Understanding Java Garbage Collection

Gil Tene

High level agenda

- Some GC fundamentals, terminology & mechanisms
- Classifying currently available collectors
- Why Stop-The-World is a problem
- 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 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 like to nag people about how they measure and think about latency

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

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 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
  - ...

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
- Monolithic: "all in one shot"; the opposite of Incremental
- 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
- "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" 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 & 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 collector in which 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

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

- “Much of the heap is not popular”
- 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 fix 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
  - Young generation pauses are only small because heaps are tiny
  - A 200GB heap will regularly have several GB of live young stuff…

Classifying common collectors

The typical combos in 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

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?
- ☞ more than 50 GB?
- ☞ more than 100 GB?

Reality check: servers in 2015

- Retail prices, major web server store (US $, circa 2015)

24 vCore, 128GB server	≈	$4K
32 vCore, 256GB server	≈	$7K
32 vCore, 512GB server	≈	$11K
64 vCore, 1TB server	≈	$24K

- Cheap (< $1/GB/Month), and roughly linear to ~1TB
- The basic building blocks in the cloud…

Current (2015) cloud stuff

Linux RHEL SLES Windows Windows with SQL Standard

Windows with SQL Web

Region: US East (N. Virginia)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
Compute Optimized - Current Generation					
c4.large	2	8	3.75	EBS Only	$0.116 per Hour
c4.xlarge	4	16	7.5	EBS Only	$0.232 per Hour
c4.2xlarge	8	31	15	EBS Only	$0.464 per Hour
c4.4xlarge	16	62	30	EBS Only	$0.928 per Hour
c4.8xlarge	36	132	60	EBS Only	$1.856 per Hour
Memory Optimized - Current Generation					
r3.large	2	6.5	15	1 x 32 SSD	$0.175 per Hour
r3.xlarge	4	13	30.5	1 x 80 SSD	$0.35 per Hour
r3.2xlarge	8	26	61	1 x 160 SSD	$0.7 per Hour
r3.4xlarge	16	52	122	1 x 320 SSD	$1.4 per Hour
r3.8xlarge	32	104	244	2 x 320 SSD	$2.8 per Hour

Current (2015) cloud stuff

Microsoft Azure

SALES 1-800-867-1389 | MY ACCOUNT | PORTAL | Search

Why Azure | Products | Documentation | Pricing | Partners | Blog | Resources | Support | FREE TRIAL

INSTANCE	CORES	RAM	DISK SIZES	PRICE
G1	2	28 GB	384 GB	$0.61/hr (~$454/mo)
G2	4	56 GB	768 GB	$1.22/hr (~$908/mo)
G3	8	112 GB	1,536 GB	$2.44/hr (~$1,815/mo)
G4	16	224 GB	3,072 GB	$4.88/hr (~$3,631/mo)
G5	32	448 GB	6,144 GB	$8.78/hr (~$6,532/mo)

GB is represented using 1024^3 bytes sometimes referred to as Gibibyte, or base 2 definition. When comparing sizes that use different base systems, remember that base 2 sizes may appear smaller than base 10 but for any specific size, a base 2 system provides more capacity than a base 10 system, because 1024^3 is greater than 1000^3

The Application Memory Wall
A simple observation:

- Application instances appear to be unable to make effective use of modern server memory capacities

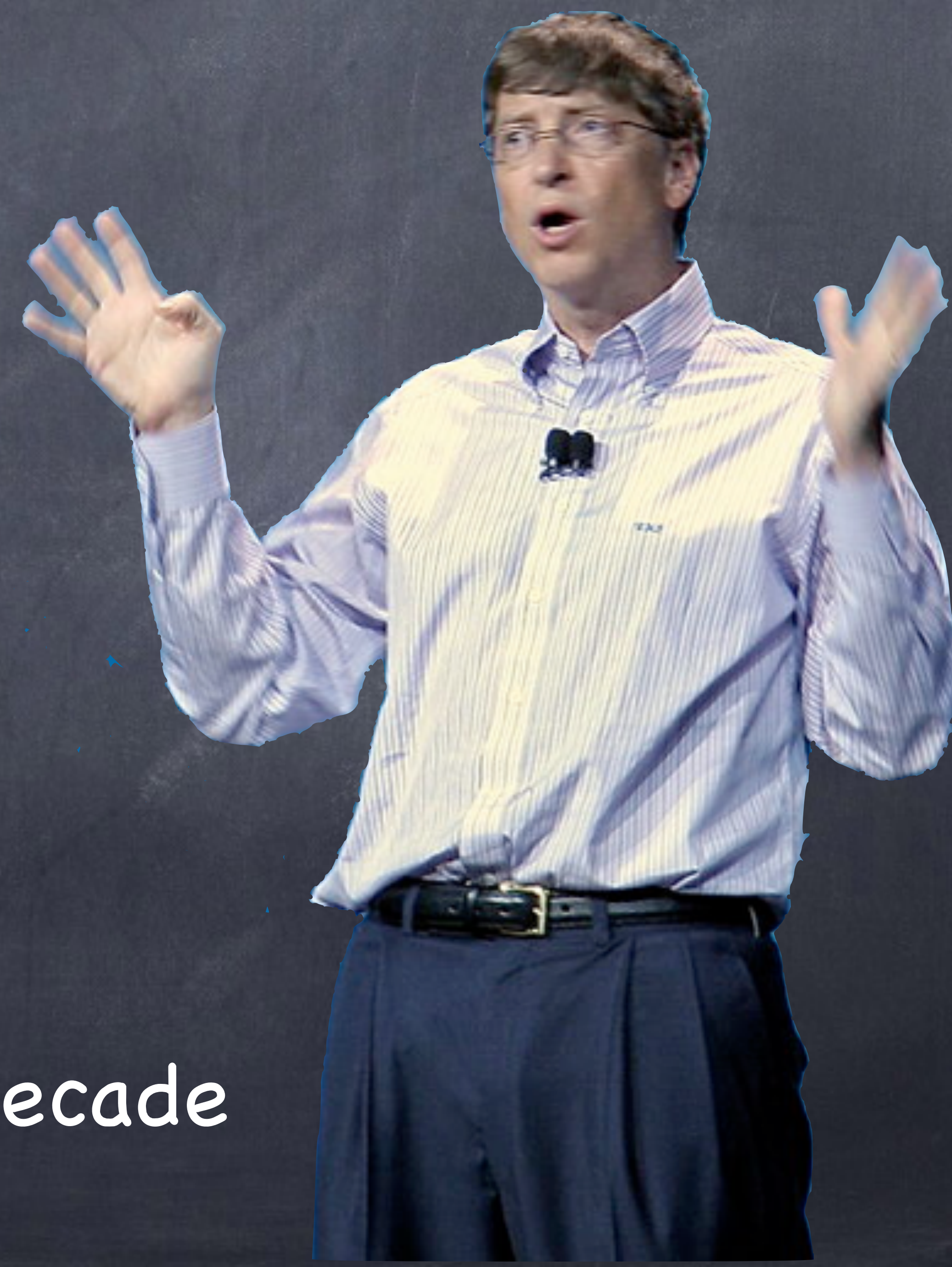
- The size of application instances as a % of a server's capacity is rapidly dropping

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?

