JVM Mechanics

A peek under the hood

Gil Tene, CTO & co-Founder, Azul Systems

AZUL SYSTEMS®

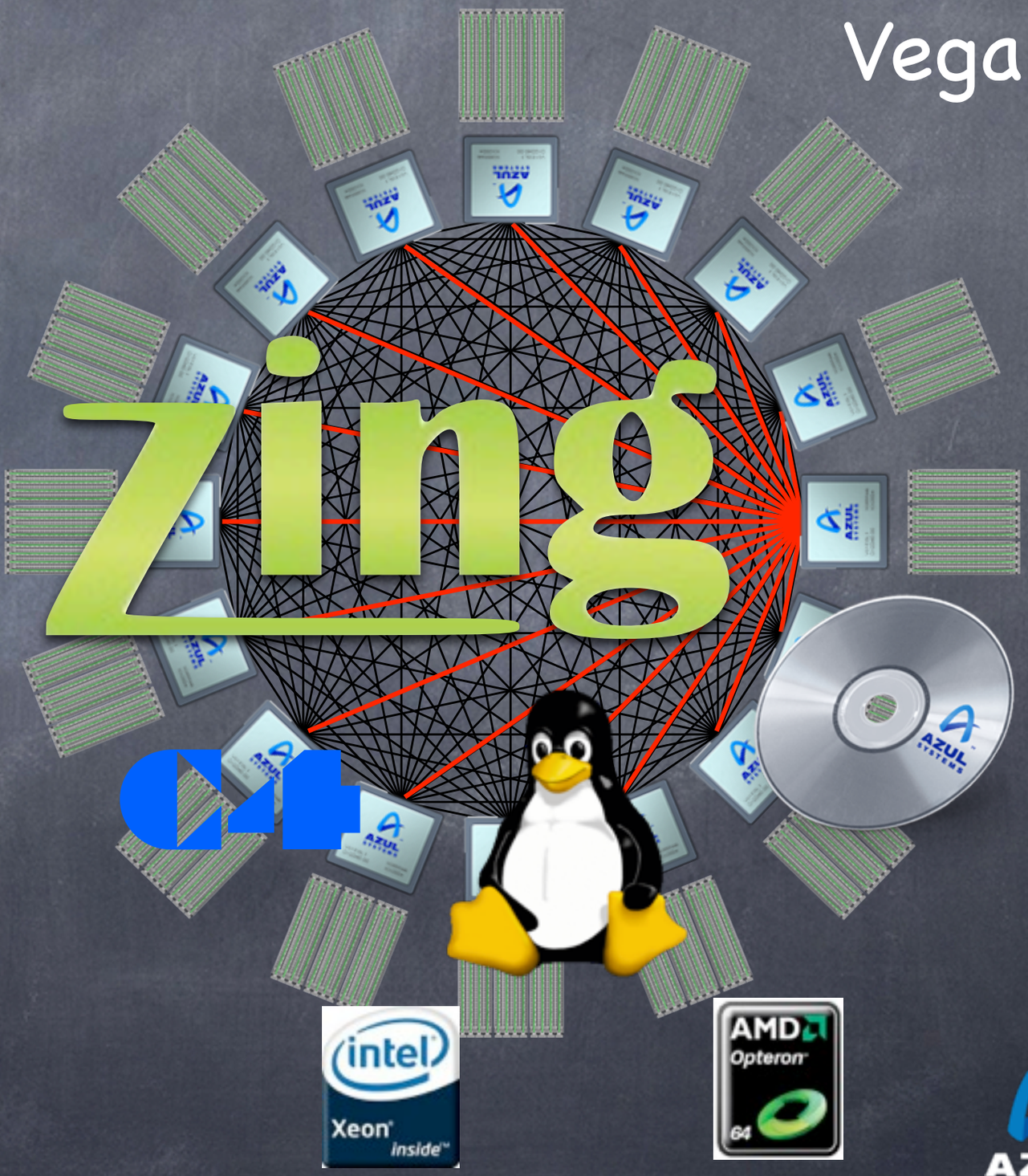
About me: Gil Tene

- co-founder, CTO @Azul Systems
- Working JVMs since 2001, Managed runtimes since 1989
- 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)**
- "Industry firsts" in Garbage collection, elastic memory, Java virtualization, memory scale



High level agenda

- Compiler stuff
- Adaptive behavior stuff
- Ordering stuff
- Garbage Collection stuff
- Some chest beating
- Open discussion

Compiler Stuff

The JIT compilers transforms code

The code actually executed can be very different than the code you write

Some simple compiler tricks

AZUL SYSTEMS®

Code can be reordered…

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reordered to:

```
int doMath(int x, int y, int z) {
   int c = z + x;
   int b = x - y;
   int a = x + y;
   return a + b;
}
```


AZUL SYSTEMS

Dead code can be removed

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   return a + b;
}
```


AZUL SYSTEMS

Values can be propagated

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
   return x + y + x - y;
}
```


AZUL SYSTEMS

Math can be simplified

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
   return x + x;
}
```


AZUL SYSTEMS

So why does this matter

- Keep your code "readable"

```
largestValueLog = Math.log(largestValueWithSingleUnitResolution);
magnitude = (int) Math.ceil(largestValueLog/Math.log(2.0));
subBucketMagnitude = (magnitude > 1) ? magnitude : 1;
subBucketCount = (int) Math.pow(2, subBucketMagnitude);
subBucketMask = subBucketCount - 1;
```

- Hard enough to follow as it is
- No value in "optimizing" human-readable meaning away
- Compiled code will end up the same anyway

AZUL SYSTEMS

Some more compiler tricks

Reads can be cached

```
int distanceRatio(Object a) {
  int distanceTo = a.getX() - start;
  int distanceAfter = end - a.getX();
  return distanceTo/distanceAfter;
}
```

Is the same as

```
int distanceRatio(Object a) {
  int x = a.getX();
  int distanceTo = x - start;
  int distanceAfter = end - x;
  return distanceTo/distanceAfter;
}
```


AZUL SYSTEMS

Reads can be cached

```
void loopUntilFlagSet(Object a) {
  while (!a.flagIsSet()) {
    loopcount++;
  }
}
```

Is the same as:

```
void loopUntilFlagSet(Object a) {
  boolean flagIsSet = a.flagIsSet();
  while (!flagIsSet) {
    loopcount++;
  }
}
```

That's what volatile is for...

AZUL SYSTEMS

Writes can be eliminated

Intermediate values might never be visible

```
void updateDistance(Object a) {
  int distance = 100;
  a.setX(distance);
  a.setX(distance * 2);
  a.setX(distance * 3);
}
```

Is the same as

```
void updateDistance(Object a) {
  a.setX(300);
}
```


AZUL SYSTEMS

Writes can be eliminated

Intermediate values might never be visible

```
void updateDistance(Object a) {
   a. setVisibleValue(0);
   for (int i = 0; i < 1000000; i++) {
      a.setInternalValue(i);
   }
   a.setVisibleValue(a.getInternalValue());
}
```

Is the same as

```
void updateDistance(Object a) {
   a.setInternalValue(1000000);
   a.setVisibleValue(1000000);
}
```

Inlining...

```
public class Thing {
   private int x;
   public final int getX() { return x };
}
...
        myX = thing.getX();
```

Is the same as

```
        Class Thing {
           int x;
        }
        ...
        myX = thing.x;
```


AZUL SYSTEMS

Things JIT compilers can do

..and static compilers can have a hard time with

AZUL SYSTEMS

Class Hierarchy Analysis (CHA)

- Can perform global analysis on currently loaded code
- Deduce stuff about inheritance, method overrides, etc.
- Can make optimization decisions based on assumptions
- Re-evaluate assumptions when loading new classes
- Throw away code that conflicts with assumptions before class loading makes them invalid

Inlining works without "final"

```
public class Thing {
   private int x;
   public int getX() { return x };
}
…
myX = thing.getX();
```

Is the same as

