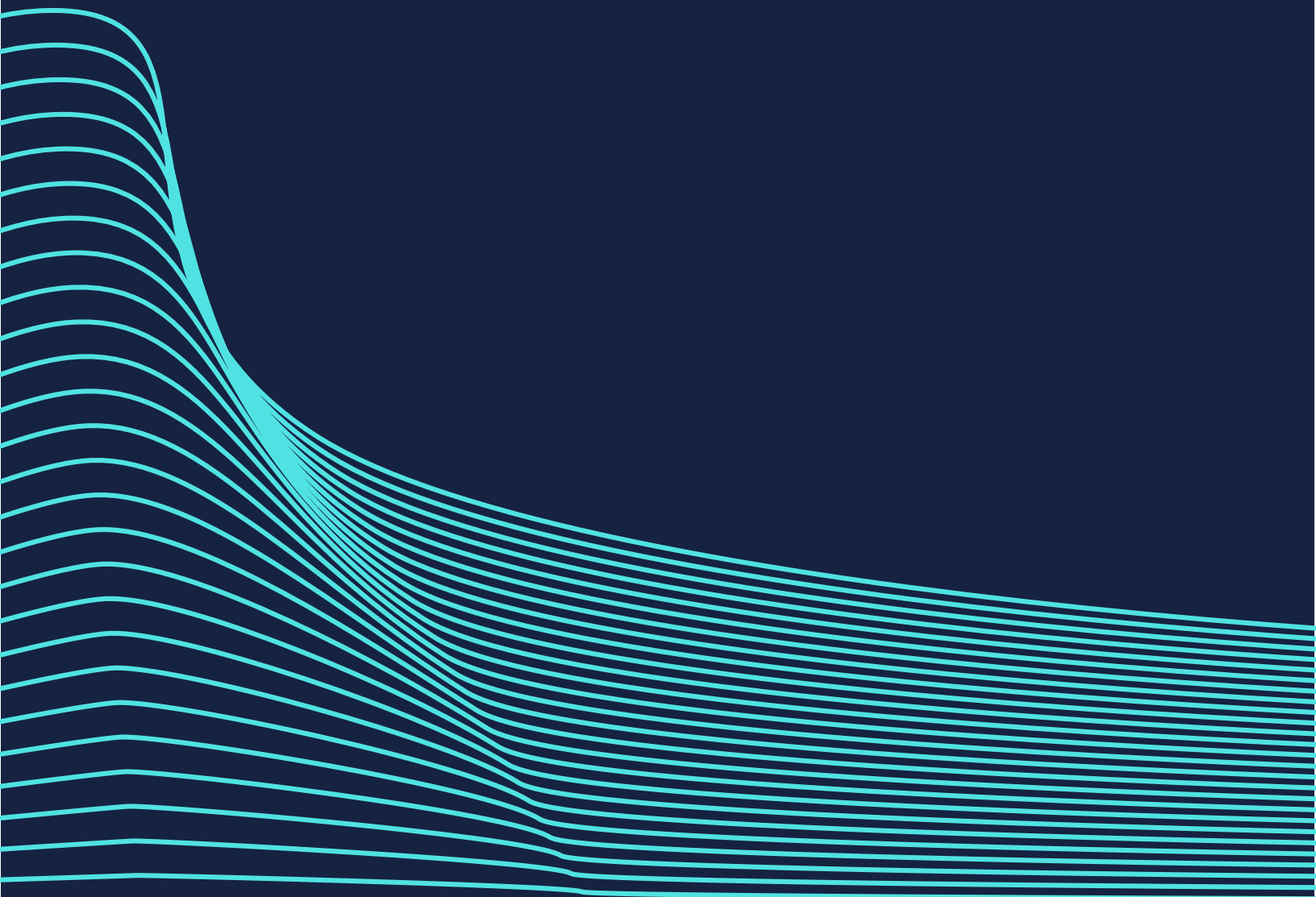
azul

Azul Prime: Java At Speed. A JVM Optimized For Modern Hardware



Azul has spent many years developing solutions to these challenges

Using the latest hardware advances and, in this white paper, we'll explain how Azul Platform Prime addresses the performance of bytecode interpretation and adaptive compilation.

