Enabling Java in Low Latency and Low Jitter Applications

Gil Tene, CTO & co-Founder, Azul Systems

Session # 8206

High level agenda

High level agenda

- Intro, jitter vs. JITTER

High level agenda

- Intro, jitter vs. JITTER
- Java in a low latency application world

High level agenda

- Intro, jitter vs. JITTER
- Java in a low latency application world
- The (historical) fundamental problems

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

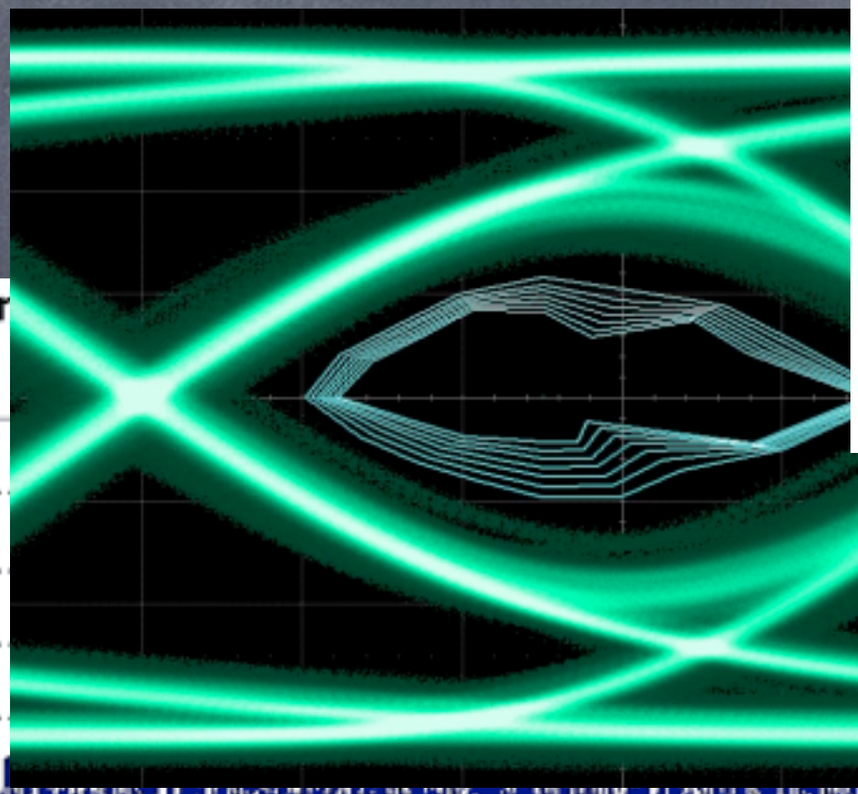
High level agenda

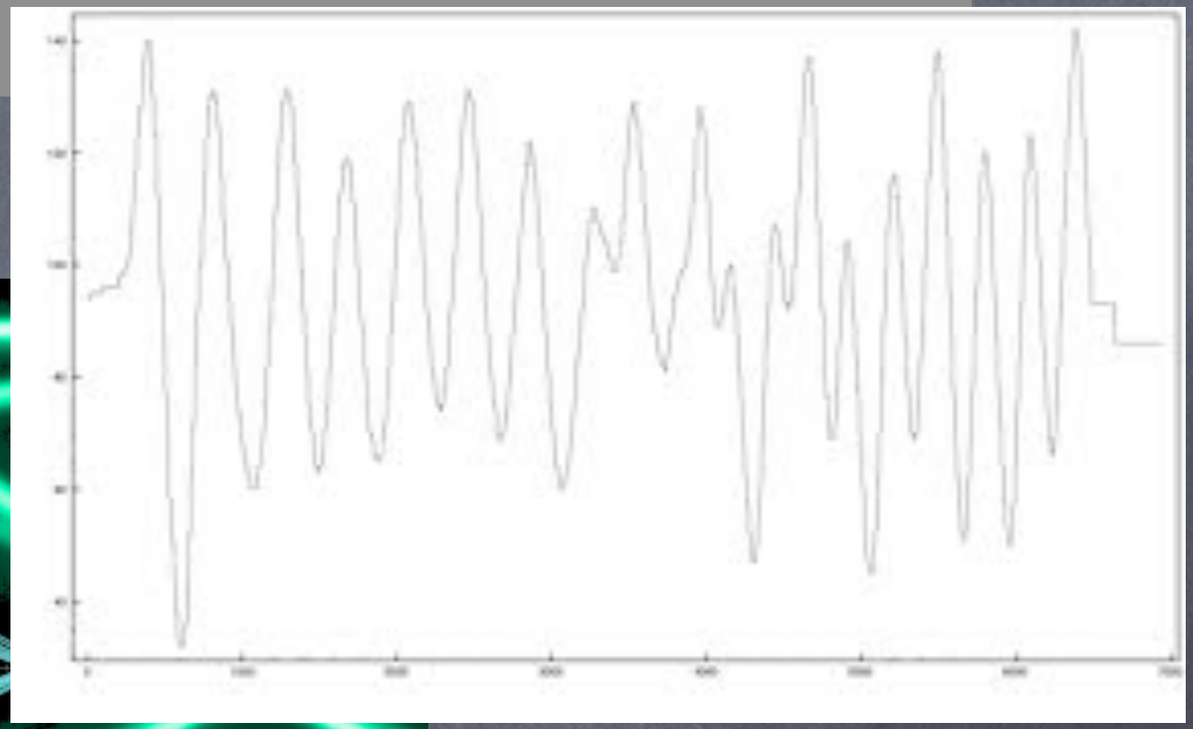
- Intro, jitter vs. JITTER
- Java in a low latency application world
- The (historical) fundamental problems
- What people have done to try to get around them
- What if the fundamental problems were eliminated?

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

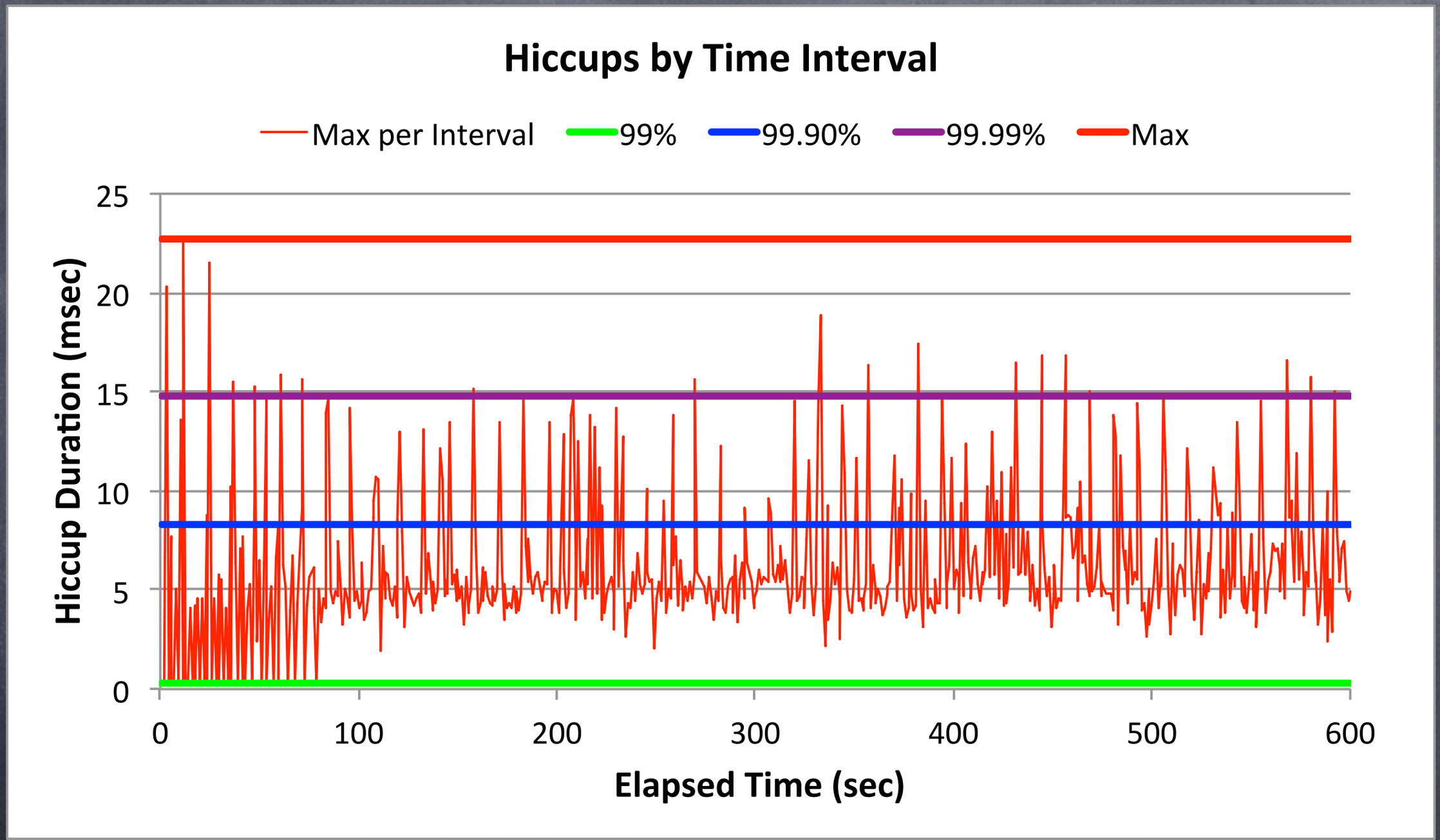
This is Jitter





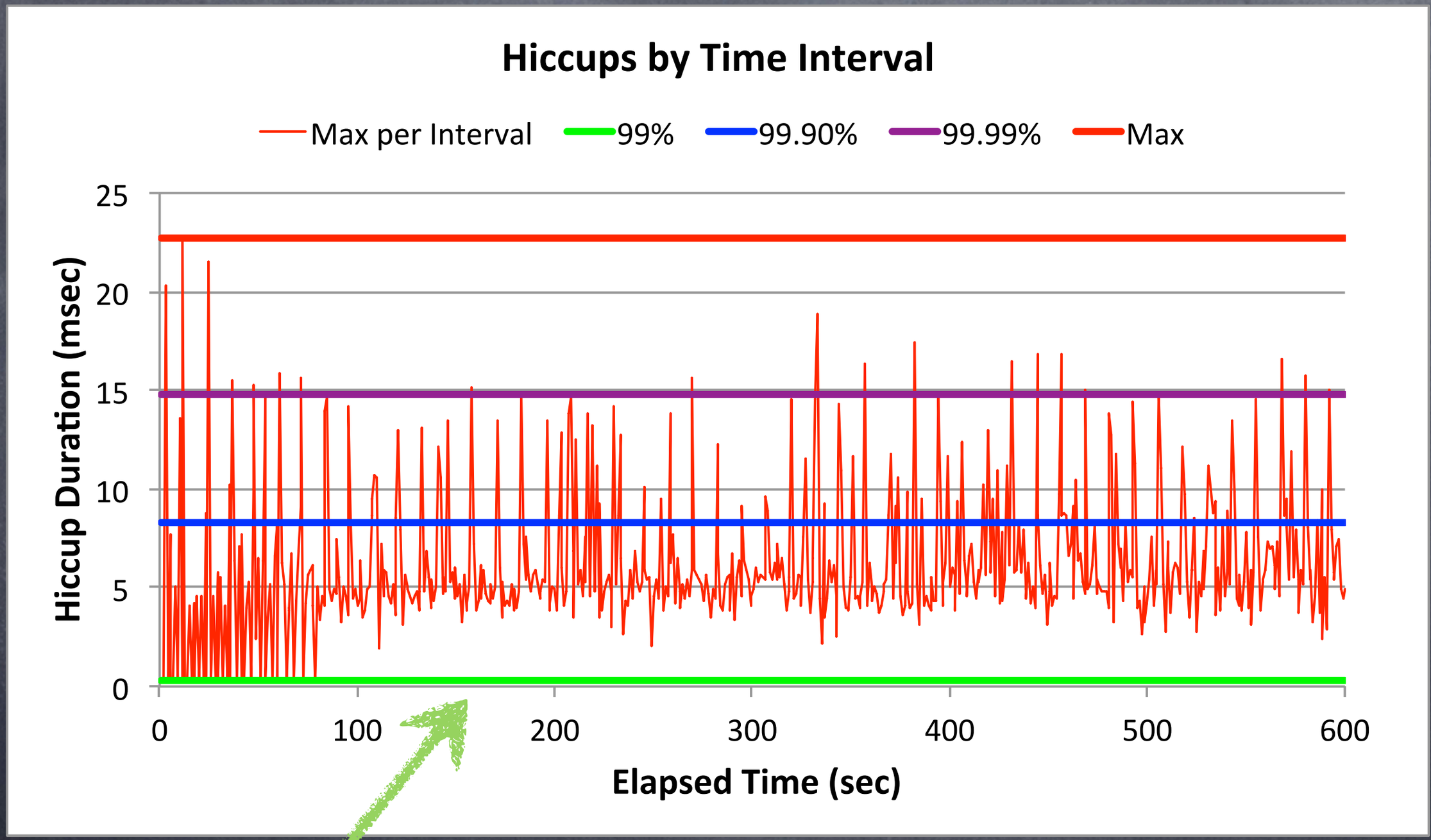
AZUL SYSTEMS

Is "jitter" a proper word for this?



AZUL SYSTEMS

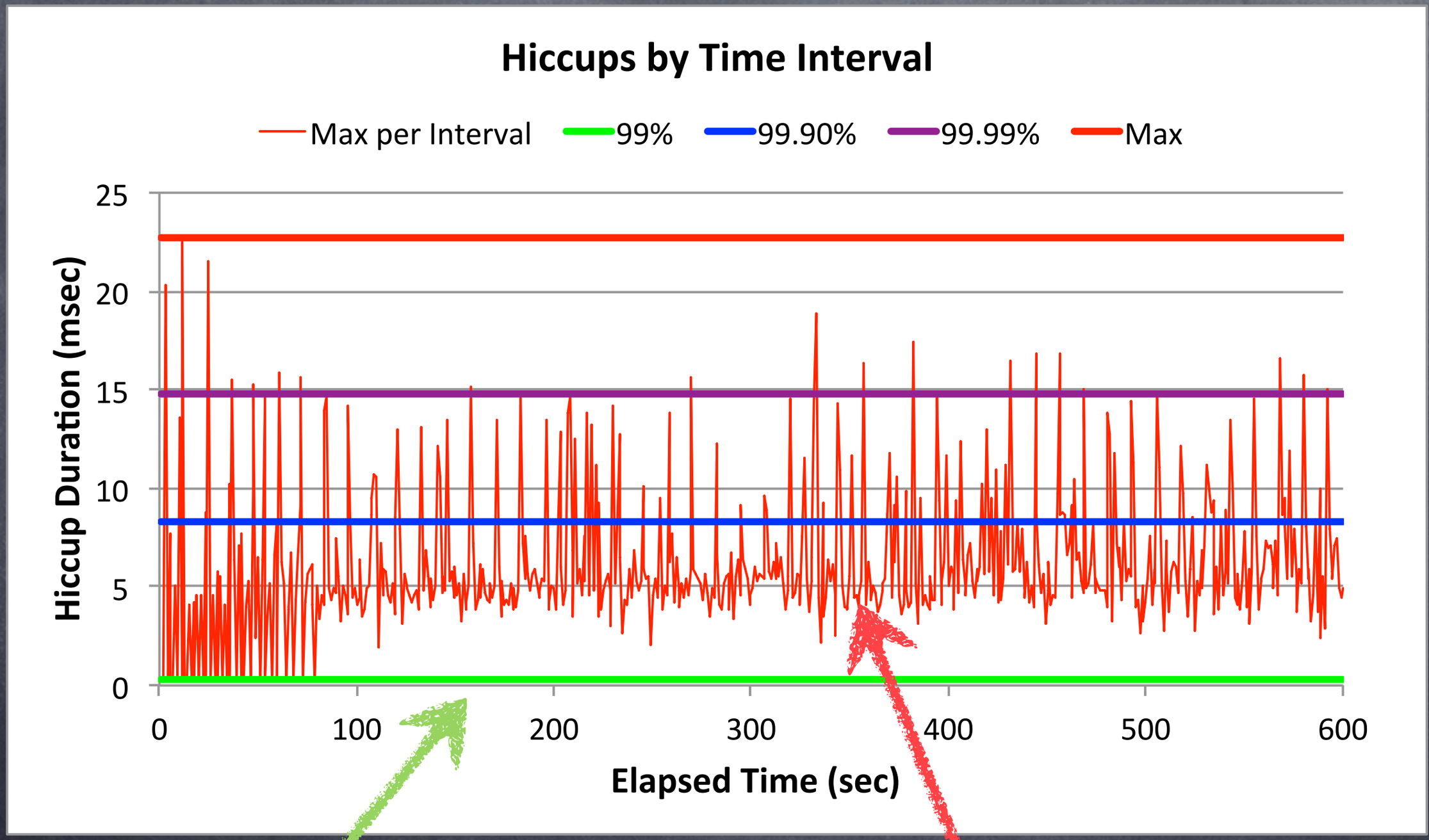
Is "jitter" a proper word for this?



99%'ile is ~60 usec

AZUL SYSTEMS

Is "jitter" a proper word for this?



