TECHNOLOGY
WHITE PAPER

AZUL SYSTEMS

Java Without the Jitter

Achieving Ultra-Low Latency

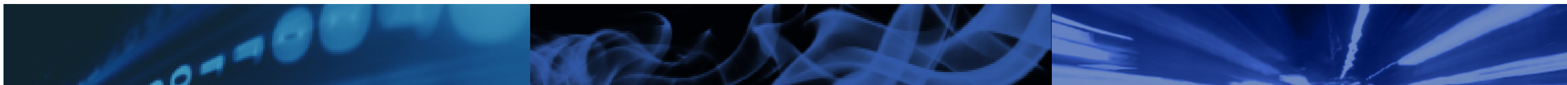


Table of Contents

Executive Summary

Messaging, trading, telco and online advertising systems require extremely low latencies, in the range of a few microseconds, to maximize revenue, meet customer expectations and deliver against strict service level agreements. This is in contrast to online commerce or other human-facing systems that can tolerate sub-second response times. For most low latency systems requirements continue to get even tighter – just a few years ago ultra-low latency systems were those below 100 milliseconds, now they are often in the microsecond range.

The Java language never used to be strong in this space. Traditional JVMs have inherent limitations that prevent achieving ultra-low latency. Chief among them is Garbage Collection (GC). This white paper discusses the historic pitfalls of Java application latency and jitter behavior and explains how Java applications can achieve the long-sought-after goal of continuously consistent latency at high sustainable throughputs. The paper then describes how to use new JVM technology to achieve worst case platform latencies of 10 msec or less “out of the box”, maximum platform latencies of 1-2 milliseconds with very little tuning effort, and even latencies in the microseconds with tuning.

Introduction

Messaging, trading, telco and online advertising systems (among others) require extremely low latencies to meet customer expectations and strict service level agreements. They demand latency in the microseconds range depending on the system. In contrast, online commerce or other customer-facing systems can tolerate sub-second response times without impacting the business. For ultra-low latency systems, requirements continue to get even tighter – just a few years ago sub-100 milliseconds was considered excellent.

Many applications that require low latency were originally written in Java to take advantage of its inherent advantages and ecosystem. Lots of programmers know Java, a variety of tools are available for both development and deployment, and many IT professionals are familiar with how to tune and manage production Java applications. However, for applications that require ultra-low latency performance, Java has inherent limitations. Chief among these is Java Garbage Collection (GC). Because the Java Virtual Machine (JVM) handles memory management for the developer, it must occasionally stop application execution to clean up old objects and defragment the heap. This can take seconds or even minutes depending on heap size.

Although many claims have been made in the industry about achieving low latency with Java applications, the reality is that without solving the GC problem low latency just isn't possible. You can delay full GC and reboot your systems during non-peak loads, but minor GC will still occur that raises average latency. In addition, if loads increase unexpectedly a full GC can still be triggered, leading to a multi-second or even multi-minute pause in application execution. For some businesses, this is ruinous.

THE BASICS OF THE JVM AND GARBAGE COLLECTION

The Java programming language utilizes a managed runtime (the Java Virtual Machine, or JVM) to improve developer productivity and provide cross-platform portability. Because different operating systems and hardware platforms vary in the ways that they manage memory, the JVM performs this function for the developer, allocating memory as objects are created and freeing it when they are no longer used. This process of freeing unused memory is called 'garbage collection' (GC), and is performed by the JVM on the memory heap during application execution.

Java garbage collection can have a big impact on application performance and throughput. As the JVM heap size grows, so does the amount of time that an application must pause to allow the JVM to perform GC. The result can be long, unexpected pauses that can delay transactions, deteriorate application throughput, cause user-session timeouts, or even more severe business related losses such as a drop in revenue. For more on how garbage collection works, see the Azul paper: Understanding Java Garbage Collection.

