Enabling Java in Latency Sensitive Applications

Gil Tene, CTO & co-Founder, Azul Systems

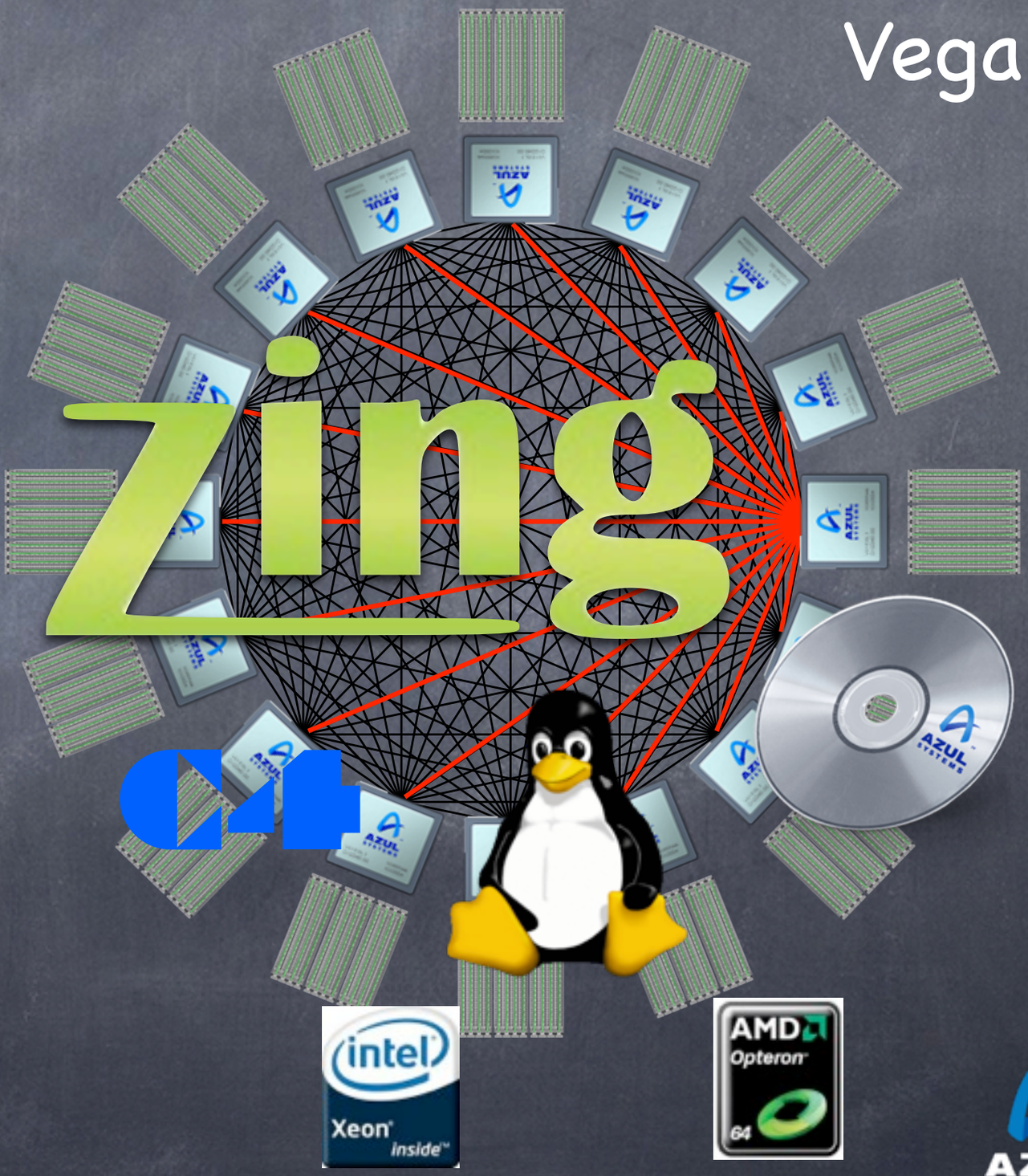
About me: Gil Tene

- 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

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)
- **Zing: Pure software for commodity x86**
- Known for Low Latency, Consistent execution, and Large data set excellence



High level agenda

- Java in a low latency application world
- The (historical) fundamental problems
- What people have done to try to get around them
- What if the fundamental problems were eliminated?
- What 2013 looks like for Low latency Java developers
- What's next?

Java in the low latency world?

Java in the low latency world?

- Why do people use Java for low latency apps?
- Are they crazy?
- No. There are good, easy to articulate reasons
  - Projected lifetime cost
  - Developer productivity
  - Time-to-product, Time-to-market, …
  - Leverage, ecosystem, ability to hire

AZUL SYSTEMS

E.g. Customer answer to: "Why do you use Java in Algo Trading?"

- Strategies have a shelf life
- We have to keep developing and deploying new ones
- Only one out of N is actually productive
- Profitability therefore depends on ability to successfully deploy new strategies, and on the cost of doing so
- Our developers seem to be able to produce 2x-3x as much when using a Java environment as they would with C/C++ ...

AZUL SYSTEMS

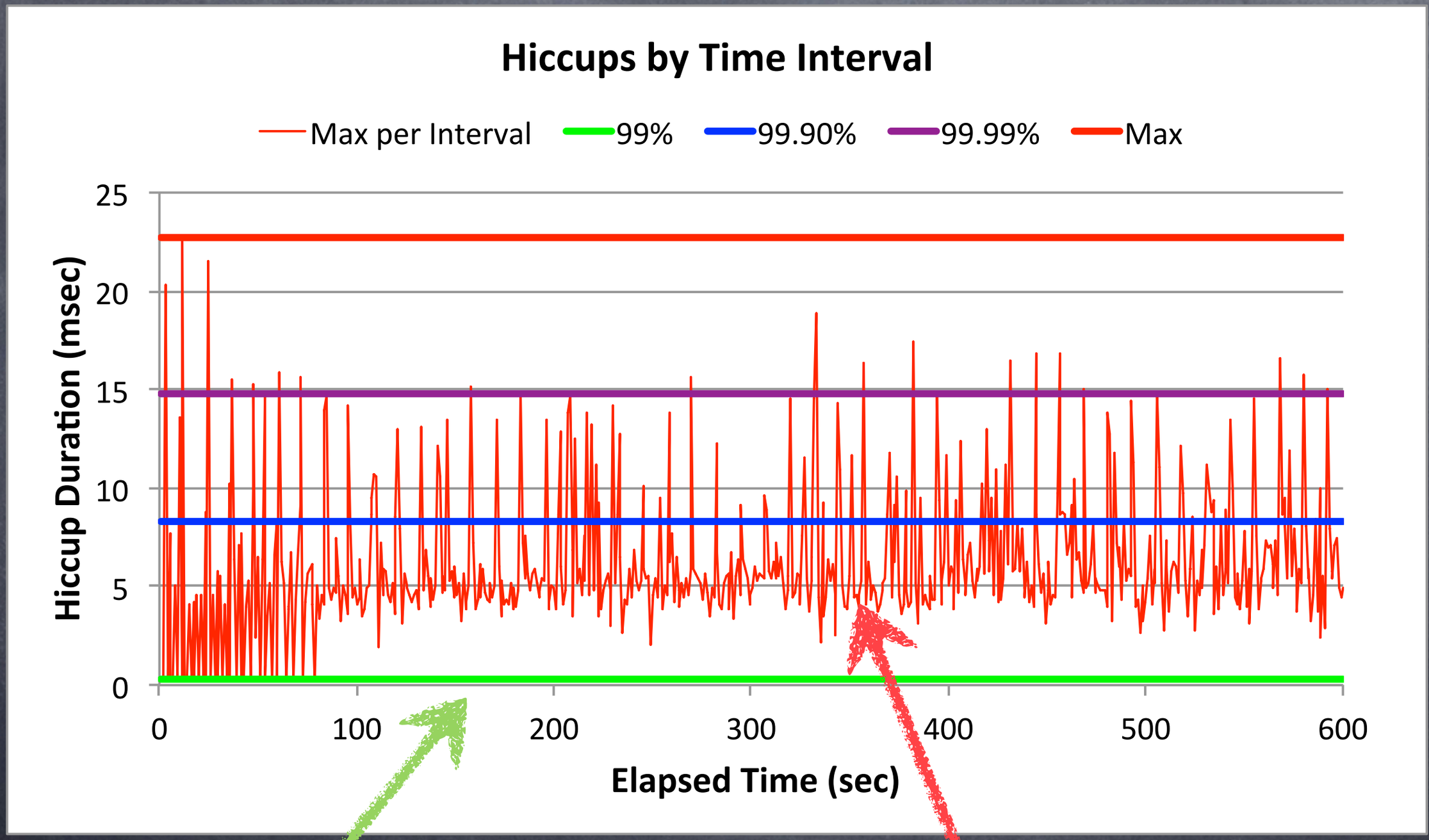
So what is the problem?

Is Java Slow?

- No...
- A good programmer will get roughly the same speed from both Java and C++
- A bad programmer won't get you fast code on either
- The 50%'ile and 90%'ile are typically excellent...
- It's those pesky occasional stutters and stammers and stalls that are the problem...
- Ever hear of Garbage Collection?

AZUL SYSTEMS®

Is "jitter" even the right word for this?



99%'ile is ~60 usec

Max is ~30,000% higher than "typical"

AZUL SYSTEMS

Stop-The-World Garbage Collection: Java's Achilles heel

- Let's ignore the bad multi-second pauses for now...
- Low latency applications regularly experience "small", "minor" GC events that range in the 10s of msec
- Frequency directly related to allocation rate
- So we have great 50%, 90%. Maybe even 99%
- But 99.9%, 99.99%, Max, all "suck"
- So bad that it affects risk, profitability, service expectations, etc.