99%'ile is ~60 usec

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

AZUL SYSTEMS

About me: Gil Tene

- co-founder, CTO
  @Azul Systems

About me: Gil Tene

- co-founder, CTO
  @Azul Systems
- Have been working on
  a "think different" GC
  approaches since 2002

About me: Gil Tene

- co-founder, CTO @Azul Systems
- Have been working on a "think different" GC approaches since 2002

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

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)

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

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

About Azul

- We make scalable Virtual Machines

AZUL SYSTEMS

About Azul

- We make scalable Virtual Machines
- Have built "whatever it takes to get job done" since 2002

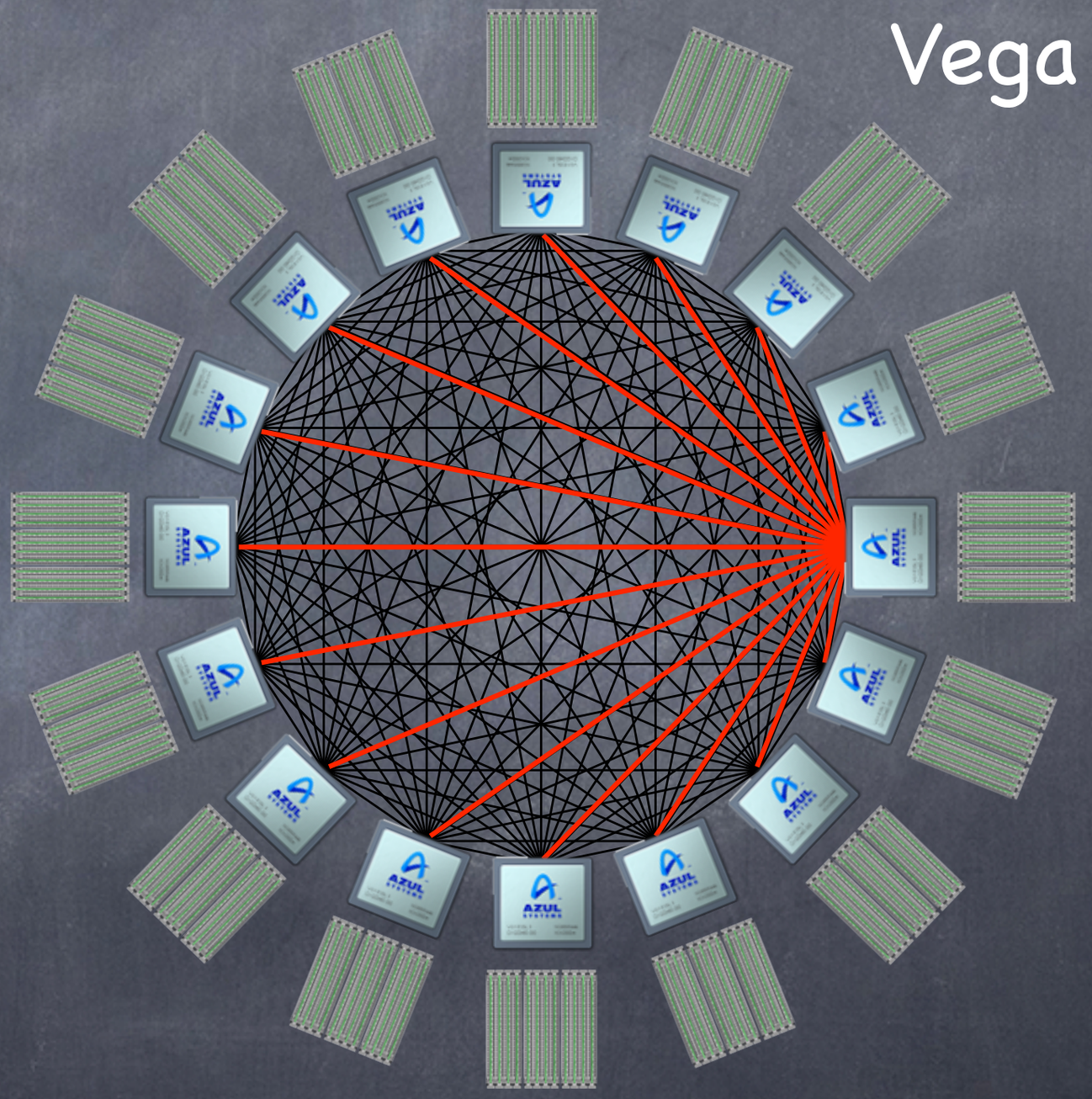
AZUL SYSTEMS

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)

About Azul

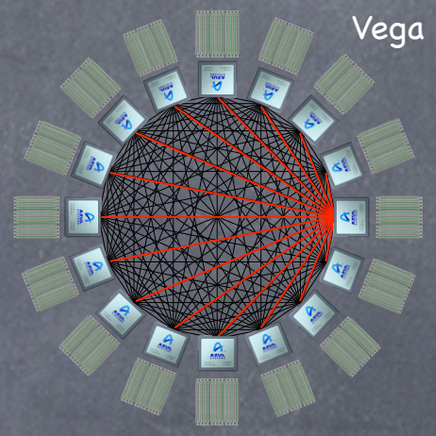
- We make scalable Virtual Machines
- Have built "whatever it takes to get job done" since 2002
- 3 generations of custom SMP Multi-core HW (Vega)



AZUL SYSTEMS

About Azul

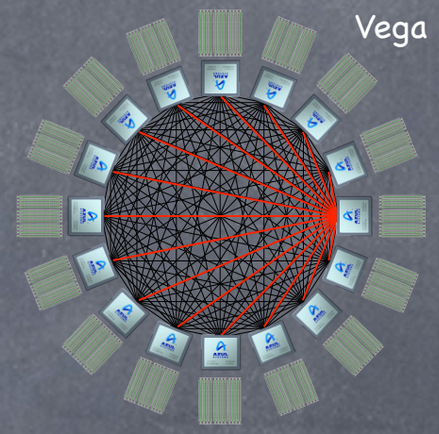
- We make scalable Virtual Machines
- Have built "whatever it takes to get job done" since 2002
- 3 generations of custom SMP Multi-core HW (Vega)



AZUL SYSTEMS

About Azul

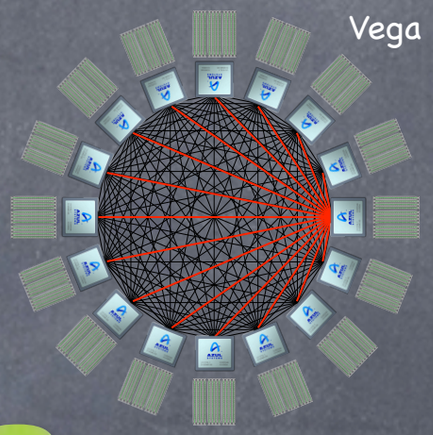
- We make scalable Virtual Machines
- Have built "whatever it takes to get job done" since 2002
- 3 generations of custom SMP Multi-core HW (Vega)
- **Now Pure software for commodity x86 (Zing)**



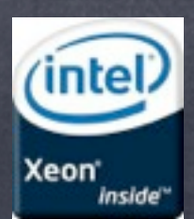
AZUL SYSTEMS

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)**



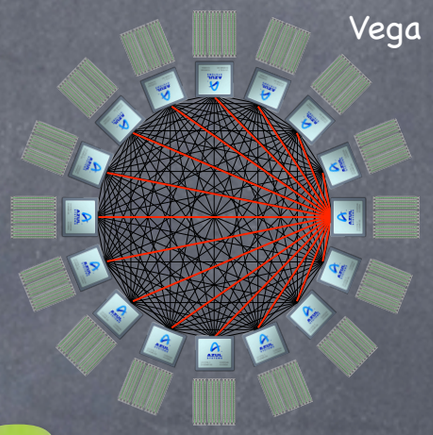
zing



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



Zing

AZUL SYSTEMS

Java in the low latency world

Java in a low latency world

Java in a low latency world

- Why do people use Java for low latency apps?

Java in a low latency world

- Why do people use Java for low latency apps?
- Are they crazy?

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

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

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

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, ...

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?"

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

- Strategies have a shelf life

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

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

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

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?

So what is the problem?

Is Java Slow?

- No

So what is the problem?
Is Java Slow?

- No
- A good programmer will get roughly the same speed from both Java and 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

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...

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...

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?

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

- Let's ignore the bad multi-second pauses for now...

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

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

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

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%

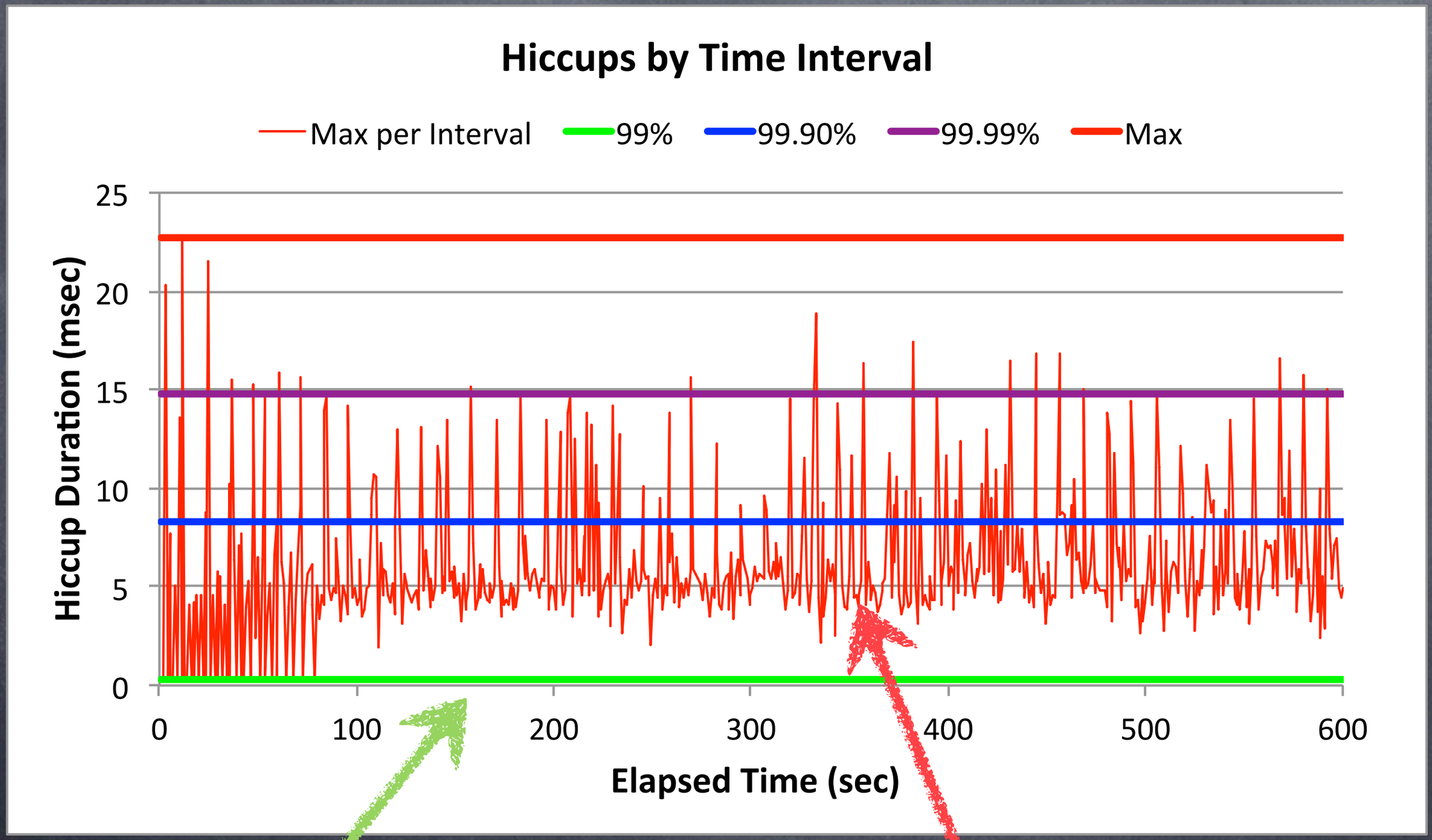
Stop-The-World Garbage Collection: How bad is it?

- 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"

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

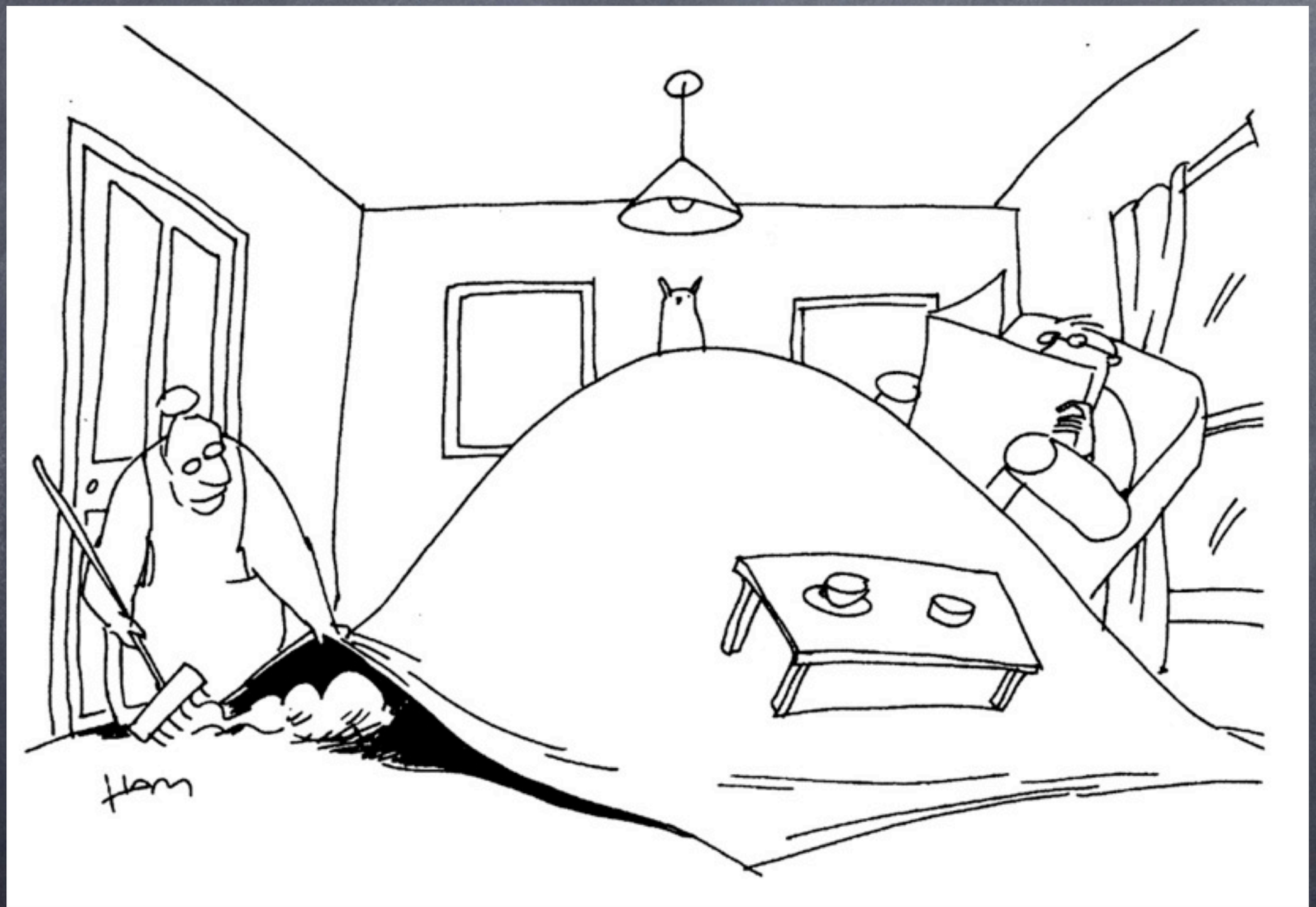
Max is ~30,000% higher than "typical"

AZUL SYSTEMS

One way to deal with Stop-The-World GC

One way to deal with Stop-The-World GC

A way to deal with Stop-The-World GC



Another way to cope: "Creative Language"

AZUL SYSTEMS

Another way to cope: "Creative Language"

- "Guarantee a worst case of 5 msec, 99% of the time"

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"

AZUL SYSTEMS

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"

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?

What do actual low latency developers do about it?

- They use "Java" instead of Java

What do actual low latency developers do about it?

- They use "Java" instead of Java
- They write "in the Java syntax"

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

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

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 libraries)

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 libraries)
- They train developers for their local discipline

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 libraries)
- 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 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 libraries)
- 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?

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

- They still see pauses (usually ranging to tens of msec)

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

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

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

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

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
- ...

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

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

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

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
  - When they say "mostly concurrent", or "mostly incremental", or "pause target", they refer only to the OldGen part of the collector
  - Focus is on trying to address the many-second pause problem
  - Usually by sweeping it farther and farther the rug

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
  - When they say "mostly concurrent", or "mostly incremental", or "pause target", they refer only to the OldGen part of the collector
  - Focus is on trying to address the many-second pause problem
  - Usually by sweeping it farther and farther the rug
- ALL use a Fallback to Full Stop-the-world Collection
  - Used for dealing with the inevitable pile of dust under the rug
  - Used to recover when other mechanisms fail
  - Hidden under the term "Mostly"...

