

Enabling Java in Latency Sensitive Environments

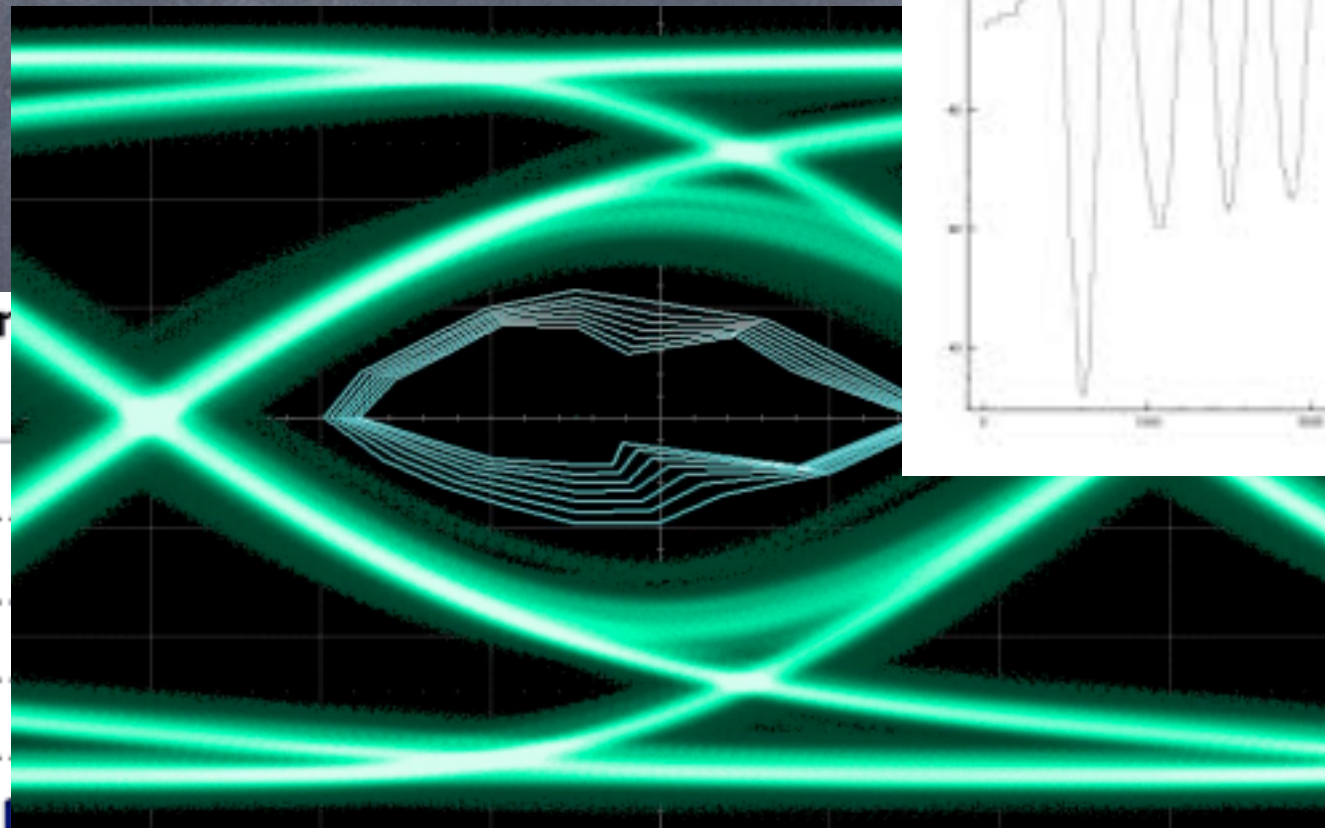
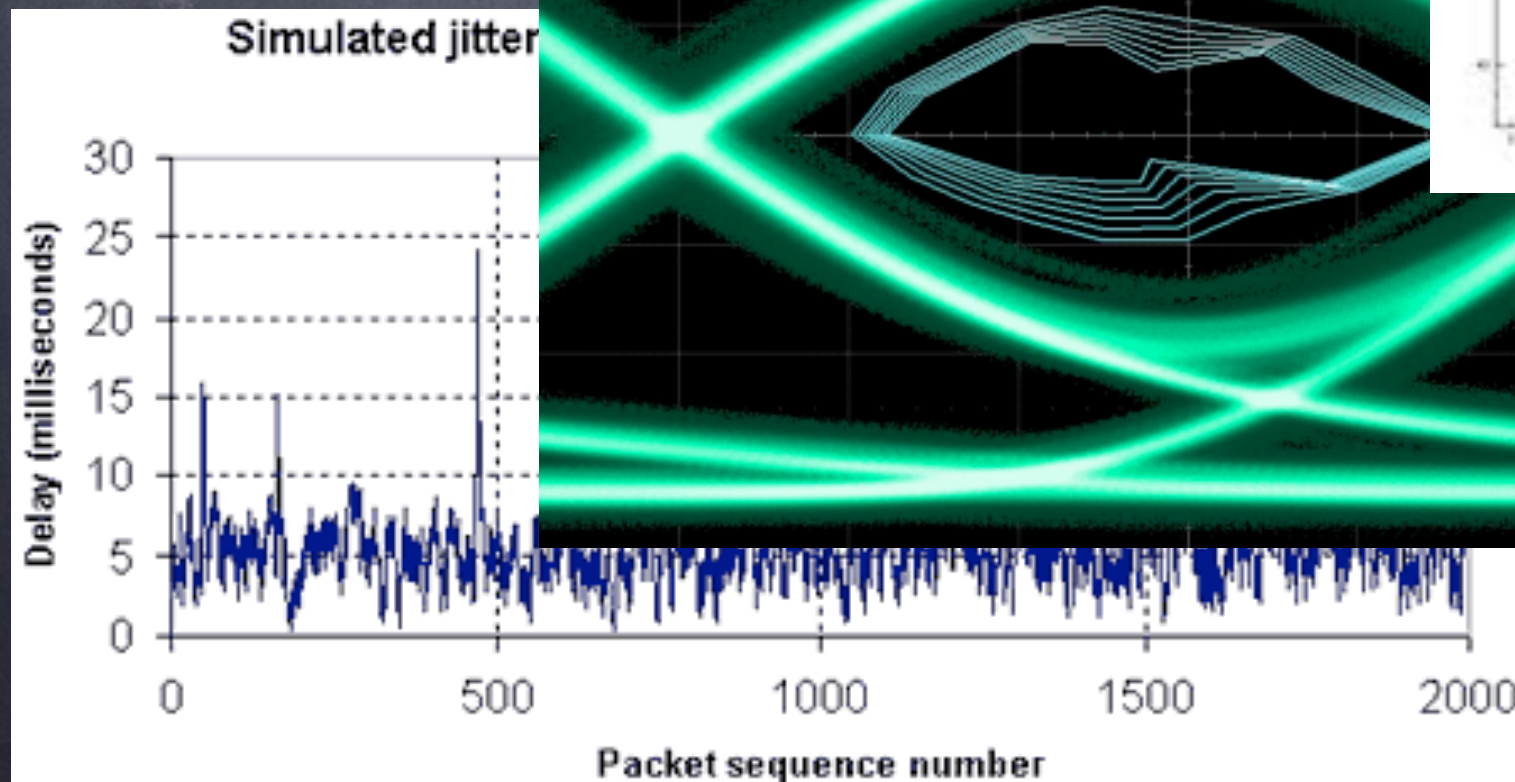
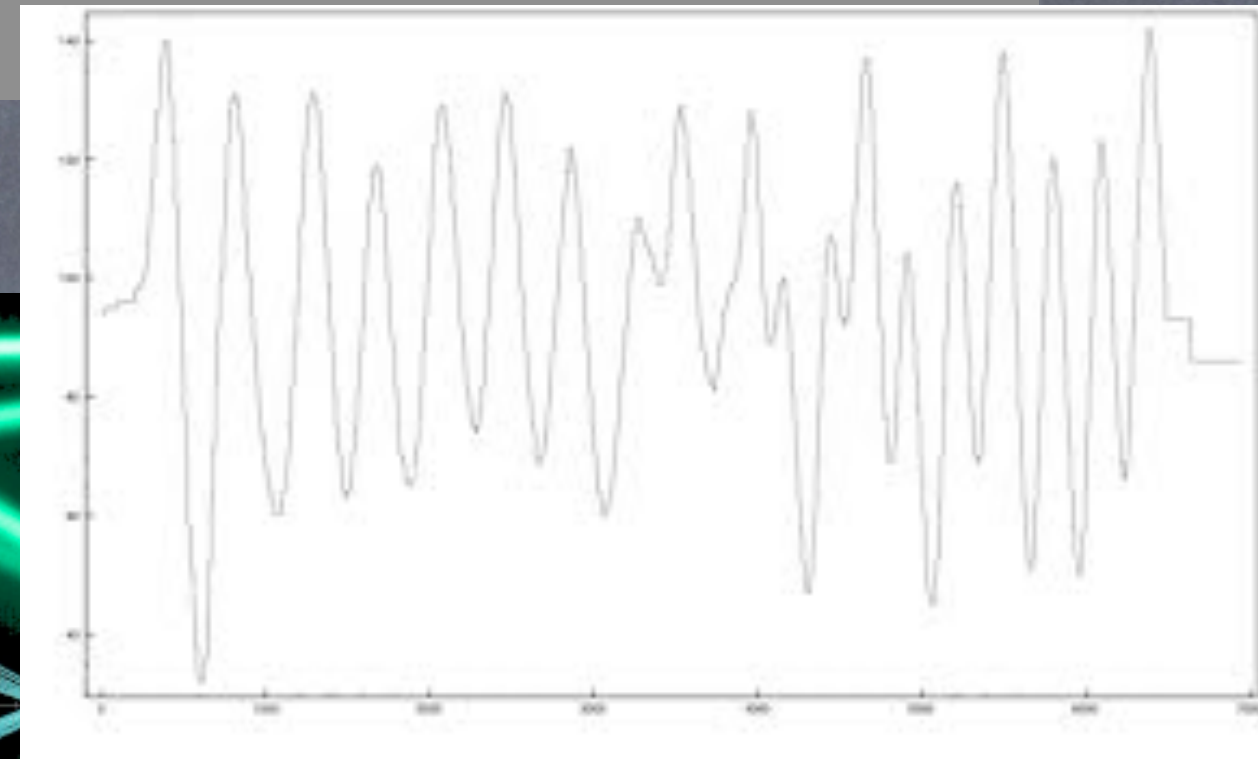
Gil Tene, CTO & co-Founder, Azul Systems



High level agenda

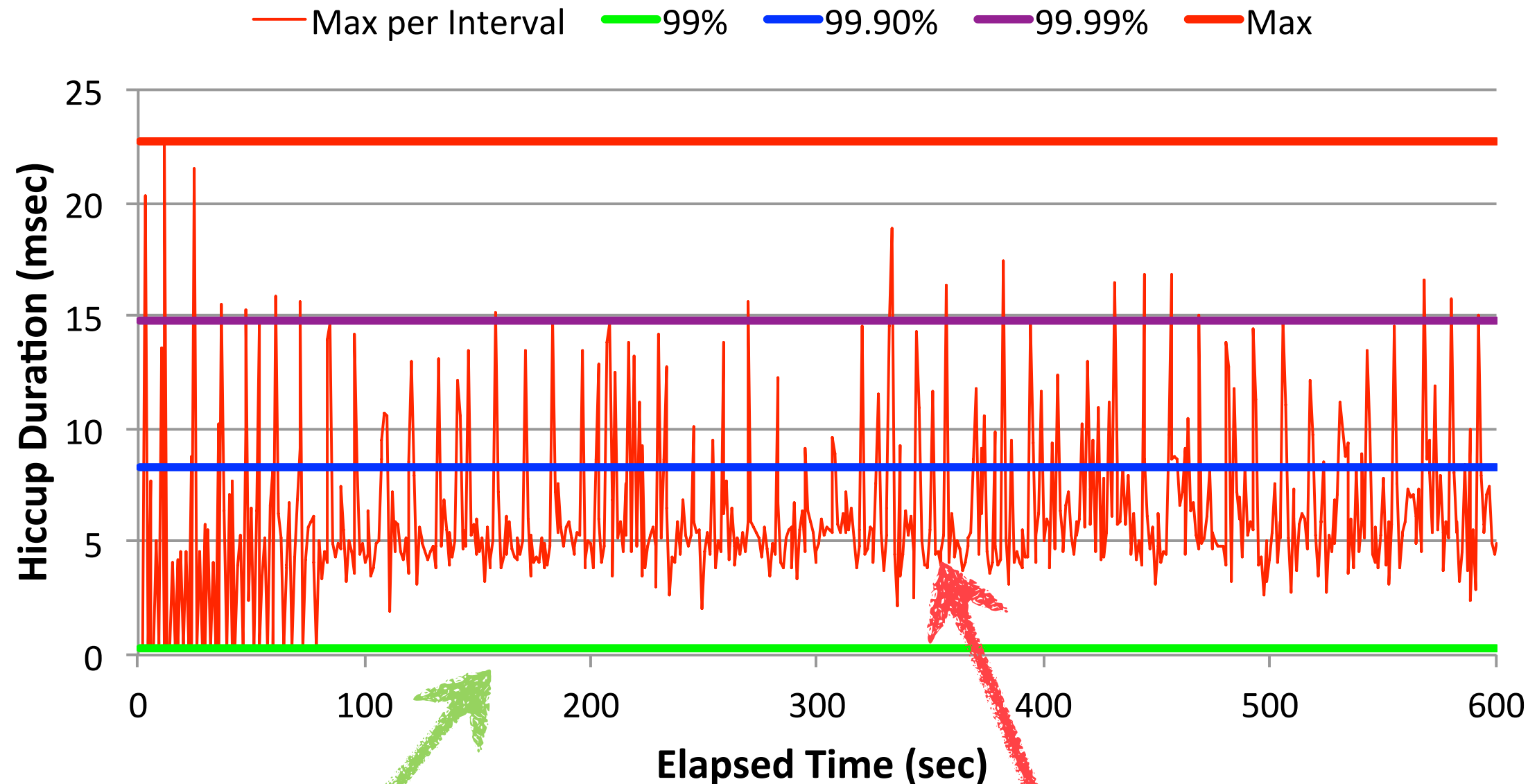
- Intro, jitter vs. JITTER
- Java in a low latency application world
- The (historical) fundamental problems
- What people have done to try to get around them
- What if the fundamental problems were eliminated?
- What 2013 looks like for Low latency Java developers

This is Jitter



Is "jitter" a proper word for this?

