In-Memory Computing SUMMIT NORTH AMERICA 2019

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

AZUL SYSTEMS GridGain

10 Mins That Saved Southwest Airlines

Apps That Require Much Lower Latency

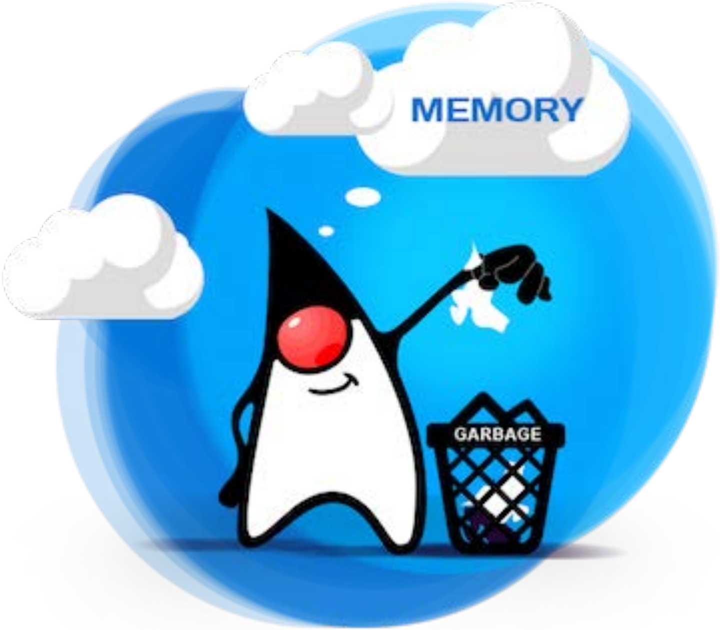
Payments Processing

Latency: 20 - 200 ms

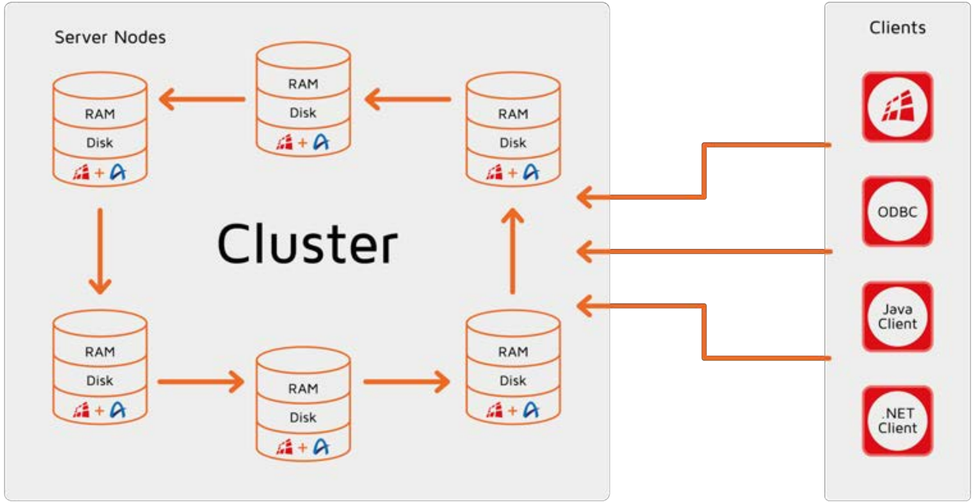
Electronic Trading

Latency: 20 - 100s μs

Garbage Collection Might Make Things Unpredictable



Unless You Select The Right Java Stack



Azul Zing - Java without the pauses

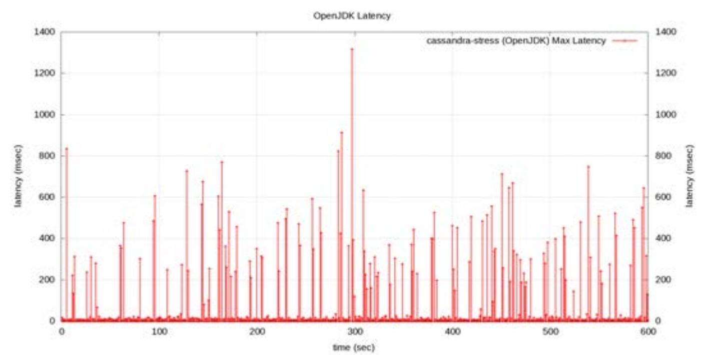
Click to add text

An overview of Zing

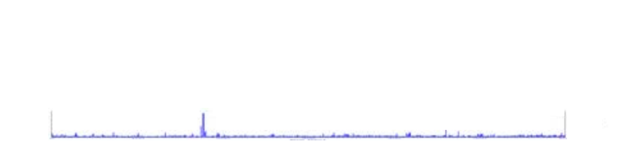
Gil Tene, CTO & co-Founder, Azul Systems

A simple visual summary

This is <Your App> on HotSpot



This is <Your App> on Zing



Any Questions?

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 Zing shines

- Wherever speed & responsiveness matter:

- Human response times…

- Machine-to-machine “stuff”…

- “Low latency” or “Latency Sensitive"…

- “Large” data and in-memory analytics…

Zing shines in
Java based infrastructure…

Solr

Cassandra

Elastic

…

…

Lucene

Kafka

GridGain

…

Ignite

…

…

HBase

Zookeeper

…

Flink

HDFS

…

Pinot

…

…

Aeron

Spark

Storm

Zing shines in Java applications

API Gateways

…

Application containers

…

Back end

…

Front End

…

…

In memory analytics

Streaming applications

…

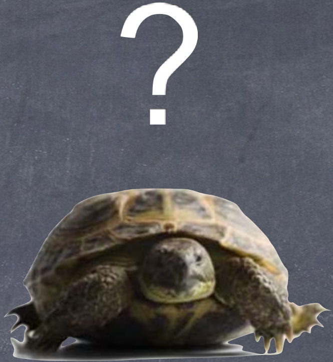
…

Zing's main feature areas

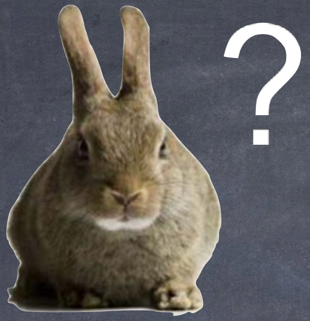
- C4: GC, solved.
- Falcon: Powerful JIT compiler. Speed.
- ReadyNow: Warmup/Startup. DevOps.

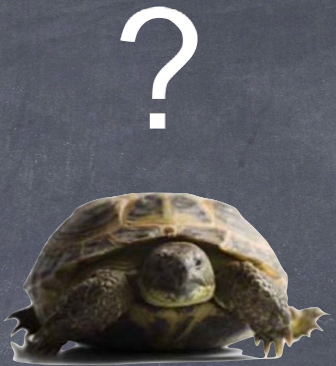
Speed

What is it good for?



Are you fast?



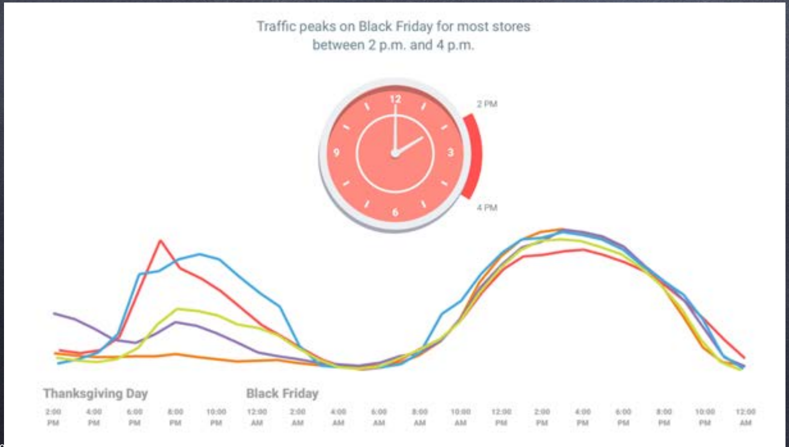


Are you fast when new code rolls out?

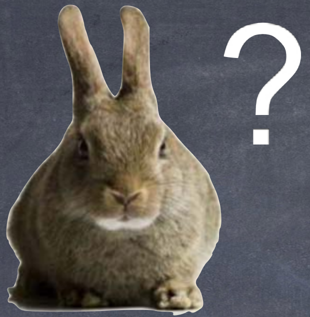
?

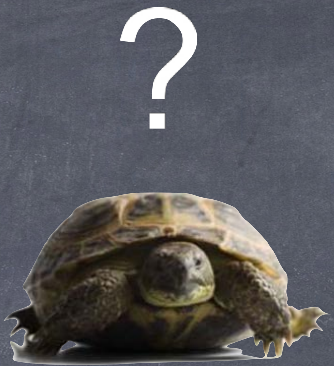
?

Are you fast when it matters?



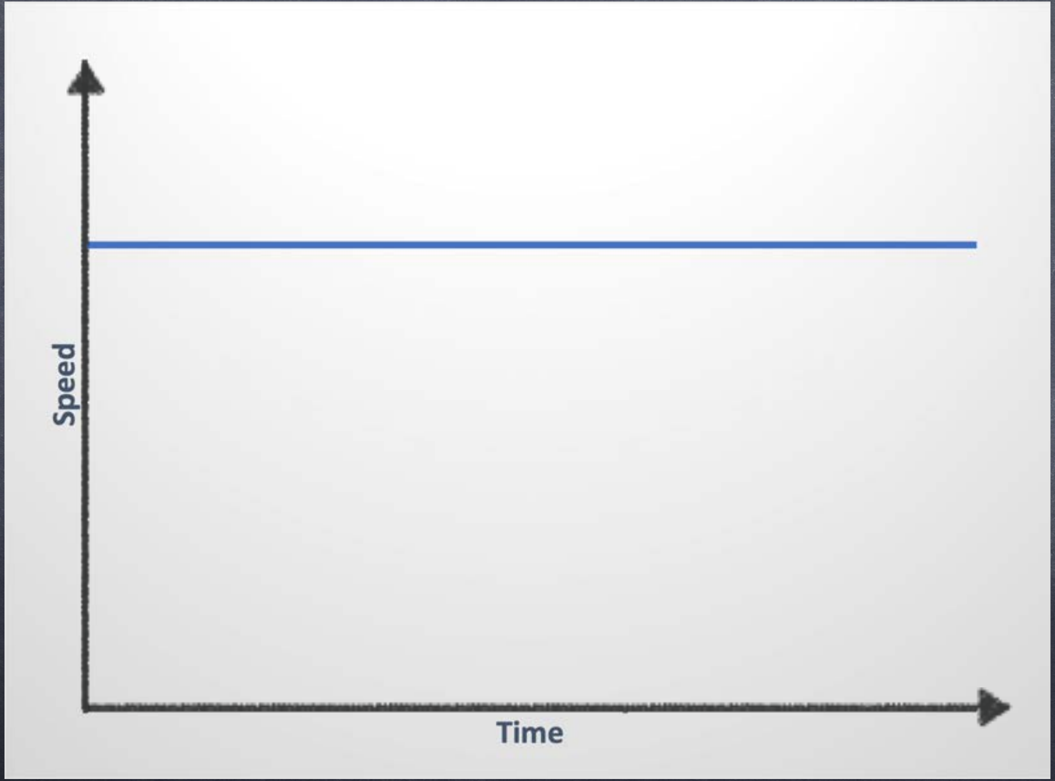
Are you fast at Market Open?





Are you reliably fast?

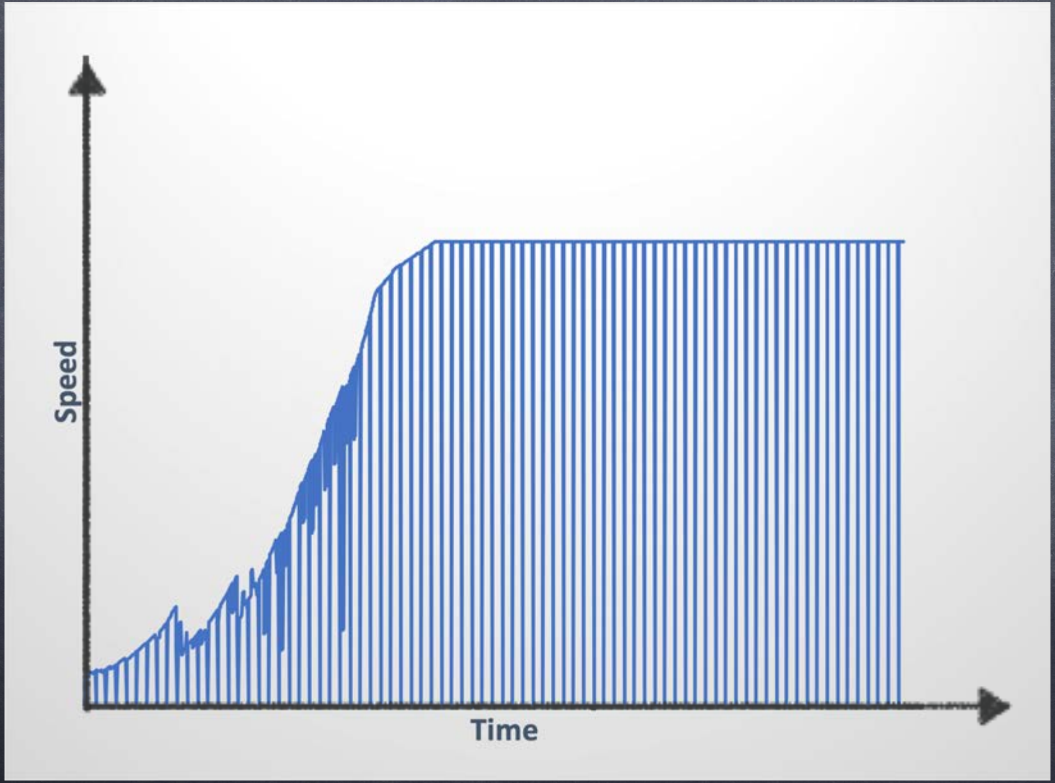
What does being “fast” mean?