Sustainable Throughput:
The throughput achieved while safely maintaining service levels

Sustainable Throughput:
The throughput achieved while safely maintaining service levels

Sustainable Throughput:
The throughput achieved while safely maintaining service levels

At Azul, STW-GC was addressed head-on

At Azul, STW-GC was addressed head-on

- We decided to focus on the right core problems

At Azul, STW-GC was addressed head-on

- We decided to focus on the right core problems
- Scale & productivity being limited by responsiveness

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

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

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

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

AZUL SYSTEMS

The Zing "C4" Collector
Continuously Concurrent Compacting Collector

- Concurrent, compacting old generation

AZUL SYSTEMS

The Zing "C4" Collector
Continuously Concurrent Compacting Collector

- Concurrent, compacting old generation

AZUL SYSTEMS

The Zing "C4" Collector
Continuously Concurrent Compacting Collector

- Concurrent, compacting old generation
- Concurrent, compacting new generation

AZUL SYSTEMS

The Zing "C4" Collector
Continuously Concurrent Compacting Collector

- Concurrent, compacting old generation
- Concurrent, compacting new generation

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

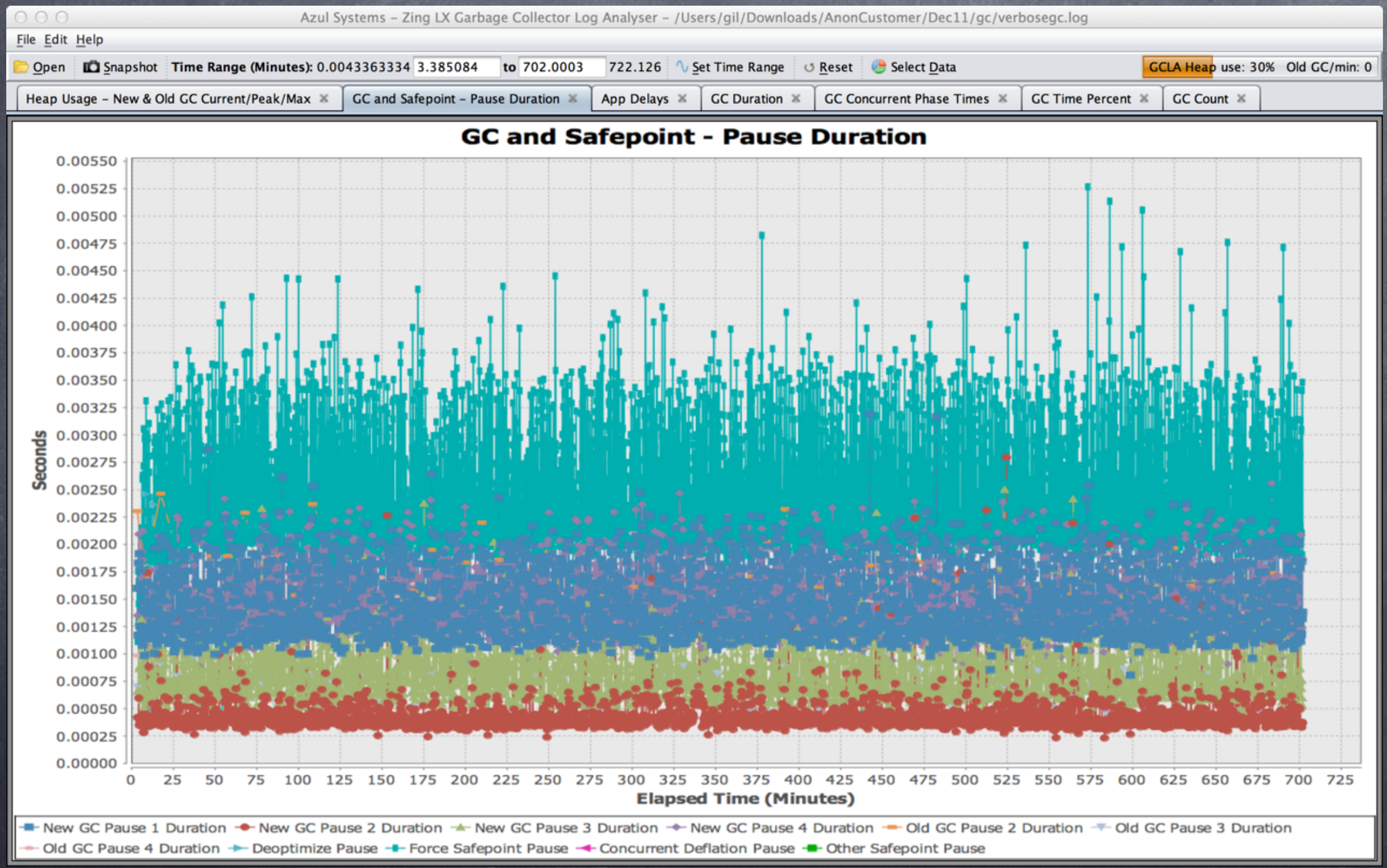
The Zing "C4" Collector
Continuously Concurrent Compacting Collector

