How NOT to Measure Latency

An attempt to share wisdom...

Matt Schuetze, Product Management Director, Azul Systems
High level agenda

- Some latency behavior background
- The pitfalls of using “statistics”
- Latency “philosophy” questions
- The Coordinated Omission Problem
- Some useful tools
- Use tools for bragging
About Gil Tene – Intended Speaker

- co-founder, CTO @Azul Systems
- Have been working on “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...
- JCP EC Member...
- Not Cloned Yet…
About me: Matt Schuetze

- Product Manager @Azul Systems
- Stewardship of Azul’s product roadmap.
- Started career in radar systems. Measured how “stealthy” is an aircraft.
- Moved to enterprise software development in 2000
- Built professional grade monitoring and profiling tools. More types of measurements.
- Measure my clone: Ben Affleck
We make scalable Virtual Machines

Have built “whatever it takes to get job done” since 2002

3 generations of custom SMP Multi-core HW (Vega)

Zing: Pure software for commodity x86

Known for Low Latency, Consistent execution, and Large data set excellence
Common fallacies

- Computers run application code continuously
- Response time can be measured as work units/time
- Response time exhibits a normal (or Gaussian or Poisson) distribution
- “Glitches” or “Semi-random omissions” in measurement don’t have a big effect.
A classic look at response time behavior

Response time as a function of load

- Average?
- Max?
- Median?
- 90%?
- 99.9%

source: IBM CICS server documentation, “understanding response times”
Response time over time

When we measure behavior over time, we often see:

“Hiccups”

source: ZOHO QEngine White Paper: performance testing report analysis
What happened here?

“Hiccups”

Source: Gil running an idle program and suspending it five times in the middle
The real world (a low latency example)

99%tile is ~60 usec

Max is ~30,000% higher than “typical”
Hiccups are [typically] strongly multi-modal

- They don’t look anything like a normal distribution
- They usually look like periodic freezes
- A complete shift from one mode/behavior to another
- Mode A: “good”.
- Mode B: “Somewhat bad”
- Mode C: “terrible”, ...
- ....
Common ways people deal with hiccups
Common ways people deal with hiccups

Averages and Standard Deviation

Always Wrong!
Better ways people can deal with hiccups
Actually measuring percentiles

Requirements

Response Time
Percentile plot line
Requirements

Why we measure latency and response times to begin with...
Latency tells us how long something took

- But what do we WANT the latency to be?
- What do we want the latency to BEHAVE like?
- Latency requirements are usually a PASS/FAIL test of some predefined criteria
- Different applications have different needs
- Requirements should reflect application needs
- Measurements should provide data to evaluate requirements
The Olympics
aka “ring the bell first”

- Goal: Get gold medals
- Need to be faster than everyone else at SOME races
- Ok to be slower in some, as long as fastest at some (the average speed doesn’t matter)
- Ok to not even finish or compete (the worst case and 99%‘ile don’t matter)
- Different strategies can apply. E.g. compete in only 3 races to not risk burning out, or compete in 8 races in hope of winning two
Pacemakers
aka “hard” real time

- Goal: Keep heart beating
- Need to never be slower than X
- “Your heart will keep beating 99.9% of the time” is not reassuring
- Having a good average and a nice standard deviation don’t matter or help
- The worst case is all that matters
“Low Latency” Trading
aka “soft” real time

- Goal A: Be fast enough to make some good plays
- Goal B: Contain risk and exposure while making plays
  - E.g. want to “typically” react within 200 usec.
  - But can’t afford to hold open position for 20 msec, or react to 30 msec stale information

- So we want a very good “typical” (median, 50%‘ile)
- But we also need a reasonable Max, or 99.99%‘ile
Interactive applications
aka “squishy” real time

- **Goal:** Keep users happy enough to not complain/leave
- **Need to have** “typically snappy” behavior
- Ok to have occasional longer times, but not too high, and not too often
- **Example:** 90% of responses should be below 0.2 sec, 99% should be below 0.5 sec, 99.9 should be better than 2 seconds. And a >10 second response should never happen.
- **Remember:** A single user may have 100s of interactions per session...
Establishing Requirements
an interactive interview (or thought) process