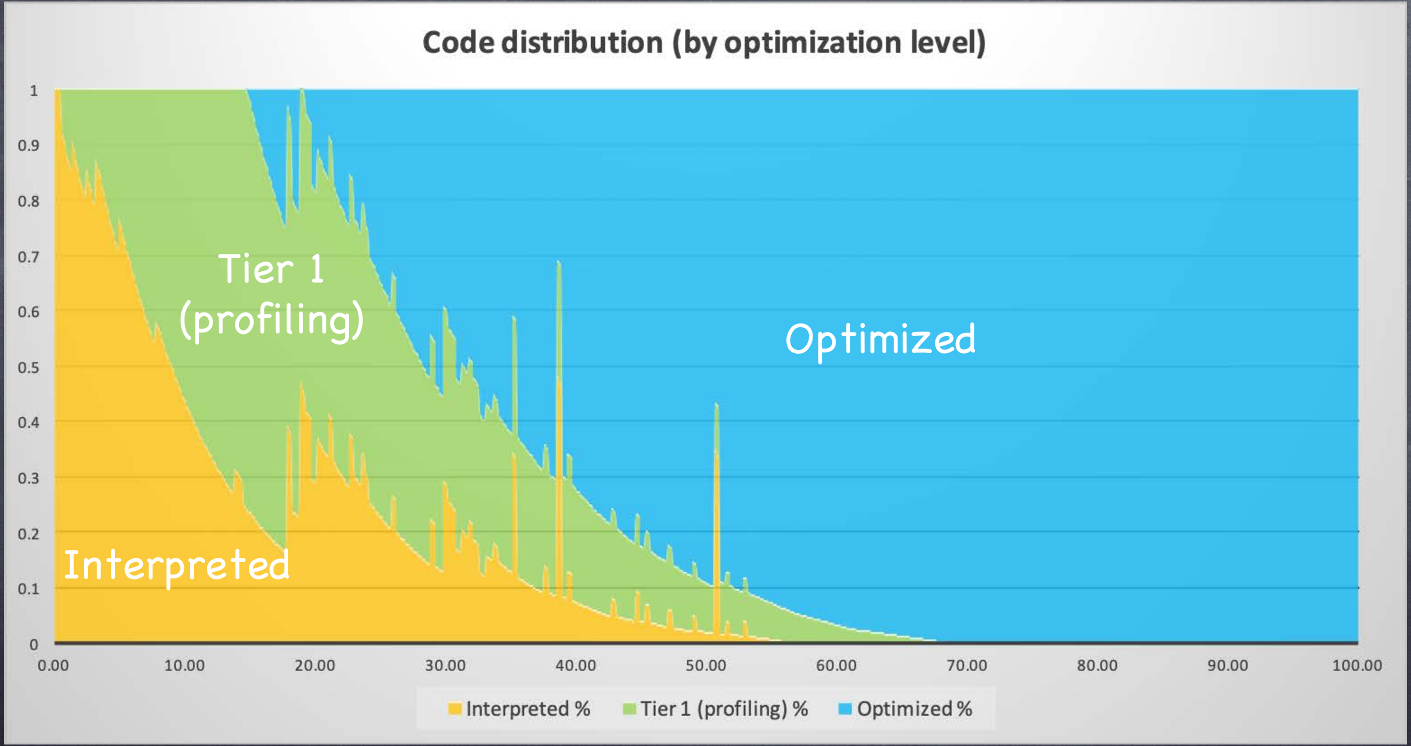
??

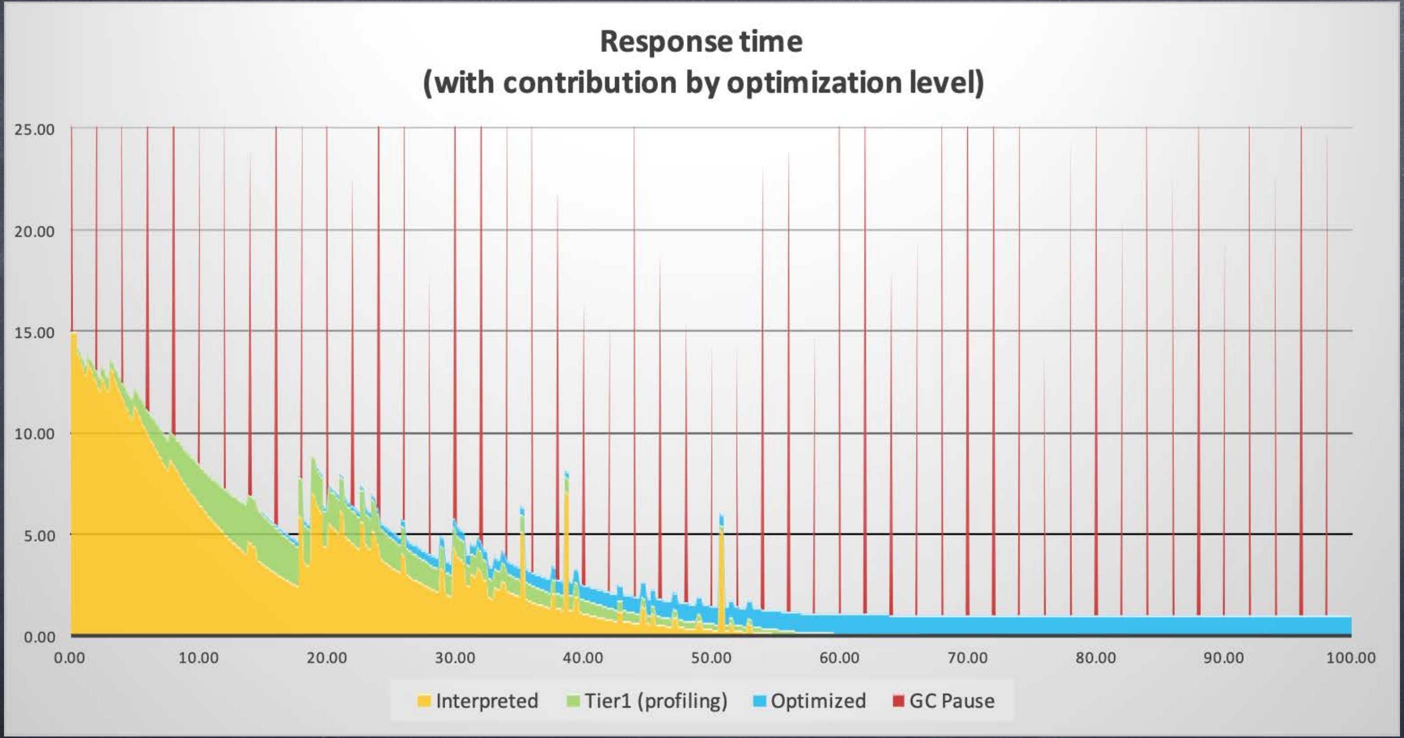
©2017 Azul Systems, Inc.

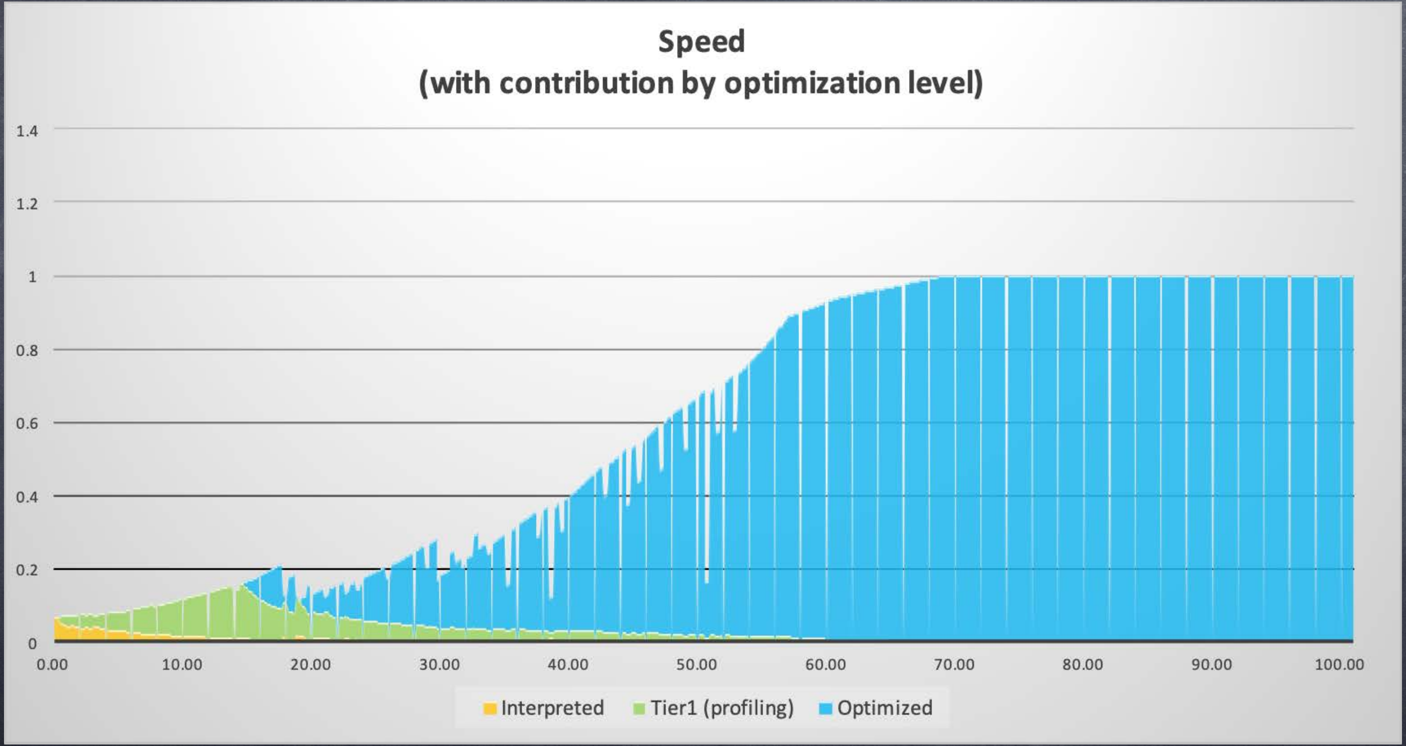
What does being “fast” mean?