Hiccups by Time Interval



99%ile is ~60 usec

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

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)
- Known for Low Latency, Consistent execution, and Large data set excellence

Vega



Java in the low latency world

Java in a 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

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?

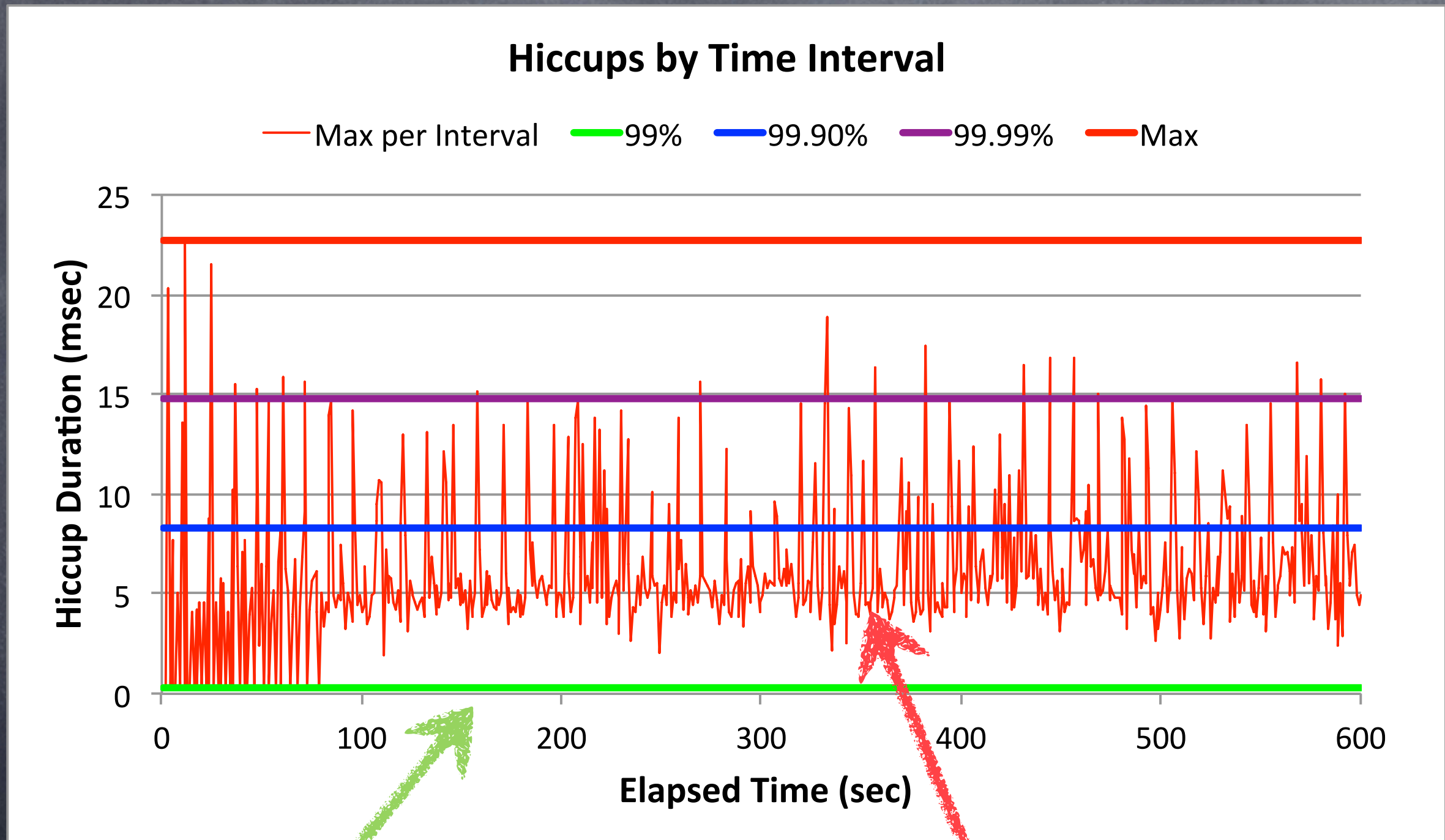
- No
- A good programmer will get roughly the same speed from both Java and C++
- A bad programmer won't get you fast code on either
- The 50%`ile and 90%`ile are typically excellent...
- It's those pesky occasional stutters and stammers and stalls that are the problem...
- Ever hear of Garbage Collection?

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



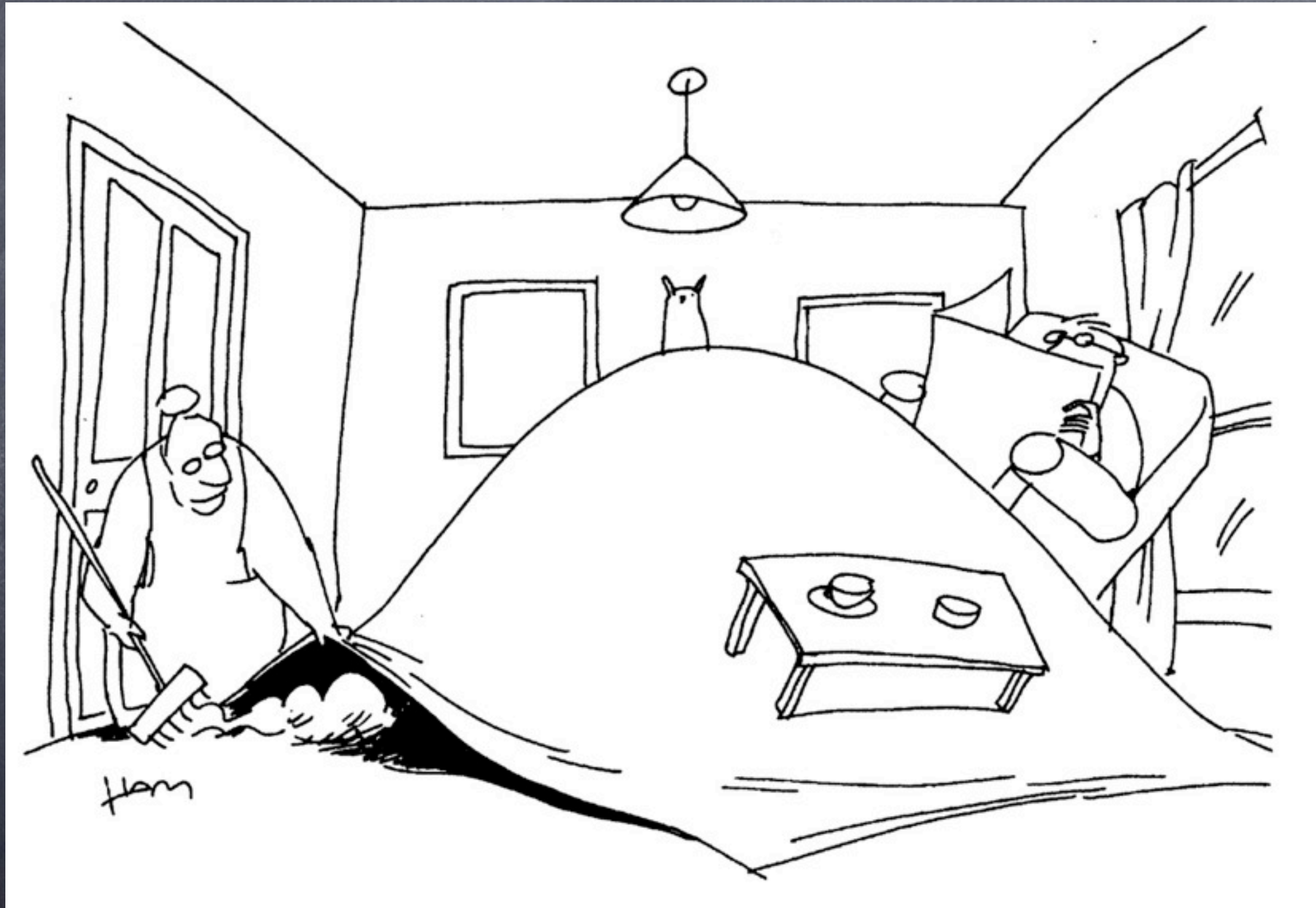
99%ile is ~60 usec

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

One way to deal with Stop-The-World GC



A way to deal with Stop-The-World 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.

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
- ...
- 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