Q: What are your latency requirements?
A: We need an avg. response of 20 msec
Q: Ok. Typical/average of 20 msec... So what is the worst case requirement?
A: We don’t have one
Q: So it’s ok for some things to take more than 5 hours?
A: No way in H%&%! 
Q: So I’ll write down “5 hours worst case…”
A: No. That’s not what I said. Make that “nothing worse than 100 msec”
Q: Are you sure? Even if it’s only two times a day?
A: Ok... Make it “nothing worse than 2 seconds…”
Establishing Requirements
an interactive interview (or thought) process

- Ok. So we need a typical of 20msec, and a worst case of 2 seconds. How often is it ok to have a 1 second response?

- A: (Annoyed) I thought you said only a few times a day

- Q: That was for the worst case. But if half the results are better than 20 msec, is it ok for the other half to be just short of 2 seconds? What % of the time are you willing to take a 1 second, or a half second hiccup? Or some other level?

- A: Oh. Let's see. We have to better than 50 msec 90% of the time, or we'll be losing money even when we are fast the rest of the time. We need to be better than 500 msec 99.9% of the time, or our customers will complain and go elsewhere

Now we have a service level expectation:

- 50% better than 20 msec
- 90% better than 50 msec
- 99.9% better than 500 msec
- 100% better than 2 seconds
Latency does not live in a vacuum
Remember this?

How much load can this system handle?

Sustainable Throughput Level

Where users complain

Where the sysadmin is willing to go

What the marketing benchmarks will say

[Graph showing response time vs. load or resource availability with three levels:
A. Good response time
B. Acceptable response time
C. Unacceptable (poor) response time]
Sustainable Throughput: The throughput achieved while safely maintaining service levels
Comparing behavior under different throughputs and/or configurations
Comparing latency behavior under different throughputs, configurations latency sensitive messaging distribution application.
Instance capacity test: “Fat Portal”
HotSpot CMS: Peaks at ~3GB / 45 concurrent users

* LifeRay portal on JBoss @ 99.9% SLA of 5 second response times
Instance capacity test: “Fat Portal”
C4: still smooth @ 800 concurrent users

* LifeRay portal on JBoss @ 99.9% SLA of 5 second response times
The coordinated omission problem

An accidental conspiracy...
The coordinated omission problem

Common Example A (load testing):
- build/buy load tester to measure system behavior
- each “client” issues requests one by one at a certain rate
- measure and log response time for each request
- results log used to produce histograms, percentiles, etc.

So what’s wrong with that?
- works well only when all responses fit within rate interval
- technique includes implicit “automatic backoff” and coordination
- But requirements interested in random, uncoordinated requests
The coordinated omission problem

Common Example B (monitoring):

- System monitors and records each transaction latency
- Latency measured between start and end of each operation
- Keeps observed latency stats of some sort (log, histogram, etc.)

So what’s wrong with that?

- Works correctly well only when no queuing occurs
- Long operations only get measured once
- Delays outside of timing window do not get measured at all
- Queued operations are measured wrong
Common Example B: Coordinated Omission in Monitoring Code

```java
/**
 * Performs the actual reading of a row out of the StorageService, fetching
 * a specific set of column names from a given column family.
 */

public static List<Row> read(List<ReadCommand> commands, ConsistencyLevel consistency_level)
    throws UnavailableException, IsBootstrappingException, ReadTimeoutException
{
    if (StorageService.instance.isBootstrapMode())
        throw new IsBootstrappingException();
    long startTime = System.nanoTime();
    List<Row> rows;
    try
    {
        rows = fetchRows(commands, consistency_level);
    }
    finally
    {
        readMetrics.addNano(System.nanoTime() - startTime);
    }
    return rows;
}
```

- Long operations only get measured once
- Delays outside of timing window do not get measured at all
How bad can this get?

System easily handles 100 requests/sec
Responds to each in 1msec

How would you characterize this system?

Overall Average response time is ~25 sec.