Introduction Java is consistently reported as the most popular programming language on the planet. Many other programming languages can be compiled into bytecodes and run by the Java Virtual Machine (JVM) in exactly the same way as compiled Java code. One of the reasons for Java (and the JVMs) popularity is how it abstracts away a number of things that developers have always had to deal with in other popular languages like C and C++. Java is famous for being, "Write once, run anywhere" by virtue of its use of an intermediate representation of application code in the form of bytecodes. In addition, the JVM allocates space for objects and automatically reclaims this space when the objects are no longer required; using what is referred to as garbage collection (GC).

With languages like C and C++, all memory management must be handled explicitly by the programmer, who must ensure that space no longer required is freed. Otherwise the application will suffer from the classic memory leak.

While these are great features for making a developer's life easier and produce more robust and reliable code, they are not without impact. The performance character-istics of applications running on a traditional JVM can look very different to statically compiled applications written in C and C++. Although the overall performance of bytecodes in the JVM can be as good as natively compiled code (and in some situations can even exceed it), the JVM introduces greater non-deterministic effects on the application performance (i.e. it is not as easy to predict the level of performance at any specific time).

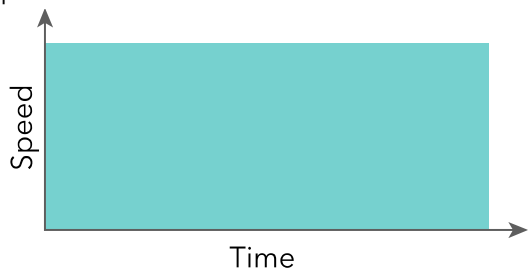
Modern hardware includes numerous ways to improve performance by the direct use of machine instructions at the silicon level. Features like vector processing can make a CPU appear to have a far higher clock speed than it really does by enabling processing of multiple data elements in a single instruction cycle.

Azul has spent many years developing solutions to these challenges using the latest hardware advances and, in this white paper, we'll explain how Azul Platform Prime addresses the performance of bytecode interpretation and adaptive compilation. This will cover two main areas:

- How the new Falcon just-in-time (JIT) compiler uses features of modern processors to deliver better overall performance in many cases.
- How the ReadyNow! feature of Azul Platform Prime can eliminate the effects of traditional Java application "warm-up".

What Do We Mean By "Speed" For An Application?

When considering application performance, the ideal graph looks like the one below.



Here we have a completely constant and predictable level of performance that will provide consistent and predictable response times for our clients. There are really four things to consider when evaluating the speed of your application:

1. Are you fast enough? What this question asks is can your application deliver the required results within the time your clients have specified. This is a typical non-functional requirement. For example, "the system will respond within 50ms 95% of the time".

2. Are you fast when new code is deployed? With most enterprises now using continuous integration and continuous deployment, it is important that

performance is unaffected when new versions of production software are deployed.

3. Are you fast from the start? Often applications need to do certain things when they start up such as loading large sets of data into memory. The important question here is how long it takes from starting an application until it is running at full speed.

4. Are you reliably fast? Like the first question, this is a primary non-functional requirement but viewed from a load perspective rather than response time, e.g. "the system must be able to process 10,000 transactions per second".

Unfortunately, no application has a performance graph like the one above and when running an application on a traditional JVM the graph will look very different.

Let's start by looking at what the challenges are for the JVM when executing your code.

Traditional JVM Performance Characteristics

When Java was first released, over twenty years ago, the designers took a different approach to executing applications to the way popular languages had used up until that point. Traditionally, source code was compiled into a set of native instructions for a specific platform (a combination of the processor and operating system on which the application would run). This is referred to as Ahead of Time (AOT) or static compilation. This fixes the platform that the application runs on, which means separate binaries must be generated for each platform the application will support.

With Java, the source files are compiled into class files that contain bytecodes. Bytecodes are instructions for a virtual machine, i.e. not one that uses any real processor or operating system. The class files are loaded by the JVM, which converts them to the machine instructions used by the underlying hardware and system calls for the chosen operating system. Some of this work is very simple, with a straightforward mapping between virtual and machine instructions. For example, the JVM has an opcode, ishl, for a logical shift left of an integer; this maps directly in the Intel x86 instruction set to the SHL opcode. However, many conversions from bytecodes to real instructions are much more complex.