Unsustainable
Throughput

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

- 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

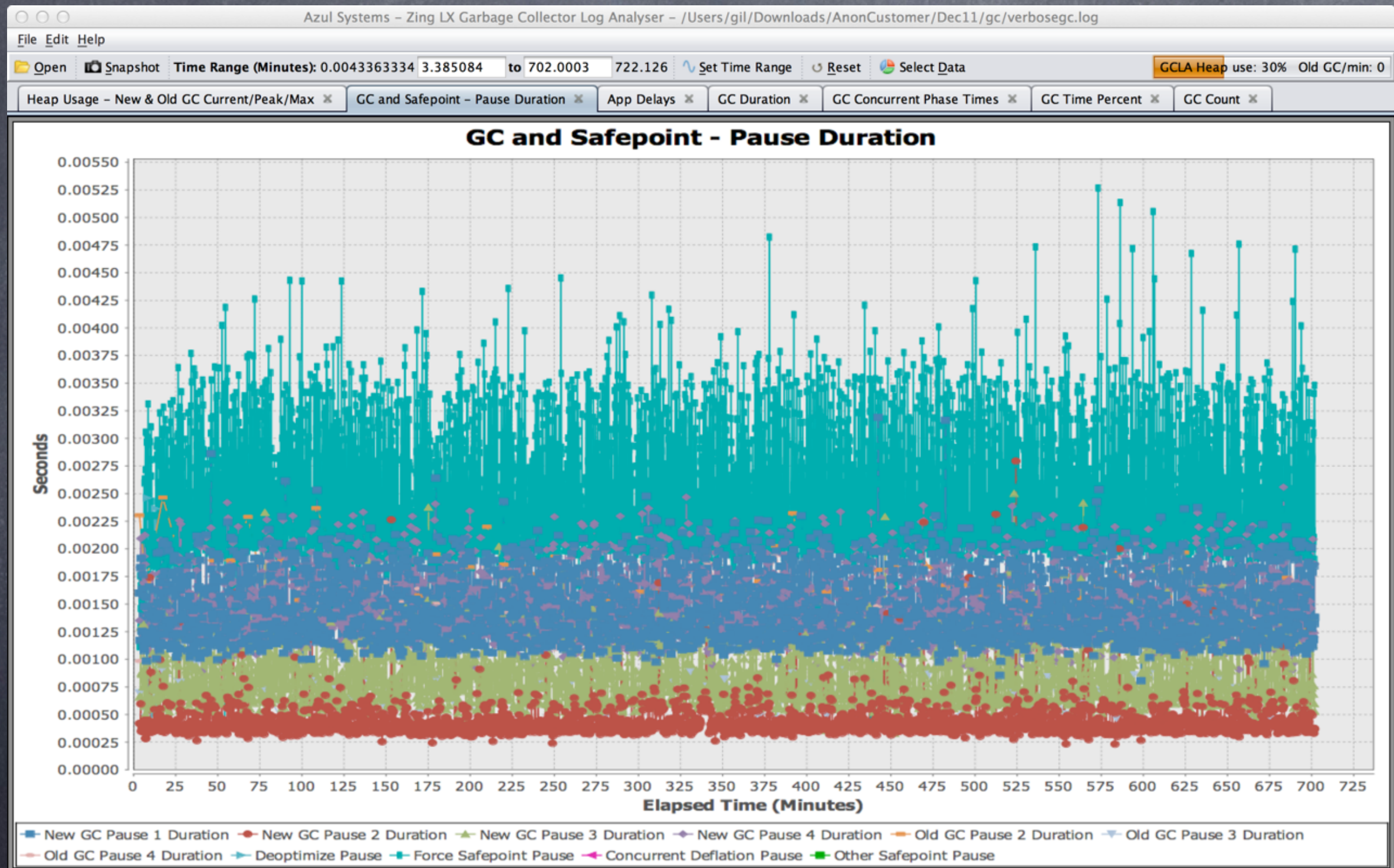
The Zing "C4" Collector

Continuously Concurrent Compacting Collector

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

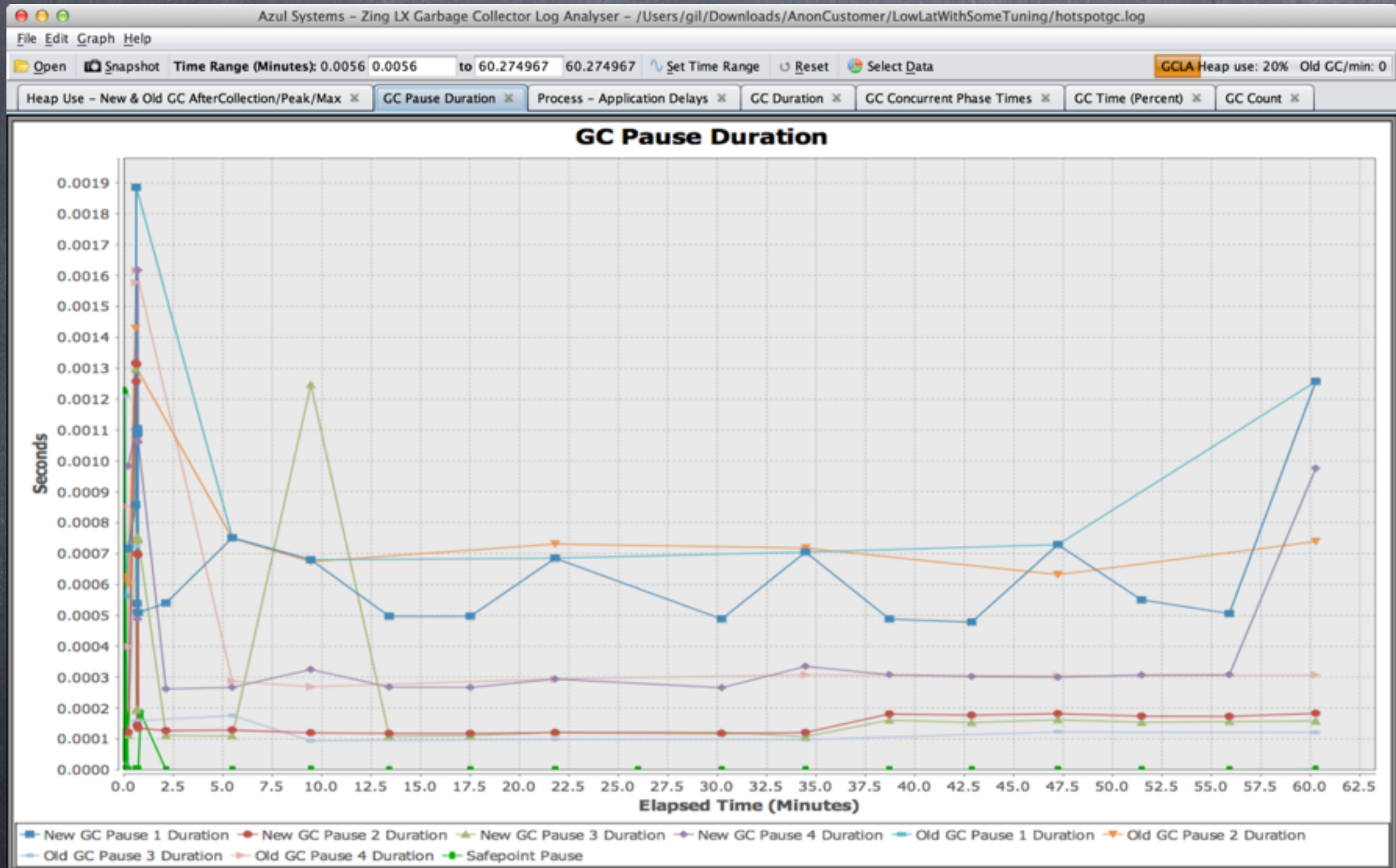
Benefits

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



An example of behavior after 4 days of system tuning

Low latency application



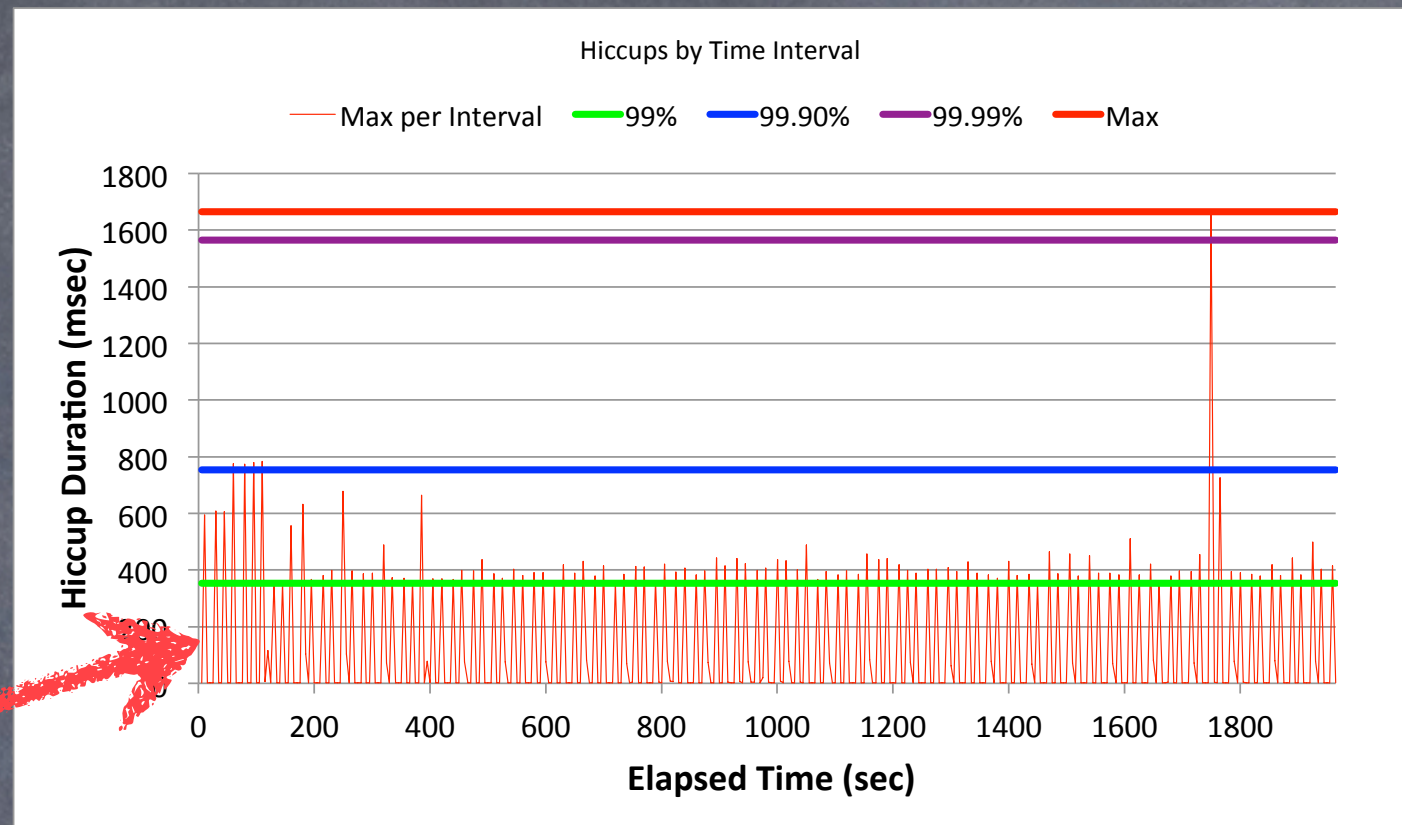
This is not “just Theory”

jHiccup:

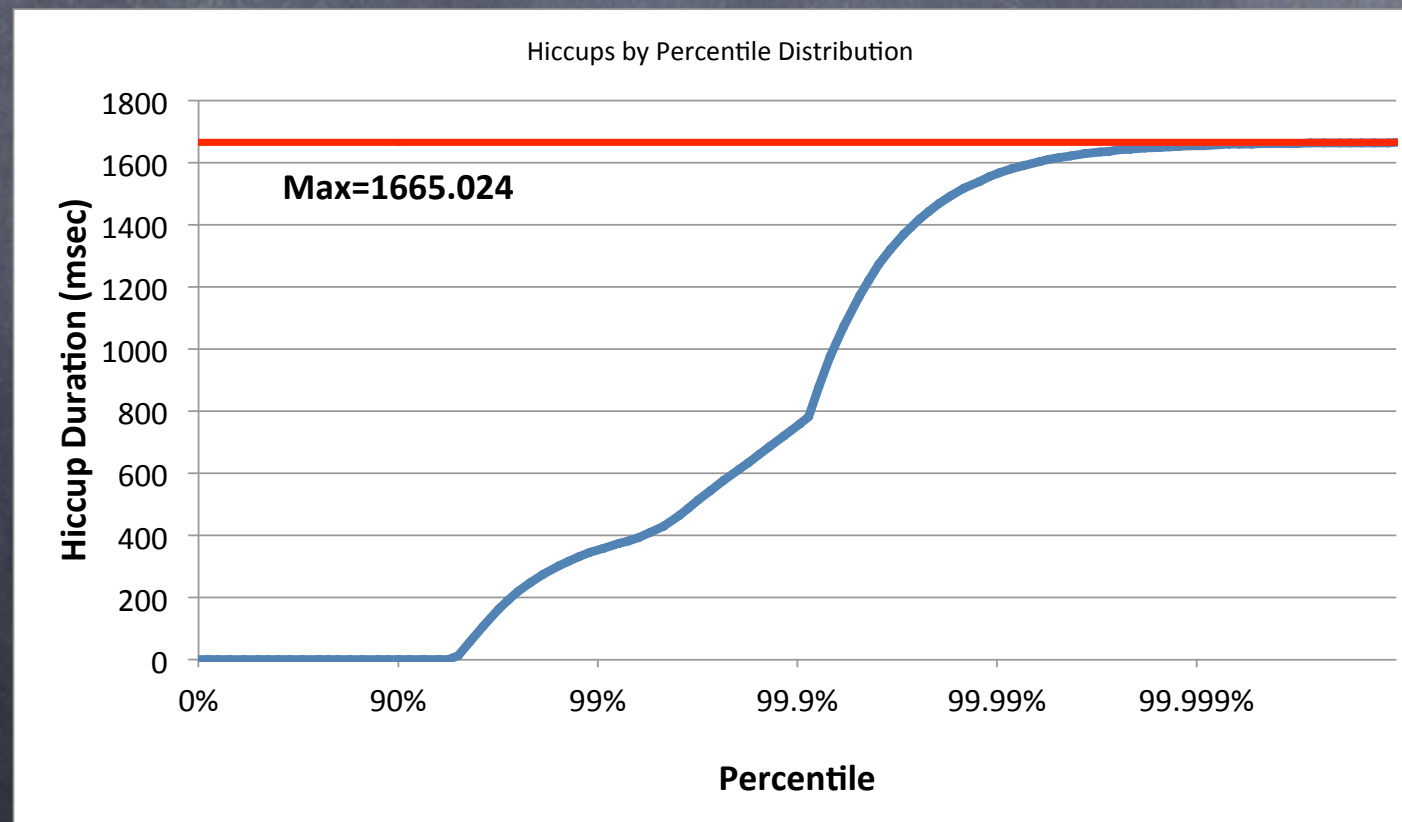
A tool that measures and reports
(as your application is running)
if your JVM is running all the time

Discontinuities in Java platform execution – Easy To Measure

We call these
“hiccups”



A telco
App with
a bit of a
“problem”



Fun with jHiccup



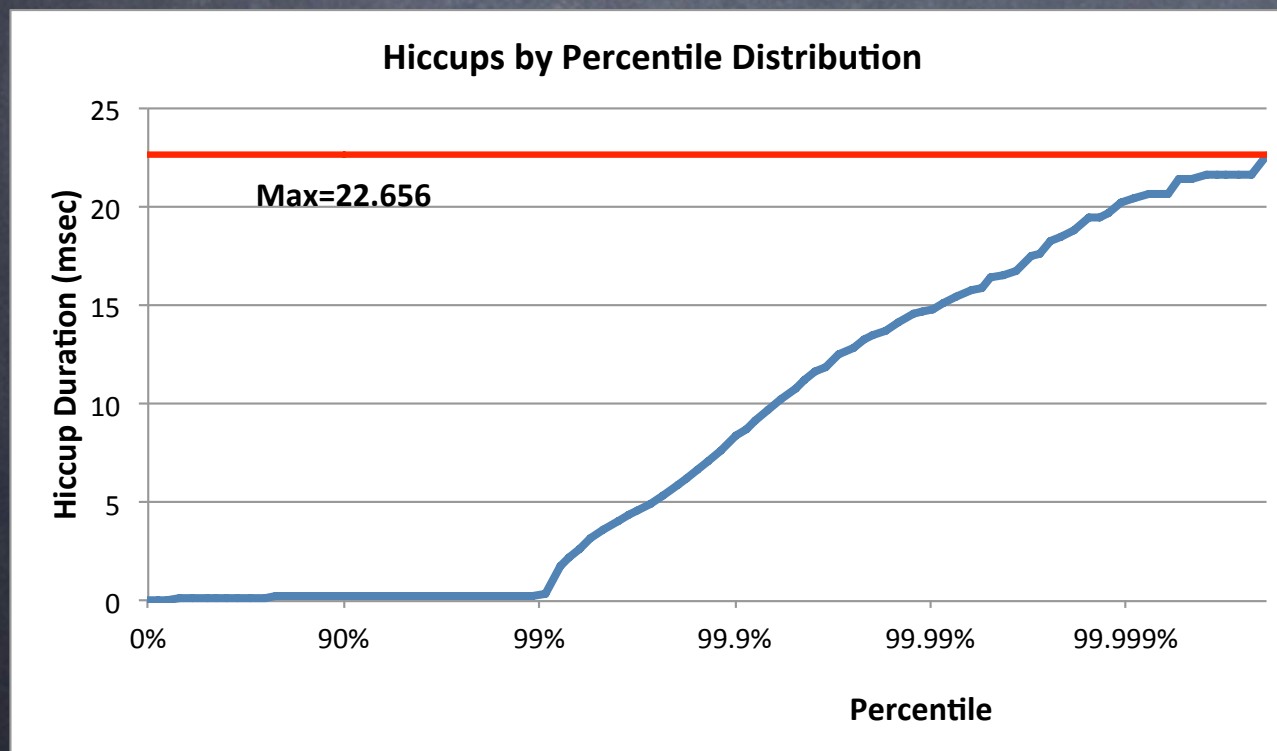
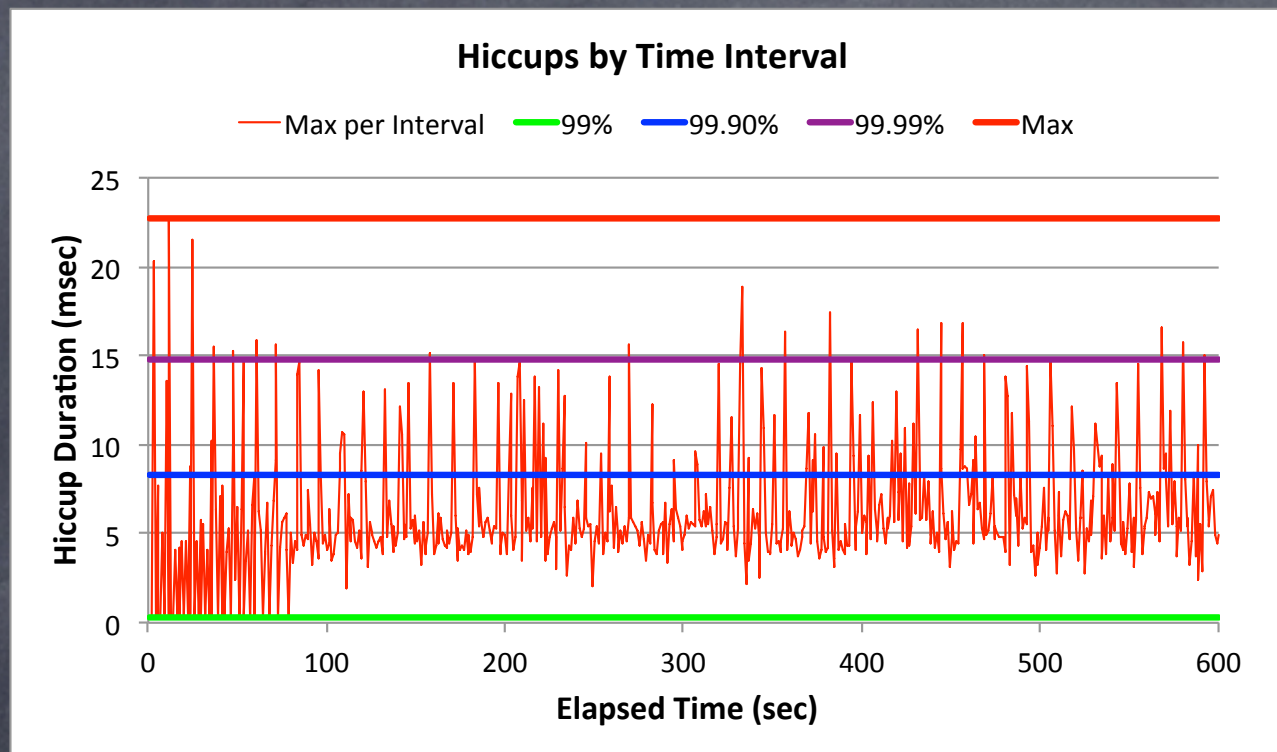
Charles Nutter @headius

