The Application Memory Wall

Thoughts on the state of the art in Garbage Collection

Gil Tene, CTO & co-Founder, Azul Systems

About me: Gil Tene

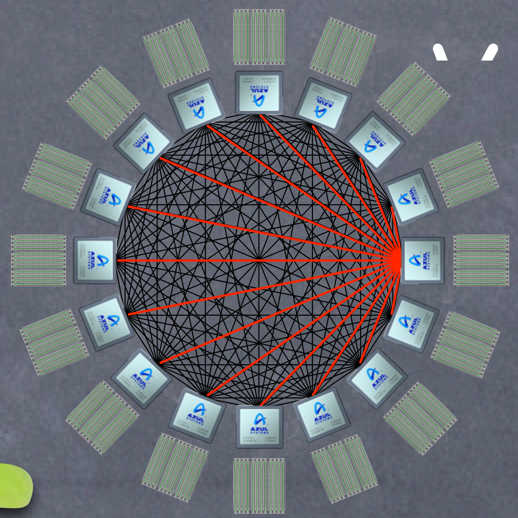
- co-founder, CTO @Azul Systems
- Have been working on a "think different" GC approaches since 2002
- Created Pauseless & C4 core GC algorithms (Tene, Wolf)
- A Long history building Virtual & Physical Machines, Operating Systems, Enterprise apps, etc...

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

AZUL SYSTEMS

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

- The original Memory Wall, and some others
- The Application Memory Wall
- Garbage Collection discussion
- How can we break through the wall?
- The C4 collector: What an actual solution looks like...

The original Memory Wall + others

The original Memory Wall

- "Hitting the Memory Wall: Implications of the Obvious"
  - Wm. A. Wolf, Sally A. McKey, Computer Science Report No. CS-94-48, December 1994
  - Widely Quoted and referenced since
- CPUs are getting faster at Moore's law rates, but:
  - Memory bandwidth is growing at much slower pace
  - Memory latency is improving at much slower pace
- Anticipated result: Applications will become memory bandwidth bound
- Prediction [1994]: Wall will be hit within a decade

AZUL SYSTEMS

Additional predicted walls

- "Frequency Wall"
  - CPU Frequency will not keep growing at same rate, limiting single threaded speed growth
  - Hardware shift in mid-2000s: moved to multi-core
- "Power Wall"
  - The ability to cool chips will limit speed and frequency
  - Hardware shift in mid-2000s: multi-core, power-efficient designs
- "Concurrency Wall"
  - Limited ability to make use of many cores in common programs
  - Much work being done to improve concurrency

Have these walls been hit?

- Do **Applications** actually hit any of these walls?
- Predictions: 1994 … 2004: Majority of applications should have hit against the predicted walls by now…
- Reality: We came close, and backed [way] off
  - Application instances don't use available memory bandwidth
  - Application instances don't use available memory capacity
  - Application instances don't use available cores
- We hit another wall…
- The "Application Memory Wall"

The "Application Memory Wall"

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?
- more than 50 GB?

AZUL SYSTEMS

Reality check: servers in 2011

- Retail prices, major web server store (US $, Nov. 2011)

24 vCore, 96GB server	≈ $5K
32 vCore, 256GB server	≈ $14K
64 vCore, 512GB server	≈ $27K
80 vCore, 1TB server	≈ $49K

- Cheap (≈ $1.2/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

Maybe 1+ to 4+ GB is simply enough?

- We hope not (or we'll all have to look for new jobs soon)
- Plenty of evidence of pent up demand for more heap:
  - Common use of lateral scale across machines
  - Common use of "lateral scale" within machines
  - Use of "external" memory with growing data sets
    - Databases certainly keep growing
    - External data caches (memcache, JCache, Data grids)
  - Continuous work on the never ending distribution problem
    - More and more reinvention of NUMA
    - Bring data to compute, bring compute to data

How much memory do applications need?

- "640KB ought to be enough for anybody"

"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

WRONG!

- So what's the right number?

