Understanding Garbage Collection

in managed runtime environments

Gil Tene, CTO & co-Founder, Azul Systems

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...

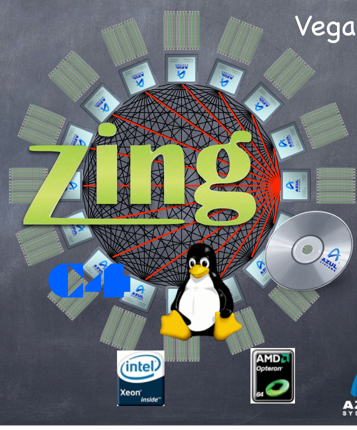
About me: Gil Tene

- co-founder, CTO @Azul Systems
- Have been working on a "think different" GC approaches since 2002
- Created Pauseless & C4 core GC algorithms (Tene, Wolf)
- A Long history building Virtual & Physical Machines, Operating Systems, Enterprise apps, etc...

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

About Azul

- We make scalable Virtual Machines
- Have built "whatever it takes to get job done" since 2002
- 3 generations of custom SMP Multi-core HW (Vega)
- **Now Pure software for commodity x86 (Zing)**
- "Industry firsts" in Garbage collection, elastic memory, Java virtualization, memory scale



High level agenda

- GC fundamentals and key mechanisms
- Some GC terminology & metrics
- Classifying current commercially available collectors
- Why Stop-The-World is a problem
- The C4 collector: What a solution to STW looks like...

Why should you care about GC?

What is Garbage Collection good for?

- Prevalent in modern languages and platforms
  - Java, .NET, Ruby, Scala, Groovy, Clojure, ...
- Productivity, stability
  - Programmers not responsible for freeing and destroying objects
  - Eliminates entire (common) areas of instability, delay, maintenance
- Guaranteed interoperability
  - No "memory management contract" needed across APIs
  - Uncoordinated libraries, frameworks, utilities seamlessly interoperate
- Facilitates practical use of large amounts of memory
  - Complex and intertwined data structures, in and across unrelated components

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 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 does stop for ~1 sec per live GB
  - No, GC does not mean you can't have memory leaks
  - No, those pauses you eliminated from your 20 minute test are not gone
  - ...

Trying to solve GC problems in application architecture is like throwing knives

- You probably shouldn't do it blindfolded
- It takes practice to do it well without hurting people
- You can get very good at it, but do you really want to?
  - Will all the code you leverage be as good as yours?
- Examples of "GC friendly" techniques:
  - Object pooling
  - Off heap storage
  - Distributed heaps
  - ...
  - (In most cases, you end up building your own garbage collector)

Some GC Terminology

A Basic Terminology example: What is a concurrent collector?

- 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
- 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)

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

What's 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")
- 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 & reclaims "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 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

How does the remembered set work?

- Generational collectors require a "Remembered set": a way to track all references into the young generation from the outside
- Each store of a NewGen reference into and OldGen object needs to be intercepted and tracked
- Common technique: "Card Marking"
  - A bit (or byte) indicating a word (or region) in OldGen is "suspect"
- Write barrier used to track references
  - Common technique (e.g. HotSpot): blind stores on reference write
  - Variants: precise vs. imprecise card marking, conditional vs. non-conditional

The typical combos in commercial server 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

Useful terms for discussing garbage collection

- Mutator
  - Your program...
- Parallel
  - Can use multiple CPUs
- Concurrent
  - Runs concurrently with program
- Pause
  - A time duration in which the mutator is not running any code
- Stop-The-World (STW)
  - Something that is done in a pause
- Monolithic Stop-The-World
  - Something that must be done in it's entirety in a single pause
- Generational
  - Collects young objects and long lived objects separately.
- Promotion
  - Allocation into old generation
- Marking
  - Finding all live objects
- Sweeping
  - Locating the dead objects
- Compaction
  - Defragments heap
  - Moves objects in memory
  - Remaps all affected references
  - Frees contiguous memory regions

