HFT Review
High Frequency & Algorithmic Trading

How to Maximize Java Application Performance in a Trading Environment

An interview with Scott Sellers, President, CEO & Co-Founder Azul Systems
(2nd April 2013)

In the world of latency-sensitive trading and HFT, we often focus on how performance can be improved at the hardware layer, for example via hardware FPGA/GPU acceleration or high-speed networking components. But what about the software stack?

In this interview, we speak with **Scott Sellers**, President, CEO & Co-Founder of **Azul Systems**, about how application performance in a Java trading environment can be massively improved by using the right Java Virtual Machine (JVM).

HFT Review: *Scott, welcome to HFT Review. Can we start with you giving some background on yourself and on your company Azul Systems?*

Scott Sellers: I started the company in 2002 along with two other individuals, including Gil Tene, our CTO. Since its inception, Azul has served a variety of different markets; everything from financial services to e-commerce to insurance to telecommunications to big SaaS companies. Over the course of our history we've identified a tremendous market opportunity for addressing many of the shortcomings prevalent in today's enterprise Java deployments. And because of the proliferation of Java that exists throughout the world - specifically in financial services - our value proposition has really resonated.

HFTR*: What sort of shortcomings are you referring to?*

SS: There are a number of limitations in today's Java platforms that can limit the inherent scalability of applications. There are also limitations impacting the business requirements of an application in terms of the percentage of time, for example, that it needs to respond within a certain latency. Customers also complain about the lack of visibility within the Java platform, so that when an application is in production it's very difficult to really understand

what's actually going on inside the run-time environment. It's like a black box, which is not very comforting for the owners of mission critical deployments. Those three key elements of pain that we continue to hear about -- scalability, inconsistent response times, and poor visibility -- are the three major problems that existed in 2002 when we started the company and they continue to exist today for people using off-the-shelf Java solutions.

HFTR: *I'd like to drill down on the problem with inconsistent response times. Can you expand on that?*

SS: Certainly. Most applications that we're typically asked to power with our products are going to have a set of SLAs or a set of multi-dimensional business requirements, and latency is always going to be one very critical element. For example, how fast can you process a trade? How quickly can you pass a message through if you're dealing with something like a FIX engine or a FIX gateway? How quickly can you update your risk analysis for a real time risk engine? Latency is a critical element, particularly latency under load. What typically happens is that an application might be meeting the business requirements and processing what it needs to do within a certain latency threshold, but only under a certain set of load conditions. So it might work 99% of the time, but in mission critical deployments 99% is nowhere near good enough. Our customers require three, four or five 9's, meaning 99.999% of the time the application must respond within a very strict latency requirement.

The nature of the challenge is that although the average latency may be acceptable for the business, the most challenging latencies - the ones that occur in the three, four or five 9's - are not acceptable. And given the fact that this is such a brutally competitive environment where pennies sometimes have to be made millions of times over in a day in order to make money, if you are unable to have an absolutely consistent platform, not only may you be losing money, quite frankly you may be out of business.

HFTR*: So where do the bottlenecks tend to occur?*

SS: The problem is always one of peeling away the onion. When you're trying to deliver a truly world-class platform that meets all of the latency requirements (which inevitably are reducing over time), there can be many elements that can cause worst-case latencies. It could be slow hardware, it could be network limitations, it could be how the operating system is tuned, it could be the application itself, and then of course it could be the piece of software

that runs the Java workload, i.e. the Java Virtual Machine (JVM). Many of those elements in the overall application stack could be a limiting factor causing latency outliers.

Intel has done a wonderful job of continuing to evolve the server platform and as long as an application is running on a reasonably modern piece of commodity hardware, typically it's not going to be the server itself that's the limiter. Networking standards have also improved and networking vendors have done tremendous work in terms of reducing latencies associated with TCP/IP networking protocols. Five years ago network latency was a real issue, but that's becoming less and less so with the evolution of the technology in that space.

From the Operating Systems perspective, there has been a number of changes over the last few years to improve the scheduler within the Linux kernel, adding some tuning flags that help with latency sensitive situations, and distributions of real-time Linux are also beginning to be used. And once the Operating System is tuned, it can generally be removed as a cause of latency outliers.