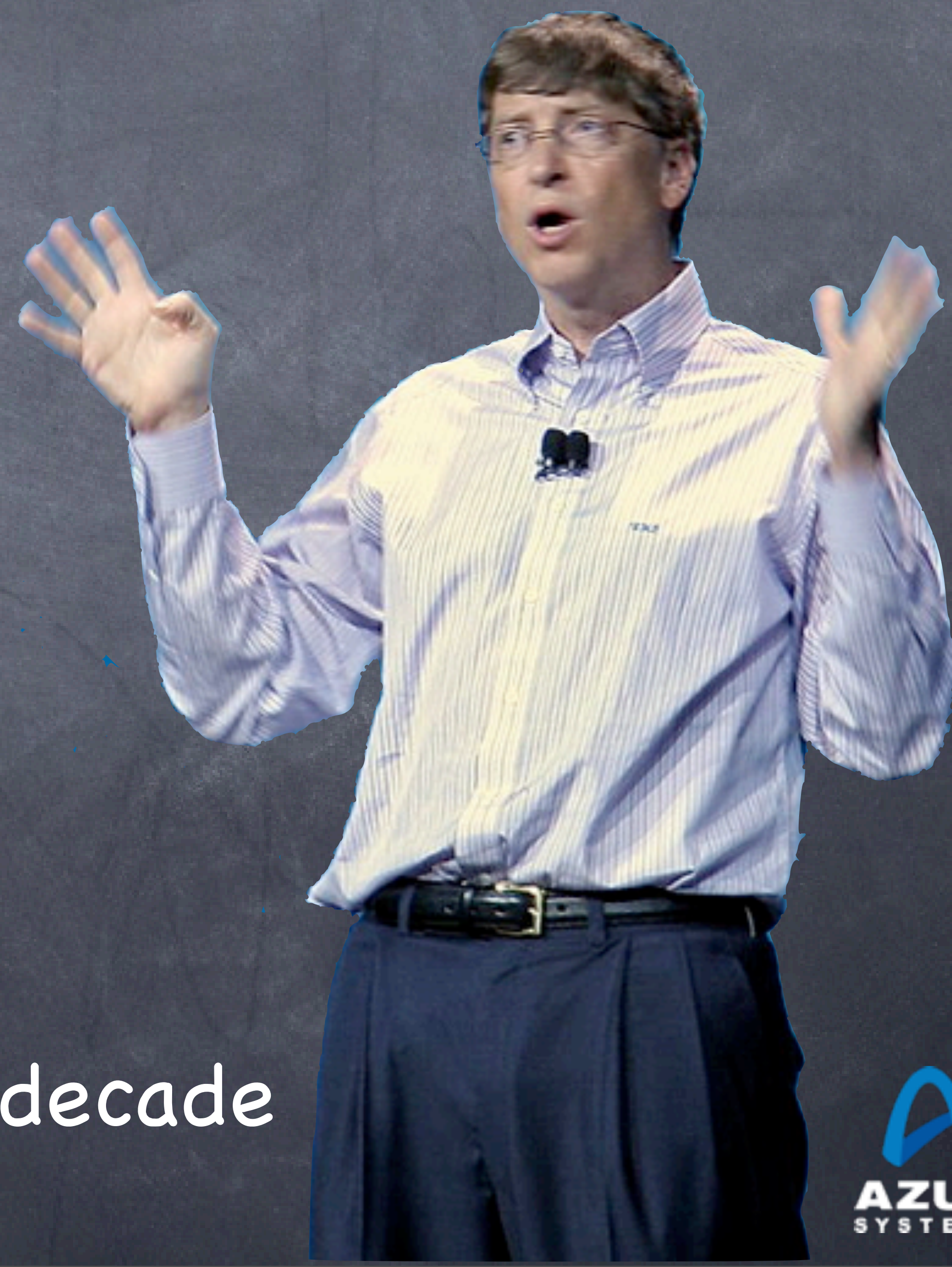
  6,400K?

  64,000K?

  640,000K?

  6,400,000K?

  64,000,000K?



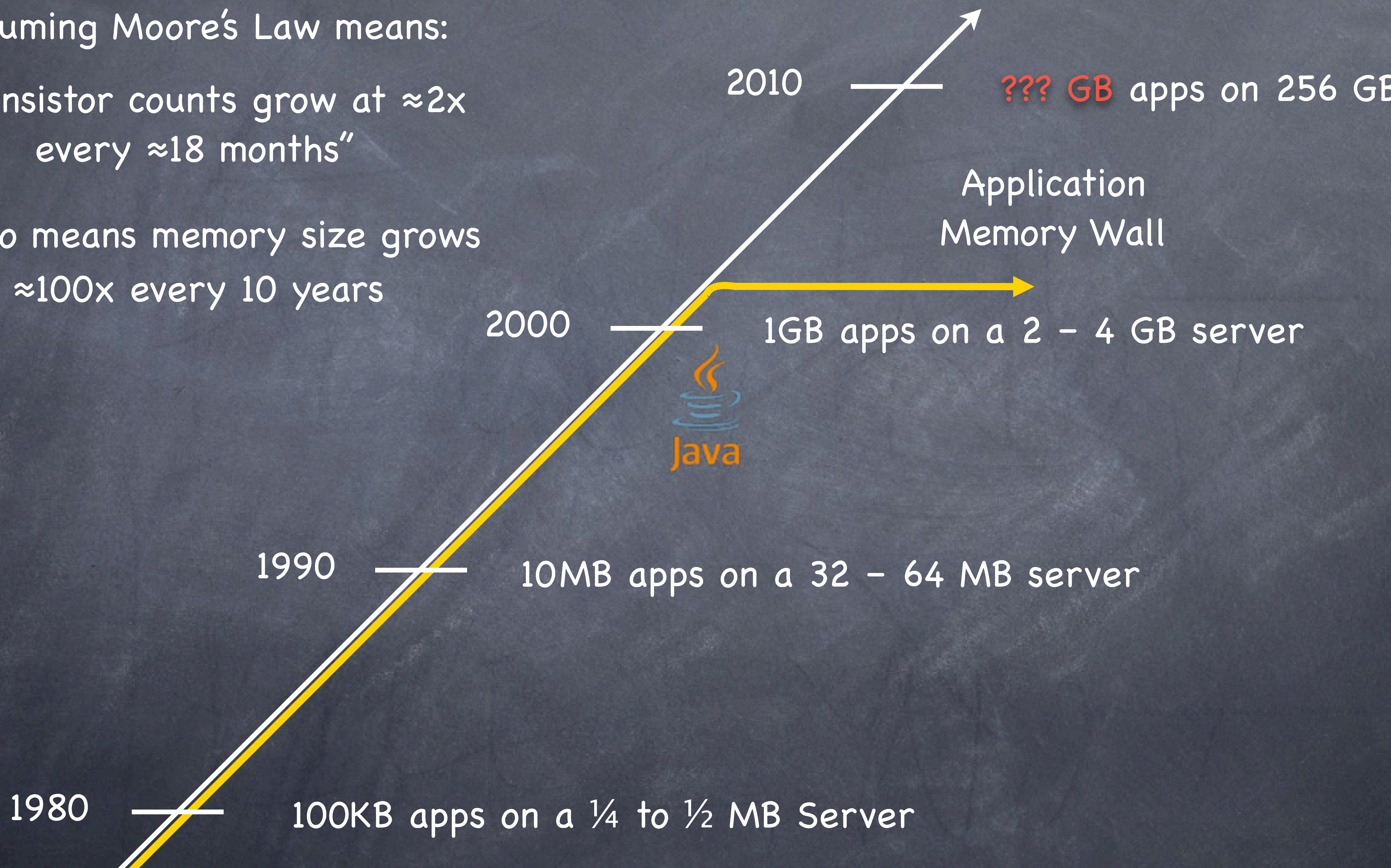
- There is no right number
- Target moves at 50x-100x per decade