20 Jan

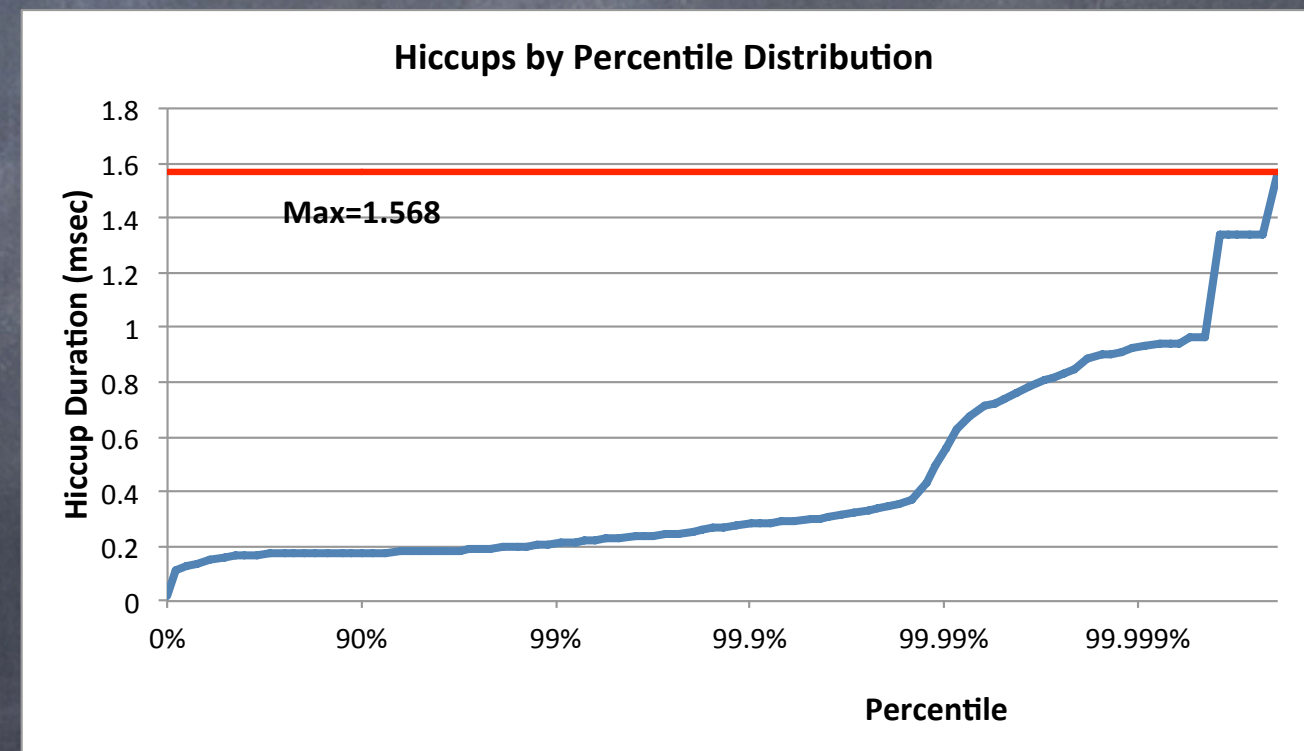
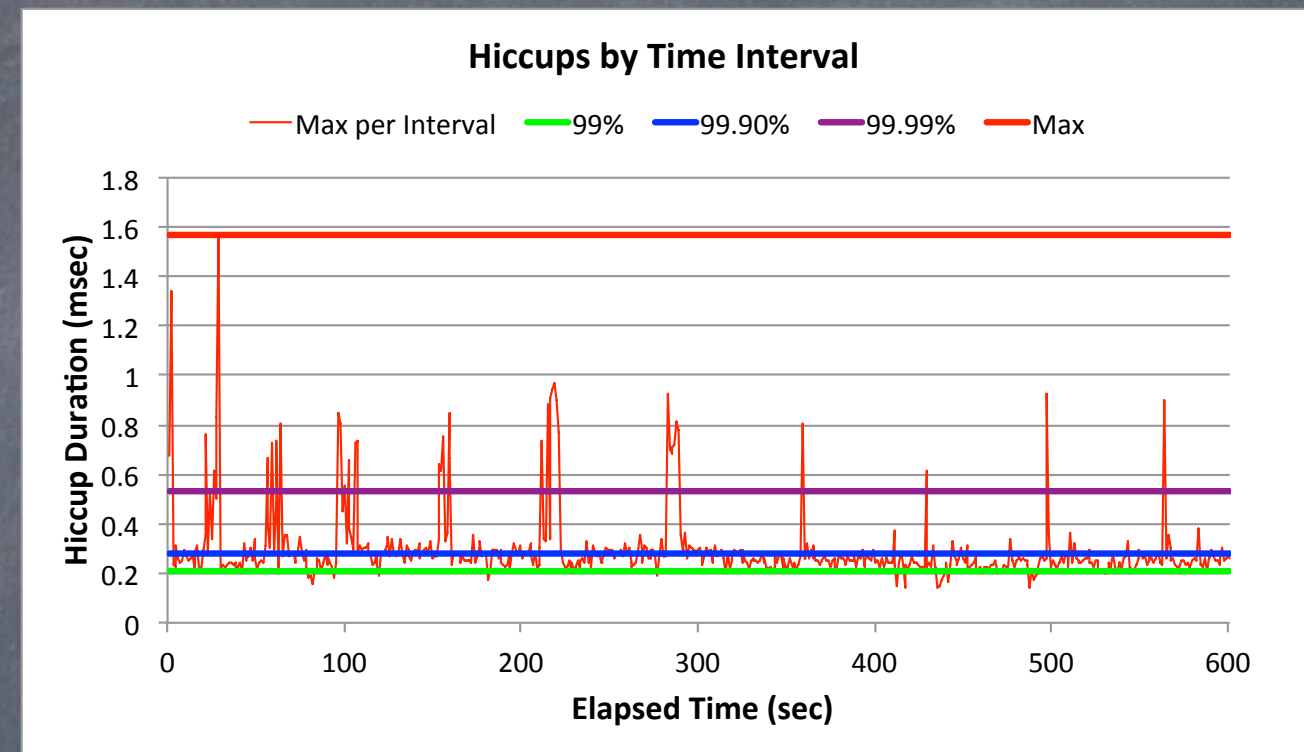
jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)

↕ Retweeted by Gil Tene

Oracle HotSpot (pure newgen)

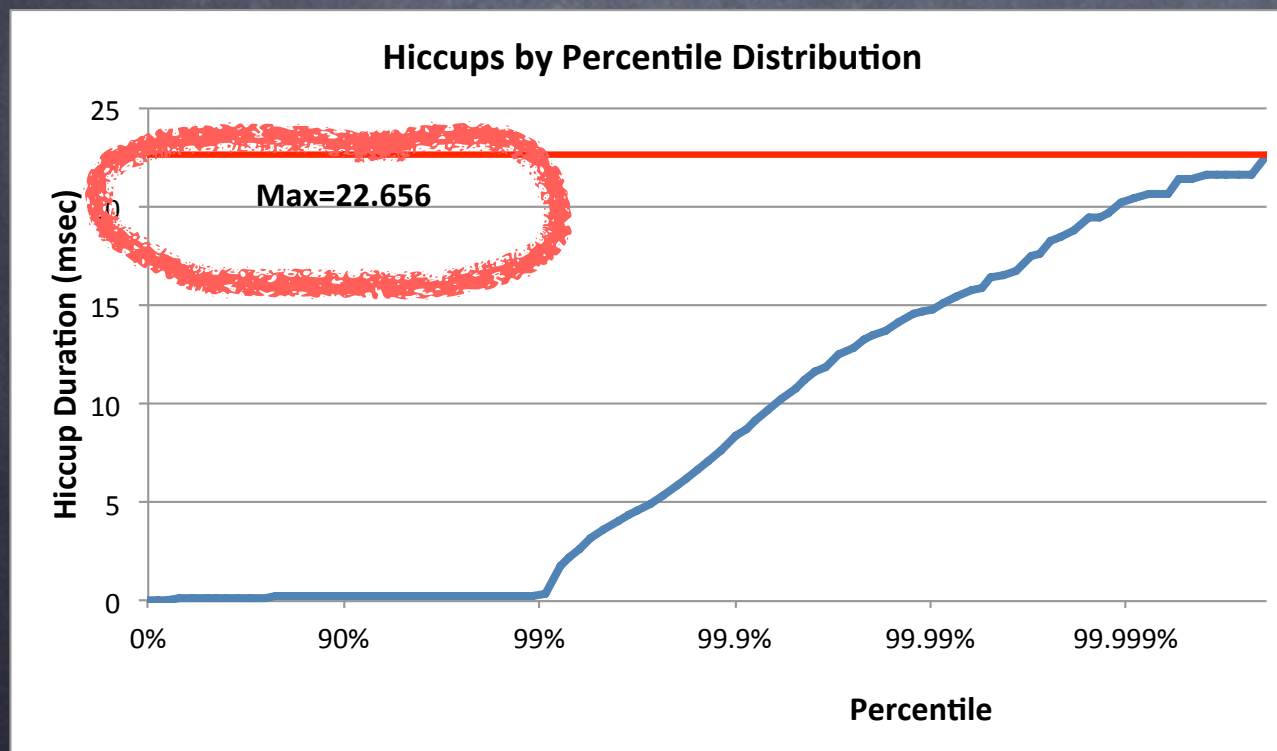
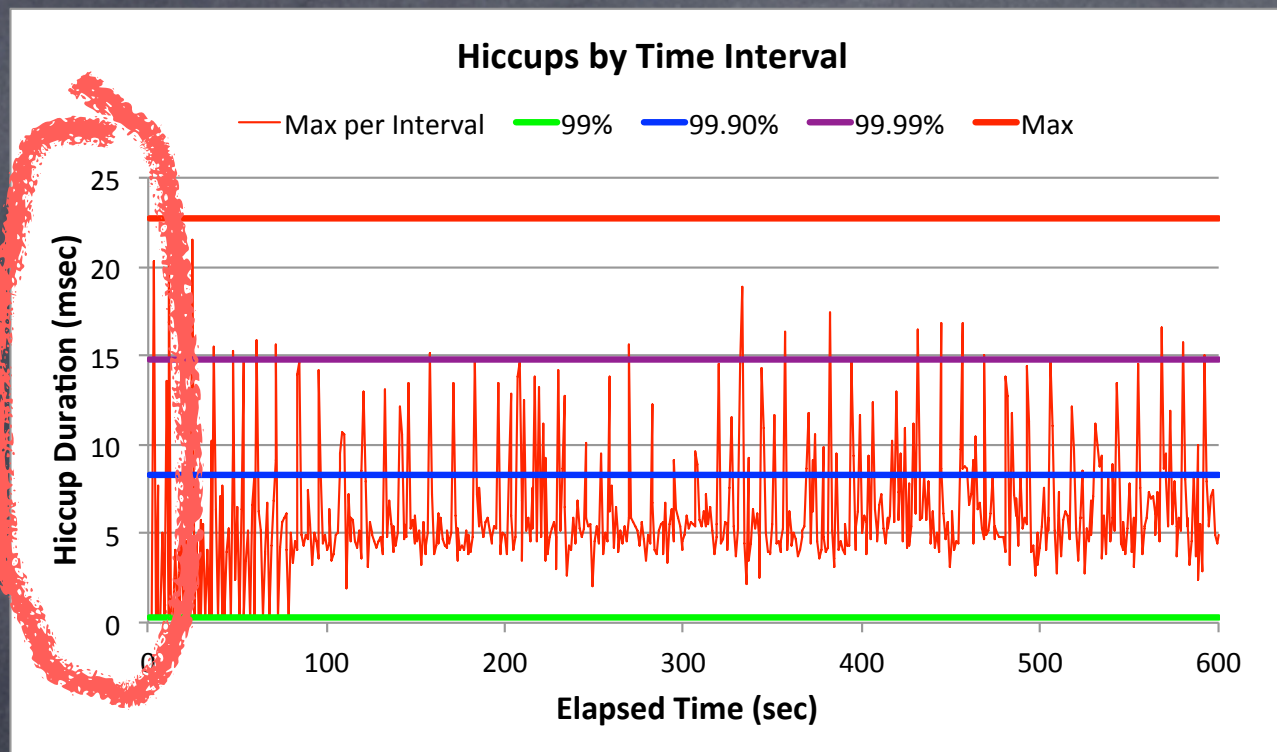