~50%‘ile is 1 msec  ~75%‘ile is 50 sec  99.99%‘ile is ~100sec
Measurement in practice

System easily handles 100 requests/sec

Responds to each in 1msec

Naïve Characterization

10,000 @ 1msec

1 @ 100 second

99.99%‘ile is 1 msec!

Average is 10.9msec!

Std. Dev. is 0.99sec!

(should be ~100sec)

(should be ~25 sec)
Proper measurement

System easily handles 100 requests/sec
Responds to each in 1msec

System Stalled for 100 Sec

10,000 results Varying linearly from 100 sec to 10 msec

10,000 results @ 1 msec each

~50%‘ile is 1 msec  ~75%‘ile is 50 sec  99.99%‘ile is ~100sec
Proper measurement

System easily handles 100 requests/sec
Responds to each in 1msec

10,000 results varying linearly from 100 sec to 10 msec

10,000 results @ 1 msec each

~50%‘ile is 1 msec    ~75%‘ile is 50 sec    99.99%‘ile is ~100sec
The coordinated omission problem

It is MUCH more common than you may think...
JMeter makes this mistake… (so do others)

Before Correction

After Correcting for Omission
The “real” world

99%‘ile MUST be at least 0.29% of total time (1.29% - 1%) which would be 5.9 seconds

26.182 seconds represents 1.29% of the total time wrong by a factor of 1,000x

Results were collected by a single client thread

[OVERALL], RunTime(ms), 2028755.0
[OVERALL], Throughput(ops/sec), 49291.31413108039
[UPDATE], Operations, 89999169
[UPDATE], AverageLatency(ms), 2.606116218695308
[UPDATE], MinLatency(ms), 0
[UPDATE], MaxLatency(ms), 26182
[UPDATE], 95thPercentileLatency(ms), 3
[UPDATE], 99thPercentileLatency(ms), 5
The "real" world

A world record SPECjEnterprise2010 result

<table>
<thead>
<tr>
<th>Response Times</th>
<th>Average</th>
<th>Std. dev.</th>
<th>Maximum</th>
<th>90th%</th>
<th>Regd 90th%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.211</td>
<td>0.63</td>
<td>284.121</td>
<td>0.280</td>
<td>2.000</td>
</tr>
<tr>
<td>Manage</td>
<td>0.133</td>
<td>0.52</td>
<td>298.800</td>
<td>0.210</td>
<td>2.000</td>
</tr>
<tr>
<td>Browse</td>
<td>0.260</td>
<td>0.82</td>
<td>300.078</td>
<td>0.320</td>
<td>2.000</td>
</tr>
<tr>
<td>CreateVehicleEJB</td>
<td>0.303</td>
<td>0.50</td>
<td>11.647</td>
<td>0.610</td>
<td>5.000</td>
</tr>
<tr>
<td>CreateVehicleWS</td>
<td>0.276</td>
<td>0.40</td>
<td>305.197</td>
<td>0.520</td>
<td>5.000</td>
</tr>
</tbody>
</table>

The max is 762 (!!!) standard deviations away from the mean.
305.197 seconds represents 8.4% of the timing run.
Real World Coordinated Omission effects

Before Correction

After Correction

Wrong by 7x
Real World Coordinated Omission effects

Duration by Percentile Distribution

Uncorrected Data

Duration (usec) vs Percentile

HotSpot Base
Real World Coordinated Omission effects

Corrected for Coordinated Omission

Uncorrected Data
Real World Coordinated Omission effects (Why I care)

A ~2500x difference in reported percentile levels for the problem that Zing eliminates

“other” JVM

Zing
Suggestions

- Whatever your measurement technique is, TEST IT.
- Run your measurement method against an artificial system that creates hypothetical pauses scenarios. See if your reported results agree with how you would describe that system behavior.
- Don’t waste time analyzing until you establish sanity.
- Don’t EVER use or derive from std. deviation.
- ALWAYS measure Max time. Consider what it means... Be suspicious.
- Measure %‘iles. Lots of them.
Some Tools
HdrHistogram
HdrHistogram

If you want to be able to produce graphs like this...