“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 2-4GB 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%, 99% 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

GC Discussion

Framing the discussion: Garbage Collection at modern server scales

- Modern Servers have 100s of GB of memory
- Each modern x86 core (when actually used) produces garbage at a rate of ¼ - ½ GB/sec +
- That's many GB/sec of allocation in a server
- Monolithic stop-the-world operations are the cause of the current Application Memory Wall

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)

What's common to all 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)

Generational Collection

- Generational Hypothesis: most objects die young
- Focus collection efforts on young generation:
  - Usually using a copying collector
  - Promote objects that live long enough to old generation
  - All known young generation collectors compact
- Tends to be (order of magnitude) more efficient
  - Great way to keep up with high allocation rate
- ALL commercial JVMs use generational collectors
  - Necessary for keeping up with processor throughput

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

AZUL SYSTEMS

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

GC Problems

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 see muli-GB of live young stuff...

AZUL SYSTEMS

Delaying the inevitable

- 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
    - 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

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.

How can we break through the Application Memory Wall?

We need to solve the right problems

- Focus on the causes of the Application Memory Wall
  - Root cause: Scale is artificially limited by responsiveness
- Responsiveness must be unlinked from scale
  - Heap size, Live Set size, Allocation rate, Mutation rate
- 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 problems that need solving

(areas where the state of the art needs improvement)