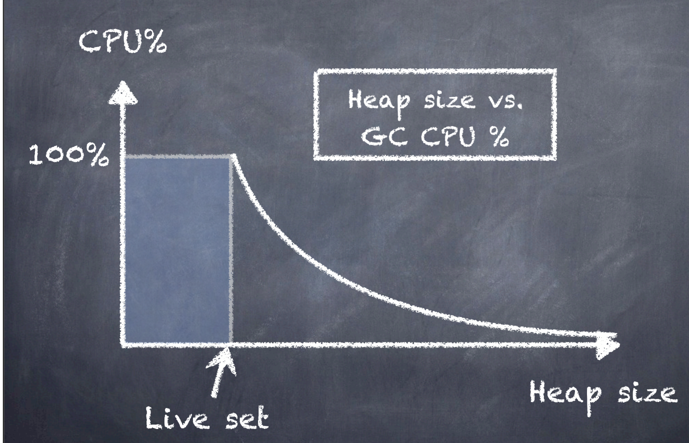
Useful metrics for discussing garbage collection

- Heap population (aka Live set)
  - How much of your heap is alive
- Allocation rate
  - How fast you allocate
- Mutation rate
  - How fast your program updates references in memory
- Heap Shape
  - The shape of the live object graph
  - * Hard to quantify as a metric...
- Object Lifetime
  - How long objects live
- Cycle time
  - How long it takes the collector to free up memory
- Marking time
  - How long it takes the collector to find all live objects
- Sweep time
  - How long it takes to locate dead objects
  - * Relevant for Mark-Sweep
- Compaction time
  - How long it takes to free up memory by relocating objects
  - * Relevant for Mark-Compact

Empty memory and CPU/throughput

Two Intuitive limits

- If we had exactly 1 byte of empty memory at all times, the collector would have to work "very hard", and GC would take 100% of the CPU time
- If we had infinite empty memory, we would never have to collect, and GC would take 0% of the CPU time
- GC CPU % will follow a rough 1/x curve between these two limit points, dropping as the amount of memory increases.



Empty memory needs

(empty memory == CPU power)

- The amount of empty memory in the heap is the dominant factor controlling the <u>amount</u> of GC work
- For both Copy and Mark/Compact collectors, the amount of work per cycle is linear to live set
- The amount of memory recovered per cycle is equal to the amount of unused memory (heap size - live set)
- The collector has to perform a GC cycle when the empty memory runs out
- A Copy or Mark/Compact collector's efficiency doubles with every doubling of the empty memory

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)
- In Mark/Sweep/Compact collectors that pause for sweeping, more empty memory means less frequent but LARGER pauses

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 (<u>mostly</u> 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
- 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

The typical combos in commercial server JVMs

- Young generation usually uses a copying collector
  - Young generation is usually monolithic, stop-the-world
- Old generation usually uses a Mark/Sweep/Compact collector
  - 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.

The "Application Memory Wall"

or: Why stop-the-world garbage collection is a problem

Memory use

How many of you use heap sizes of:

- more than ½ GB?
- more than 1 GB?
- more than 2 GB?
- more than 4 GB?
- more than 10 GB?
- more than 20 GB?

Reality check: servers in 2012

- Retail prices, major web server store (US $, July 2012)

16 vCore, 96GB server	≈	$5K
16 vCore, 256GB server	≈	$9K
24 vCore, 384GB server	≈	$14K
32 vCore, 1TB server	≈	$35K

- Cheap (< $1/GB/Month), and roughly linear to ~1TB
- 10s to 100s of GB/sec of memory bandwidth

The Application Memory Wall A simple observation:

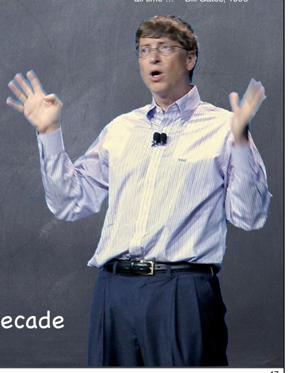
- Application instances appear to be unable to make effective use of modern server memory capacities
- The size of application instances as a % of a server's capacity is rapidly dropping

How much memory do applications need?