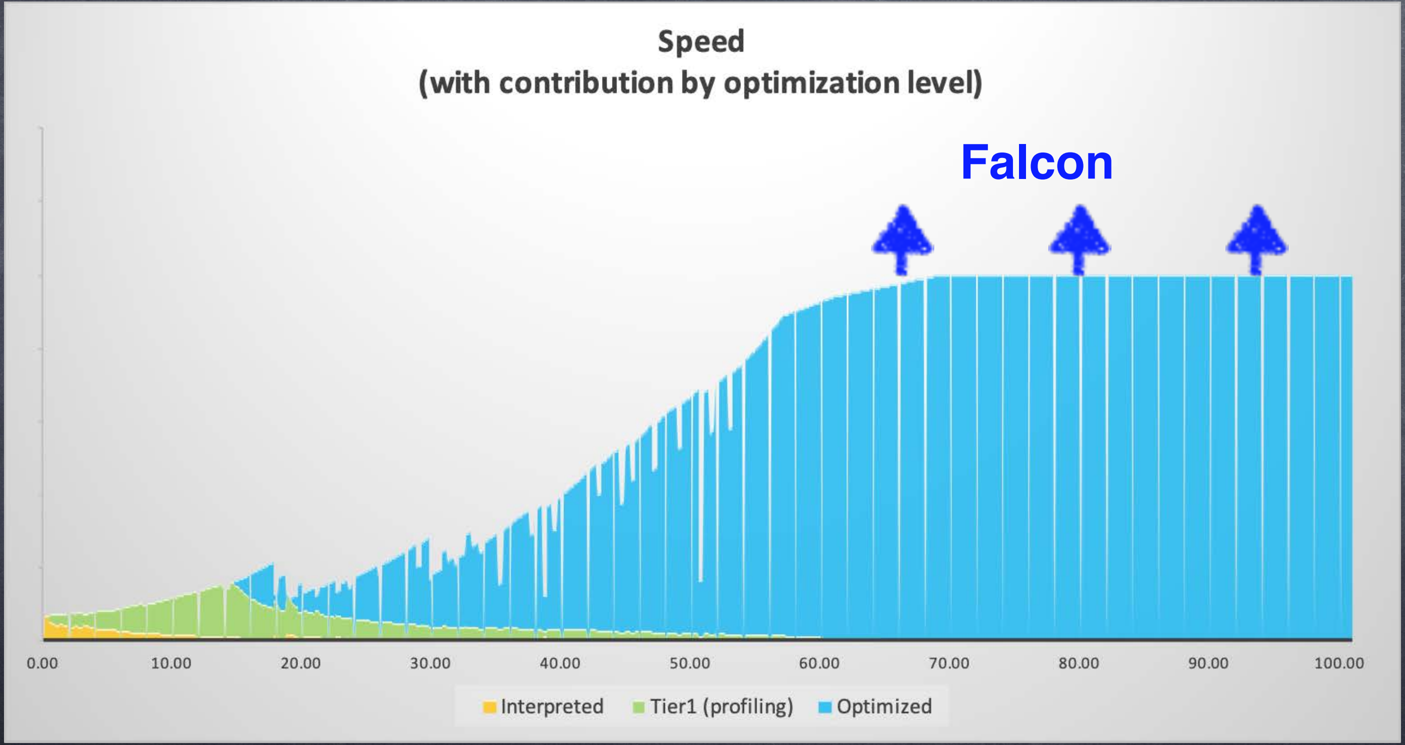
Speed in the Java world…


Code distribution (by optimization level)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.00
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00
Tier 1 (profiling)
Optimized
Interpreted
🟨 Interpreted %
🟩 Tier 1 (profiling) %
🟦 Optimized %

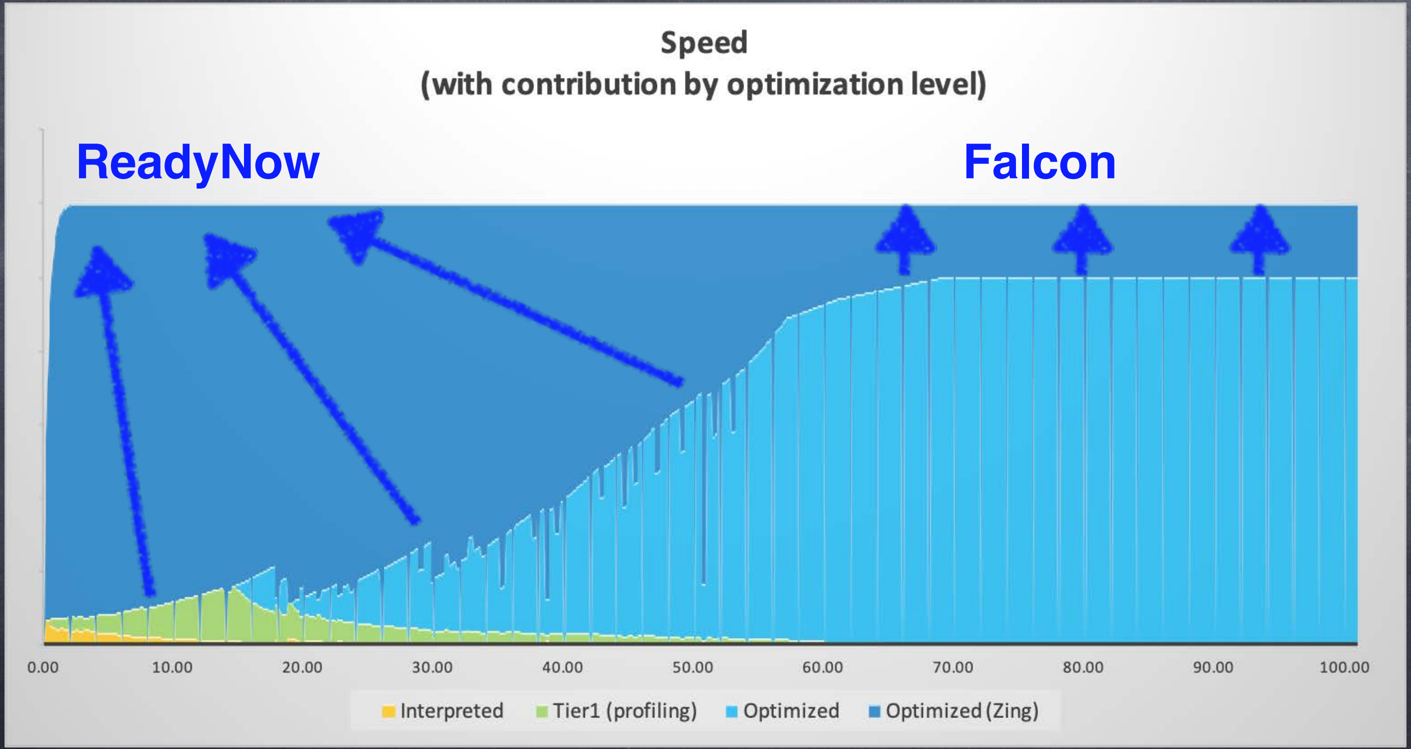
Response time
(with contribution by optimization level)
25.00
20.00
15.00
10.00
5.00
0.00
0.00
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00
Interpreted
Tier1 (profiling)
Optimized
GC Pause


Speed
(with contribution by optimization level)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.00
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00
Interpreted
Tier1 (profiling)
Optimized