```
Class Thing {
   int x;
}
…
myX = thing.x;
```

As long as there is only one implementer of getX()

AZUL SYSTEMS

Speculative stuff

- The power of the "uncommon trap"
- Being able throw away wrong code is very useful
- Speculatively assuming callee type
  - polymorphic can be "monomorphic" or "megamorphic"
  - Can make virtual calls static even without CHA
  - Can speculatively inline things without CHA
- Speculatively assuming branch behavior
  - We've only ever seen this thing go one way, so....

Adaptive compilation make cleaner code practical

- Reduces need to trade off clean design against speed
- E.g. "final" should be used on methods only when you want to prohibit extension, overriding. Has no effect on speed.
- E.g. branching can be written "naturally"

AZUL SYSTEMS

Interesting side effects

Adaptive compilation is... adaptive

- Measuring actual behavior is harder
- Micro-benchmarking is an art
- JITs are moving target
- "Warmup" techniques can often fail

Warmup problems

- Common Example:
  - Trading system wants to have the first trade be fast
  - So run 20,000 "fake" messages through the system to warm up
    - let JIT compilers optimize code, and deopt before actual trades
- What really happens
  - Code is written to do different things "if this is a fake message"
  - e.g. "Don't send to the exchange if this is a fake message"
  - JITs optimize for fake path, including speculatively assuming "fake"
  - First real message through deopts...

AZUL SYSTEMS

Warmup tips...
(to get "first real thing" to be fast)

- System should not distinguish between "real" and "fake"
  - Make that an external concern
  - Avoid all conditional code based on "fake"
  - Avoid all class-specific calls based on "fake" ("object oriented ifs").