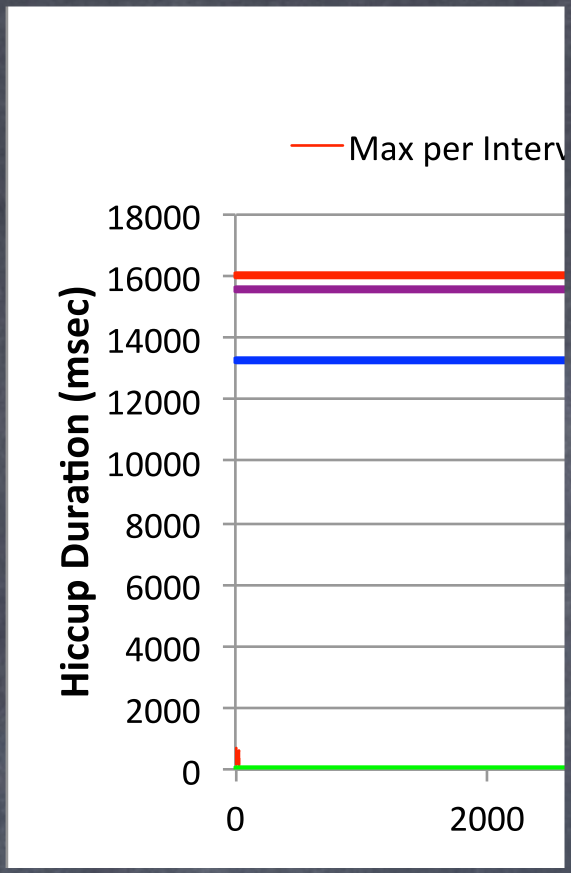
- There is no right number

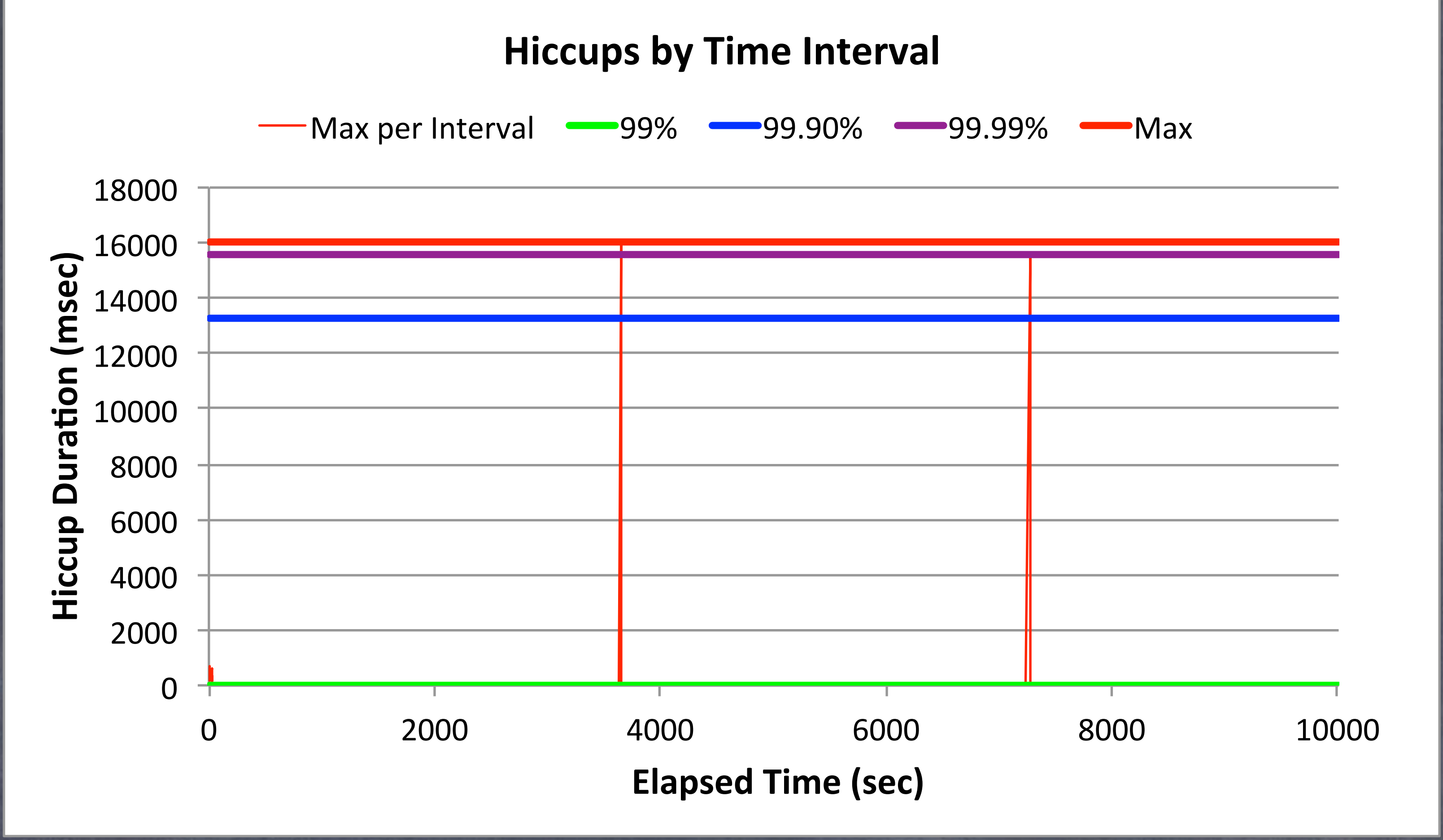
- Target moves at 50x-100x per decade

Monolithic-STW GC Problems

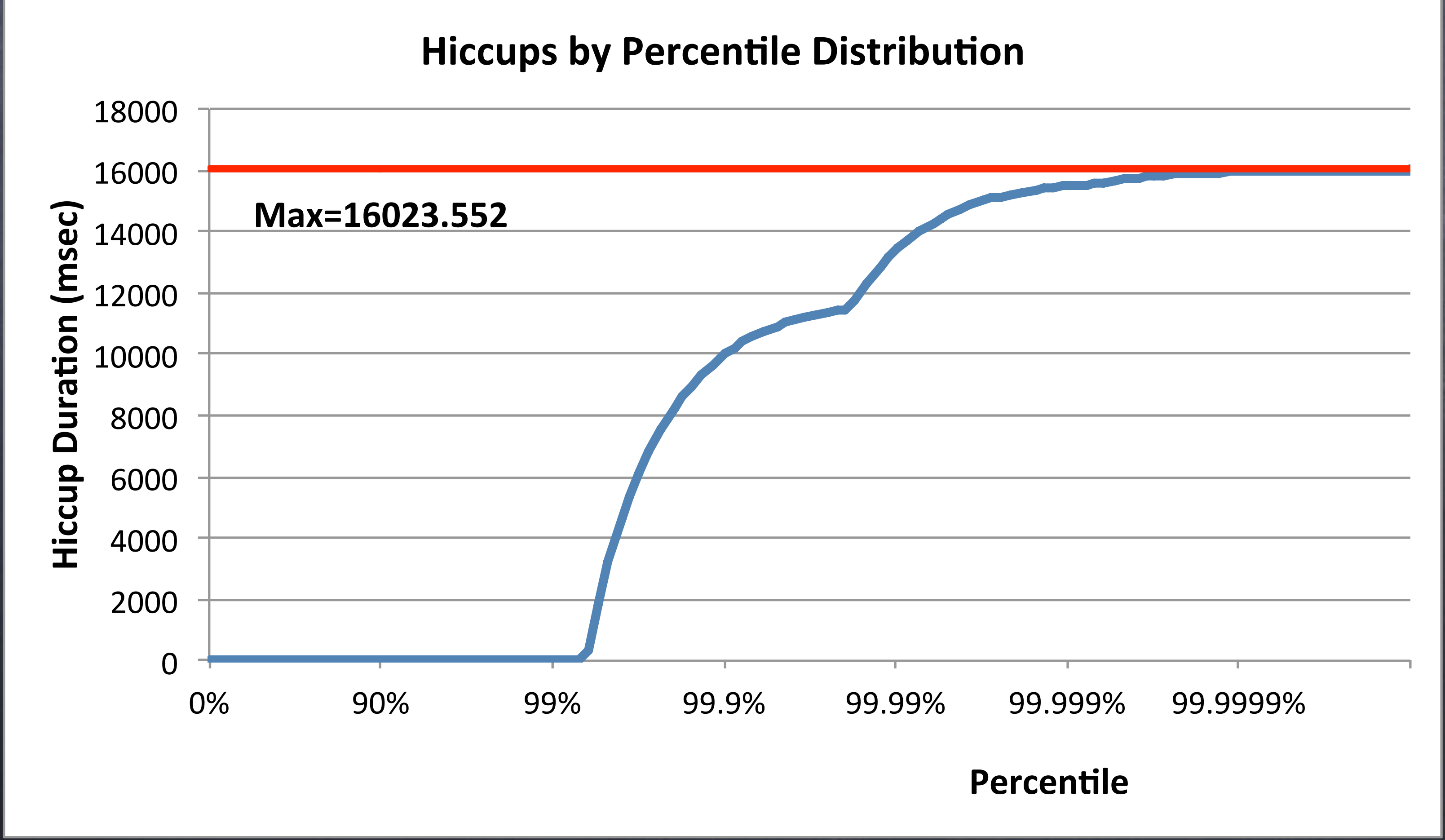
One way to deal with Monolithic-STW GC

640KB/MB should be enough…

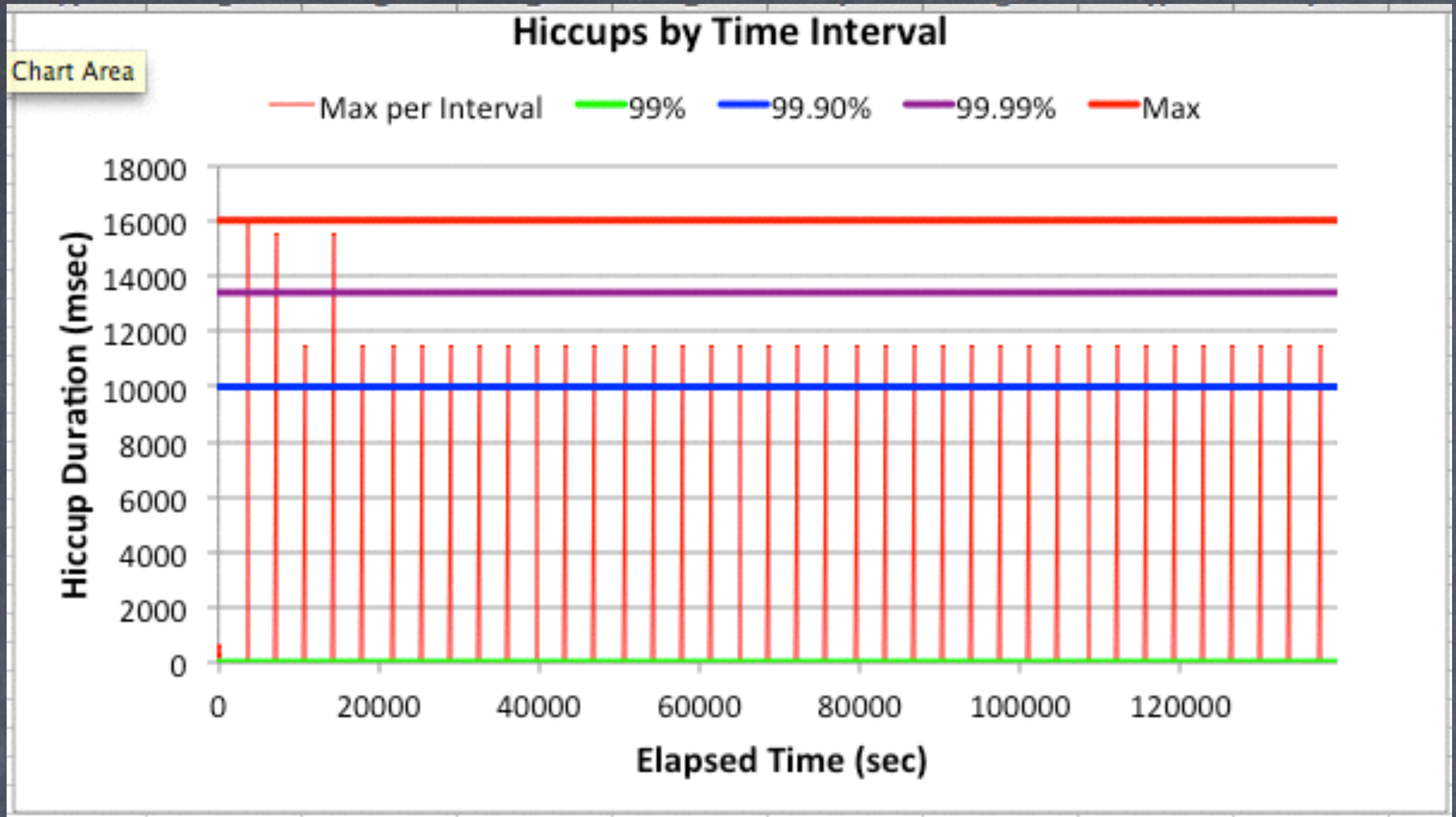

Max per Interv
Hiccup Duration (msec)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2000