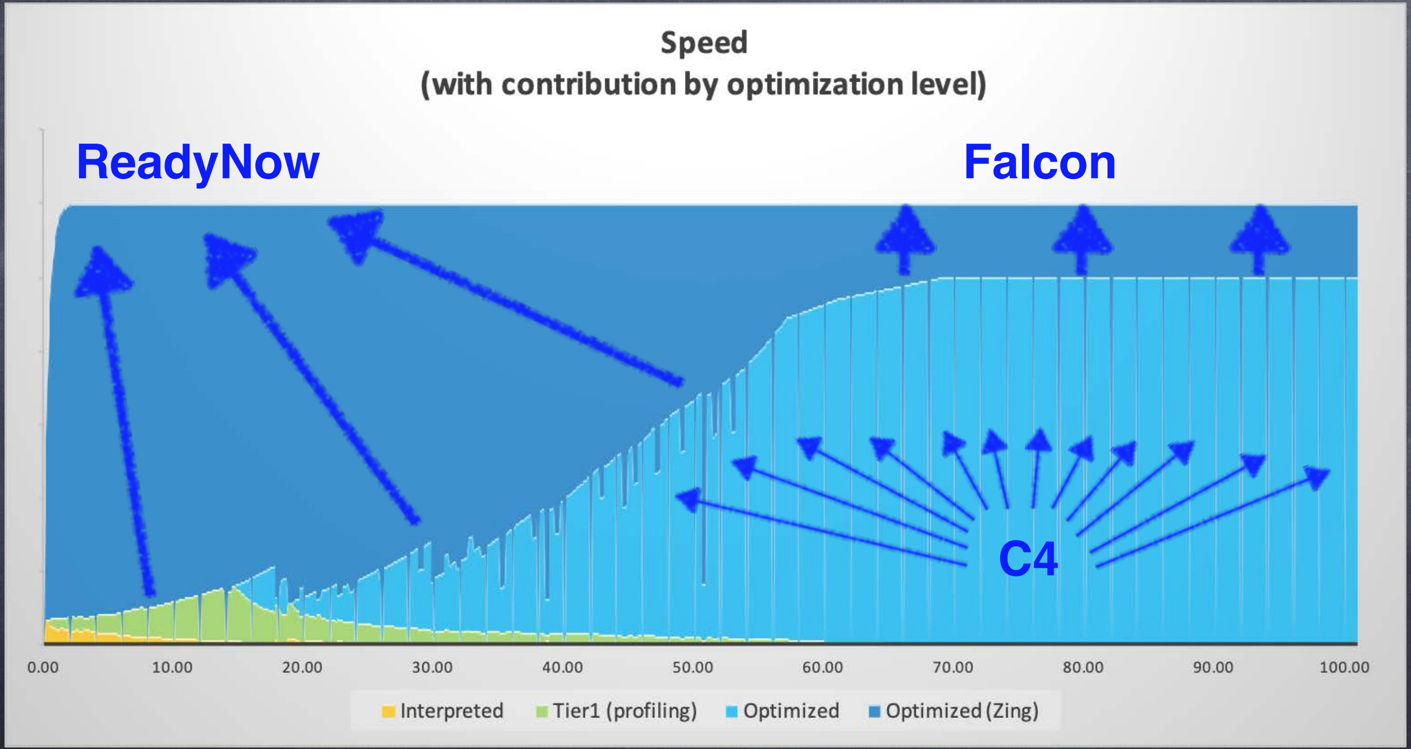
Falcon is basically about speed



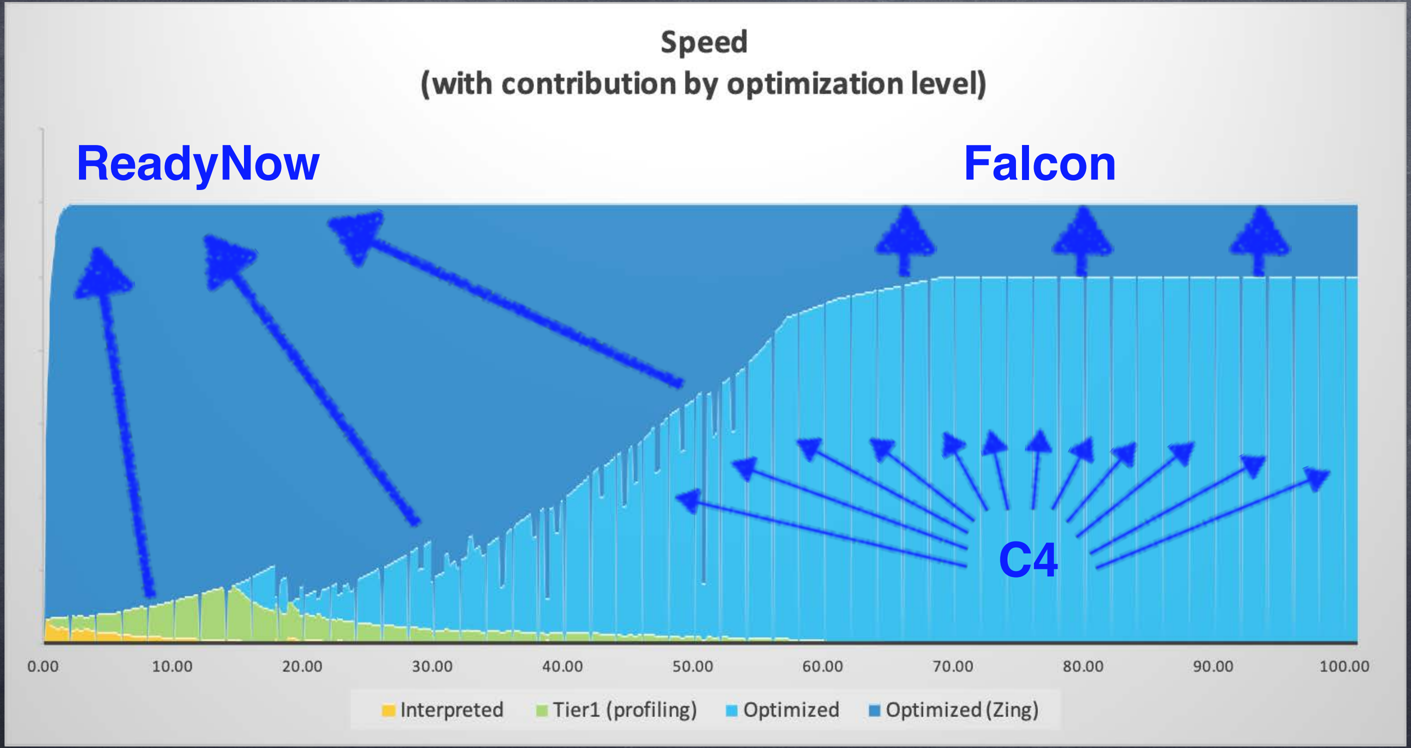
ReadyNow is focused on warmup



C4 takes out the stalls



Start Fast, Go Fast, Stay Fast



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

CMSOldPLABMax
CMSLoopWarn
CMSScavengeBeforeRemark
CMSPrecleanNumerator
CMSRemarkVerifyVariant
GCTimeLimit
CMSParallelRemarkEnabled
CMSOldPLABToleranceFactor
HeapBaseMinAddress
PrintHeapAtGCExtended
CMSExtrapolateSweep
CMSPrecleanSurvivors2
GCPauseIntervalMillis
CMSSplitIndexedFreeListBlocks
MinPermHeapExpansion
ExplicitGCInvokesConcurrentAndUnloadsClasses
PrintClassHistogramBeforeFullGC
GCTimeRatio
CMSSmallCoalSurplusPercent
CMSLargeCoalSurplusPercent
MaxNewSize
CMSIndexedFreeListReplenish
CMSYoungGenPerWorker
HeapFirstMaximumCompactionCount
Xms
server
verbose:gc
CMSInitiatingPermOccupancyFraction
CMSClassUnloadingEnabled
CMSTriggerRatio
PrintClassHistogramAfterFullGC
HeapDumpOnOutOfMemoryError
CMSLargeSplitSurplusPercent
CMSOldPLABReactivityCeiling
BindGCTaskThreadsToCPUs
ParGCUseLocalOverflow
MaxGCPauseMillis
loggc
CMSCleanOnEnter
CMSYield
ProtectJavaHeap
CMSPrecleanSurvivors1
CMSWaitDuration
PrintReferenceGC
CMSIncrementalOffset
PrintHeapAtGC
CMSConcurrentMTEnabled
CMSIncrementalDutyCycle
CMSOldPLABReactivityFactor
CMSFullGCsBeforeCompaction
CMSSmallSplitSurplusPercent
CMSCoordinatorYieldSleepCount
CMSPrecleaningEnabled
MaxHeapSize
CMSPermGenPrecleaningEnabled
CMSReplenishIntermediate
G1HeapRegionSize
ParGCArrayScanChunk
HeapMaximumCompactionInterval
CMSPLABRecordAlways
PrintGCDateStamps
CMSPrintChunksInDump
CMSScheduleRemarkEdenPenetration
CMSMaxAbortablePrecleanLoops
UseParNewGC
CMSAbortablePrecleanMinWorkPerIteration
NewSize
UseConcMarkSweepGC
CMSDumpAtPromotionFailure
CMSMaxAbortablePrecleanTime
UncommitYoungGenOnGC
MaxPermHeapExpansion
GCLockerInvokesConcurrent
AutoGCSelectPauseMillis
PermSize
CMSOldPLABNumRefills
CMSPrecleanIter
InitialHeapSize
CMSOldPLABMin
MinHeapDeltaBytes
CMSExpAvgFactor
CMSParPromoteBlocksToClaim
ScavengeBeforeFullGC
HeapDumpBeforeFullGC
CMSOldPLABResizeQuicker
CMSBootstrapOccupancy
CMSRevisitStackSize
UncommitOldGenOnGC
CMSYieldSleepCount
CMSPrintObjectsInDump
CMSIncrementalMode
ParGCTrimOverflow
ConcGCThreads
MaxGCMinorPauseMillis
MaxHeapFreeRatio
CMSConcMarkMultiple
CMSBitMapYieldQuantum
AdaptivePermSizeWeight
CMSWorkQueueDrainThreshold
CMSIncrementalDutyCycleMin
NewSizeThreadIncrease
CMSClassUnloadingMaxInterval
CMSScheduleRemarkSamplingRatio
CMSIncrementalSafetyFactor
CMSScheduleRemarkEdenSizeThreshold
PrintGCApplicationConcurrentTime
CMSPermGenSweepingEnabled
PrintGC
PrintGCTaskTimeStamps
PrintGCTimeStamps
GCDrainStackTargetSize
PrintCMSStatistics
CMSAbortSemantics
CMSPrecleanRefLists2
CMSSamplingGrain
HeapDumpAfterFullGC
ExplicitGCInvokesConcurrent
CMSTriggerPermRatio
CMSInitiatingOccupancyFraction
CMSCompactWhenClearAllSoftRefs
HeapDumpPath
AdaptiveSizeMajorGCDecayTimeScale
LargePageHeapSizeThreshold
GCTaskTimeStampEntries
PrintHeapAtSIGBREAK
ErgoHeapSizeLimit
PermGenPadding
InitiatingHeapOccupancyPercent
DisableExplicitGC
MaxPermSize
NewRatio
CMSIncrementalPacing
GCLockerEdenExpansionPercent
ParGCDesiredObjsFromOverflowList
Xmx
GCOverheadReportingPeriodMS
CMSAbortablePrecleanWaitMillis
CMSIsTooFullPercentage
PrintGCApplicationStoppedTime
PermMarkSweepDeadRatio
PrintCMSInitiationStatistics
CMSPrecleanDenominator
CMSPrecleanThreshold
GCOverheadReporting
GCHeapFreeLimit
PrintGCDetails
CMSPrecleanRefLists1
CMSRescanMultiple
MinHeapFreeRatio
CMSUseOldDefaults

