Enabling Java applications for low-latency use cases at scale with Azul Zing and GridGain

Gil Tene  
CTO & Co-Founder  
Azul Systems

Denis Magda  
VP, Product Management  
GridGain Systems
10 Mins That Saved Southwest Airlines
Apps That Require Much Lower Latency

Payments Processing

- Latency: 20 - 200 ms

Electronic Trading

- Latency: 20 - 100s μs
Garbage Collection Might Make Things Unpredictable
Unless You Select The Right Java Stack
Azul Zing - Java without the pauses
An overview of Azul Zing

Gil Tene, CTO & co-Founder, Azul Systems
A simple visual summary

This is <Your App> on HotSpot

This is <Your App> on Zing

Any Questions?
Azul Zing

- A JVM for Linux/x86 (servers, clouds, containers)
- "Not just Fast. Always Fast."
- Improves application behavior metrics
- Increases practical carrying capacity
- Makes developers and their managers happier
- Delivers a continuously responsive execution platform
- ELIMINATES Garbage Collection as a concern
- Reduces negative impacts of frequent code deployment
- VERY wide operating range
- from GBs to TBs, from low latency to streaming and batch
Areas where Azul Zing shines

- Wherever speed & responsiveness matter:
  - Human response times...
  - Machine-to-machine “stuff”...
  - “Low latency” or “Latency Sensitive”...
  - “Large” data and in-memory analytics...
Azul Zing shines in Java based infrastructure...

- Cassandra
- ... (ellipsis)
- Kafka
- ... (ellipsis)
- HBase
- ... (ellipsis)
- HDFS
- ... (ellipsis)
- Zookeeper
- ... (ellipsis)
- Aeron
- ... (ellipsis)
- Solr
- ... (ellipsis)
- Lucene
- ... (ellipsis)
- Elastic
- ... (ellipsis)
- GridGain
- Ignite
- ... (ellipsis)
- Flink
- ... (ellipsis)
- Pinot
- Spark
- Storm
Zing shines in Java applications

API Gateways

Front End

Back end

Streaming applications

Application containers

In memory analytics
Azul Zing’s main feature areas

- **C4:** GC, solved.
- **Falcon:** Powerful JIT compiler. Speed.
- **ReadyNow:** Warmup/Startup. DevOps.
Speed

What is it good for?
Are you fast?
Are you fast when new code rolls out?
Are you fast when it matters?
Are you fast at Market Open?
Are you reliably fast?
What does being “fast” mean?

Speed

Time
What does being “fast” mean?
Speed in the Java world…
Response time
(with contribution by optimization level)
Speed
(with contribution by optimization level)
Falcon is basically about speed

Speed
(with contribution by optimization level)
ReadyNow is focused on warmup
C4 takes out the stalls

Speed
(with contribution by optimization level)

ReadyNow

Falcon

C4 takes out the stalls

Interpreted  Tier1 (profiling)  Optimized  Optimized (Zing)
Start Fast, Go Fast, Stay Fast

Speed
(with contribution by optimization level)

ReadyNow

Falcon

C4

Interpreted  Tier1 (profiling)  Optimized  Optimized (Zing)
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 more GC tuning flags

Source: Word Cloud created by Frank Pavageau in his Devoxx FR 2012 presentation titled "Death by Pauses"
The complete guide to modern GC tuning**

```
java -Xmx40g
java -Xmx20g
java -Xmx10g
java -Xmx5g
```

** It's 2019, Zing is widely available. Tweaking 10s of GC flags is a thing of the past.
Cassandra under heavy load, Intel E5-2690 v4 server

Yup, that’s the 1 msec mark
A real world use case with In Memory Computing:

GridGain in a Credit Card payments processing application
Payments Benchmark: Configuration

- 3 nodes GridGain cluster
  - 3 x AWS i3en.6xlarge
  - 72 cores
  - 600 GB RAM and 45 TB disk

- Tested Scenarios
  - Azul Zing C4 vs. OpenJDK G1 for
  - 100% in RAM, no disk (200 GB)
  - 100% in RAM, 100% on disk (200 GB)
  - 30% in RAM, 100% on disk (600 GB)
Payments Benchmark: Workload

- Each transaction accesses 20 records

- Distributed Transactional Reads
  - Target throughput - 1000 reads/sec
  - Target latency - 15ms for 99.99th percentile

- Distributed Transactional Updates
  - Target throughput - 2000 updates/sec
  - Target latency - 50ms for 99.99th percentile
  - RAM and disk have to be updated for primary and backup copies

- Metrics Collection
  - Micrometer and jHiccup
  - 2 hours run
Transactional Reads
100% in RAM (200 GB)
Transactional Reads
100% in RAM (200 GB) [equalized scale]
Transactional Reads
100% in RAM (200 GB) [equalized scale]
Transactional Updates:
100% in RAM (200 GB)
Transactional Updates
100% in RAM (200 GB) [equalized scale]
Transactional Updates
100% in RAM (200 GB) [equalized scale]
Transactional Reads With Persistence
100% in RAM, 100% on Disk (200 GB)
Transactional Reads With Persistence
100% in RAM, 100% on Disk (200 GB) [equalized scale]
Transactional Reads With Persistence
100% in RAM, 100% on Disk (200 GB) [equalized scale]
Transactional Updates With Persistence
100% in RAM, 100% on Disk (200 GB)
Transactional Updates With Persistence
100% in RAM, 100% on Disk (200 GB) [equalized scale]
Transactional Updates With Persistence
100% in RAM, 100% on Disk (200 GB) [equalized scale]
GridGain - In-Memory Computing Platform That Scales
GridGain Let’s Us Scale To Terabytes
Across RAM and Disk Space

Unlimited off-heap memory and disk space for data

Java Heap for objects generated in runtime
Transactional Persistence

- **Distributed Persistence Tier**
  - Fully transactional and consistent
  - No need to cache 100% of data in RAM
  - No need to warm-up RAM on restarts

- **Performance vs. Cost Tradeoff**
  - Cache more for fastest performance
  - Cache less to reduce infrastructure costs
Transactional Reads with Persistence
30% in RAM, 100% on Disk (600 GB)
Transactional Reads with Persistence
30% in RAM, 100% on Disk (600 GB) [equalized scale]
Transactional Reads with Persistence
30% in RAM, 100% on Disk (600 GB) [equalized scale]
Transactional Updates with Persistence
30% in RAM, 100% on Disk (600 GB)
Transactional Updates with Persistence
30% in RAM, 100% on Disk (600 GB) [equalized scale]
Transactional Updates with Persistence
30% in RAM, 100% on Disk (600 GB) [equalized scale]
Is Java Ready for Low-Latency Scenarios?

- Eliminate GC pauses with Azul Zing
- Scale Out with GridGain across RAM and Disk
- Select a configuration you need to meet infrastructure costs
Q&A

Gil - @giltene

Denis - @denimagda