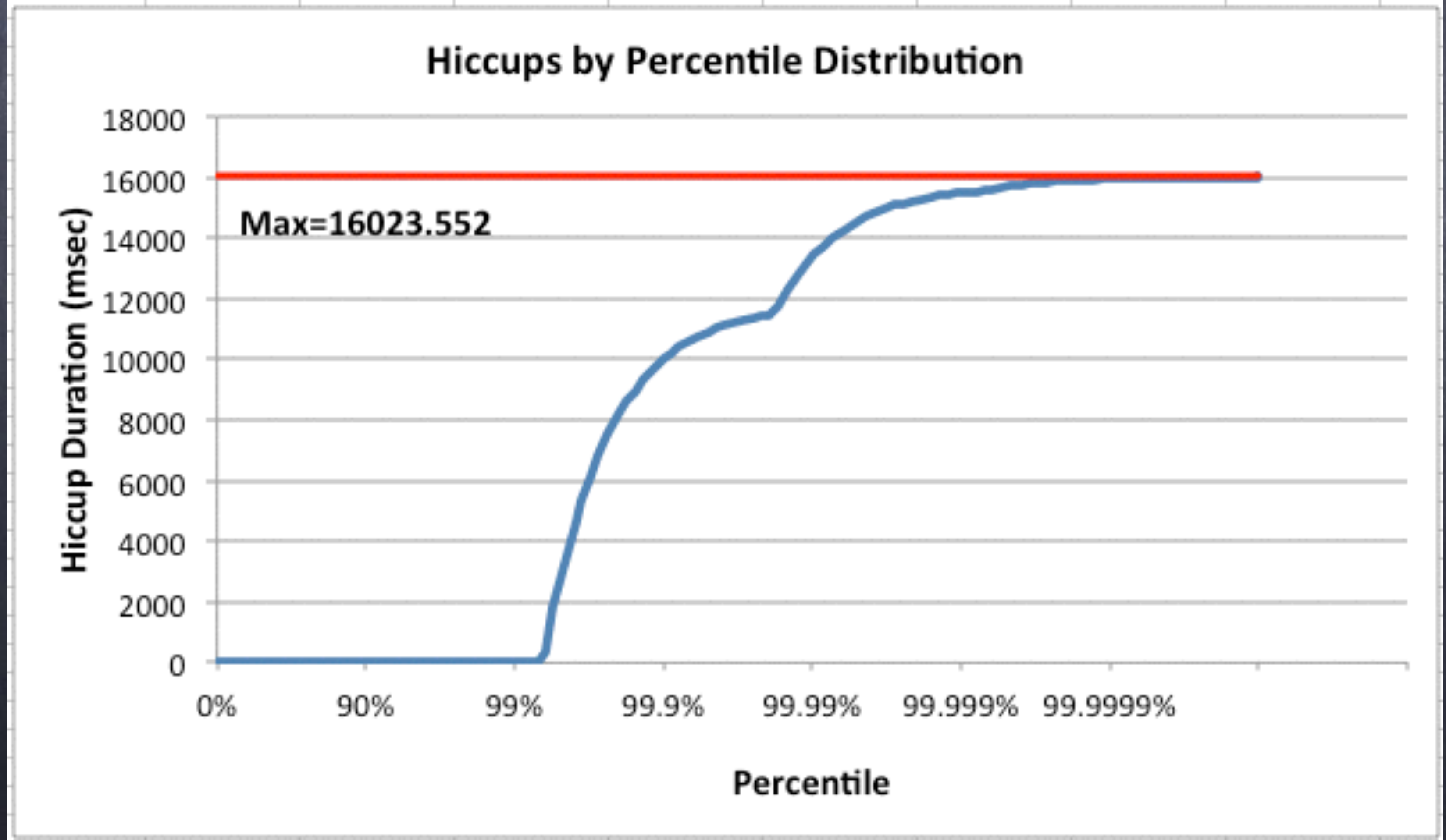


Hiccups by Time Interval
Max per Interval
99%
99.90%
99.99%
Max
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Hiccup Duration (msec)
0
2000
4000
6000
8000
10000
Elapsed Time (sec)



Hiccups by Percentile Distribution
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Hiccup Duration (msec)
Max=16023.552
0%
90%
99%
99.9%
99.99%
99.999%
99.9999%
Percentile





AZUL SYSTEMS

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"

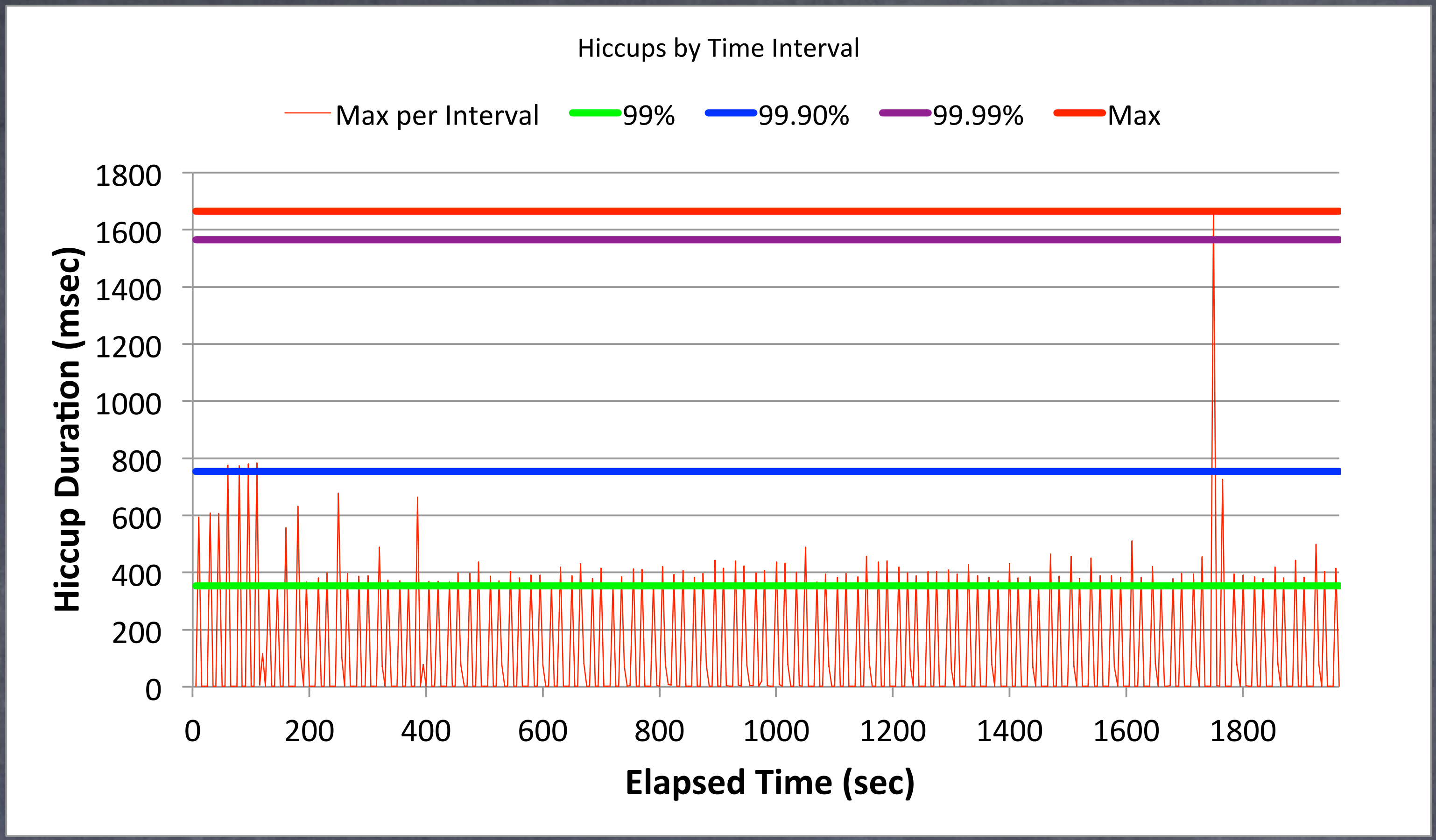
- "Typical pauses in the tens of milliseconds"

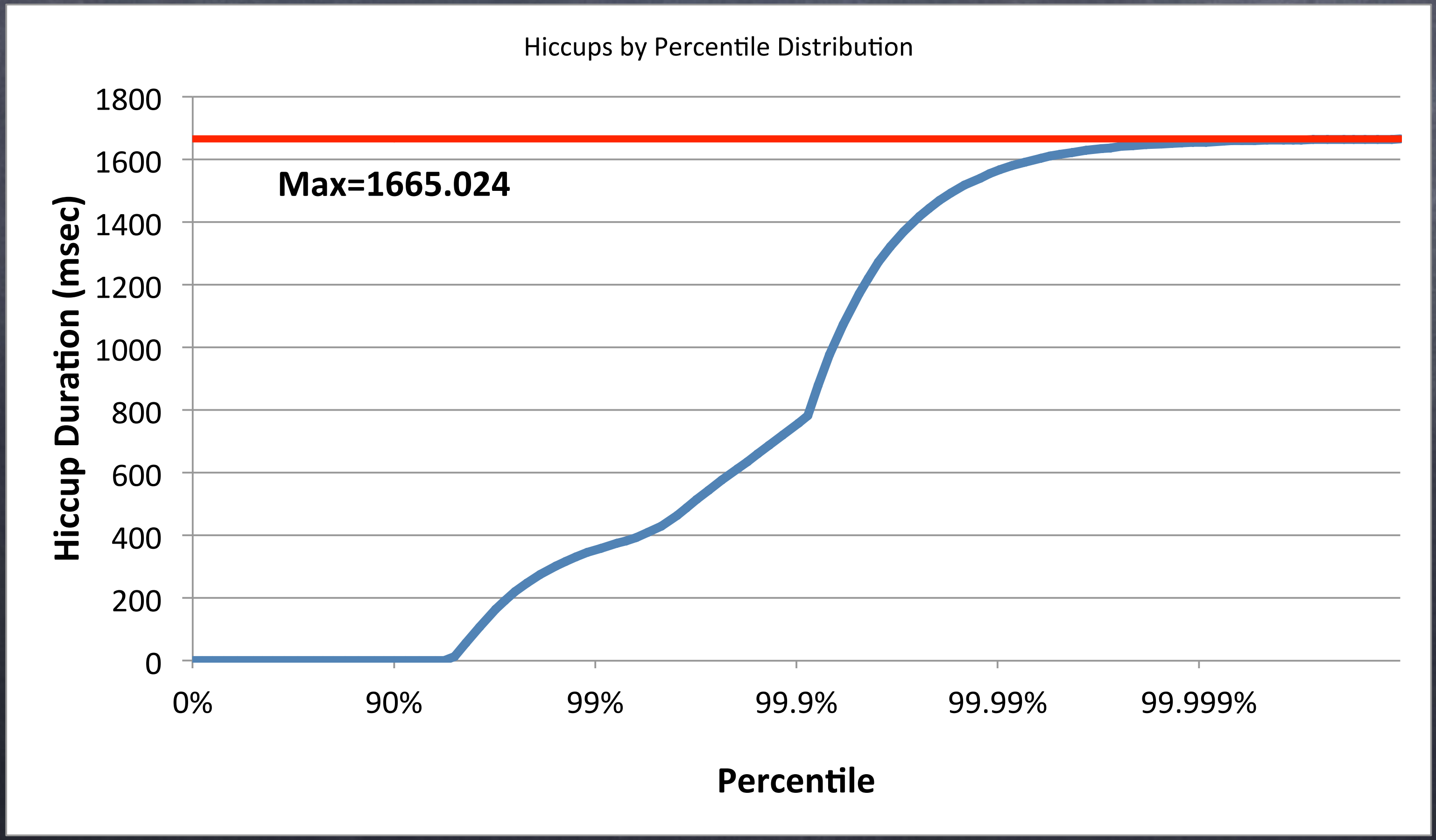
Translation: "Some pauses are much longer than tens of milliseconds"

Actually measuring things

(e.g. jHiccup)

Incontinuities in Java platform execution





C4: Solving Stop-The-World

We needed to solve the right problems

- Motivation: 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 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 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