Why Java Pauses Can’t Be Tuned Away

Java is very fast – when it’s fast. If you measure clock-optimized JIT-compiled code that’s been running for a while, it’s comparable to C or C++. The issue for Java is what happens every once in a while. When Java is slow, it’s not only very slow, it actually stops. The mean or median response time isn’t the problem in this case, it’s the ‘99.something’ percentile of responses that’s actually very, very bad. Unlike some systems that slow down 20% or so, Java freezes for a period of time before it allows execution to continue.

For most JVMs these GC-related pauses are proportional to the size of their heaps: approximately 1 second for each gigabyte of live objects. So, a larger heap means a longer pause. Worse yet, if you run a 20 minute test and tune until all the pauses go away, the likelihood is that you’ve simply moved the pause to the 21st minute. So unfortunately, the pause will still happen and your application will suffer. Most enterprise applications will see very infrequent but very big multi-second pauses.

The good news is Java does provide some level of GC control. Developers and architects can make decisions that can help and even hurt application performance. For example, in financial high frequency trading, developers spend an enormous amount of time tuning and changing application code in specific ways to delay GC. They try to push those pauses so far out into the future that they can reboot before it actually happens.

What the financial companies can’t prevent are the frequent, smaller pauses. In Java, these are usually 10s of milliseconds long. If the company is really, really lucky and all the developers are careful to avoid allocating any memory, these pauses can be as low as a handful of milliseconds. However, with current JVMs this is as good as it gets. Java has an inherent jitter much greater than one millisecond built into the system. This isn’t the applications’ fault; it’s not the systems’ fault; it’s the JVM’s fault and just how it behaves.

Modern Servers Have Huge Capacities – Why Hasn't Latency Improved?

Modern x86 servers have 10s and 100s of gigabytes of memory and a dozen or more vCores at very affordable prices. At the time of this writing, a 16 vCore 256GB server was only $9k (list price, retail) on the web. Yet companies regularly deploy servers with only a fraction of these resources available for their Java application instances. Why?

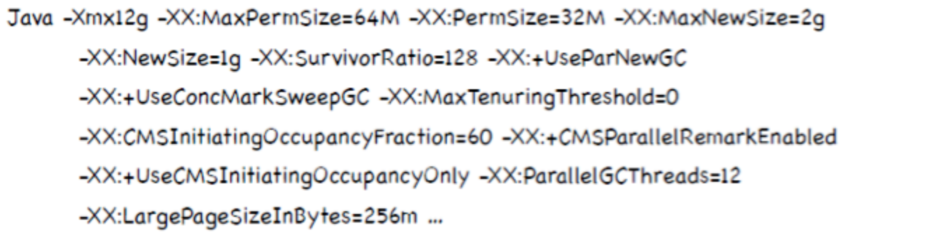
The answer is that traditional JVMs have GC pauses that are so painful at large heap sizes that companies will only go as far as the pain allows them. For applications that are human-facing, that pain starts at response times more than a few seconds. In machine-to-machine or other low latency applications that pain begins in the millisecond or sometimes even the microsecond range.

The garbage that needs to be collected builds up fast. Each core on a server will generate one-quarter to one-half a gigabyte of garbage per second when it's being used at some natural rate. Once the system has a couple of gigabytes of data on the heap the pause times are already a few seconds. In the world of low latency these longer pauses are delayed with the hope of restarting the servers before they hit, but the shorter pauses still happen. The result is that with Java, companies place more and more tiny instances on resource-rich servers.