Another culprit of latency outliers may be the application itself. Perhaps poor programming constructs or practices are causing long application delays. Developers can use profilers to determine areas in their code that are not performing and make required changes. But once they've been addressed there's only thing left, and that's the JVM. And this is one of the most frustrating elements for users of Java because you only have so much control over a JVM, and what happens inside the JVM is very complicated. It's responsible for compiling the Java application onto whatever platform it's being run on (typically x86 Linux), it's responsible for running the application itself, it's responsible for all the threading involved in the applications, and very importantly it's also involved with the garbage collection process, which quite frankly is the Achilles' heel of the Java run-time.

***HFTR**: Can you spend a couple of minutes just explaining why the garbage collection process can have such a significant impact on the performance of running applications?*

SS: Certainly. In Java - or any managed run-time type of language - it is not the responsibility of the application itself to worry about the memory management, it's actually the responsibility of the run-time layer, and in the case of Java that's the JVM. The reason it's important is that the management of memory is a very tricky thing to get right. With

programs written in C and C++ the memory management is the responsibility of the programmer or the application itself and as a result of that there are often bugs in the application and/or security vulnerabilities, because the way the memory is managed can be exploited by viruses and other threats that find holes in the application and do malicious things.

The combination of the trickiness of getting memory management code correct, bug free and scalable in performance, combined with the inherent security vulnerabilities of having memory managed by the application itself, led the computer science world to this concept of managed run-time environments, like Java. One of the reasons that Java has been and remains the world's most popular programming language is because of the programmer efficiencies and the productivities of using a language where you only need to worry about coding the business logic, you don't have to worry about subtleties such as memory management. That's the good news.

The bad news is that when memory management is the responsibility of the run-time, as the application is running and objects are being created, used, thrown away and then eventually reclaimed, the memory used by the running application gets fragmented. An analogy is disk fragmentation that occurs on your desktop computer or your laptop. At some point in time what used to be a nice, clean memory space ends up looking like Swiss cheese. As the application runs, at some point the memory becomes so fragmented that no more memory can be effectively allocated, so the memory needs to be cleaned up and defragmented. And just like when the disk defragmenter kicks in on your desktop computer and your work productivity plummets, the same thing is true in a running Java application. When the garbage collector - which is the process within the JVM responsible for cleaning up this memory - kicks in, it can have a dramatic impact on the running Java application because, in essence, the application has to be paused while the JVM goes around and cleans up the memory, remarks it, packs it and condenses it, in order to be able to remove those holes and make it a nice, clean, continuous memory space so that memory can be allocated once again and the application's happy.

***HFTR**: And presumably the impact of the garbage collection process on the application's latency can be quite significant?*

SS: Well, the technology and the algorithms involved in doing this have improved over the years. In the early days of Java this was truly an "all or nothing" type of capability. Either the application was running or the garbage collector was cleaning up the memory, and when that was happening, it had to stop the application completely, what's called a "stop the world" (STW) pause. Those familiar with tuning Java still have nightmares about STW pauses because if you're an application with real-time requirements, it's very difficult to predict when that garbage collector will actually kick in and freeze the application itself. And when that application is stopped, that means trades aren't happening, messages aren't being passed, risk is not being analysed, matching engines freeze, all these things pause whilst this crazy process called garbage collection is just doing its job.

***HFTR**: Presumably this is where Azul comes in?*

SS: Yes. The state of the market today is that there are really only two major JVMs being used within the financial services community on trading platforms. It's either going to be the HotSpot JVM that originated from Sun or it's going to be Azul's Zing JVM. The difference between the two is that in the context of strict latency requirements, the garbage collection algorithms inside the HotSpot JVM were never designed for the demanding low latency requirements mandated by today's financial services markets. Even though the application pauses due to garbage collection with Hotspot have improved over time, they're nowhere near the levels that are required by today's aggressive trading platforms, which require worst-case pauses in the low single digit milliseconds and often times in microsecond levels.

The big benefit of Zing is specifically around this point of the worst-case pauses associated with the garbage collection process. Not only have we dramatically reduced those pauses down to sub-millisecond levels, but we have also done it in a way that is independent of the size of the memory being used, which is another key element. HotSpot, with lots of tuning, lots of effort and very, very small heap sizes (the heap is the amount of memory used for a running Java application), can perhaps meet the business requirements of a given application. But what happens is that today's applications want to use more and more memory to be able to handle higher volumes and to do more in-memory analytics. So instead of accessing systems across a network (whether that's a database system or talking to another platform), in today's modern environments everything wants to be held in memory as close to the application as possible. As a result, the memory size requirements for these Java applications are growing all the time. Within HotSpot, there's a direct relationship

between the amount of memory being used and the worst-case pauses associated with garbage collection and there's nothing you can do about it, the more memory you use, the longer the pauses are going to be.