C4's Prime Directives

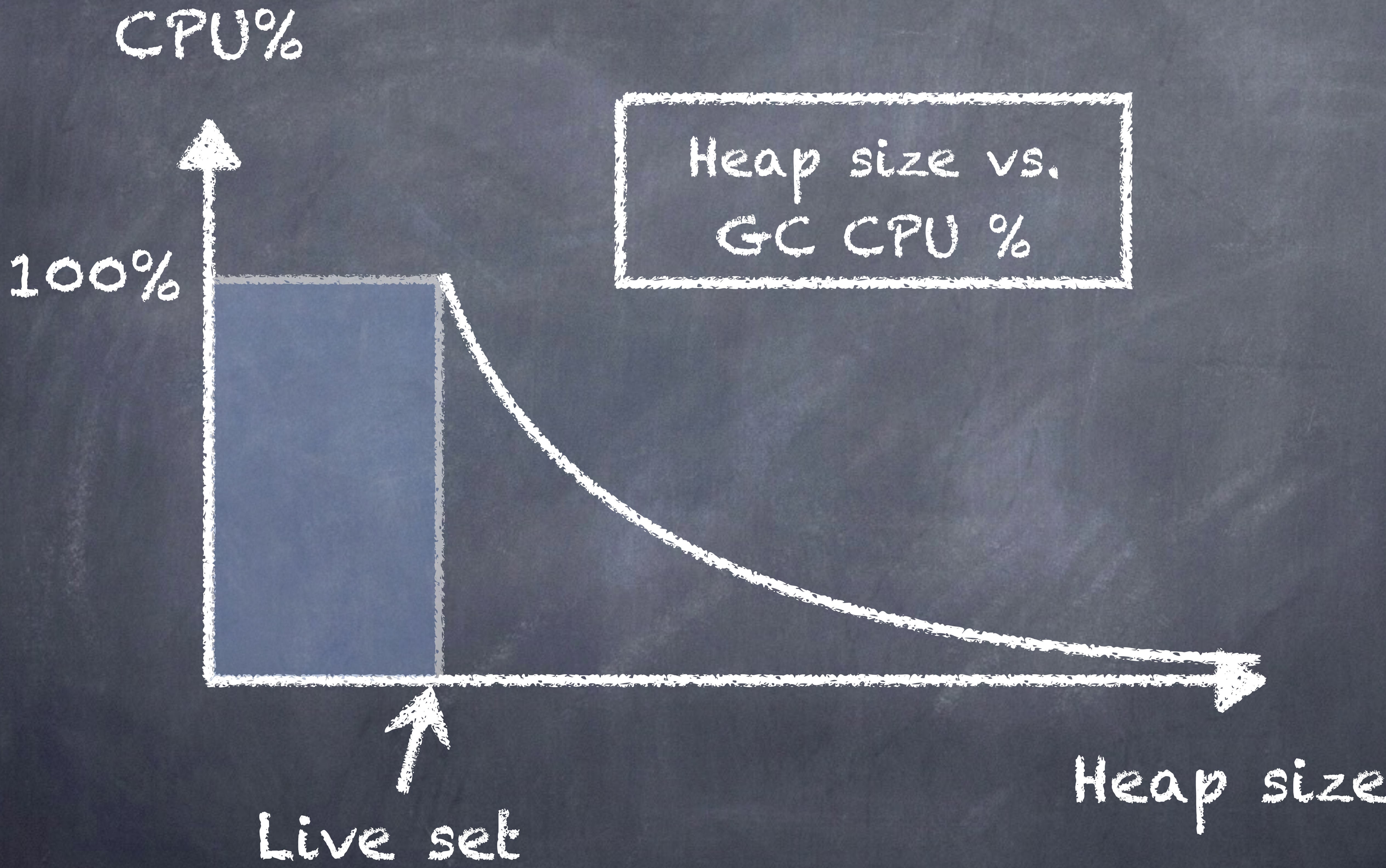
- Always do the same thing
  - Avoid the temptation to "solve" things by delaying them
  - Avoid rare code paths
  - Running under load for an hour should exercise the whole thing
- Don't be in a hurry
  - Avoid the "if we don't do this quickly it will get worse" trap
    - e.g. multi-pass marking
    - 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"
  - Keeping up with allocation rate should be the only reason for "pace"

Good Latency vs. Good Throughput

Why "vs."?

We can have both!

The secret to GC efficiency


CPU%
Heap size vs.
GC CPU %
100%
Live set
Heap size

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 reduce pause times (only their frequency)
- In fact, *IF* you do GC work in a pause, more empty memory usually means larger pauses
- With C4, we get the upside with no downside...

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

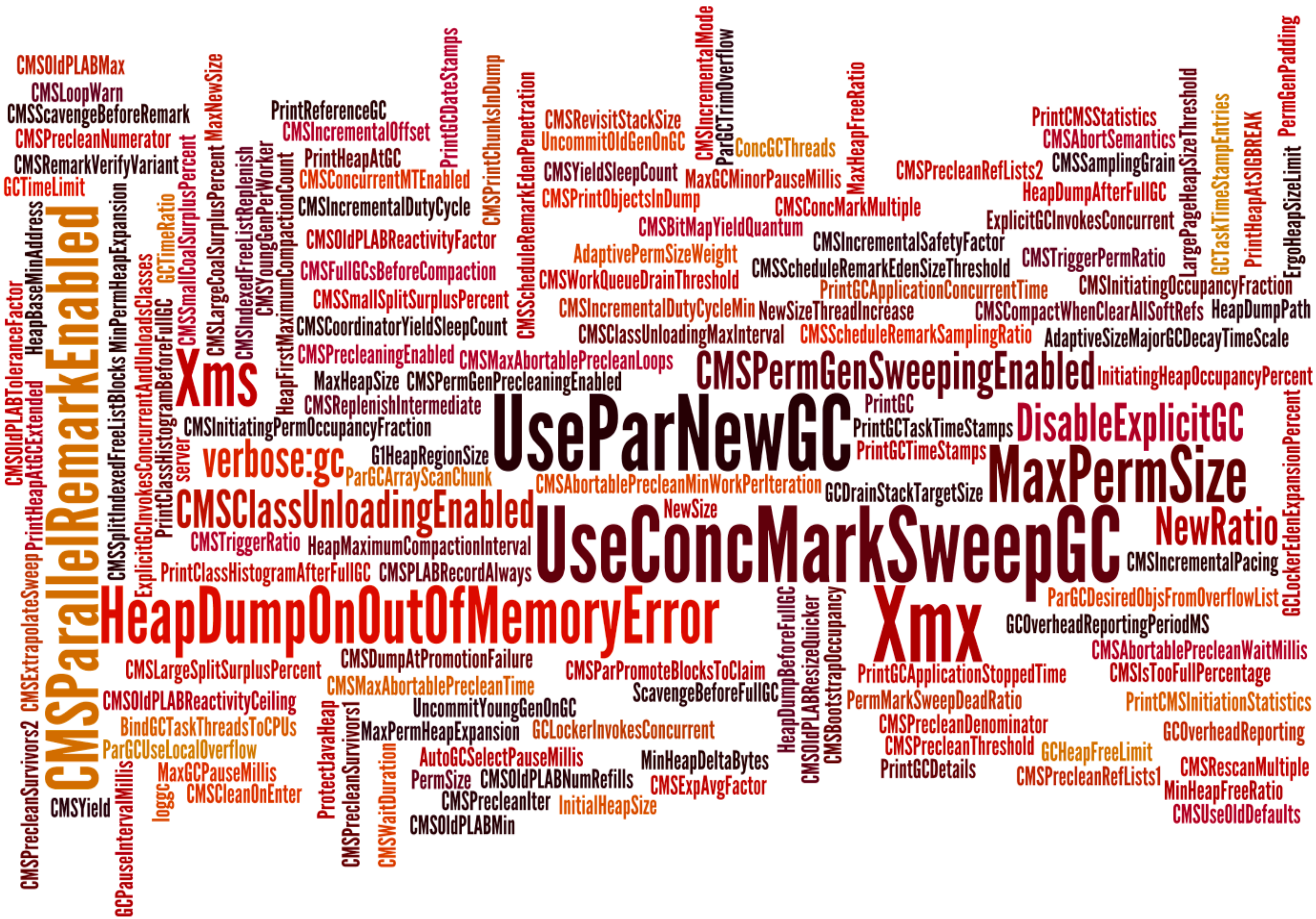
Java GC tuning is "hard"...

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:+CMSConcurrentMTEnabled
    -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**