Zing

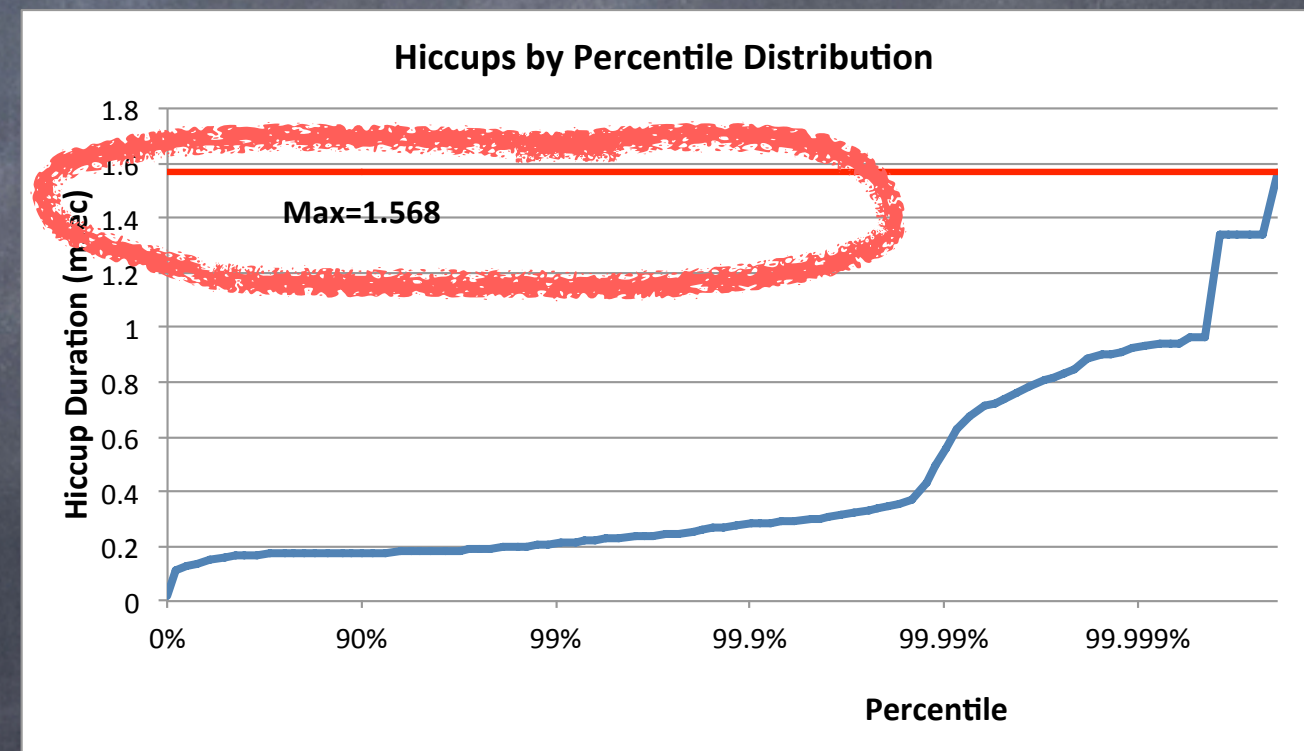
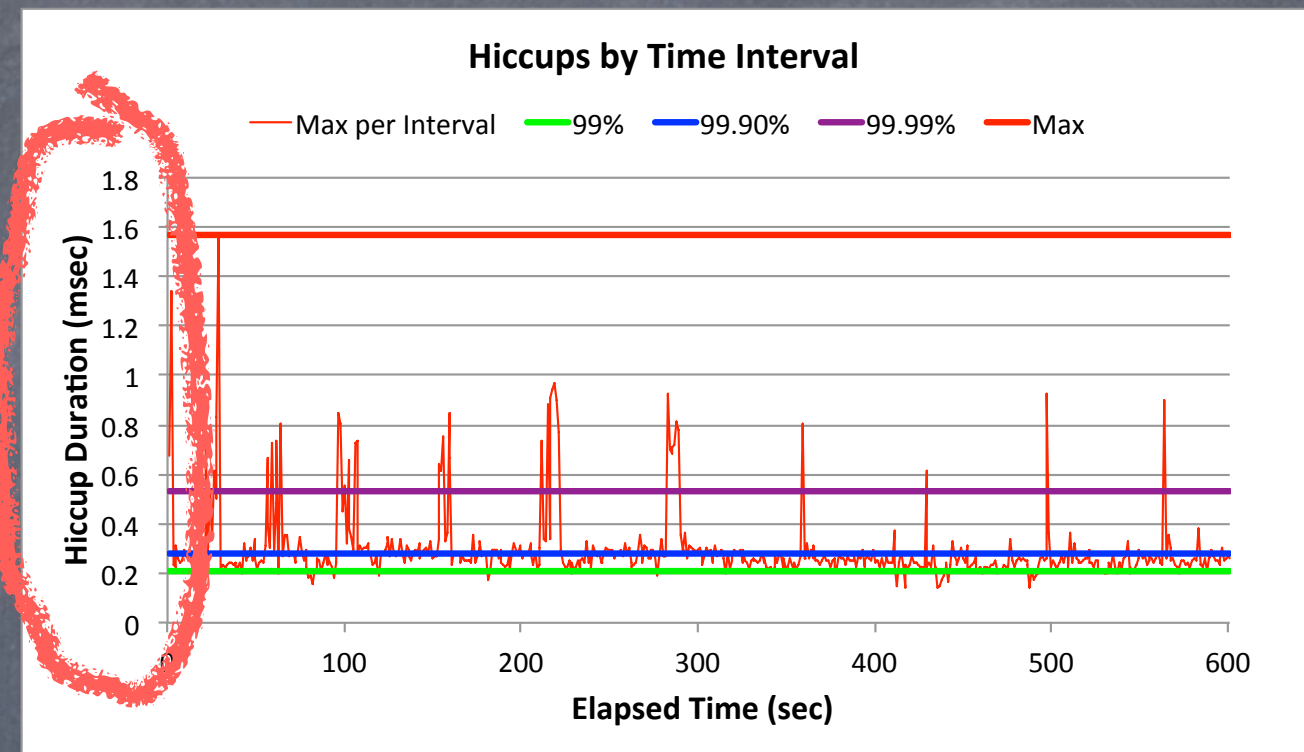


Low latency trading application

Oracle HotSpot (pure newgen)



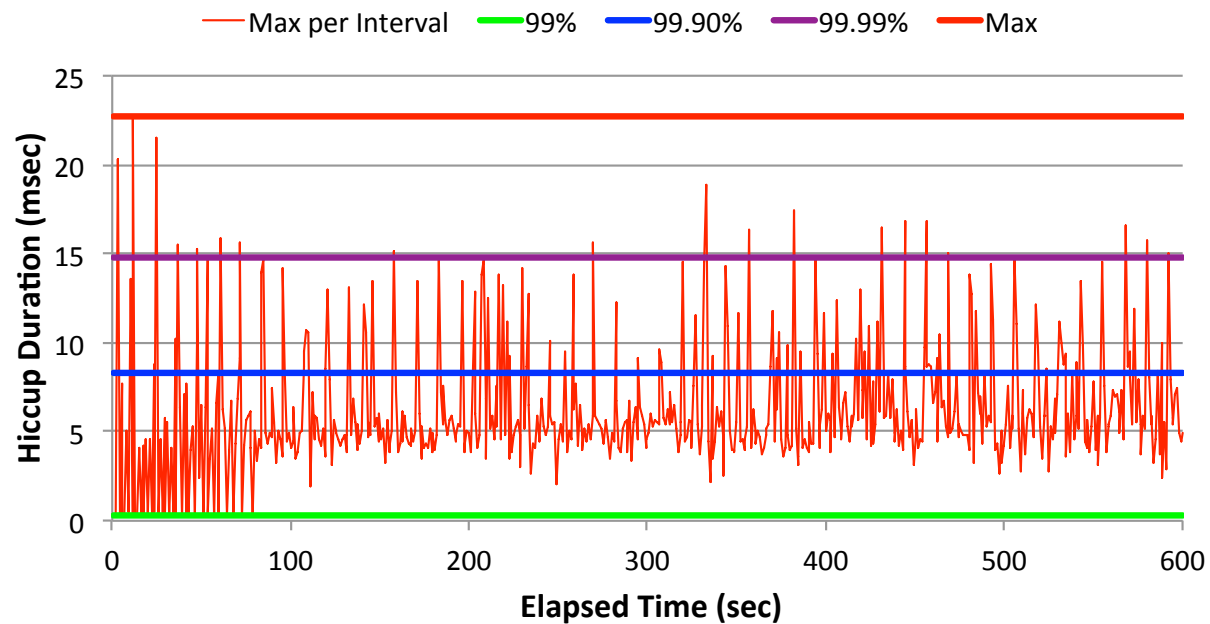
Zing



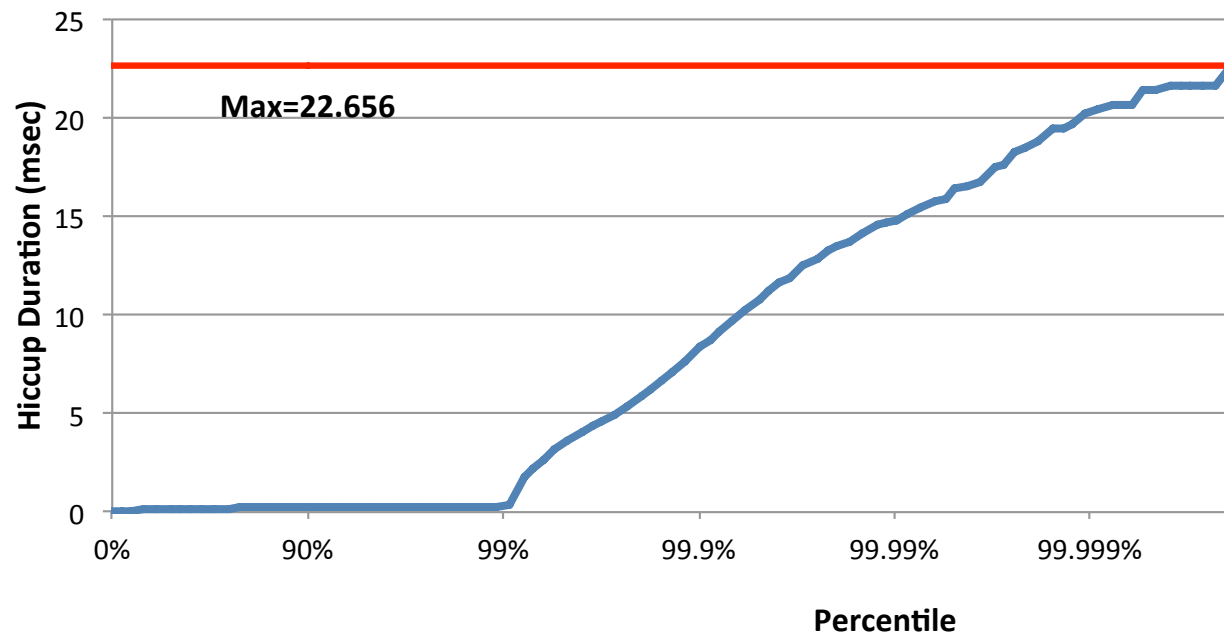
Low latency trading application

Oracle HotSpot (pure newgen)