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

The complete guide to modern GC tuning**

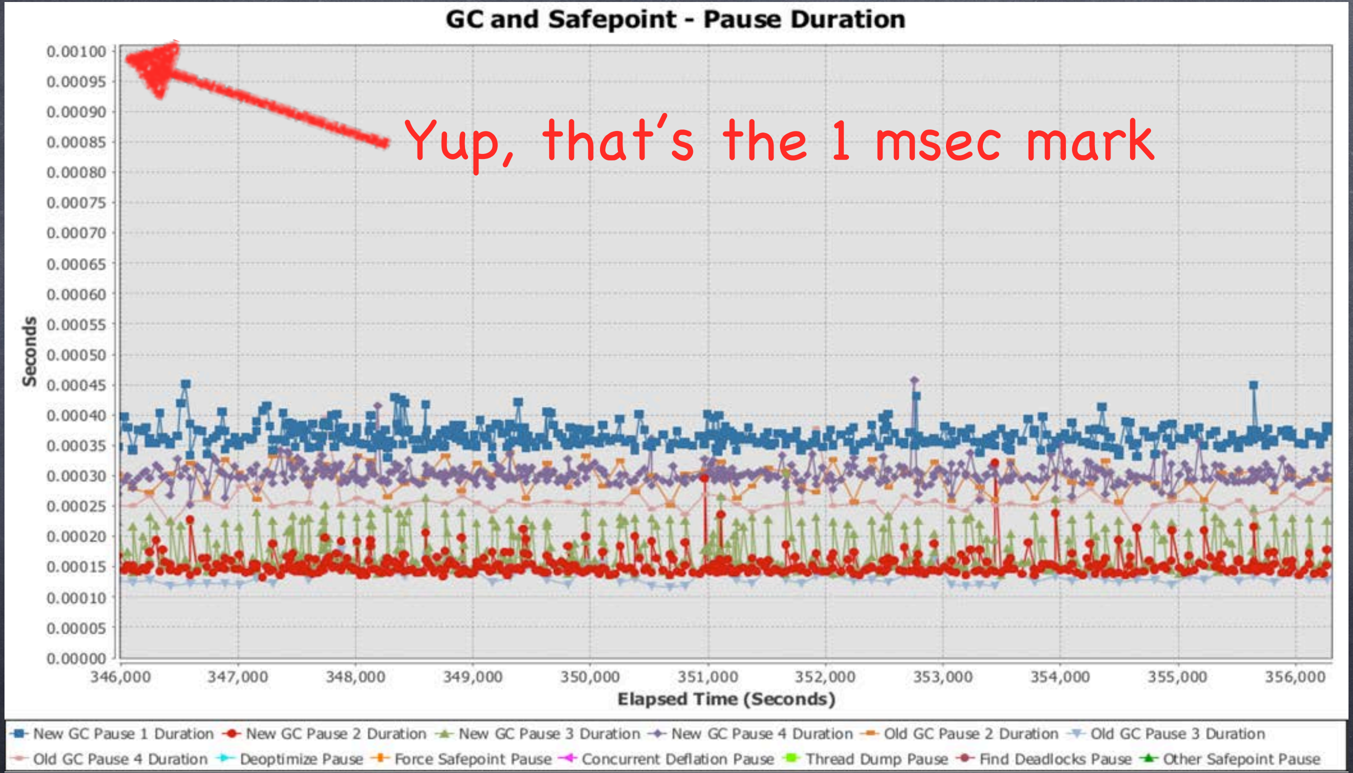
```
java -Xmx40g
java -Xmx20g

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

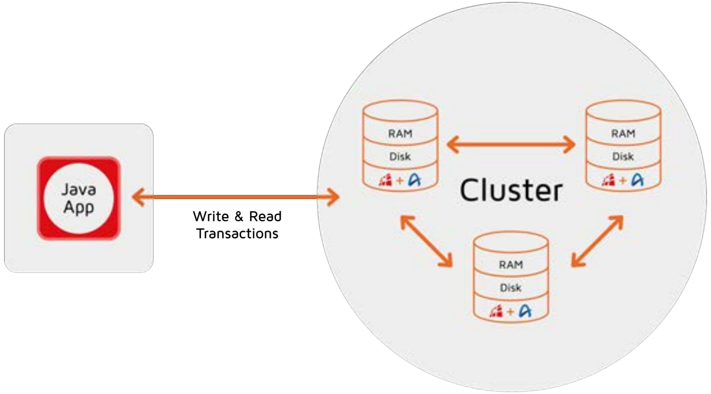
A real world use case with In Memory Computing:

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)

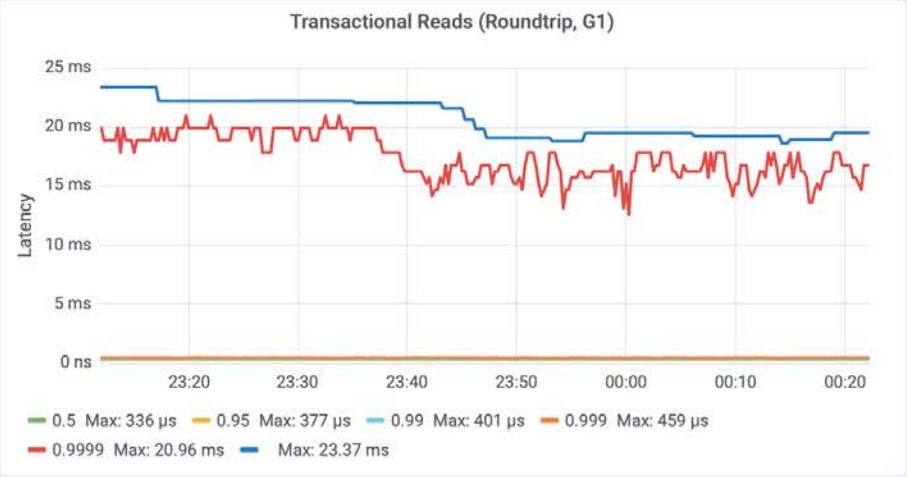


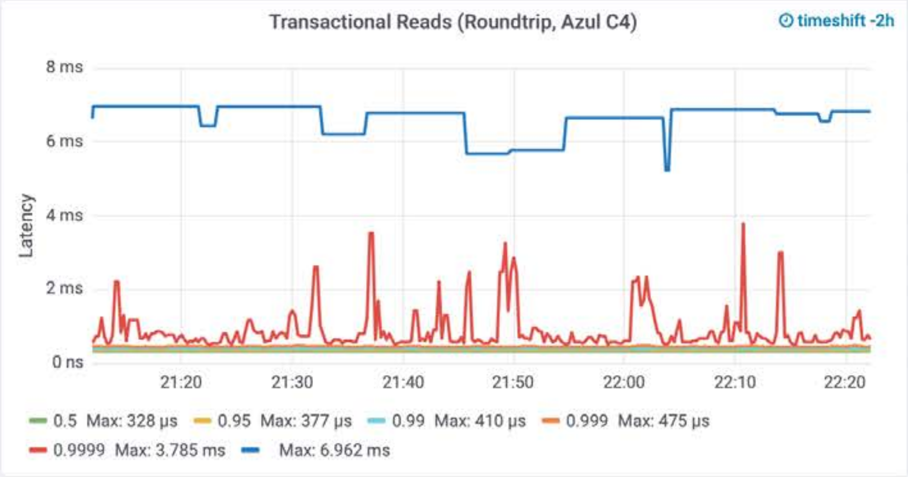
Payments Benchmark: Workload

- Each transactions 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

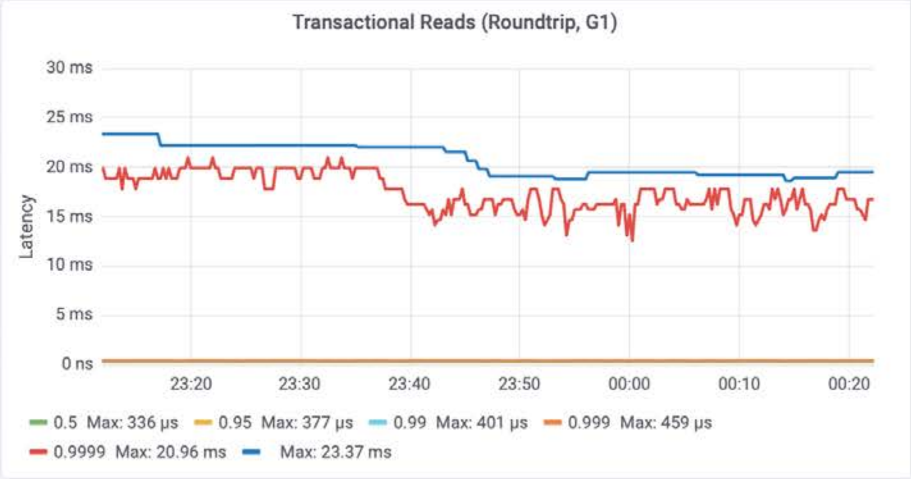
100% in RAM (200 GB)

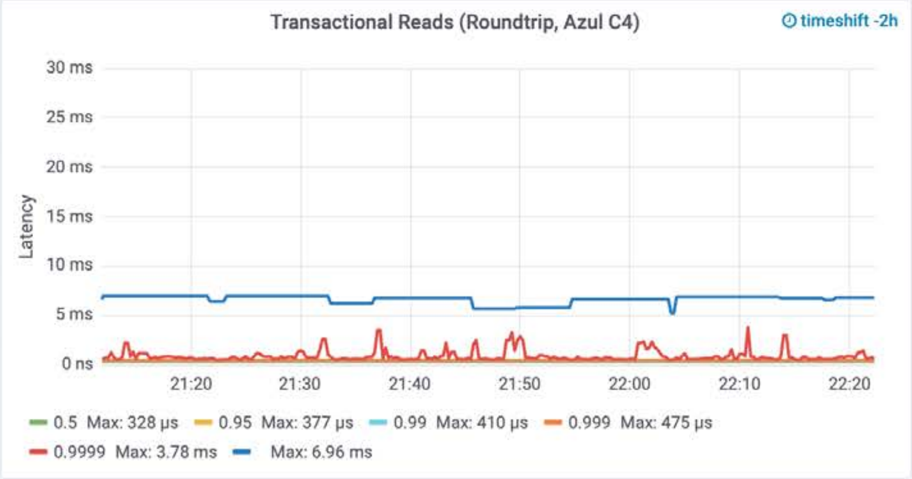




Transactional Reads

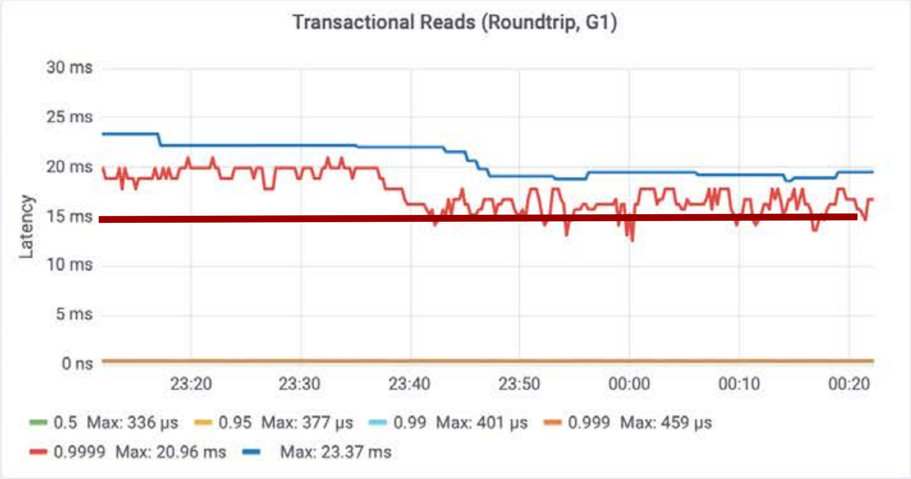
100% in RAM (200 GB) [equalized scale]

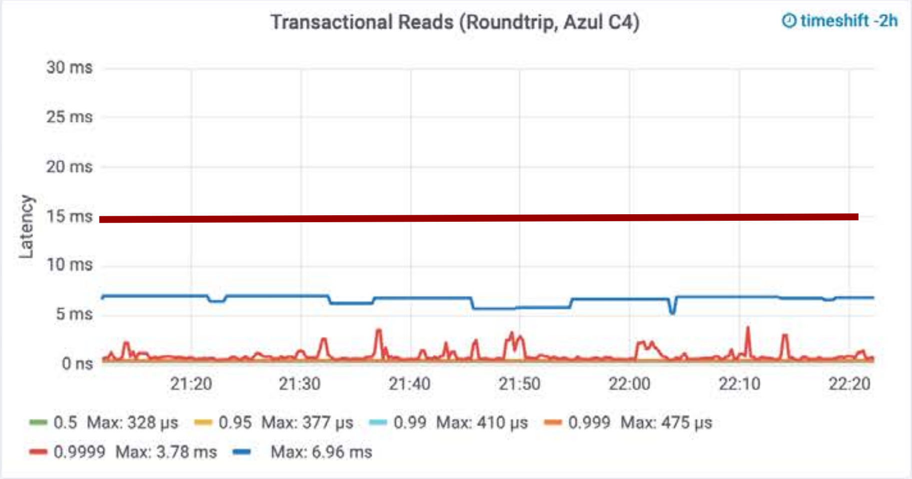




Transactional Reads

100% in RAM (200 GB) [equalized scale]

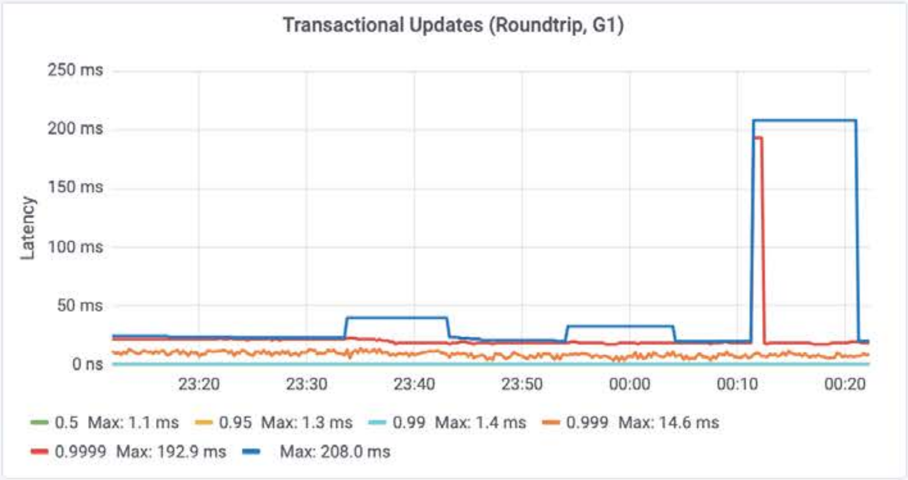


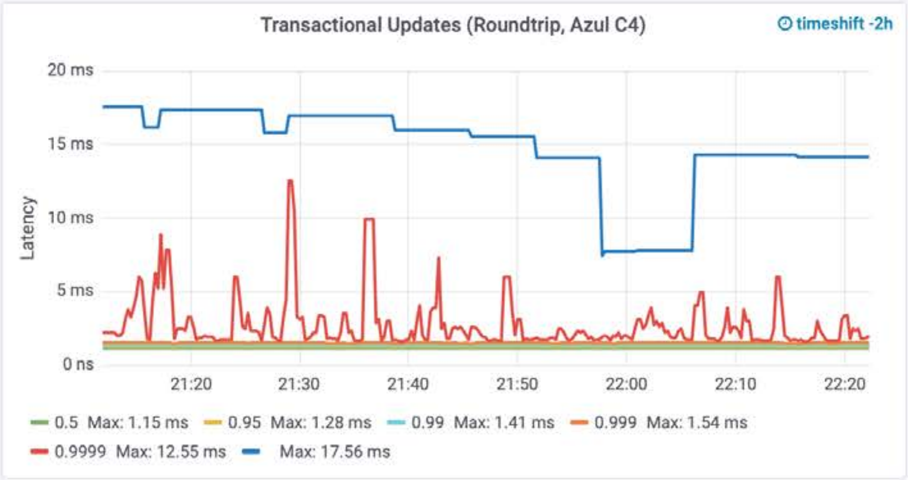


━━━ - target latency

Transactional Updates:

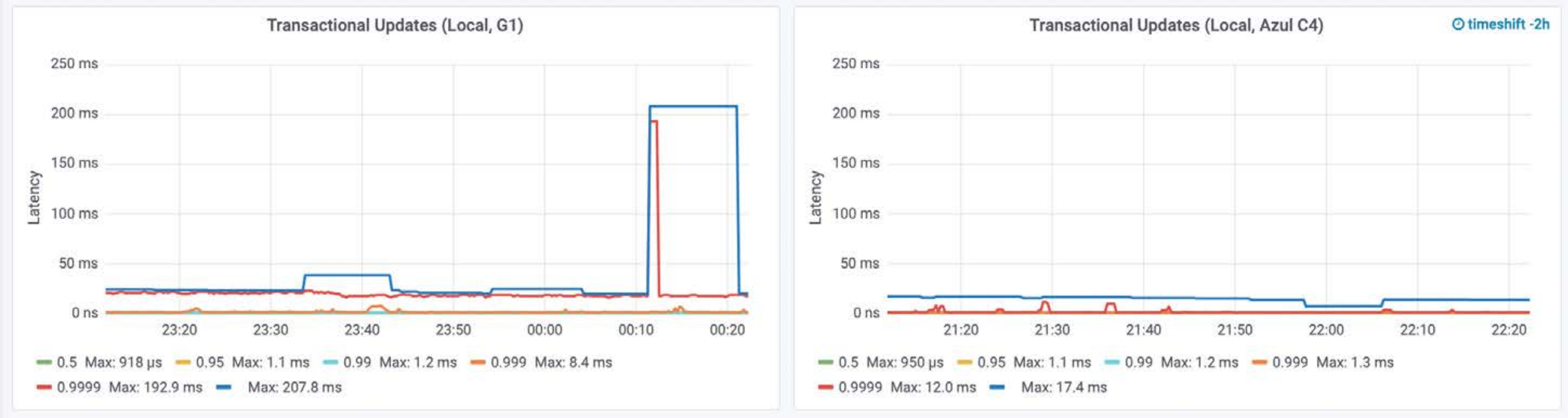
100% in RAM (200 GB)