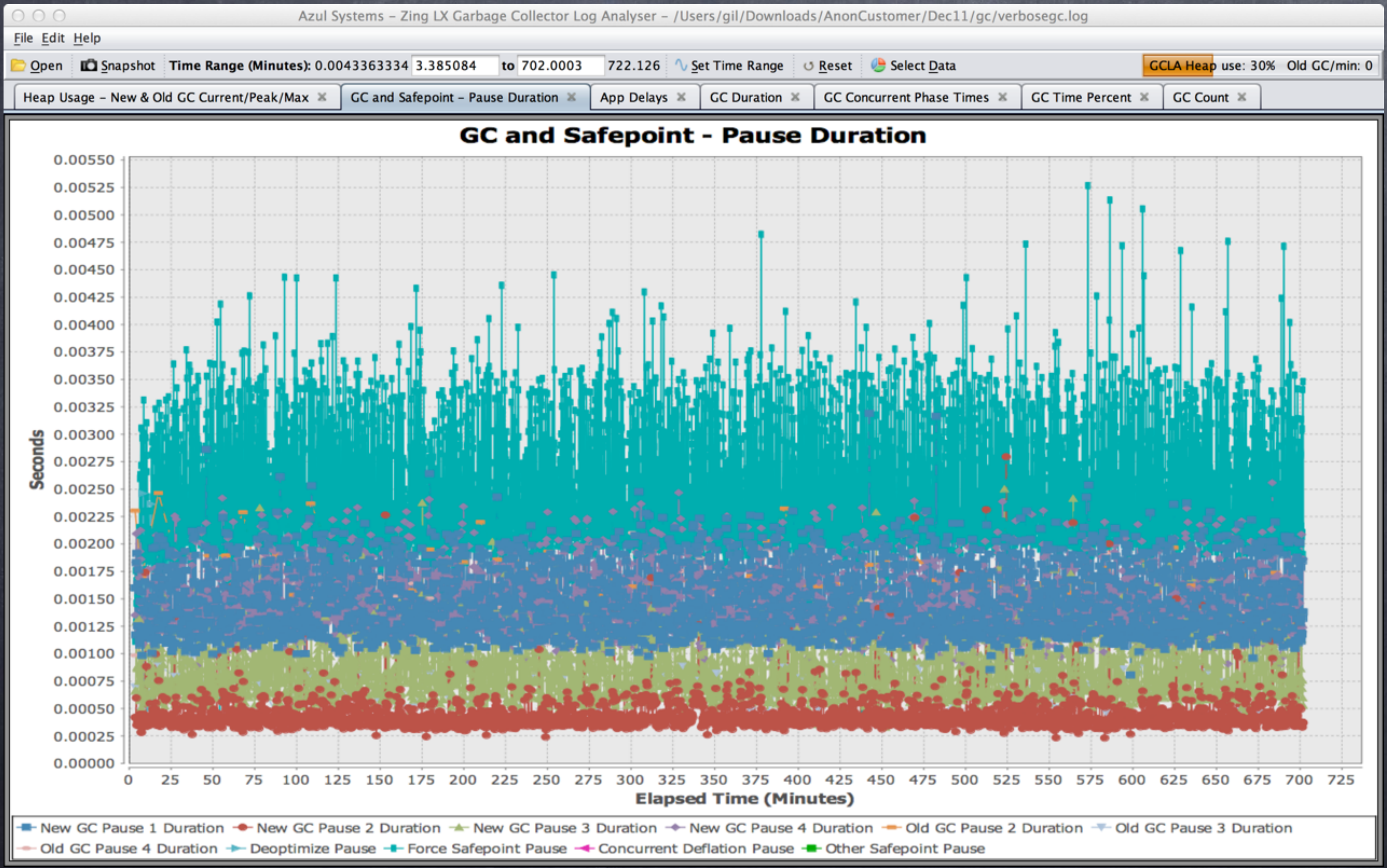
java -Xmx40g

java -Xmx20g

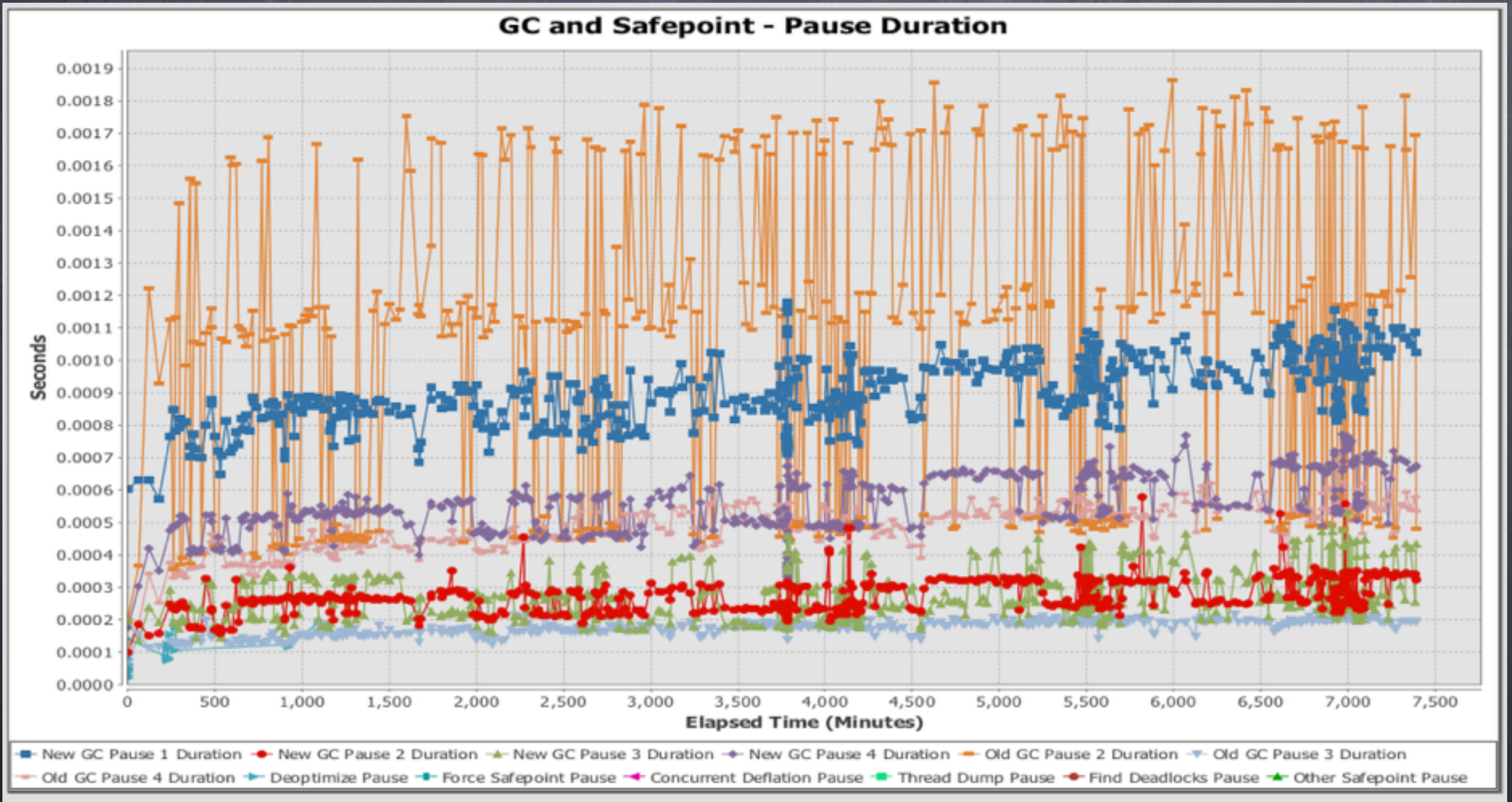
java -Xmx10g

java -Xmx5g

An example of "First day's run" behavior
E-Commerce application

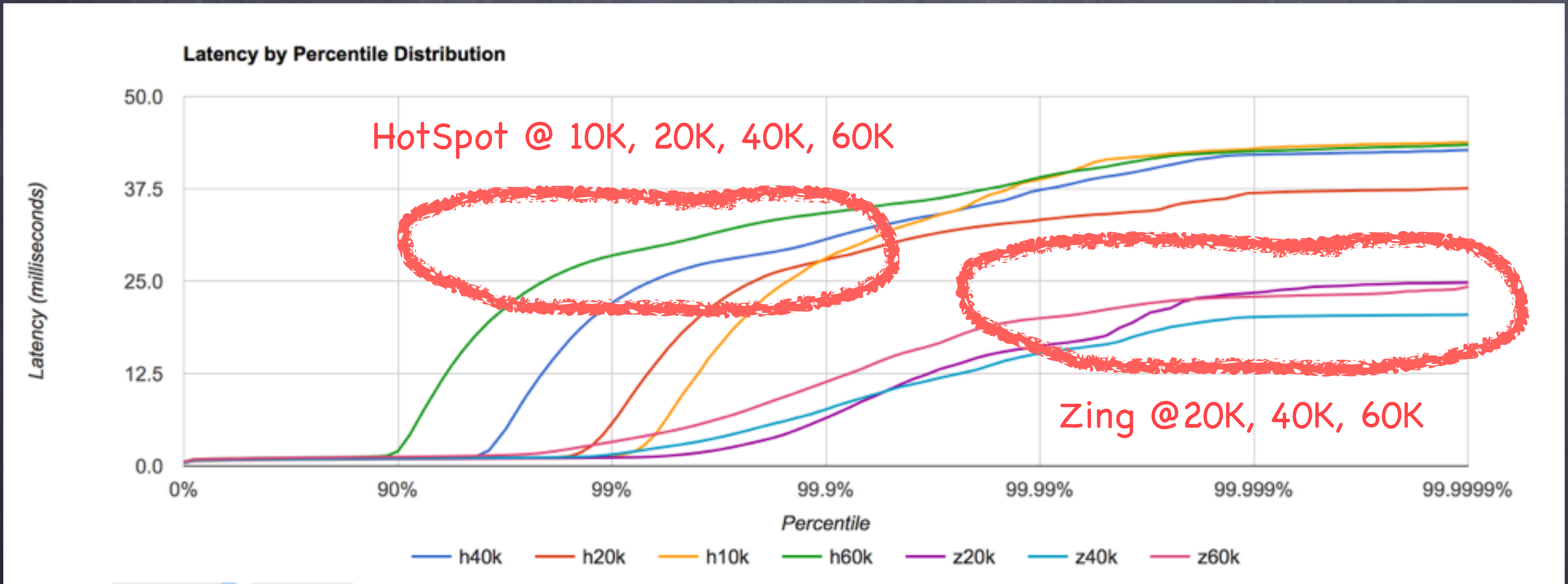


A production FX trading system over a whole week



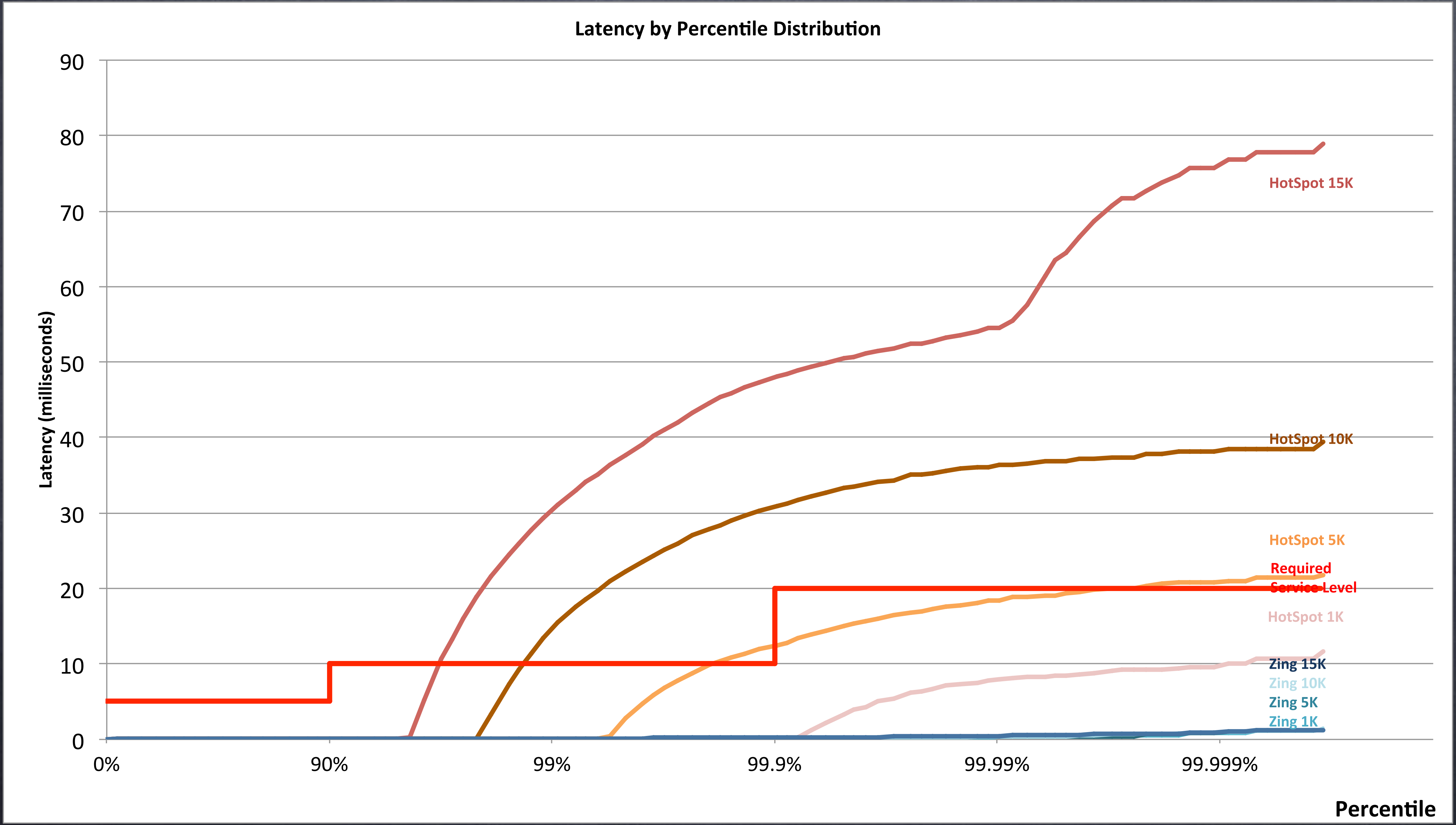
Sustainable Throughput:
The throughput achieved while safely maintaining service levels

Cassandra Query behavior (newgen-only)



Lots of conclusions can be drawn from the above…
E.g. C4 delivers a consistent 100x reduction in the rate of occurrence of >20msec query times

Comparing latency behavior under different throughputs, configurations latency sensitive messaging distribution application