- Use "real" input sources and output targets instead
  - Have system output to a sink target that knows it is "fake"
  - Make output decisions based on data, not code
  - E.g. array of sink targets, with array index computed from message payload

AZUL SYSTEMS

Ordering stuff

Ordering of operations

- Within a thread, it's trivial : "happens before"
- Across threads?
- News flash: CPU memory ordering doesn't matter
- There is a much bigger culprit at play: Compilers
  - The code will be reordered before your cpu ever sees it
  - The only ordering rules that matter are the JMM ones.
- Intuitive read:
  - "Happens before holds within threads"
  - Things can move into, but not out of synchronized blocks
  - Volatile stuff is a bit more tricky...

Ordering of operations

Required barriers	*2nd operation*			
1st operation	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

Source: http://g.oswego.edu/dl/jmm/cookbook.html

AZUL SYSTEMS

Garbage Collection Stuff

AZUL SYSTEMS®

Most of what People seem to "know" about Garbage Collection is wrong

- In many cases, it's much better than you may think
  - GC is extremely efficient. Much more so that malloc()
  - Dead objects cost nothing to collect
  - GC will find <u>all</u> the dead objects (including cyclic graphs)
  - ...
- In many cases, it's much worse than you may think
  - Yes, it really does stop for ~1 sec per live GB (in most JVMs).
  - No, GC does not mean you can't have memory leaks
  - No, those pauses you eliminated from your 20 minute test are not gone
  - ...

Generational Collection

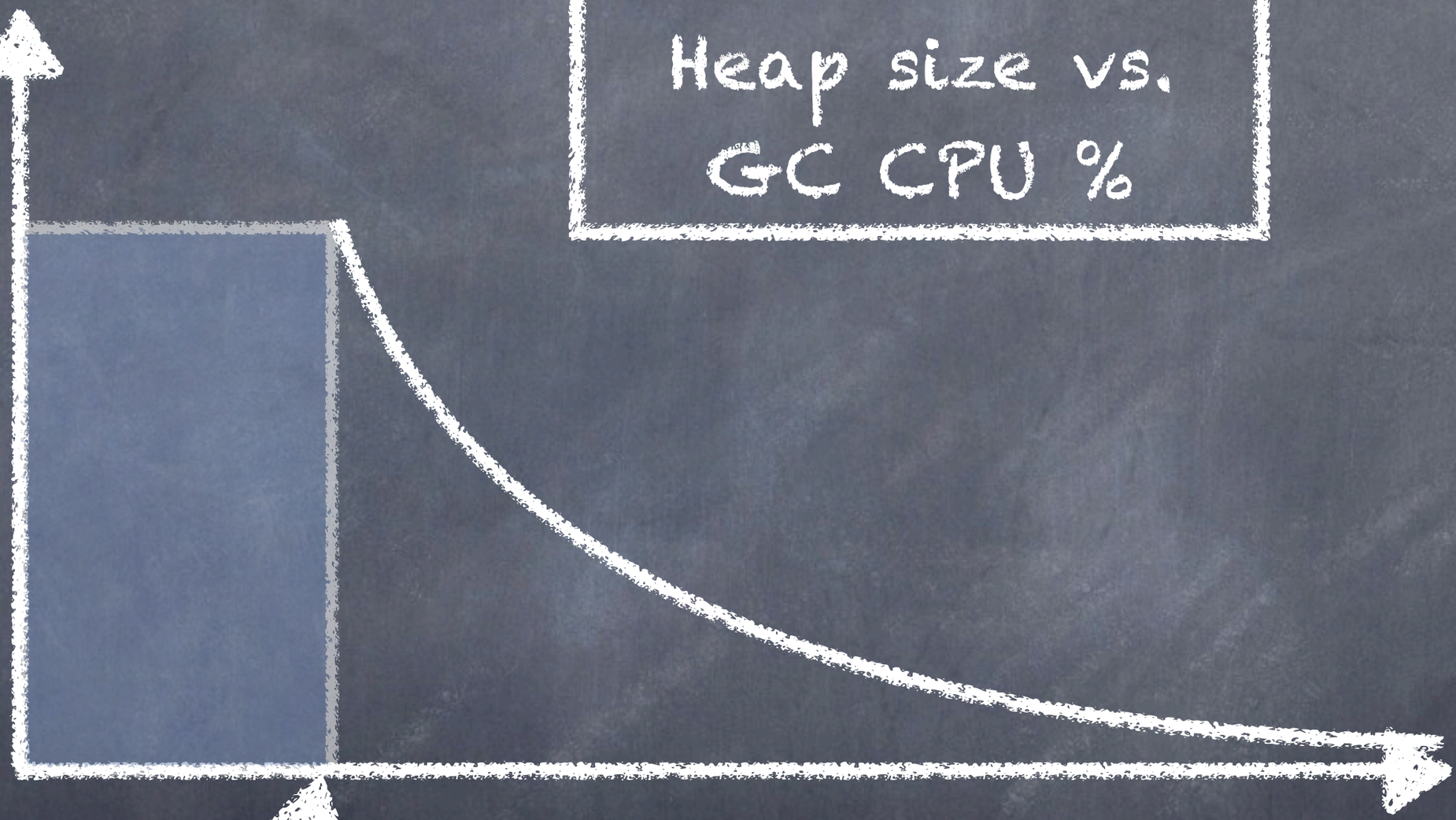
- Weak Generational Hypothesis; "most objects die young"
- Focus collection efforts on young generation:
  - Use a moving collector: work is linear to the live set
  - The live set in the young generation is a small % of the space
  - Promote objects that live long enough to older generations
- Only collect older generations as they fill up
  - "Generational filter" reduces rate of allocation into older generations
- Tends to be (order of magnitude) more efficient
  - Great way to keep up with high allocation rate
  - Practical necessity for keeping up with processor throughput

GC Efficiency:
Empty memory vs. CPU

CPU%

100%

Heap size vs.
GC CPU %



Live set

Heap size

Two Intuitive limits

- If we had exactly 1 byte of empty memory at all times, the collector would have to work "very hard", and GC would take 100% of the CPU time
- If we had infinite empty memory, we would never have to collect, and GC would take 0% of the CPU time
- GC CPU % will follow a rough 1/x curve between these two limit points, dropping as the amount of memory increases.

Empty memory needs
(empty memory == CPU power)

- The amount of empty memory in the heap is the dominant factor controlling the amount of GC work
- For both Copy and Mark/Compact collectors, the amount of work per cycle is linear to live set
- The amount of memory recovered per cycle is equal to the amount of unused memory (heap size) - (live set)
- The collector has to perform a GC cycle when the empty memory runs out
- A Copy or Mark/Compact collector's efficiency doubles with every doubling of the empty memory

What empty memory controls

- Empty memory controls efficiency (amount of collector work needed per amount of application work performed)
- Empty memory controls the frequency of pauses (if the collector performs any Stop-the-world operations)