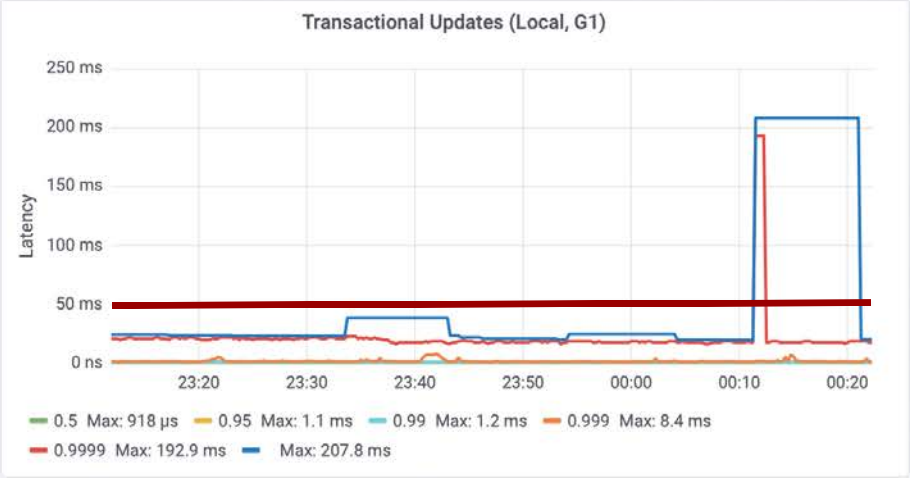
Transactional Updates

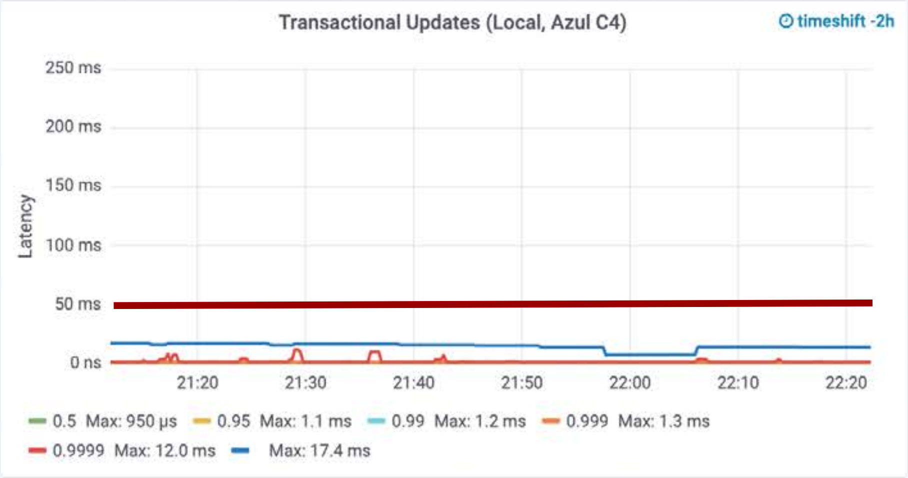
100% in RAM (200 GB) [equalized scale]



Transactional Updates

100% in RAM (200 GB) [equalized scale]

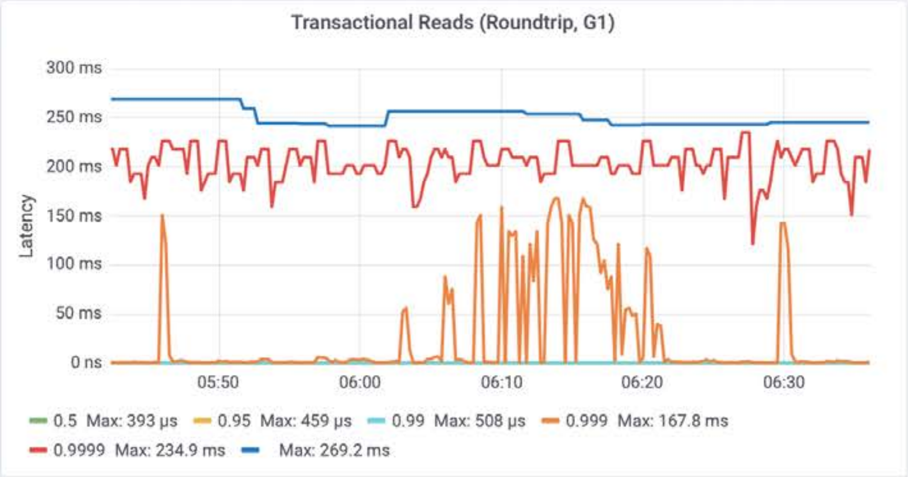


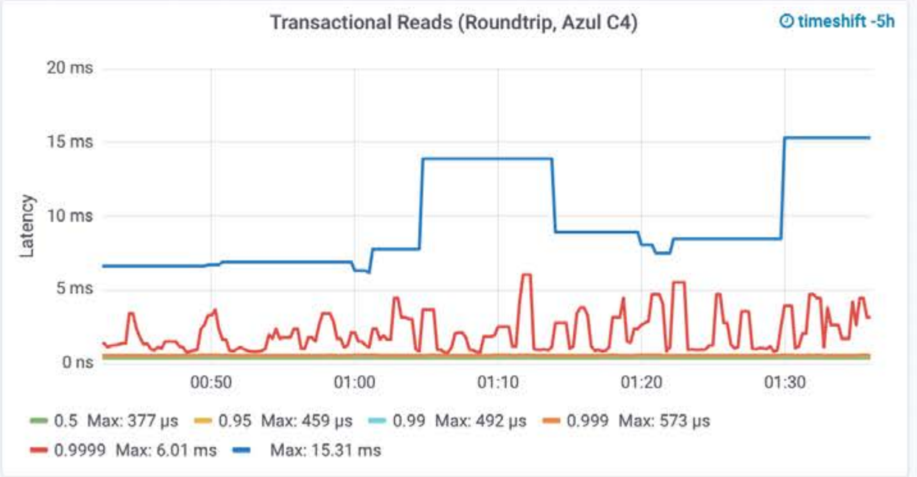


▬ **- target latency**

Transactional Reads With Persistence

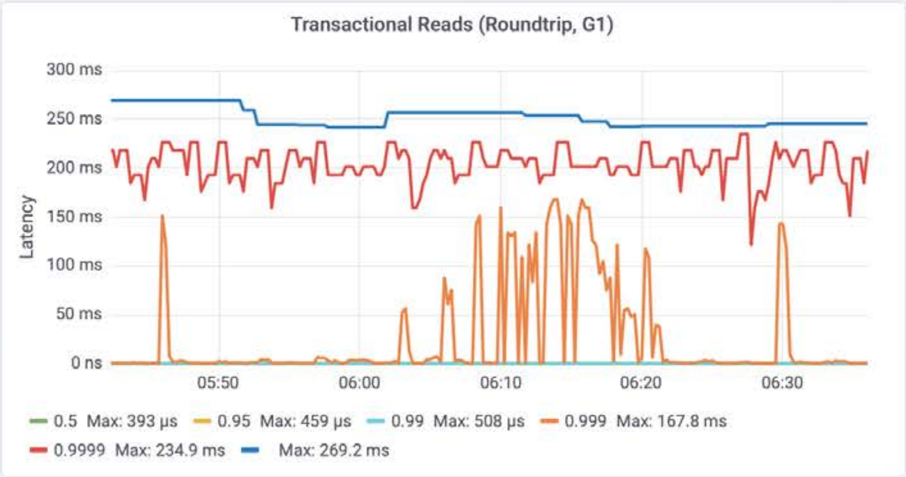
100% in RAM, 100% on Disk (200 GB)

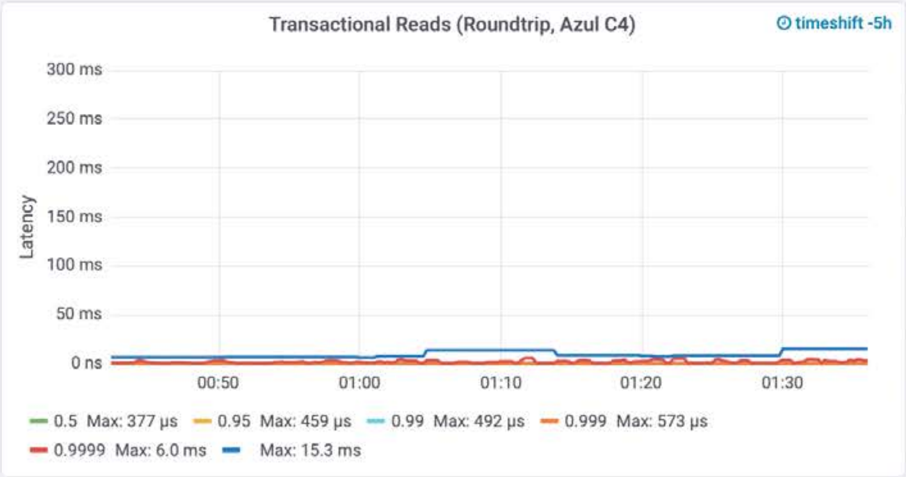




Transactional Reads With Persistence

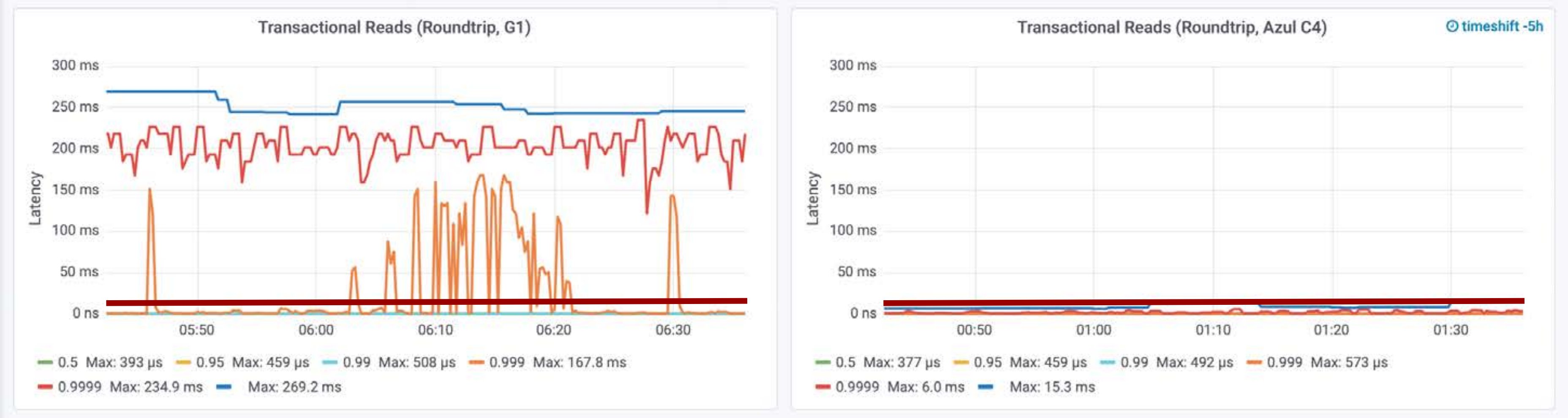
100% in RAM, 100% on Disk (200 GB) [equalized scale]





Transactional Reads With Persistence

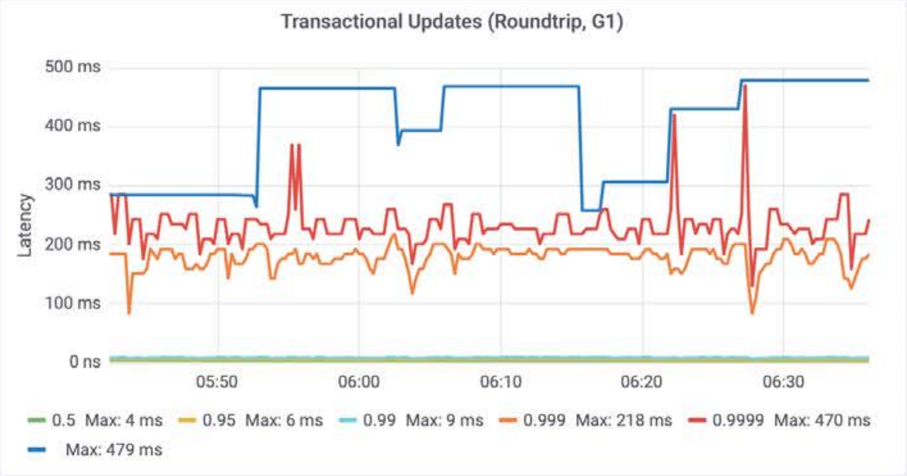
100% in RAM, 100% on Disk (200 GB) [equalized scale]

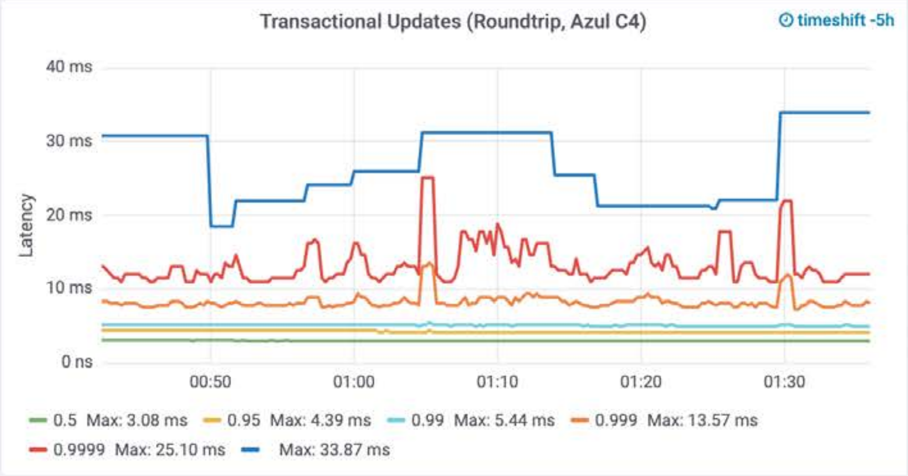


— - target latency

Transactional Updates With Persistence

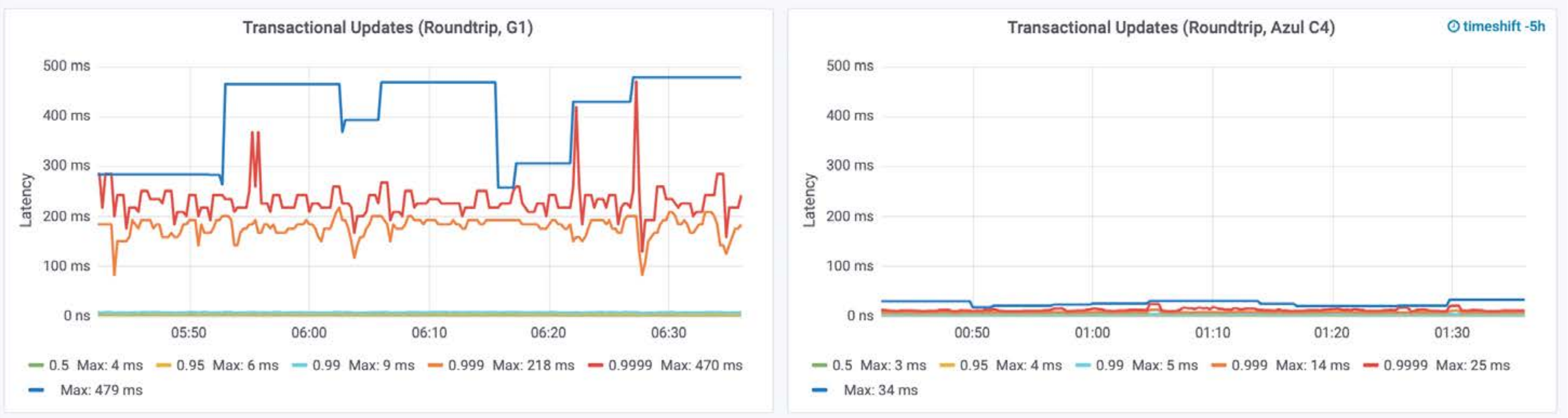
100% in RAM, 100% on Disk (200 GB)





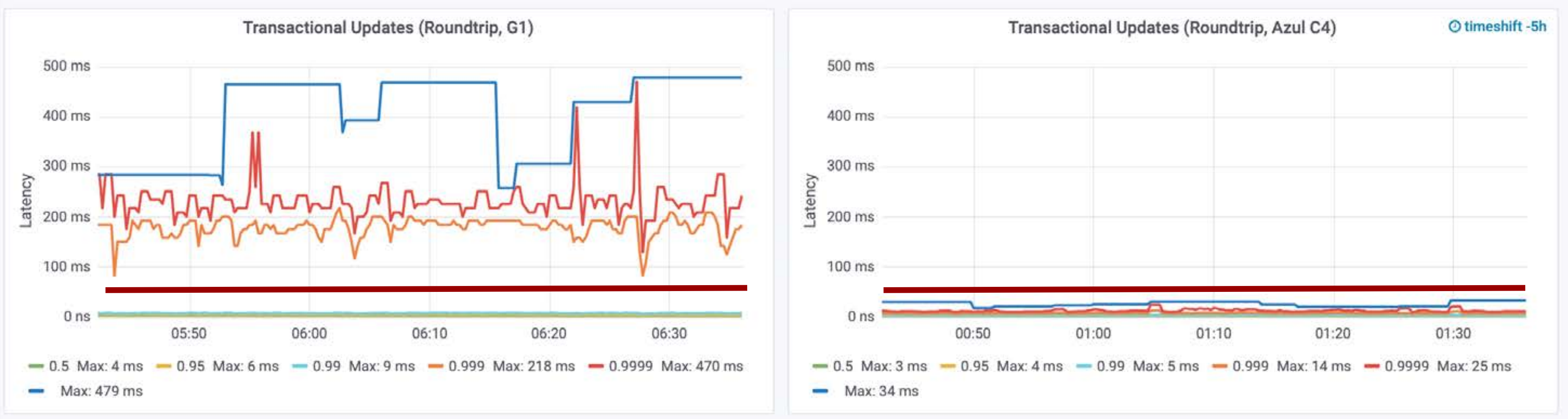
Transactional Updates With Persistence

100% in RAM, 100% on Disk (200 GB) [equalized scale]