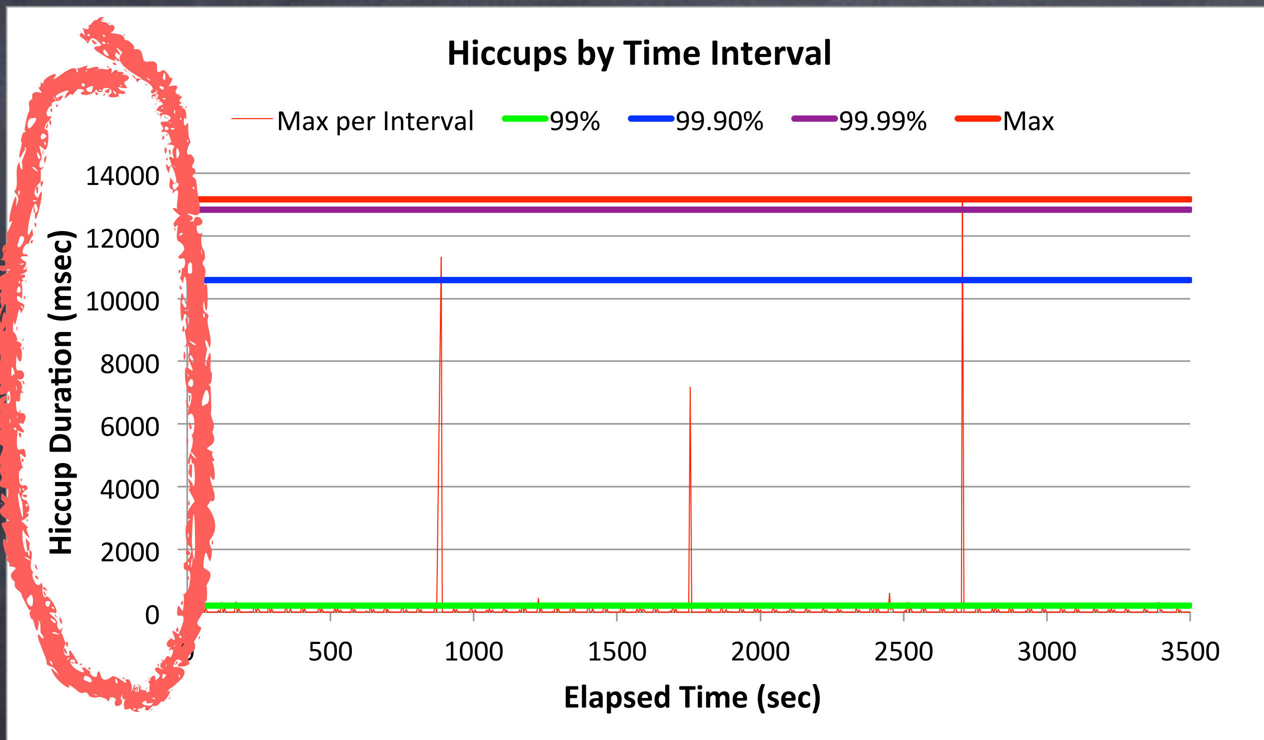
Fun with jHiccup

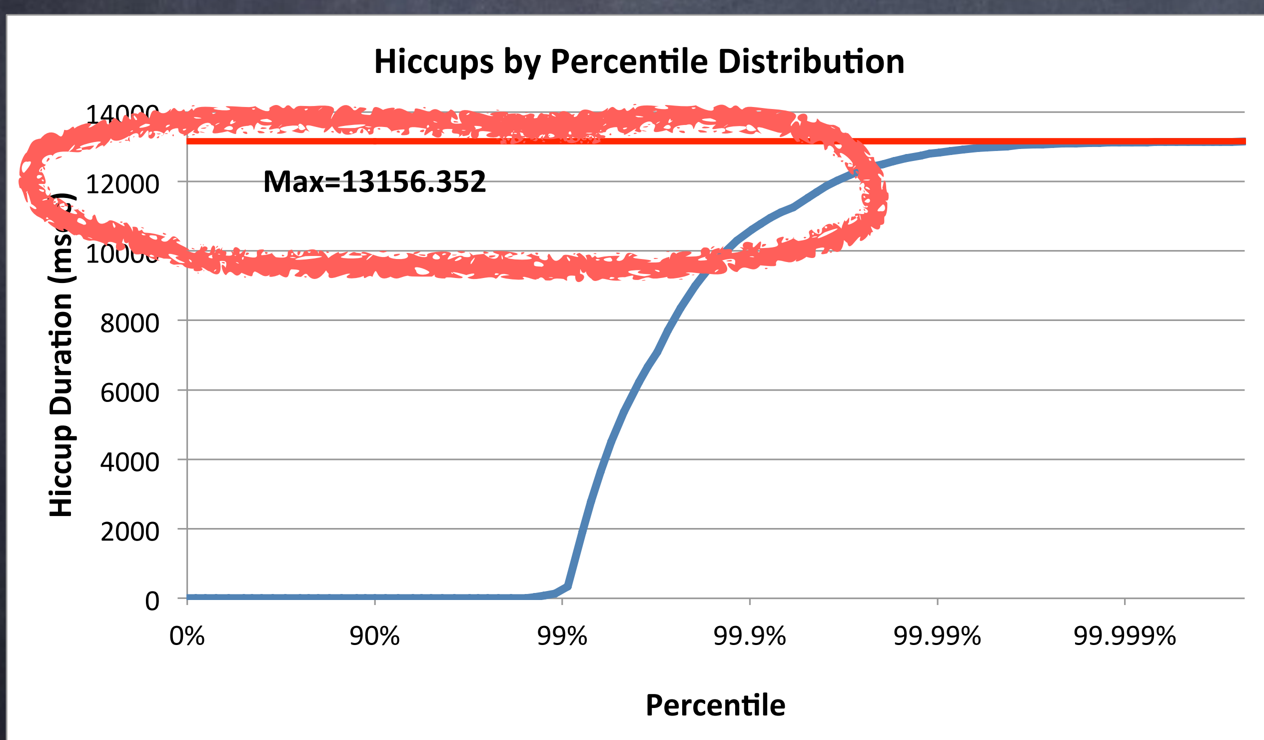
Charles Nutter @headius 20 Jan

jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)