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

AZUL SYSTEMS

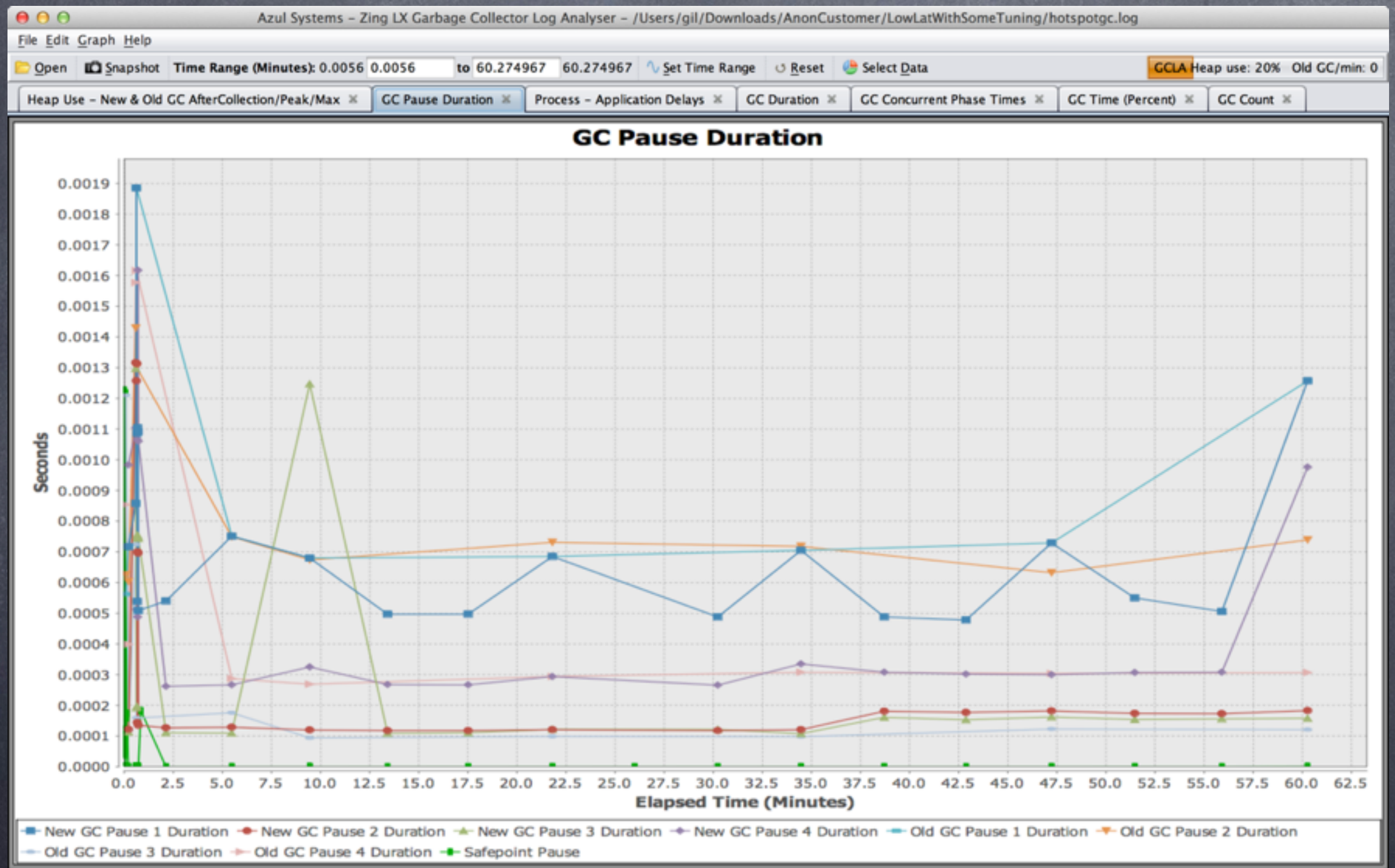
Benefits

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

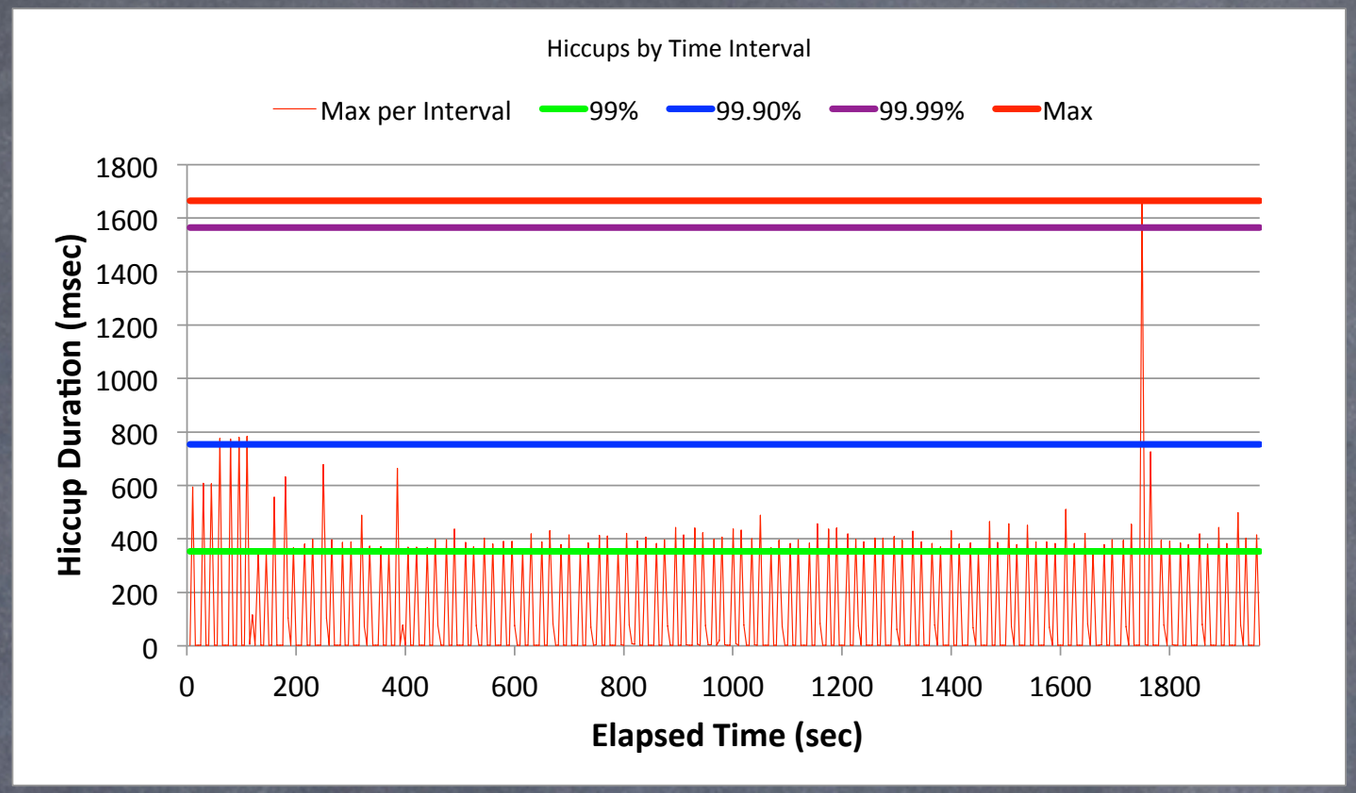


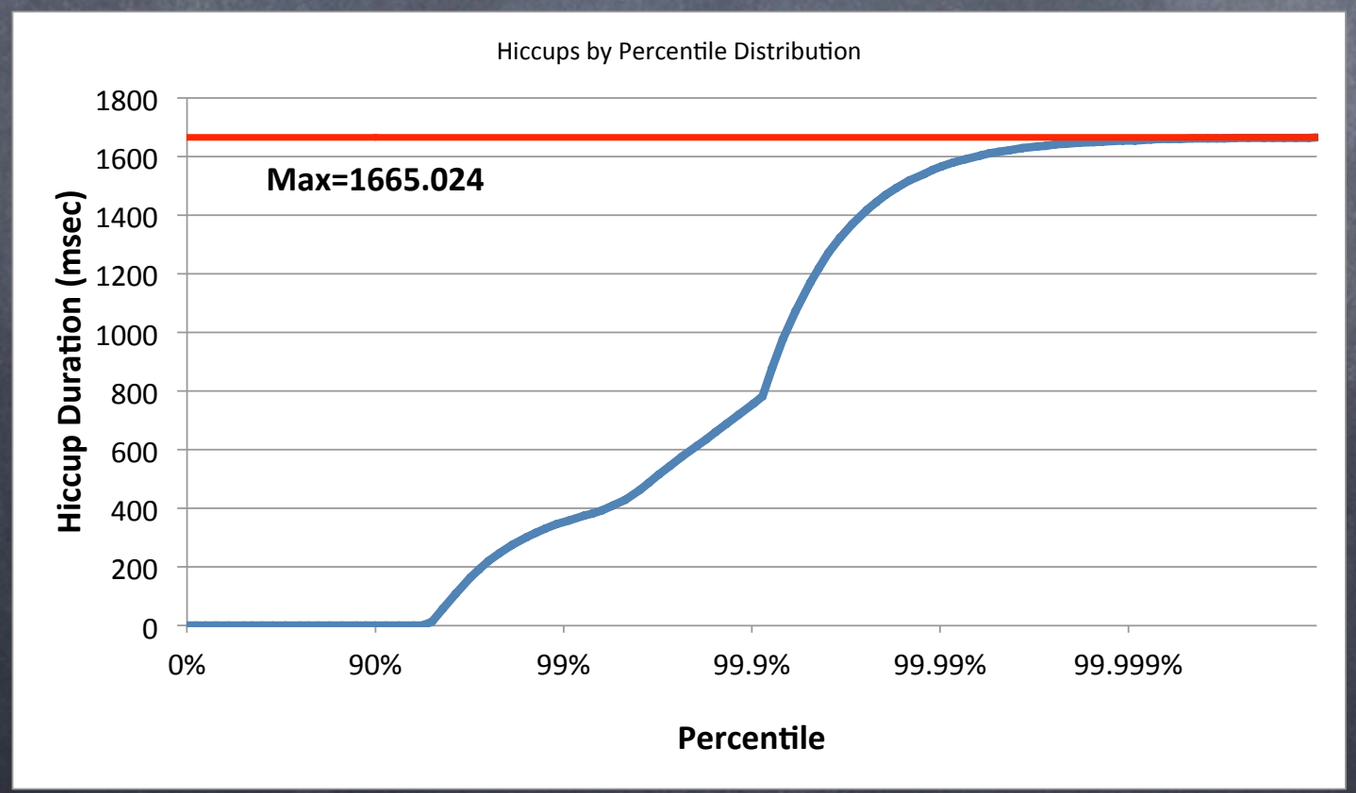
AZUL SYSTEMS

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



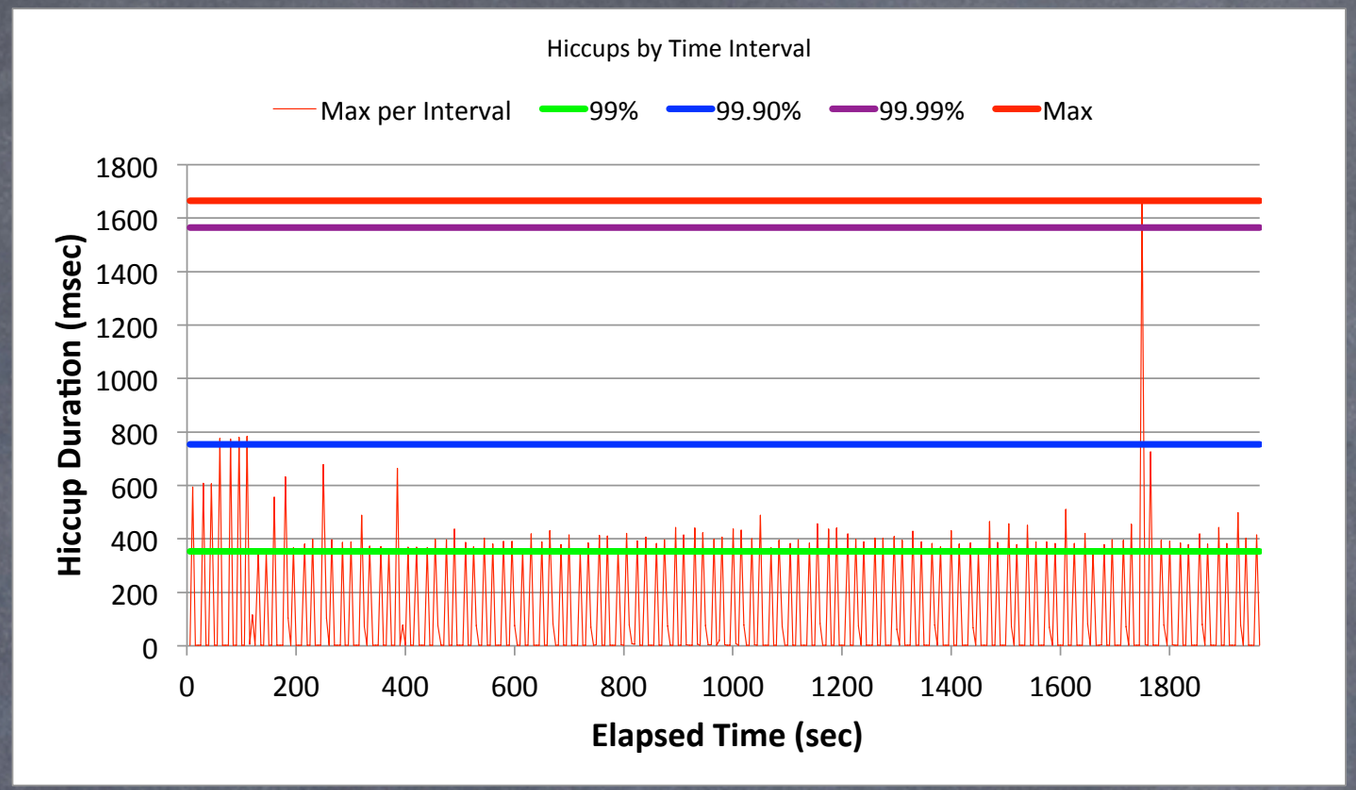


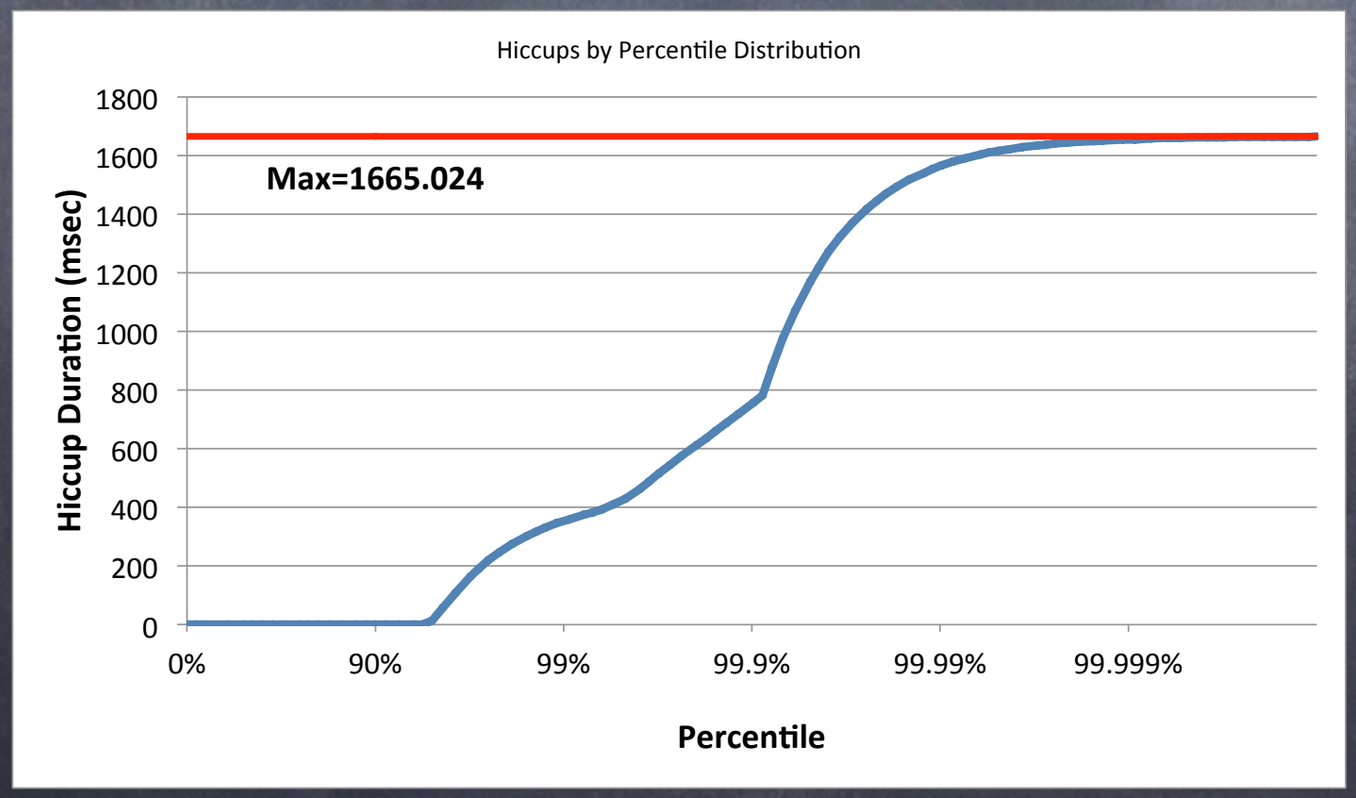
A telco App with a bit of a "problem"

AZUL SYSTEMS

Discontinuities in Java platform execution - Easy To Measure

We call these "hiccups"



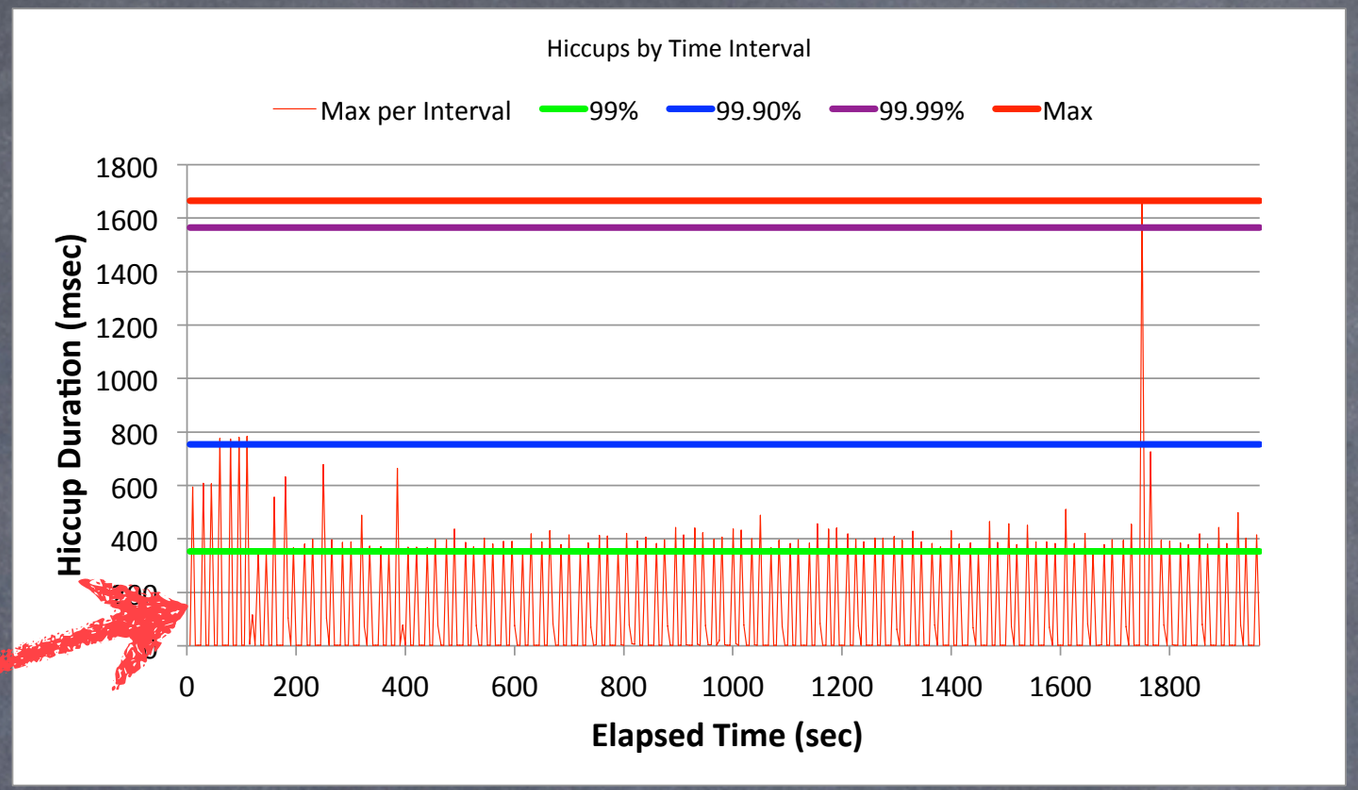


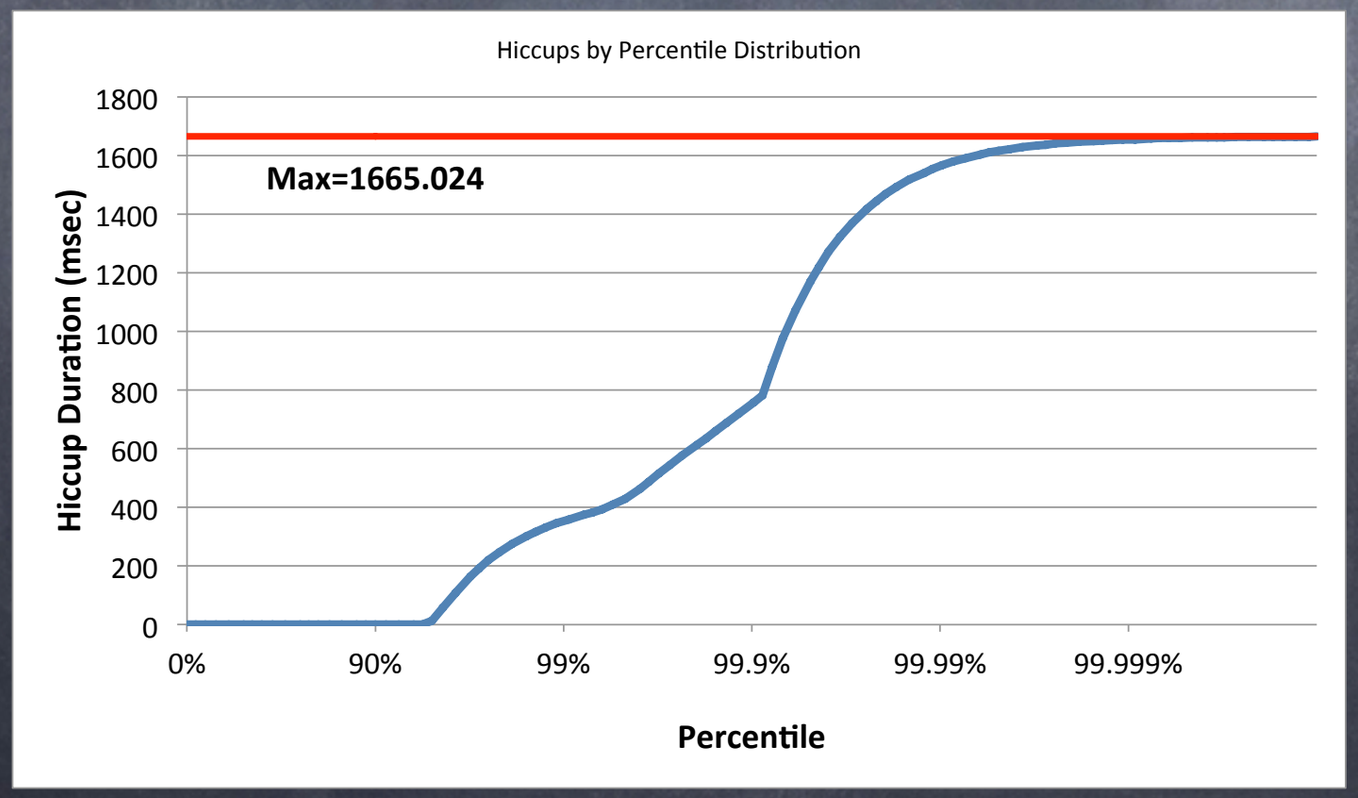
A telco App with a bit of a "problem"

AZUL SYSTEMS

Discontinuities in Java platform execution - Easy To Measure

We call these "hiccups"





A telco App with a bit of a "problem"

AZUL SYSTEMS

Fun with jHiccup

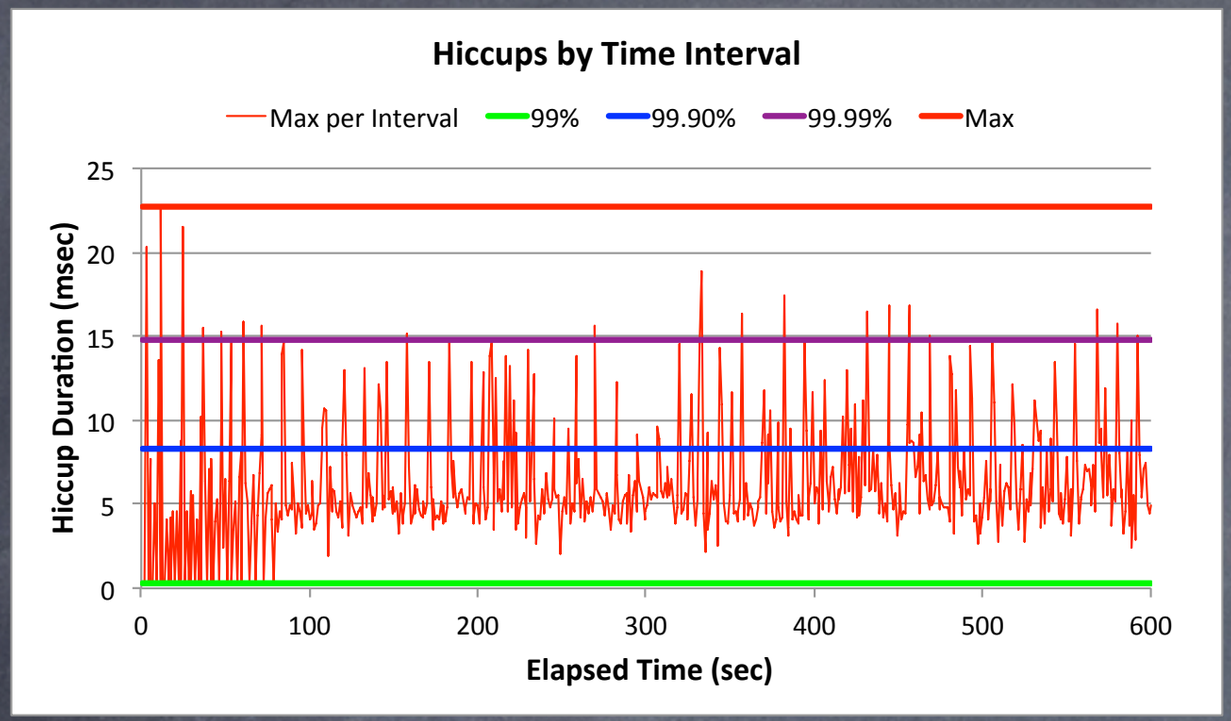
Charles Nutter @headius 20 Jan

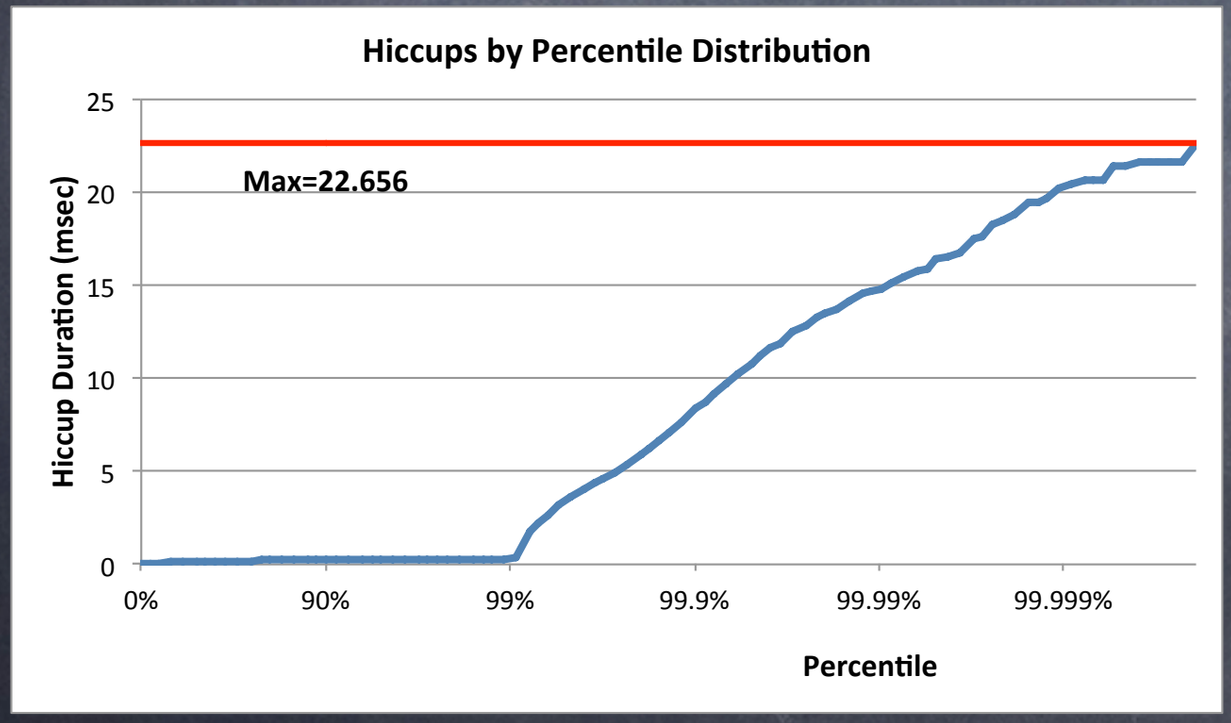
jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)

Retweeted by Gil Tene

AZUL SYSTEMS

Oracle HotSpot (pure newgen)

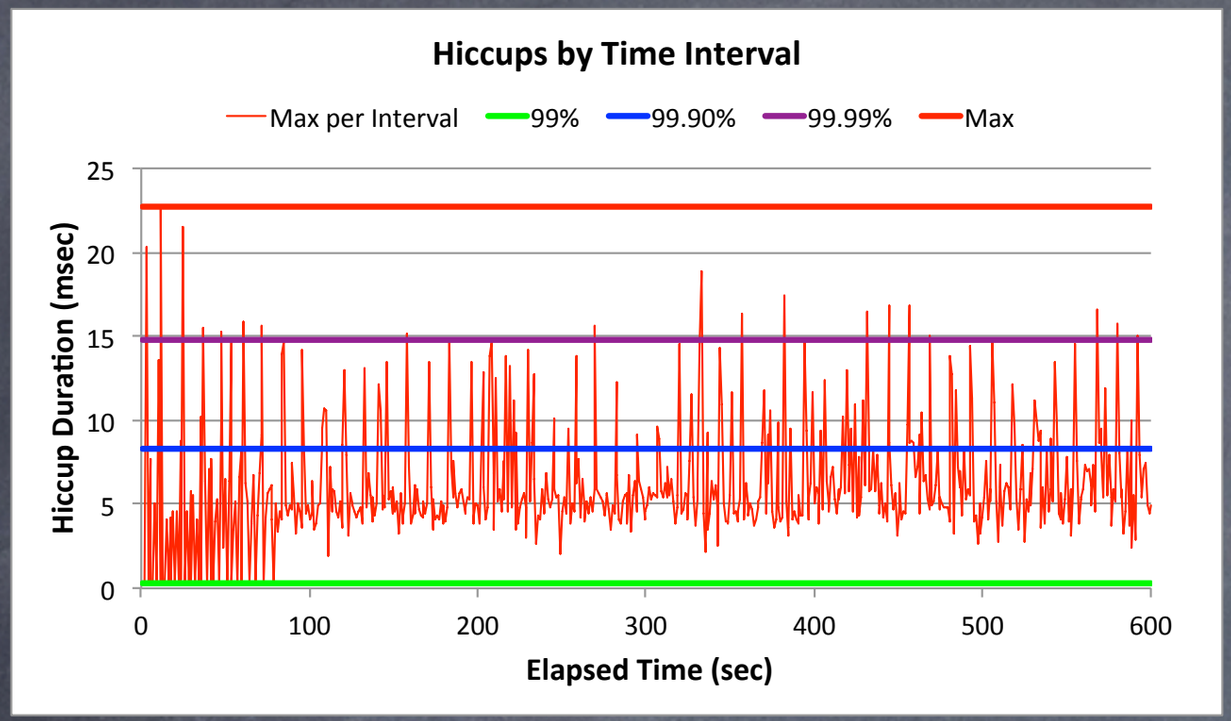


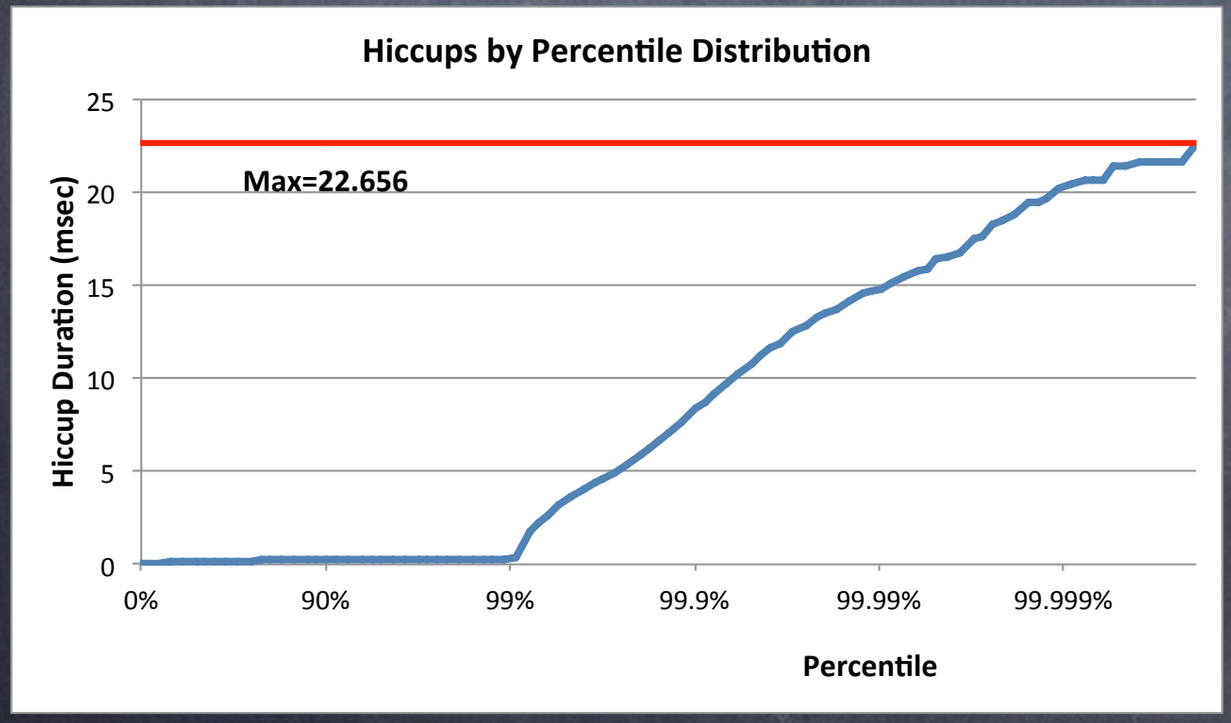


Low latency trading application

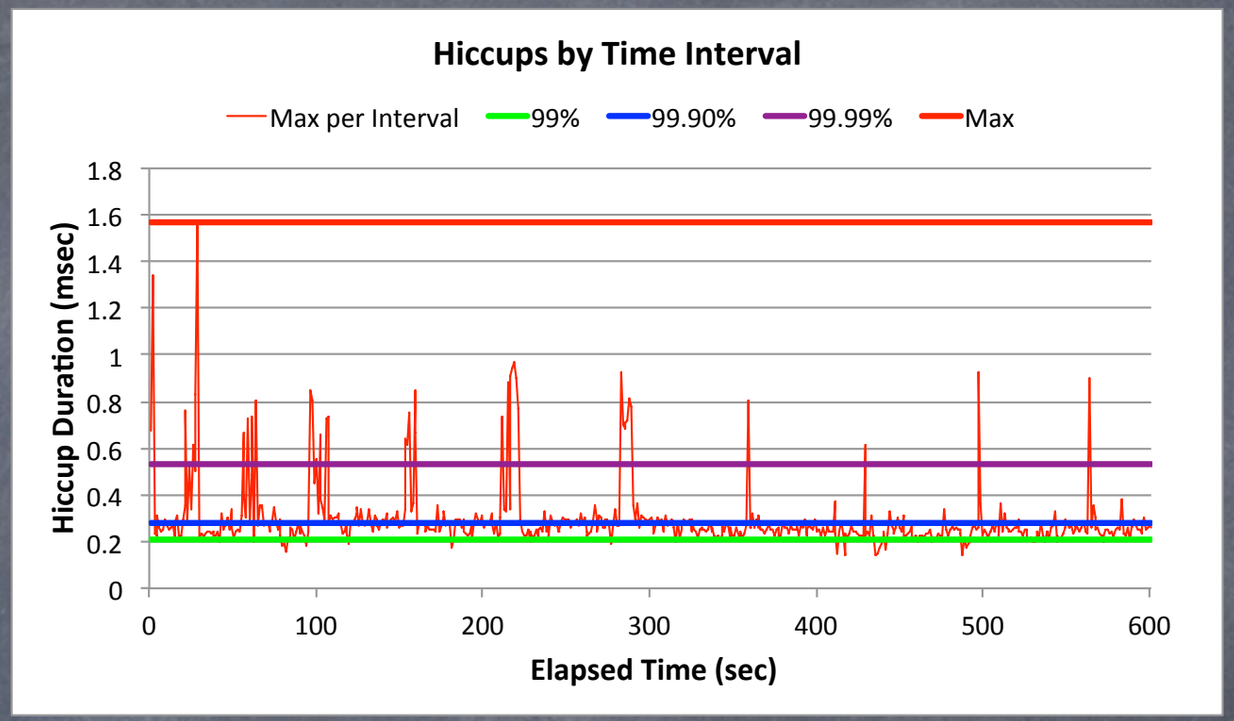
AZUL SYSTEMS

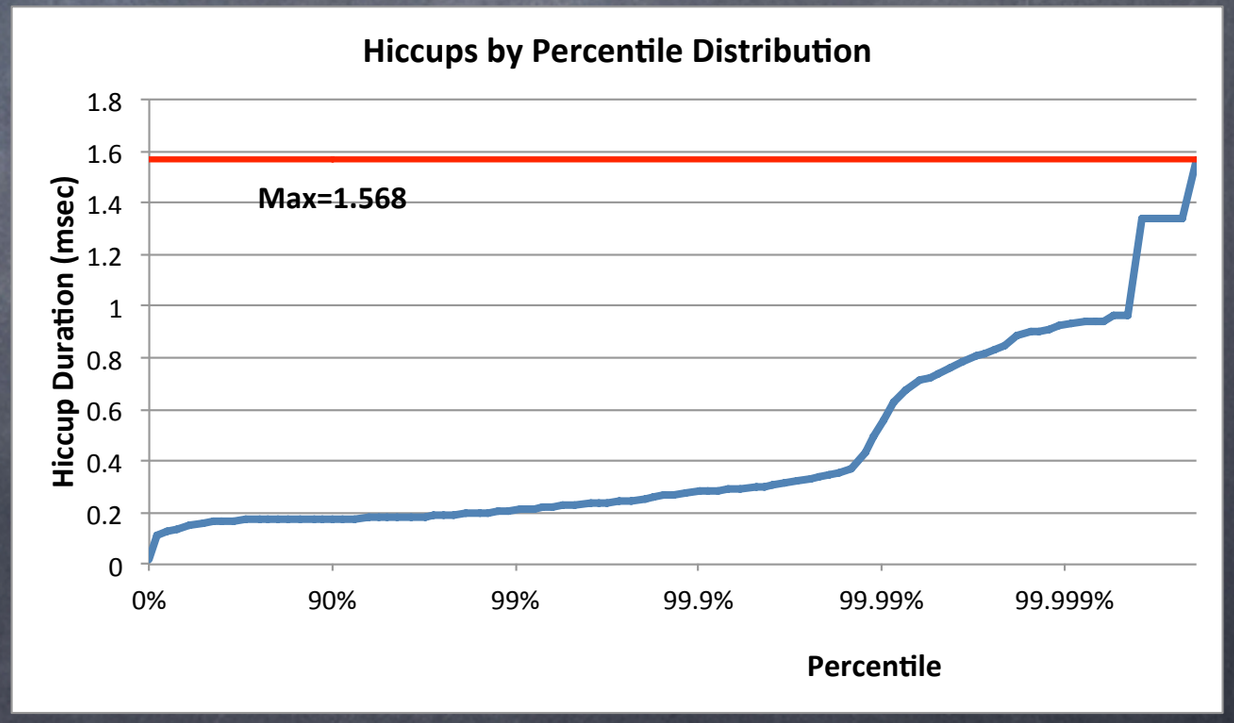
Oracle HotSpot (pure newgen)