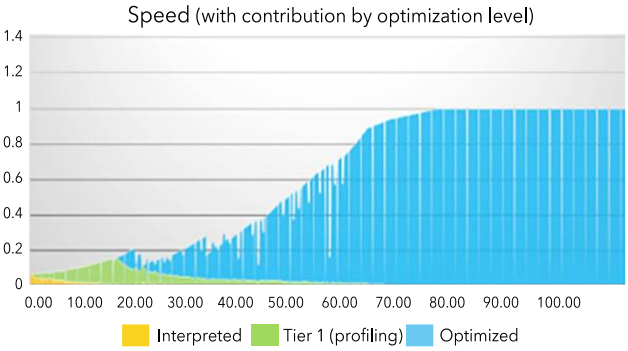
When the bytecodes from a class file are read, they will be interpreted, as they are needed. Interpreting means that each bytecode is translated into one or more native processor instructions, which are passed to the CPU for execution. This delivers sub-optimal performance for two main reasons:

1. Each time a bytecode is read it has to be interpreted as if it was the first time it is being used. No attempt is made to cache the native instructions.

2. Each bytecode is treated in isolation, so no optimizations are made based on sequences of instructions (as is performed by static compilation). There are many simple (and complex) optimizations that the interpreter does not use such as dead-code elimination and loop unrolling.

Clearly, this is not the most efficient way to execute bytecodes on the JVM and will give much lower performance than statically compiled code. To alleviate this problem shortly after Java was released a new, improved virtual machine was developed called Hotspot. Hotspot profiles the bytecodes of the application as they are executed, looking for sections of code that are used repeatedly (hot spots in the code path, hence the name). As an example, code running in a loop, especially if it is a loop with many iterations, is quickly identified as a hotspot. The JVM can then compile the bytecodes of the hotspot section using a more traditional (think C and C++) back-end compiler. This is adaptive compilation using a just-in-time (JIT) compiler. The JIT can perform optimizations as the code is compiled. This compiled code can also be cached so that subsequent iterations of a loop use the stored code rather than interpreting each bytecode or recompiling the code of the loop each time.

If we look at a typical performance profile for an application running on a traditional JVM we will see something like this:



When the application starts, it is running in interpreted mode, so much slower than code that has been compiled. As the application executes sequences of bytecodes the JVM profiles and compiles the hotspots, which then execute much more quickly. The curve of the graph demonstrates the classic warm-up phase of a Java application. There are also noticeable dips in performance both during warm up and even when a steady state has been reached. Many of these are due to GC pauses; something else that Azul Platform Prime drastically reduces through the use of the C4 GC algorithm.

The obvious question many people ask is, "Why can't we use static compilation for Java? Wouldn't that solve the whole interpreting performance problem?"

To answer that question, we must delve deeper into how adaptive compilation works in the JVM.

One of the most common optimizations used by compilers is method inlining. When a method is called, a new stack frame needs to be created, parameters pushed, and a jump performed to the code at the start of the method. This overhead can be avoided by taking the instructions of the method and placing them where the method call takes place (there is a little more to it than this as the method parameters need to be mapped correctly).

Although Java is a statically-typed language (and therefore the JVM has been designed for this), classes can be loaded dynamically at runtime. With static compilation, the compiler must be very conservative about method inlining since it is often unable to guarantee that the code for the method it inlines is the actual code that will be executed at runtime. Static compilation will typically only inline final methods since they cannot be overridden. A JIT compiler compiles the code when the application is running; the JVM knows exactly which classes are loaded and so can use method-inlining anywhere it is useful. Similarly, classes are not initialized at the same time they are loaded. Initialization only happens when a new object of that type is instantiated, a static field of the class is referenced, or a static method of the class is called. Static compilation must assume that all classes are uninitialized and place checks into the generated machine code. This leads to a considerable degradation in performance over machine code produced by a JIT, which knows precisely which classes are initialized.

JIT compilers are also able to perform what is called speculative optimizations that are not possible with static compilation. Let's look at an example of this. We will use the "path never taken" example, using the code below

```
int computeMagnitude(int value) {
if (value > 10)
bias = computeBias(value);
else
bias = 1;
return Math.log10(bias + 99);
}
```

In this code, we may have a situation where passing a value that is greater than ten is where we have to deal with some very unusual condition. Under normal circumstances, the value will always be less than, or equal to ten. With statically compiled code, the compiler must compile the code as it is written so the Math.log10() method must be called every time. The JVM, however, profiles the code as it is being run in interpreted mode so has a clearer picture of what is actually happening. The JIT can identify that, so far, the value has never been greater than ten. The code it compiles will actually be as shown below

```
int computeMagnitude(int value) {
if (value > 10)
uncommonTrap();
return 2;
}
```

Clearly, the code generated will be more efficient because there is no longer a call to Math.log10(). In the event that the value is greater than ten, the uncommontrap() method is called, which will cause the JVM to abandon the compiled code (because it is now incorrect) and revert to interpreting the bytecodes. This is referred to as a deoptimization and explains some of the dips in performance on the graph on the previous page, especially during the warm-up phase.