Transactional Updates With Persistence

100% in RAM, 100% on Disk (200 GB) [equalized scale]



— - target latency

GridGain - In-Memory Computing Platform That Scales

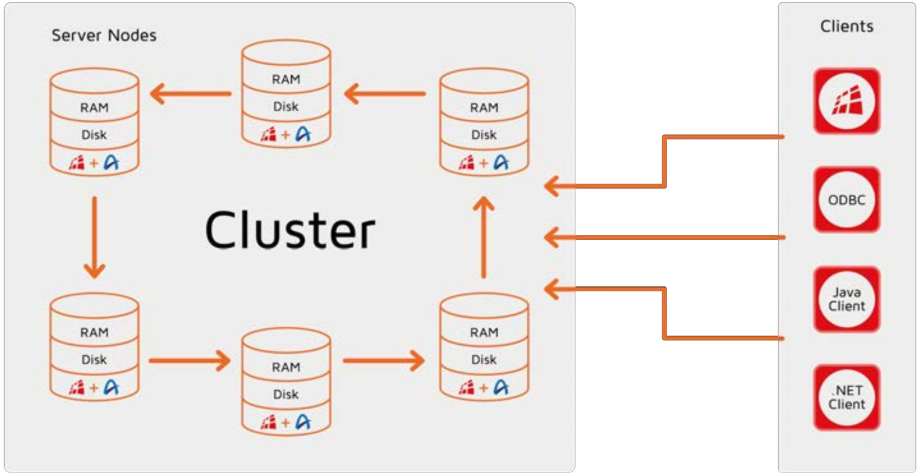
Click to add text

In-Memory Computing SUMMIT NORTH AMERICA 2019

GridGain Let’s Us Scale To Terabytes

Across RAM and Disk Space

Unlimited off-heap memory and disk space for data

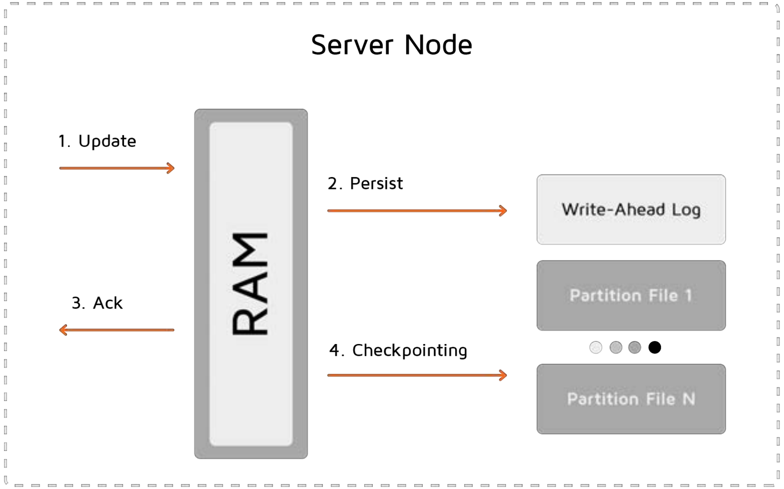


Java Heap for objects generated in runtime

Transactional Persistence

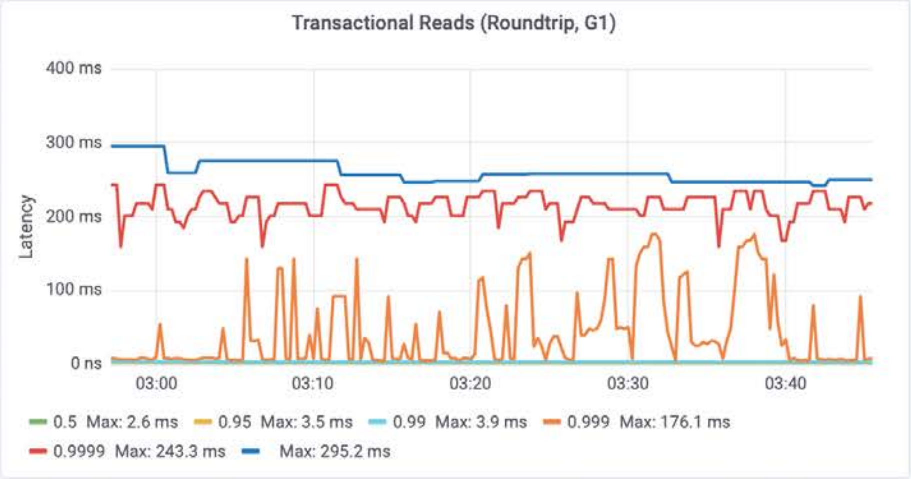
- Distributed Persistence Tier
  - Fully transactional and consistent
  - No need to cache 100% of data in RAM
  - No need to warm-up RAM on restarts

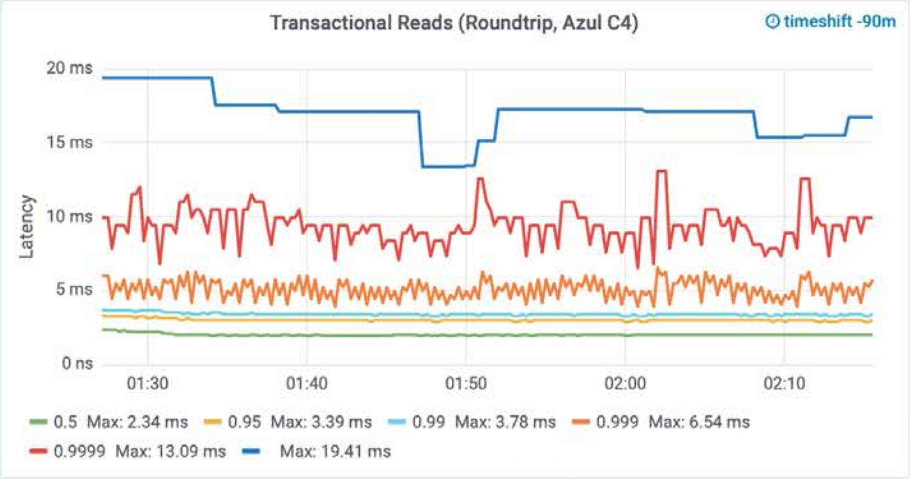
- Performance vs. Cost Tradeoff
  - Cache more for fastest performance
  - Cache less to reduce infrastructure costs



Transactional Reads with Persistence

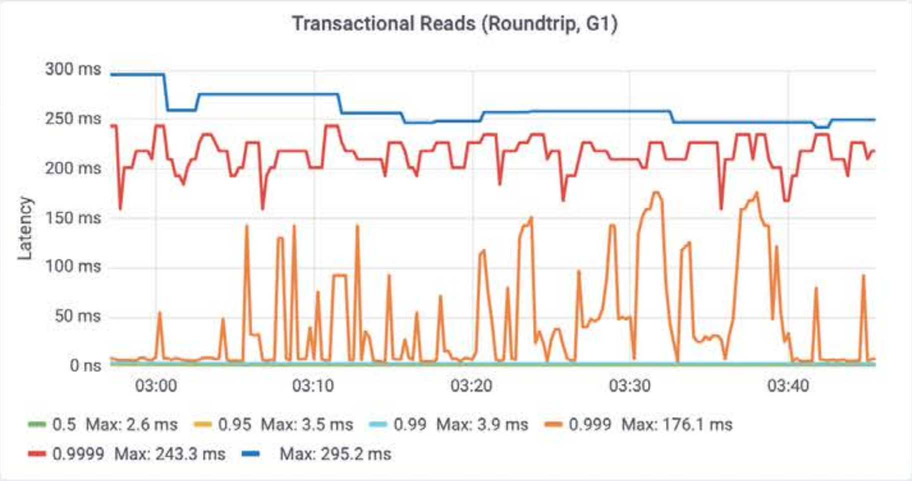
30% in RAM, 100% on Disk (600 GB)

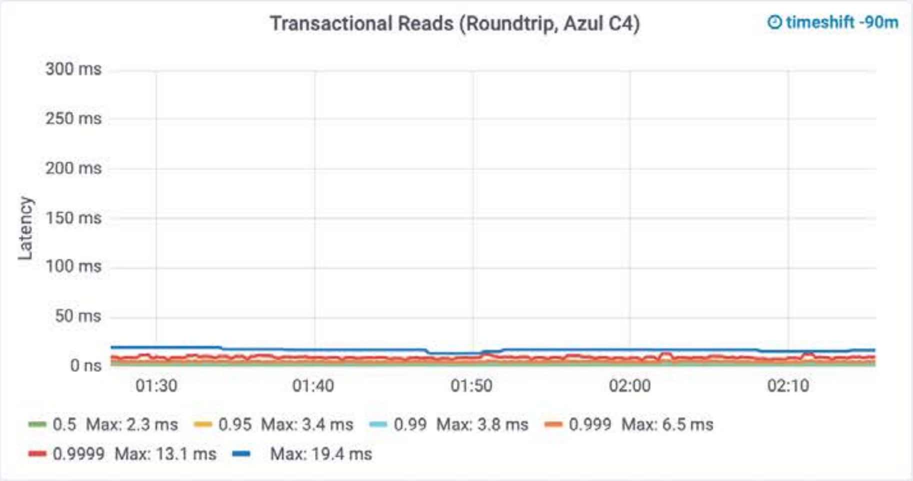




Transactional Reads with Persistence