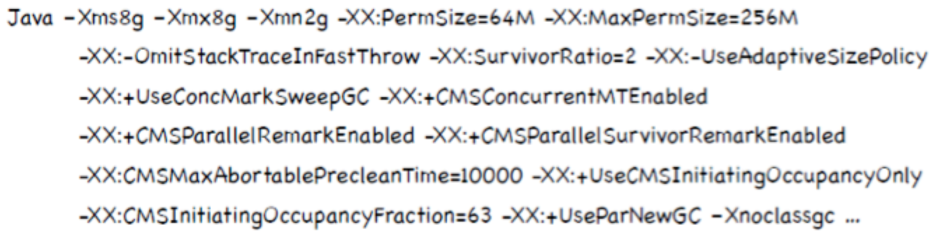
For low latency apps, the issue isn't just about using all the memory on the server it's about not being able to use the Java language anymore as your latency requirements keep going down. The core problem is 'monolithic stop-the-world' garbage collection events. The system has to stop everything (stop-the-world) to do garbage collection, and it has to run all the way to completion without interruption (monolithic). If it wasn't for the monolithic part, we could break up the work into incremental pieces and keep overall latency down. If each increment was small (say, less than 1 millisecond) and the system spread them over time the increase in latency would be more tolerable. However, traditional JVMs require monolithic GC events. Here's why.

Suppose the JVM's garbage collector needs to compact memory (i.e. move objects to create larger open blocks). It moves an object from point A to point B, but that object can have thousands or even millions or billions of pointers pointing to it. Before your application runs its next instruction, the garbage collector has to find and fix every one of those pointers. Moving the object is something the system can't take back, so once the move happens the application has to wait until every last pointer is identified and updated. This behavior is what almost all JVMs do today, and it is preventing applications from achieving consistently low latency.

Sample CMS Settings for the HotSpot JVM

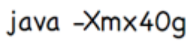


```
Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
        -XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
        -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
        -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
        -XX:+UseCMSInitiatingOccupancyOnly -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 ...
```

Sample Azul Zing C4 Collector Settings



```
java -Xmx40g
```

Achieving Ultra-Low Latency Java With Azul Zing®

Zing is an innovative JVM that completely eliminates garbage collection as an issue for enterprise and low latency applications. Unlike other JVMs that require extensive tuning or tweaking to minimize GC pauses, Zing eliminates these GC events. It completely decouples scale metrics such as larger heaps, more transactions or more users from response time. Zing is also the only JVM that has elastic memory capabilities to provide survivability and stability even in unforeseen situations.

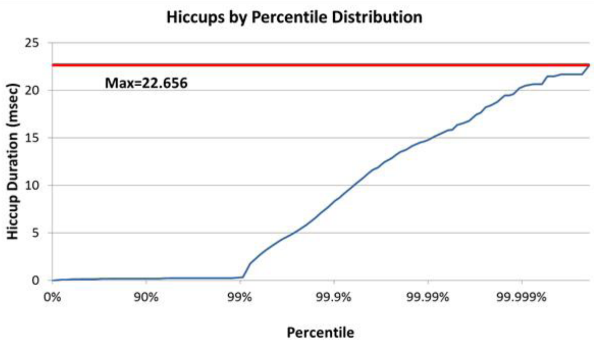
Zing doesn't require any specific tuning to deliver sub-10 millisecond response times consistently (assuming your current system at idle has less than 10 millisecond jitter). With a little tuning, usually about one or two days, Zing can easily achieve one to two millisecond worst case behavior for the JVM platform. An app that does 100 microseconds of work will be in that range, and the JVM would only add one to two milliseconds of jitter to an app that does 10 milliseconds of work. Zing also greatly simplifies tuning parameters.

GC Tuning: Traditional JVM vs. Zing

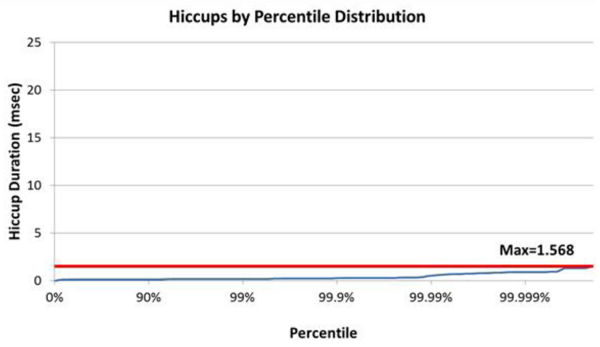
Garbage collection tuning for most collectors is hard to get right, even when you understand the characteristic of your application. The figure above shows two sets of tuning parameters for the Oracle CMS collector. While they may use similar parameters, they are very different and in some areas diametrically opposed. Yet the performance of your application could be optimized with either set, depending on its particular characteristics. With most collectors no 'one size fits all' answer exists. Developers and architects have to tune garbage collection carefully and retune every time the application, environment or anticipated load changes. Getting these parameters wrong can cause unexpected, long pauses during peak load times.