- "640KB ought to be enough for anybody"

WRONG!

"I've said some stupid things and some wrong things, but not that. No one involved in computers would ever say that a certain amount of memory is enough for all time …" - Bill Gates, 1996



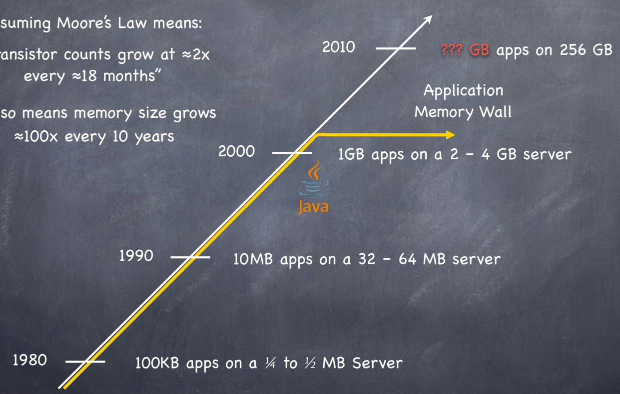
- So what's the right number?
  - 6,400K?
  - 64,000K?
  - 640,000K?
  - 6,400,000K?
  - 64,000,000K?
- There is no right number
- Target moves at 50x-100x per decade

"Tiny" application history

Assuming Moore's Law means:

"transistor counts grow at ≈2x every ≈18 months"

It also means memory size grows ≈100x every 10 years



* "Tiny": would be "silly" to distribute

What is causing the Application Memory Wall?

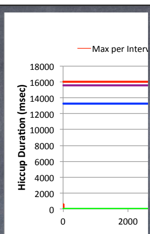
- Garbage Collection is a clear and dominant cause
- There seem to be practical heap size limits for applications with responsiveness requirements
- [Virtually] All current commercial JVMs will exhibit a multi-second pause on a normally utilized 2-6GB heap.
  - It's a question of "When" and "How often", not "If".
  - GC tuning only moves the "when" and the "how often" around
- Root cause: The link between scale and responsiveness

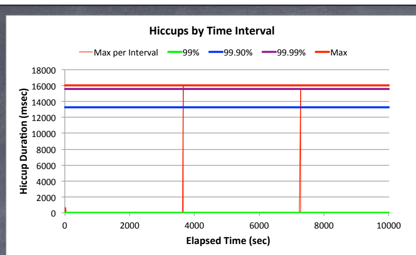
What quality of GC is responsible for the Application Memory Wall?

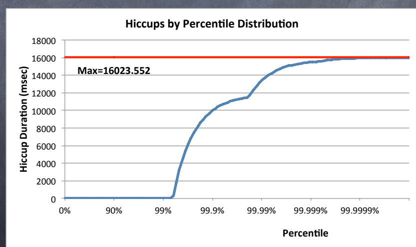
- It is NOT about overhead or efficiency:
  - CPU utilization, bottlenecks, memory consumption and utilization
- It is NOT about speed
  - Average speeds, 90%, 95% speeds, are all perfectly fine
- It is NOT about minor GC events (right now)
  - GC events in the 10s of msec are usually tolerable for most apps
- It is NOT about the frequency of very large pauses
- It is ALL about the worst observable pause behavior
  - People avoid building/deploying visibly broken systems

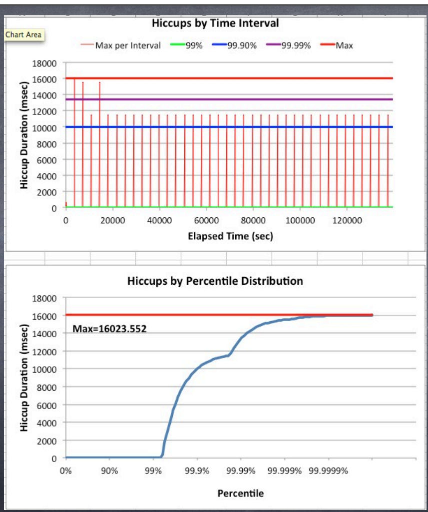
Monolithic-STW GC Problems

One way to deal with Monolithic-STW GC









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"