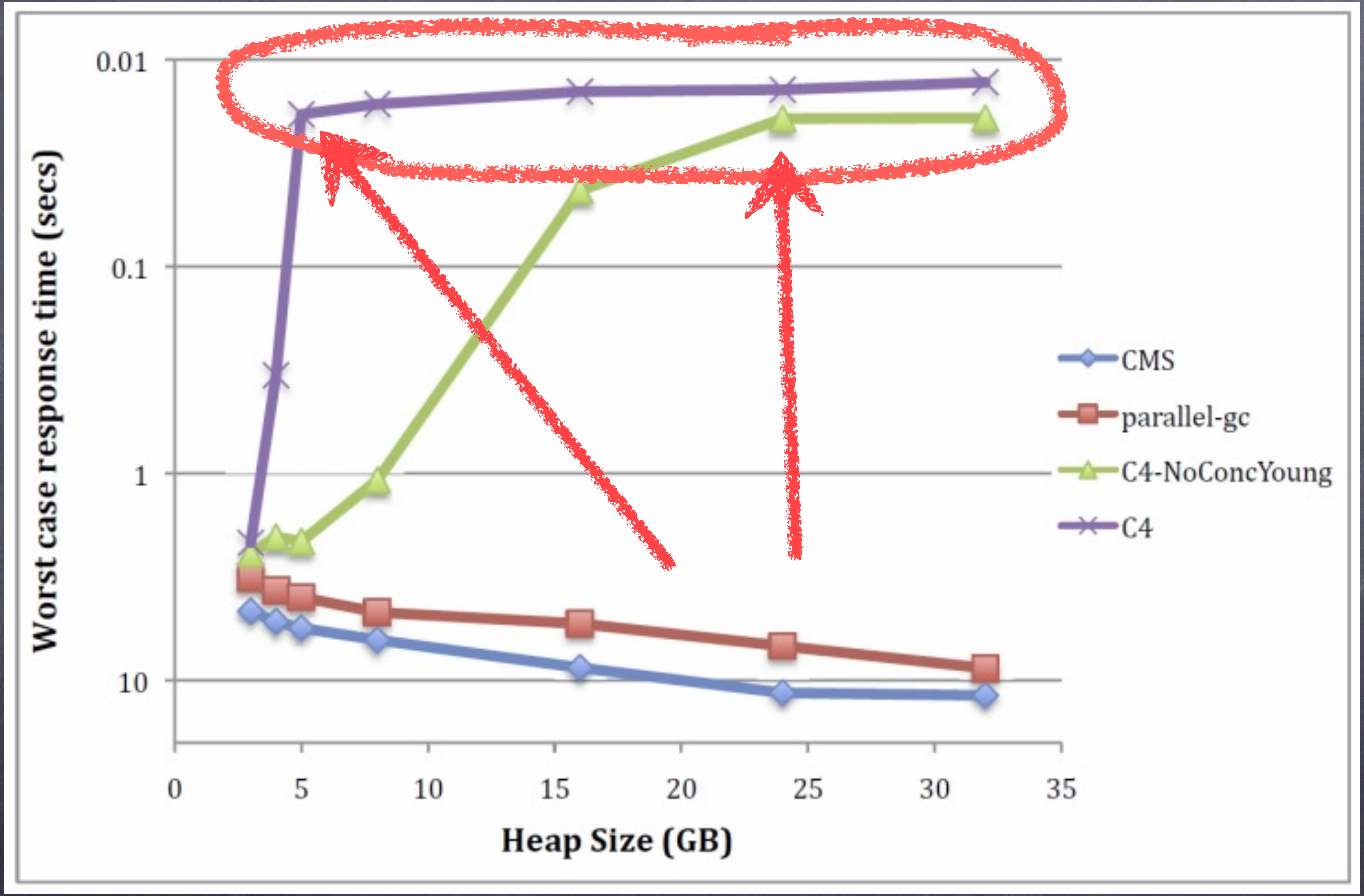
- Robust Concurrent Marking
  - In the presence of high mutation and allocation rates
  - Cover modern runtime semantics (e.g. weak refs)
- Compaction that is not monolithic-stop-the-world
  - Stay responsive while compacting many-GB heaps
  - Must be robust: not just a tactic to delay STW compaction
  - [current "incremental STW" attempts fall short on robustness]
- Non-monolithic-stop-the-world Generational collection
  - Stay responsive while promoting multi-GB data spikes
  - Concurrent or "incremental STW" may be both be ok
  - Surprisingly little work done in this specific area

Azul's "C4" Collector

Continuously Concurrent Compacting Collector

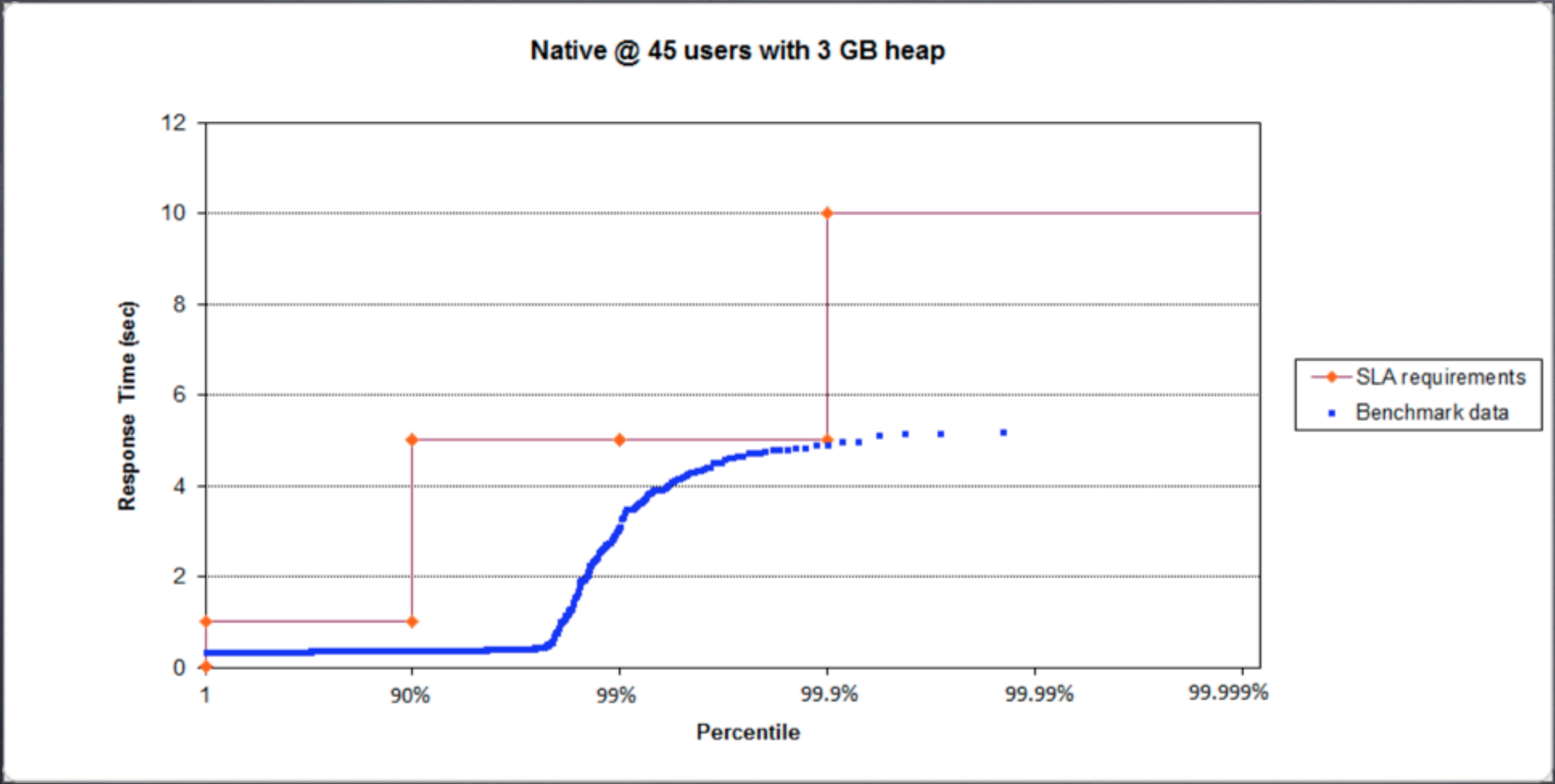
- Concurrent, compacting new generation
- Concurrent, compacting old generation
- Concurrent guaranteed-single-pass marker
  - Oblivious to mutation rate
  - Concurrent ref (weak, soft, final) processing
- Concurrent Compactor
  - Objects moved without stopping mutator
  - References remapped without stopping mutator
  - Can relocate entire generation (New, Old) in every GC cycle
- No stop-the-world fallback
  - Always compacts, and always does so concurrently