Zing

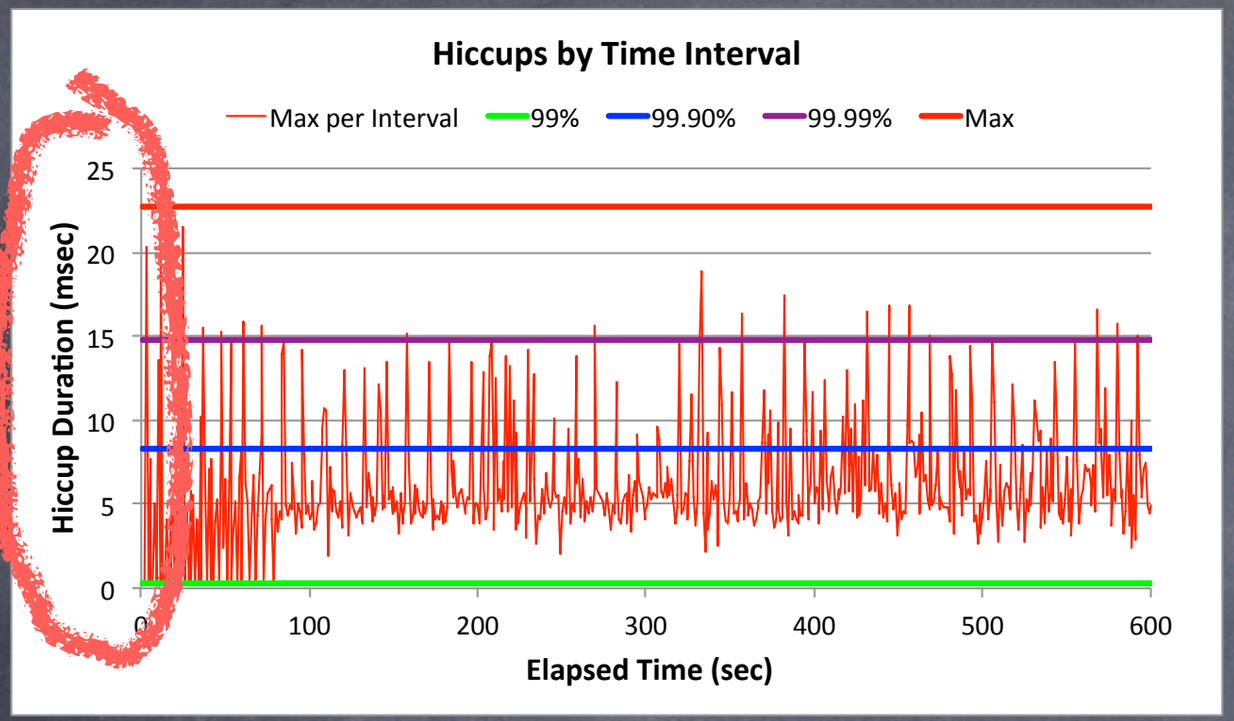


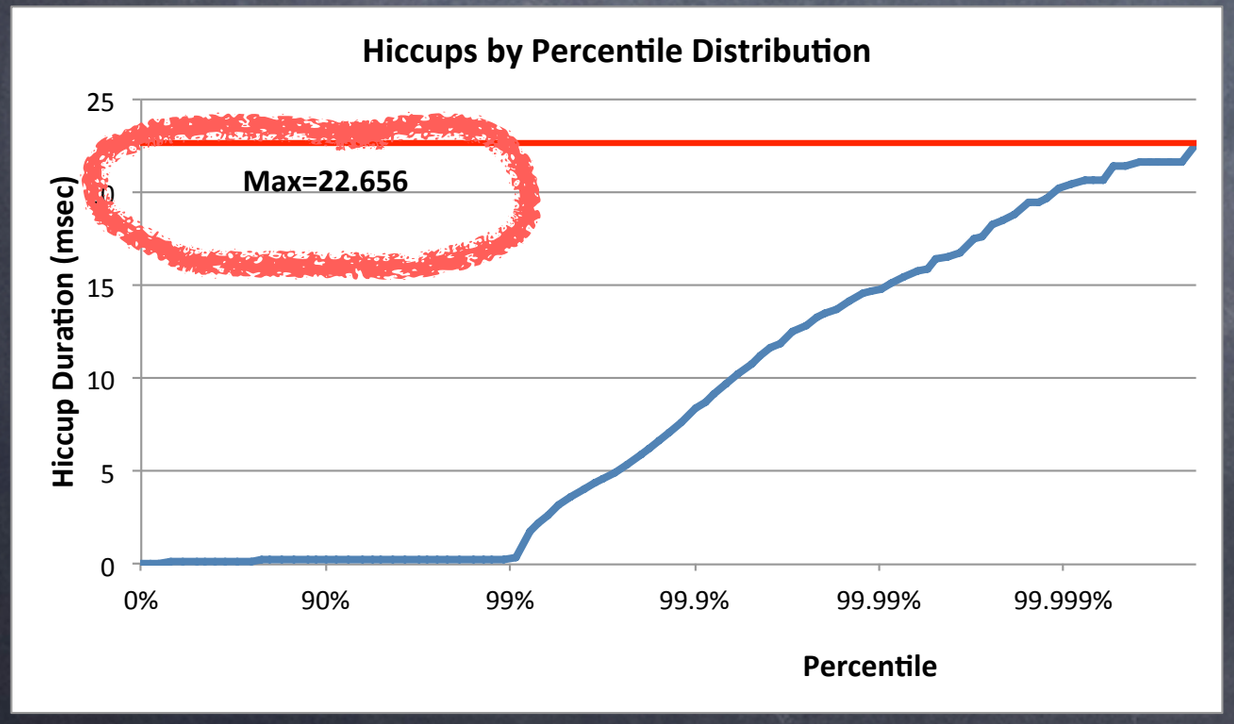


Low latency trading application

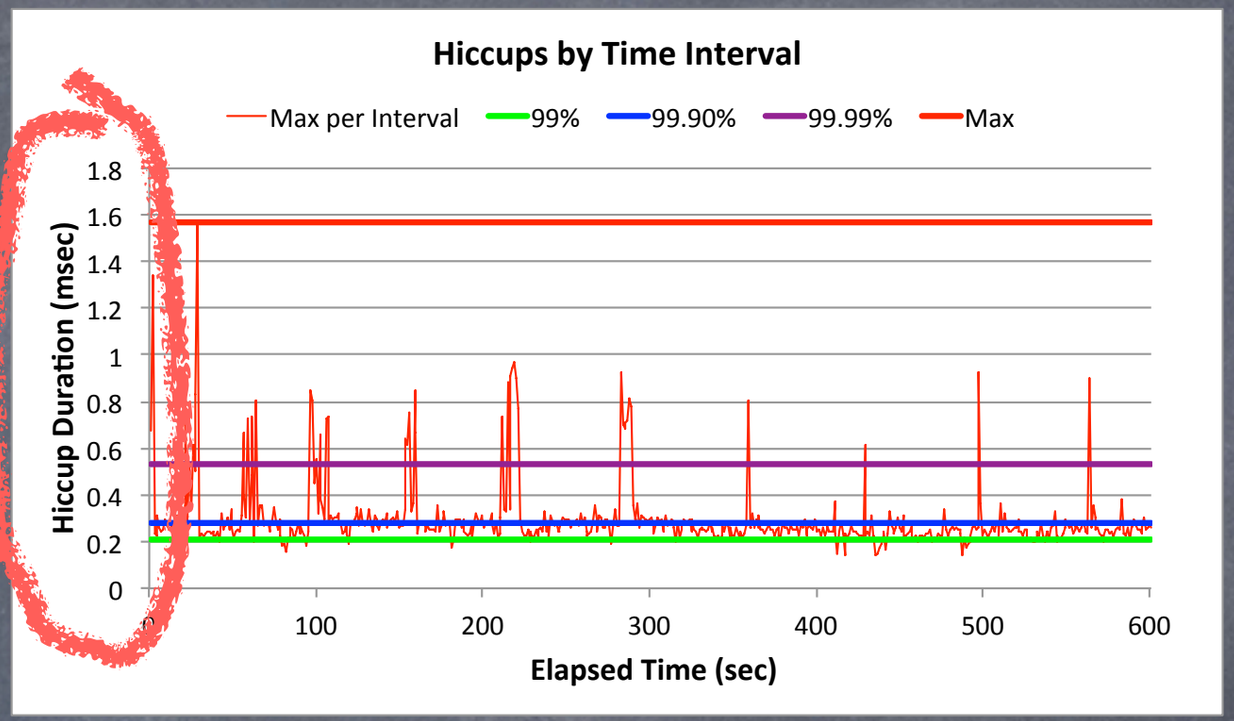
AZUL SYSTEMS

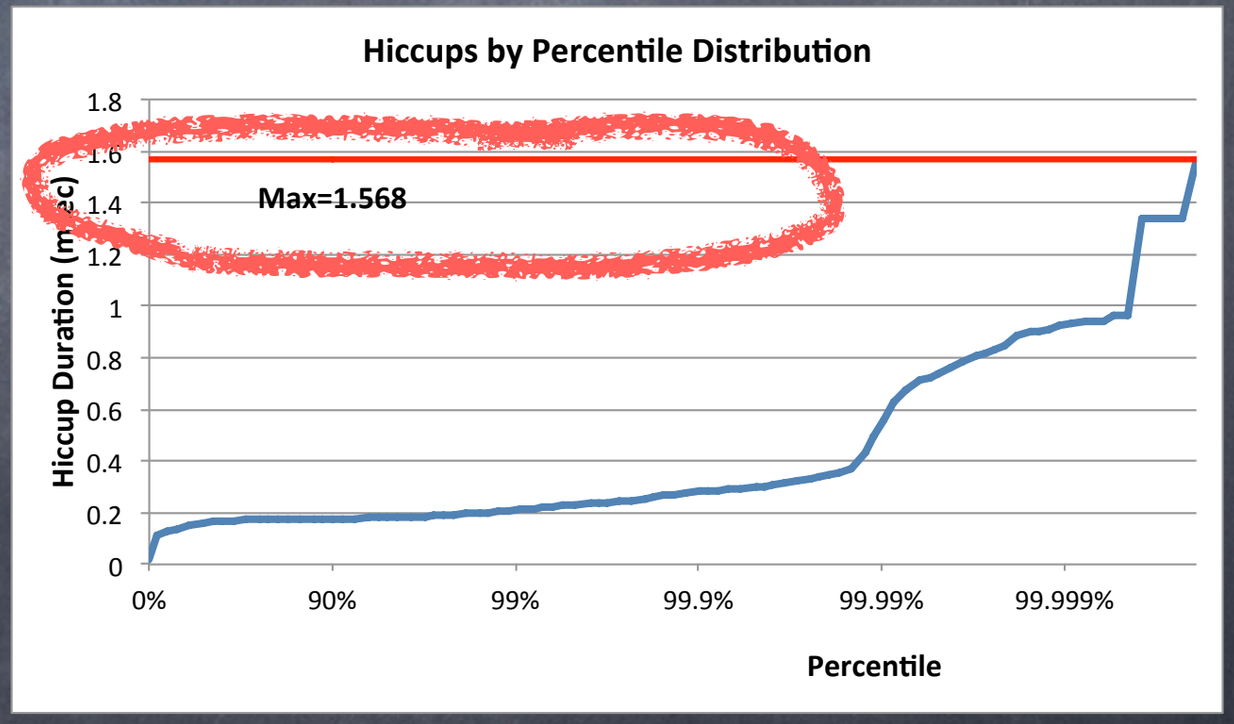
Oracle HotSpot (pure newgen)





Zing

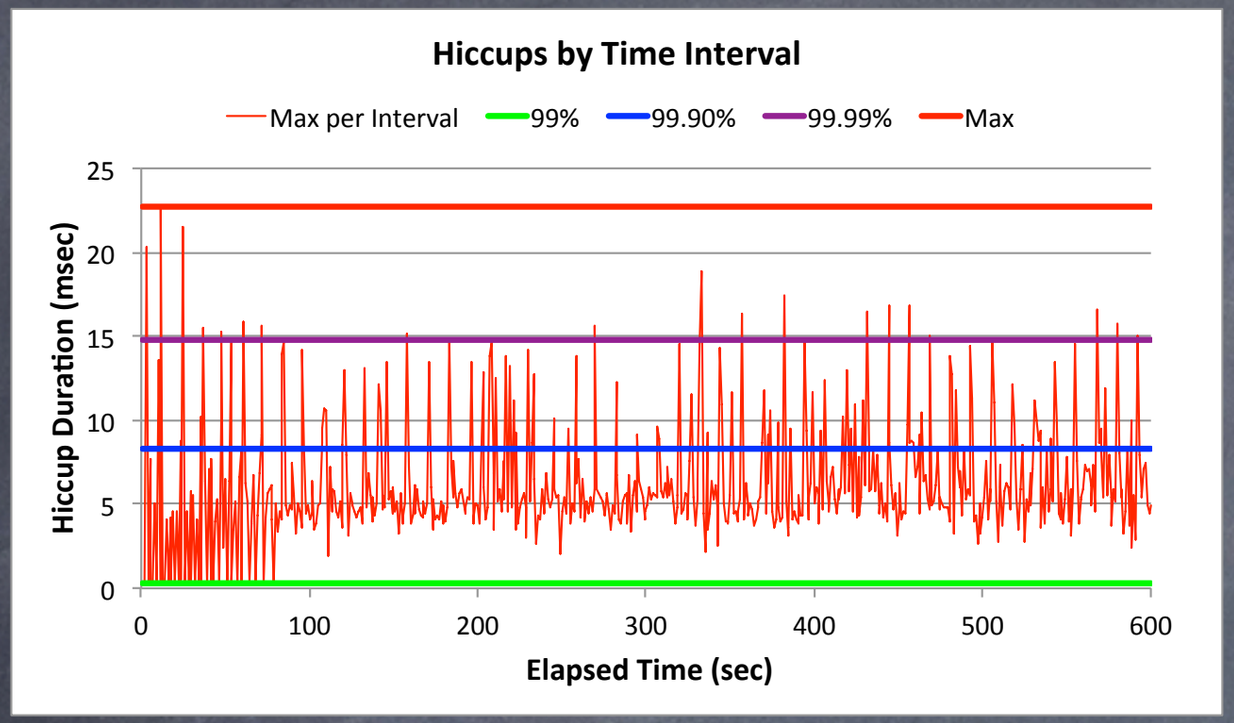


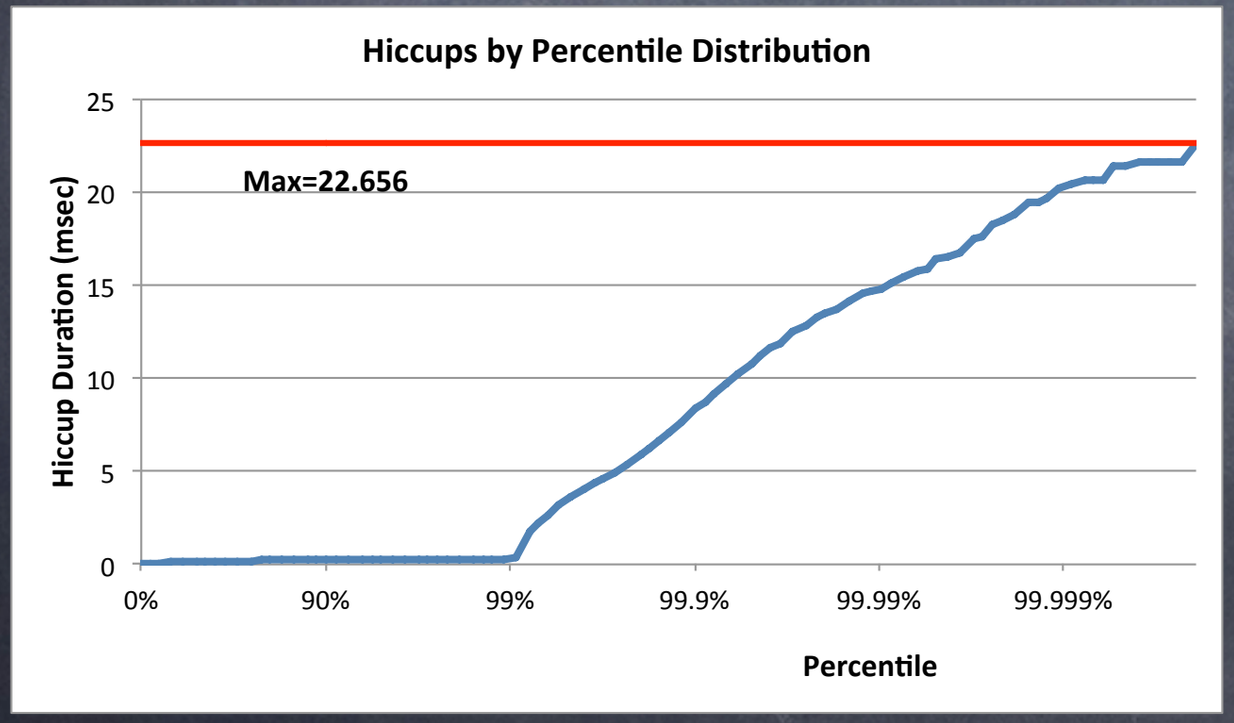


Low latency trading application

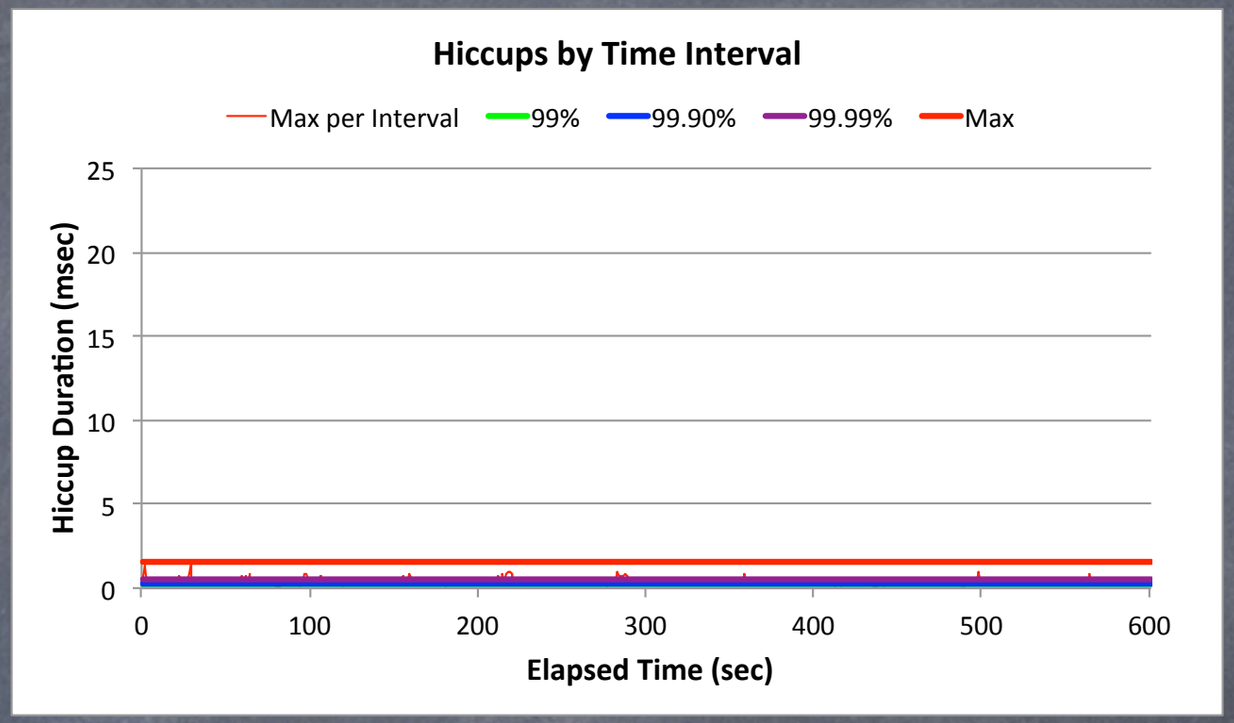
AZUL SYSTEMS

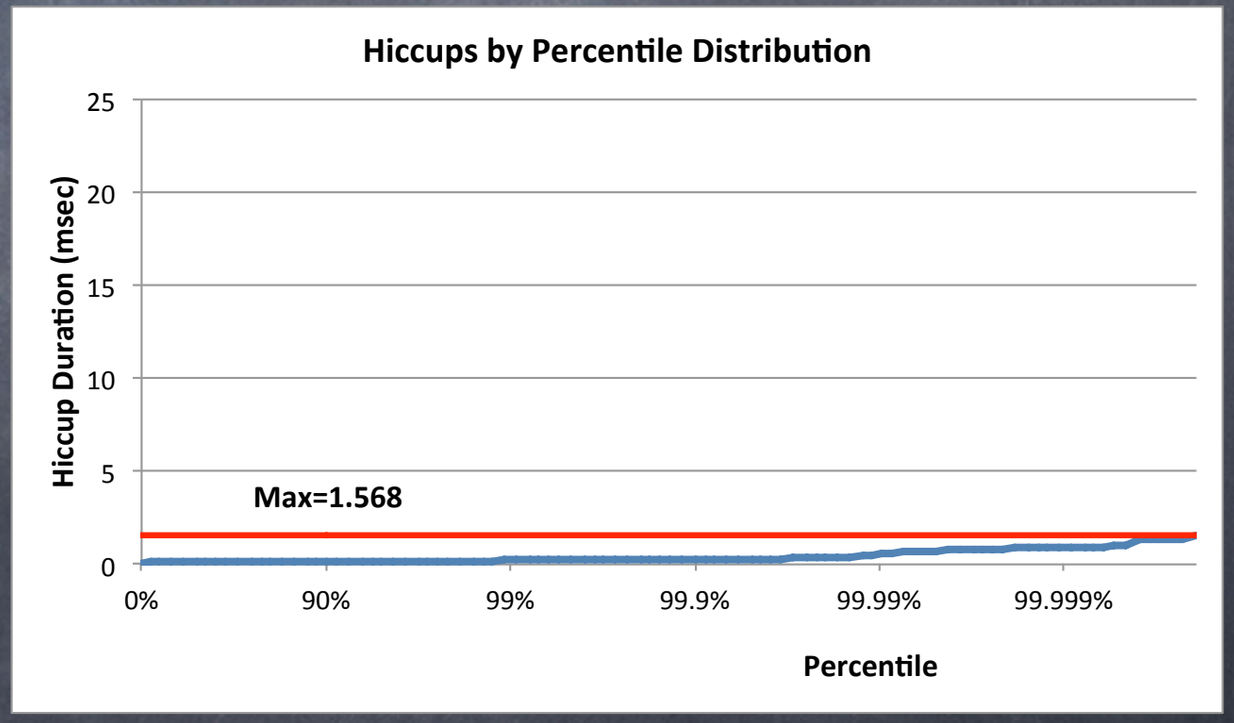
Oracle HotSpot (pure newgen)