Hiccups by Time Interval

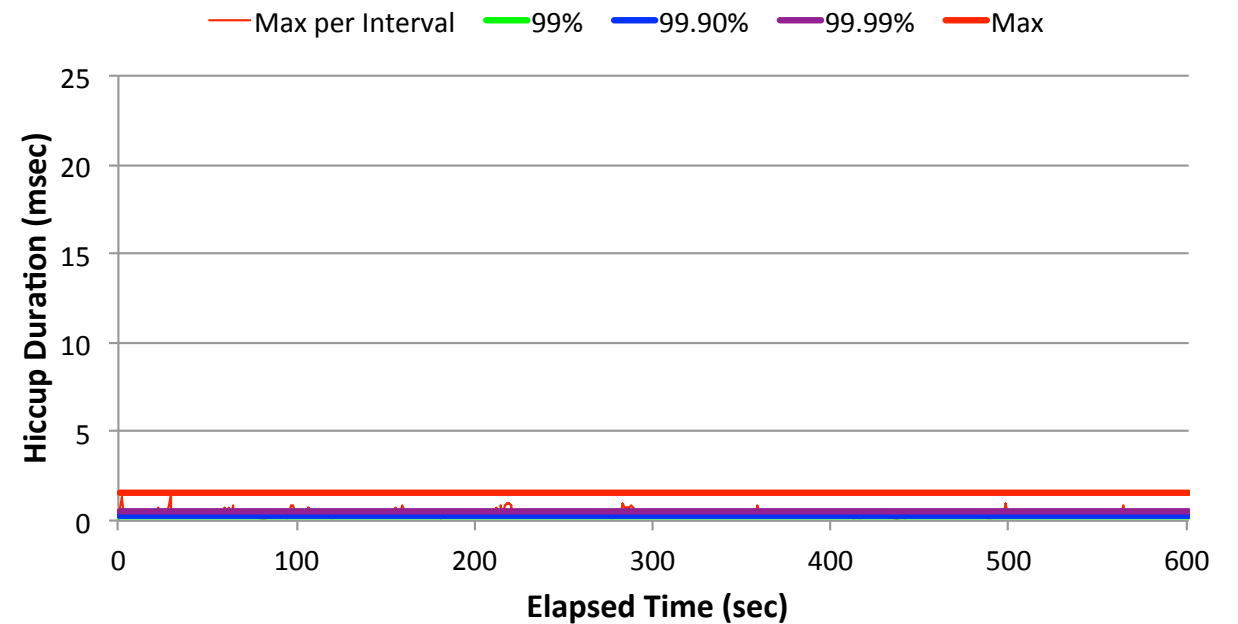


Hiccups by Percentile Distribution

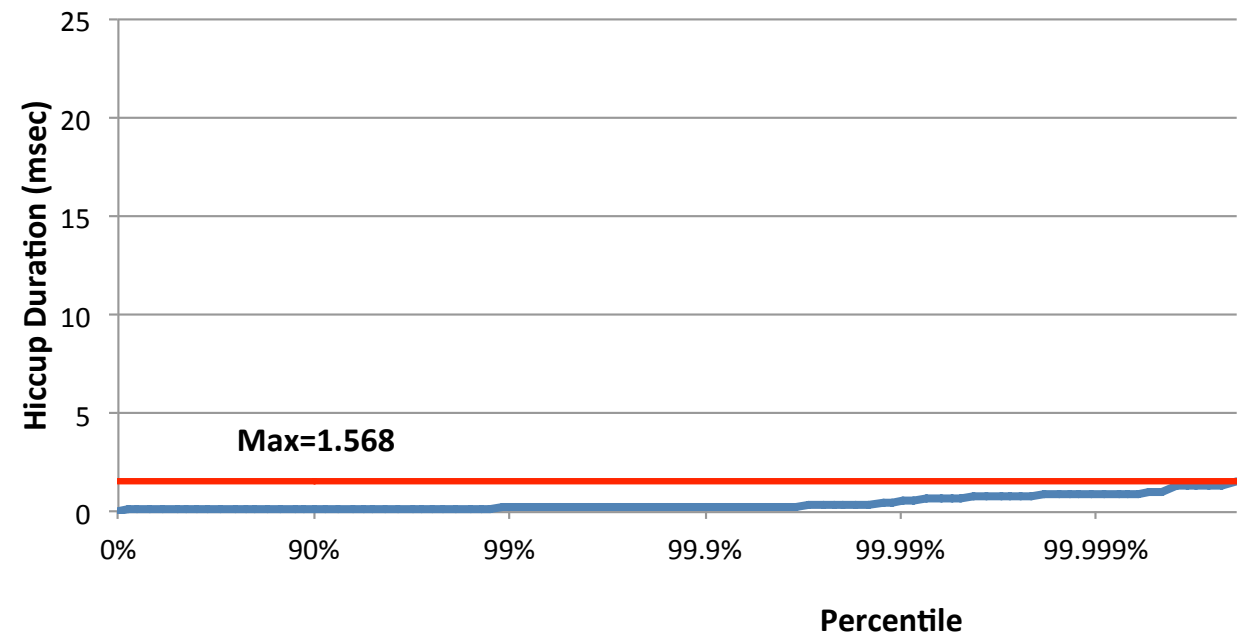


Zing

Hiccups by Time Interval



Hiccups by Percentile Distribution

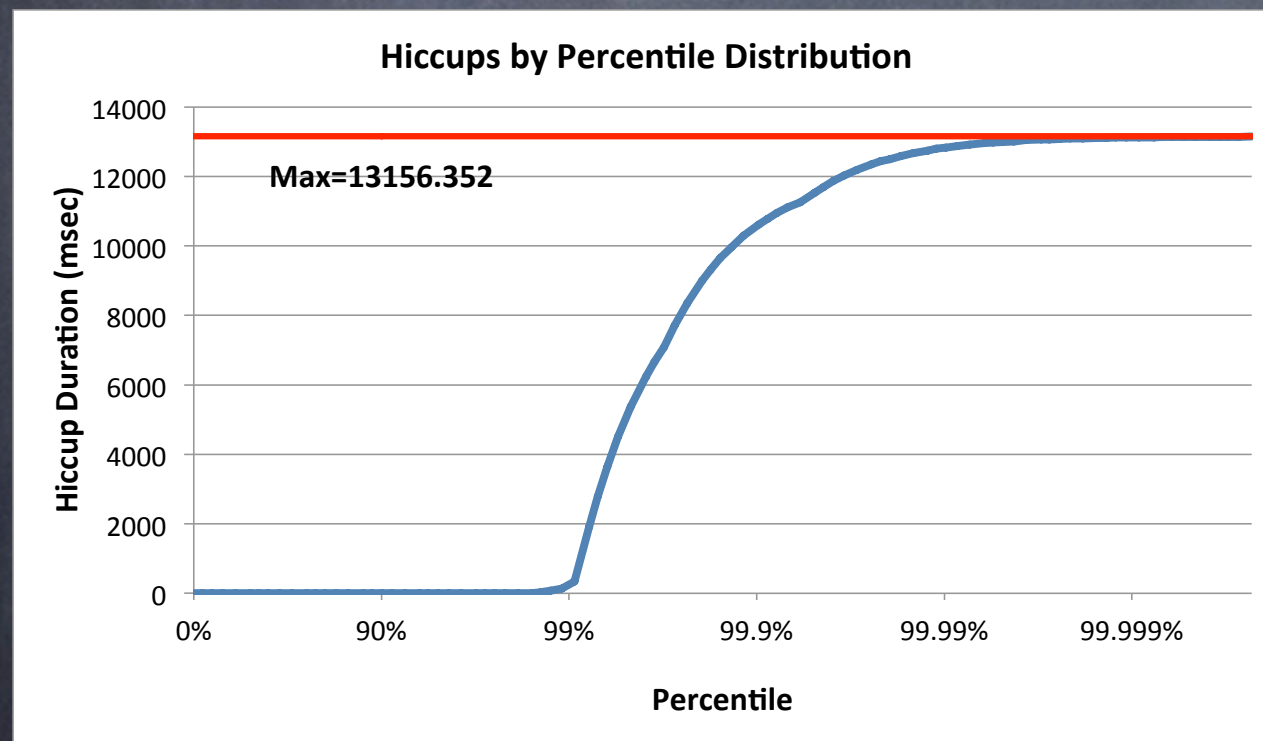
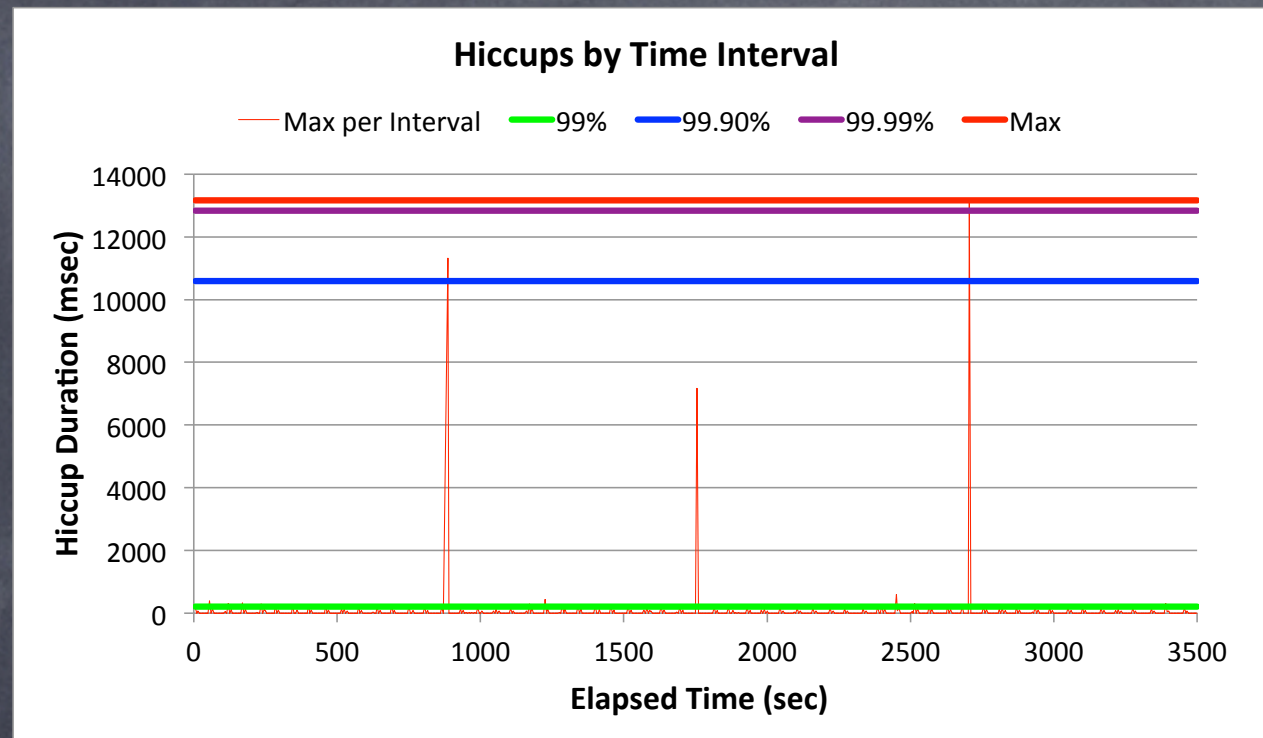


Low latency – Drawn to scale

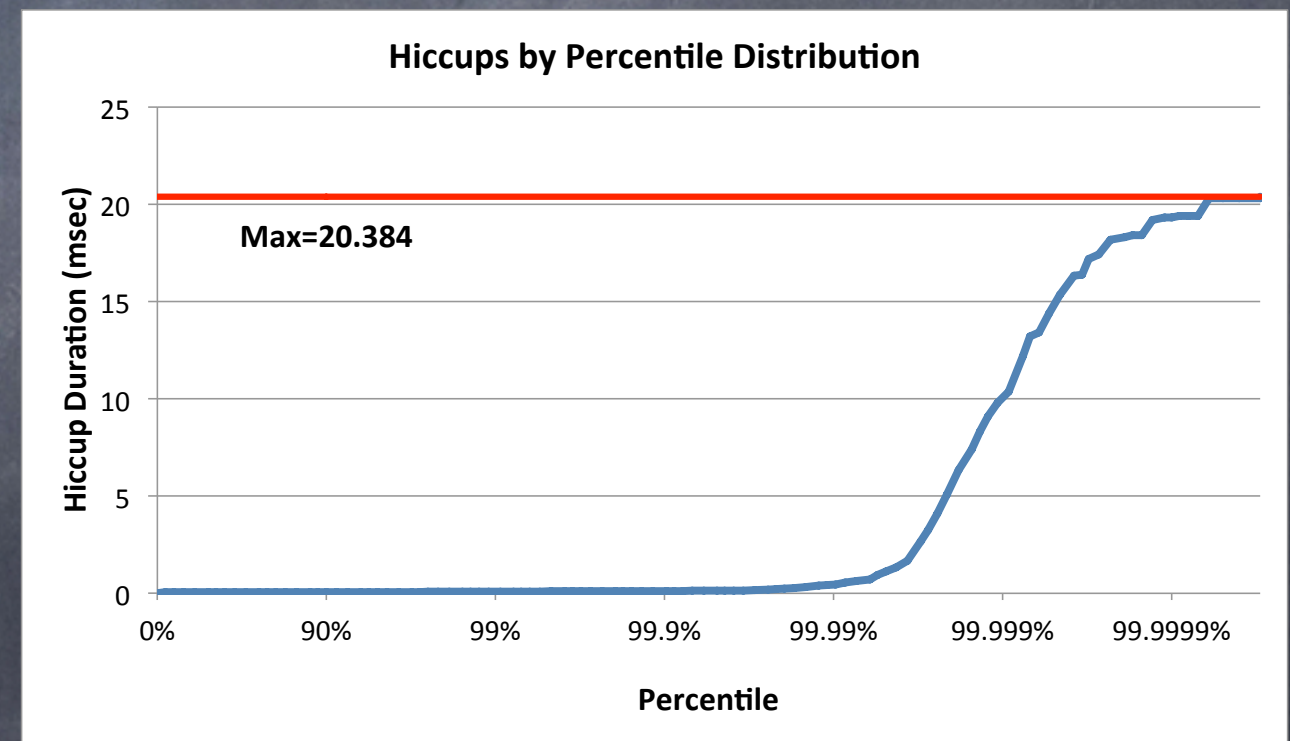
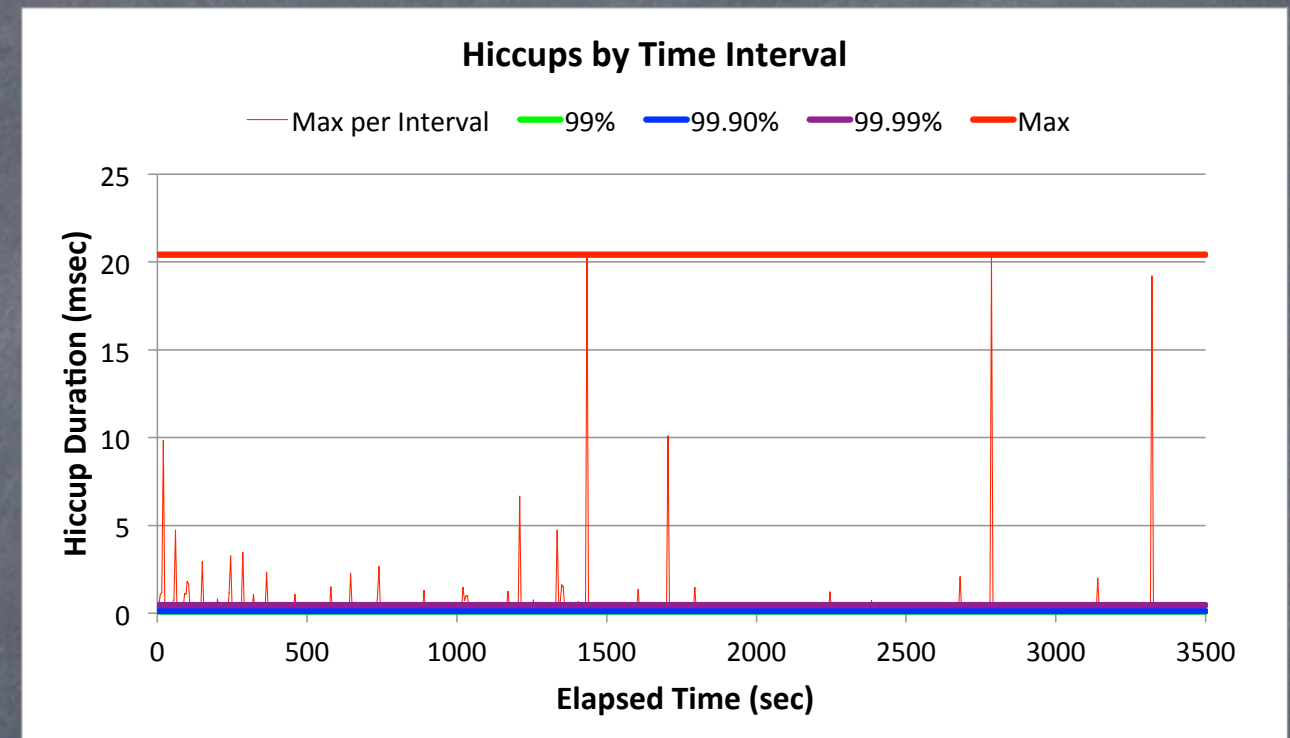
It's not just for
Low Latency

