TECHNOLOGY WHITE PAPER

AZUL SYSTEMS®

Falcon: Improving Java Performance Through Better Compilation

Azul has developed a new compiler used internally in the JVM, called Falcon. This replaces the existing C2 compiler, providing much greater flexibility and maintainability.

Introduction

Java is a mature software platform that uses a number of techniques to make the lives of developers and administrators easier than more traditional languages like C and C++. