Enabling Java in Latency Sensitive Environments

Matt Schuetze

Azul Director of Product Management

Gateway Java Users Group

St. Louis, Missouri

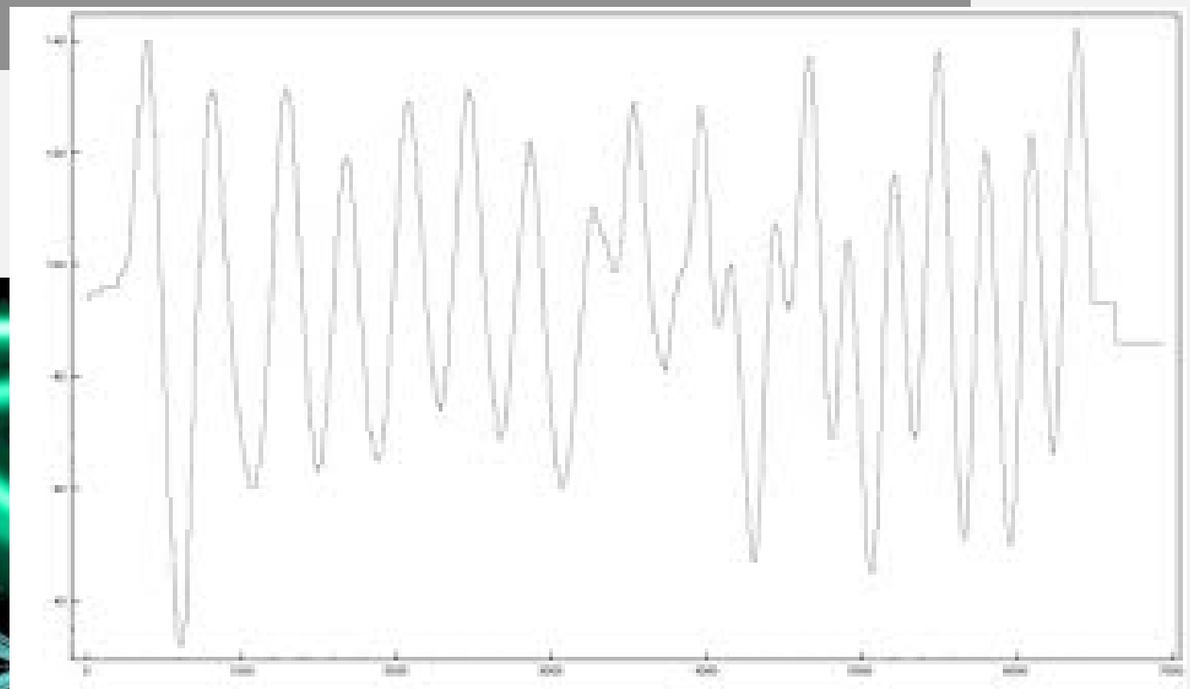
@azulsystems | azulsystems.com

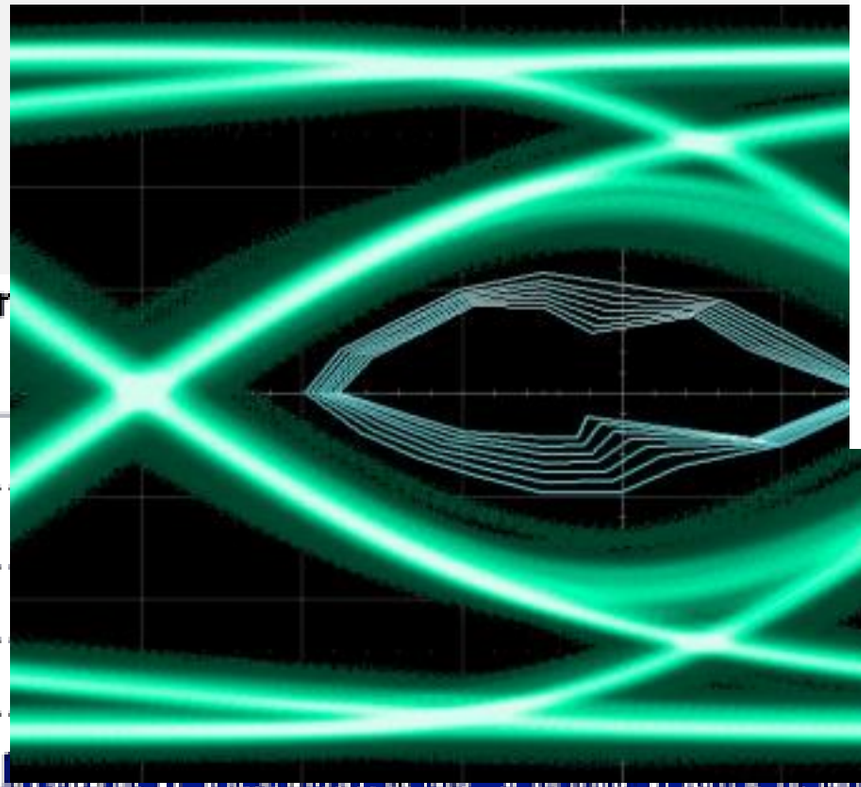
High Level Agenda

Welcome to all Gateway JUG members!

- Intro, jitter vs. JITTER
- 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 2015 looks like for Low Latency Java developers
- Real World Case Studies

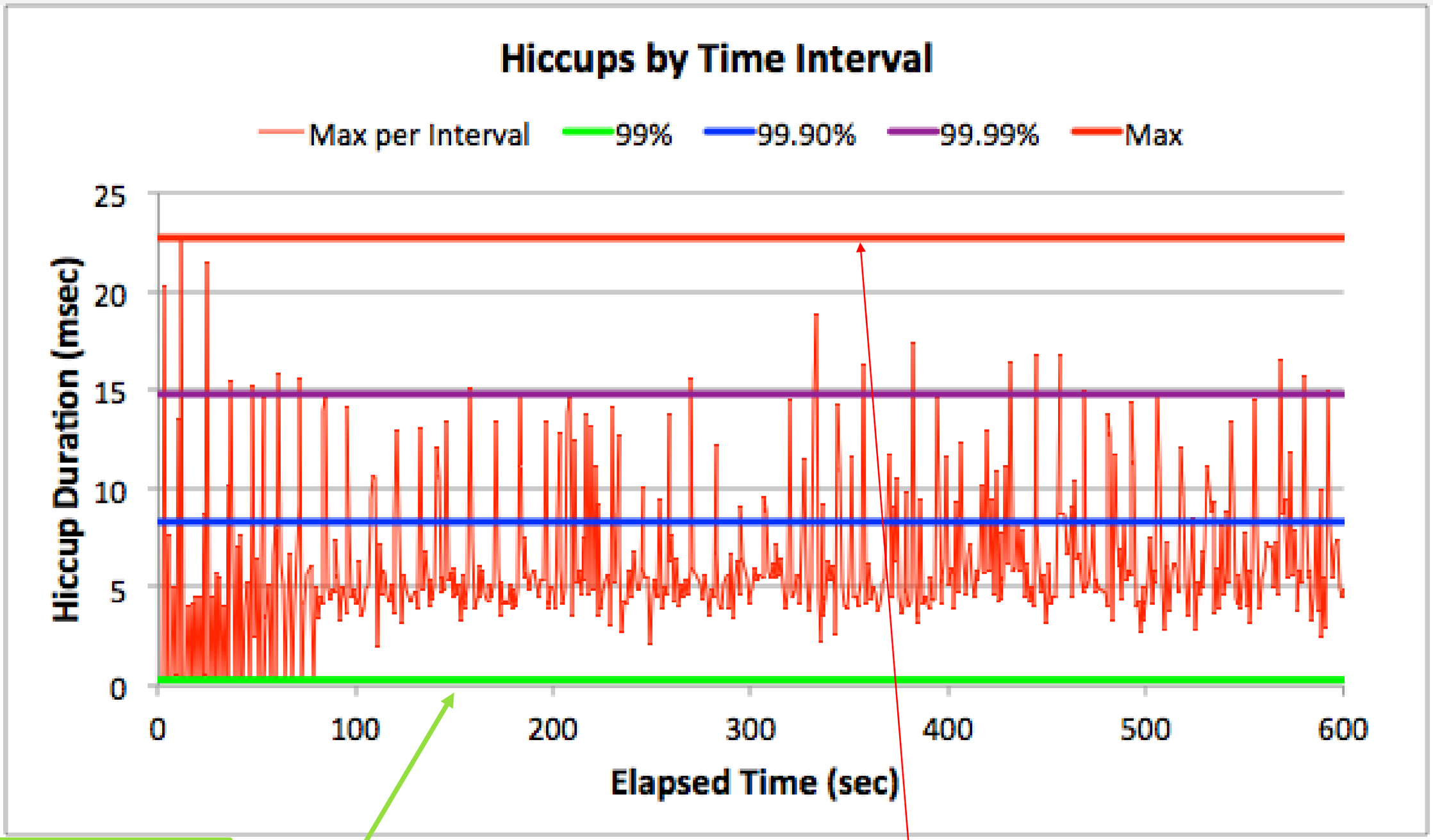
This is Jitter





Is “jitter” a proper word for this?

Answer: no its not jitter at all. It’s phase changes.



99%‘ile is ~60 usec

Max is ~30,000% higher than “typical”

About Azul Systems

Zing, Zulu, and everything about Java Virtual Machines

- 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)**
- Certified OpenJDK (Zulu)
- Known for Low Latency, Consistent execution, and Large data set excellence

Java in the low latency world

Java in a low latency world

Yep, Java latencies are goin' down for real…

- 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, Time-to-performance ...
- Leverage, ecosystem, ability to hire

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++ ...

So what is the problem?
Is Java Slow?

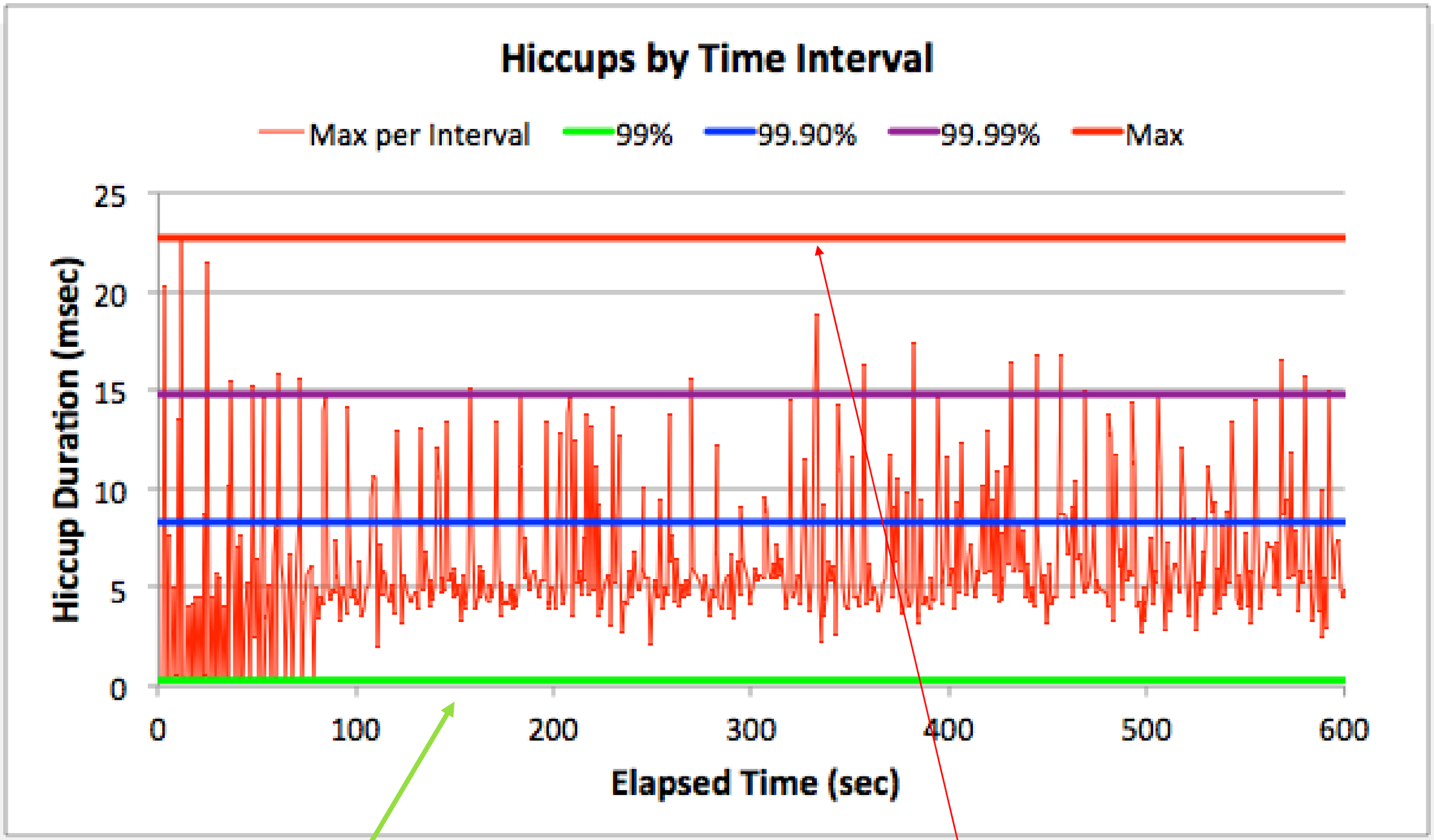
- No
- A good programmer will get roughly the same speed from both Java and C++
- A bad programmer won't get you fast code on either
- The 50%'ile and 90%'ile are typically excellent...
- It's those pesky occasional stutters and stammers and stalls that are the problem...
- Ever hear of Garbage Collection?

Java’s Achilles heel

Stop-The-World Garbage Collection: How bad is it?

- 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
- In turn, directly related to throughput
- 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.

STW-GC effects in a low latency application



99%‘ile is ~60 usec

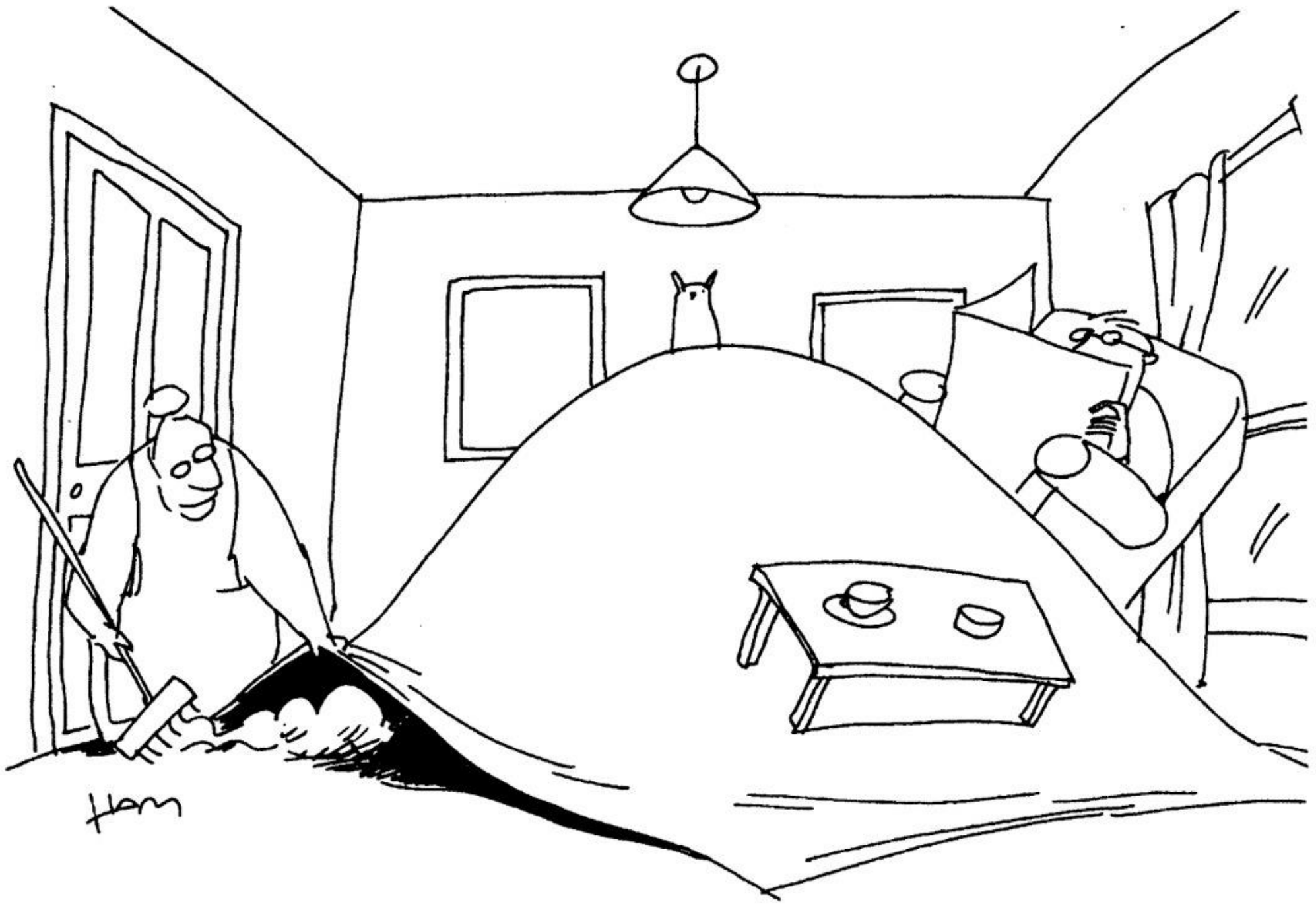
Max is ~30,000% higher than “typical”

One way to deal with Stop-The-World GC

I cannot see it, so it cannot see me.

More Stop-The-World GC avoidance

Time for a bigger rug.



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 lose out on much of Java.

Another way to cope: “Creative Language”

Drawn from evil vendor marketing literature

“Guarantee a worst case of 5 msec, 99% of the time”

Translation: “1% will be far worse than worst case”

“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 low latency (Java) developers get for all their effort?

- They still see pauses (usually ranging to tens of msec)
- But they 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”...

There is a fundamental problem:

Stop-The-World GC mechanisms are contradictory to the fundamental requirements of low latency & low jitter apps

Sustainable Throughput

The throughput achieved while safely maintaining service levels

It’s an industry-wide problem

It *was* an industry-wide problem

It’s 2015... Now we have Zing.

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"...

At Azul, STW-GC was addressed head-on

Trivia: Azul as a company founded predominantly around this one premise plaguing then Java servers

- We decided to focus on the right core problems
  - Scale & productivity being limited by responsiveness
  - Even “short” GC pauses are 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
  - Any STW fallback is a real-world failure

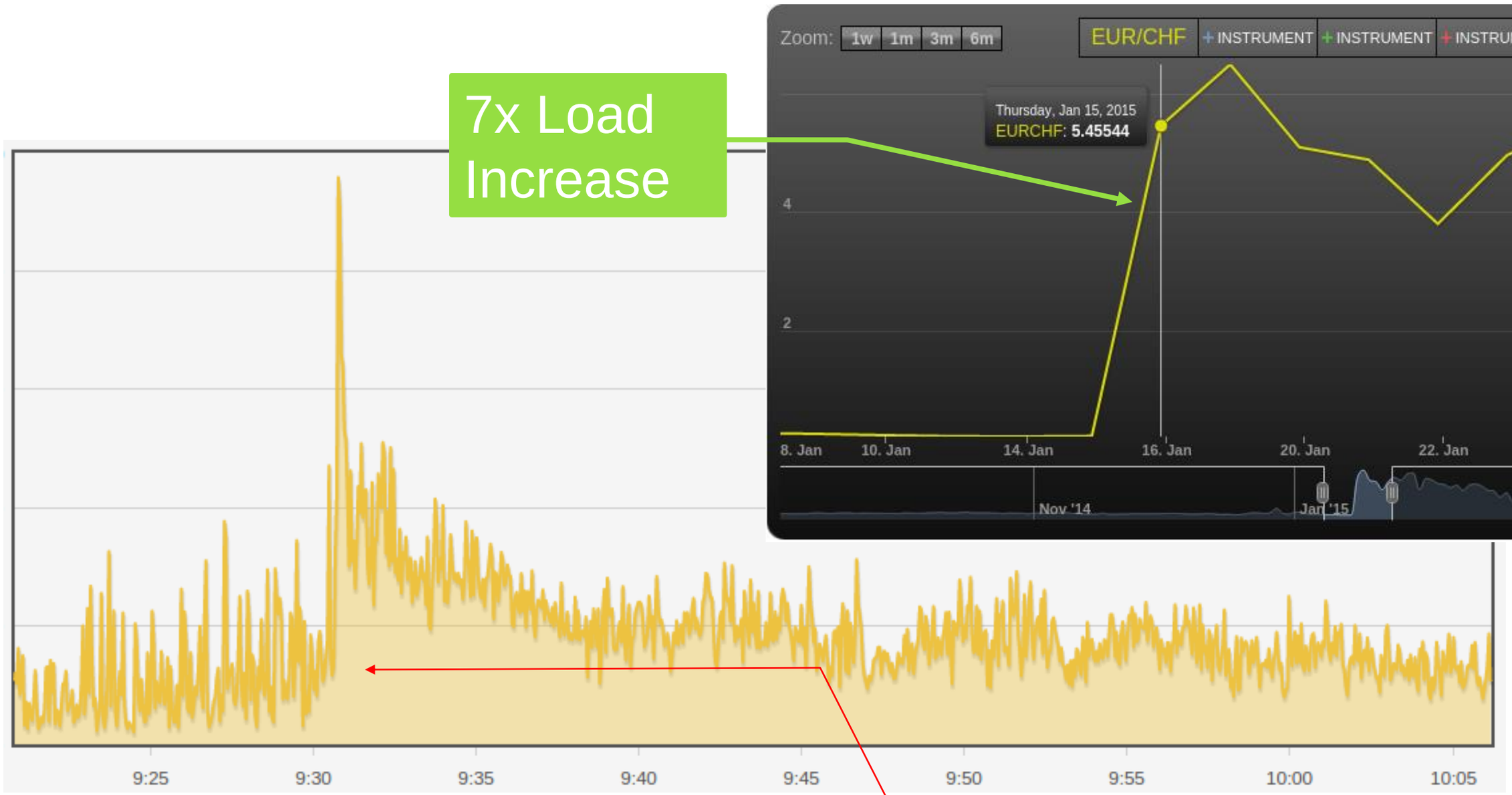
The Zing "C4" Collector
Continuously Concurrent Compacting Collector

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

Benefits

Stay Responsive

Even when traffic patterns change without warning

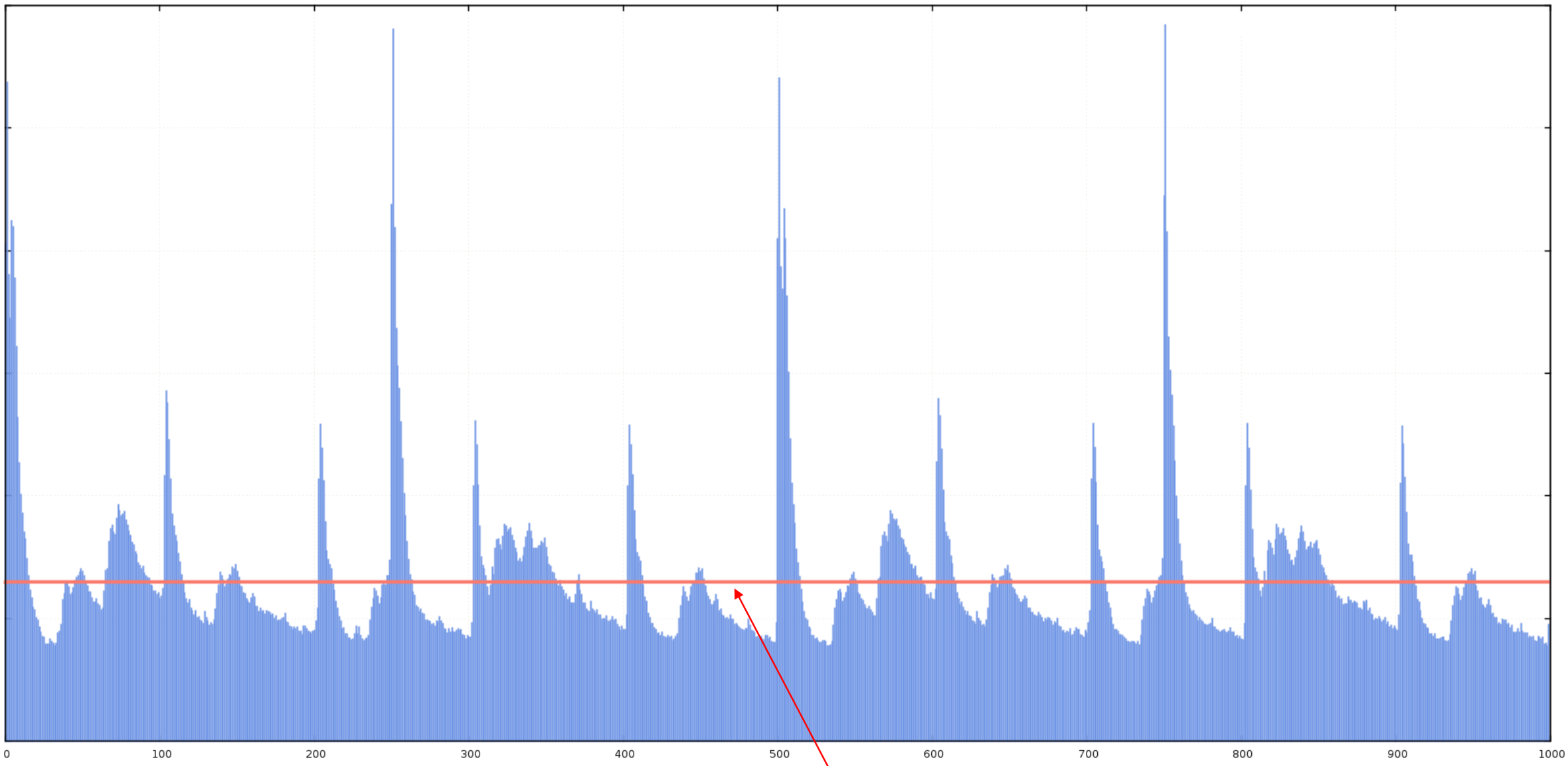


LMAX EXCHANGE™

AZUL SYSTEMS®

Handle Real World traffic patterns

One second view of transactions. Not constant. Not random either. Bursty is normal.



LMAX EXCHANGE™

Red line shows where order pricing arrival rate would be if constant

Achieve Measureable Benefits

From joint LMAX/Azul talk at QCon London, March 2015

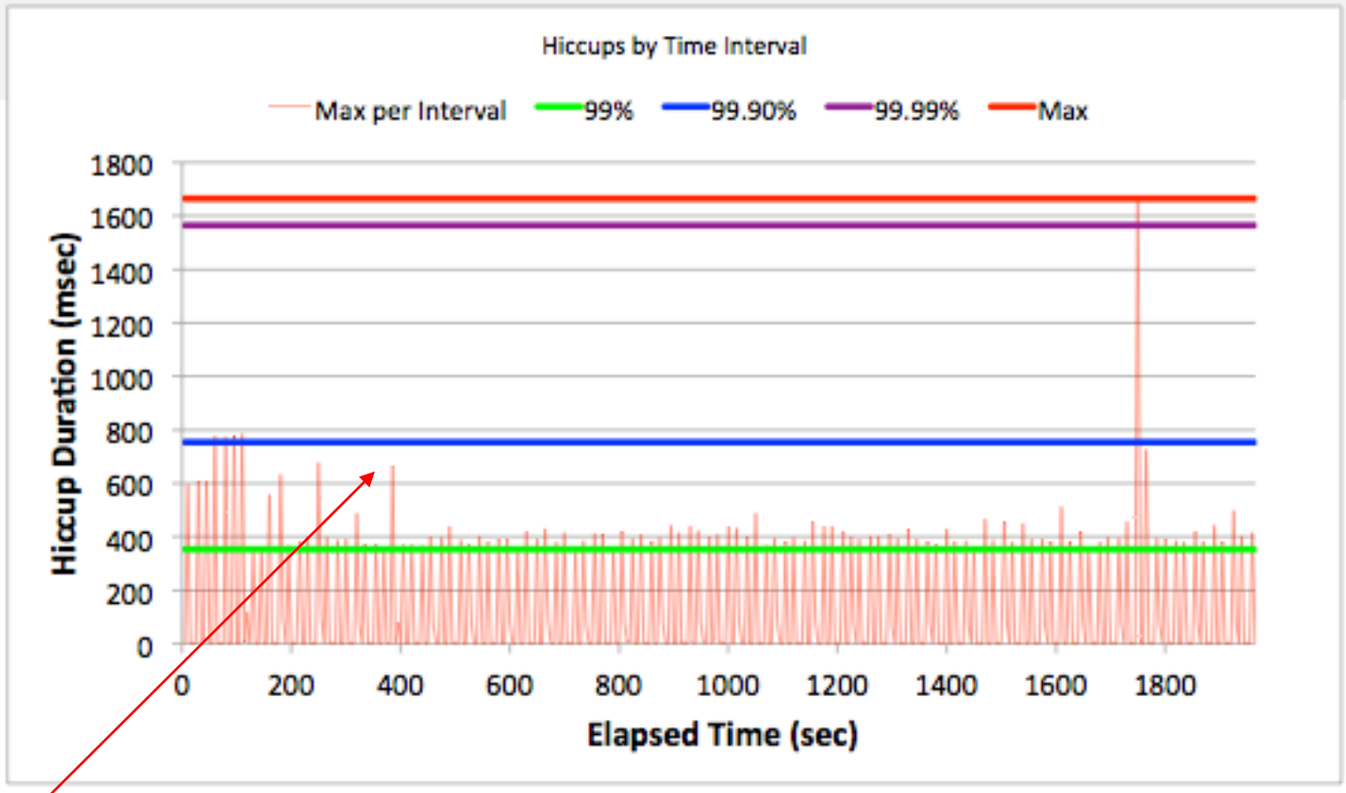
- Zing helped LMAX tame GC-related latency outlier pauses
  - Highly-engineered system: 4ms every 30 seconds down to 1ms every 2 hours
  - Less well-tuned system: 50ms every 30 seconds down to 3ms every 15 minutes
- No more unexpected/unwanted old-gen pauses caused by external behavior
  - CMS STW intra-day, generally ~500ms, gone
  - Removed source of backpressure on latency critical path.
  - Pre-Azul these would occur less predictably, but multiple times a week.

This is not “just Theory”

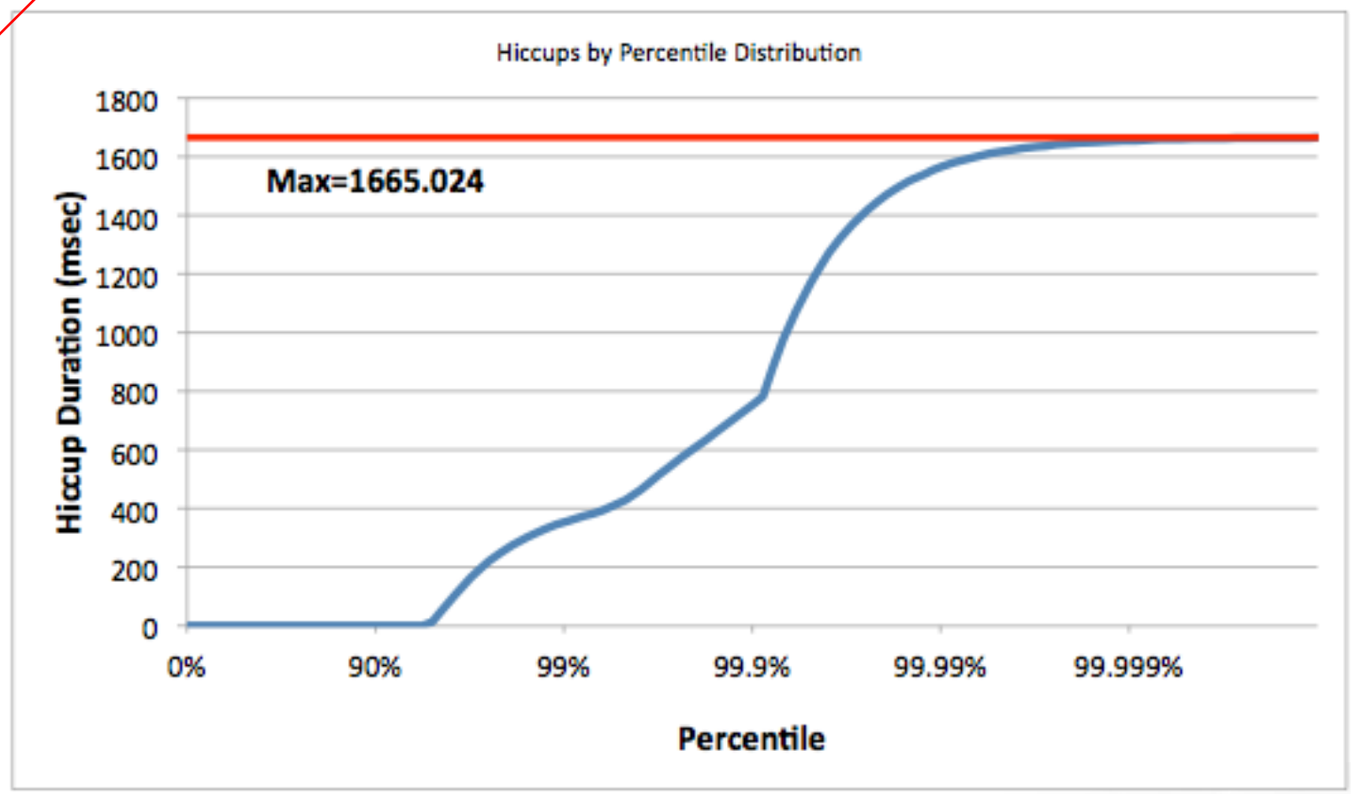
jHiccup

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

Discontinuities in Java execution - Easy To Measure

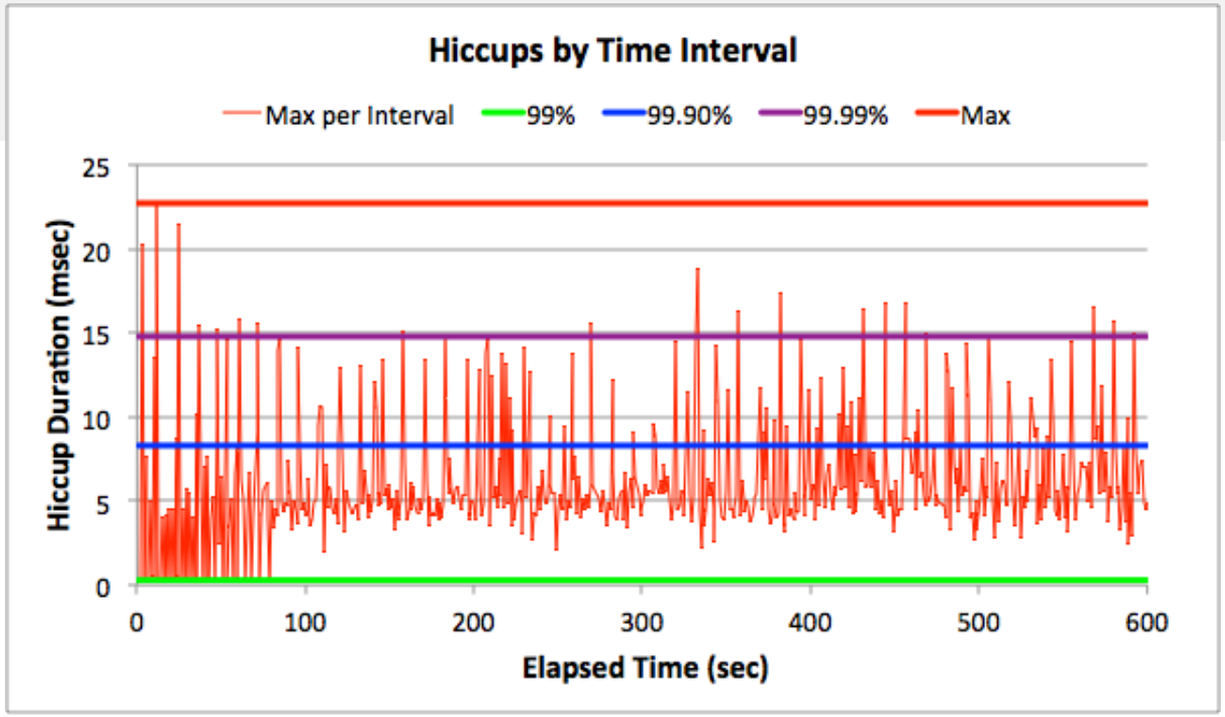


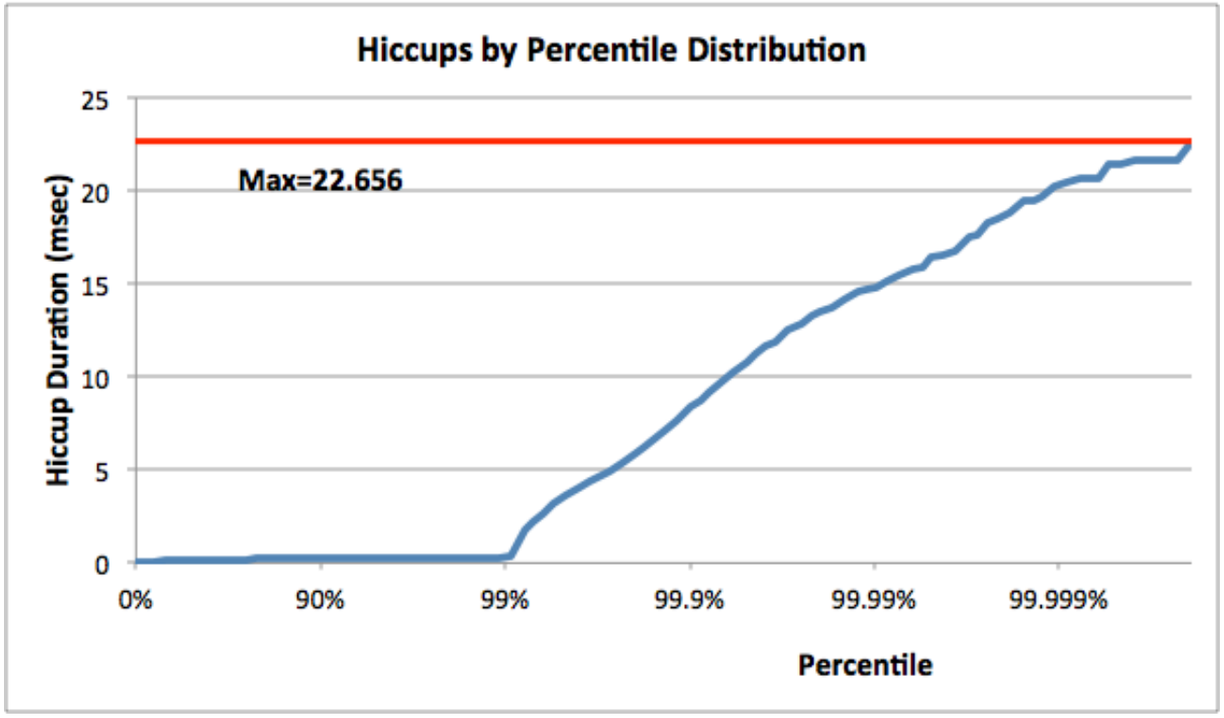
We call these **“hiccups”**



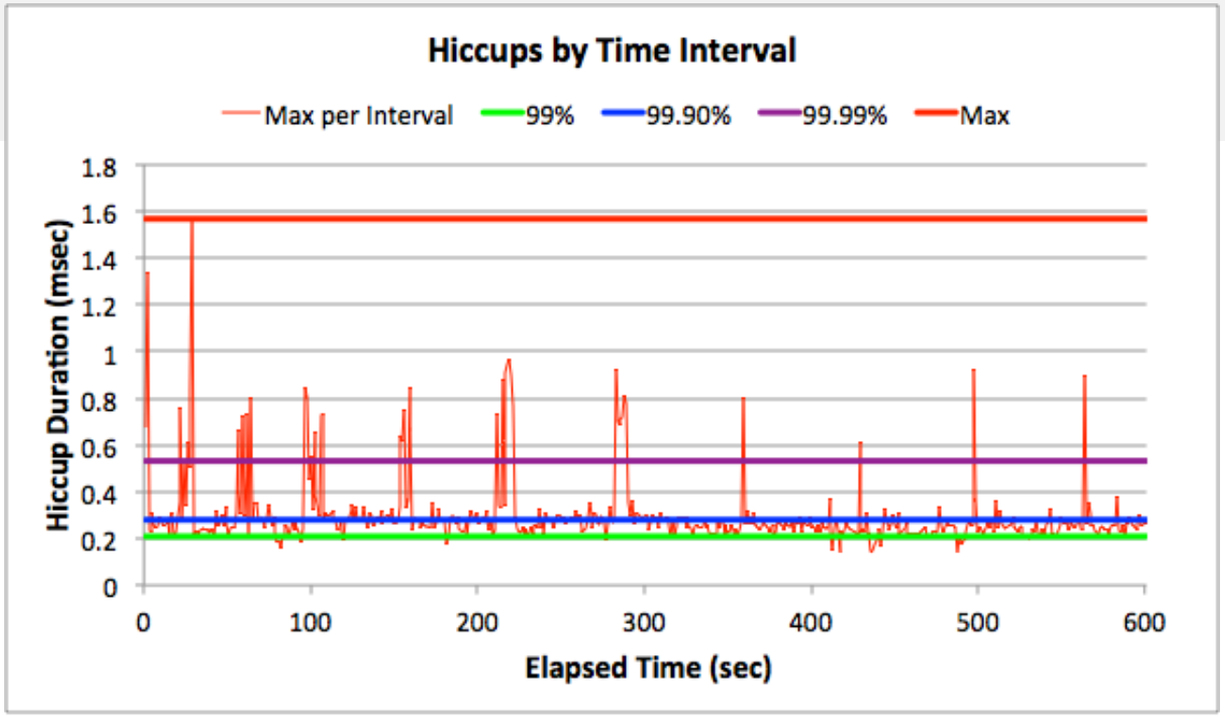
A telco App with a bit of a “problem”

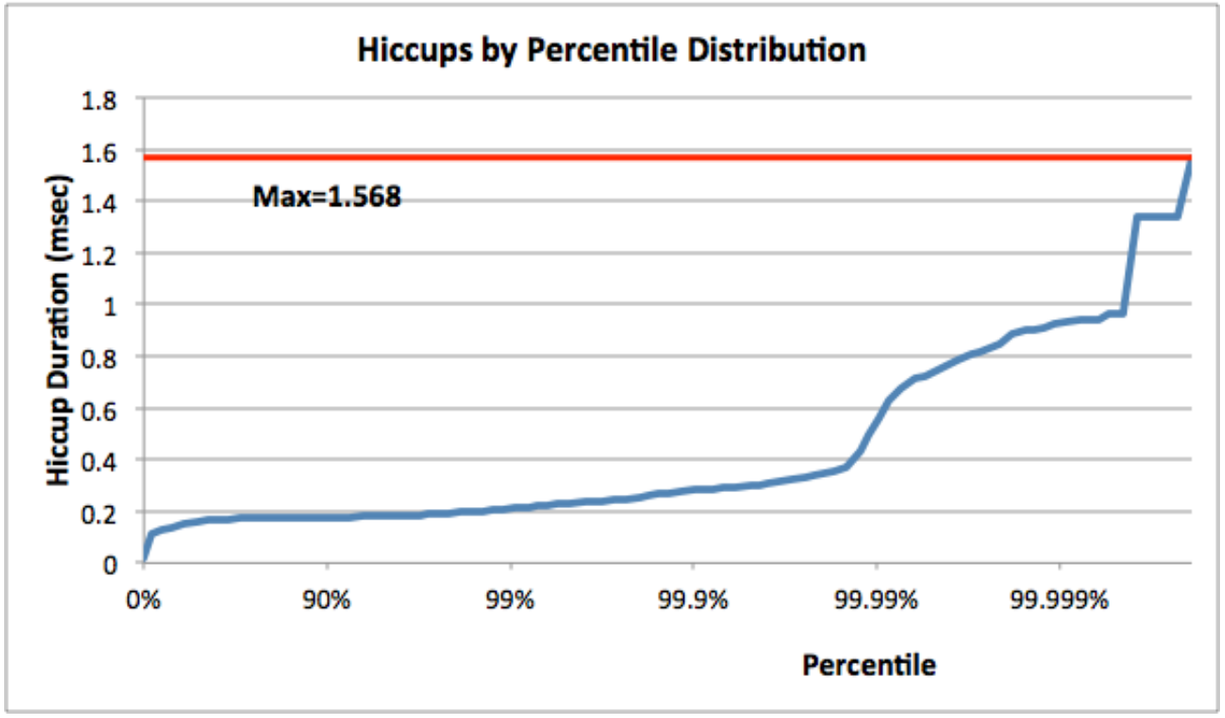
Oracle HotSpot (pure newgen)





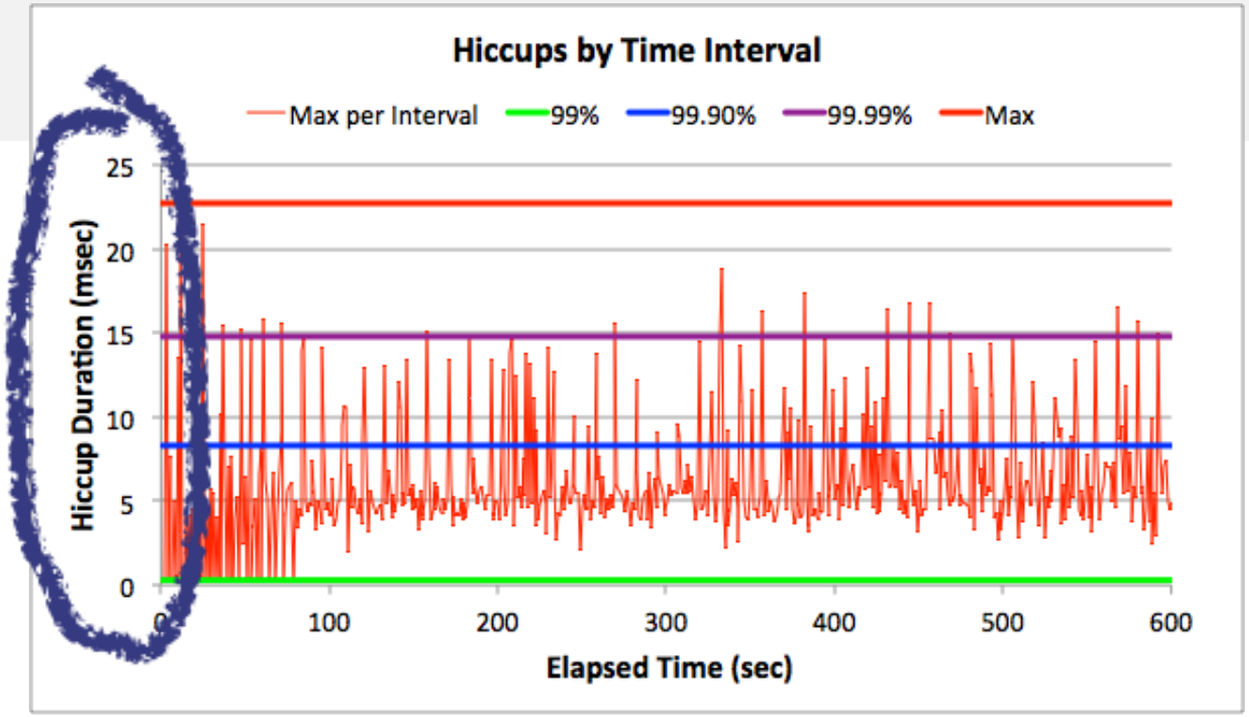
Zing

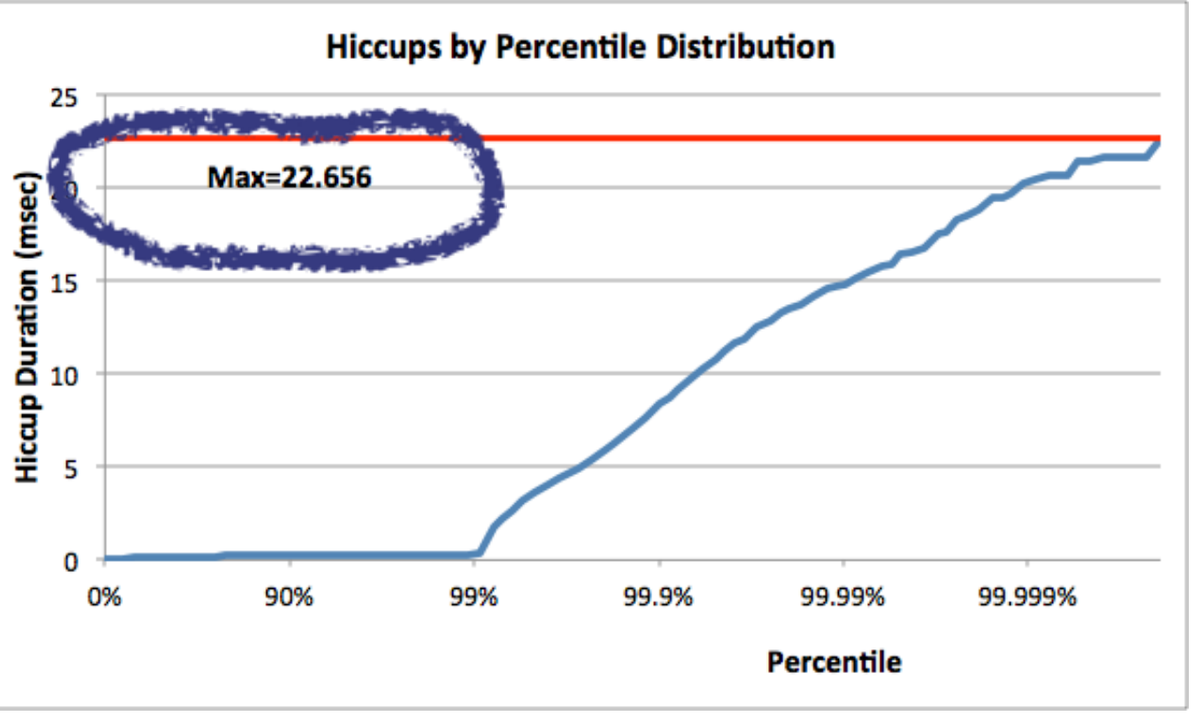




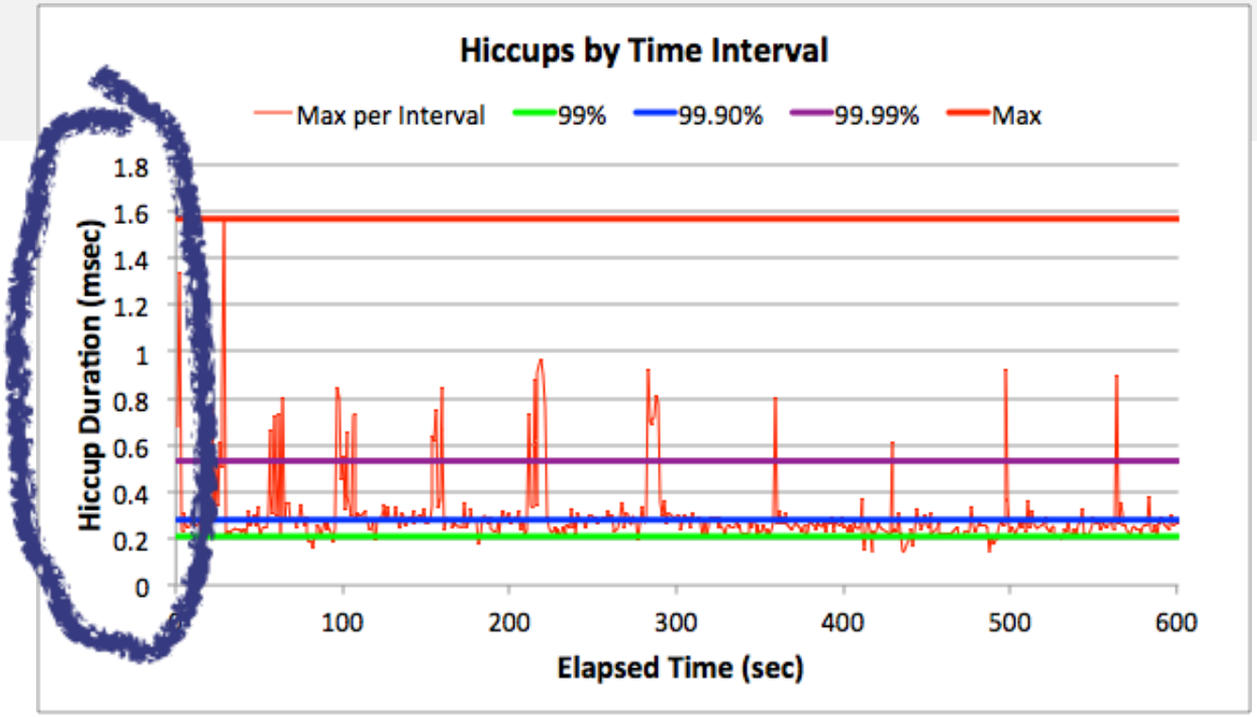
Low latency trading application

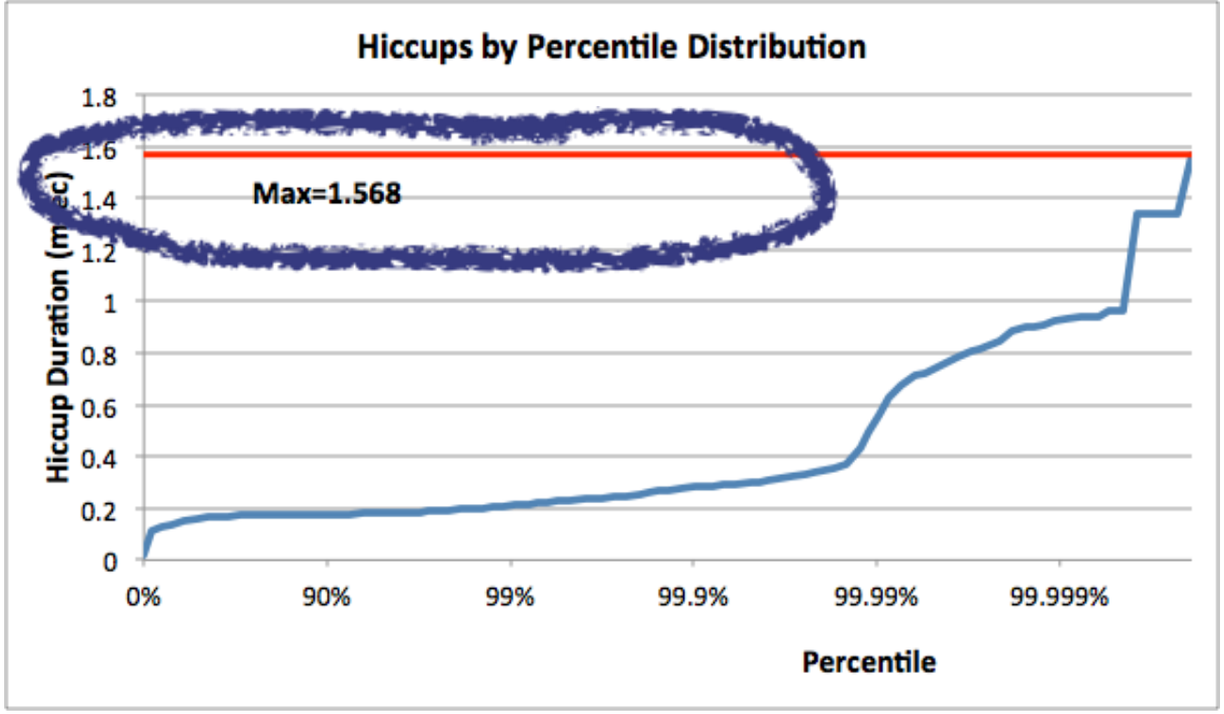
Oracle HotSpot (pure newgen)





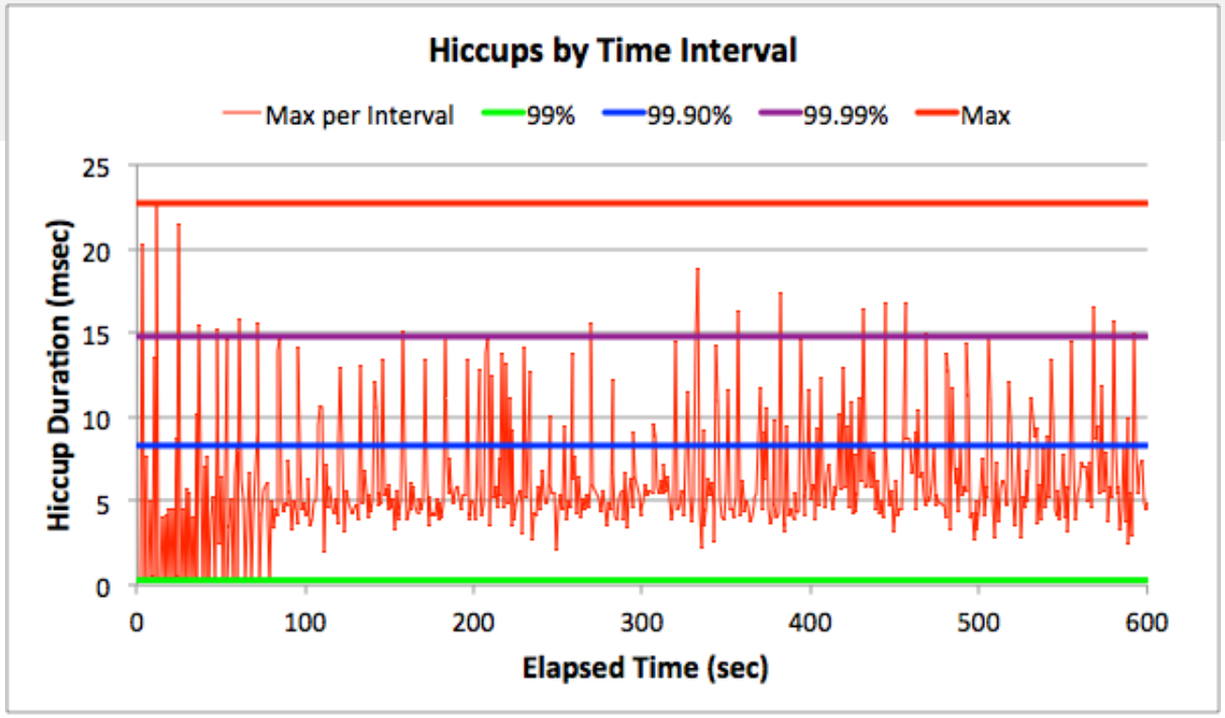
Zing

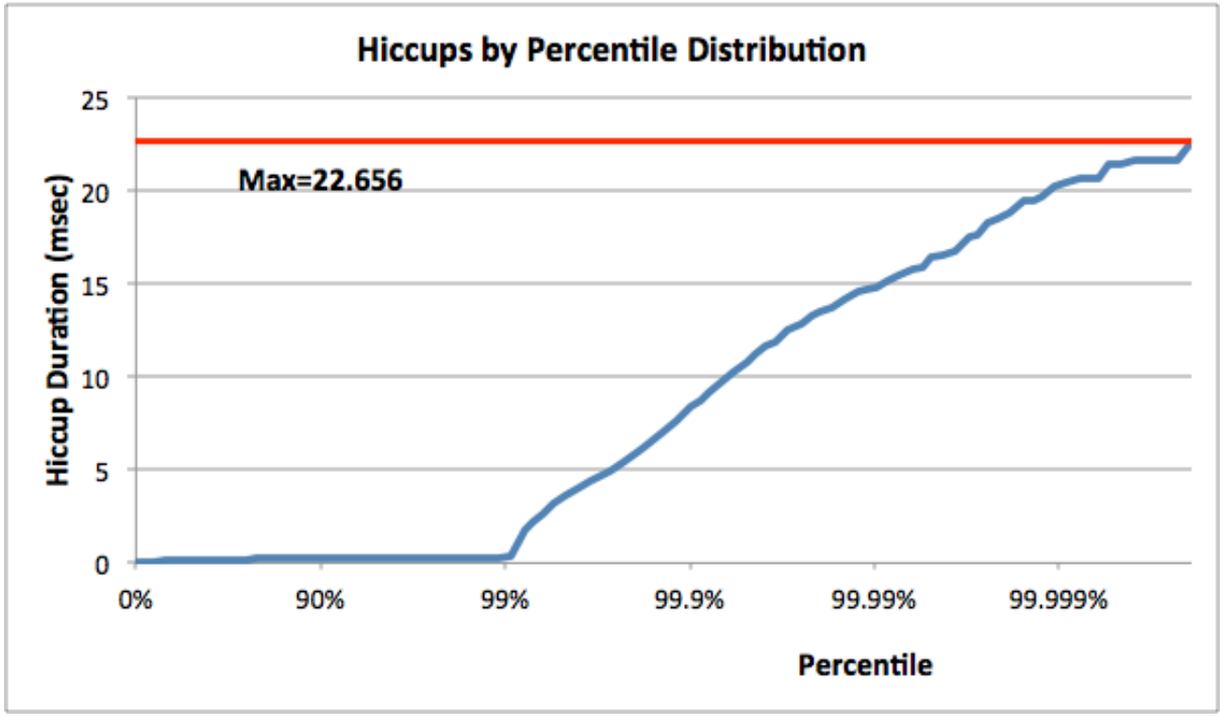




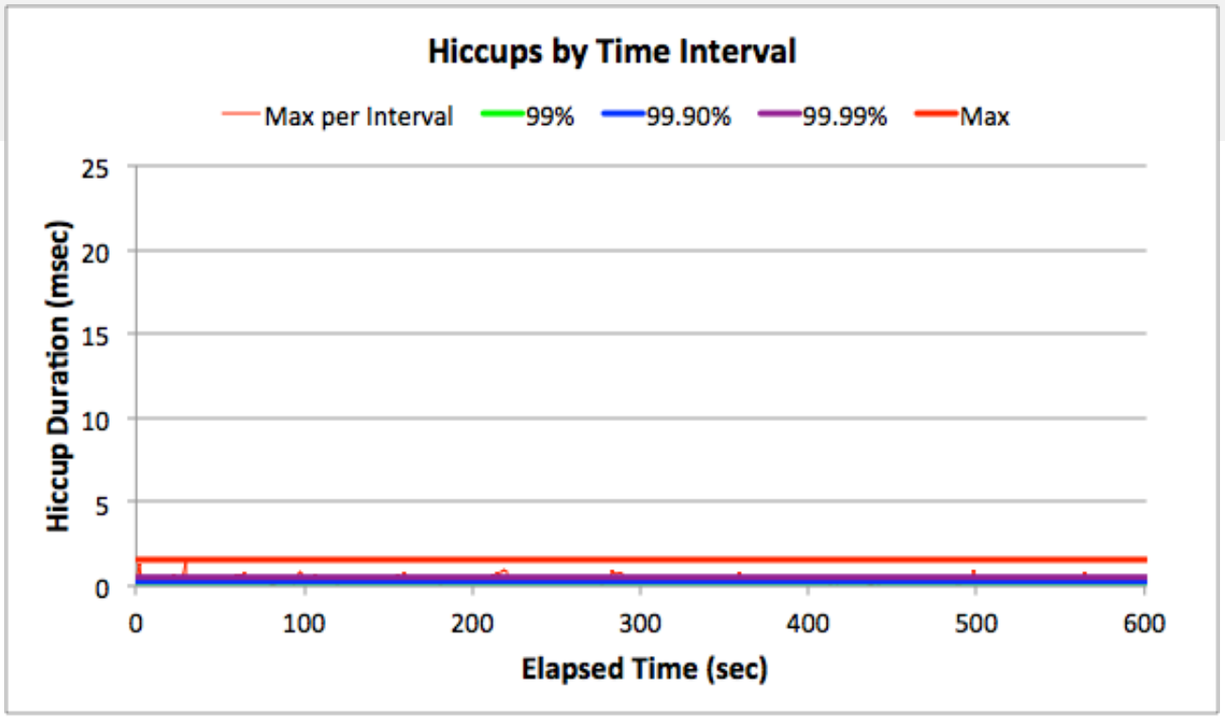
Low latency trading application

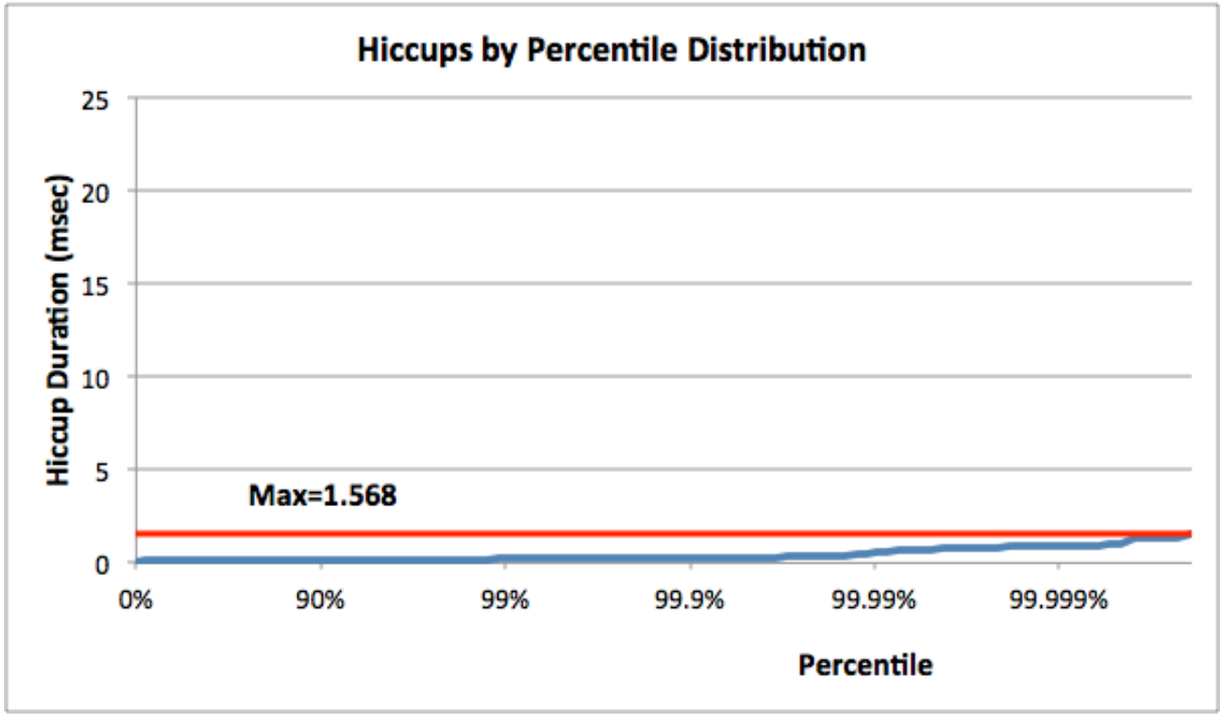
Oracle HotSpot (pure newgen)





Zing



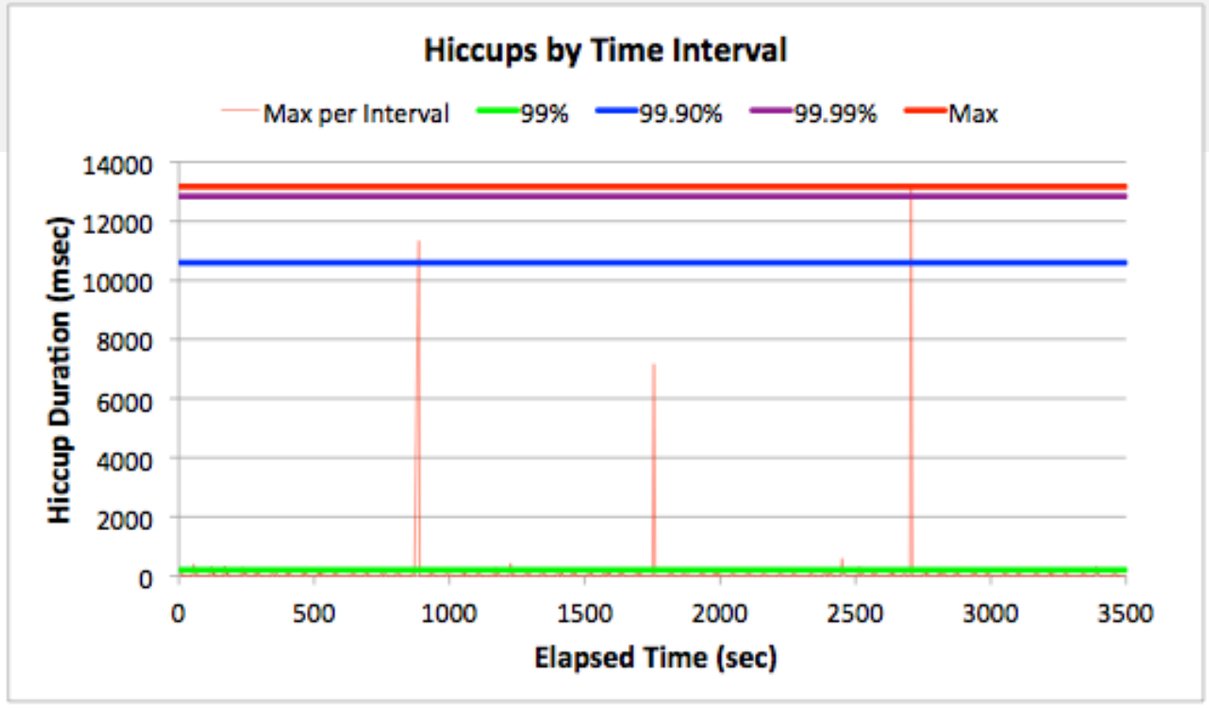


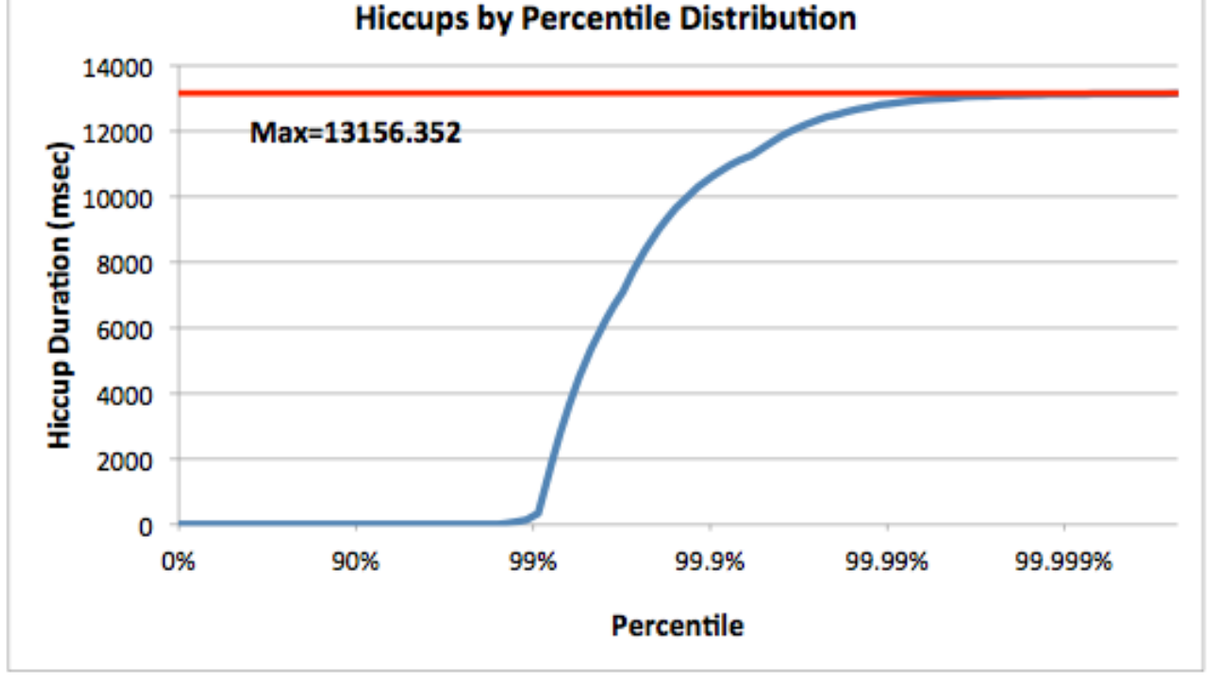
Low latency - Drawn to scale

It’s not just for Low Latency

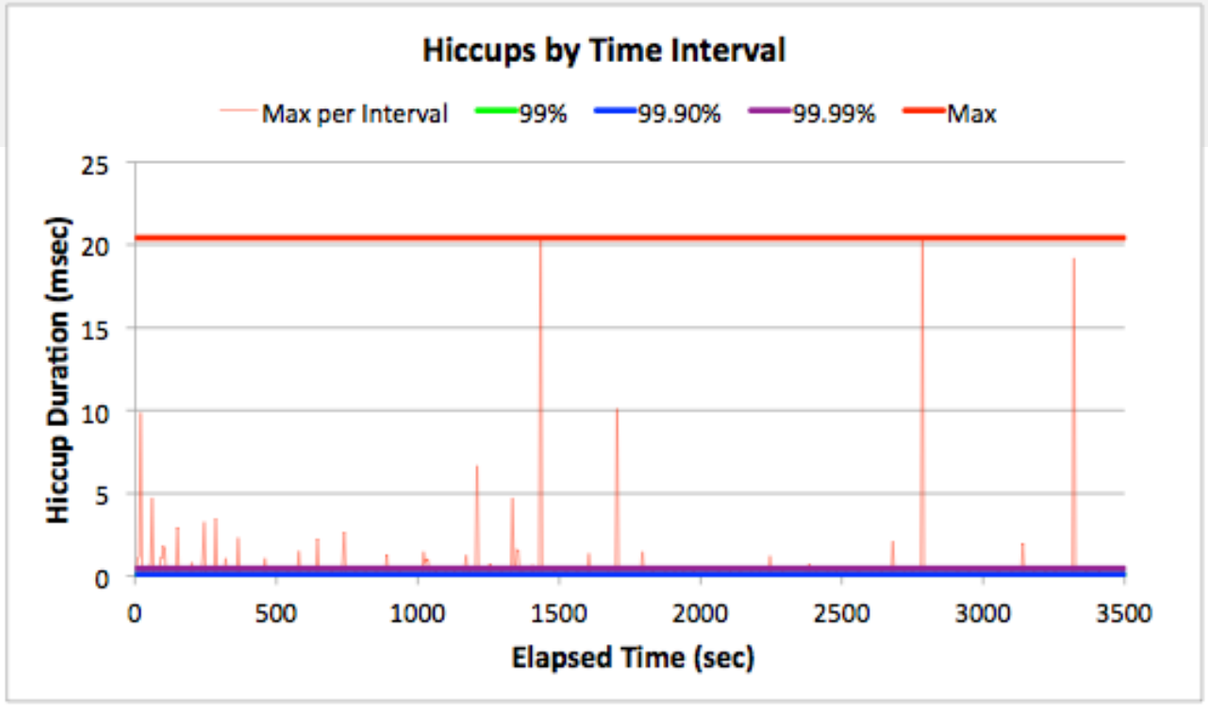
Just as easy to demonstrate for human-response-time apps

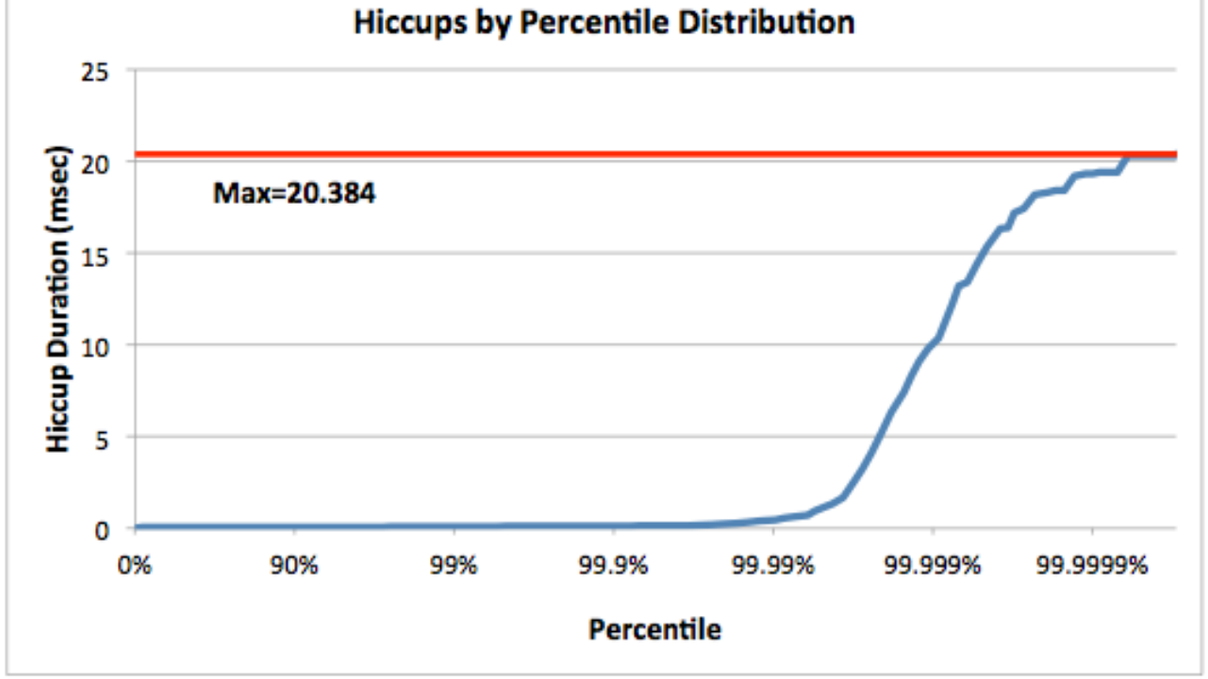
Oracle HotSpot CMS, 1GB in an 8GB heap





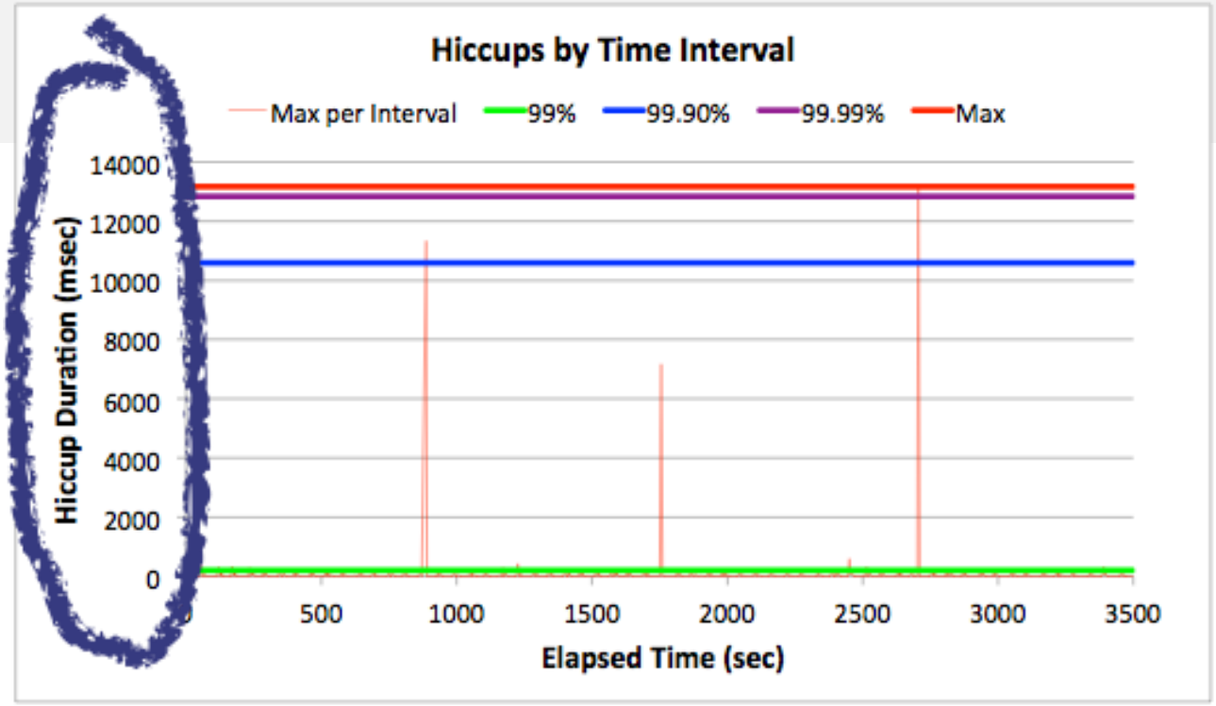
Zing, 1GB in an 8GB heap

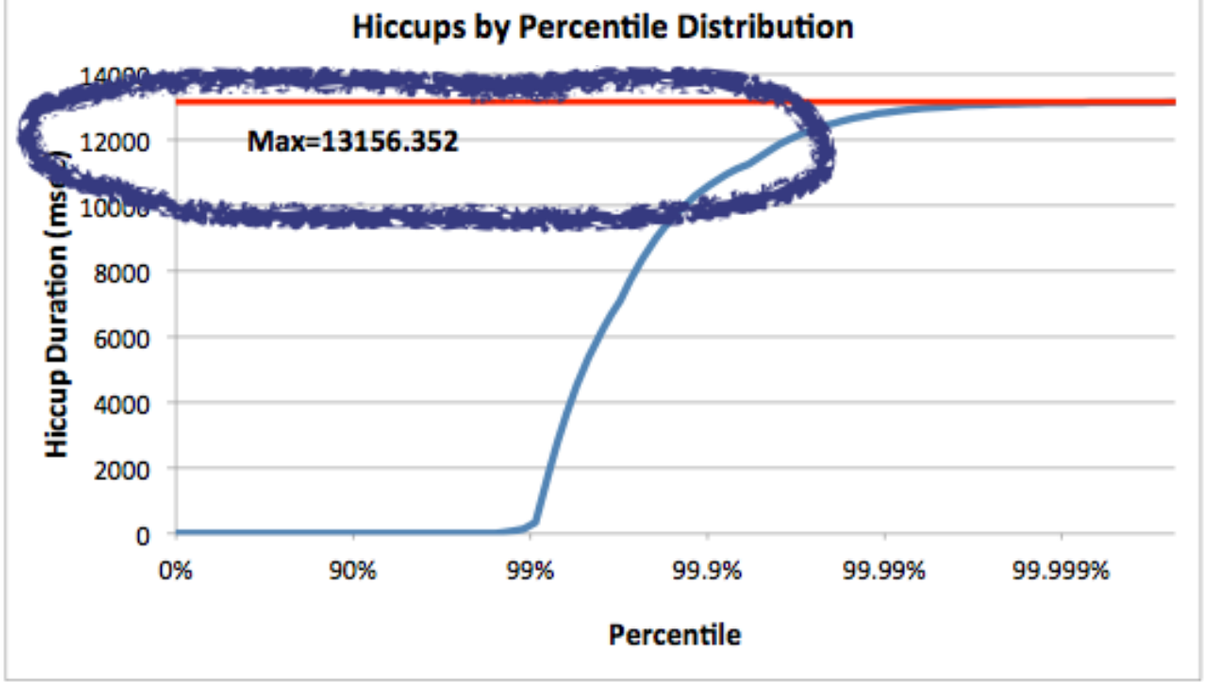




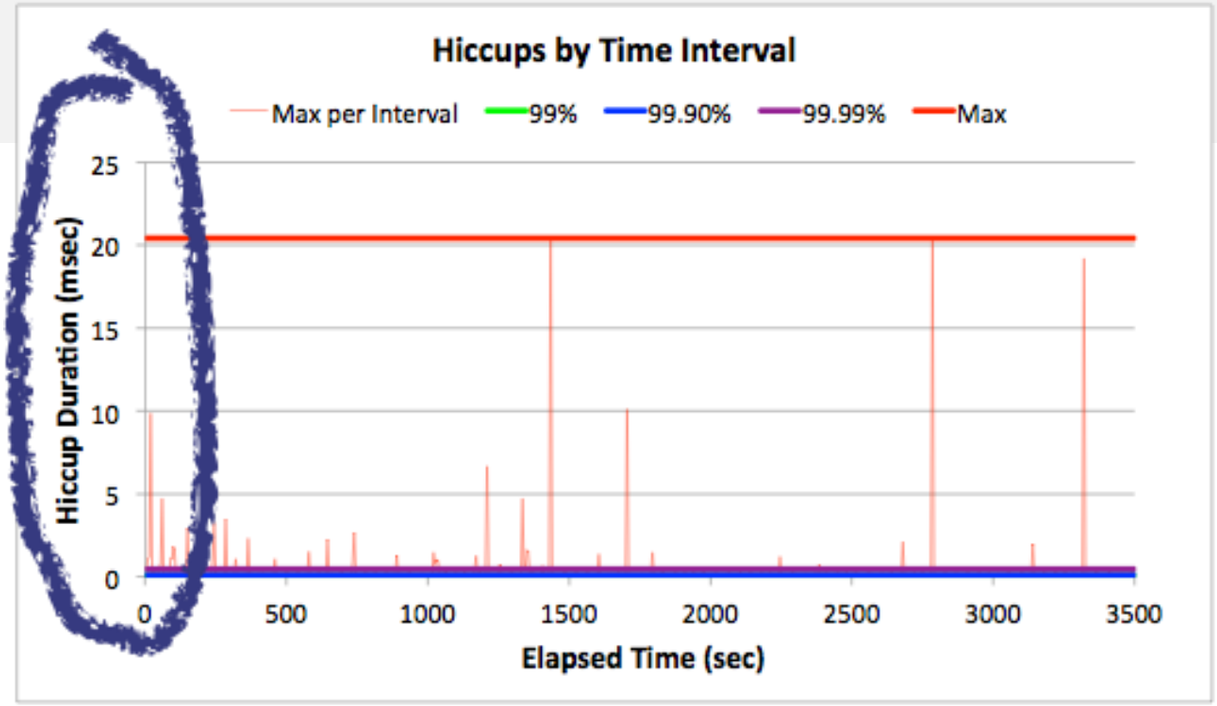
Portal Application, slow Ehcache “churn”

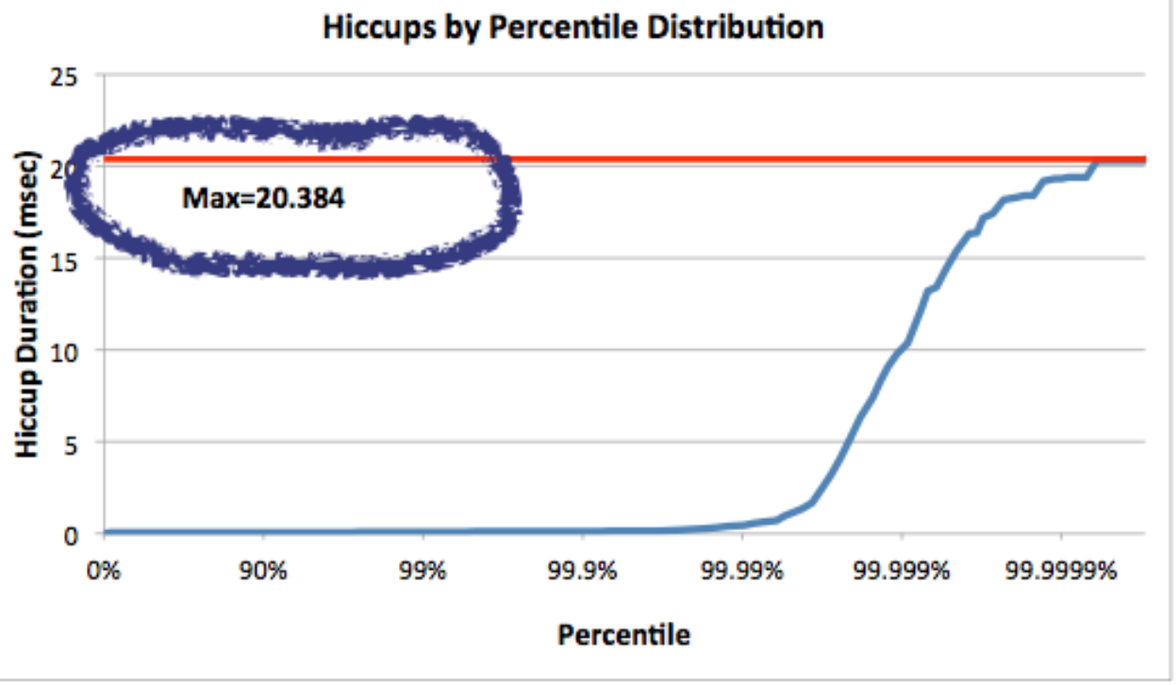
Oracle HotSpot CMS, 1GB in an 8GB heap





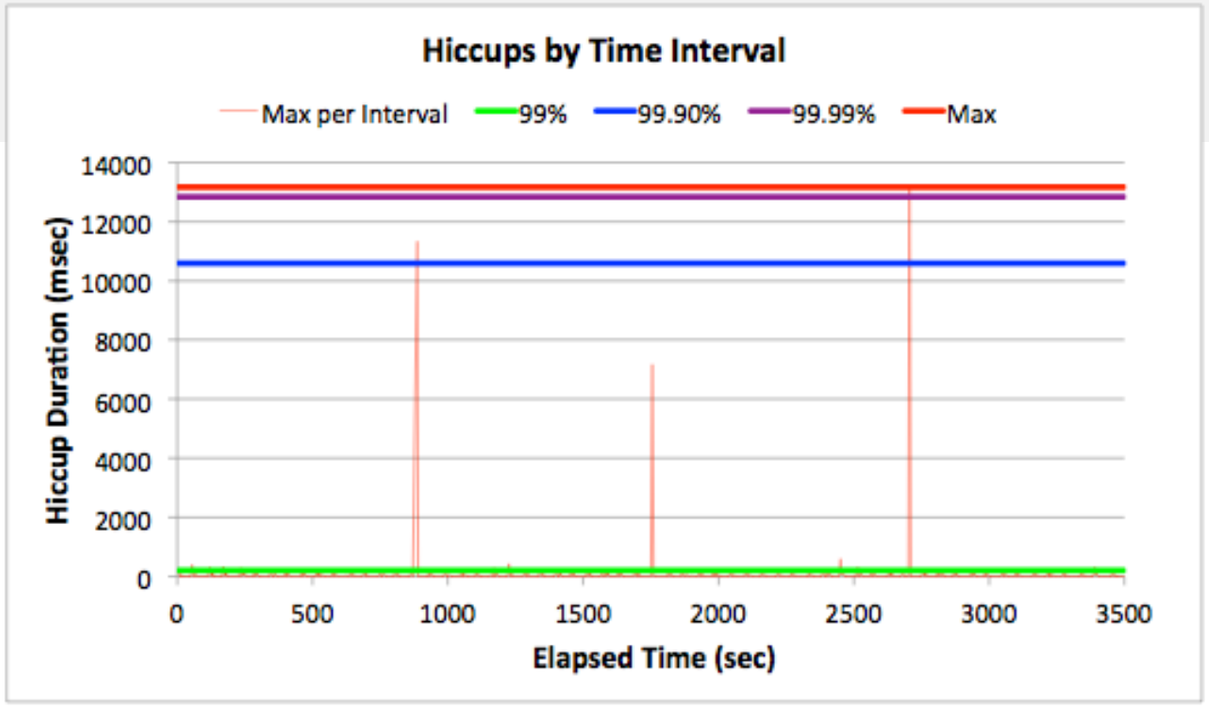
Zing, 1GB in an 8GB heap

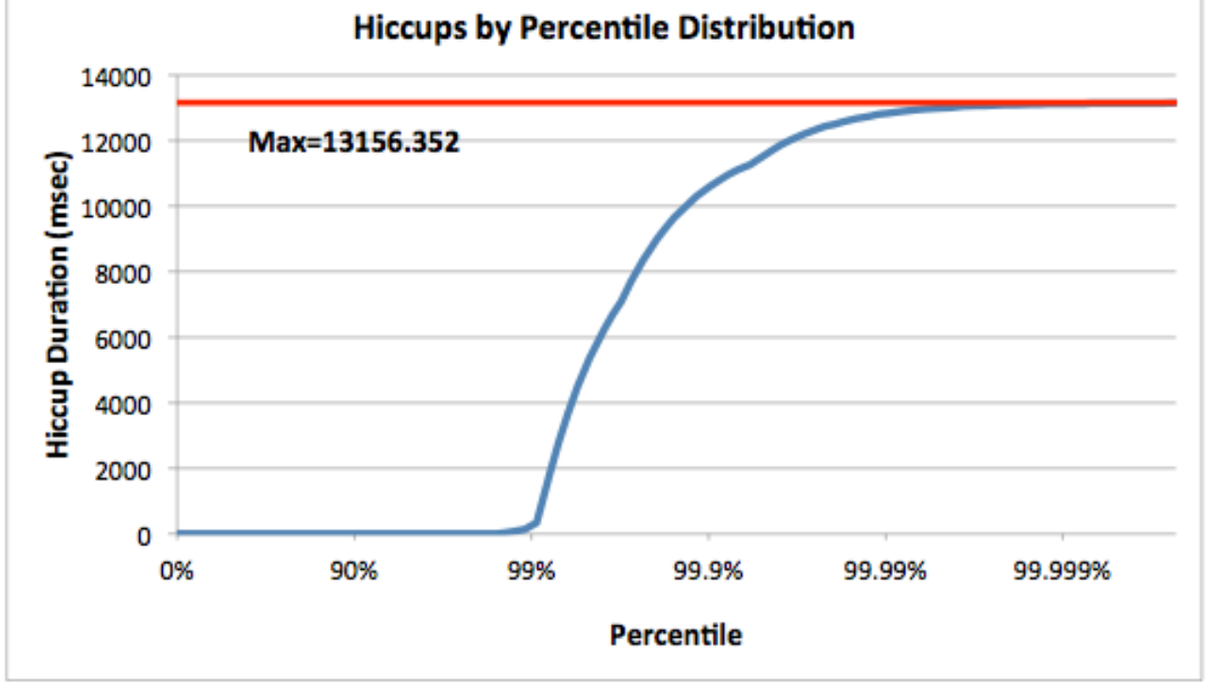




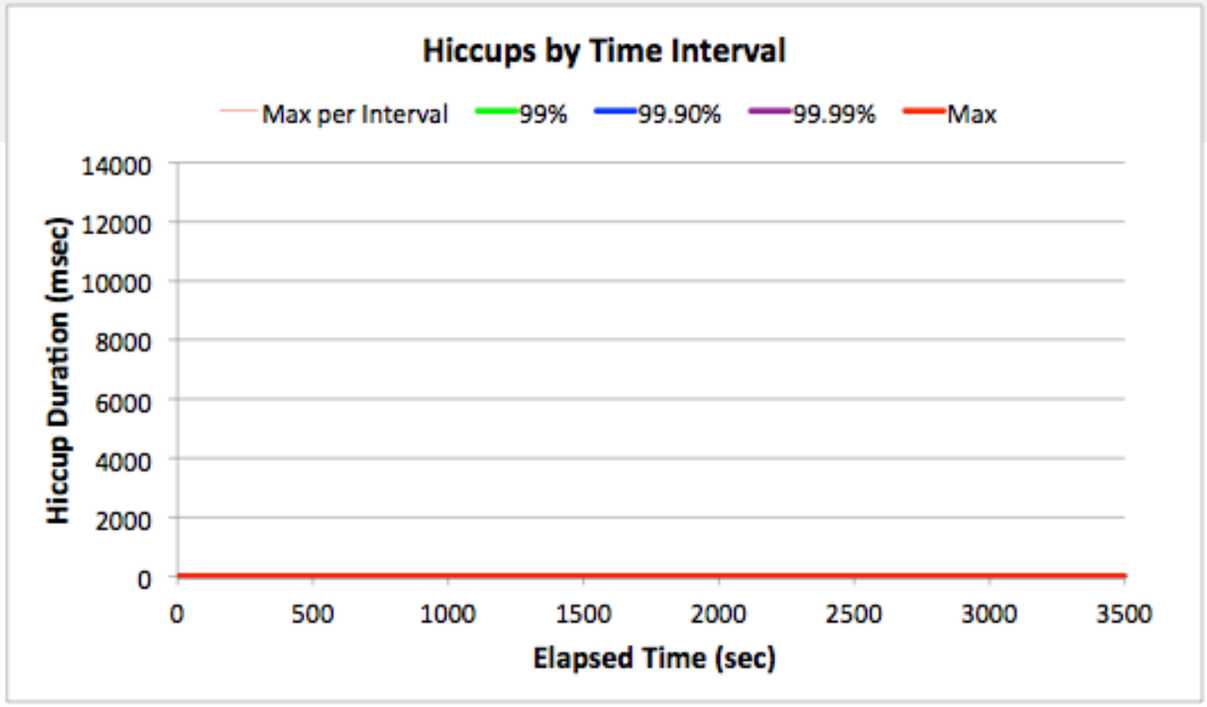
Portal Application, slow Ehcache “churn”

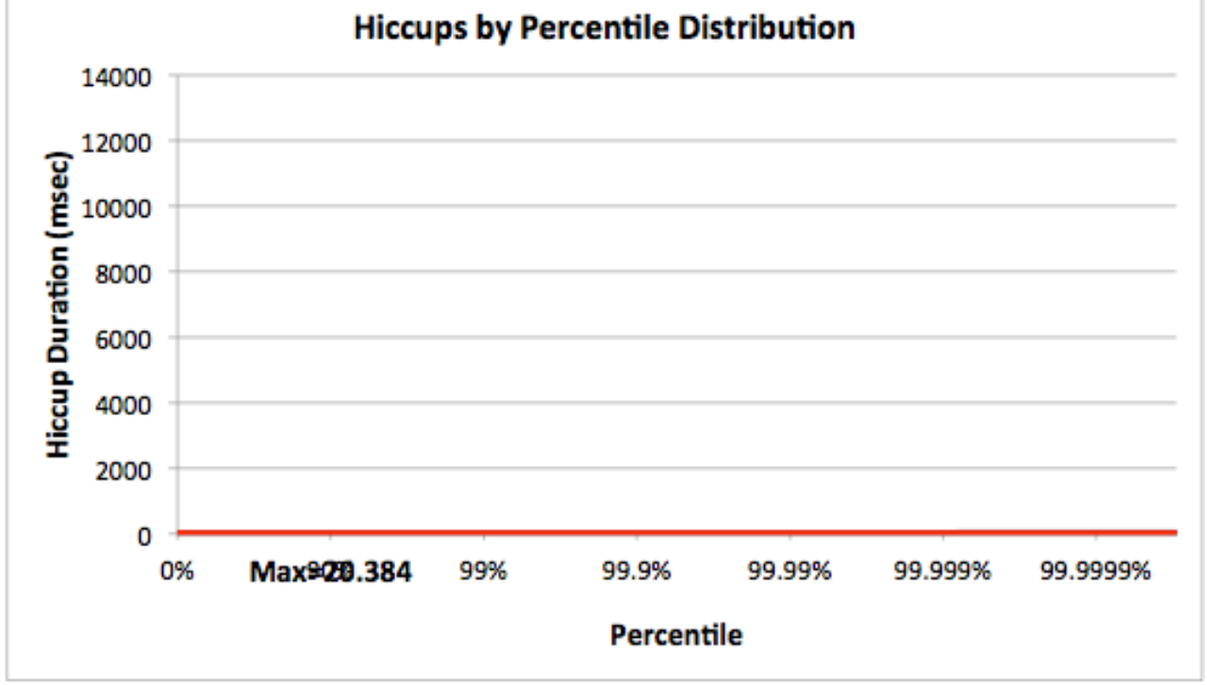
Oracle HotSpot CMS, 1GB in an 8GB heap





Zing, 1GB in an 8GB heap



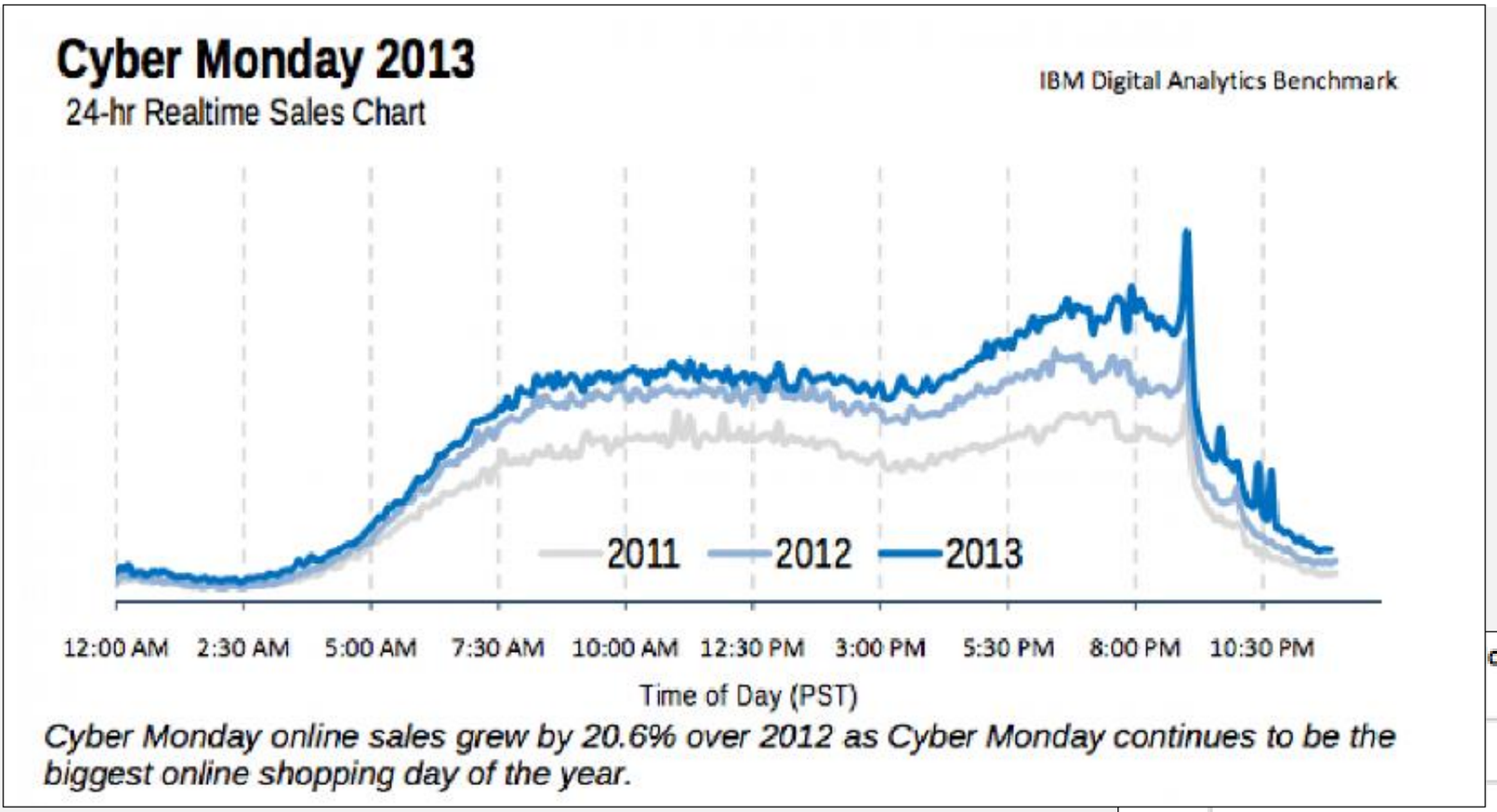


Portal Application - Drawn to scale

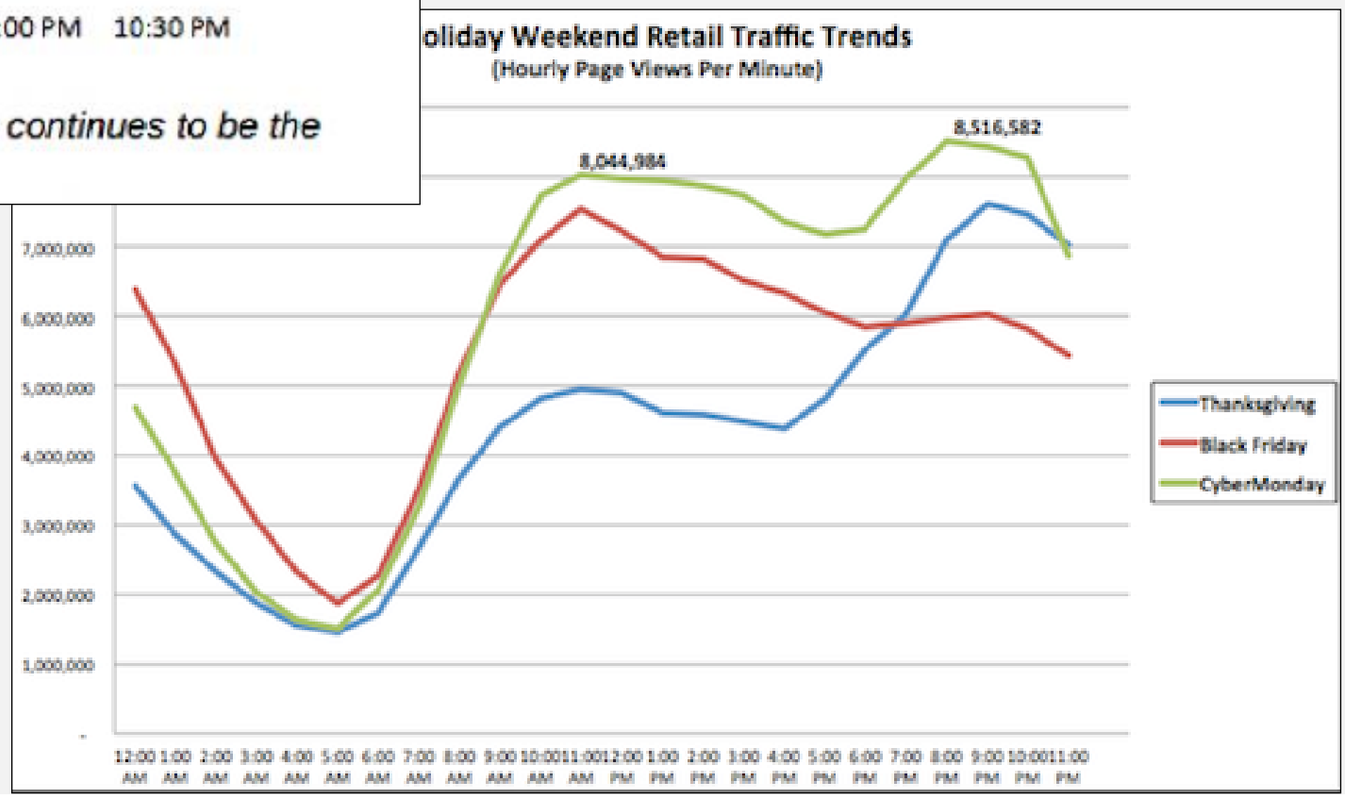
A Recent E-Commerce Case Study

Cyber Monday comes earlier every year…

General trends of real world e-commerce traffic



Cyber Monday online sales grew by 20.6% over 2012 as Cyber Monday continues to be the biggest online shopping day of the year.



Human-Time Real World Latency Case

Specific e-tail customer based in Salt Lake City, Utah.

- Web retail site faces spike loads every year over Thanksgiving through Cyber Monday.
- Site latency suffers at peak viewing and buying times, discouraging shoppers and leaving abandoned carts.
- Hard to predict height of surge, just know its big, far higher than regular traffic 362 other days of the year.
- New features like gallery search (Solr/Lucene) added higher memory footprint, longer GC times.
- Staff spent lots of effort tuning HotSpot.

Real World Latency Results

Timeframe was fall 2014.

- Customer studied Azul, met at Strata, SF
- Discussion led to Zing as viable alternative
- Customer ran pilot tests with positive results. Needed one Linux setting adjustment, otherwise same server gear.
- POC on customer live system (Amazon EC2 nodes) showed better than expected latency profiles.
- No more GC tuning!
- Experienced a stable and profitable Thanksgiving 2014 weekend.

Remind me how GC tuning sucks

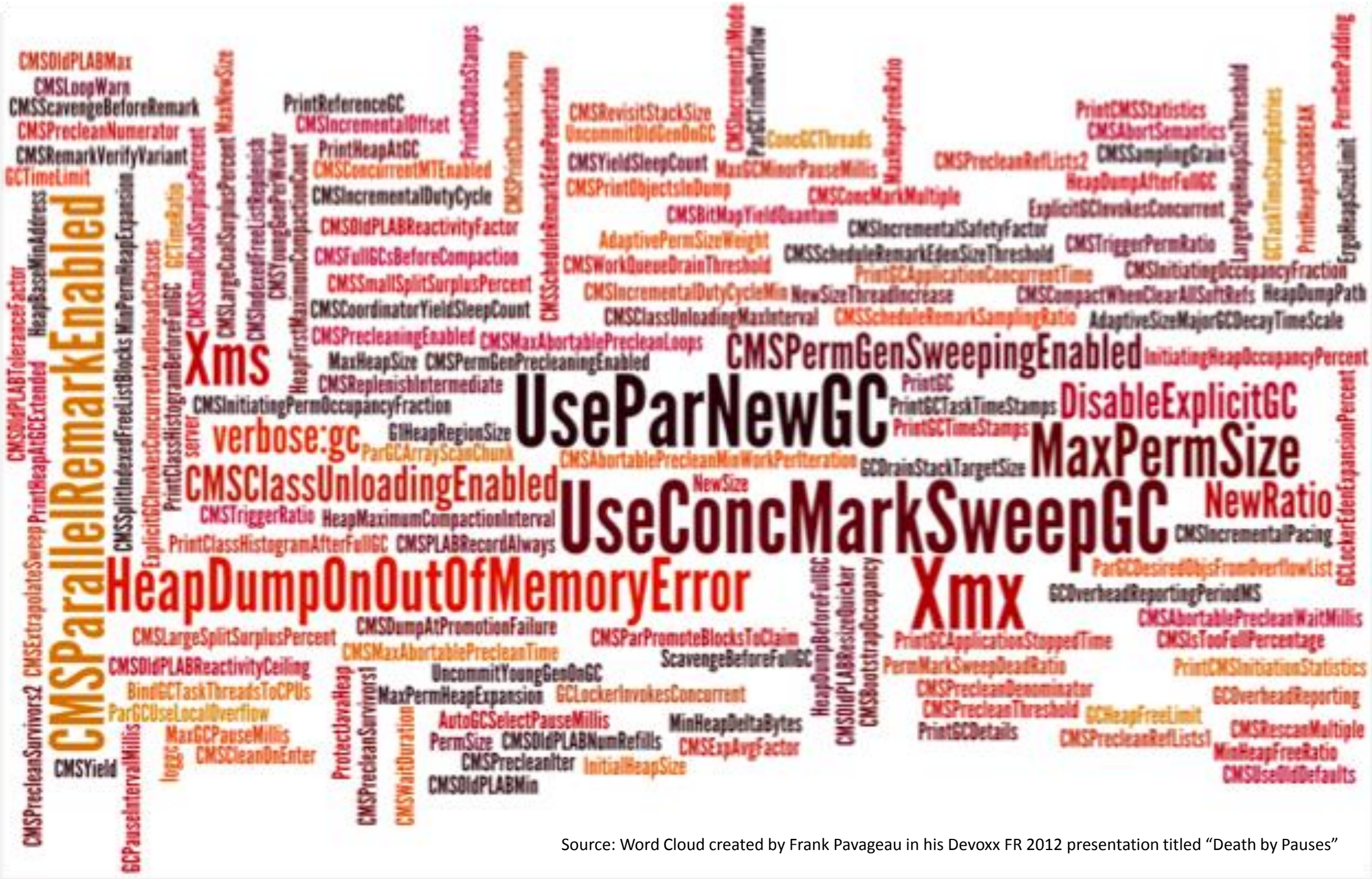
Java GC tuning is “hard”…

- Examples of actual command line GC tuning terms:

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”

Complete guide to Zing GC tuning

java -Xmx40g

Any other problems beyond GC?

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)

Time To Safepoint: 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 analyze” 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. Consider this a variation on Amdahl’s law.
- Many code paths in the JVM are long...

Time To Safepoint (TTSP), the most common examples

- Array copies and object clone()
- Counted loops
- Many other variants in the runtime...
- Measure, Measure, Measure...

- Zing has a built-in TTSP profiler
- At Azul, the CTO walks around with a 0.5msec beat down stick...

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).
- ...

Takeaway: In 2015, “Real” Java is finally viable for low latency applications

- GC is no longer a dominant issue, even for outliers
- 2-3 msec worst case with “easy” tuning
- < 1 msec worst 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

One-liner Takeaway:

Zing: the cure for your Java hiccups

Compulsory Marketing Pitch

Azul Hot Topics

Zing 15.05 imminent

- 1TB heap
- ReadyNow!
- JMX
- Oracle Linux

Zing for Cloud

- Amazon AMIs
- Rackspace OnMetal compat
- Docker in R&D

Zing for Big Data

- Cloudera CDH5 cert
- Cassandra paper
- Spark is in Zing open source program

Zulu

- Azure Gallery
- JSE Embedded
- 8u45 available
- 7u80 is a new era

Q&A and In Closing…

- Go get some **Zing** today!
- At very least download **JHiccup**.
- Grab a Zing **Free Trial** card.
- Gooey Butter Cake **vs.** The Concrete

@schuetzematt **azul.com**