- Empty memory DOES NOT control pause times (only their frequency)
- In Mark/Sweep/Compact collectors that pause for sweeping, more empty memory means less frequent but LARGER pauses

GC and latency:
That pesky stop-the-world thing

Delaying the inevitable

- Some form of copying/compaction is inevitable in practice
  - And compacting anything requires scanning/fixing all references to it
- Delay tactics focus on getting "easy empty space" first
  - This is the focus for the vast majority of GC tuning
- Most objects die young [Generational]
  - So collect young objects only, as much as possible. Hope for short STW.
  - But eventually, some old dead objects must be reclaimed
- Most old dead space can be reclaimed without moving it
  - [e.g. CMS] track dead space in lists, and reuse it in place
  - But eventually, space gets fragmented, and needs to be moved
- Much of the heap is not "popular" [e.g. G1, "Balanced"]
  - A non popular region will only be pointed to from a small % of the heap
  - So compact non-popular regions in short stop-the-world pauses
  - But eventually, popular objects and regions need to be compacted

Memory use

How many of you use heap sizes of:

- more than ½ GB?
- more than 1 GB?
- more than 2 GB?
- more than 4 GB?
- more than 10 GB?
- more than 20 GB?
- more than 50 GB?

Reality check: servers in 2012

- Retail prices, major web server store (US $, Oct 2012)

24 vCore, 128GB server	≈	$5K
24 vCore, 256GB server	≈	$8K
32 vCore, 384GB server	≈	$14K
48 vCore, 512GB server	≈	$19K
64 vCore, 1TB server	≈	$36K

- Cheap (< $1/GB/Month), and roughly linear to ~1TB
- 10s to 100s of GB/sec of memory bandwidth

AZUL SYSTEMS

The Application Memory Wall
A simple observation:

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

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

How much memory do applications need?

- "640KB ought to be enough for anybody"

  WRONG!

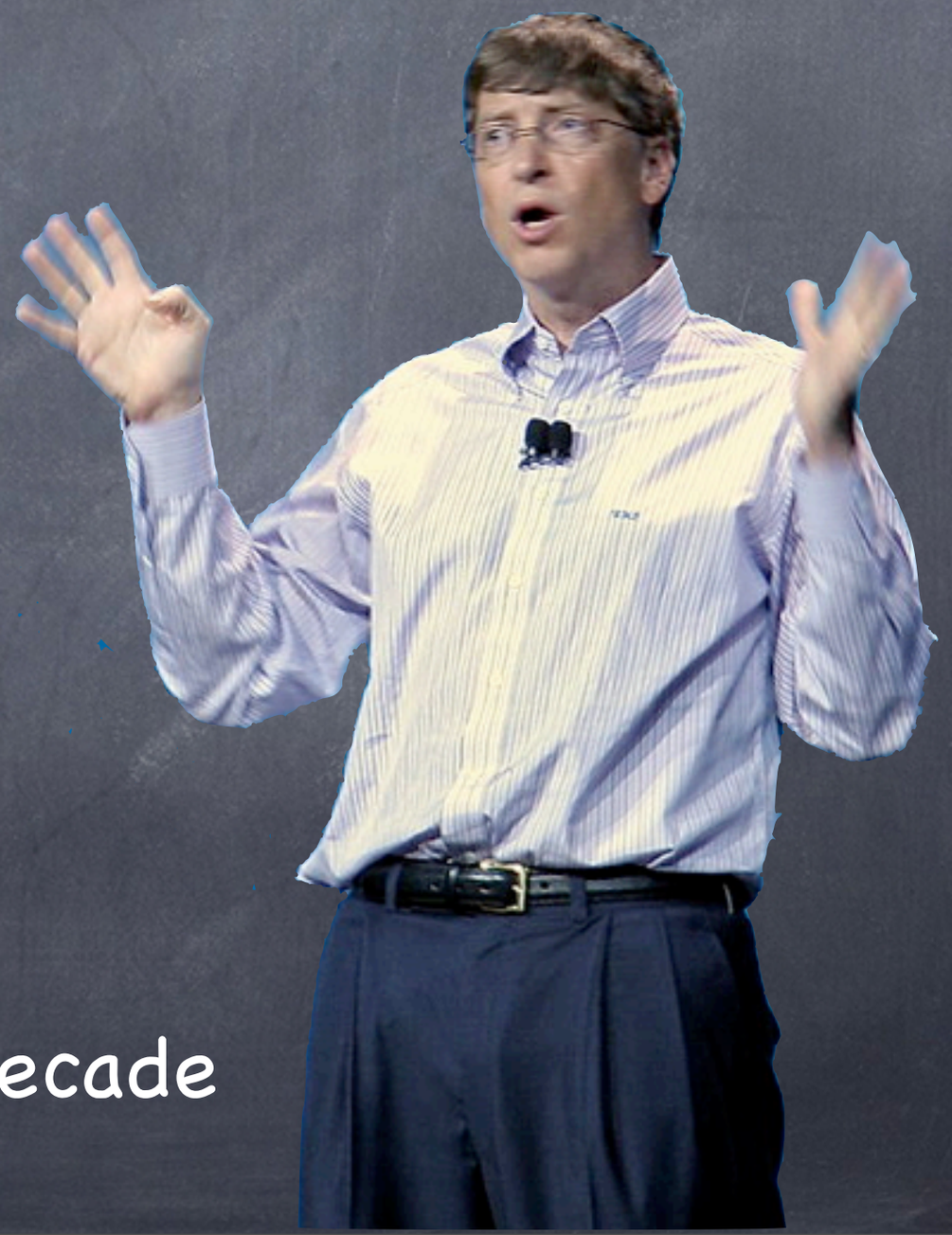
- So what's the right number?

  6,400K?
  64,000K?
  640,000K?
  6,400,000K?
  64,000,000K?

- There is no right number

- Target moves at 50x-100x per decade

"I've said some stupid things and some wrong things, but not that. No one involved in computers would ever say that a certain amount of memory is enough for all time …" - Bill Gates, 1996

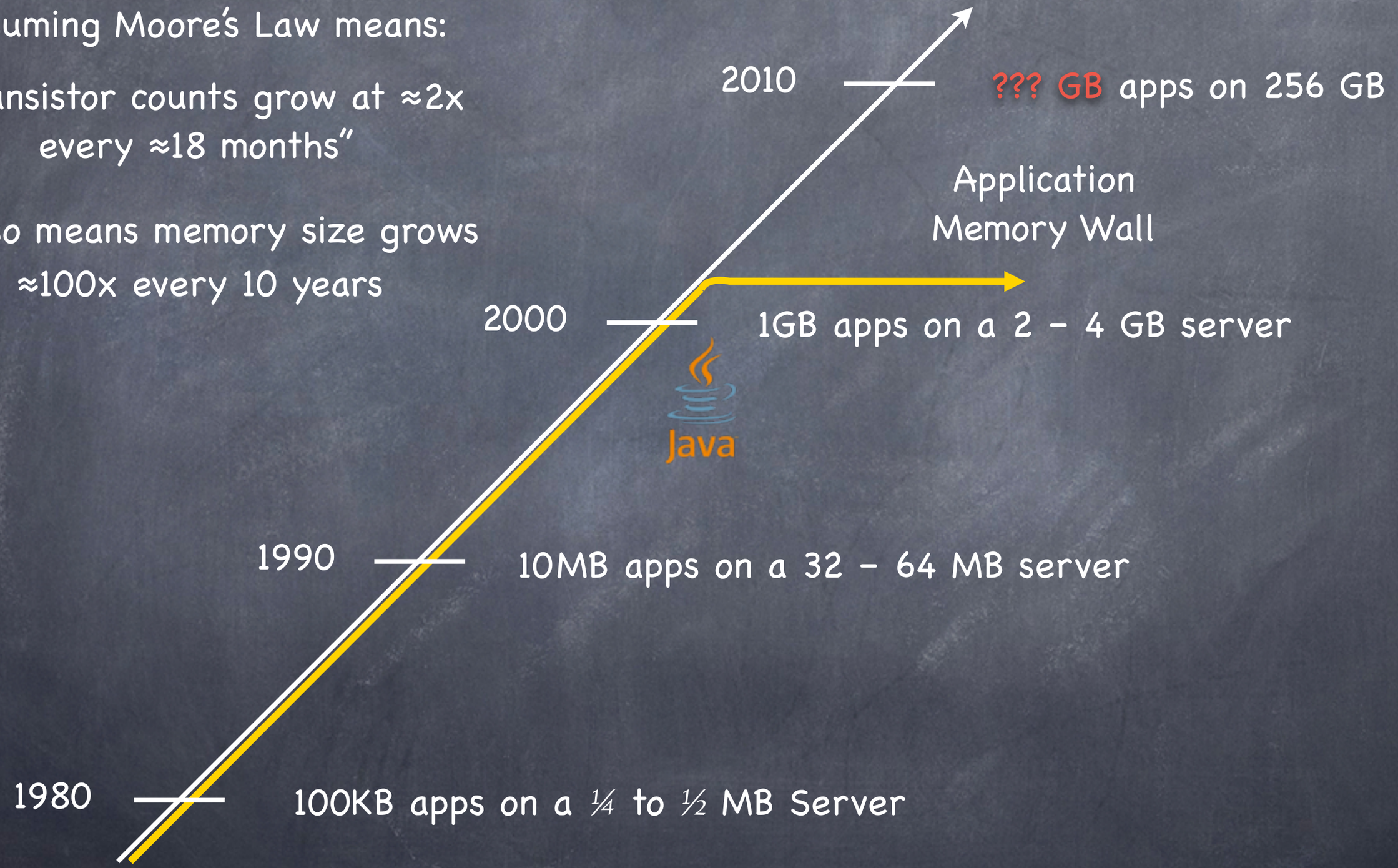


"Tiny" application history

Assuming Moore's Law means:

"transistor counts grow at ≈2x every ≈18 months"

It also means memory size grows ≈100x every 10 years



* "Tiny": would be "silly" to distribute

What is causing the Application Memory Wall?

- Garbage Collection is a clear and dominant cause
- There seem to be practical heap size limits for applications with responsiveness requirements
- [Virtually] All current commercial JVMs will exhibit a multi-second pause on a normally utilized 2-6GB heap.
  - It's a question of "When" and "How often", not "If".
  - GC tuning only moves the "when" and the "how often" around
- Root cause: The link between scale and responsiveness

AZUL SYSTEMS

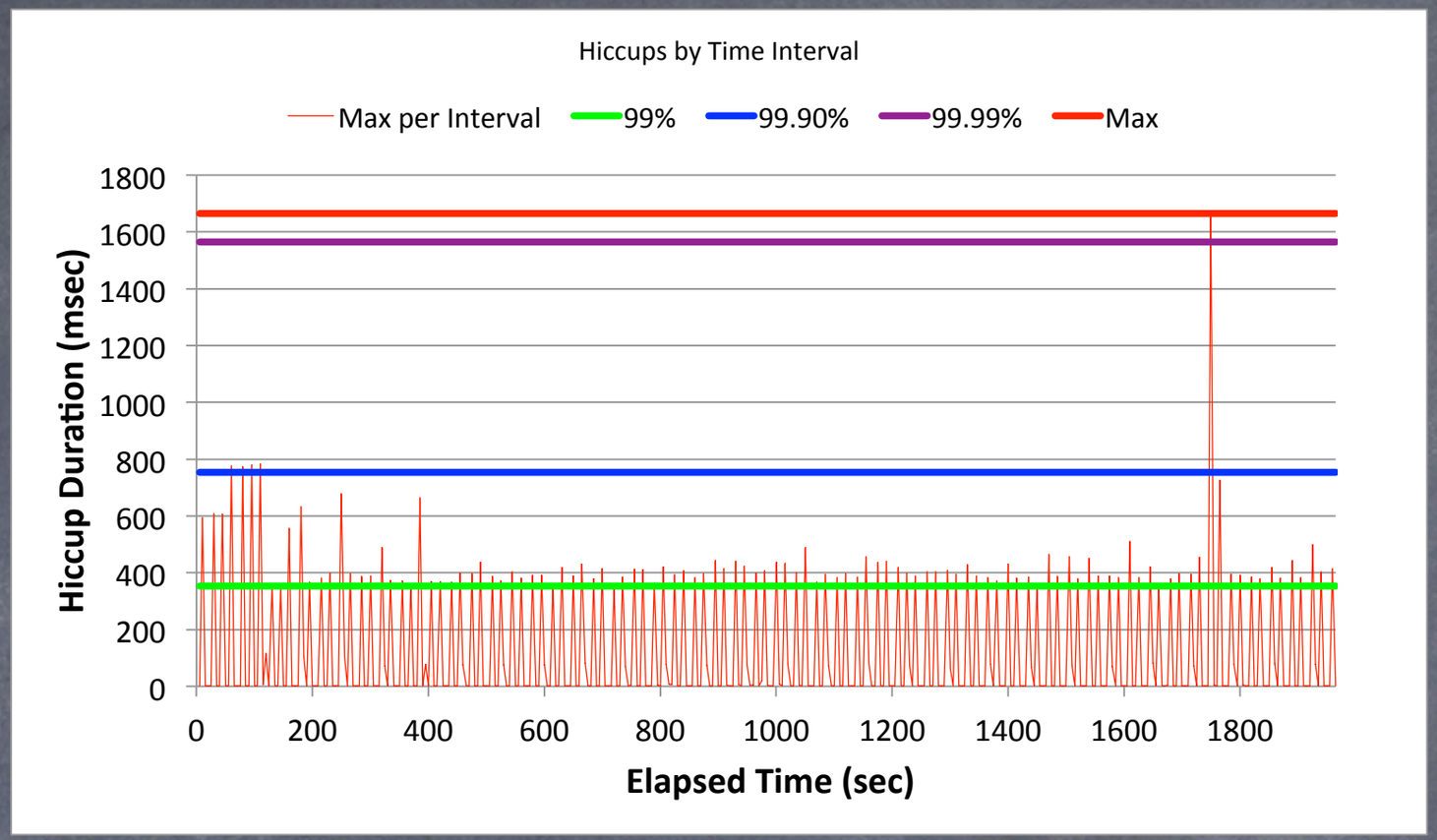
The problems that need solving
(areas where the state of the art needs improvement)

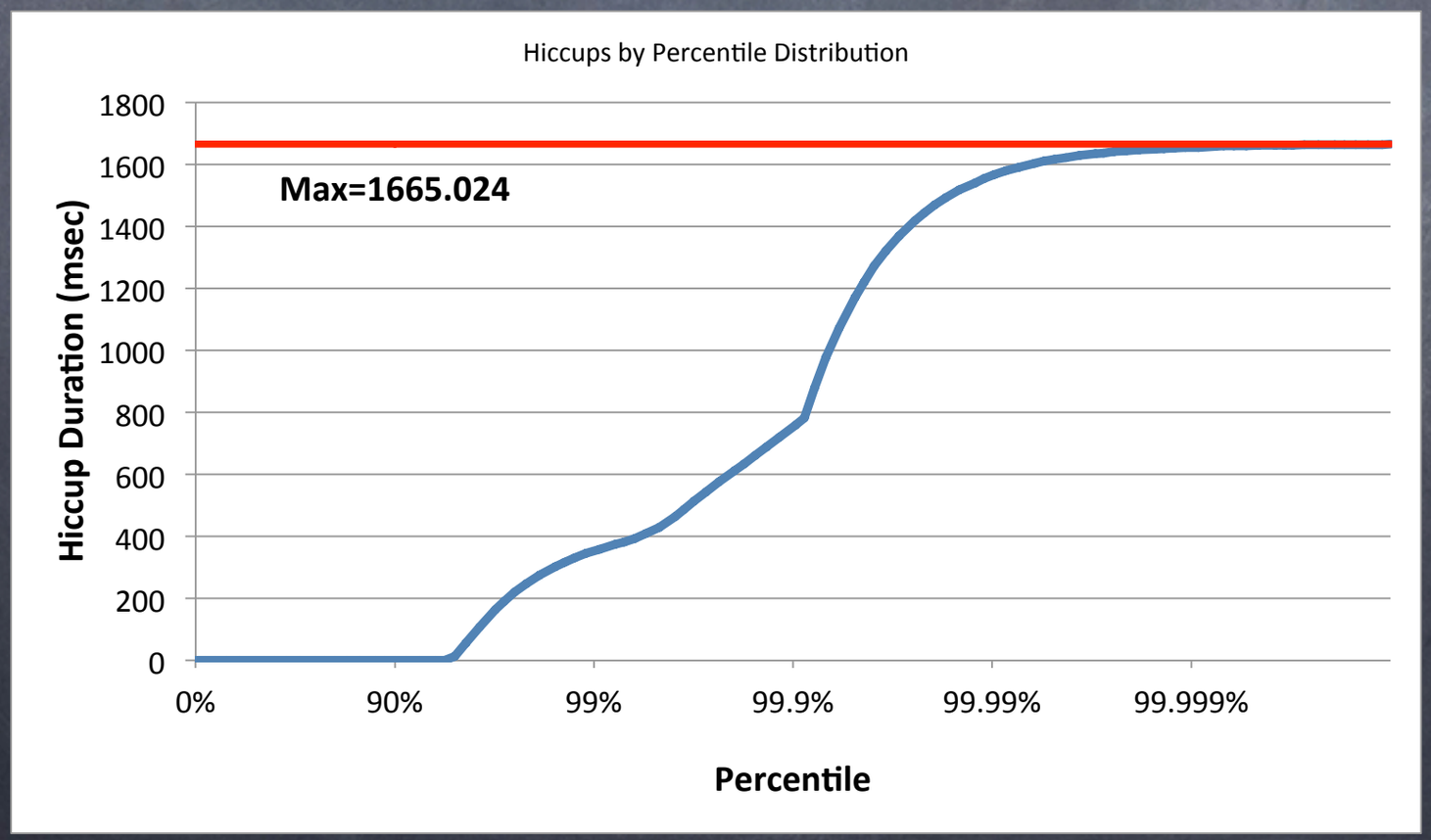
- Robust Concurrent Marking
  - In the presence of high mutation and allocation rates
  - Cover modern runtime semantics (e.g. weak refs, lock deflation)
- Compaction that is not monolithic-stop-the-world
  - E.g. stay responsive while compacting ¼ TB heaps