Zing





Low latency - Drawn to scale

AZUL SYSTEMS

It's not just for Low Latency

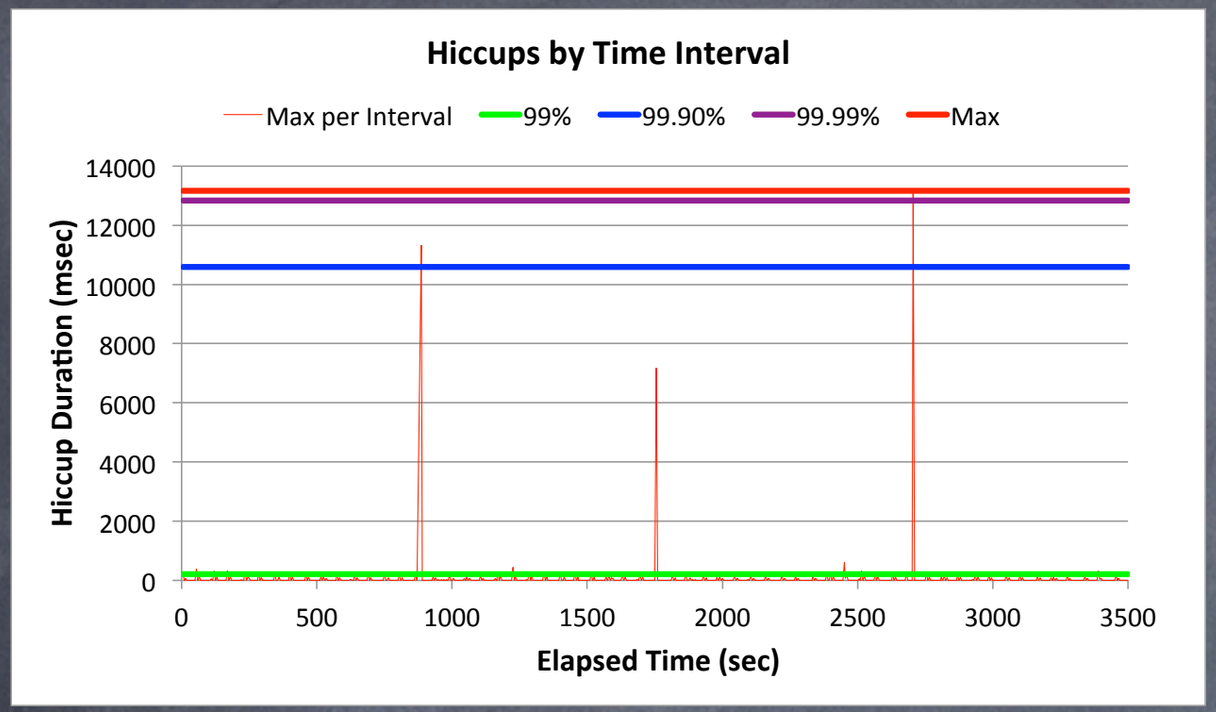
AZUL SYSTEMS

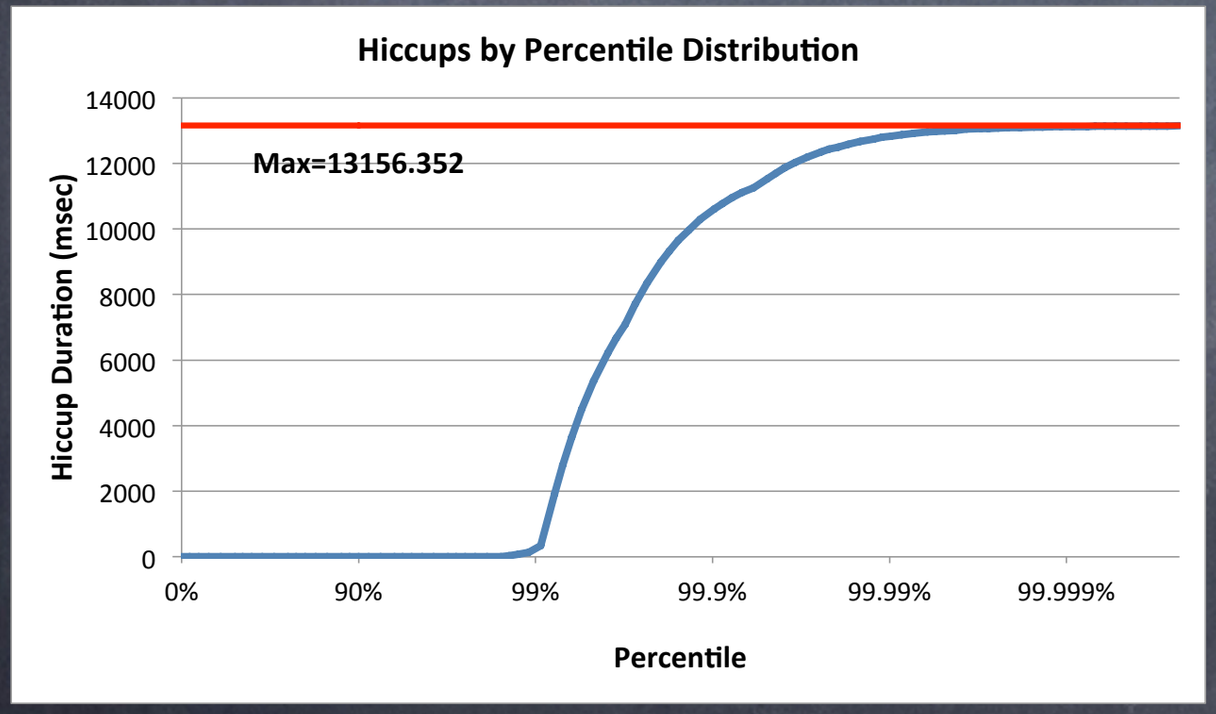
It's not just for
Low Latency

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

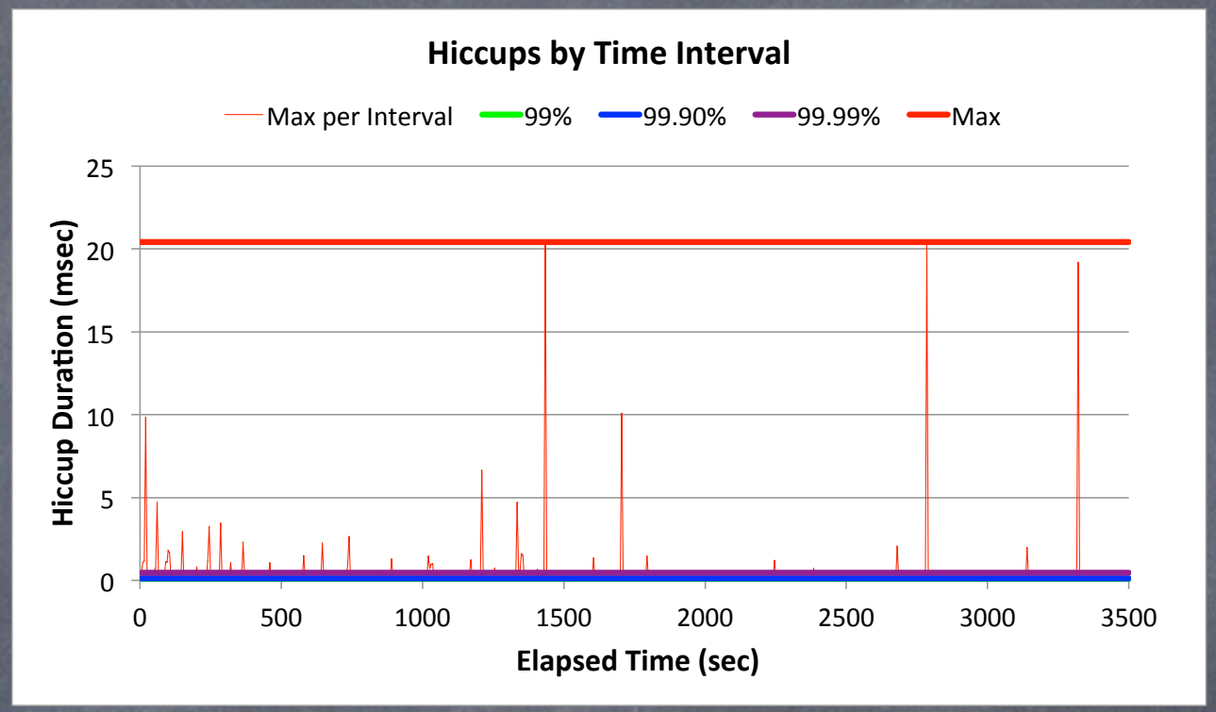
AZUL SYSTEMS

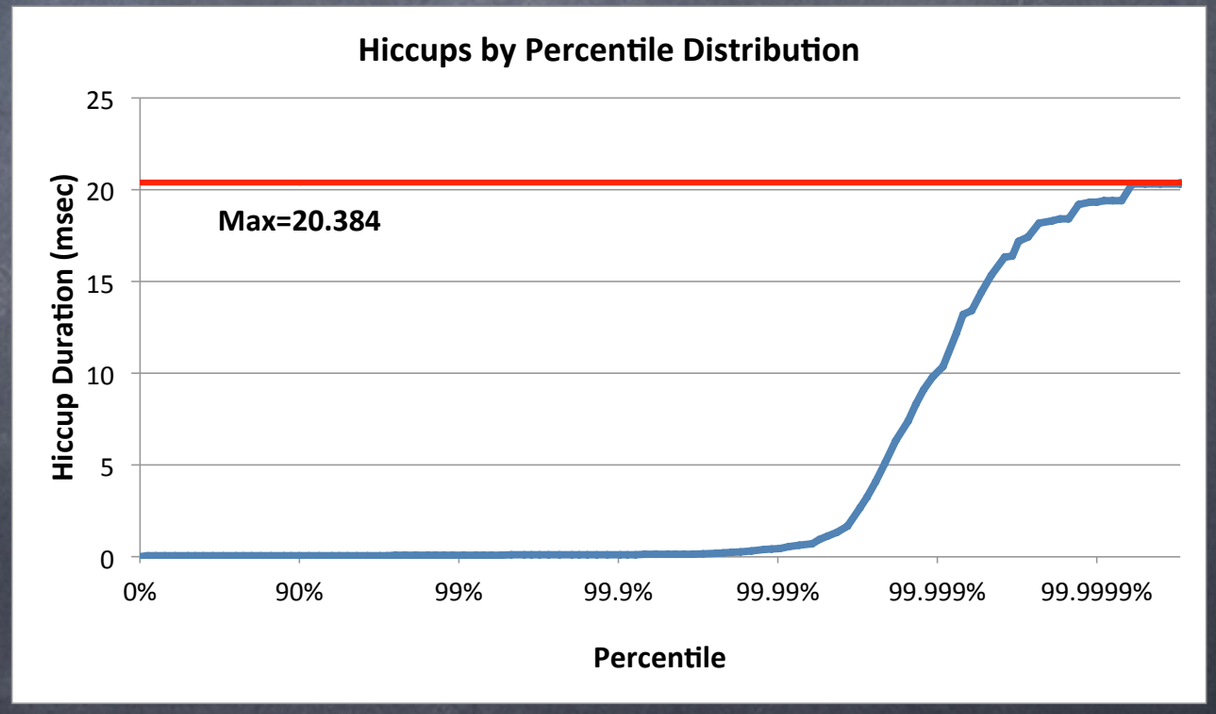
Oracle HotSpot CMS, 1GB in an 8GB heap





Zing 5, 1GB in an 8GB heap

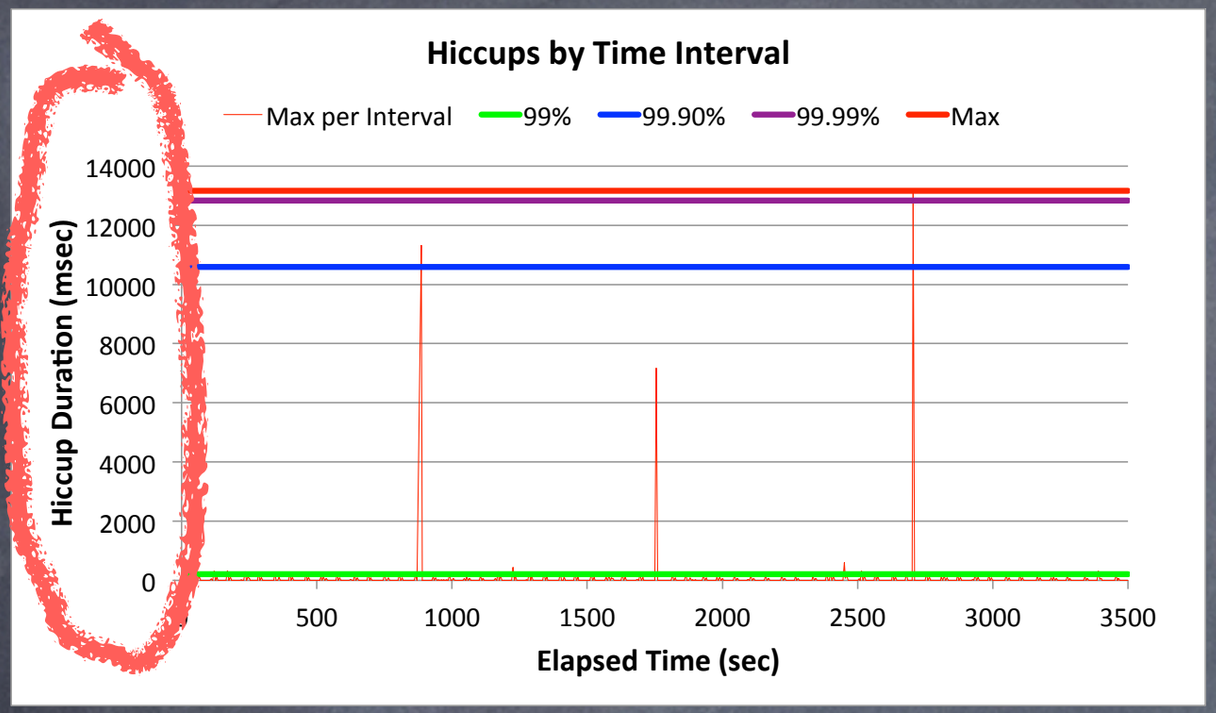


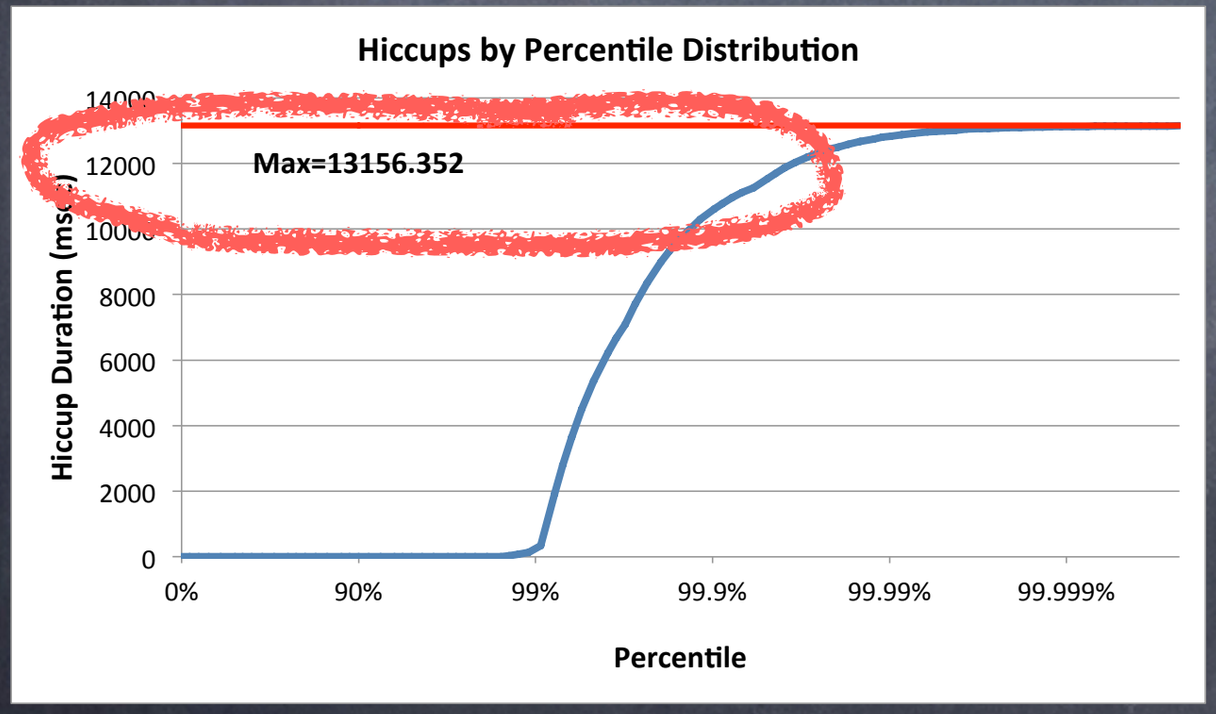


Portal Application, slow Ehcache "churn"