Understandably this causes angst amongst HotSpot users within the Java community. This pressure to keep memory small to try to keep the garbage collection pauses manageable is in conflict with modern computer science programming, where you want do more in-memory analytics, you don't want to take the penalties of crossing networks and hopping over other systems. So the benefit of Azul's Zing is the fact that not only does it slash latencies for any size memory, but it can also host microsecond-level worst-case pause times with hundreds of gigabytes of heap being used for Java processes. It's truly the best of all worlds and quite frankly why Zing is being used so prevalently is today's financial markets.

***HFTR**: Can you give us a few examples of where Zing has been implemented in the financial markets, how it's helped and what kind of results have been achieved?*

SS: Certainly. In a general sense we've seen it used predominantly in the front and middle office. Obviously those are the areas where latency and faster response times are really critical for the business. Specifically, in the front office, we see Zing first and foremost in trading platforms that are processing millions of trades a day across the equities markets, across foreign exchange, across fixed income and a variety of other instruments, where the trader needs the absolute, most consistent performance and fastest results. There are many other areas that require similarly consistent response times, so we power many of the FIX engines and FIX gateways that are commonly used throughout banks and hedge funds. Our customers include market makers whose pricing engines need to be more responsive and more competitive all the time. We power matching engines.

In the mid-office, risk is moving from a batch process into real time and as a result, the SLAs for the real time risk engines are now as strict as the trading platforms. The need to be able to constantly assess risk in real time, to be able to re-analyse risk based on changing prices and changing positions means that we're also seeing a lot of use of Zing in this area, in terms of being able to reduce latency to keep up this never ending need for faster and faster performance and more and more consistent latency.

***HFTR**: When swapping out an existing JVM with Azul Zing, how much tuning is typically required in order to get optimum performance?*

SS: That's a good question. The answer is perhaps ironic in the sense that we find a lot less tuning is needed with Zing compared to HotSpot. Often when we engage with a business we find that they have spent literally man-years of effort to try to tune HotSpot to get to where they need to be in terms of the requirements of the business, and all of their efforts have ultimately resulted in a failure to meet those demands. The beauty of Zing is that it's a 100% compatible JVM, it's fully compliant and compatible with all the Java specifications. So instead of what literally can be hundreds of tuning flags associated with HotSpot and all of the trial and error of trying to get it to do something reasonable to meet the business demands, Zing is incredibly simple and has very few flags and arguments associated with it. Most of the time the only thing a customer has to do is just set the heap size and they're off and running. The same application that runs on HotSpot runs on Zing out of the box, there's no application change, re-coding, re-compilation, none of that stuff is necessary.

We can very quickly derive quantitative metrics that compare Zing vs HotSpot, and what we typically see is that worst-case latencies will be reduced by factors of 100x or 1000x and average latencies also reduced. We also see maximum volume increasing because you can use more memory and it's a much more scalable JVM. The fact that Zing is truly a plug and play solution that doesn't require a bunch of tuning facilitates a very easy pilot/evaluation process, which also results in an easy quantitative comparison between HotSpot and Zing.

***HFTR**: I understand you offer a free open source toolkit. What can you tell me about that?*

SS: We find in certain cases, users of Java don't really understand where their applications are failing to meet the business requirements, so we have a set of tools that help them with that. The most common one used is something called jHiccup, a 100% open source free tool that can be run with Zing or with HotSpot. It is simple to install and runs alongside an application and measures the "hiccups" or the pauses of the latency outliers, allowing an operator to quickly assess whether or not those latency outliers are due to the JVM. It's very beneficial because it helps users try to figure out what layer of the onion is a potential bottleneck, so it's very popular and is used by all sorts of people in the Java community.

***HFTR**: Finally, what can you tell us about your future plans?*

SS: Clearly the success we've had in financial services is a foundation for future development. We are becoming the de facto standard for Java applications in the financial markets. More and more, Java applications that have strict SLAs and competitive latency requirements are using Zing. Where we're headed is to continue to add features within Zing that are unique and also specifically designed for use within capital markets. Without getting into specifics, the fact that we are powering so many of these mission critical deployments within the financial space means that we have a unique perspective on some of the remaining problems that exist, which we're working into our roadmap.

And of course we have plans to support all of the forthcoming versions of Java, so we'll continue to support the industry standards.

***HFTR**: Scott, thank you very much.*