To improve things further, the JVM uses two JITs, sometimes referred to as client and server but more often as C1 and C2. The reason for having two JITs is that they have different performance profiles. The C1 JIT is designed to generate code at the optimum level of performance as quickly as possible, whereas the C2 JIT will take longer to generate code. The code generated by the C2 JIT will be more heavily optimised so give a better, overall, level of application performance. Since Java SE 6 the JVM has had the ability to do 'tiered compilation'. This uses the C1 JIT when the application starts to improve performance rapidly and then switches to the C2 JIT to improve the speed of the application further.

Modern Processor Design

The Intel x86 architecture dates all the way back to the 8086 processor launched in 1978, but the first real x86 32-bit processor was the 80386 launched in 1985. In the early 2000s, the switch was made to 64-bit processors, and the x86 architecture became the x64 (although the base instruction set remained essentially the same).

Over thirty years the x86/x64 architectures have become the dominant processor architecture in laptops, workstations and most servers. During this time there have been many improvements to the design. Initially, the focus was on increased clock speed: execute the same instructions faster. About fifteen years ago clock speeds reached a plateau due to the physical problems of dissipating the amount of heat generated by the processor (air cooling with fans just isn't efficient enough, and people don't really want water cooled laptops or workstations).

To take advantage of Moore's law, which predicts the rapidly increasing density of transistors in a given amount of space on the processor, chip designers turned to increasing the number of processing units on a physical chip. This has led to multi-core processors as we see in almost all computers today.

The other area that chip designers have been working on to improve overall performance is looking at how to do things on the processor ("in silicon") using a single instruction rather than a sequence of instructions. With a greater number of transistors to use it has become possible to provide a larger number of increasingly sophisticated instructions (compare that the original 8086 processor, which had less than a hundred instructions to the most recent x64 processors, which have over seven hundred).

It is also possible to increase the size of the registers used from 32 or 64 bits (the size of a processor word) all the way up to 512 bits in the latest Skylake processors. These exceptionally wide registers are used to hold several data words at the same time and can be used for vector processing, which uses single instruction-multiple data (SIMD) instructions.

However, having a processor that provides all these facilities will not improve application performance unless the code generated by the JIT compiler is able to generate code to use them.

The LLVM Compiler Project

Azul has always been focused on improving the performance of the JVM with the goal of making your applications perform better and more reliably. Initially, this took the form of replacing the garbage collector

(GC) in the JVM with a different design called the Continuous Concurrent Compacting Collector (C4).

Having solved the problems of GC, Azul turned their attention to how to improve the performance of JIT compilation and eliminating, as far as possible, the warm-up time of an application.

The existing C2 compiler was written at the end of the 1990s. Attempts to add new optimizations and improve existing ones quickly made it clear that the design of the C2 JIT was just not adequate to allow continuous incremental inclusion of new features. What was needed was a more modular approach to the construction of the compiler.

To address this need, Azul selected the LLVM project as a starting point for our new JIT compiler for Azul Platform Prime.

The LLVM compiler infrastructure project (formerly Low-Level Virtual Machine) was started in 2000 at the University of Illinois at Urbana–Champaign. It is a "collection of modular and reusable compiler and tool chain technologies" used to develop compiler front ends and back ends. LLVM is released under an open-source license and is used and supported by a number of high-profile companies such as Apple (used in all its Mac OS and iOS development tools), Sony (in the SDK for the PS4), Intel and Nvidia, among many others.

Having input from engineers from these companies as well as many others made it a natural choice for Azul to use to start building a new JIT for the JVM. However, taking the code from the LLVM project was just the start. Work was required in terms of adapting it to work in a managed runtime environment. The compiler needs to work in conjunction with the garbage collector and understand how to deal with safe points. It was also necessary to look at how the compiler would address the dynamic nature of code replacement in the JVM, which is not something that needs to be handled by a static compiler. Having enhanced the code to work in a dynamic, managed environment, Azul followed the open source principle and pushed the changes back to the LLVM project so they would be freely available.