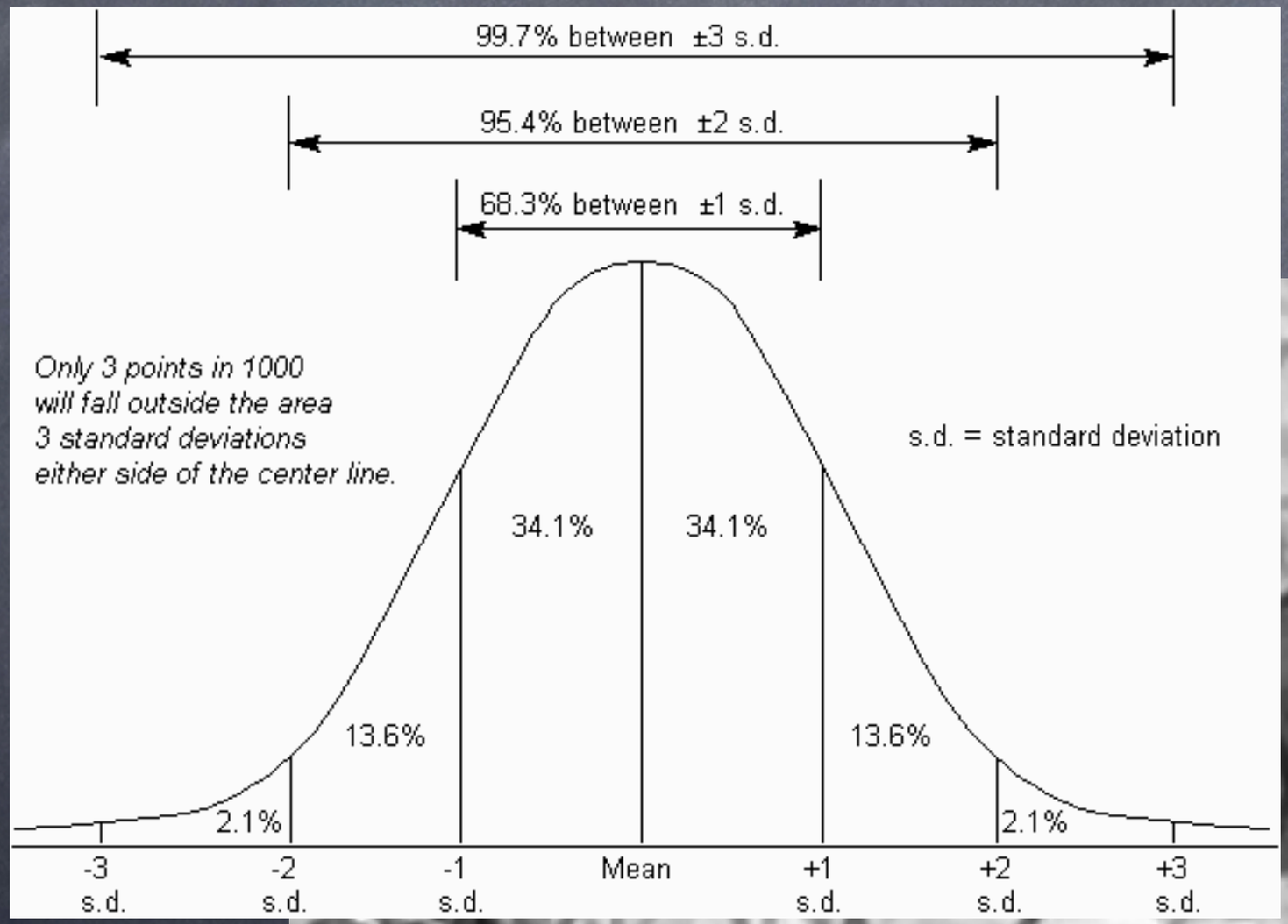
AZUL SYSTEMS

One way to deal with Stop-The-World GC

AZUL SYSTEMS®

A common way to "deal" with STW-GC

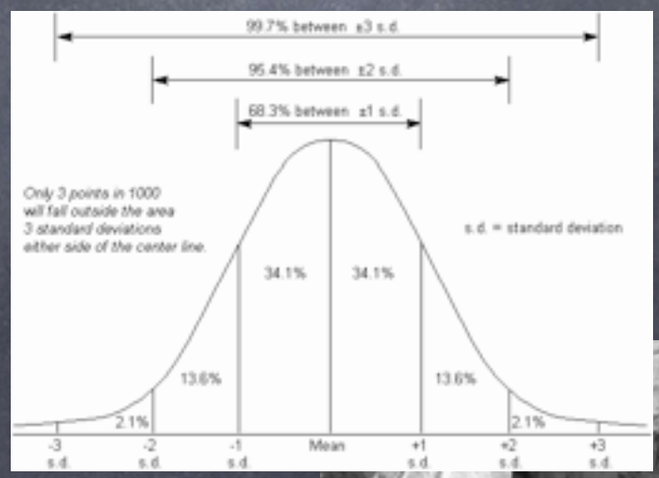
Averages and Standard Deviation

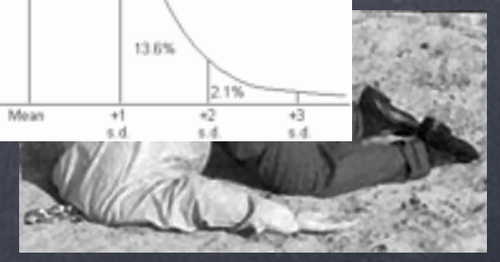


AZUL SYSTEMS

Reality: Latency is usually strongly "multi-modal"

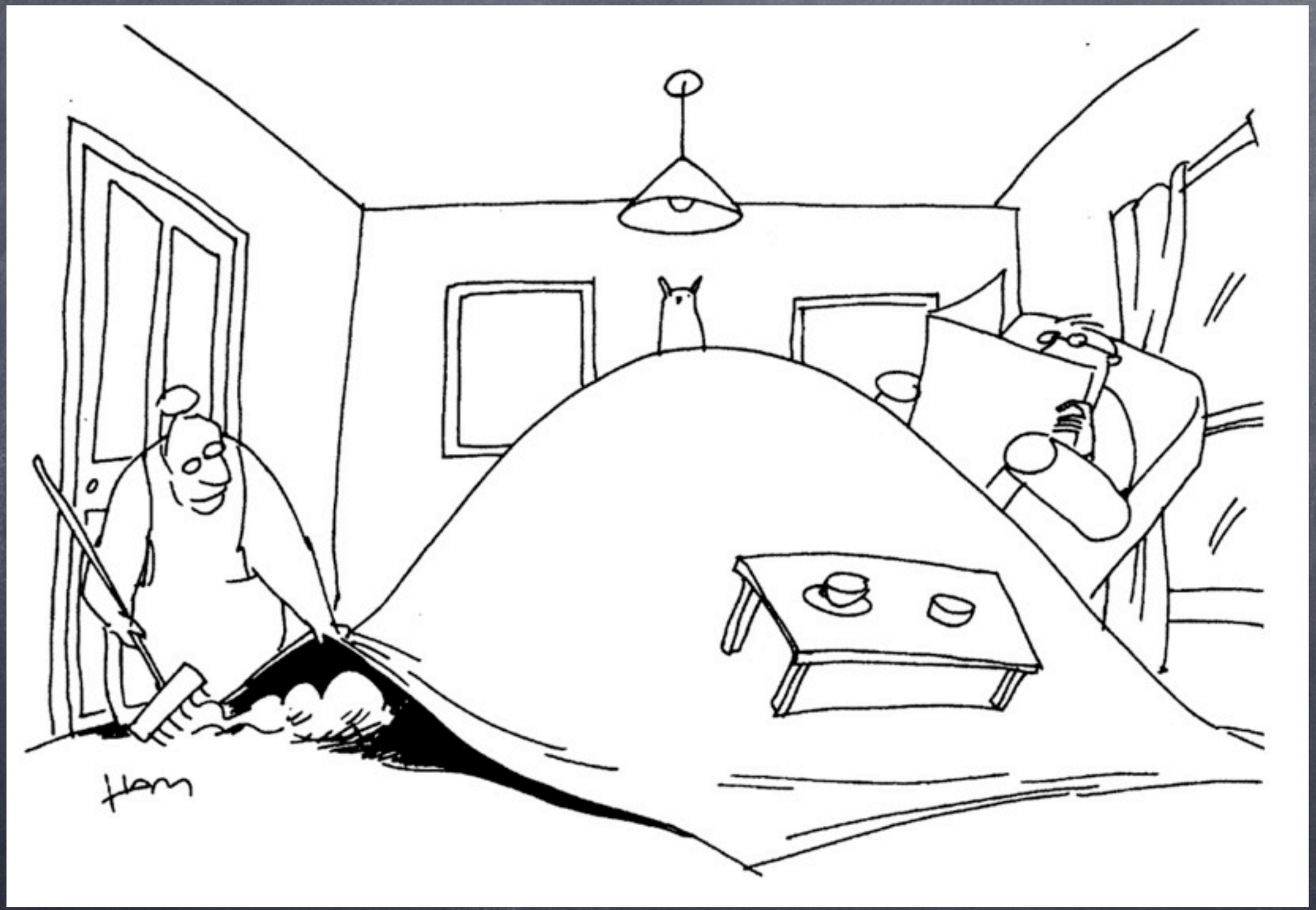
- Usually does't look anything like a normal distribution
- In software systems, usually sees periodic freezes
- Complete shifts from one mode/behavior to another
- Mode A: "good".
- Mode B: "Somewhat bad"
- Mode C: "terrible", ...
- ....





AZUL SYSTEMS

Another way to deal with STW-GC



Another way to cope: "Creative Language"

- "Guarantee a worst case of 5 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"

What do actual low latency developers do about it?

- They use "Java" instead of Java
- They write "in the Java syntax"
- They avoid allocation as much as possible
- E.g. They build their own object pools for everything
- They write all the code they use (no 3rd party libs)
- They train developers for their local discipline
- In short: They revert to many of the practices that hurt productivity. They loose out on much of Java.

AZUL SYSTEMS

What do low latency (Java) developers with all this effort?

- They still see pauses (usually ranging to tens of msec)
- They do get fewer (as in less frequent) pauses
- And they see fewer people able to do the job
- And they have to write EVERYTHING themselves
- And they get to debug malloc/free patterns again
- And they can only use memory in certain ways
- ...
- Some call it "fun"... Others "duct tape engineering"...

was
It ~~is~~ an industry-wide problem

Stop-The-World GC mechanisms contradict the fundamental requirements of low latency & low jitter apps

It's 2013... We now have Zing.

AZUL SYSTEMS®

The common GC behavior across ALL currently shipping (non-Zing) JVMs

- ALL use a Monolithic Stop-the-world NewGen
  - "small" periodic pauses (small as in 10s of msec)
  - pauses more frequent with higher throughput or allocation rates
- Development focus for ALL is on Oldgen collectors
  - Focus is on trying to address the many-second pause problem
  - Usually by sweeping it farther and farther the rug
  - "Mostly X" (e.g. "mostly concurrent") hides the fact that they refer only to the OldGen part of the collector
  - E.g. CMS, G1, Balanced.... all are OldGen-only efforts
- ALL use a Fallback to Full Stop-the-world Collection
  - Used to recover when other mechanisms (inevitably) fail
  - Also hidden under the term "Mostly"...

AZUL SYSTEMS

A Recipe: address STW-GC head-on

- At Azul, we decided to focus on the core problems
- Scale & productivity limited by responsiveness/latency
- And it's not the "typical" latency, it's the outliers...
- Even "short" GC pauses must be considered a problem

- Responsiveness must be unlinked from key metrics:
  - Transaction Rate, Concurrent users, Data set size, etc.
  - Heap size, Live Set size, Allocation rate, Mutation rate
  - Responsiveness must be continually sustainable
  - Can't ignore "rare but periodic" events

- Eliminate ALL Stop-The-World Fallbacks

AZUL SYSTEMS

The Zing "C4" Collector
Continuously Concurrent Compacting Collector

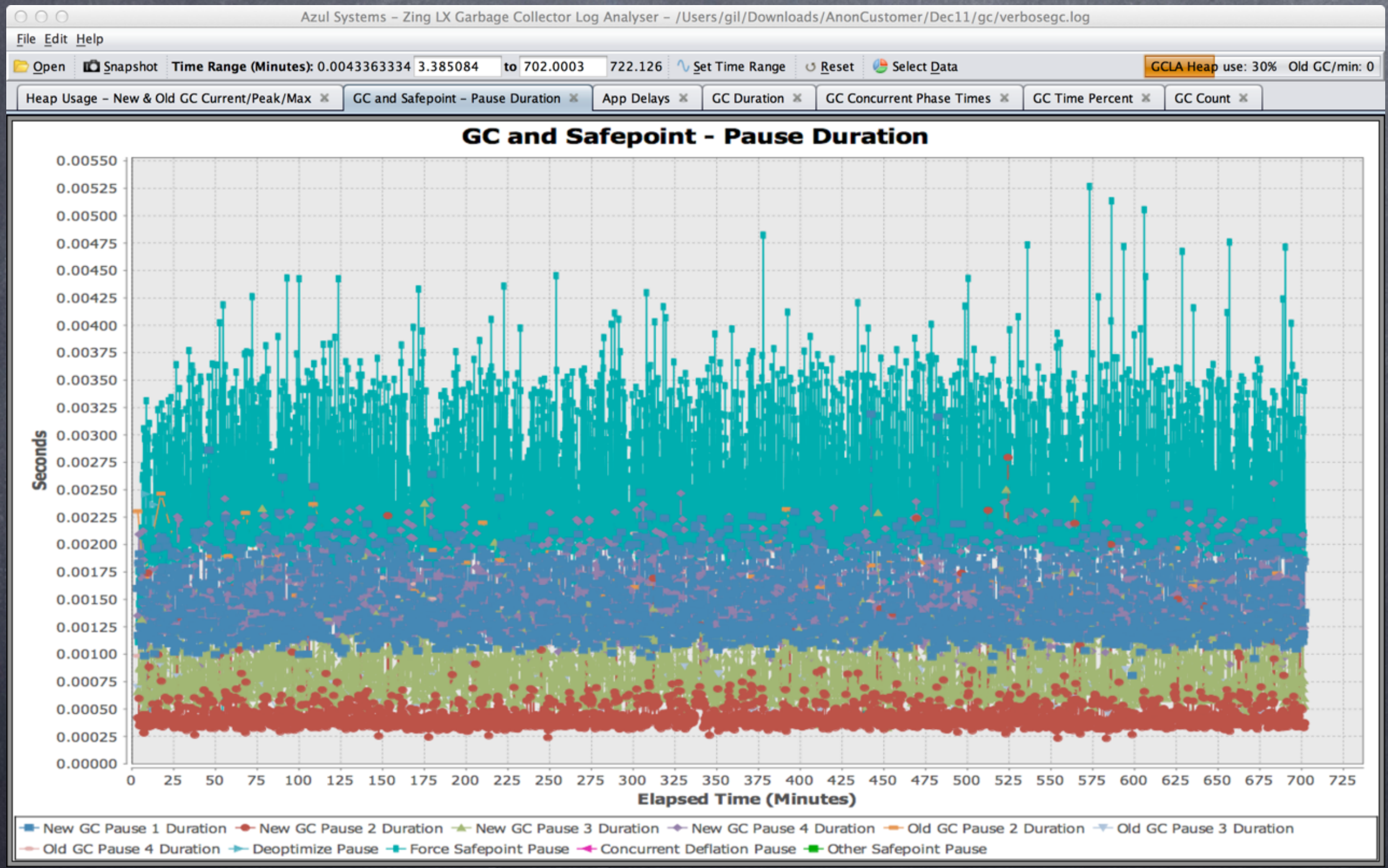
- Concurrent, compacting old generation
- Concurrent, compacting new generation
- No stop-the-world fallback
  - Always compacts, and always does so concurrently

AZUL SYSTEMS®

Benefits

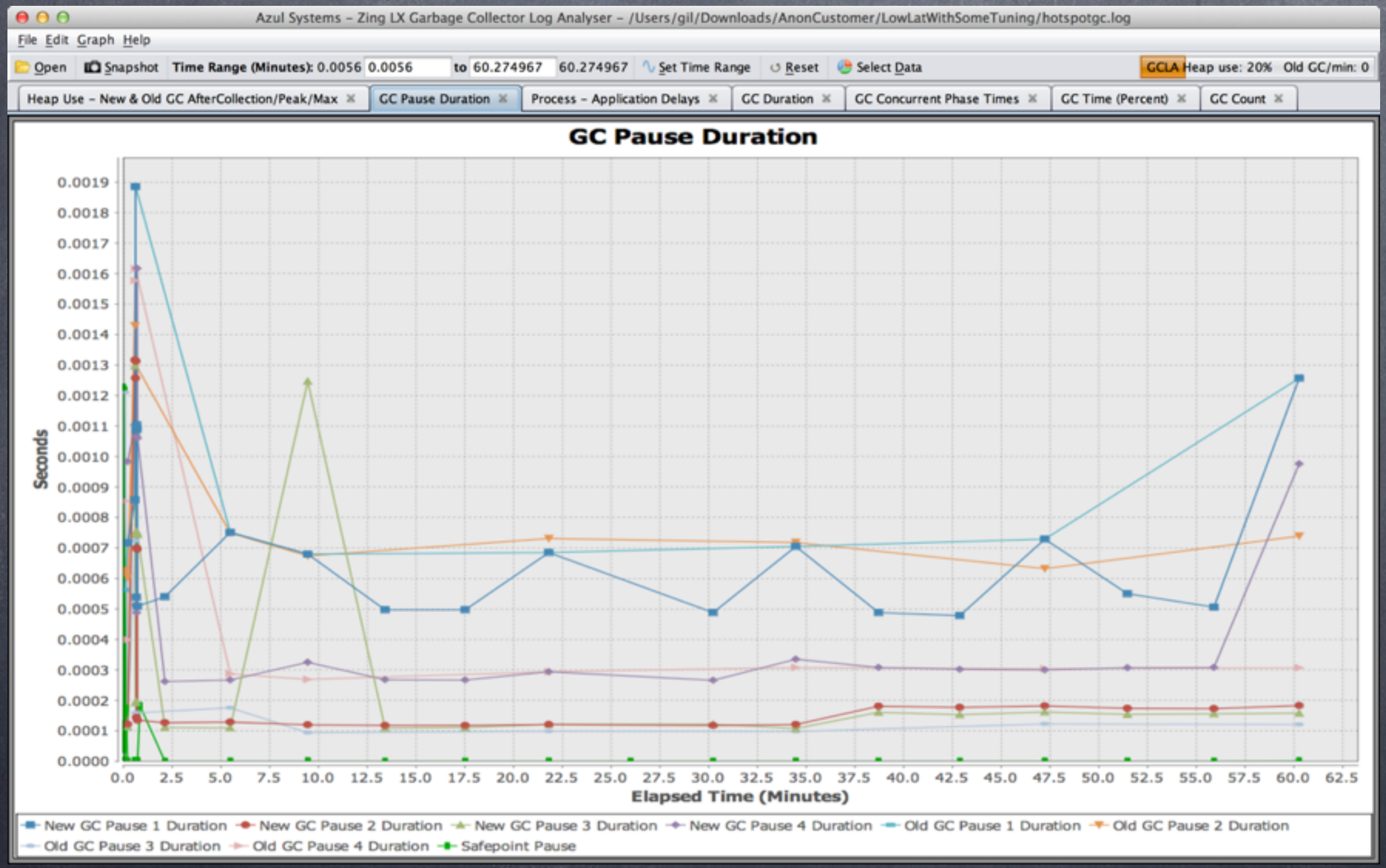
AZUL SYSTEMS®

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



AZUL SYSTEMS

An example of behavior after 4 days of system tuning
Low latency application



AZUL SYSTEMS

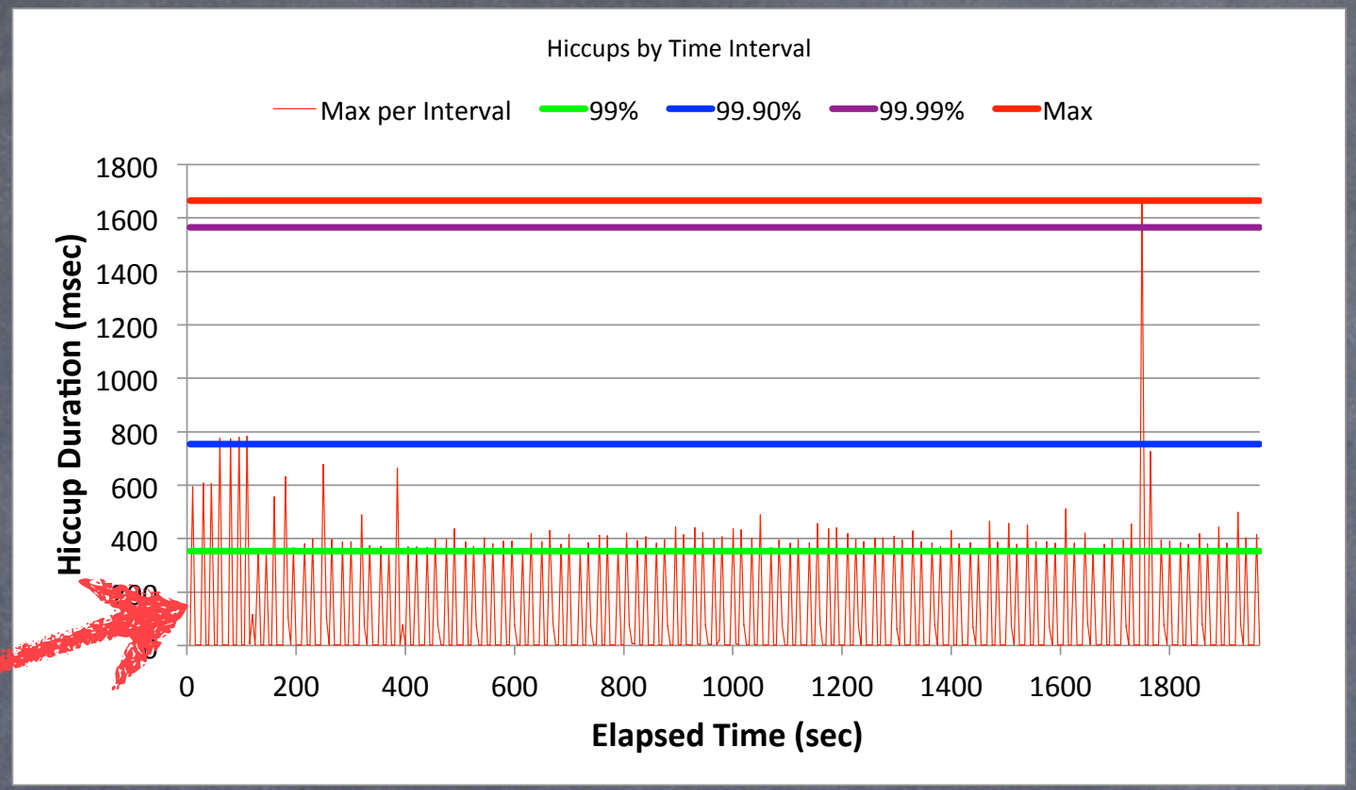
Measuring Theory in Practice

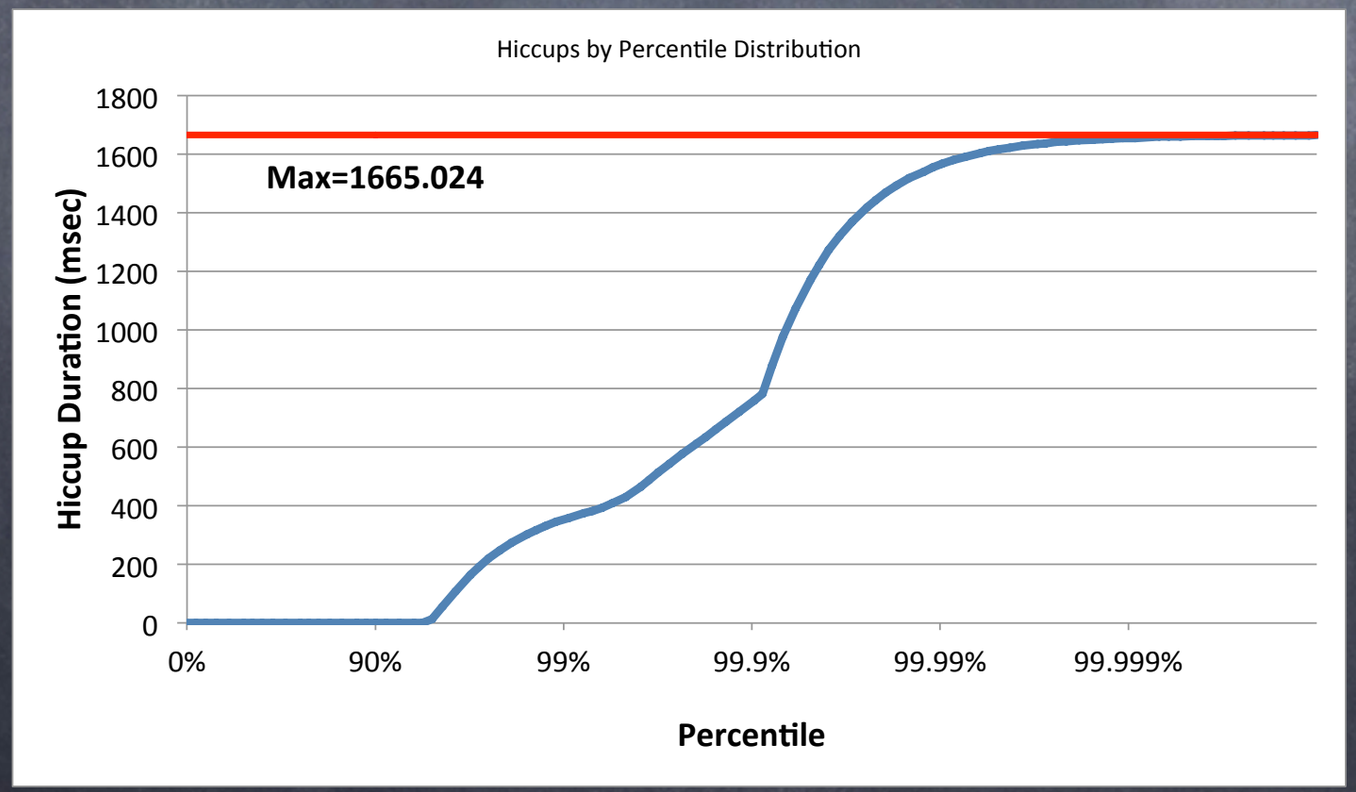
jHiccup:
A tool that measures and reports
(as your application is running)
if your JVM is actually running
all the time

AZUL SYSTEMS

Discontinuities in Java platform execution - Easy To Measure

I call these "hiccups"





A telco App with a bit of a "problem"

AZUL SYSTEMS

Fun with jHiccup

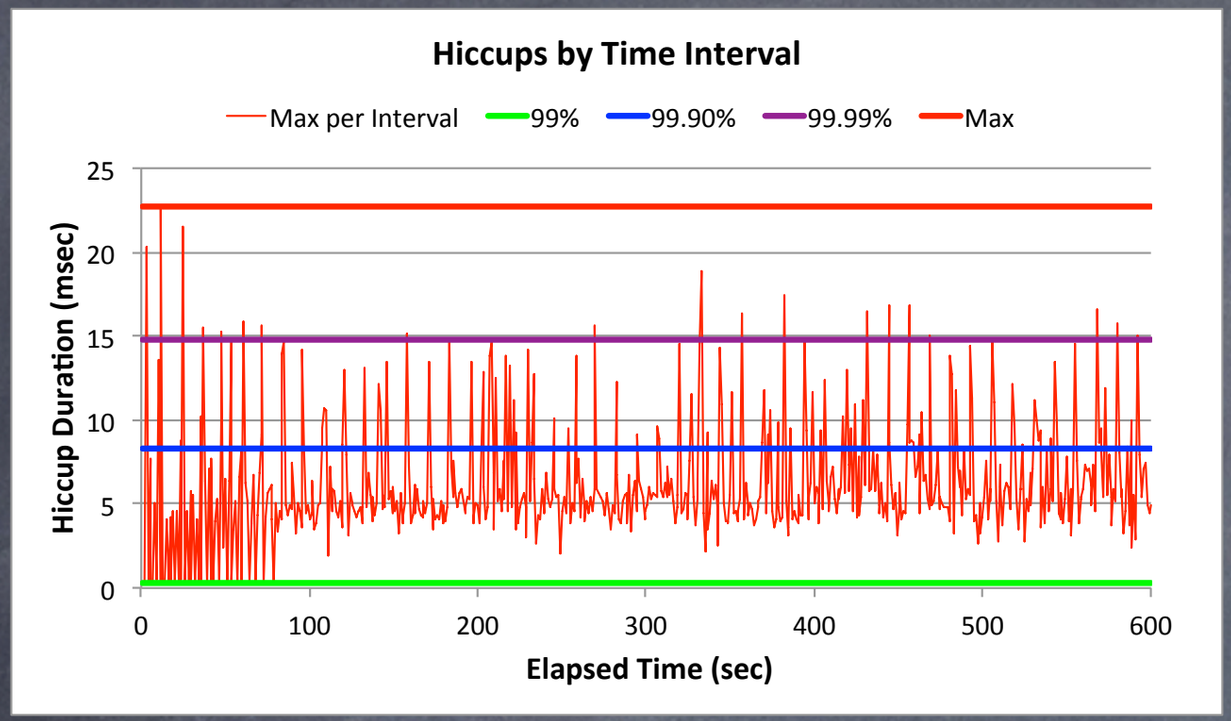
Charles Nutter @headius 20 Jan

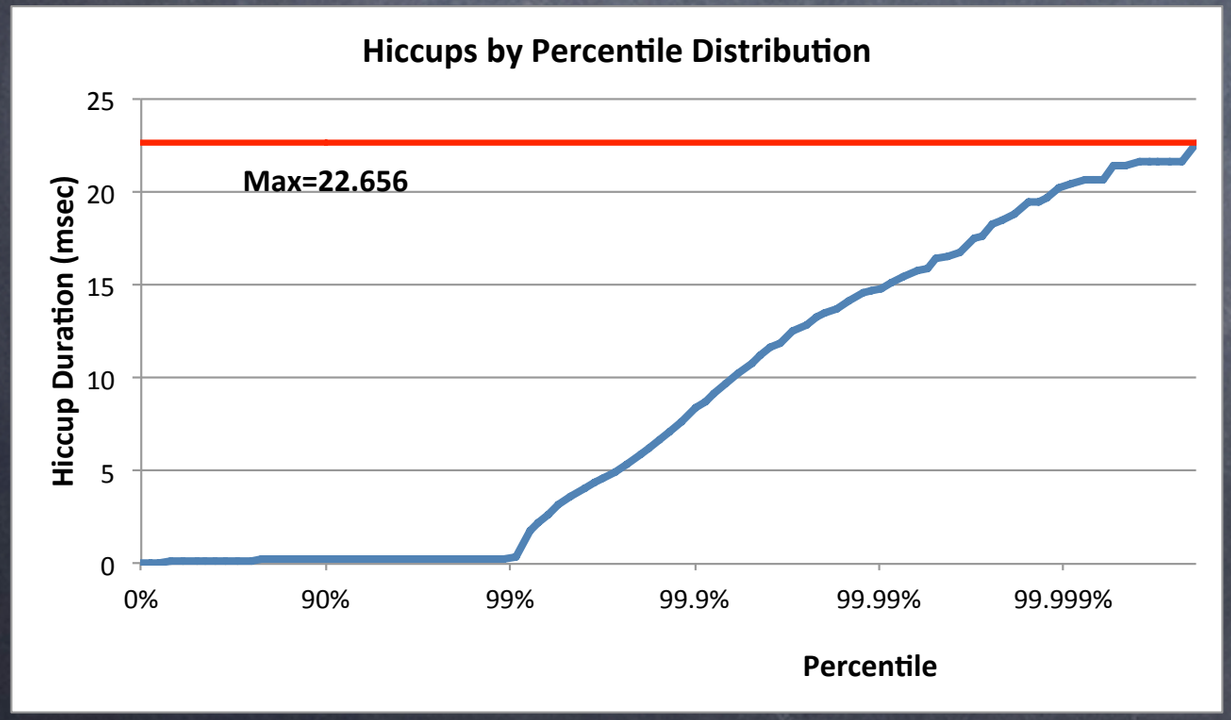
jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)

Retweeted by Gil Tene

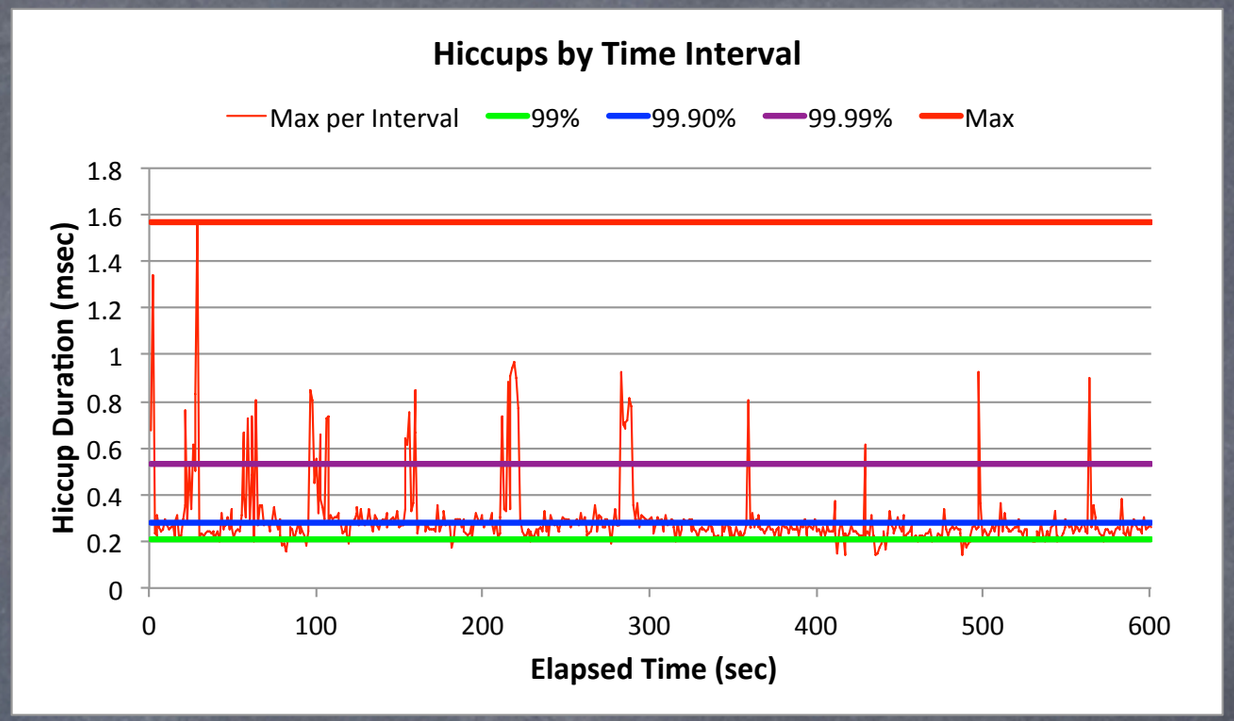
AZUL SYSTEMS

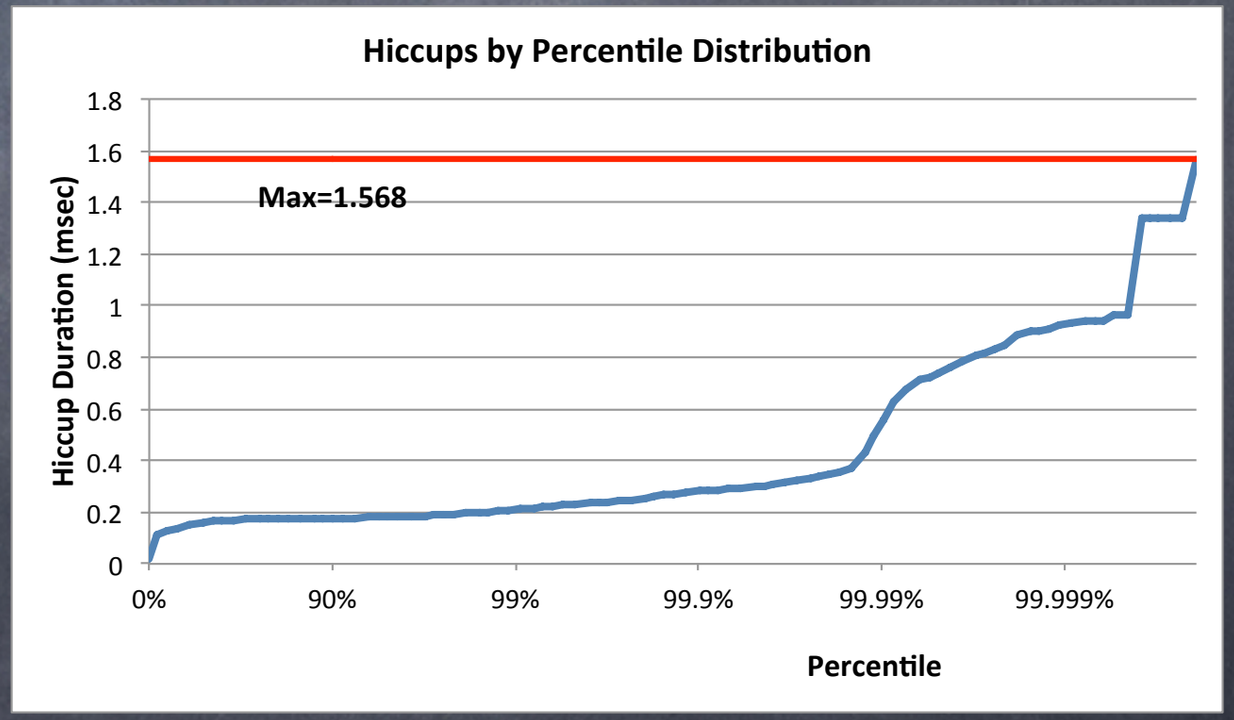
Oracle HotSpot (pure newgen)





Zing

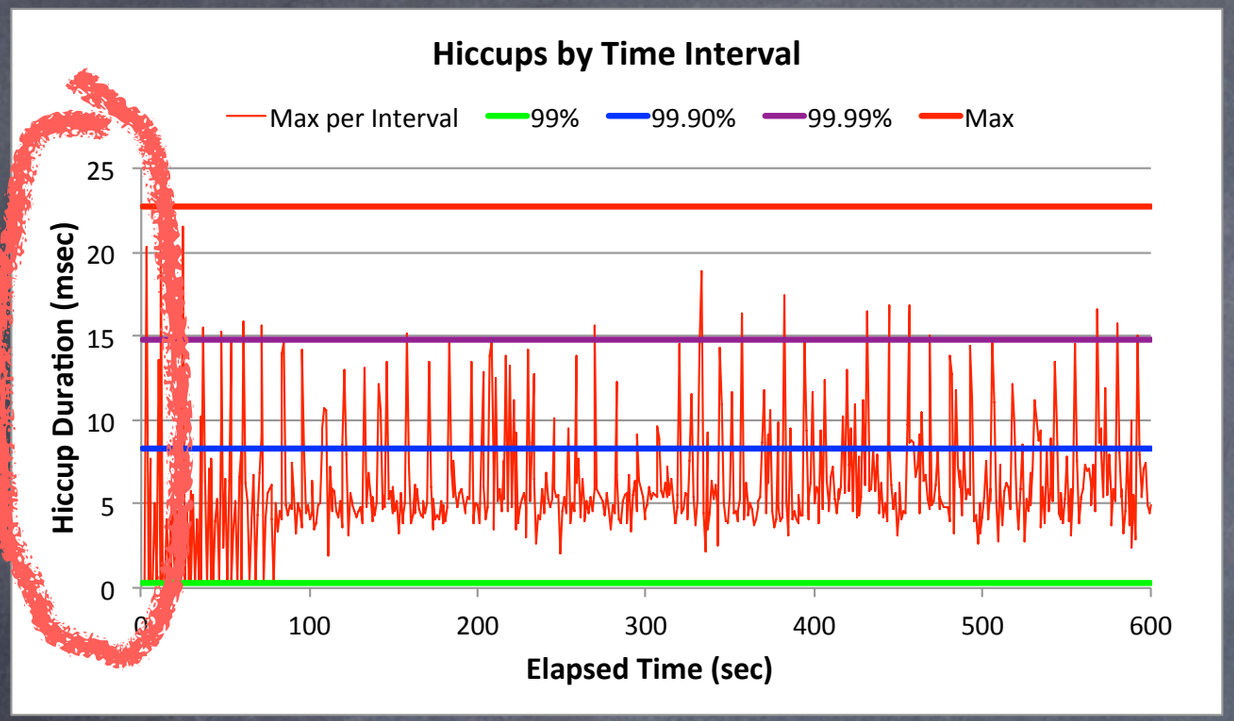


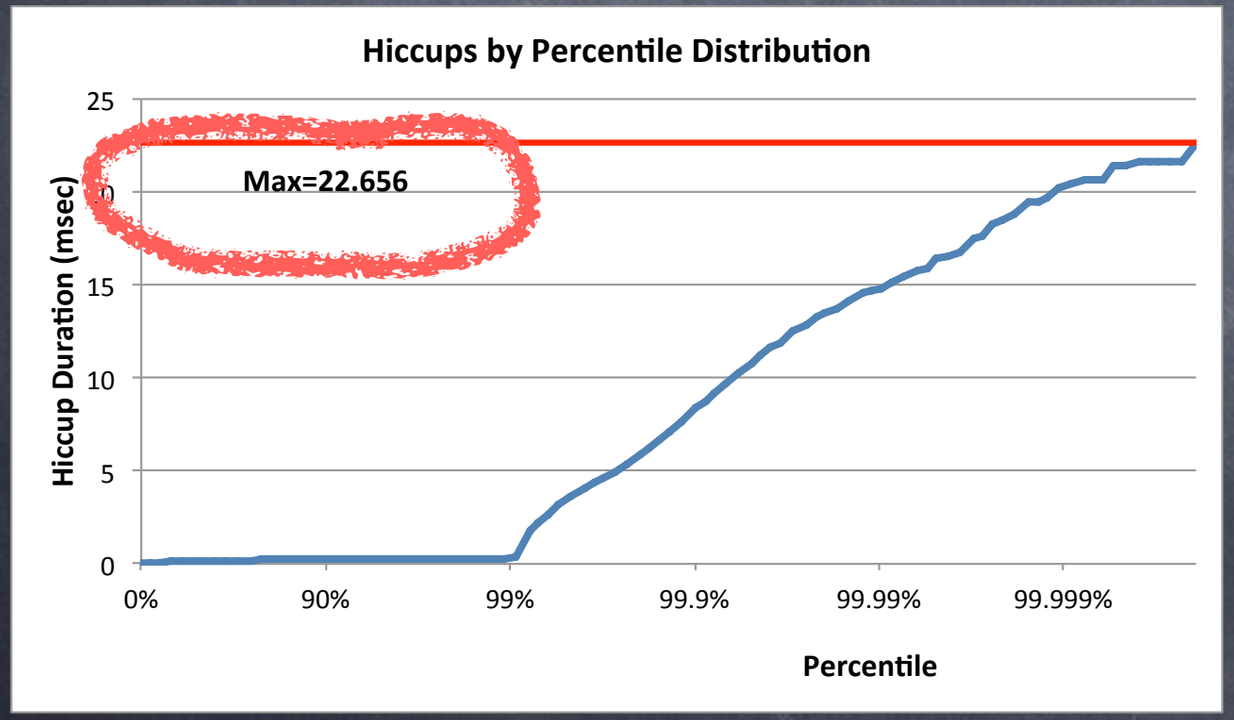


Low latency trading application

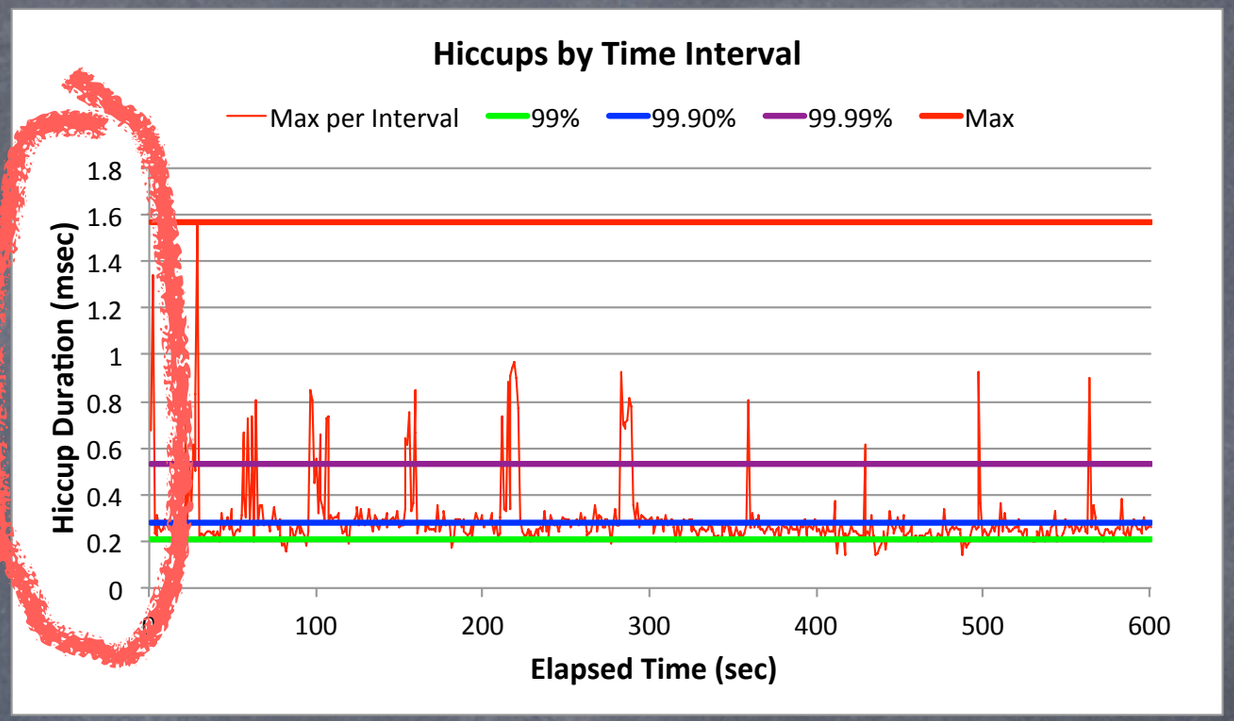
AZUL SYSTEMS

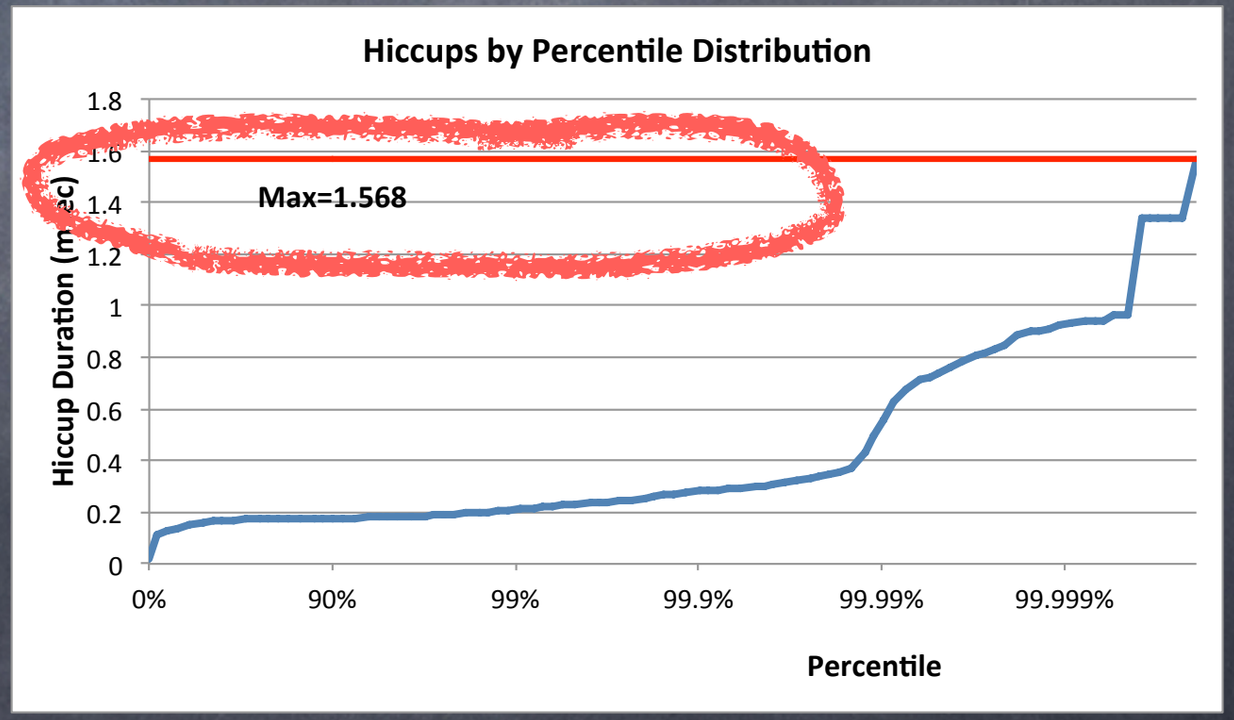
Oracle HotSpot (pure newgen)





Zing

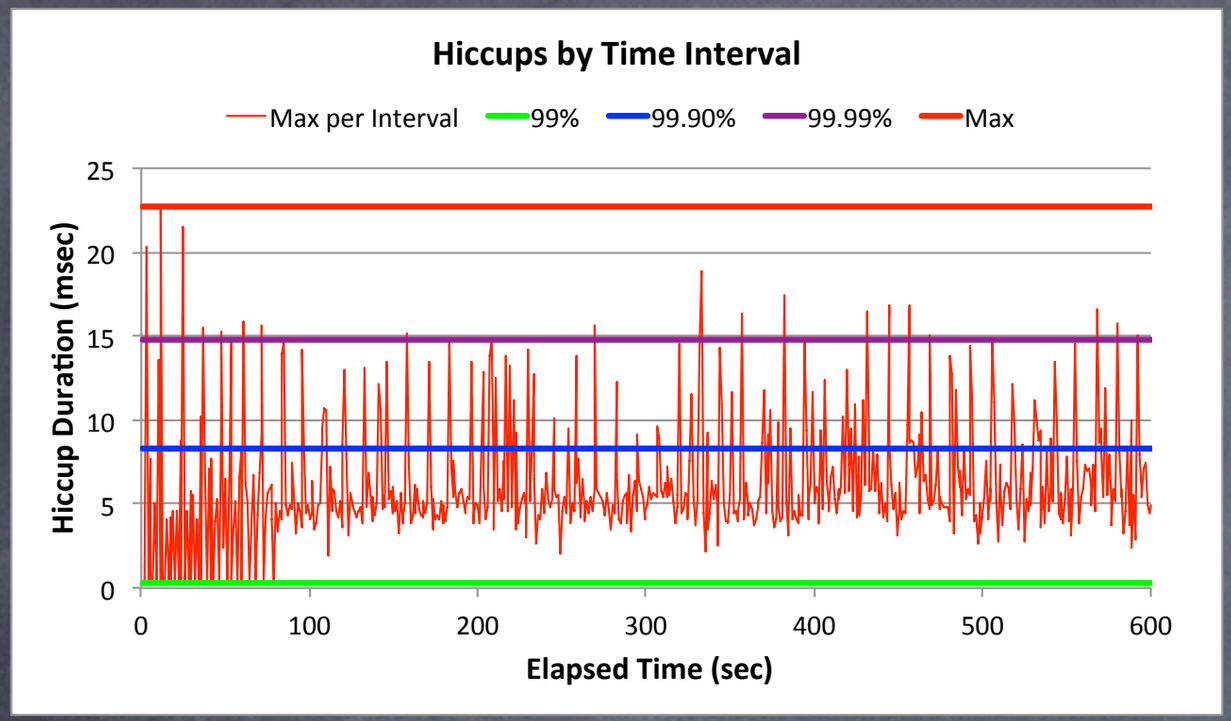


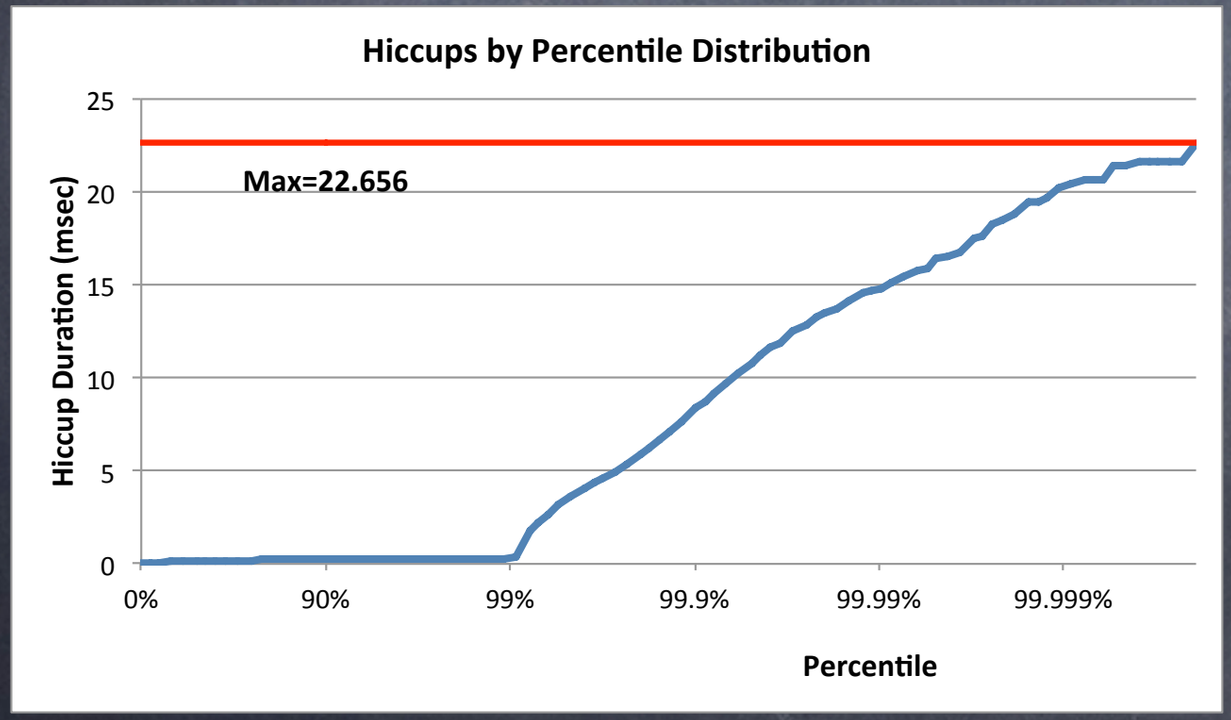


Low latency trading application

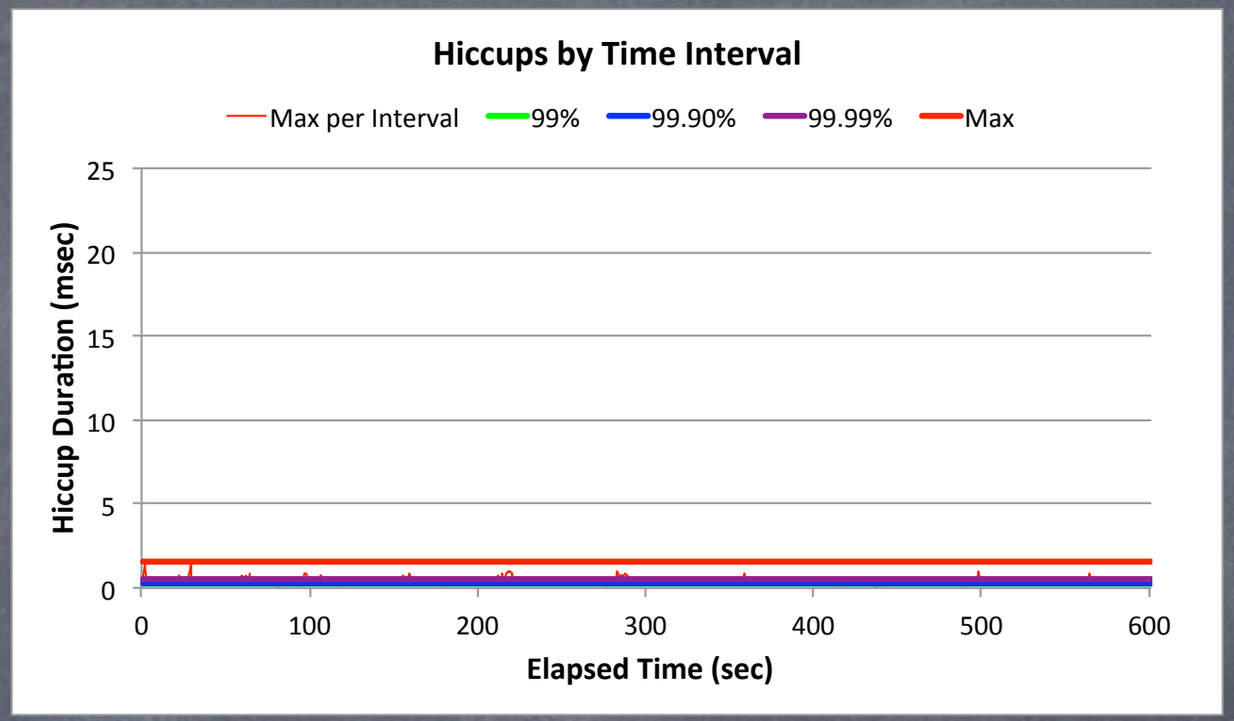
AZUL SYSTEMS

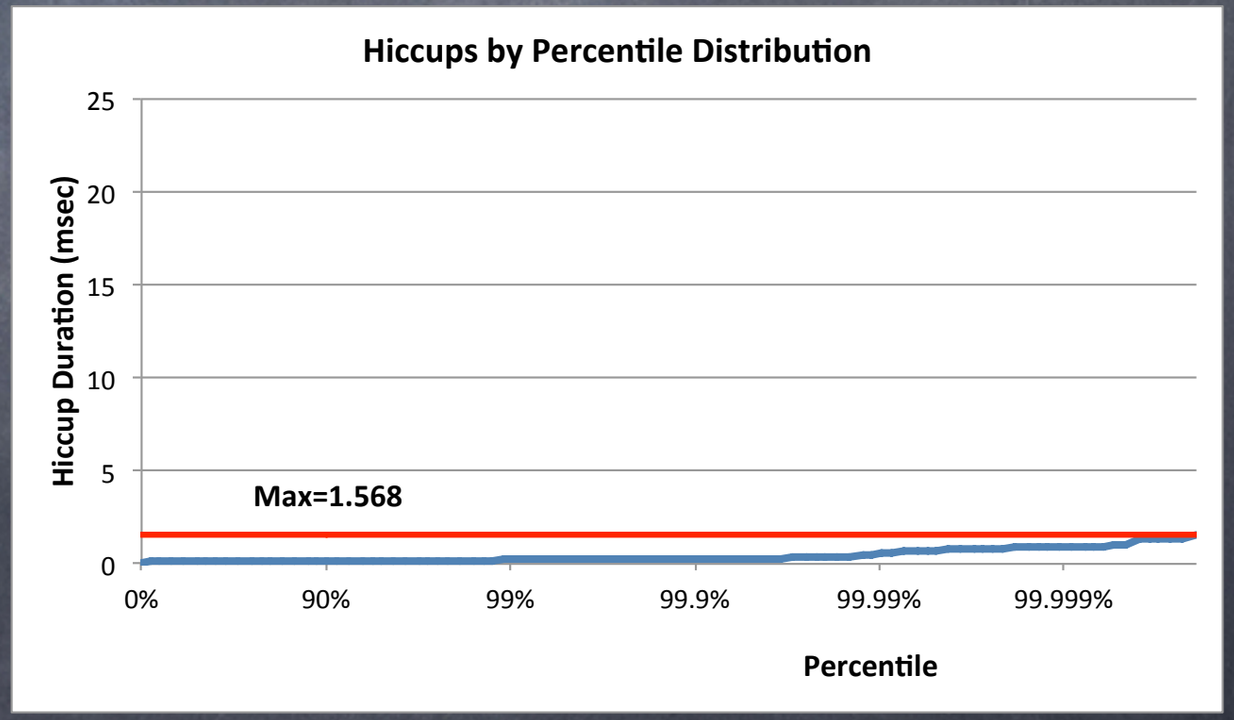
Oracle HotSpot (pure newgen)





Zing





Low latency - Drawn to scale

AZUL SYSTEMS

Lets not forget about GC tuning

AZUL SYSTEMS®

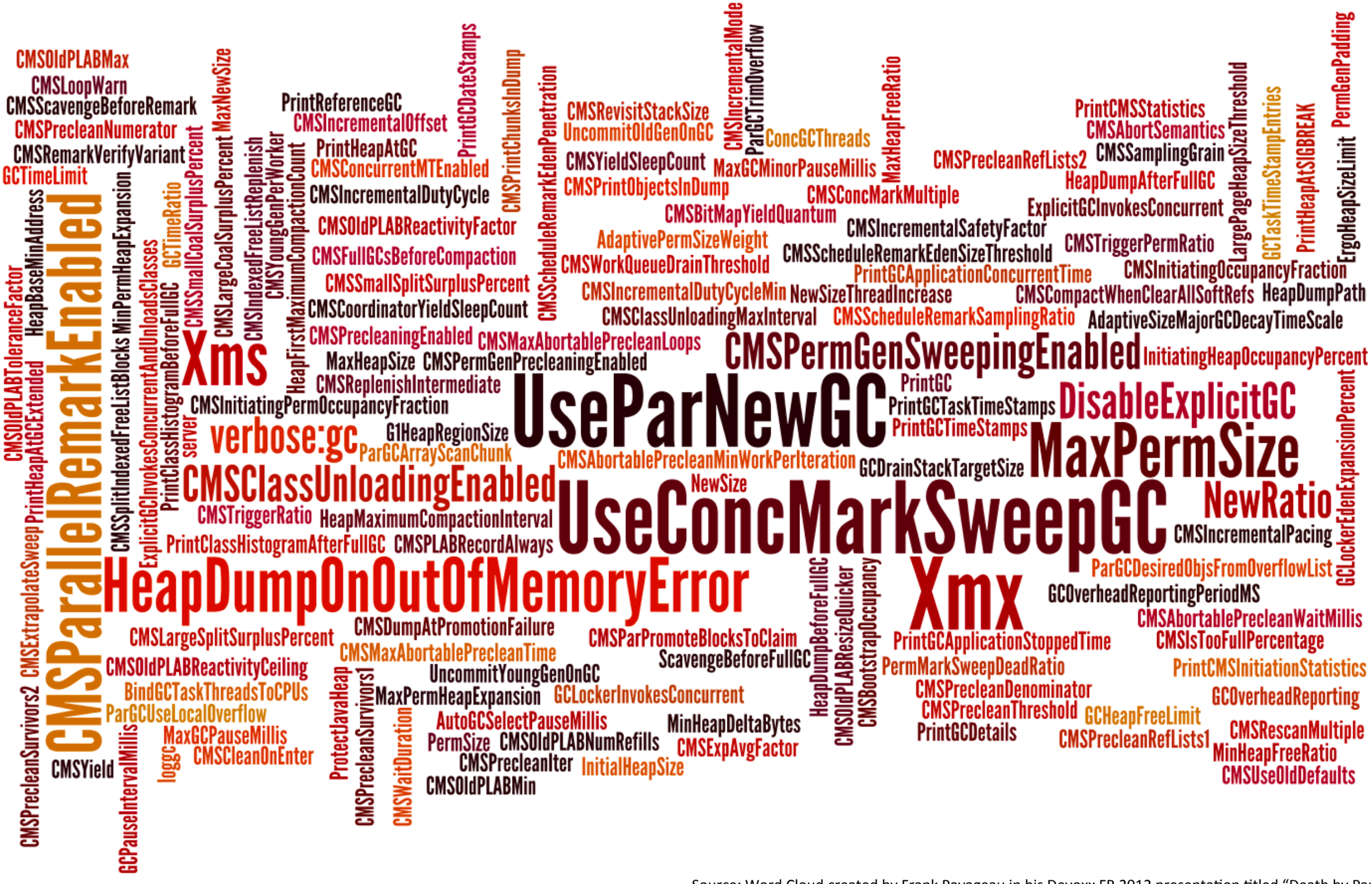
Java GC tuning is "hard"...

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 GC tuning flags



Source: Word Cloud created by Frank Pavageau in his Devoxx FR 2012 presentation titled “Death by Pauses”

The complete guide to Zing GC tuning

java -Xmx40g

AZUL SYSTEMS®

So what's next?

GC is only the biggest problem...

AZUL SYSTEMS®

JVMs make many tradeoffs often trading speed vs. outliers

- Some speed techniques come at extreme outlier costs
  - E.g. ("regular") biased locking
  - E.g. counted loops optimizations
- Deoptimization
- Lock deflation
- Weak References, Soft References, Finalizers
- Time To Safe Point (TTSP)

AZUL SYSTEMS

Time To Safepoint (TTSP)
Your new #1 enemy

- (Once GC itself was taken care of)
- Many things in a JVM (still) use a global safepoint
  - All threads brought to a halt, at a "safe to analuze" point in code, and then released after work is done.
  - E.g. GC phase shifts, Deoptimization, Class unloading, Thread Dumps, Lock Deflation, etc. etc.
- A single thread with a long time-to-safepoint path can cause an effective pause for all other threads
- Many code paths in the JVM are long...

AZUL SYSTEMS

Time To Safepoint (TTSP)
the most common examples

- Array copies and object clone()
- Counted loops
- Many other other variants in the runtime...
- Measure, Measure, Measure...
- Zing has a built-in TTSP profiler
- At Azul, I walk around with a 0.5msec stick...

AZUL SYSTEMS

OS related stuff (once GC and TTSP are taken care of)

- OS related hiccups tend to dominate once GC and TTSP are removed as issues.
- Take scheduling pressure seriously (Duh?)
- Hyper-threading (good? bad?)
- Swapping (Duh!)
- Power management
- Transparent Huge Pages (THP).
- ...

AZUL SYSTEMS

Takeaway: In 2013, "Real" Java is finally viable for low latency applications

- GC is no longer a dominant issue, even for outliers
- 2-3msec **worst observed case** with "easy" tuning
- < 1 msec **worst observed case** is very doable
- No need to code in special ways any more
  - You can finally use "real" Java for everything
  - You can finally 3rd party libraries without worries
  - You can finally use as much memory as you want
  - You can finally use regular (good) programmers

AZUL SYSTEMS

One-liner Takeaway:

Zing: A cure for the Java hiccups

AZUL SYSTEMS

Q & A

One-liner Takeaway:

Zing: A cure for the Java hiccups

jHiccup:

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

AZUL SYSTEMS