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

Oracle HotSpot CMS, 1GB in an 8GB heap

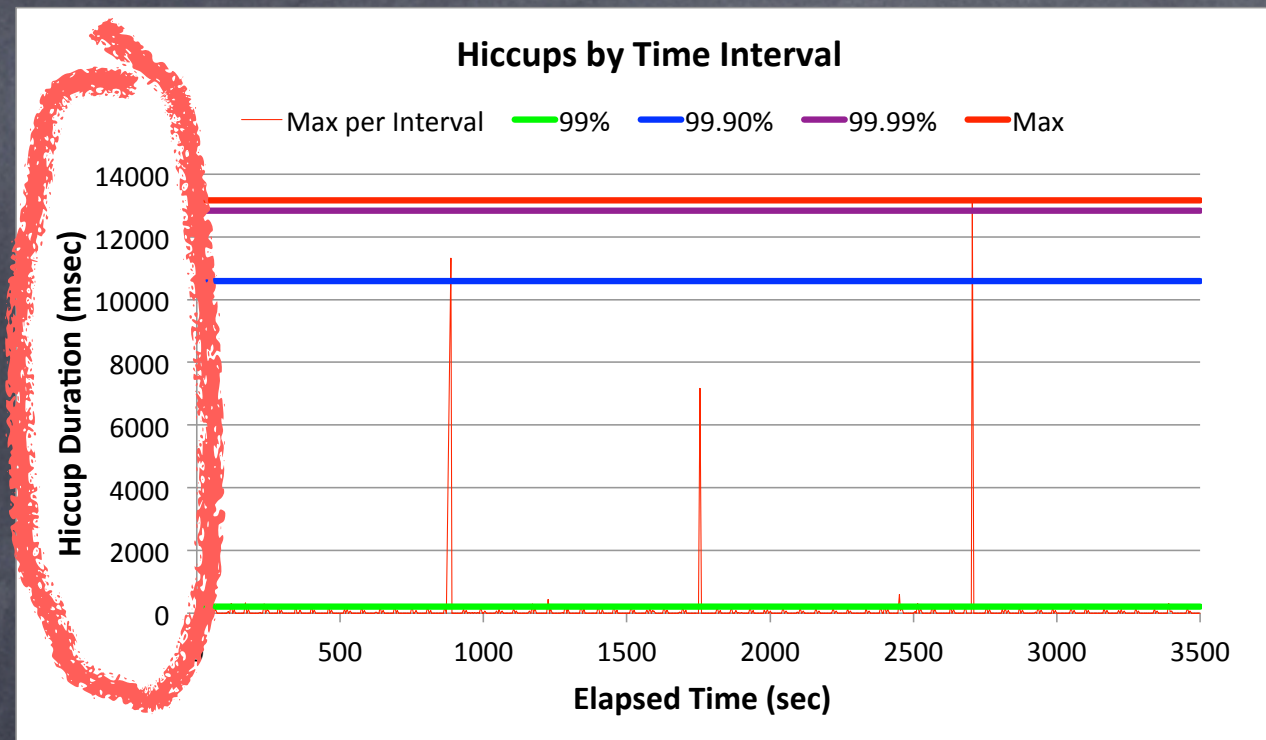


Zing 5, 1GB in an 8GB heap

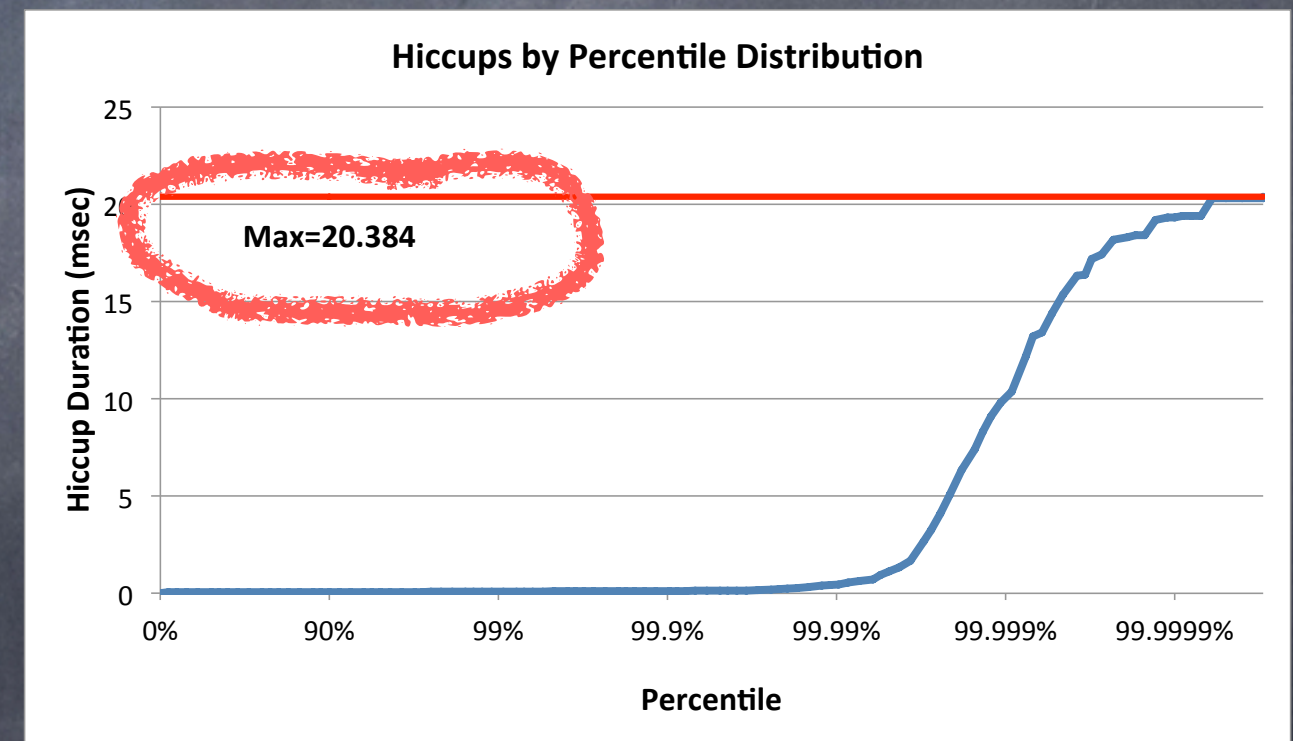
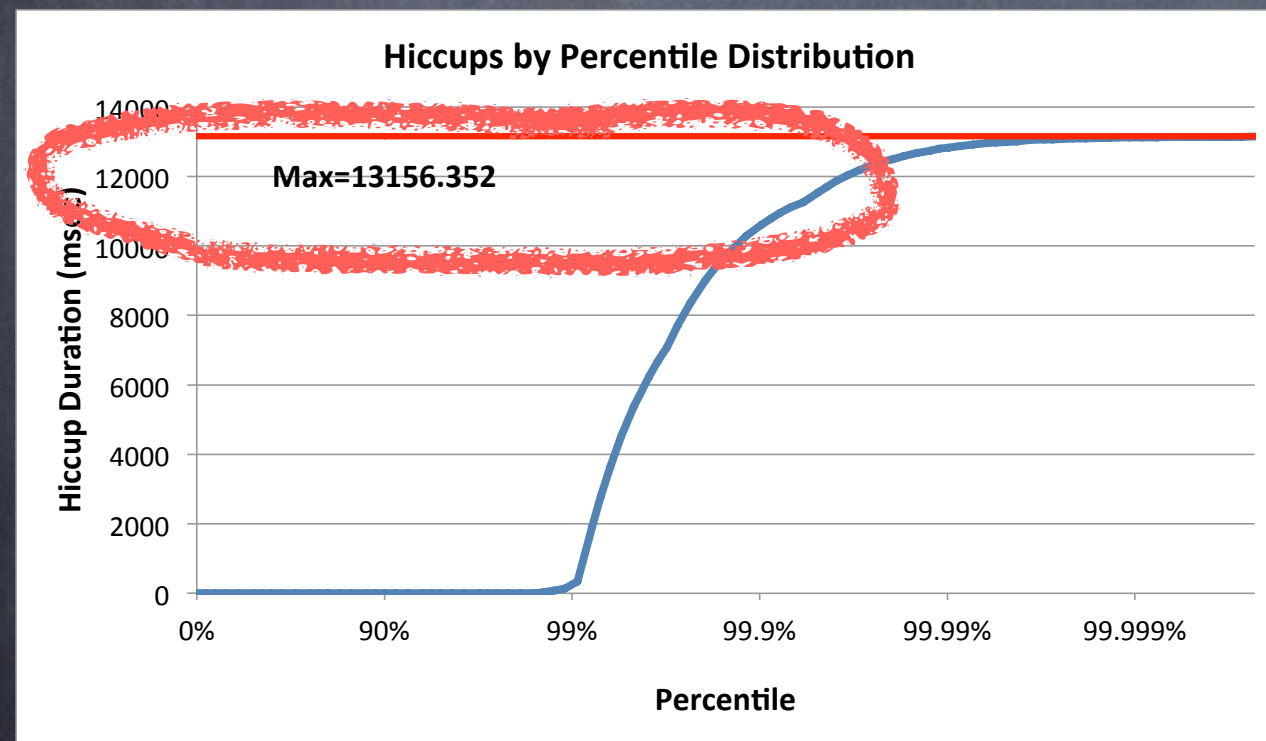
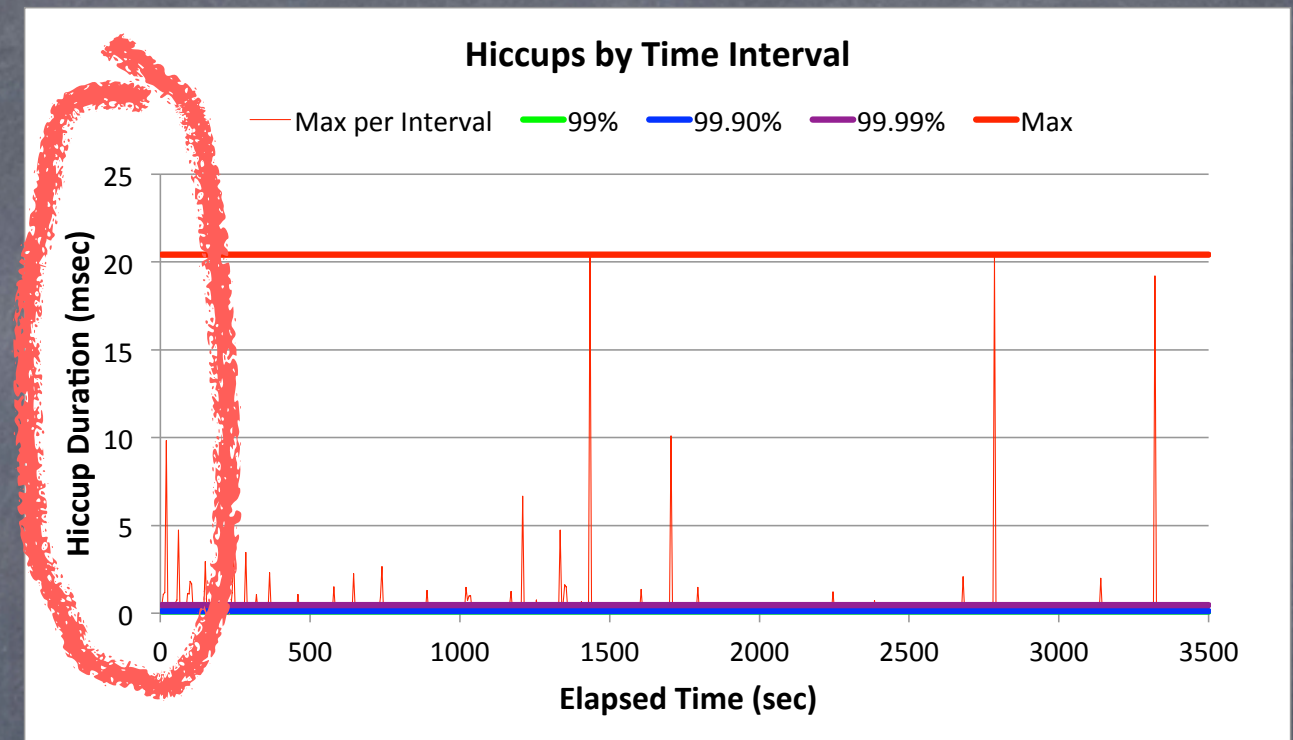


Portal Application, slow Ehcache "churn"