Sample responsiveness improvement



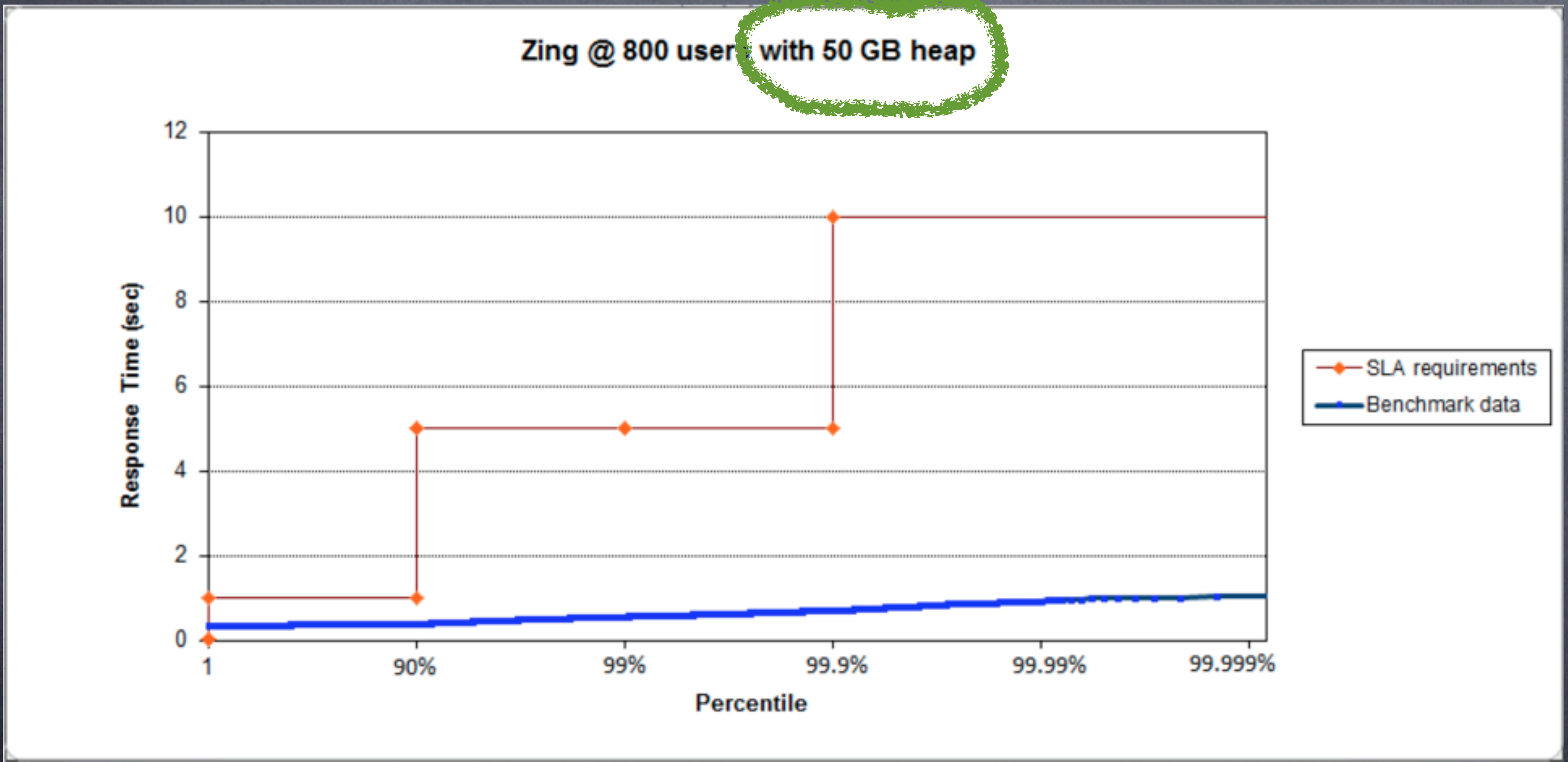
- SpecJBB + Slow churning 2GB LRU Cache
- Live set is ~2.5GB across all measurements
- Allocation rate is ~1.2GB/sec across all measurements