Actual Performance: Zing vs. the HotSpot JVM

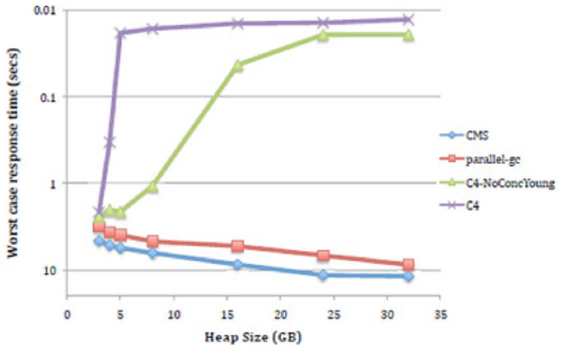
Hotspot JVM



Zing



Worst Case Response Times



However, performance of an application on Azul Zing is insensitive to the ‘usual’ tuning parameters. The only important parameter is heap size. Zing does have tuning flags, and for low latency applications some of them may make sense to use. For example, GC could be limited to a single thread to keep scheduling delays under 10 milliseconds. However, Zing doesn’t require any special tuning to deliver sub-10 millisecond response times consistently.

Some users of Zing have achieved worst case response times under one millisecond, though this requires more extensive tuning and may require changes to the application, assigning CPUs for certain functions or directing interrupts to certain cores. For an example of this type of tuning, see the Mechanical Sympathy blog.

Zing takes the JVM out of the way. It lowers JVM jitter to levels that allow you to achieve ultra-low latency with your application, even below one millisecond worst case.

Implementing the Zing With Your Application

Zing meets the Java SE standard and is based on the HotSpot JVM. Zing is simple to install and requires no coding change to the application. Simply point your application or startup scripts to use Zing.

Zing is optimized for Linux and x86 deployments and utilizes unique Azul Systems technology originally developed for its Java accelerator appliance. Key to Zing's performance is the Azul C4 (Continuously Concurrent Compacting Collector) which eliminates "stop-the-world" pauses that limit the scalability of all traditional JVMs. Zing also includes a management and monitoring platform with a zero-overhead, always-on visibility tool called Zing Vision and an instance management tool.

To check compatibility of your environment with Zing, see the full list of supported hardware and operating systems on our web site and/or run the free Azul Inspector tool against your existing infrastructure.

Summary

In many industries, required maximum latencies for Java applications continue to shrink. Unfortunately, traditional JVMs are limited by garbage collection pauses that increase latency, even when major GC events are delayed until the system can be rebooted.

Zing is unique in its ability to deliver sub-10 millisecond worst case response times out-of-the-box, with no tuning. It is the only commercially available JVM that completely decouples scale metrics such as larger heaps, more transactions or more users from response time. Zing takes the JVM out of the way of achieving low latency.

About Azul Systems

Azul Systems delivers high-performance and elastic Java Virtual Machines (JVMs) with unsurpassed scalability, manageability and production-time visibility. Designed and optimized for x86 servers and enterprise-class workloads, Azul's Zing is the only Java runtime that supports highly consistent and pauseless execution for throughput-intensive and QoS-sensitive Java applications. Azul's products enable organizations to dramatically simplify Java deployments with fewer instances, greater response time consistency, and dramatically better operating costs.

To discover how Zing can reduce the latency of your applications, contact:

Azul Systems
1.650.230.6500
info@azulsystems.com
www.azulsystems.com/lowlatencyjava