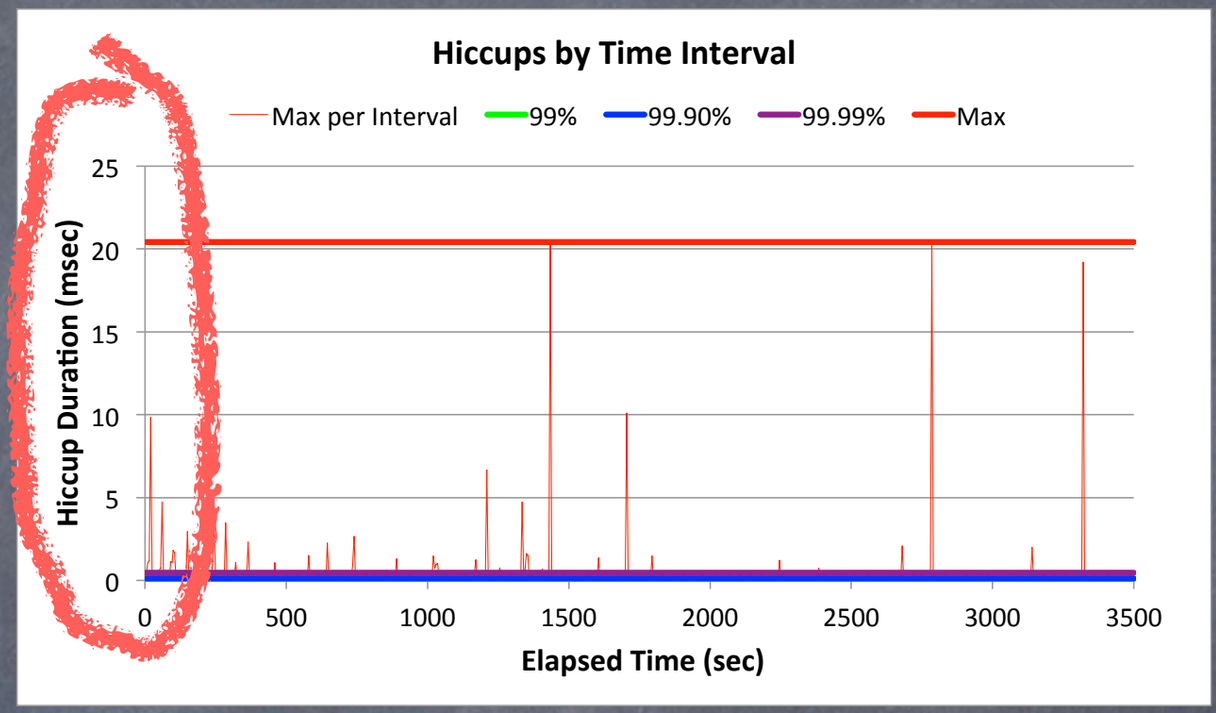
AZUL SYSTEMS

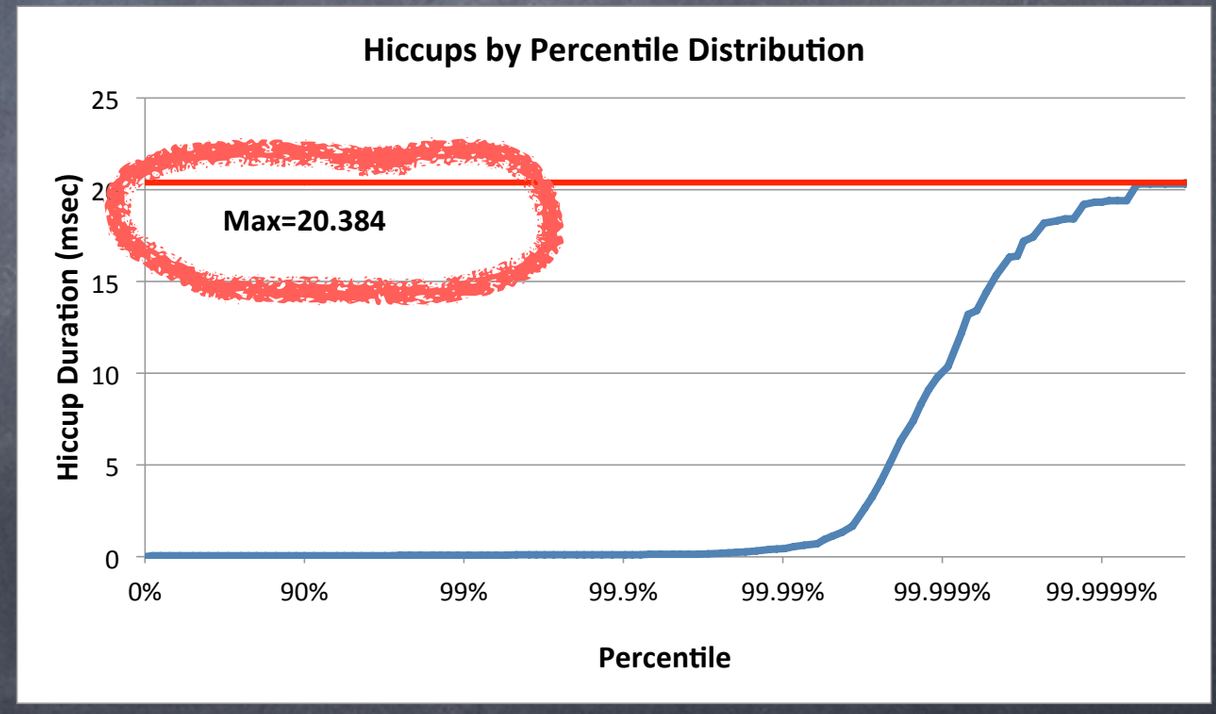
Oracle HotSpot CMS, 1GB in an 8GB heap





Zing 5, 1GB in an 8GB heap

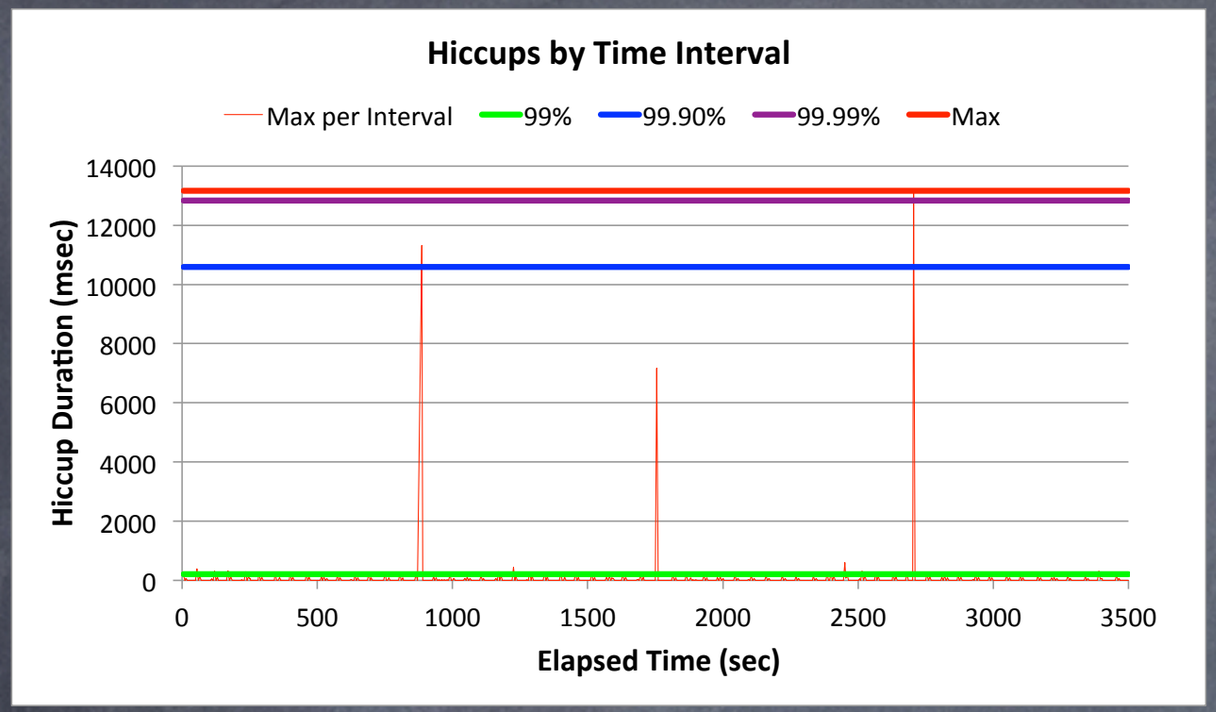


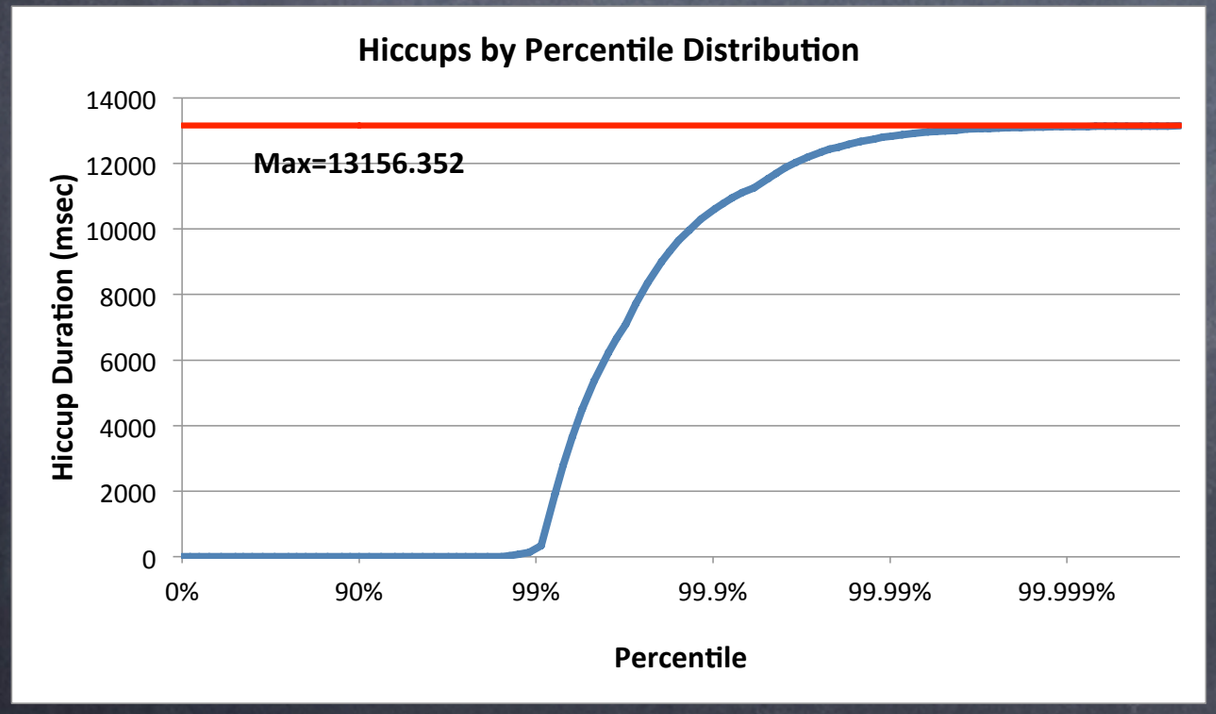


Portal Application, slow Ehcache "churn"

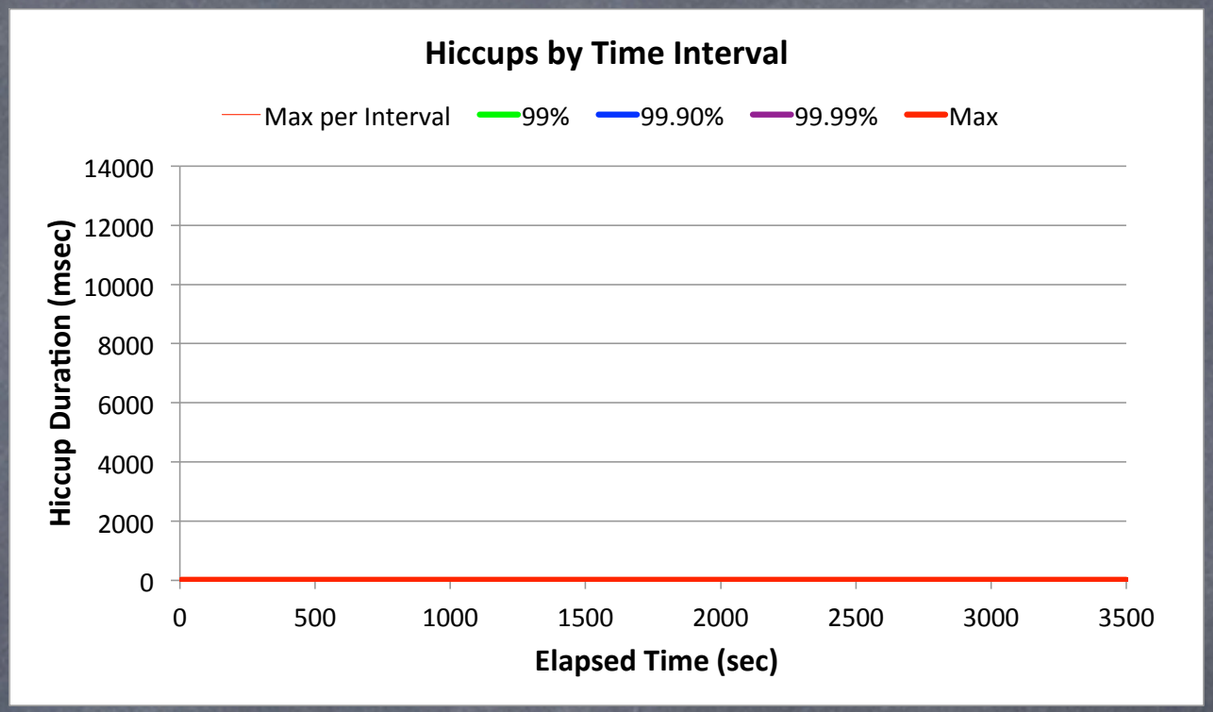
AZUL SYSTEMS

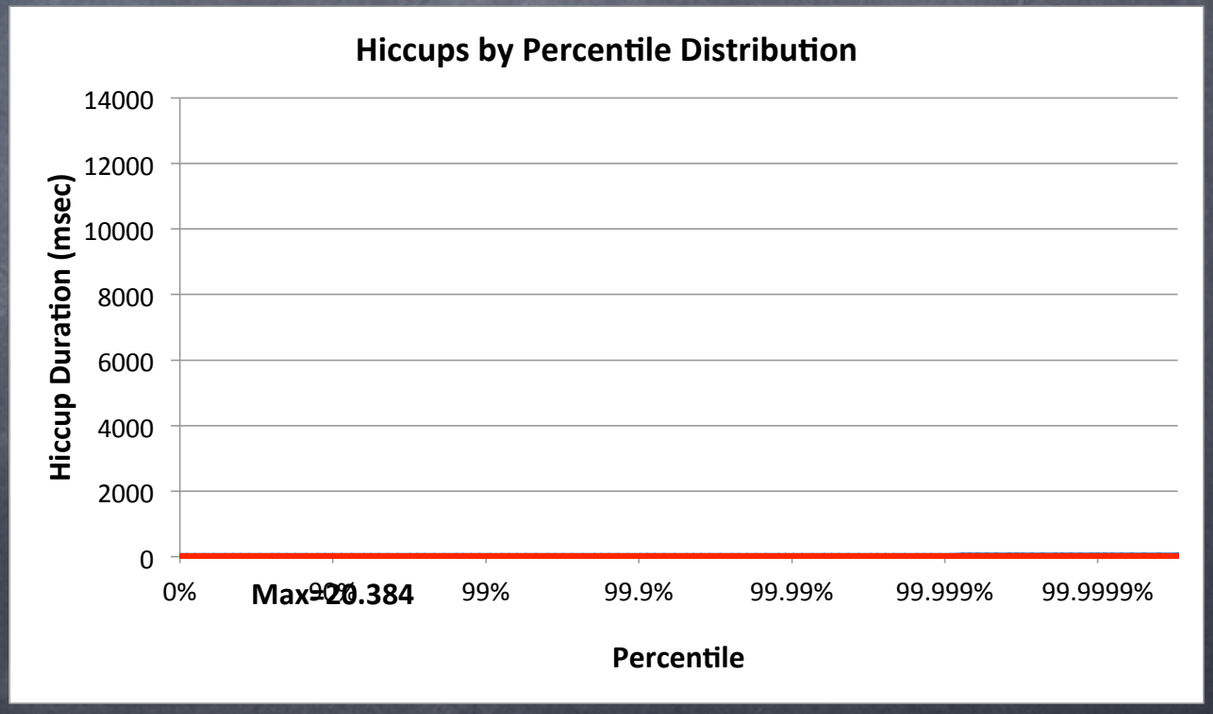
Oracle HotSpot CMS, 1GB in an 8GB heap





Zing 5, 1GB in an 8GB heap





Portal Application - Drawn to scale

AZUL SYSTEMS

Lets not forget about GC tuning

Java GC tuning is "hard"...

Java GC tuning is "hard"...

Examples of actual command line GC tuning parameters:

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 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 ...
```

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 ...
```

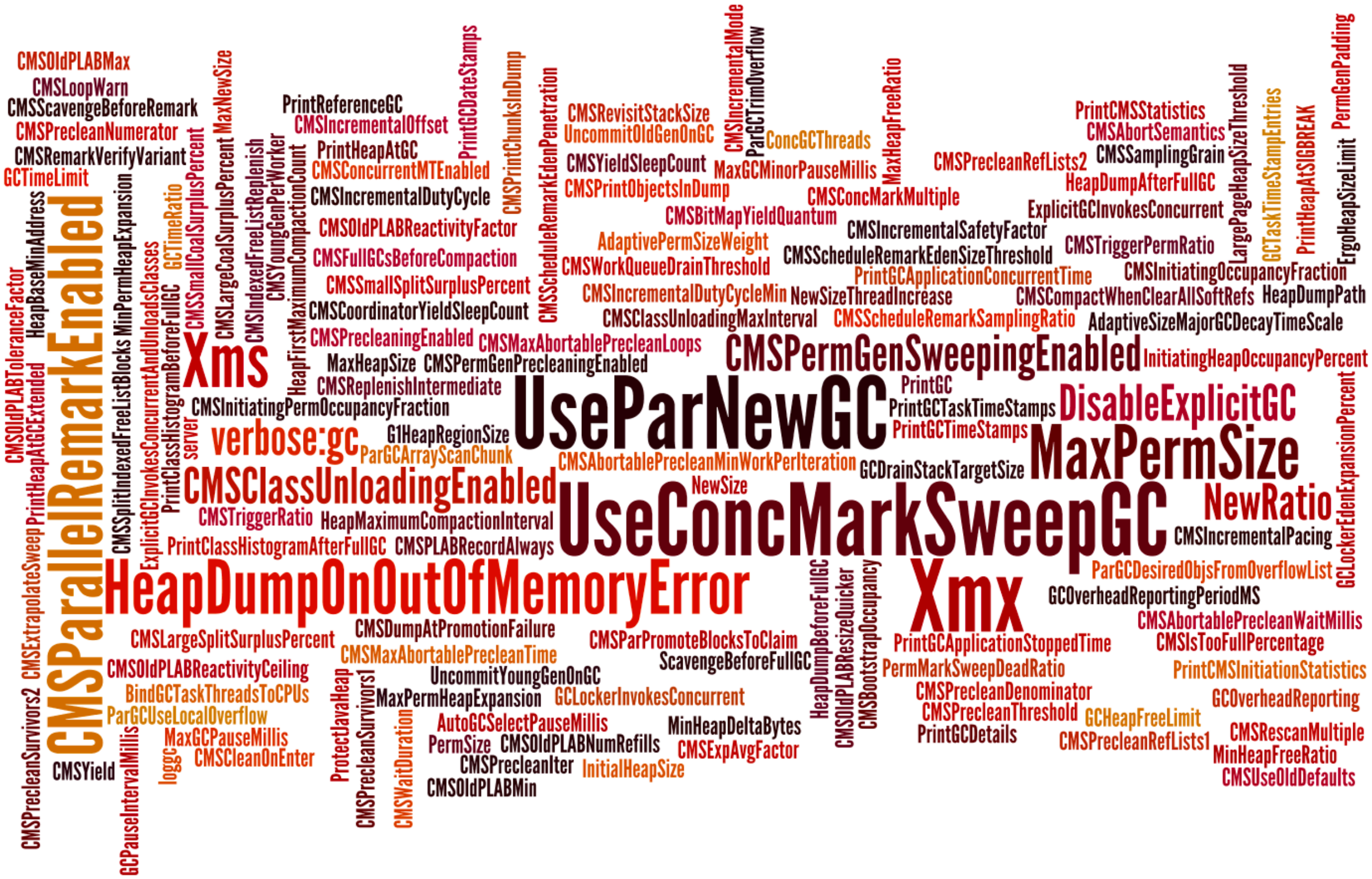
Java GC tuning is "hard"...

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

AZUL SYSTEMS

The complete guide to Zing GC tuning

java -Xmx40g

GC is only the biggest problem...

JVMs make many tradeoffs
often trading speed vs. outliers

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

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

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

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

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

Time To Safepoint (TTSP)
Your new #1 enemy

- (Once GC itself was taken care of)

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, and then released
  - E.g. GC phase shifts, Deoptimization, Class unloading, Thread Dumps, Lock Deflation, etc. etc.

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, and then released
    - 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

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, and then released
    - 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

Time To Safepoint (TTSP)
the most common examples

- Array copies and object clone()

AZUL SYSTEMS

Time To Safepoint (TTSP)
the most common examples

- Array copies and object clone()
- Counted loops

AZUL SYSTEMS

Time To Safepoint (TTSP)
the most common examples

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

AZUL SYSTEMS

Time To Safepoint (TTSP)
the most common examples

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

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...

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...
- At Azul, I walk around with a 0.5msec stick...

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...
- At Azul, I walk around with a 0.5msec stick...
- Zing has a built-in TTSP profiler

AZUL SYSTEMS

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

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.

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?)

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?)

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!)

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

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).

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

AZUL SYSTEMS

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

- GC is no longer a dominant issue, even for outliers

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

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 case** case with "easy" tuning
- < 1 msec **worst case** is very doable

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 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

AZUL SYSTEMS

Q & A

Session # 8206

One-liner Takeaway:

Zing: A cure for the Java hiccups

jHiccup:

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

AZUL SYSTEMS