30% in RAM, 100% on Disk (600 GB) [equalized scale]

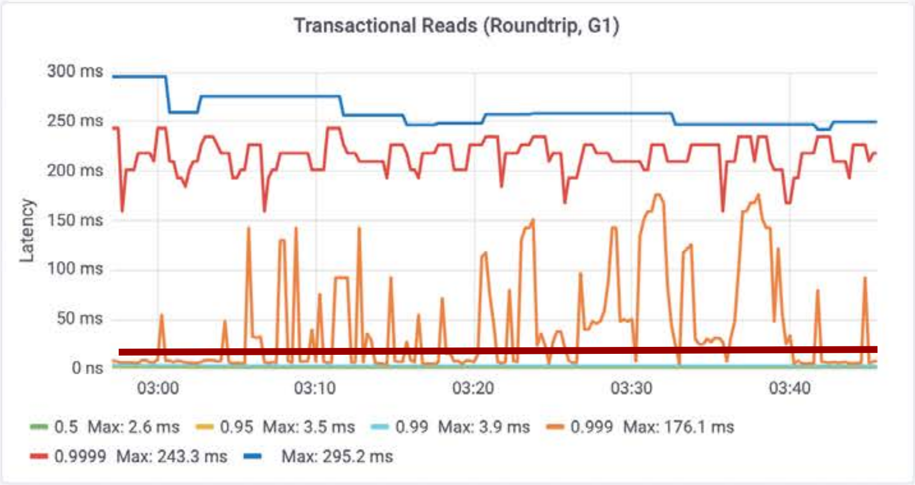

Transactional Reads (Roundtrip, G1)

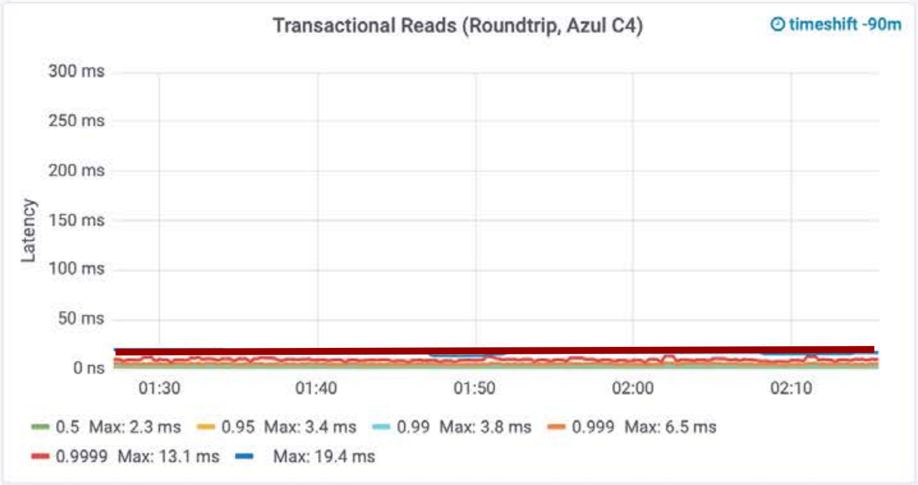

Transactional Reads (Roundtrip, Azul C4)

timeshift -90m

Transactional Reads with Persistence

30% in RAM, 100% on Disk (600 GB) [equalized scale]

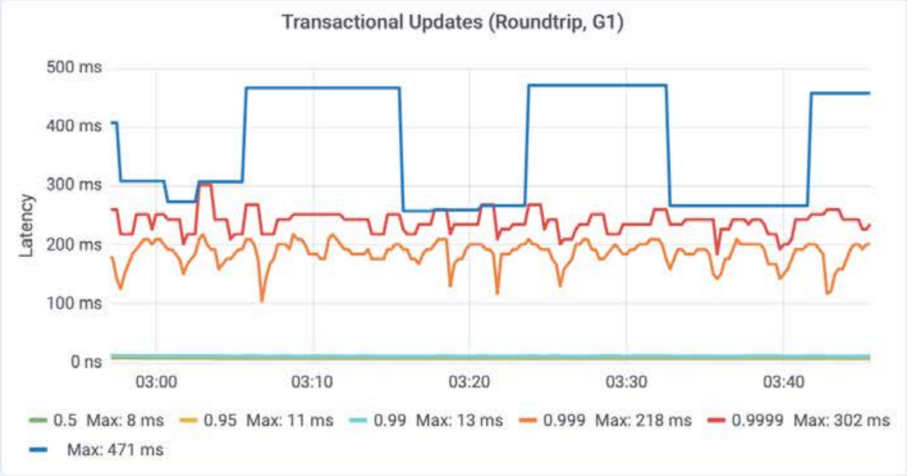


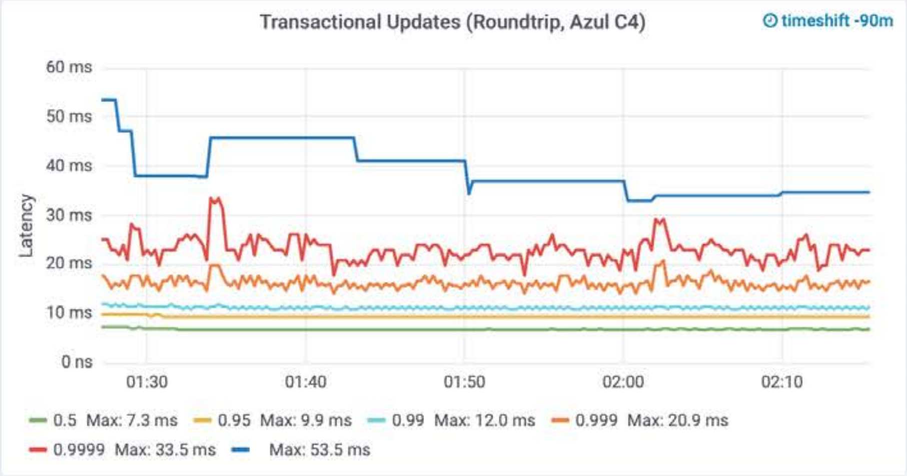


▬ **- target latency**

Transactional Updates with Persistence

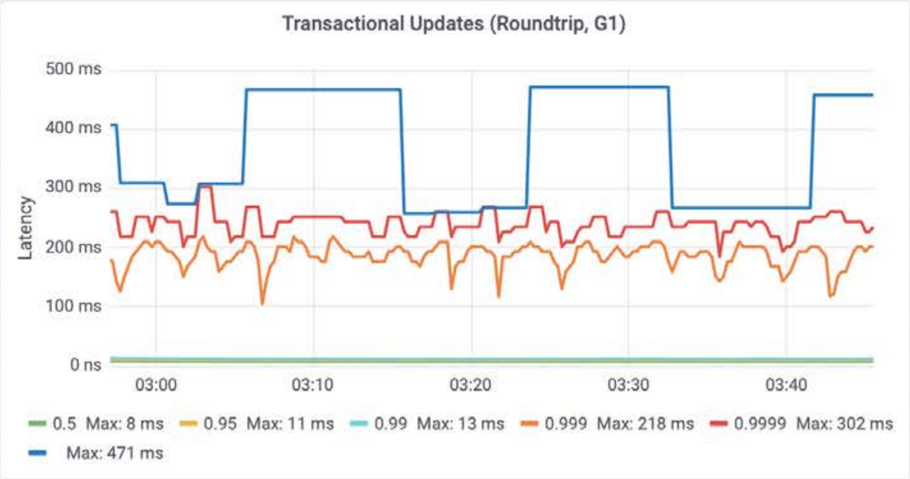
30% in RAM, 100% on Disk (600 GB)

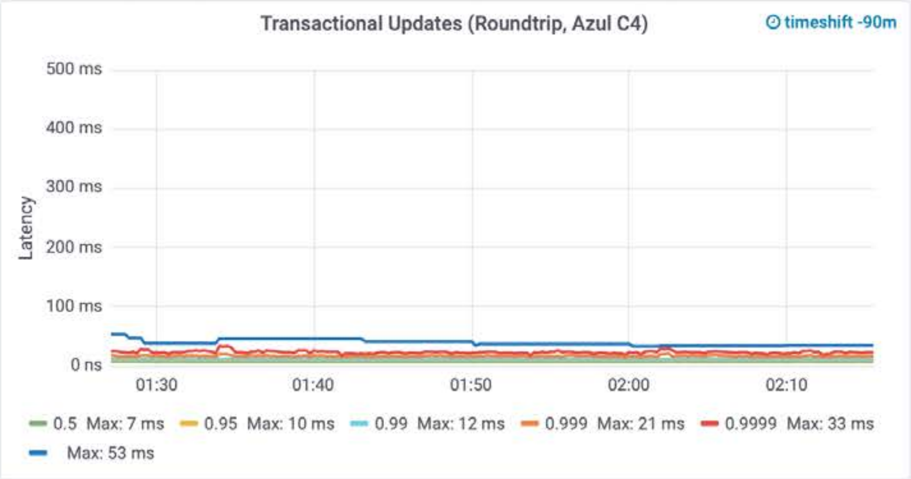




Transactional Updates with Persistence

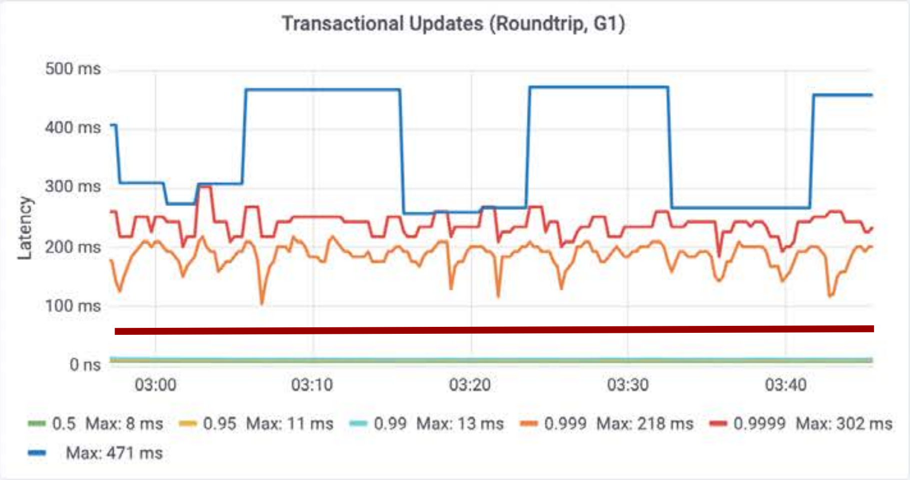
30% in RAM, 100% on Disk (600 GB) [equalized scale]

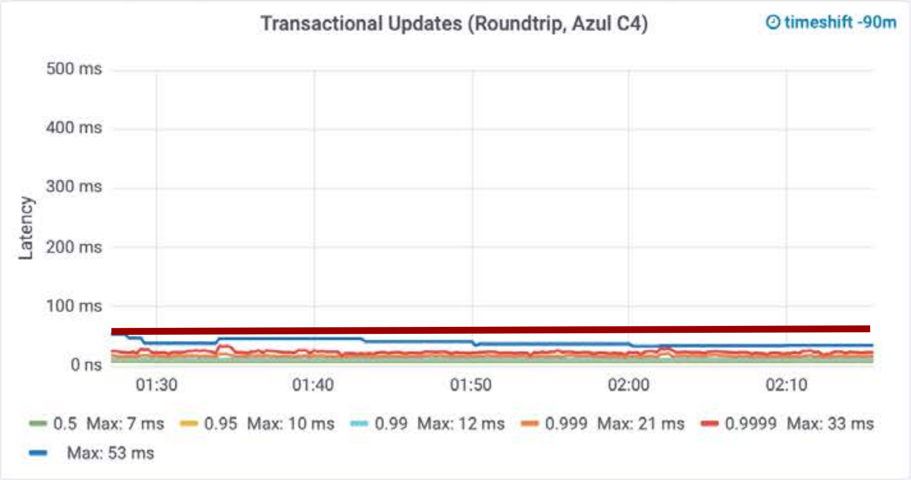

Transactional Updates (Roundtrip, G1)


Transactional Updates (Roundtrip, Azul C4)

Transactional Updates with Persistence

30% in RAM, 100% on Disk (600 GB) [equalized scale]

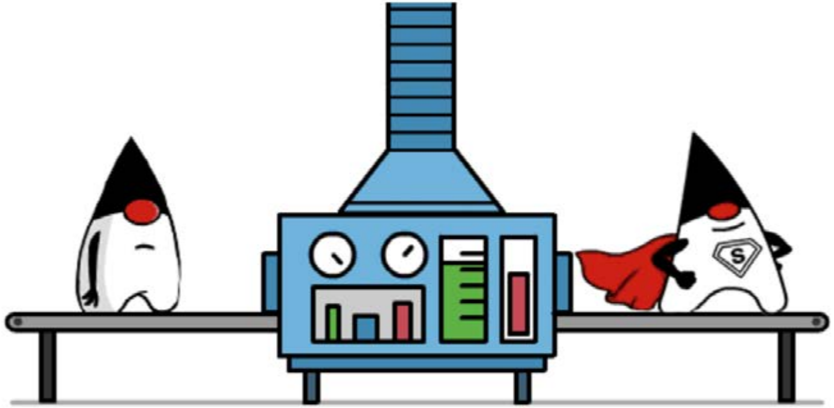




▬ - **target latency**

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