The Falcon JIT Compiler

The new JIT compiler in Azul Platform Prime, using the code from the LLVM project, is called Falcon. The name was selected because the Falcon is the fastest animal on the planet; the Peregrine Falcon has been recorded diving at a speed of over 200 miles per hour!

Using LLVM, a modular and a well-supported open-source project, means Azul Platform Prime can quickly and easily take advantage of optimizations for modern hardware. Let's look at an example of how Java code can be optimized in different ways.

```
private int sumArray(int[] data) {
int sum = 0;
for (int i = 0; i < data.length; i++)
sum += data[i];
return sum;
}
```

The code above is simple and existing JIT compilers can easily recognize that vector processing instructions (like AVX on Intel) can be used. By taking advantage of SIMD processing, the performance of this method can be significantly improved.

However, let's look at a slightly more complex example.

```
private void addArraysIfEven(int[] a, int[] b) {
if (a.length != b.length)
throw new RuntimeException("Length mismatch");
for (int i = 0; i < a.length; i++)
if ((b[i] & 0x1) == 0)
a[i] += b[i];
}
```

In this case, the application of vector processing instructions is not straightforward and traditional JIT compilers will not use those instructions but will use more basic techniques such as loop unrolling to

perform multiple operations for each iteration of the loop.

Because Falcon is based on LLVM, it can immediately benefit from optimization improvements made in that project. If we look at the instructions generated by Falcon for this code we see the following:

		0x3001455e	nop
0.15%	4	0x30014560	vmovdqu -96(%r11), %ymm2
12.31%	320	0x30014566	vmovdqu -64(%r11), %ymm3
0.50%	13	0x3001456c	vmovdqu -32(%r11), %ymm4
2.04%	53	0x30014572	vmovdqu (%r11), %ymm5
0.31%	8	0x30014577	vpand %ymm0, %ymm2, %ymm6
4.54%	118	0x3001457b	vpand %ymm0, %ymm3, %ymm7
0.69%	18	0x3001457f	vpand %ymm0, %ymm4, %ymm8
1.35%	35	0x30014583	vpand %ymm0, %ymm5, %ymm9
0.42%	11	0x30014587	vpcmpeqd %ymm1, %ymm6, %ymm6
2.58%	67	0x3001458b	vpmaskmovd -96(%rcx), %ymm6, %ymm10
3.58%	93	0x30014591	vpcmpeqd %ymm1, %ymm7, %ymm7
2.12%	55	0x30014595	vpmaskmovd -64(%rcx), %ymm7, %ymm11
12.12%	315	0x3001459b	vpcmpeqd %ymm1, %ymm8, %ymm8
1.50%	39	0x3001459f	vpmaskmovd -32(%rcx), %ymm8, %ymm12
3.69%	96	0x300145a5	vpcmpeqd %ymm1, %ymm9, %ymm9
1.81%	47	0x300145a9	vpmaskmovd (%rcx), %ymm9, %ymm13
12.27%	319	0x300145ae	vpaddd %ymm2, %ymm10, %ymm2
0.58%	15	0x300145b2	vpaddd %ymm3, %ymm11, %ymm3
0.19%	5	0x300145b6	vpaddd %ymm4, %ymm12, %ymm4
0.58%	15	0x300145ba	vpaddd %ymm5, %ymm13, %ymm5
3.27%	85	0x300145be	vpmaskmovd %ymm2, %ymm6, -96(%rcx)
7.15%	186	0x300145c4	vpmaskmovd %ymm3, %ymm7, -64(%rcx)
13.65%	355	0x300145ca	vpmaskmovd %ymm4, %ymm8, -32(%rcx)
4.58%	119	0x300145d0	vpmaskmovd %ymm5, %ymm9, (%rcx)
6.81%	177	0x300145d5	subq $-128, %r11
0.69%	18	0x300145d9	subq $-128, %rcx
0.31%	8	0x300145dd	addq $-32, %rbx

The sections highlighted in red show the AVX2 instructions generated by Falcon for the example method (here we used a Broadwell E5-2690 v4 processor). AVX2 instructions work on 256-bit registers, so each instruction group is able to process eight elements of the array. Given the way these instructions work, the effect is to make the processor (assuming a clock speed of 2.5GHz) appear to be running at 9GHz for this method.