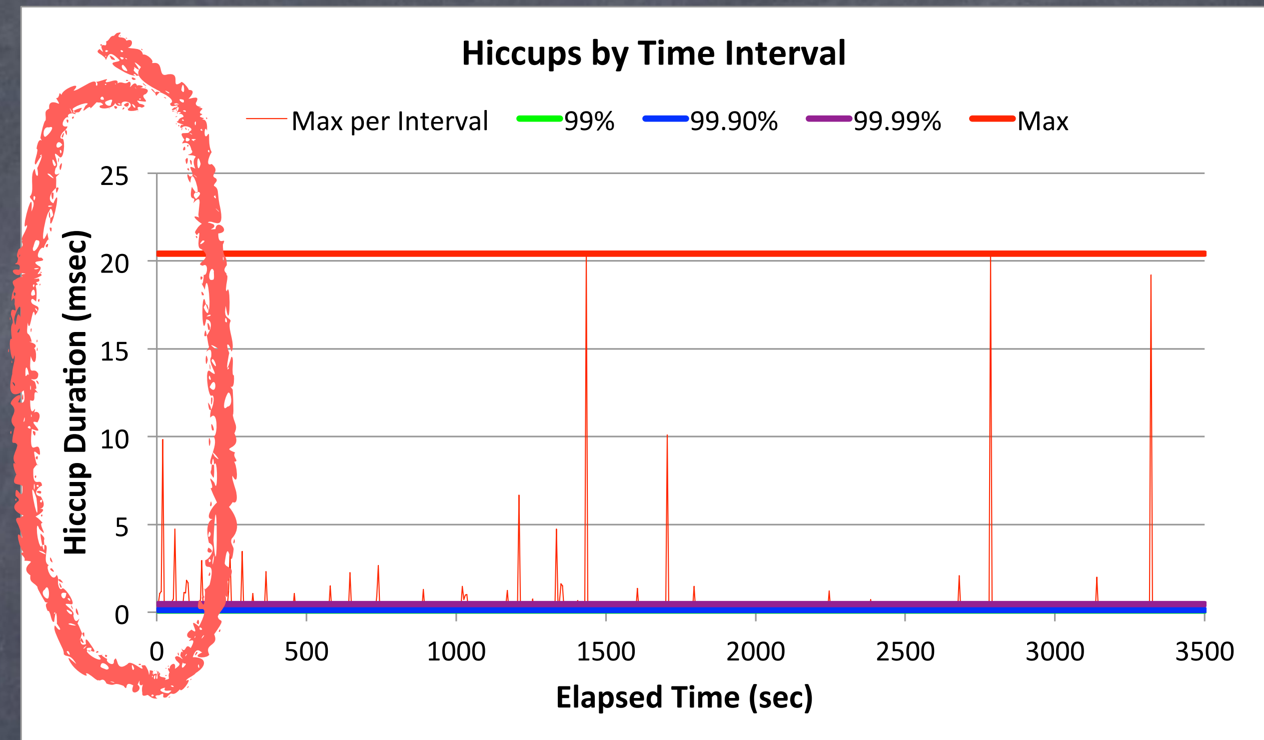
Retweeted by Gil Tene

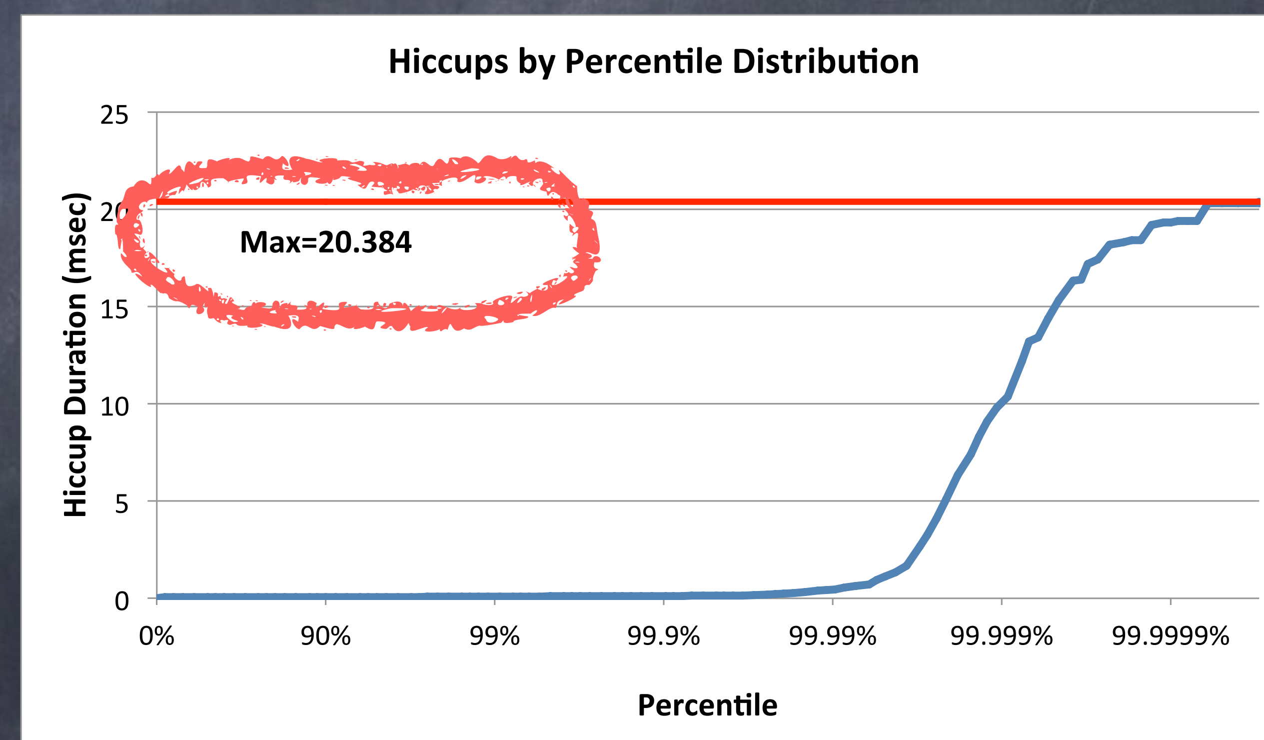
Oracle HotSpot CMS, 1GB in an 8GB heap





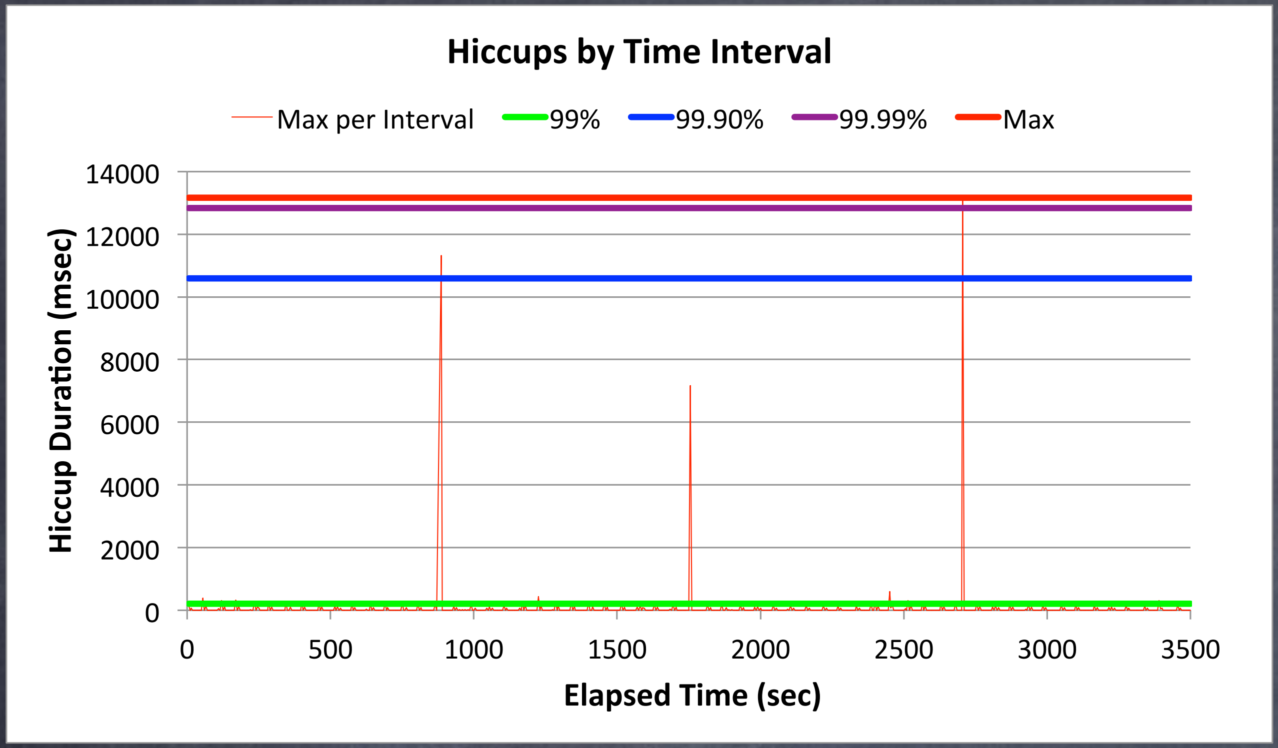
Zing 5, 1GB in an 8GB heap

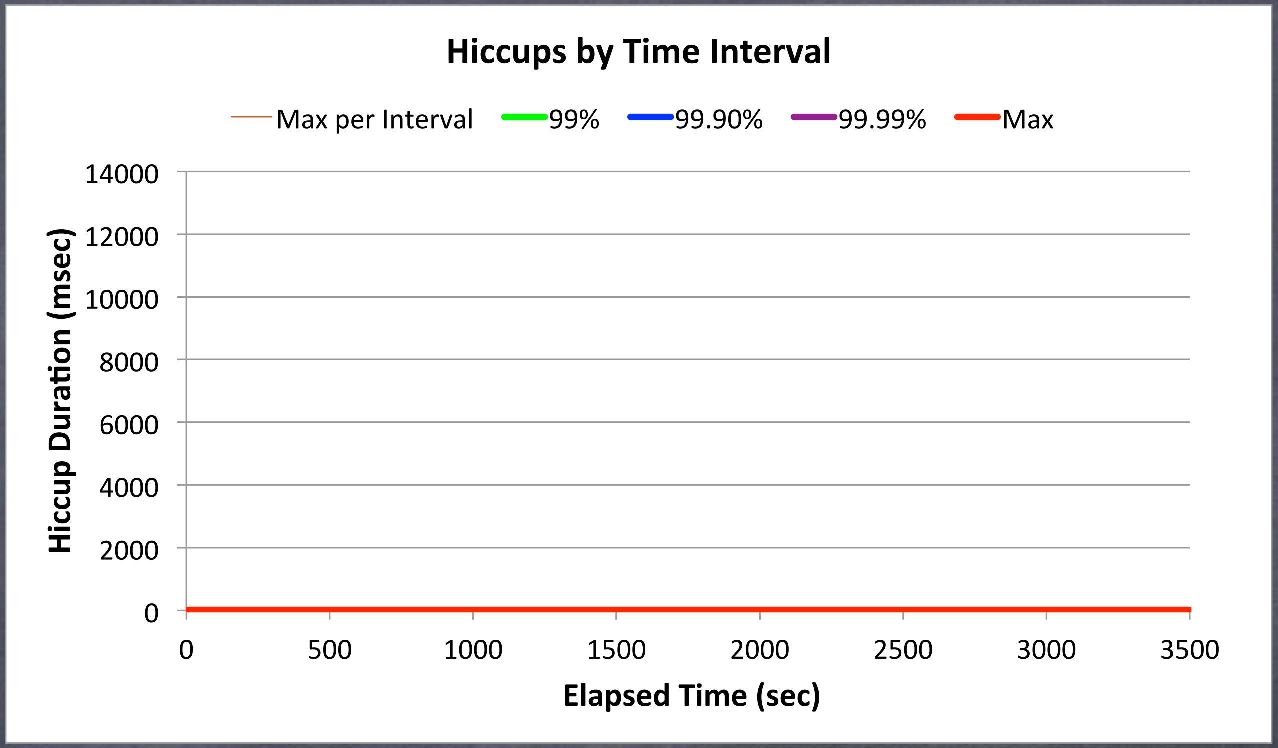


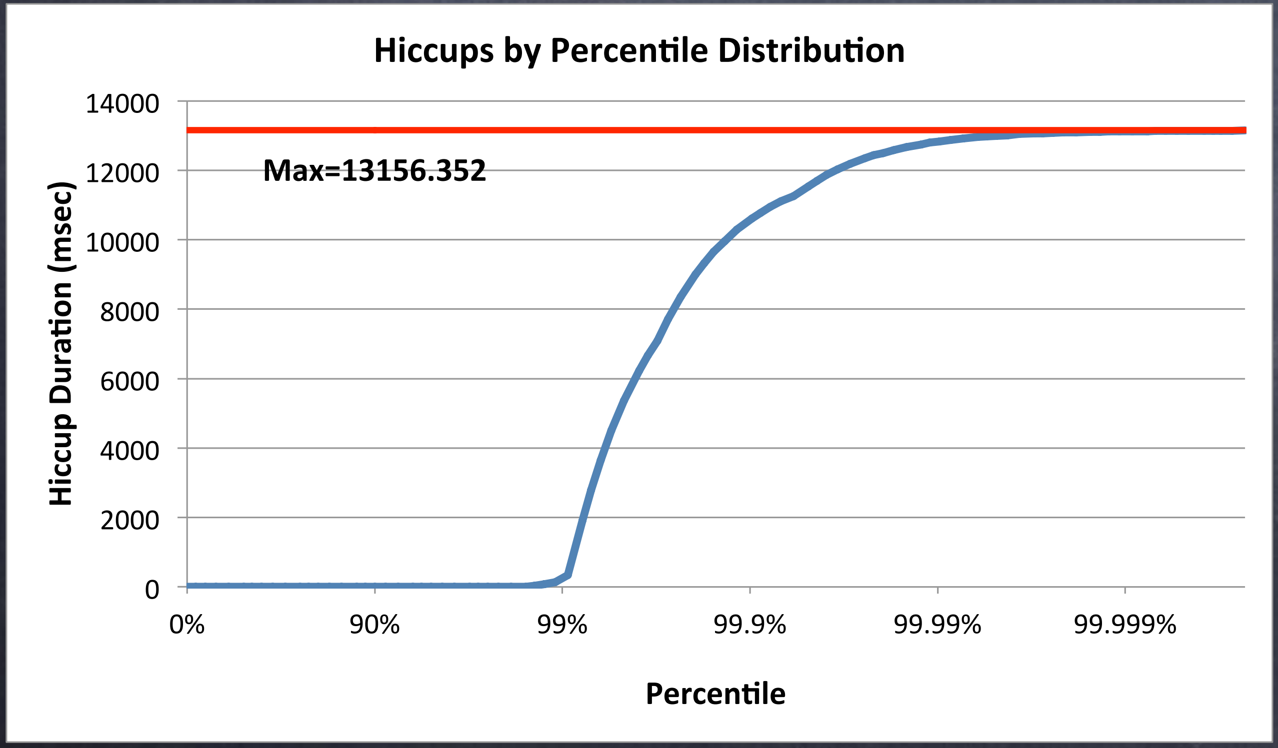


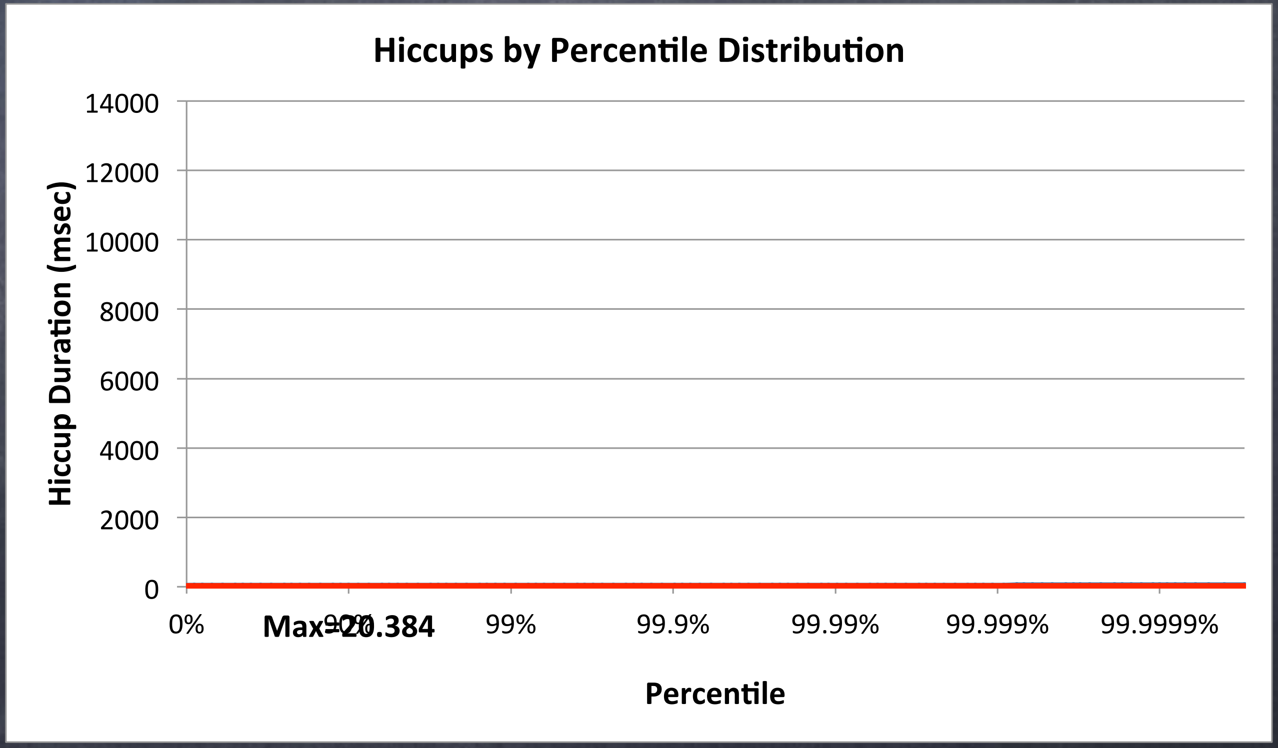
Oracle HotSpot CMS, 1GB in an 8GB heap

Zing 5, 1GB in an 8GB heap





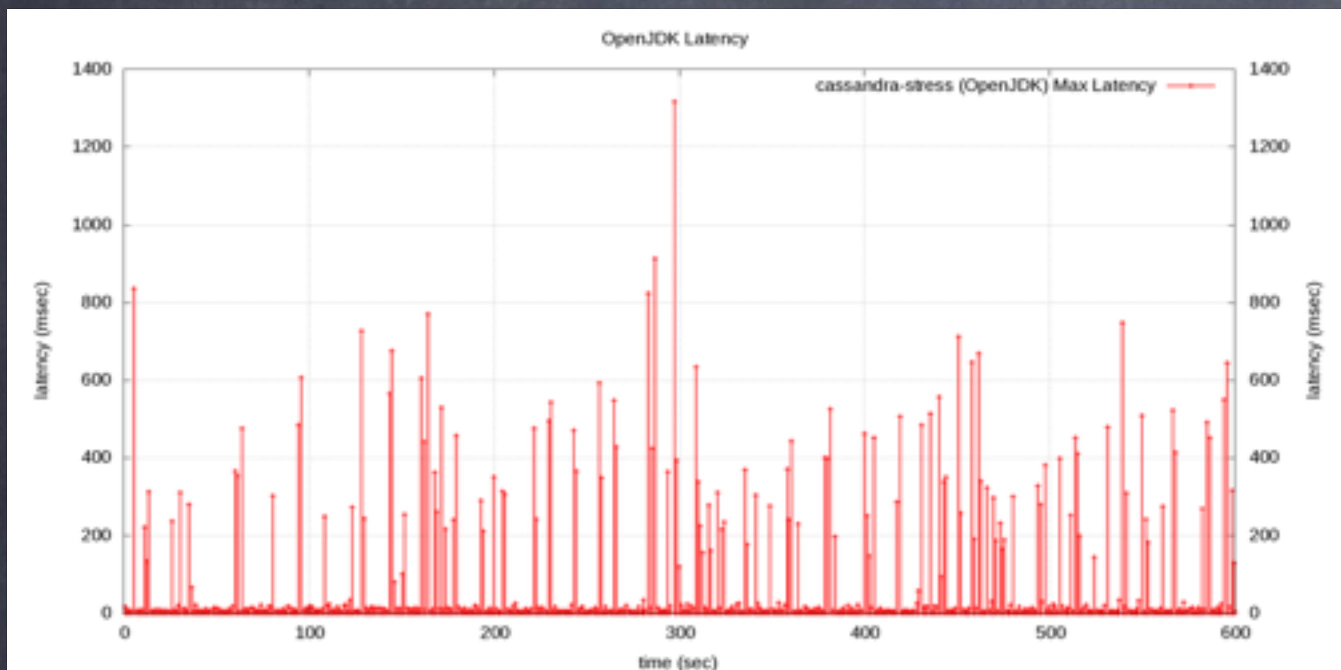




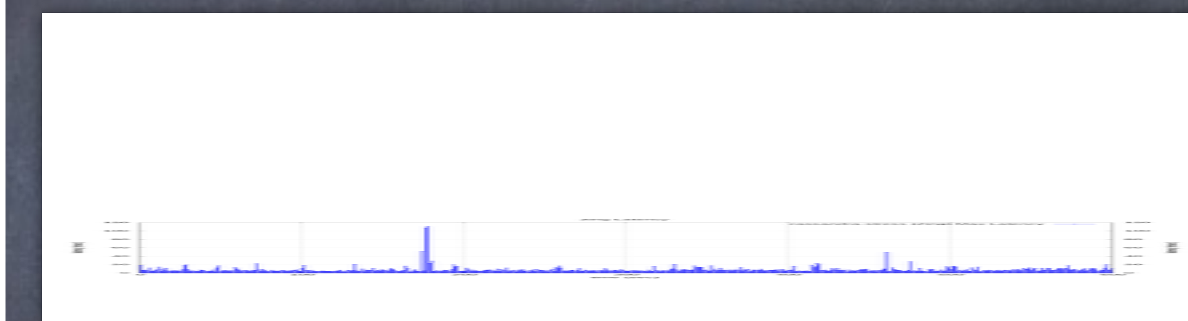
Drawn to scale

cassandra-stress

OpenJDK: 200-1400 msec stalls



Zing (drawn to scale)



```
op rate                      : 40001
partition rate               : 26996
row rate                     : 26996
latency mean                 : 30.6 (0.7)
latency median               : 0.5 (0.5)
latency 95th percentile      : 244.4 (1.1)