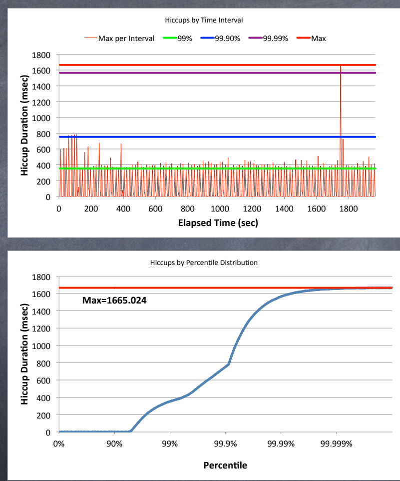
- "Typical pauses in the tens of milliseconds"

Translation: "Some pauses are much longer than tens of milliseconds"

Actually measuring things

(e.g. jHiccup)

Incontinuities in Java platform execution



Getting past a monolithic-STW Garbage Collection world

We need to solve the right problems

- Scale is artificially limited by responsiveness
- Responsiveness must be unlinked from scale:
  - Heap size, Live Set size, Allocation rate, Mutation rate
  - Transaction Rate, Concurrent users, Data set size, etc.
  - Responsiveness must be continually sustainable
  - Can't ignore "rare" events
- Eliminate all Stop-The-World Fallbacks
  - At modern server scales, any STW fall back is a failure

The things that seem "hard" to do in GC

- Robust concurrent marking
  - References keep changing
  - Multi-pass marking is sensitive to mutation rate
  - Weak, Soft, Final references "hard" to deal with concurrently
- [Concurrent] Compaction…
  - It's not the moving of the objects…
  - It's the fixing of all those references that point to them
  - How do you deal with a mutator looking at a stale reference?
  - If you can't, then remapping is a [monolithic] STW operation
- Young Generation collection at scale
  - Young Generation collection is generally monolithic, Stop-The-World
  - Young generation pauses are only small because heaps are tiny
  - A 100GB heap will regularly have several GB of live young stuff…

The problems that need solving

(areas where the state of the art needs 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 Compacting 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

AZUL SYSTEMS

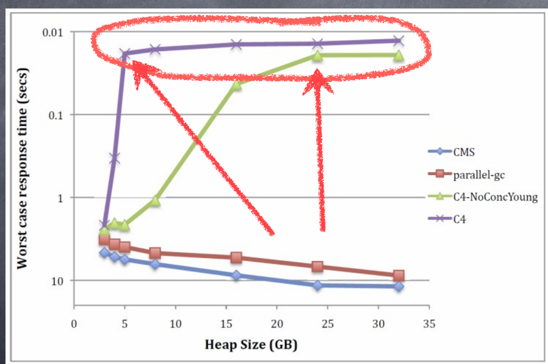
C4 algorithm highlights

- 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

AZUL SYSTEMS

Benefits

Sample responsiveness behavior



- SpecJBB + Slow churning 2GB LRU Cache
- Live set is ~2.5GB across all measurements
- Allocation rate is ~1.2GB/sec across all measurements