  - Must be robust: not just a tactic to delay STW compaction
  - [current "incremental STW" attempts fall short on robustness]
- Young-Gen that is not monolithic-stop-the-world
  - Stay responsive while promoting multi-GB data spikes
  - Concurrent or "incremental STW" may both be ok
  - Surprisingly little work done in this specific area

jHiccup

AZUL SYSTEMS®

Incontinuities in Java platform execution





AZUL SYSTEMS

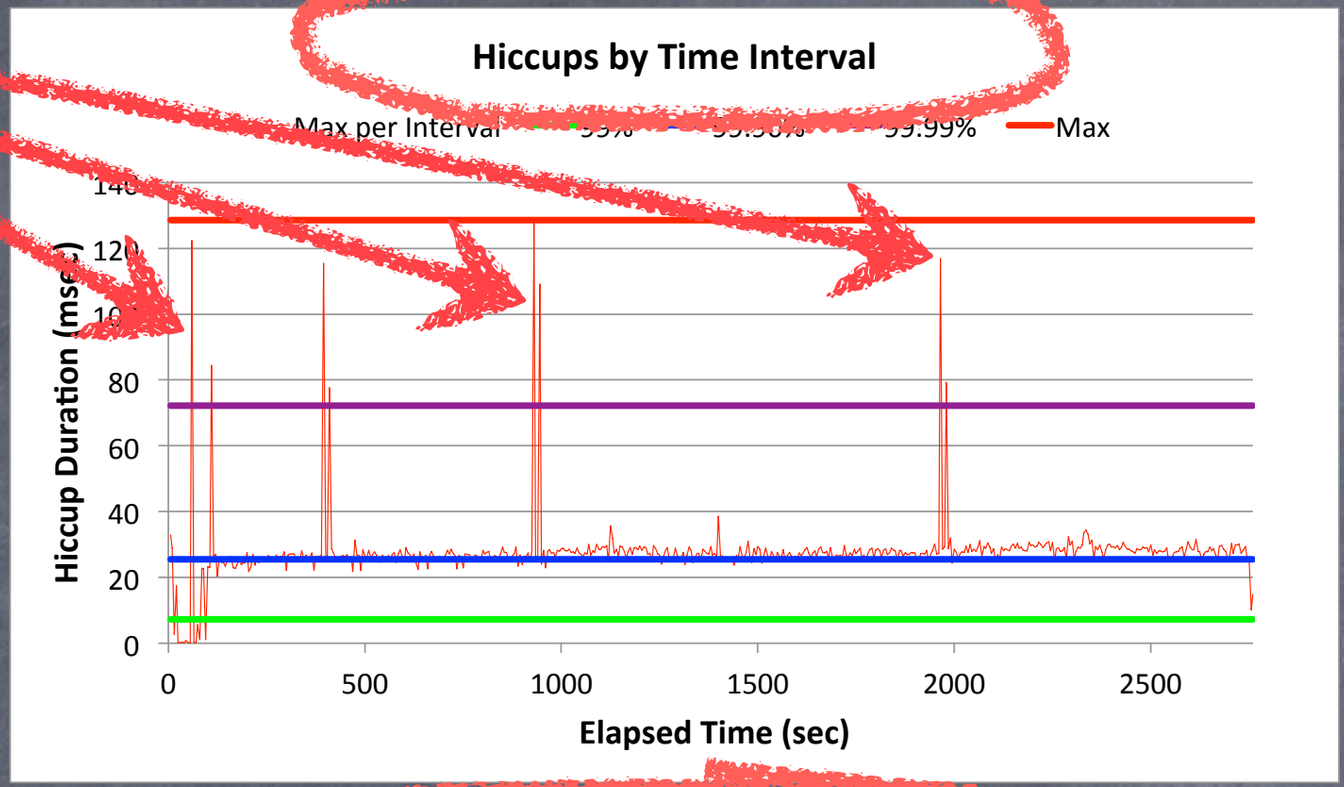
jHiccup

- A tool for capturing and displaying platform hiccups
  - Records any observed non-continuity of the underlying platform
  - Plots results in simple, consistent format
- Simple, non-intrusive
  - As simple as adding the word "jHiccup" to your java launch line
  - % jHiccup java myflags myApp
  - (Or use as a java agent)
  - Adds a background thread that samples time @ 1000/sec
- Open Source
  - Released to the public domain, creative commons CC0

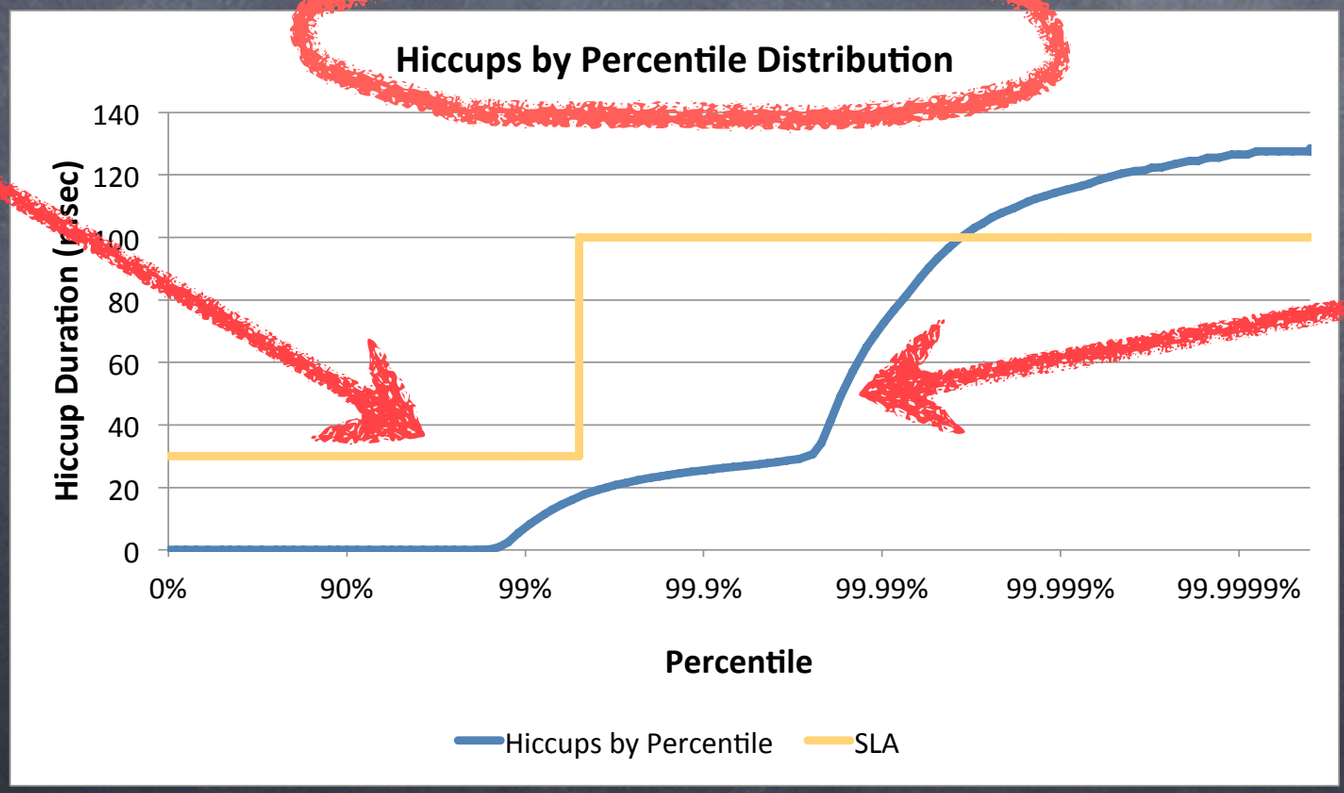
AZUL SYSTEMS

Telco App Example

Max Time per interval



Optional SLA plotting



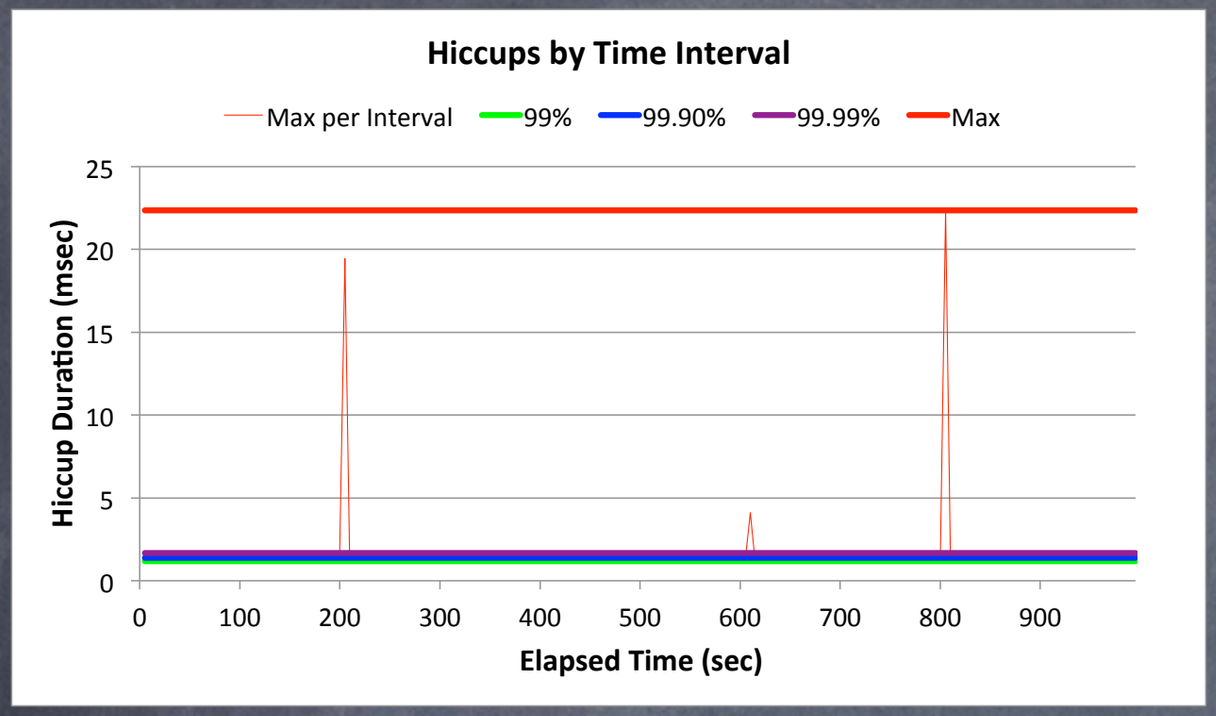
Hiccup duration at percentile levels

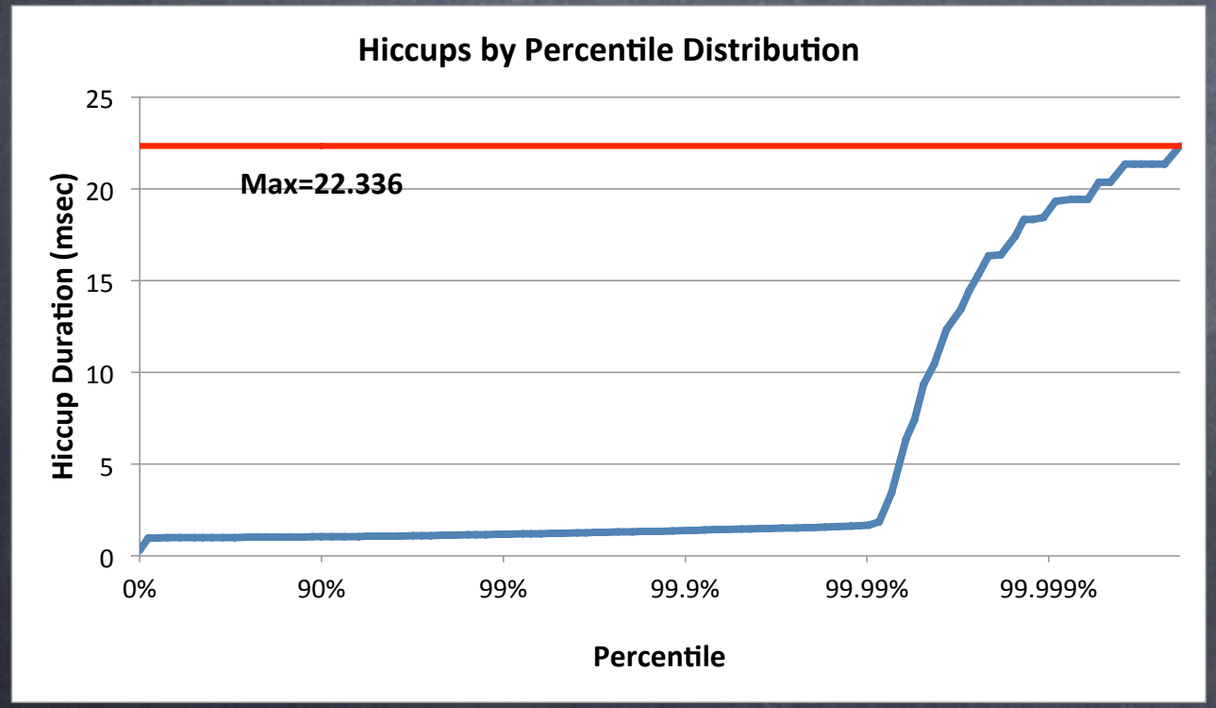
AZUL SYSTEMS

Examples

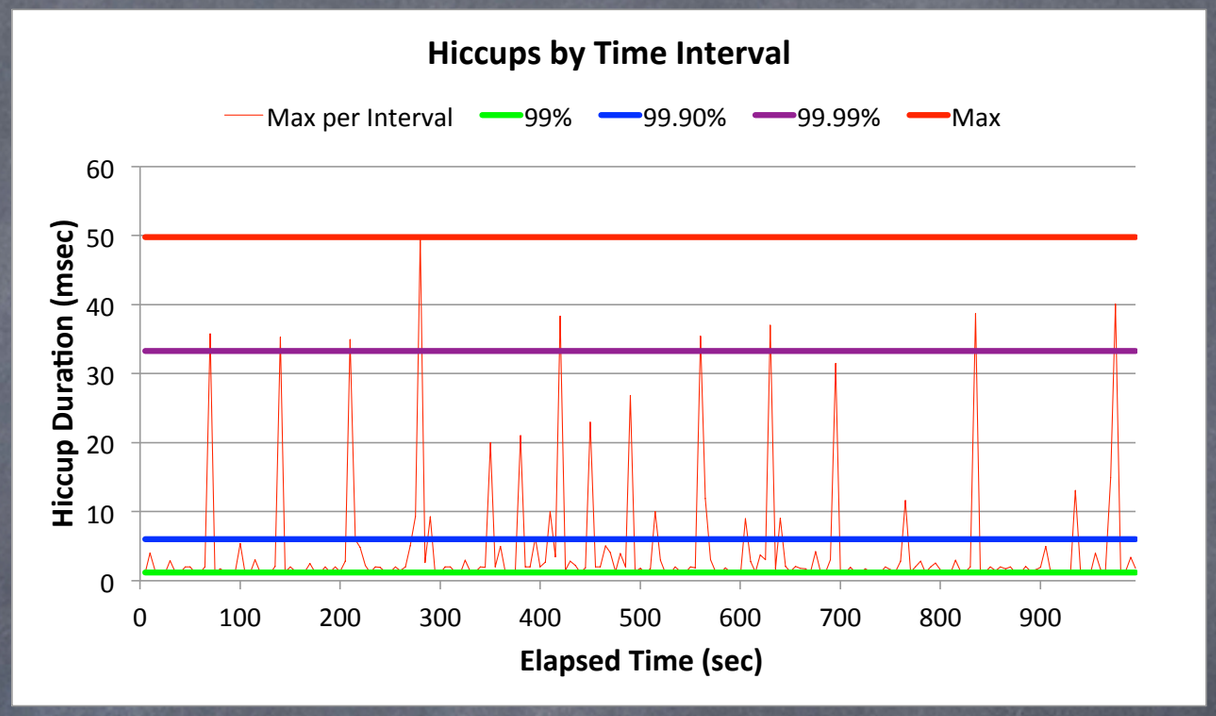
AZUL SYSTEMS®

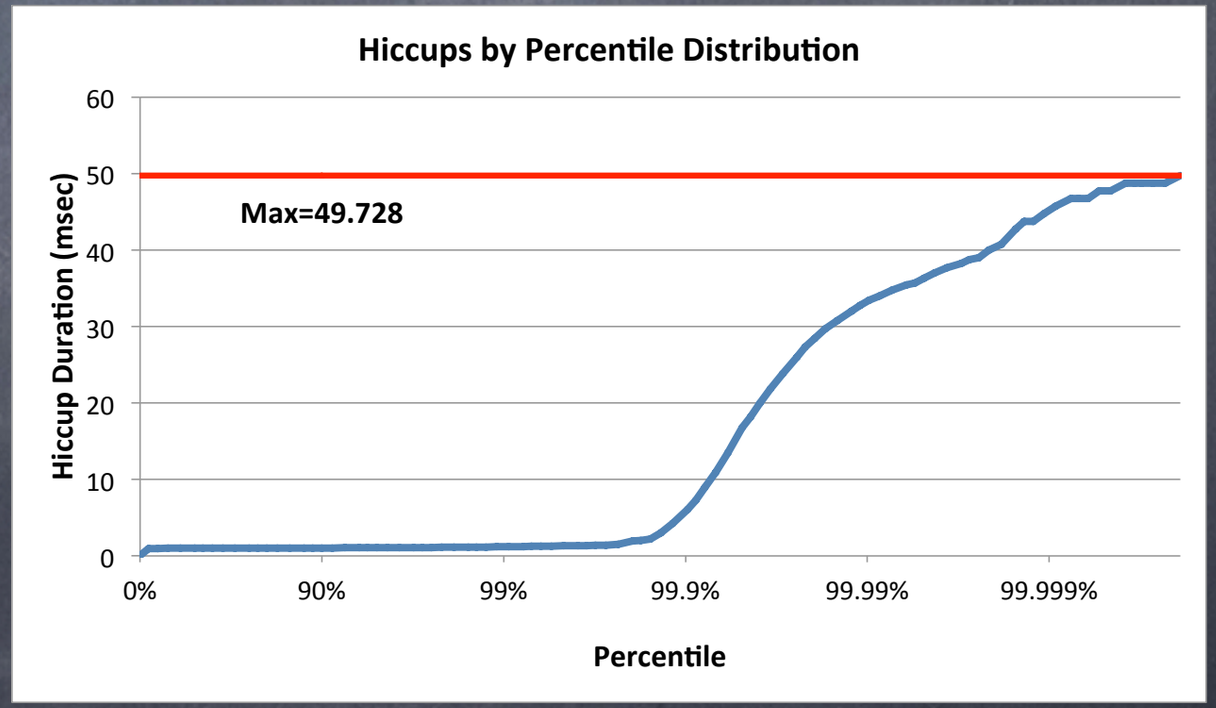
Idle App on Quiet System





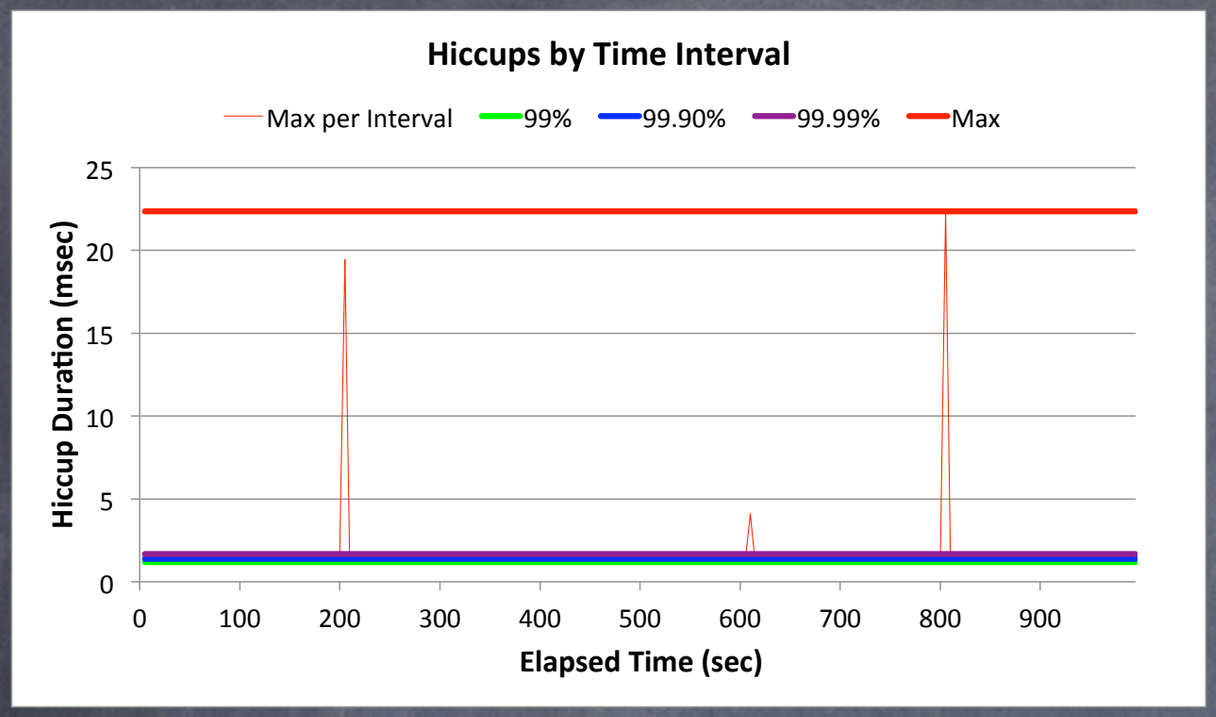
Idle App on Busy System

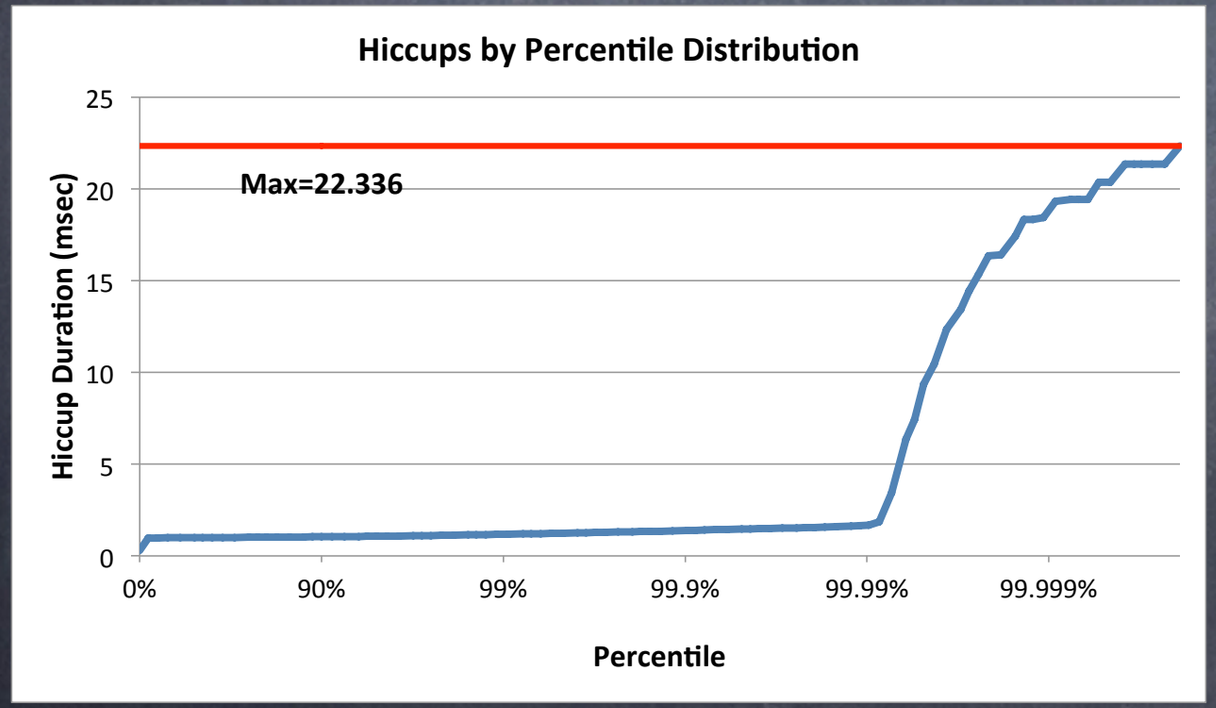




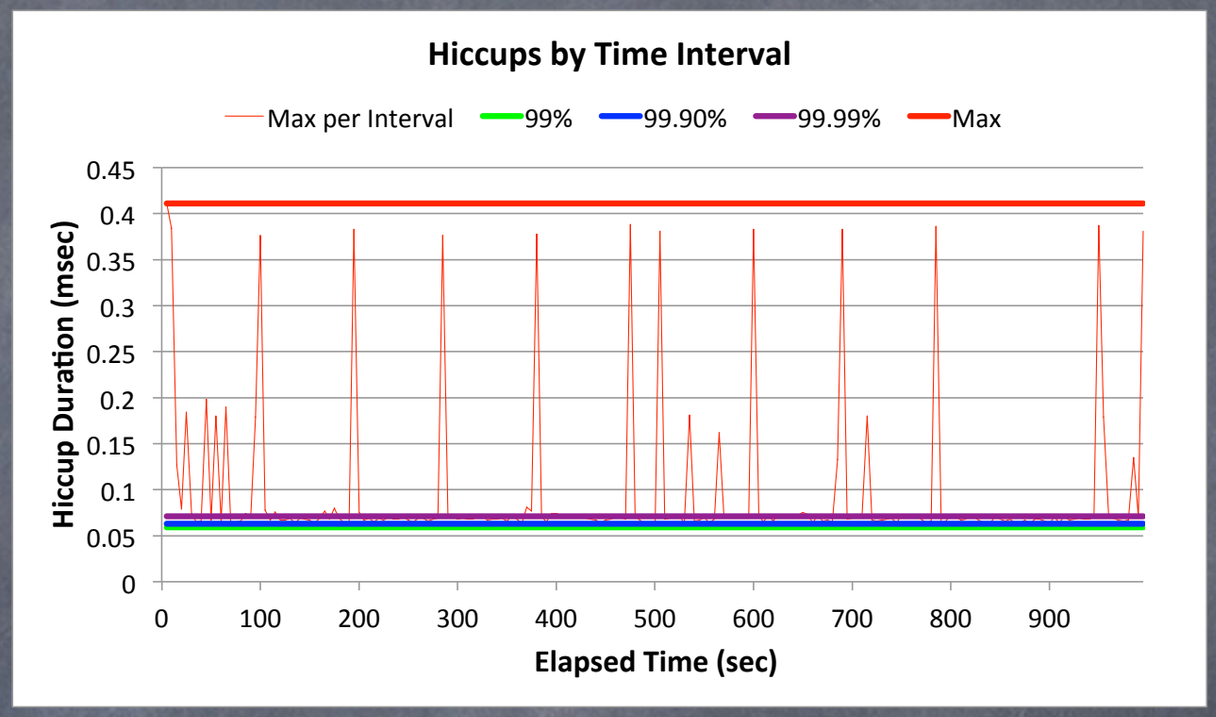
AZUL SYSTEMS

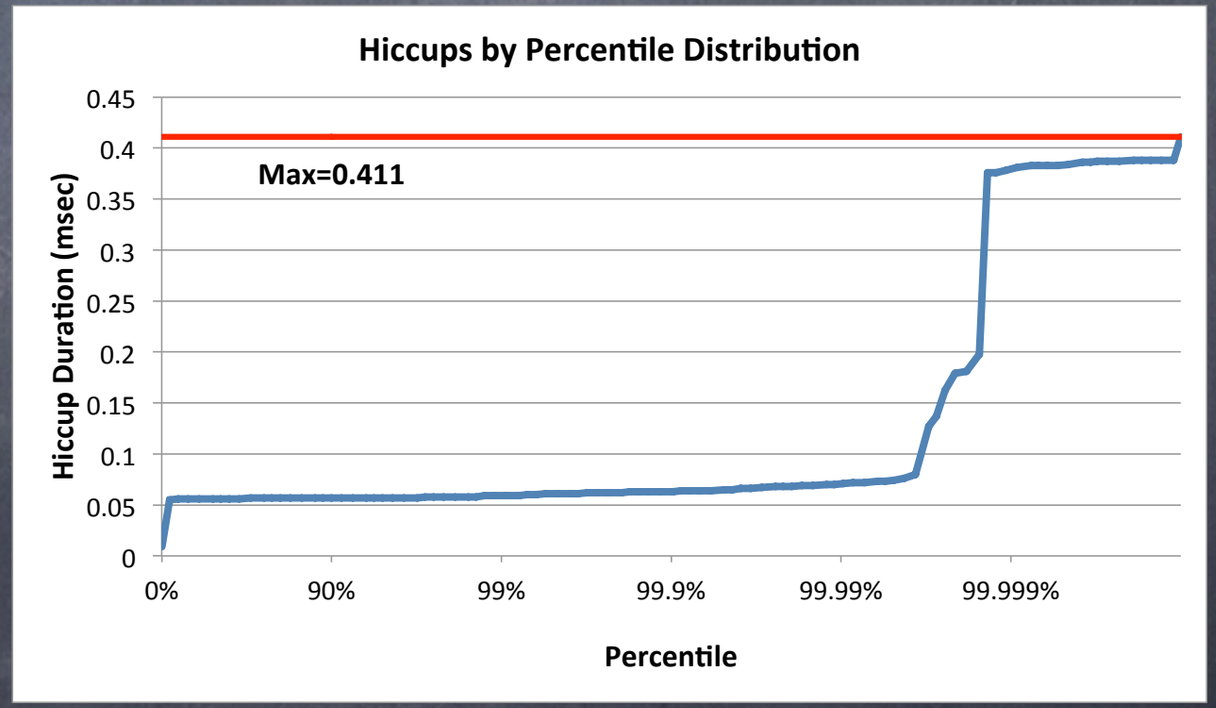
Idle App on Quiet System





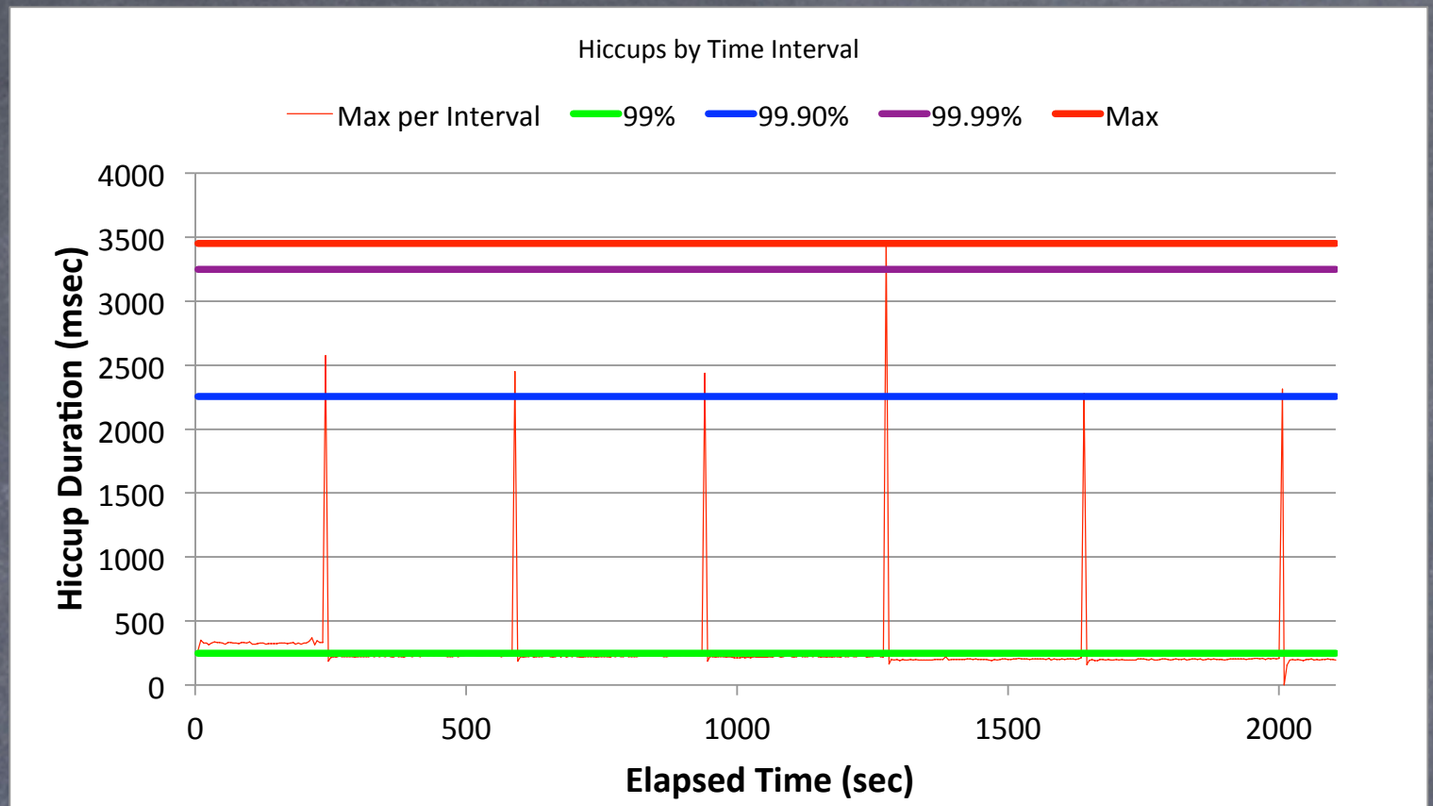
Idle App on Dedicated System

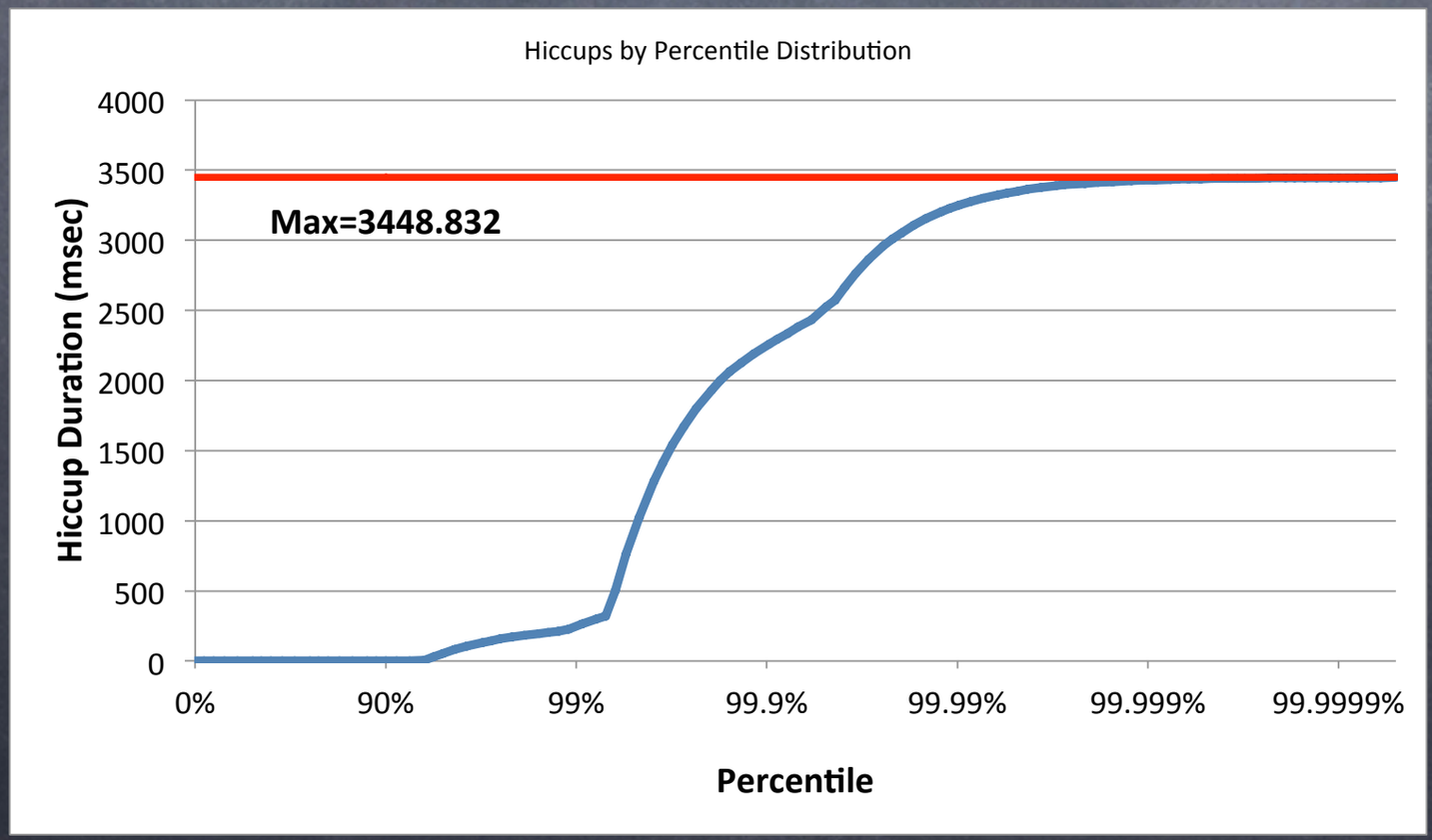




AZUL SYSTEMS

EHCache: 1GB data set under load





AZUL SYSTEMS

