Remap scan must cover all references that could possibly point to relocated objects

Note: work is generally linear to “live set”
Copy

- A copying collector moves all lives objects from a “from” space to a “to” space & reclams “from” space.
- At start of copy, all objects are in “from” space and all references point to “from” space.
- Start from “root” references, copy any reachable object to “to” space, correcting references as we go.
- At end of copy, all objects are in “to” space, and all references point to “to” space.
- Note: work generally linear to “live set”
Mark/Sweep/Compact, Copy, Mark/Compact

- Copy requires $2x$ the max. live set to be reliable
- Mark/Compact [typically] requires $2x$ the max. live set in order to fully recover garbage in each cycle
- Mark/Sweep/Compact only requires $1x$ (plus some)
- Copy and Mark/Compact are linear only to live set
- Mark/Sweep/Compact is linear (in sweep) to heap size
- Mark/Sweep/(Compact) may be able to avoid some moving work
- Copying is [typically] “monolithic”
Generational Collection

Weak Generational Hypothesis; “most objects die young”

Focus collection efforts on young generation:
- Use a moving collector: work is linear to the live set
- The live set in the young generation is a small % of the space
- Promote objects that live long enough to older generations

Only collect older generations as they fill up
- “Generational filter” reduces rate of allocation into older generations

Tends to be (order of magnitude) more efficient
- Great way to keep up with high allocation rate
- Practical necessity for keeping up with processor throughput
Generational Collection

- Requires a “Remembered set”: a way to track all references into the young generation from the outside
- Remembered set is also part of “roots” for young generation collection
- No need for 2x the live set: Can “spill over” to old gen
- Usually want to keep surviving objects in young generation for a while before promoting them to the old generation
  - Immediate promotion can significantly reduce gen. filter efficiency
  - Waiting too long to promote can eliminate generational benefits
The typical combos in production JVMs

- Young generation *usually* uses a copying collector
- Young generation is *usually* monolithic, stop-the-world

- Old generation *usually* uses Mark/Sweep/Compact
- Old generation may be STW, or Concurrent, or mostly-Concurrent, or Incremental-STW, or mostly-Incremental-STW
Some non monolithic-STW stuff
Concurrent Marking

Mark all reachable objects as “live”, but object graph is “mutating” under us.

Classic concurrent marking race: mutator may move reference that has not yet been seen by the marker into an object that has already been visited

- If not intercepted or prevented in some way, will corrupt the heap

Example technique: track mutations, multi-pass marking

- Track reference mutations during mark (e.g. in card table)
- Re-visit all mutated references (and track new mutations)
- When set is “small enough”, do a STW catch up (mostly concurrent)

Note: work grows with mutation rate, may fail to finish
Incremental Compaction

- Track cross-region remembered sets (which region points to which)
- To compact a single region, only need to scan regions that point into it to remap all potential references
- Identify regions sets that fit in limited time
  - Each such set of regions is a Stop-the-World increment
  - Safe to run application between (but not within) increments
- Note: work can grow with the square of the heap size
  - The number of regions pointing into a single region is generally linear to the heap size (the number of regions in the heap)
Delaying the inevitable

- Some form of copying/compaction is inevitable in practice
  - And compacting anything requires scanning/fixing all references to it

- Delay tactics focus on getting “easy empty space” first
  - This is the focus for the vast majority of GC tuning

- Most objects die young [Generational]
  - So collect young objects only, as much as possible. Hope for short STW.
  - But eventually, some old dead objects must be reclaimed...

- ...
Delaying the inevitable

... Most old dead space can be reclaimed without moving it
  [e.g. CMS] track dead space in lists, and reuse it in place
  But eventually, space gets fragmented, and needs to be moved

Much of the heap is not “popular” [e.g. G1, “Balanced”]
  A non popular region will only be pointed to from a small % of the heap
  So compact non-popular regions in short stop-the-world pauses
  But eventually, popular objects and regions need to be compacted
Classifying common collectors
Classifying common legacy collectors
The typical combos in production JVMs

- Young generation *usually* uses a copying collector
- Young generation is *usually* monolithic, stop-the-world

- Old generation *usually* uses Mark/Sweep/Compact
- Old generation may be STW, or Concurrent, or mostly-Concurrent, or Incremental-STW, or mostly-Incremental-STW
HotSpot™ ParallelGC
Collector mechanism classification