You need both good dynamic range and good resolution
HdrHistogram background

Goal: Collect data for good latency characterization...
- Including acceptable precision at and between varying percentile levels

Existing alternatives
- Record all data, analyze later (e.g. sort and get 99.9%‘ile).
- Record in traditional histograms

Traditional Histograms: Linear bins, Logarithmic bins, or Arbitrary bins
- Linear requires lots of storage to cover range with good resolution
- Logarithmic covers wide range but has terrible precisions
- Arbitrary is.... arbitrary. Works only when you have a good feel for the interesting parts of the value range
HdrHistogram

A High Dynamic Range Histogram
- Covers a configurable dynamic value range
- At configurable precision (expressed as number of significant digits)

For Example:
- Track values between 1 microsecond and 1 hour
- With 3 decimal points of resolution

Built-in [optional] compensation for Coordinated Omission

Open Source
- On github, released to the public domain, creative commons CC0
HdrHistogram

Fixed cost in both space and time
- Built with “latency sensitive” applications in mind
- Recording values does not allocate or grow any data structures
- Recording values uses a fixed computation to determine location (no searches, no variability in recording cost, FAST)
- Even iterating through histogram can be done with no allocation

Internals work like a “floating point” data structure
- “Exponent” and “Mantissa”
- Exponent determines “Mantissa bucket” to use
- “Mantissa buckets” provide linear value range for a given exponent. Each have enough linear entries to support required precision
Provides tools for iteration
- Linear, Logarithmic, Percentile

Supports percentile iterators
- Practical due to high dynamic range

Convenient percentile output
- 10% intervals between 0 and 50%  5%
- intervals between 50% and 75%  2.5%
- intervals between 75% and 87.5%
- Very useful for feeding percentile distribution graphs...

<table>
<thead>
<tr>
<th>Value, Percentile, TotalCountIncludingThisValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.003 0.000000000000000 7</td>
</tr>
<tr>
<td>0.057 0.000000000000000 807222</td>
</tr>
<tr>
<td>0.058 0.000000000000000 1235747</td>
</tr>
<tr>
<td>0.059 0.000000000000000 1694413</td>
</tr>
<tr>
<td>0.060 0.000000000000000 1994719</td>
</tr>
<tr>
<td>0.062 0.000000000000000 2373326</td>
</tr>
<tr>
<td>0.064 0.000000000000000 2620309</td>
</tr>
<tr>
<td>0.066 0.000000000000000 2795011</td>
</tr>
<tr>
<td>0.070 0.000000000000000 3036116</td>
</tr>
<tr>
<td>1.280 0.700000000000000 3228296</td>
</tr>
<tr>
<td>5.552 0.750000000000000 3458862</td>
</tr>
<tr>
<td>7.712 0.775000000000000 3574491</td>
</tr>
<tr>
<td>9.856 0.800000000000000 3689655</td>
</tr>
<tr>
<td>12.016 0.825000000000000 3805210</td>
</tr>
<tr>
<td>14.176 0.850000000000000 3920746</td>
</tr>
<tr>
<td>16.320 0.875000000000000 4036366</td>
</tr>
<tr>
<td>17.408 0.887500000000000 4094471</td>
</tr>
<tr>
<td>18.464 0.900000000000000 4150910</td>
</tr>
<tr>
<td>19.584 0.912500000000000 4209006</td>
</tr>
<tr>
<td>20.832 0.925000000000000 4267165</td>
</tr>
<tr>
<td>22.208 0.937500000000000 4324157</td>
</tr>
<tr>
<td>22.976 0.943750000000000 4352952</td>
</tr>
<tr>
<td>23.808 0.950000000000000 4381652</td>
</tr>
<tr>
<td>24.736 0.956250000000000 4418732</td>
</tr>
<tr>
<td>25.760 0.962500000000000 4439554</td>
</tr>
<tr>
<td>26.880 0.968750000000000 4467918</td>
</tr>
<tr>
<td>27.488 0.971875000000000 4482272</td>
</tr>
<tr>
<td>28.160 0.975000000000000 4496805</td>
</tr>
<tr>
<td>28.896 0.978125000000000 4511389</td>
</tr>
<tr>
<td>29.696 0.981250000000000 4525422</td>
</tr>
<tr>
<td>30.656 0.984375000000000 4539898</td>
</tr>
<tr>
<td>31.200 0.985937500000000 4547261</td>
</tr>
<tr>
<td>31.776 0.987500000000000 4554465</td>
</tr>
<tr>
<td>32.384 0.989062500000000 4561828</td>
</tr>
<tr>
<td>33.088 0.990625000000000 4569070</td>
</tr>
</tbody>
</table>
HdrHistogram