Fun with jHiccup

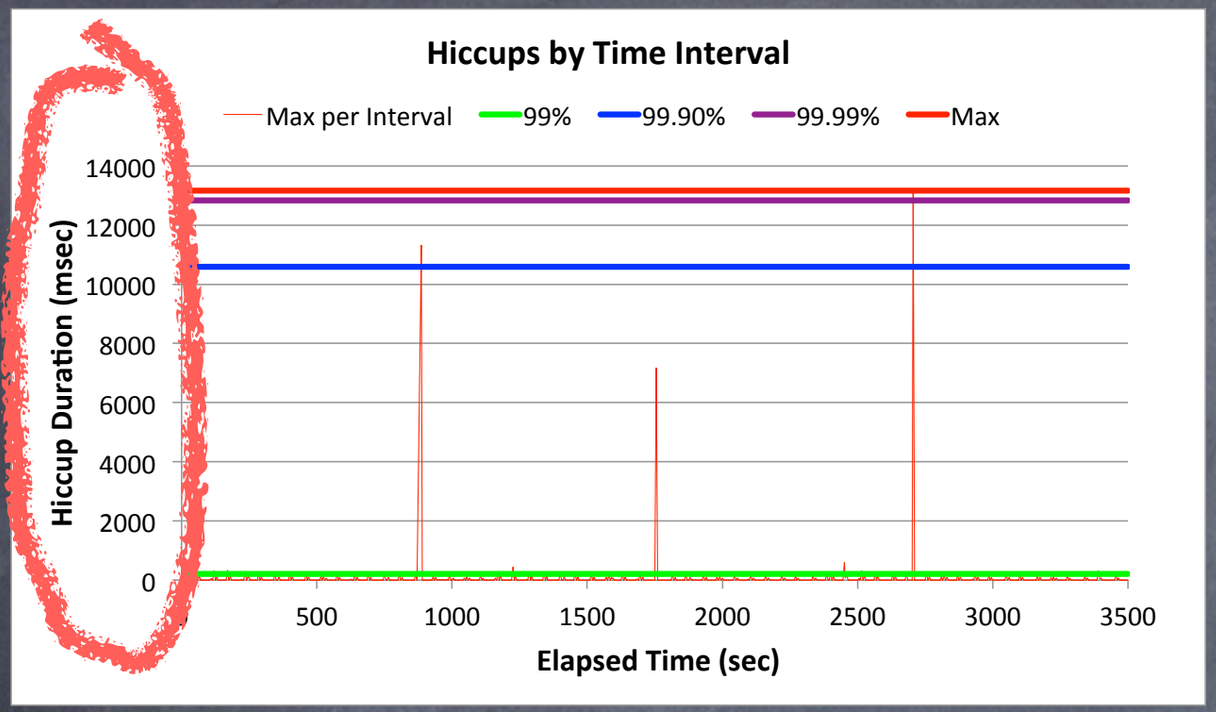
Charles Nutter @headius 20 Jan

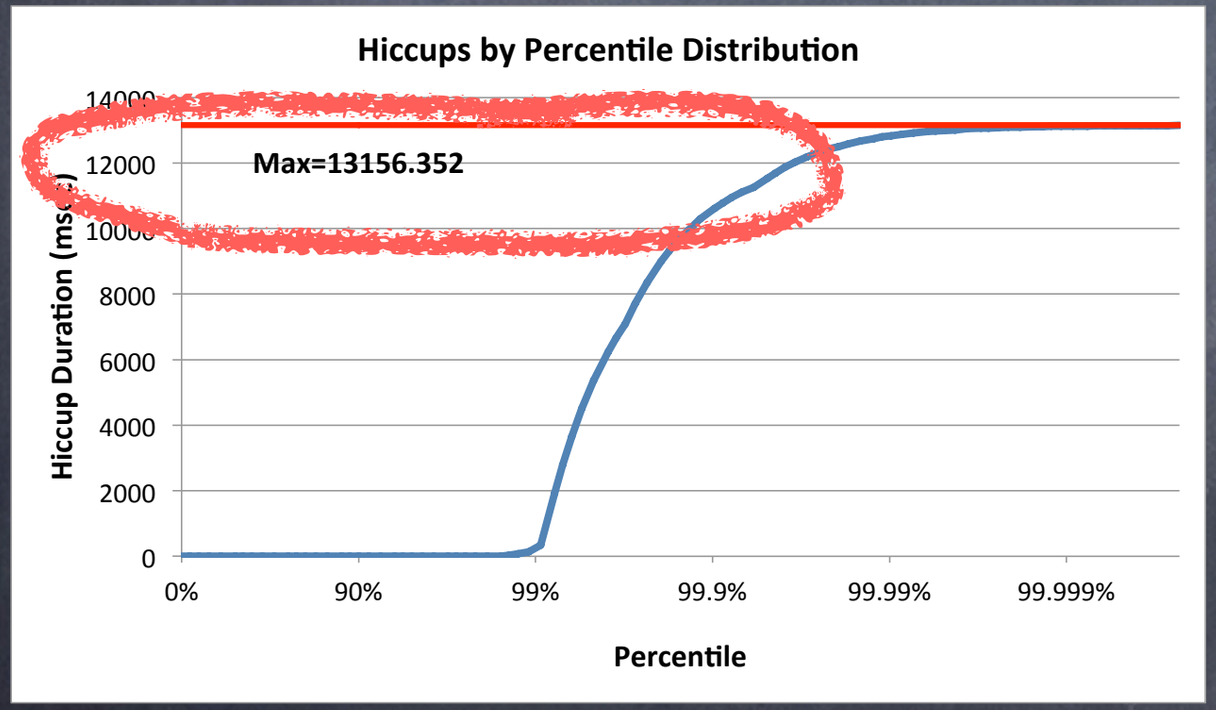
jHiccup, @AzulSystems' free tool to show you why your JVM sucks compared to Zing: bit.ly/wsH5A8 (thx @bascule)

Retweeted by Gil Tene

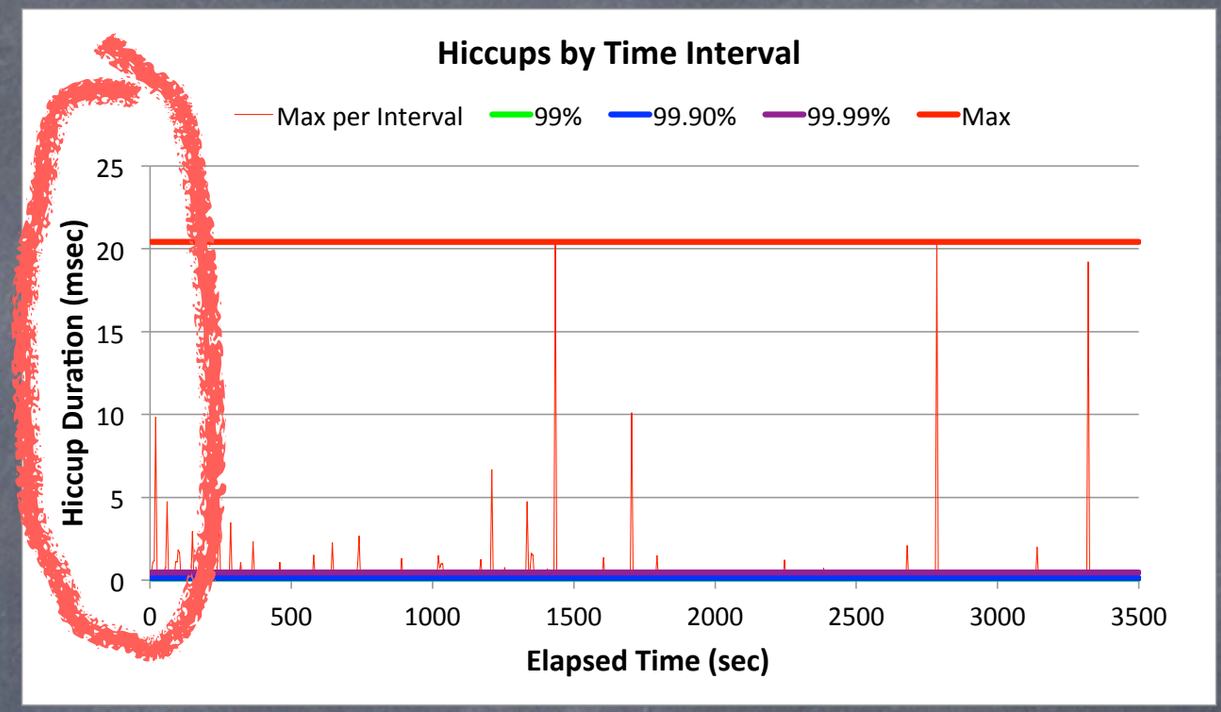
AZUL SYSTEMS

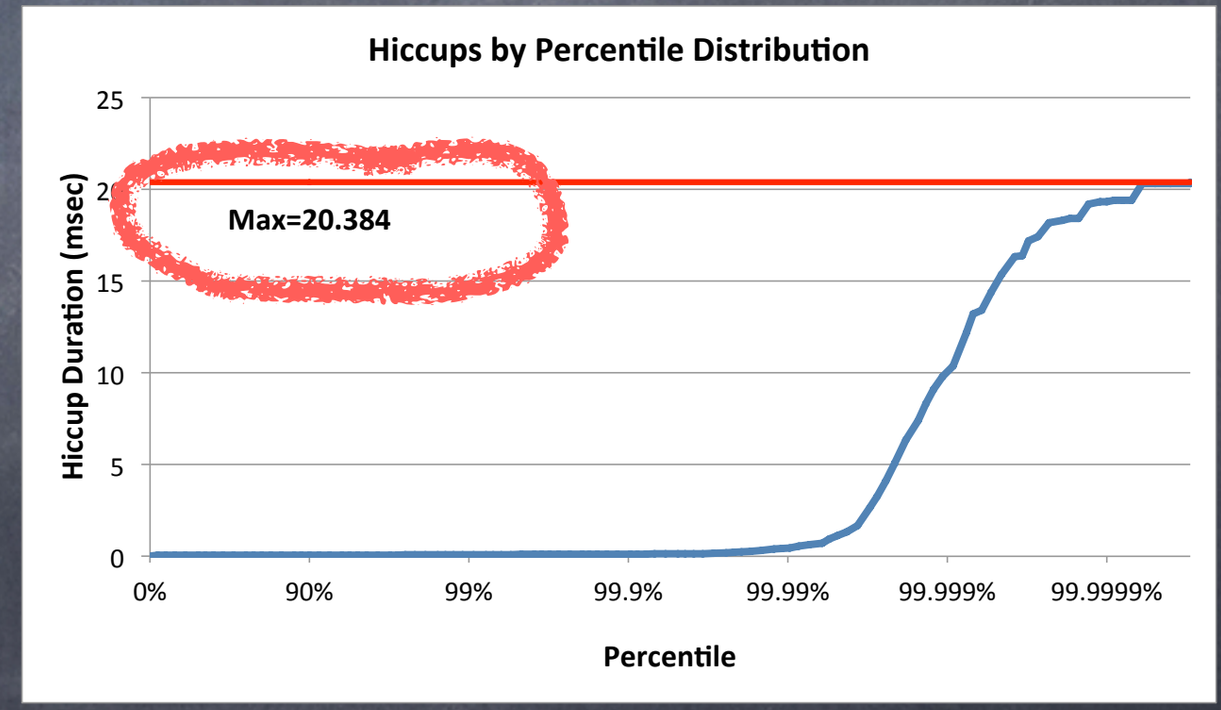
Oracle HotSpot CMS, 1GB in an 8GB heap





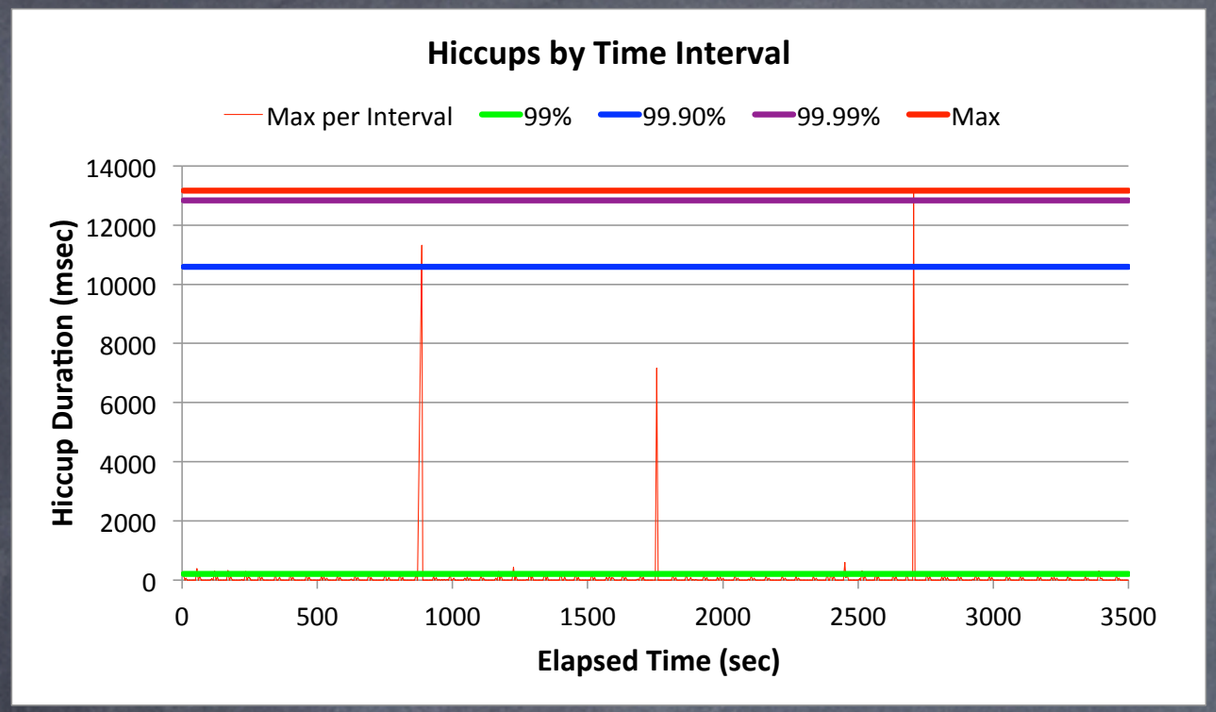
Zing 5, 1GB in an 8GB heap

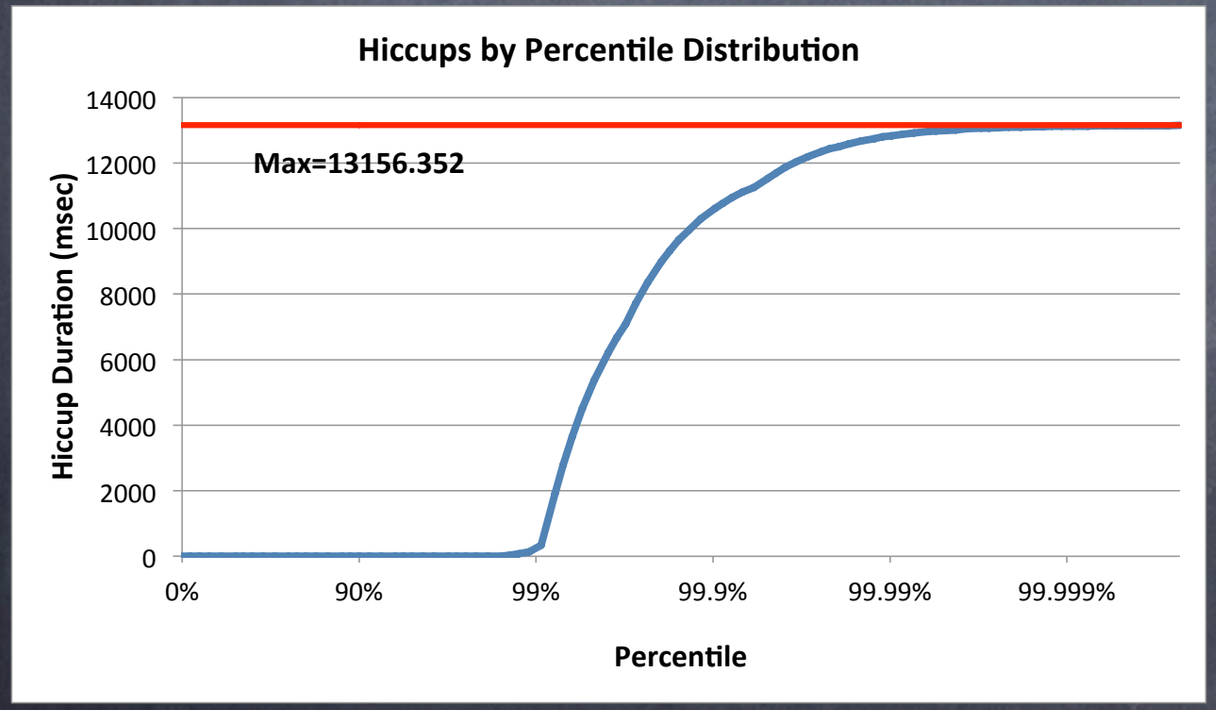




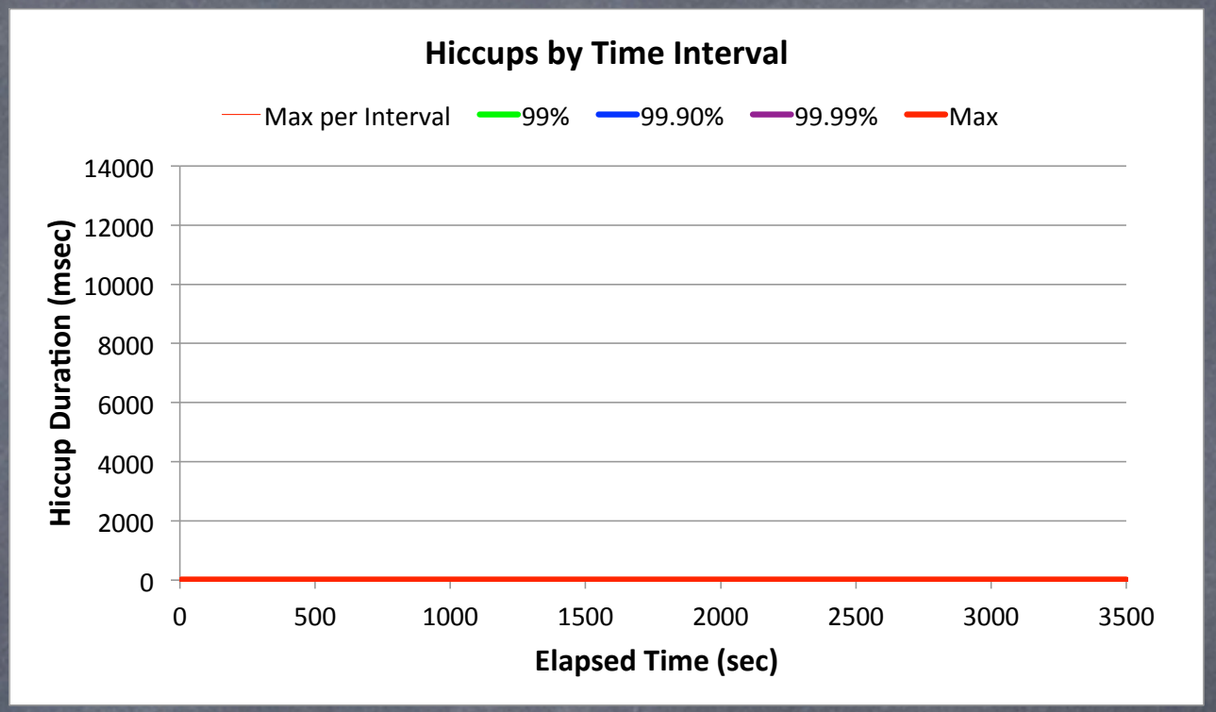
AZUL SYSTEMS

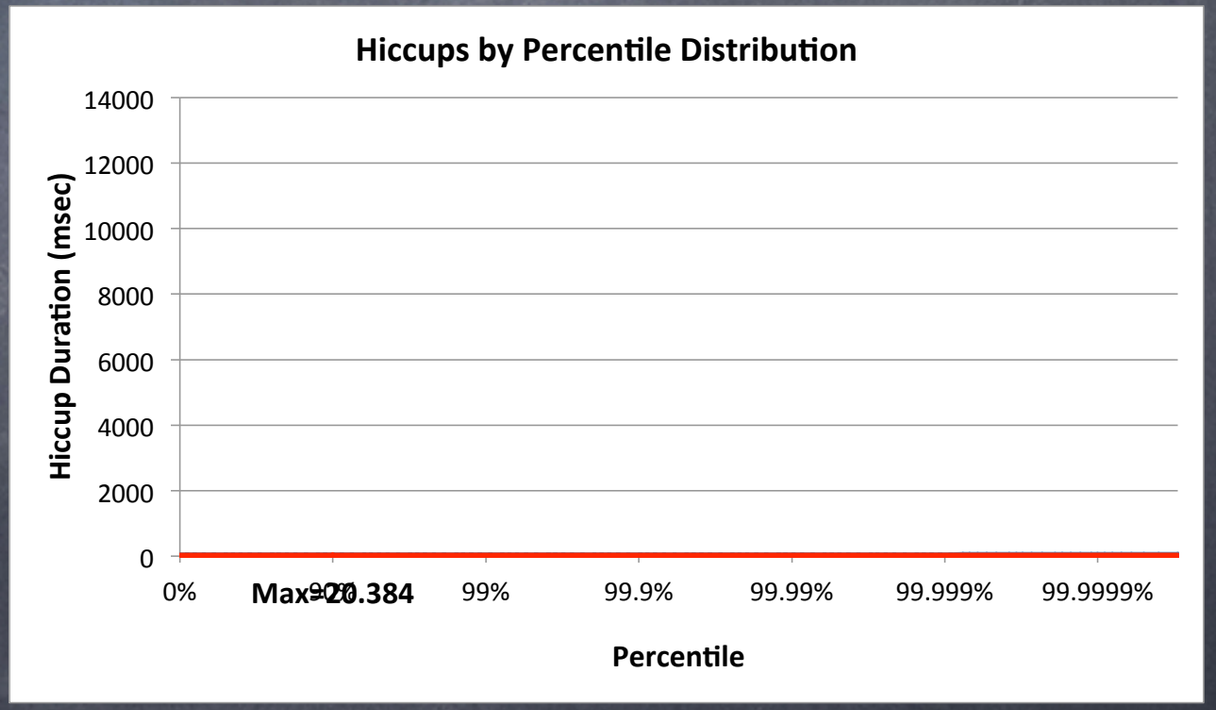
Oracle HotSpot CMS, 1GB in an 8GB heap





Zing 5, 1GB in an 8GB heap





Drawn to scale

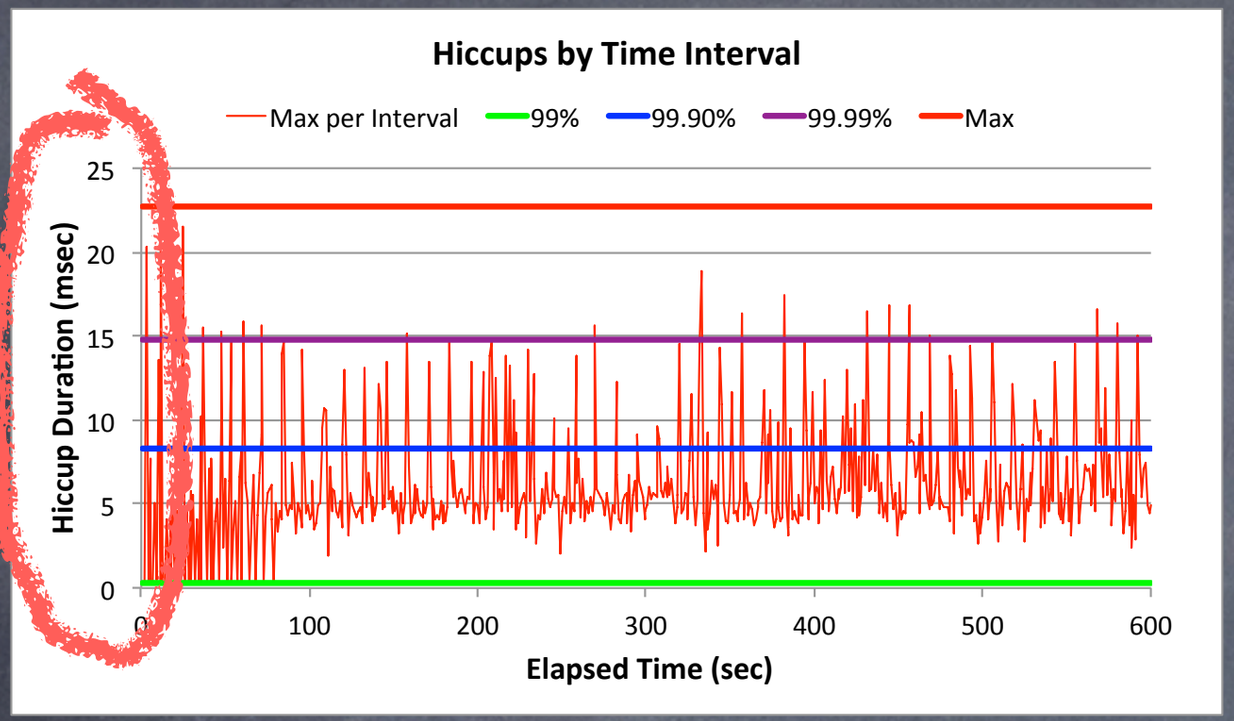
AZUL SYSTEMS

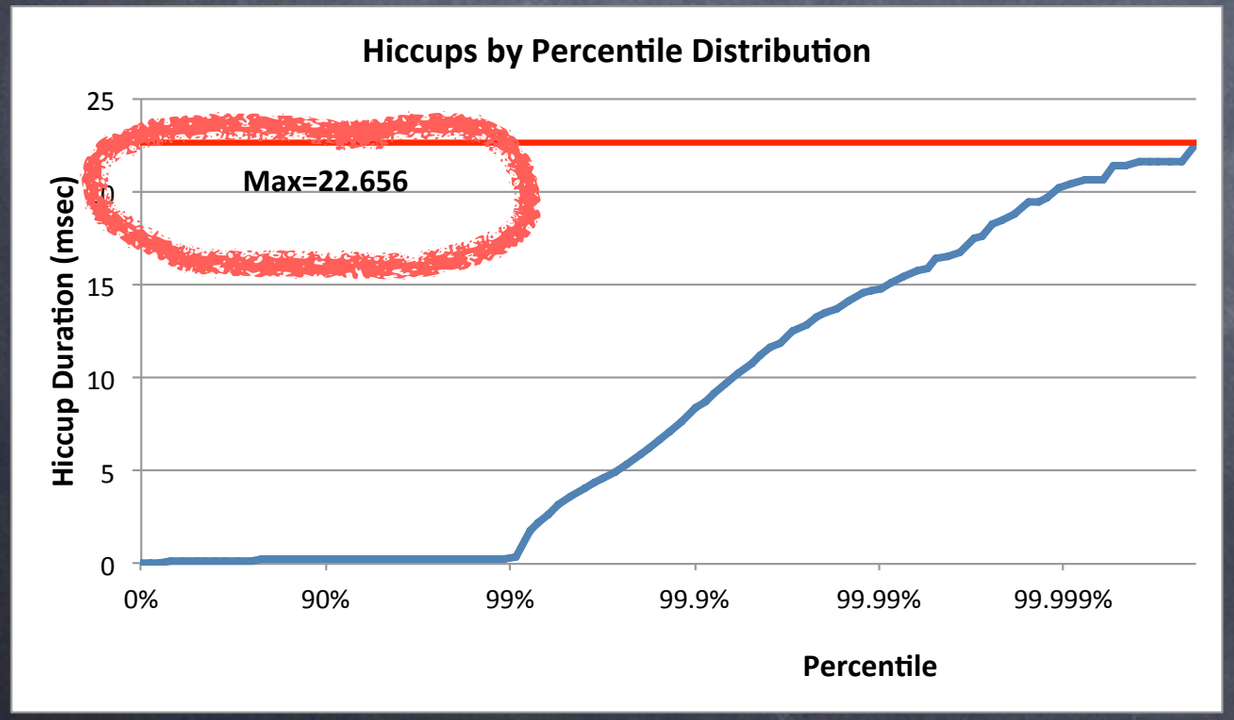
What you can expect (from Zing) in the low latency world

- Assuming individual transaction work is "short" (on the order of 1 msec), and assuming you don't have 100s of runnable threads competing for 10 cores...
- "Easily" get your application to < 10 msec **worst case**
- With some tuning, 2-3 msec **worst case**
- Can go to below 1 msec **worst case**...
  - May require heavy tuning/tweaking
  - Mileage WILL vary

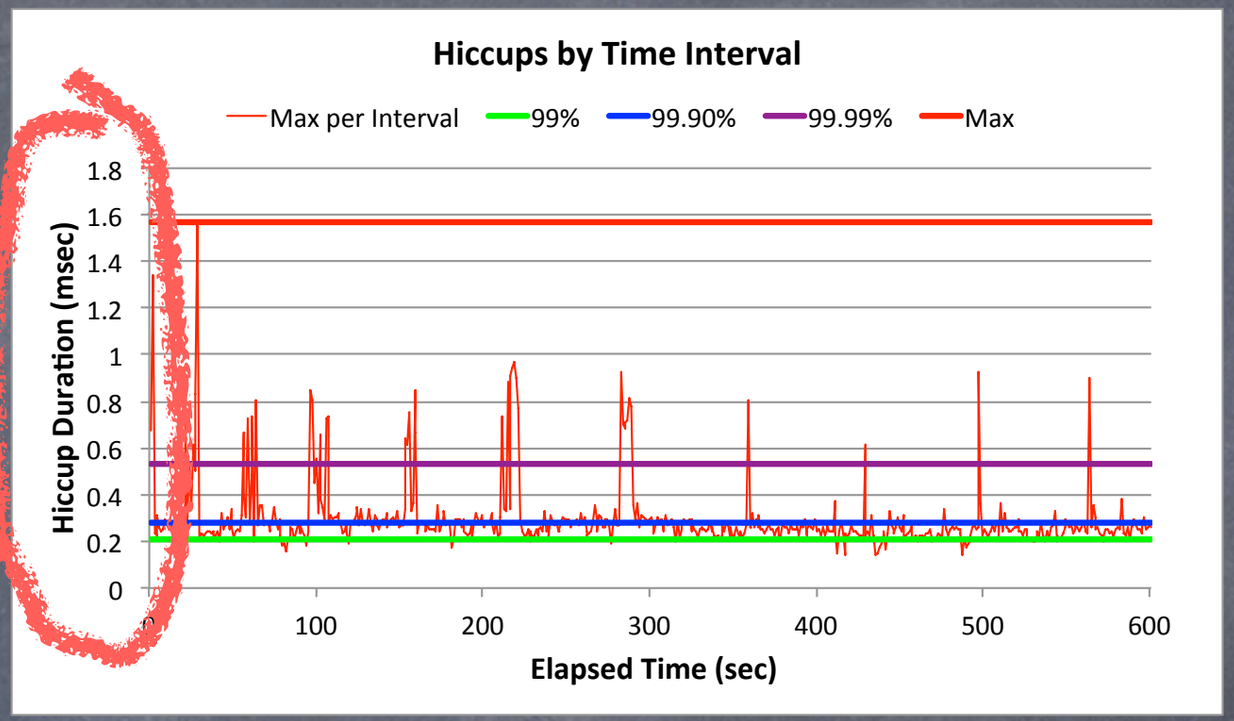
AZUL SYSTEMS

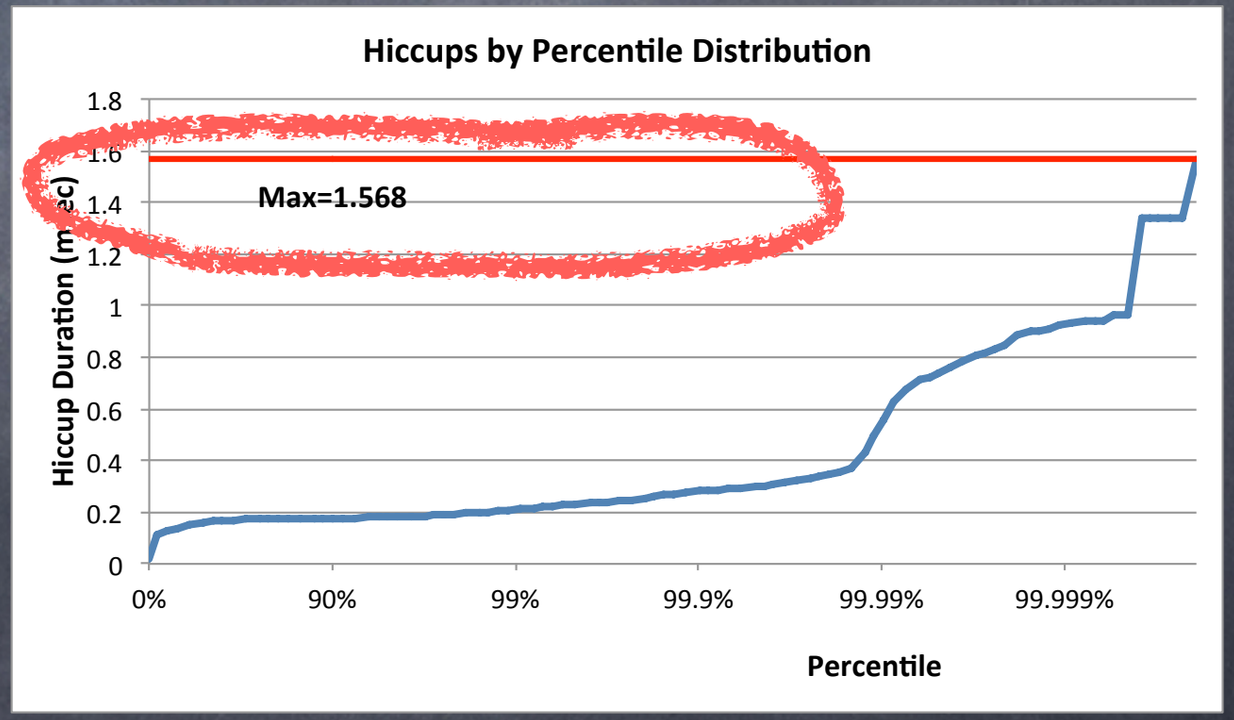
Oracle HotSpot (pure newgen)





Zing

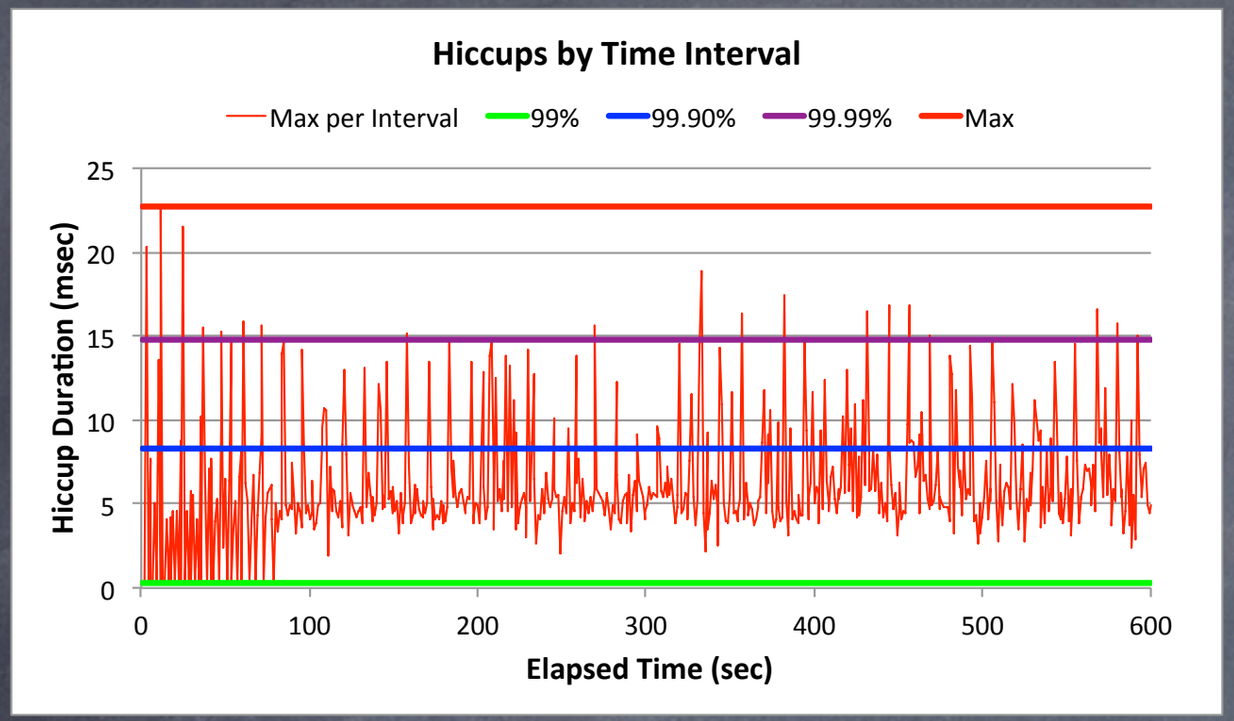


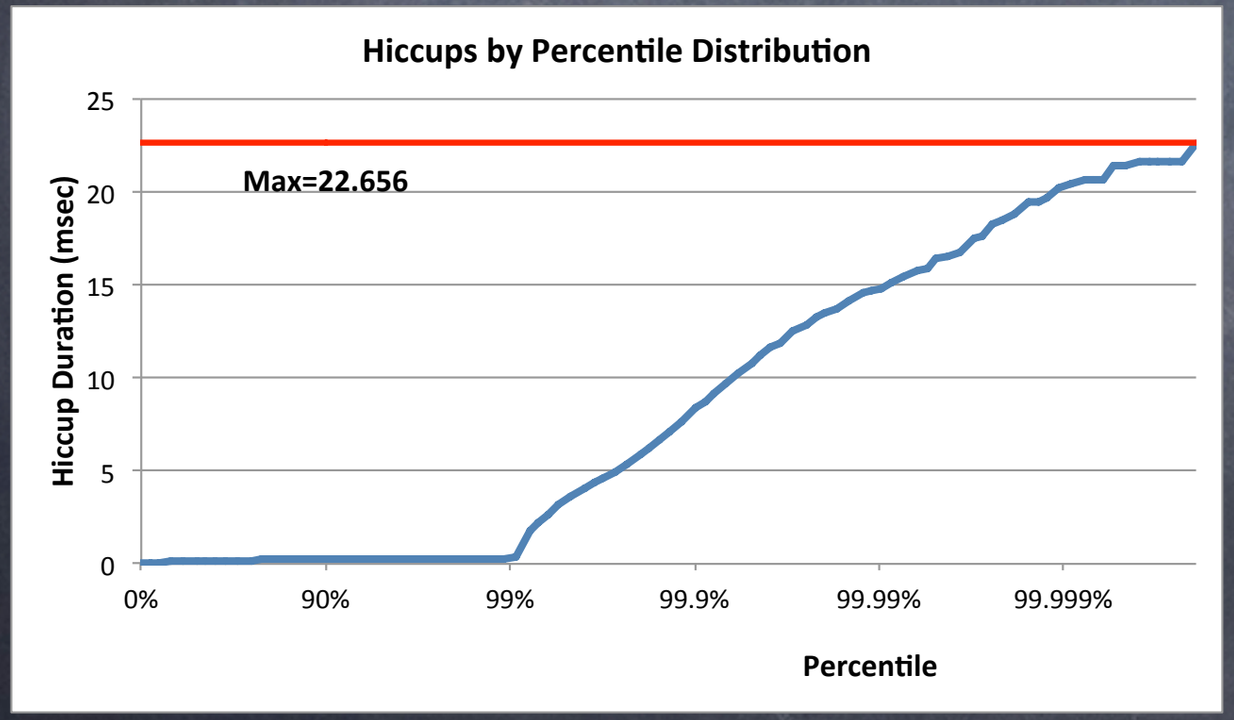


Low latency trading application

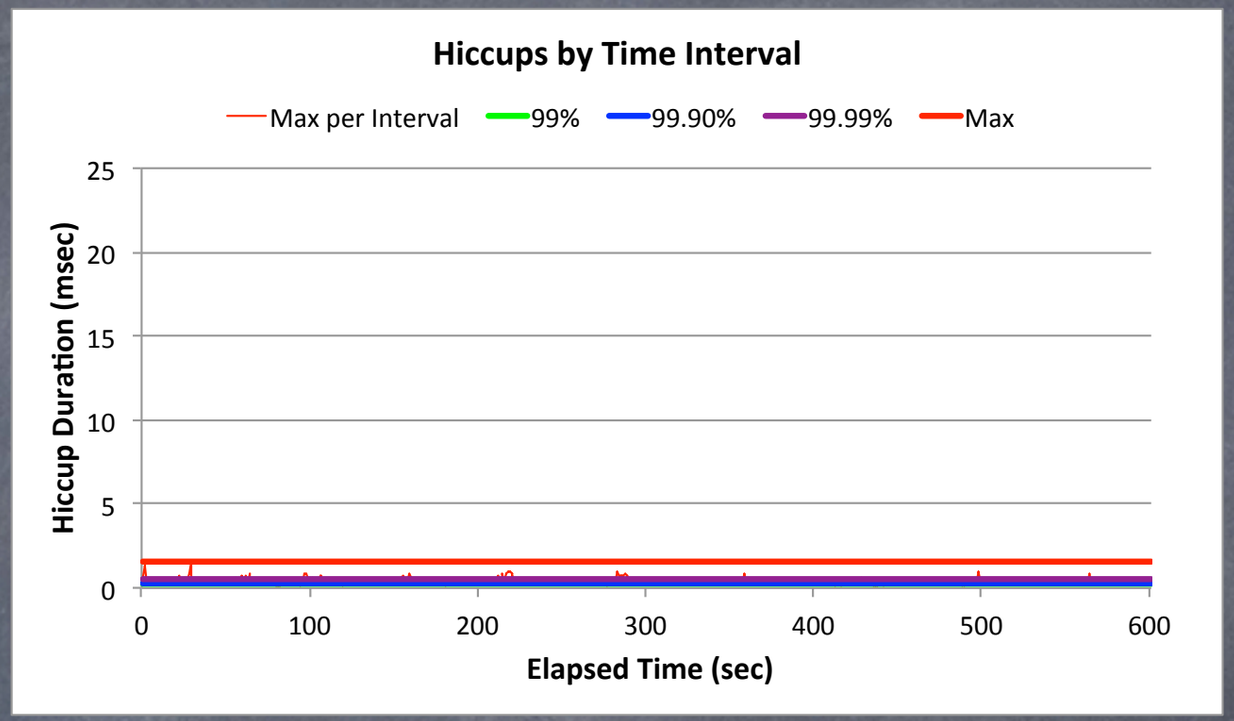
AZUL SYSTEMS

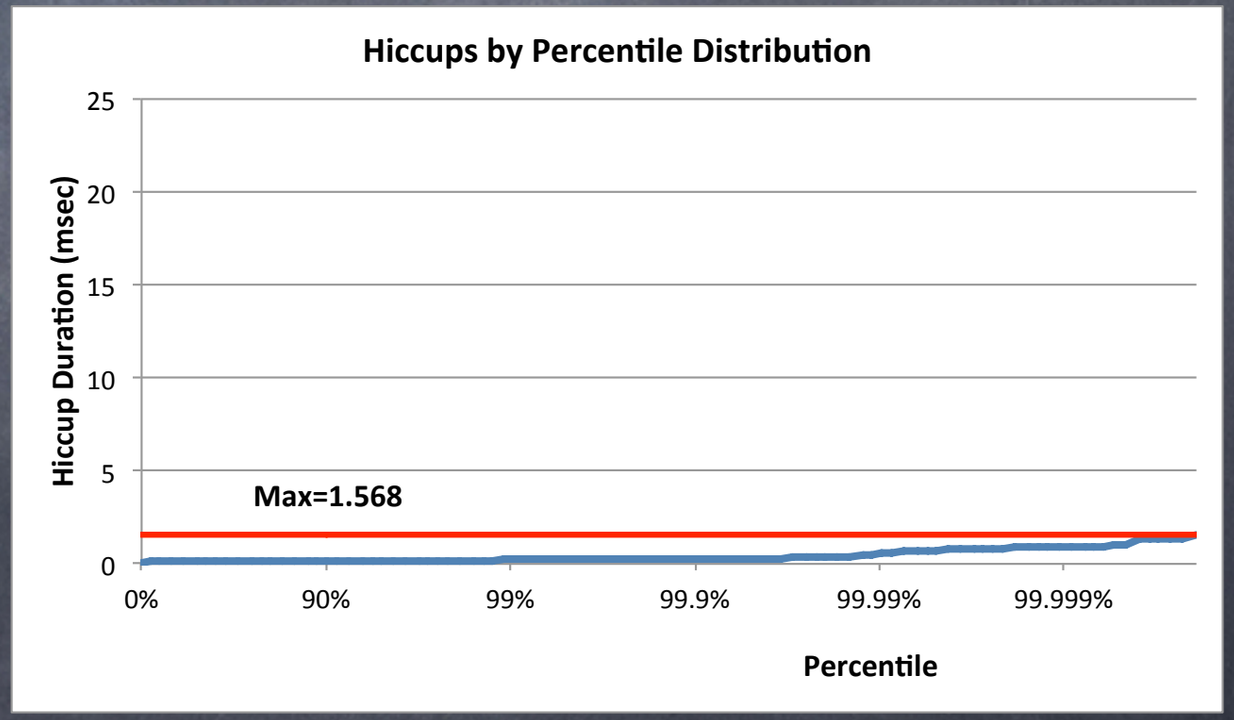
Oracle HotSpot (pure newgen)





Zing





Low latency - Drawn to scale

AZUL SYSTEMS

Open Discussion

http://www.azulsystems.com

http://www.jhiccup.com