The vectorization code is also executed four times enabling us to process 32 array values for each iteration of the loop.

The Falcon compiler also supports the newer AVX-512 instructions so, had this been run on an Intel Skylake processor the number of array elements processed per iteration would have been doubled.

This is only one example of how the Falcon JIT compiler can generate better performing code. As further optimizations become available through the LLVM project, these will be quickly integrated into Falcon, as well as improvements made by Azul's own engineers.

ReadyNow! Eliminating JVM Warm-Up

Looking back at the traditional JVM performance graph earlier, one of the fundamental differences to the perfect graph is the need for the JVM to profile the application, identify methods to compile and then compile them. This overhead is what leads to the warm-up time associated with Java applications running on traditional JVMs.

Many people ask why it is not possible to just take a snapshot of the compiled code the JVM is using when it gets to a steady state and reload it when the application is started again. The specification for the JVM imposes strict rules on how classes and methods are used, specifically around class loading and initialization. These restrictions, along with the possibility of method implementations changing between application invocations makes the reuse of compiled code impractical.

However, ReadyNow!, which is included in Azul Platform Prime, is able to do things that can almost eliminate application warm-up time.

We already know that the JVM profiles an application as it runs to identify methods to compile. To avoid the delay associated with warm-up, ReadNow! records profiling data from a running application into a file, specifically:

- A list of classes the JVM currently has loaded.
- A list of classes the JVM has initialized.
- Instruction profile data. This includes things such as whether a particular call has raised a NullPointerException, whether an array access has gone out of bounds and so on.
- Speculative optimization failure data that highlight optimization paths to avoid.

When the application is started again, the file containing the ReadyNow! profile can be used as input to the JVM. ReadyNow! reads this information and pre-compiles required methods at application start-up. ReadyNow! uses the information in the profile log to speculatively load all the classes that are required.

ReadyNow! will also initialize classes that do not require running code to be initialized. (One of the restrictions of the JVM specification is that classes can only be initialized via the instantiation of an object of that type or access to a static field or method of the class). Using this eager-loading and eager initialization of classes allows for the proactive compilation of most methods before the main() method of the application is run.

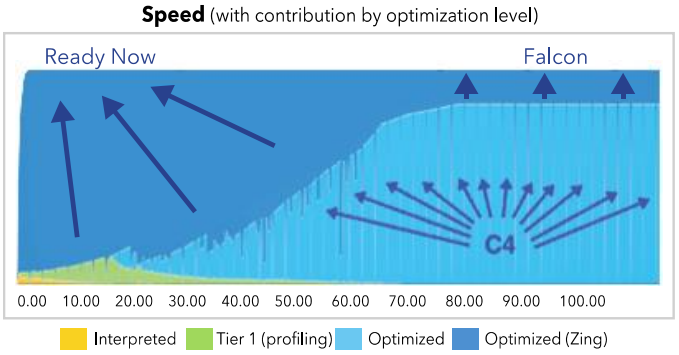
The overall effect of this is very similar to reloading a compiled code snapshot but with a number of significant advantages. One of those advantages is that ReadyNow! can benefit from far longer sampling periods than used by the JIT resulting in far fewer deoptimizations (typically as much as 80% less).

Conclusions

At the start of this paper, we saw what a perfect performance graph would look like and how a traditional JVM graph differed from that.

The graph below shows how the earlier example can be improved by the use of features in Azul Platform Prime

.



- The Falcon replacement for the C2 JIT helps to improve the overall speed of the application by applying optimizations not used before, such as the use of modern processor instructions and vector processing in more complex situations.
- ReadyNow! almost eliminates the problems of application warm-up and deoptimizations through the recording and reuse of JIT profiling information from previous runs of the application.
- The C4 GC (not covered in this paper) helps to eliminate pauses caused by garbage collection needing to suspend application threads while objects are relocated.

As you can see from this graph, not only does Azul Platform Prime get JVM performance very close to the perfect profile, it also raises the level of performance available to your application above that of traditional JVMs.

For additional information:

- Azul Platform Prime product overview
- ReadyNow! Data sheet
- C4 white paper
- Azul Platform Prime eval download page

Contact Azul

To discover how Azul Platform Prime can improve scalability, consistency and performance of all your Java deployments, contact:

385 Moffett Park Drive, Suite 115
Sunnyvale, CA 94089 USA
+1.650.230.6500

www.azul.com

info@azul.com