latency 99th percentile      : 537.4 (2.0)
latency 99.9th percentile    : 1052.2 (8.4)
latency max                  : 1314.9 (1312.8)
```

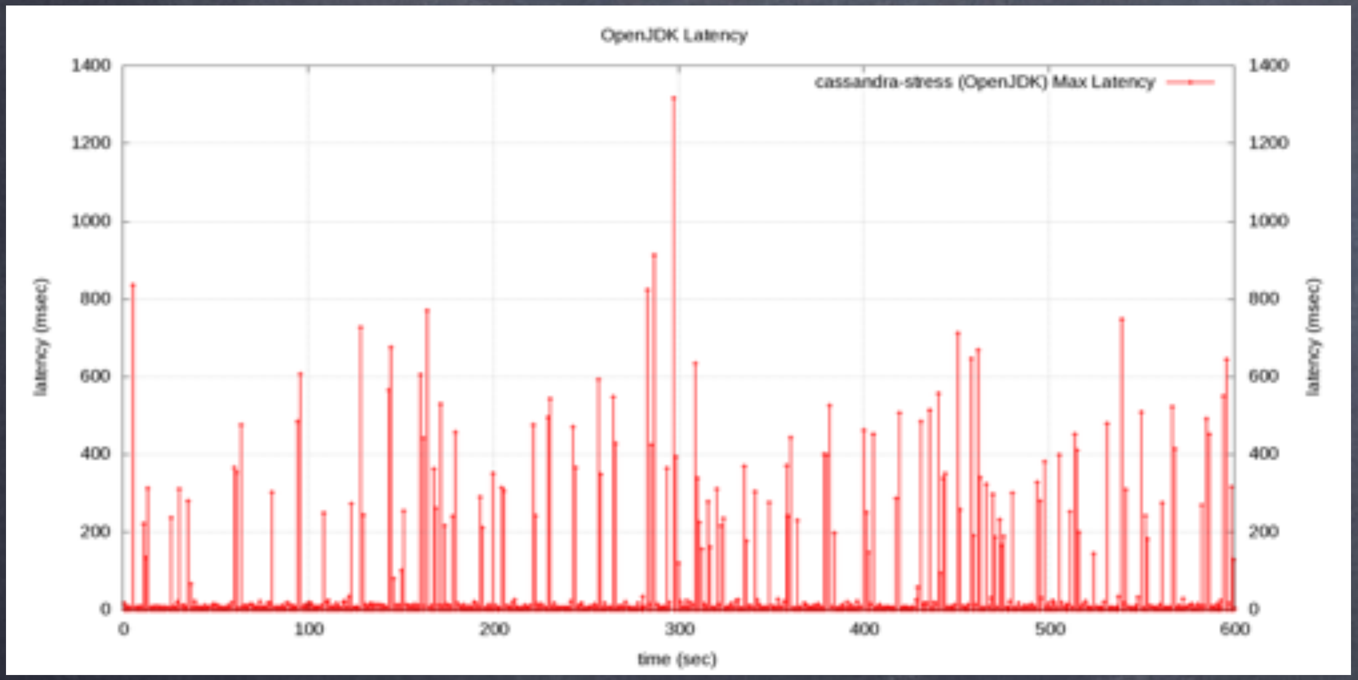
Response Time   Service time

```
op rate                      : 40001
partition rate               : 26961
row rate                     : 26961
latency mean                 : 0.6 (0.5)
latency median               : 0.5 (0.5)
latency 95th percentile      : 1.0 (0.9)
latency 99th percentile      : 2.7 (1.9)
latency 99.9th percentile    : 13.3 (3.8)
latency max                  : 110.6 (28.2)
```

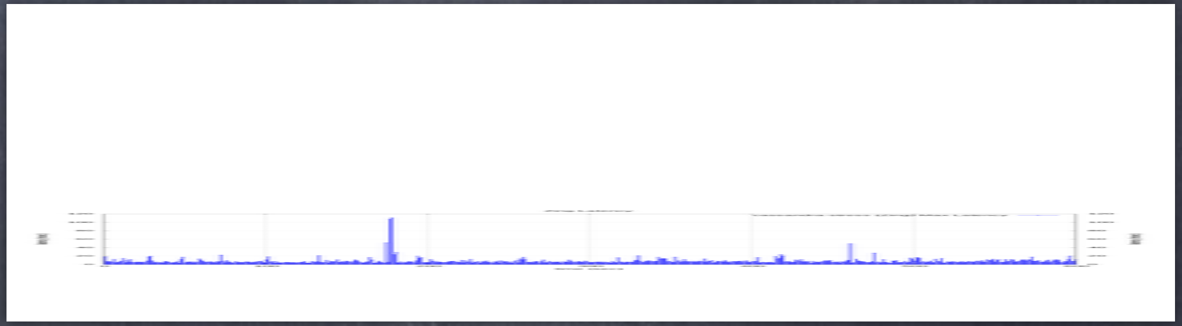
Response Time   Service time

A simple visual summary

This is Cassandra on HotSpot



This is Cassandra on Zing



Any Questions?

Any Questions?

http://www.azulsystems.com

http://www.jhiccup.com

http://giltene.github.com/HdrHistogram