Latency by Percentile Distribution

Max=137.472

Latency (msec)

Percentile

0%  90%  99%  99.9%  99.99%  99.999%  99.9999%

©2012 Azul Systems, Inc.
jHiccup
Incontinuities in Java platform execution

Hiccups by Time Interval
- Max per Interval
- 99%
- 99.90%
- 99.99%
- Max

Hiccups by Percentile Distribution
Max=1665.024
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 jHiccup.jar as a java agent:
  - % java -javaagent=jHiccup.jar myApp myflags
- or attaching jHiccup to a running process:
  - % jHiccup -p <pid>
- Adds a background thread that samples time @ 1000/sec into an HdrHistogram

Open Source. Released to the public domain
Telco App Example

Max Time per interval

Optional SLA plotting

Hiccup duration at percentile levels
Fun with jHiccup

Charles Nutter  @headius

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 CMS, 1GB in an 8GB heap

Zing 5, 1GB in an 8GB heap
Oracle HotSpot CMS, 1GB in an 8GB heap

Zing 5, 1GB in an 8GB heap

Drawn to scale
Good for both

“squishy” real time
(human response times)

and

“soft” real time
(low latency software systems)
Oracle HotSpot (pure newgen)

Hiccups by Time Interval

- Max per Interval
- 99%
- 99.90%
- 99.99%
- Max

Hiccup Duration (msec)

Elapsed Time (sec)

Zing

Hiccups by Time Interval

- Max per Interval
- 99%
- 99.90%
- 99.99%
- Max

Hiccup Duration (msec)

Elapsed Time (sec)

Hiccups by Percentile Distribution

Max=22.656

Hiccup Duration (msec)

Percentile

Max=1.568

Hiccup Duration (msec)

Percentile

Low latency trading application
Low latency - Drawn to scale
Shameless bragging
Zing

- A JVM for Linux/x86 servers
- ELIMINATES Garbage Collection as a concern for enterprise applications
- Very wide operating range: Used in both low latency and large scale enterprise application spaces
- Decouples scale metrics from response time concerns
  - Transaction rate, data set size, concurrent users, heap size, allocation rate, mutation rate, etc.
- Leverages elastic memory for resilient operation
What is Zing good for?

- If you have a server-based Java application
- And you are running on Linux
- And you use using more than ~300MB of memory
- Then Zing will likely deliver superior behavior metrics
Where Zing shines

- **Low latency**
  - Eliminate behavior blips down to the sub-millisecond-units level

- **Machine-to-machine “stuff”**
  - Support higher *sustainable* throughput (the one that meets SLAs)

- **Human response times**
  - Eliminate user-annoying response time blips. Multi-second and even fraction-of-a-second blips will be completely gone.
  - Support larger memory JVMs *if needed* (e.g. larger virtual user counts, or larger cache, in-memory state, or consolidating multiple instances)

- **“Large” data and in-memory analytics**
  - Make batch stuff “business real time”. Gain super-efficiencies.
Takeaways

- Standard Deviation and application latency should never show up on the same page...
- If you haven’t stated percentiles and a Max, you haven’t specified your requirements
- Measuring throughput without latency behavior is [usually] meaningless
- Mistakes in measurement/analysis can cause orders-of-magnitude errors and lead to bad business decisions
- jHiccup and HdrHistogram are pretty useful
- The Zing JVM is cool...
Q & A

http://www.azulsystems.com

http://www.jhiccup.com

http://giltene.github.com/HdrHistogram