- Monolithic Stop-the-world copying NewGen
- Monolithic Stop-the-world Mark/Sweep/Compact OldGen
HotSpot™ ConcMarkSweepGC (aka CMS)
Collector mechanism classification

- **Monolithic Stop-the-world copying NewGen (ParNew)**

- **Mostly Concurrent, non-compacting OldGen (CMS)**
  - Mostly Concurrent marking
    - Mark concurrently while mutator is running
    - Track mutations in card marks
    - Revisit mutated cards (repeat as needed)
    - Stop-the-world to catch up on mutations, ref processing, etc.
  - Concurrent Sweeping
  - Does not Compact (maintains free list, does not move objects)

- **Fallback to Full Collection (Monolithic Stop the world)**
  - Used for Compaction, etc.
HotSpot™ G1GC (aka “Garbage First”)

Collector mechanism classification

- **Monolithic Stop-the-world copying NewGen**
- **Mostly Concurrent, OldGen marker**
  - Mostly Concurrent marking
    - Stop-the-world to catch up on mutations, ref processing, etc.
    - Tracks inter-region relationships in remembered sets
- **Stop-the-world mostly incremental compacting old gen**
  - Objective: “Avoid, as much as possible, having a Full GC…”
  - Compact sets of regions that can be scanned in limited time
  - Delay compaction of popular objects, popular regions
- **Fallback to Full Collection (Monolithic Stop the world).**
  - Used for compacting popular objects, popular regions, etc.
Monolithic-STW GC Problems
One way to deal with Monolithic-STW GC
Hiccups by Time Interval

- Max per Interval
- 99%
- 99.9%
- 99.99%
- Max

Hiccups by Percentile Distribution

Max = 16023.552
Another way to cope: Creative Language

- “Guarantee a worst case of $X$ msec, 99% of the time”
- “Mostly” Concurrent, “Mostly” Incremental
  Translation: “Will at times exhibit long monolithic stop-the-world pauses”
- “Fairly Consistent”
  Translation: “Will sometimes show results well outside this range”
- “Typical pauses in the tens of milliseconds”
  Translation: “Some pauses are much longer than tens of milliseconds”
Looking at modern collectors

**ALL** modern collector designs for Java are based on concurrent compaction
Looking at modern collectors

- C4
  - The production collector in the Zing JVM (7, 8, 11, 13)

- Shenandoah
  - An experimental collector in the latest OpenJDK (12+)

- ZGC
  - An experimental collector in the latest OpenJDK (11+)
C4: Solving Stop-The-World
The problems that needed solving
(areas where the state of the art needed improvement)

Robust Concurrent Marking
- In the presence of high mutation and allocation rates
- Cover modern runtime semantics (e.g. weak refs, lock deflation)

Compaction that is not monolithic-stop-the-world
- E.g. stay responsive while compacting ¼ TB heaps
- Must be robust: not just a tactic to delay STW compaction
- [current “incremental STW” attempts fall short on robustness]

Young-Gen that is not monolithic-stop-the-world
- Stay responsive while promoting multi-GB data spikes
- Concurrent or “incremental STW” may both be ok
- Surprisingly little work done in this specific area
Azul’s “C4” Collector
Continuously Concurrent Compact Collector

- Concurrent guaranteed-single-pass marker
  - Oblivious to mutation rate
  - Concurrent ref (weak, soft, final) processing

- Concurrent Compactor
  - Objects moved without stopping mutator
  - References remapped without stopping mutator
  - Can relocate entire generation (New, Old) in every GC cycle

- Concurrent, compacting old generation
- Concurrent, compacting new generation

- No stop-the-world fallback
  - Always compacts, and always does so concurrently
C4’s Prime Directives

Always do the same thing
- Avoid the temptation to “solve” things by delaying them
- Avoid rare code paths
- Running under load for an hour should exercise the whole thing

Don’t be in a hurry
- Avoid the “if we don’t do this quickly it will get worse” trap
  - e.g. multi-pass marking
  - or pauses that depend on scale metrics
  - or being consistently slow during an entire phase of GC
- Allow collector to be “lazy” and run at a “relaxed pace”
- Keep up with allocation rate is only “pacing” reason
The secret to GC efficiency
Heap size vs. GC
CPU %
What empty memory controls

- Empty memory controls efficiency (amount of collector work needed per amount of application work performed)
- Empty memory controls the frequency of pauses (if the collector performs any Stop-the-world operations)
- Empty memory DOES NOT control pause times (only their frequency)
- With C4, we get the upside with no downside...
The secret to **maintaining** GC concurrency
The secret to maintaining GC concurrency