Instance capacity test: "Fat Portal"
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



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

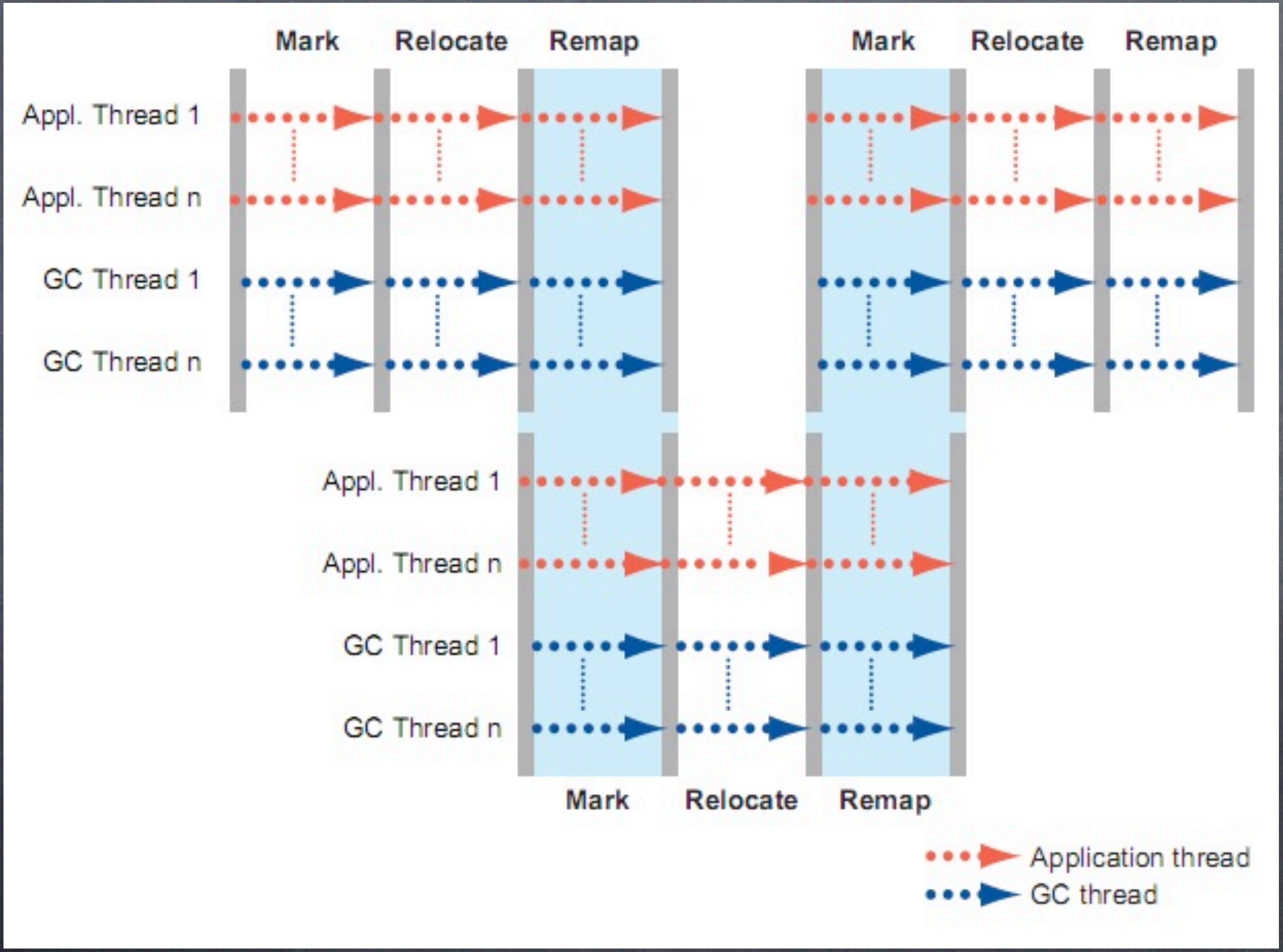
java -Xmx40g

C4 Algorithm fundamentals

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

The C4 GC Cycle



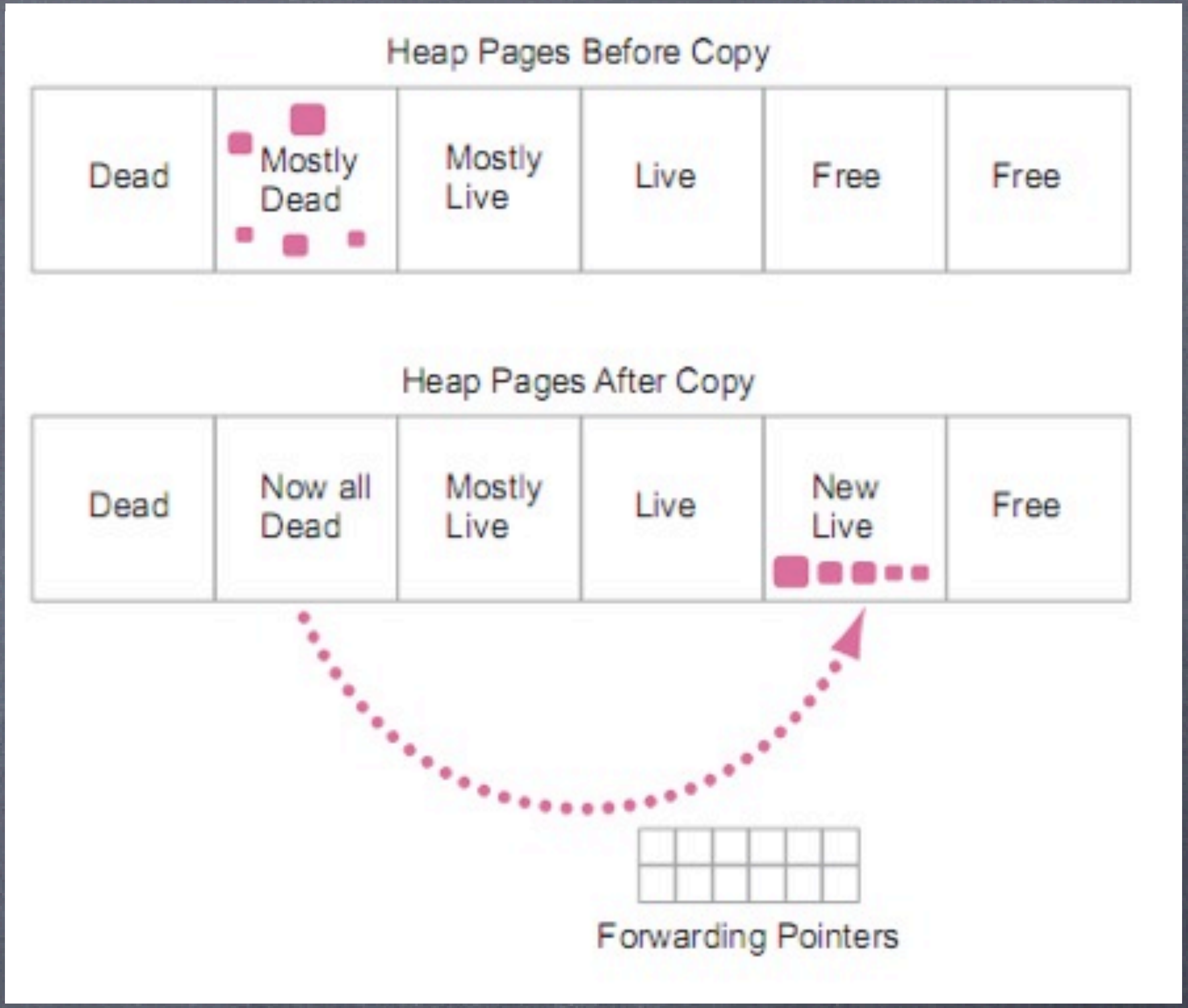
Mark Phase

- Mark phase finds all live objects in the Java heap
- Concurrent, predictable: always complete in a single pass
- Uses LVB to defeat concurrent marking races
  - Tracks object references that have been traversed by using an "NMT" (not marked through) metadata state in each object reference
  - Any access to a not-yet-traversed reference will trigger the LVB
  - Triggered references are queued on collector work lists, and reference NMT state is corrected
  - "Self healing" corrects the memory location that the reference was loaded from
- Marker tracks total live memory in each memory page
  - Compaction uses this to go after the sparse pages first

  (But each cycle will tend to compact the entire heap...)

Relocate Phase

- Compacts to reclaim heap space occupied by dead objects in "from" pages without stopping mutator
- Protects "from" pages.
- Uses LVB to support concurrent relocation and lazy remapping by triggering on any access to references to "from" pages
- Relocates any live objects to newly allocated "to" pages
- Maintains forwarding pointers outside of "from" pages
- Virtual "from" space cannot be recycled until all references to relocated objects are remapped