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:+CMSConcurrentMTEnabled
     -XX:+CMSParallelRemarkEnabled -XX:+CMSParallelSurvivorRemarkEnabled
     -XX:CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly
     -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC –Xnoclassgc …
```

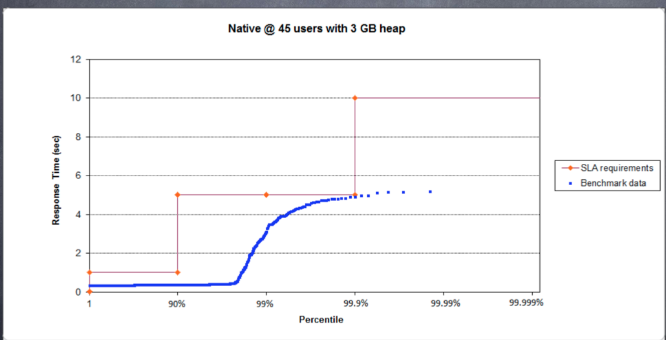

The complete guide to Zing GC tuning

java -Xmx40g

Sustainable Throughput: The throughput achieved while safely maintaining service levels

Instance capacity test: "Fat Portal"

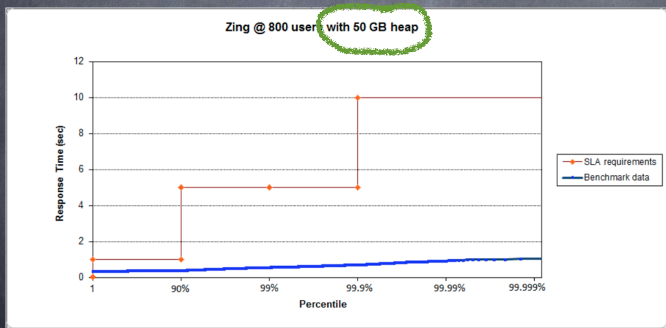
HotSpot CMS: Peaks at ~ 3GB / 45 concurrent users



* LifeRay portal on JBoss @ 99.9% SLA of 5 second response times

Instance capacity test: "Fat Portal"

C4: still smooth @ 800 concurrent users

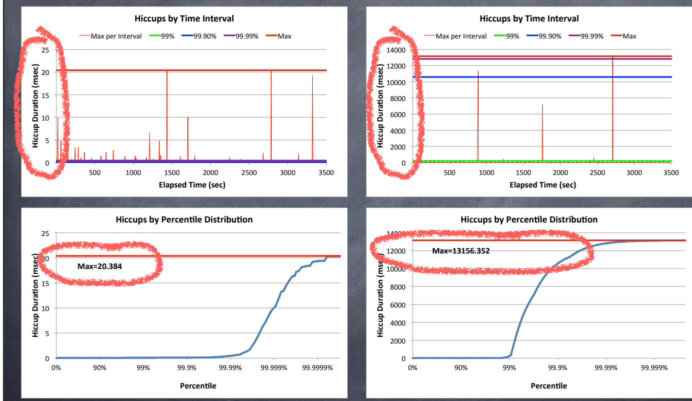


Fun with jHiccup

Charles Nutter @headius 20 Jan
jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)
Retweeted by Gil Tene

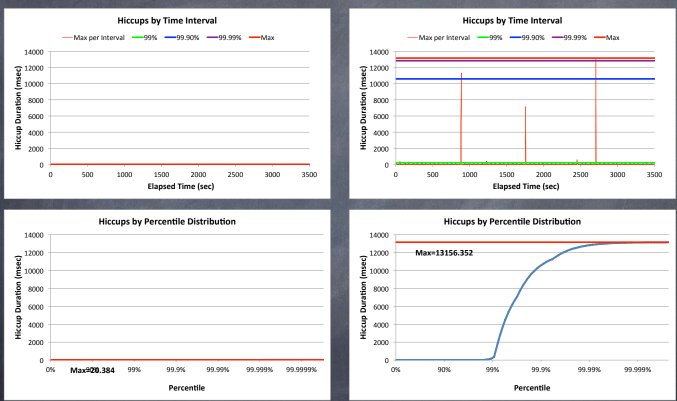
Zing 5, 1GB in an 8GB heap

Oracle HotSpot CMS, 1GB in an 8GB heap



Zing 5, 1GB in an 8GB heap

Oracle HotSpot CMS, 1GB in an 8GB heap



Q & A

GC :

G. Tene, B. Iyengar and M. Wolf
C4: The Continuously Concurrent Compacting Collector
In Proceedings of the international symposium on Memory management, ISMM'11, ACM, pages 79-88

Jones, Richard; Hosking, Antony; Moss, Eliot (25 July 2011).
The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Press. ISBN 1420082795.

jHiccup:

http://www.azulsystems.com/dev_resources/jhiccup