- in a single (non generational) heap:
  - You must complete each GC cycle before you run out of free memory
  - It takes \( n \text{Seconds} = \frac{\text{EmptyHeap}}{\text{AllocationRate}} \) to run out of memory
  - It takes \( m \text{Seconds} = \frac{\text{LiveSet}}{\text{MarkingMBPerSeconds}} \) to mark
  - \( \text{AllocationRate} : \text{MarkingMBPerSeconds} > \frac{\text{EmptyHeap}}{\text{LiveSet}} \)

- E.g. at 10GB/sec allocation, and 1GB/sec marking: Need 90% empty heap
The secret to maintaining GC concurrency

©2017 Azul Systems, Inc.

In a generational heap:

- You must complete each GC cycle before you run out of free memory.

- It takes \( n \text{Seconds} = \frac{\text{EmptyHeap}}{\text{AllocationRate}} \) to run out of memory.

- It takes \( m \text{Seconds} = \frac{\text{NewGenLiveSet}}{\text{NewGenMarMBPerSeconds}} \) to mark the newgen.

- \( \text{AllocationRate} : \text{MarkingMBPerSeconds} > \frac{\text{EmptyHeap}}{\text{NewGenLiveSet}} \)

- Typically a 10x-20x win.
Looking at modern collectors

- C4
  - The production collector in the Zing JVM (7, 8, 11, 13)
  - Generational: Both OldGen and NewGen are concurrent

- Shenandoah
  - An experimental collector in the latest OpenJDK (12+)
  - Non-generational: each collection must mark the whole heap

- ZGC
  - An experimental collector in the latest OpenJDK (11+)
  - Non-generational: each collection must mark the whole heap
C4 algorithm highlights

- Same core mechanism used for both generations
  - Concurrent Mark-Compact

- A Loaded Value Barrier (LVB) is central to the algorithm
  - Every heap reference is verified as “sane” when loaded
  - “Non-sane” refs are caught and fixed in a self-healing barrier

- Refs that have not yet been “marked through” are caught
  - Guaranteed single pass concurrent marker

- Refs that point to relocated objects are caught
  - Lazily (and concurrently) remap refs, no hurry
  - Relocation and remapping are both concurrent

- Uses “quick release” to recycle memory
  - Forwarding information is kept outside of object pages
  - Physical memory released immediately upon relocation
  - “Hand-over-hand” compaction without requiring empty memory
Benefits

ELIMINATES Garbage Collection as a concern for enterprise applications
GC Tuning
Java GC tuning is “hard”...

Examples of actual command line GC tuning parameters:

Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
   -XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
   -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
   -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
   -XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=12
   -XX:LargePageSizeInBytes=256m ...

Java -Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
   -XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2 -XX:-UseAdaptiveSizePolicy
   -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled
   -XX:+CMSParallelSurvivorRemarkEnabled
   -XX:CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly
   -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc ...
A few more GC tuning flags

Source: Word Cloud created by Frank Pavageau in his Devoxx FR 2012 presentation titled "Death by Pauses"
The complete guide to modern GC tuning**

java -Xmx40g

java -Xmx20g

java -Xmx10g

java -Xmx5g
The complete guide to modern GC tuning**

@mjpt777 My GC tuning is well practised. 1) Enable Zing. 2) Open beer. /cc @giltene
Cassandra under heavy load, Intel E5-2690 v4 server

This is 1 msec
Sustainable Throughput:
The throughput achieved while safely maintaining service levels
Warning: results may be too good
A practical real-world example:
Improve Digital (Video Advertising)

- Cassandra cluster running on 6x AWS i3.2xlarge
  - Approx. 80/20 write/read split
  - Data read and written with quorum consistency
  - 6 client machines sending requests collocated in the same AZs

- SLA requirements for read operations:
  - 20ms at 99.9%
  - 50ms at 99.99%
  - 100ms at 99.998% (not a typo, last 9 hard to maintain on AWS)

- HotSpot w/G1: can maintain ~4K TPS before SLA breach
- Zing: can maintain ~21K TPS before SLA breach
Q & A

(Really) Understanding Garbage Collection

@giltene

http://www.azul.com

http://giltene.github.com/jHiccup

http://giltene.github.com/HdrHistogram