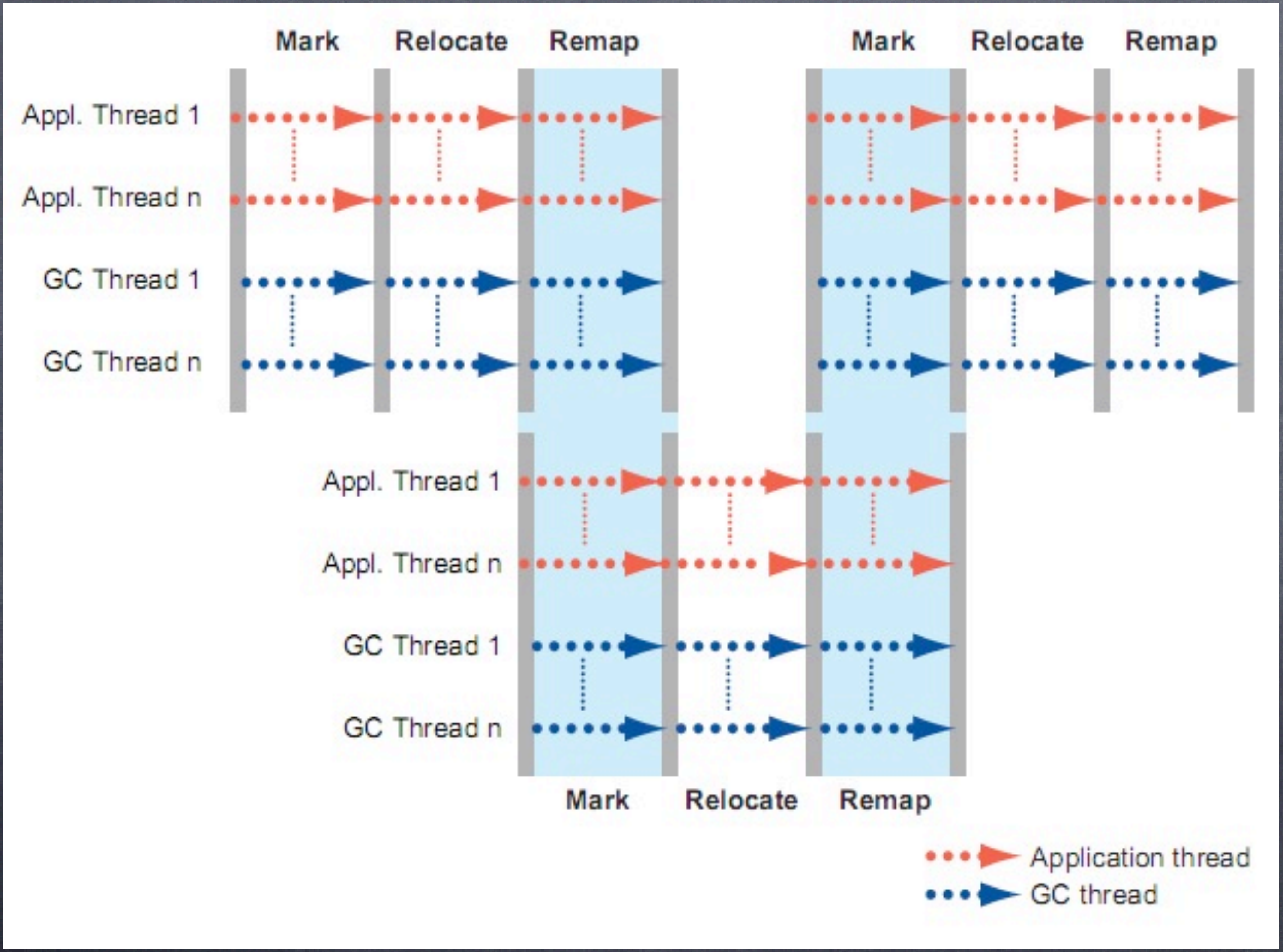


- "Quick Release": Physical memory can be immediately reclaimed, and used to feed further compaction or allocation

Remap Phase

- Scans all live objects in the heap
- Looks for references to previously relocated objects, and updates ("remaps") them to point to the new object locations
- Uses LVB to support lazy remapping
  - Any access to a not-yet-remapped reference will trigger the LVB
  - Triggered references are corrected to point to the object's new location by consulting forwarding pointers
  - "Self healing" corrects the memory location the reference was loaded from
- Overlaps with the next mark phase's live object scan
  - Mark & Remap are executed as a single pass

The C4 GC Cycle



Summary

- The Application Memory Wall is HERE, NOW
  - Driven by detrimental link between scale and responsiveness
- Solving a handful of problems can lead to breakthrough
  - Robust Concurrent Marking
  - [Concurrent] Compaction
  - non-monolithic STW young generation collection
  - All at modern server-scales
- Solving it will [hopefully] allow application to resume their natural rate of consuming computer capacity

Implications of breaking past the Application Memory Wall

- Improve quality of current systems:
  - Better & consistent response times, stability & availability
  - Reduce complexity, time to market, and cost
- Scale Better:
  - Large or variable number of concurrent users
  - High or variable transaction rates
  - Large data sets
- Change how things are done:
  - Aggressive Caching, in-memory data processing
  - Multi-tenant, SaaS, PaaS
  - Cloud deployments
- Build applications that were not possible before...

How can we break through the Application Memory Wall?

Simple: Deploy Zing 5.0 on Linux

Q & A

How can we break through the Application Memory Wall?

Simple: Deploy Zing 5.0 on Linux

http://www.azulsystems.com

AZUL SYSTEMS