Oracle HotSpot CMS, 1GB in an 8GB heap

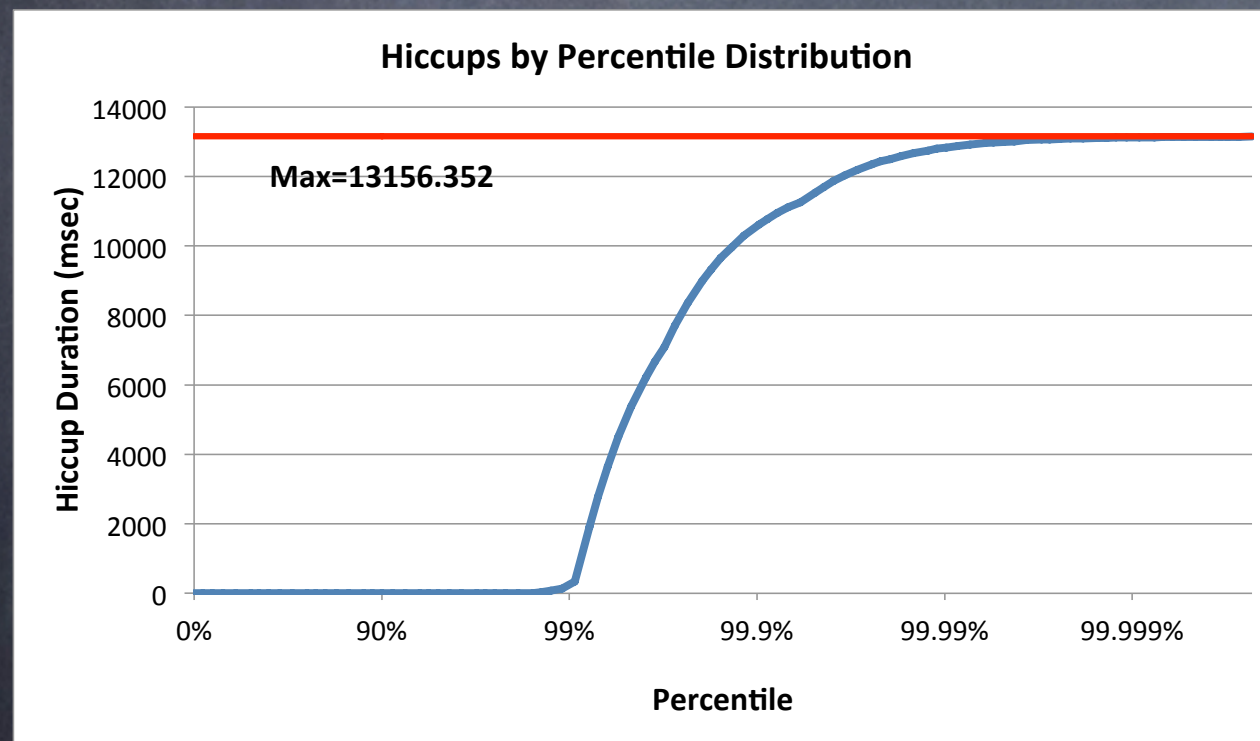
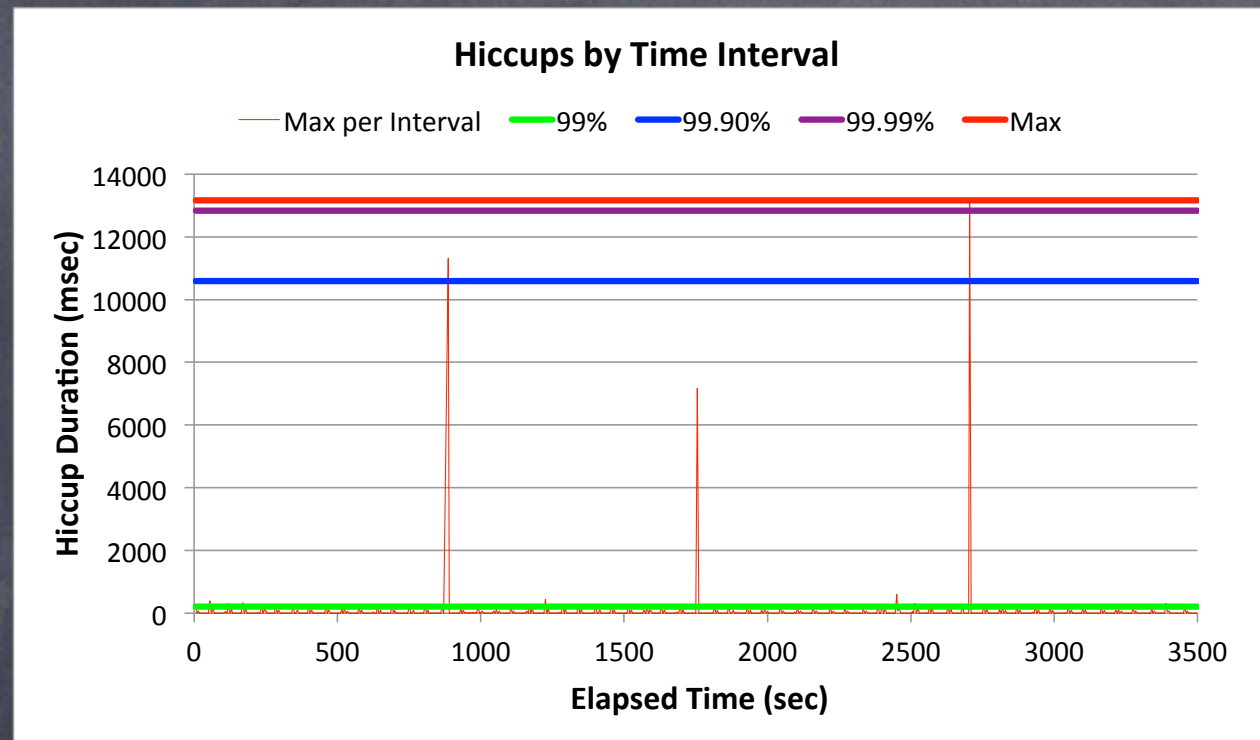


Zing 5, 1GB in an 8GB heap

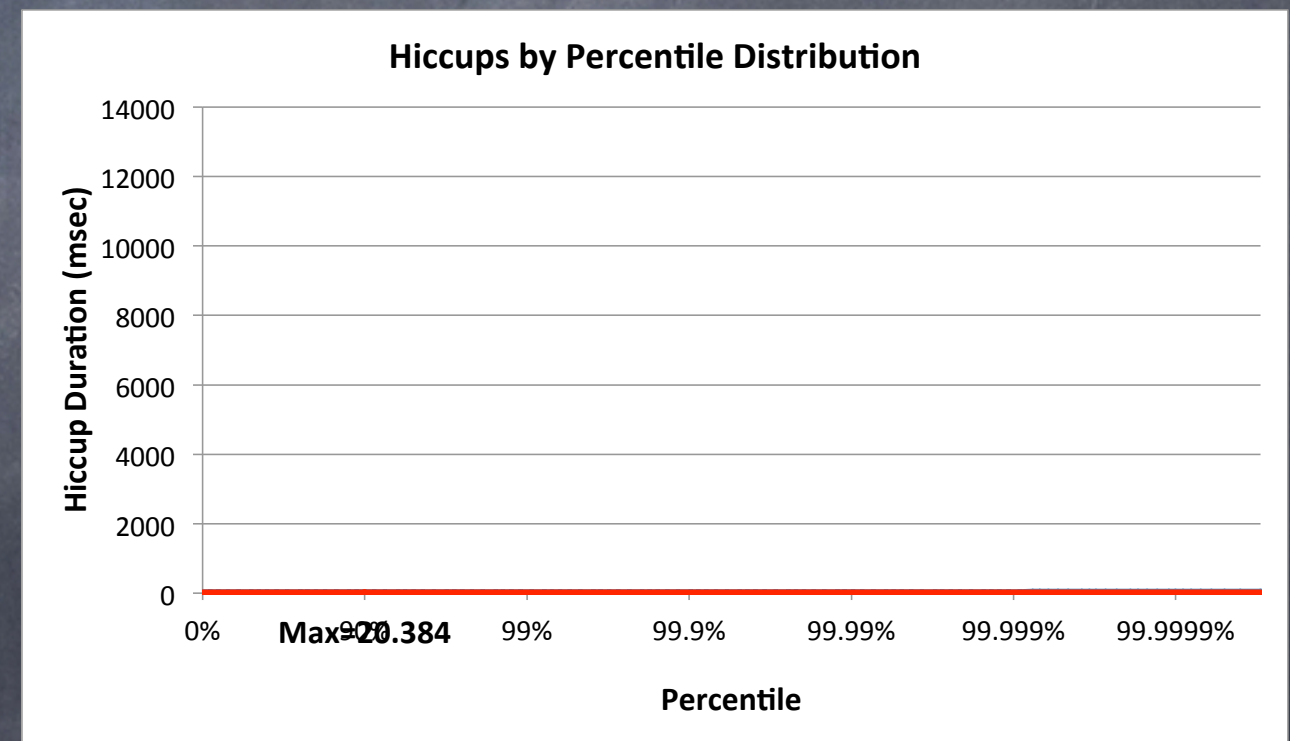
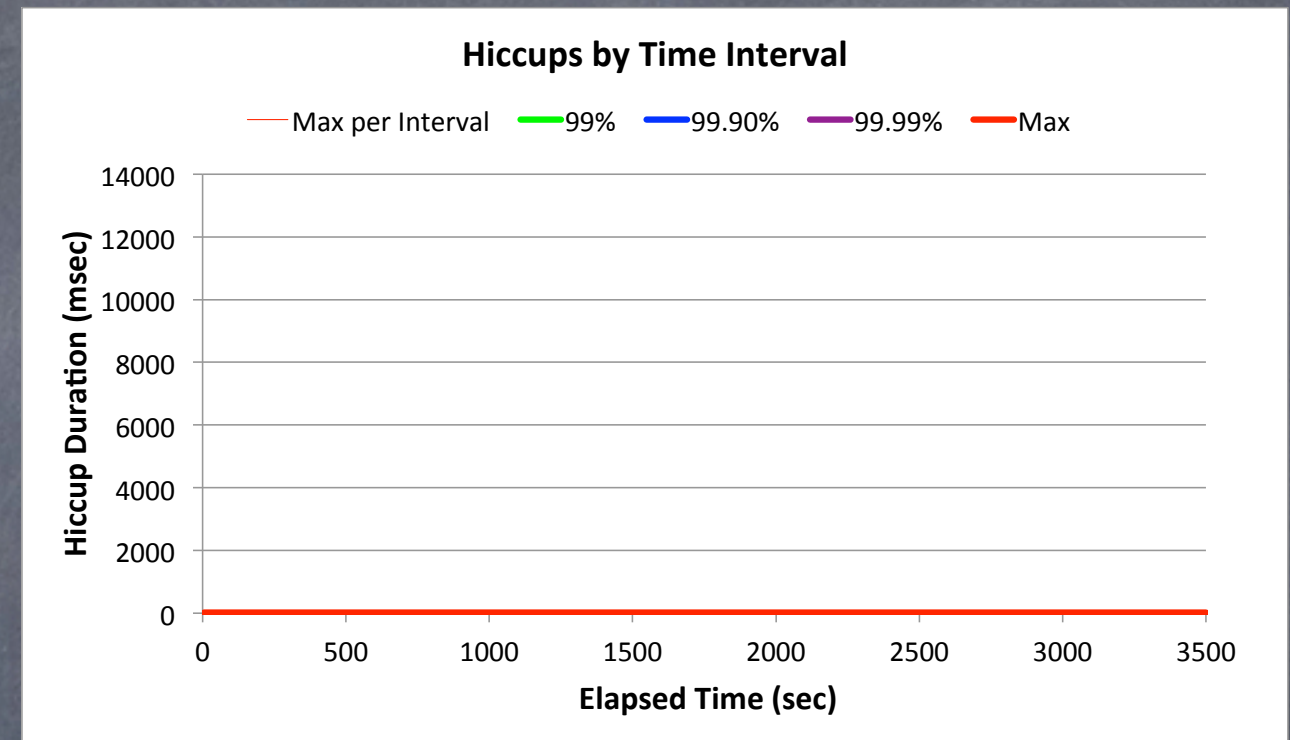


Portal Application, slow Ehcache "churn"

Oracle HotSpot CMS, 1GB in an 8GB heap



Zing 5, 1GB in an 8GB heap



Portal Application – Drawn to scale

Lets not forget about GC tuning

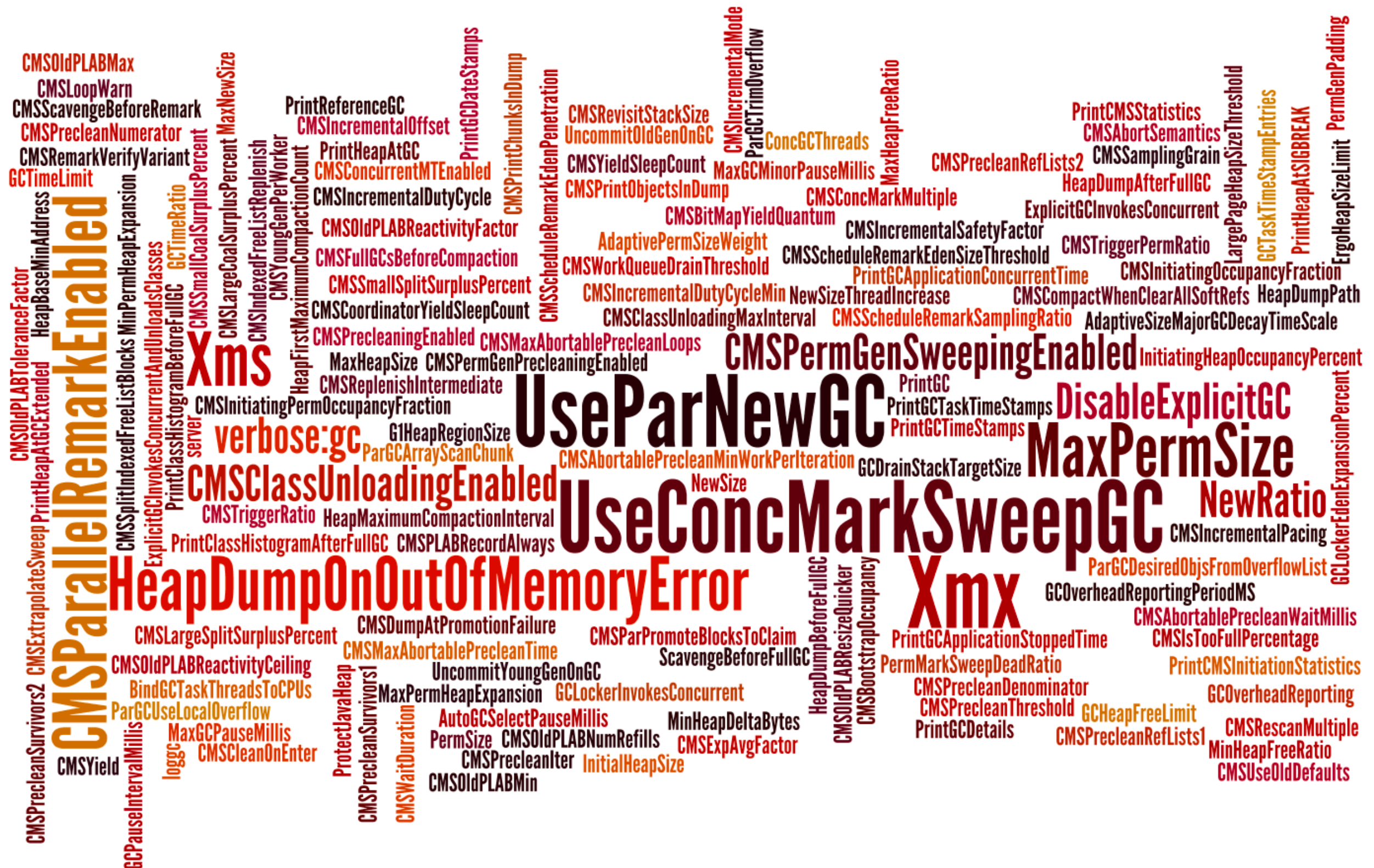
Java GC tuning is "hard"...

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 GC tuning flags



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

The complete guide to Zing GC tuning

```
java -Xmx40g
```


GC is only the biggest problem...

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 (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...

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...

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 2013, “Real” Java is finally viable for low latency applications

- GC is no longer a dominant issue, even for outliers
- 2–3msec **worst case** 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: A cure for the Java hiccups

Q & A

One-liner Takeaway:

Zing: A cure for the Java hiccups

jHiccup:

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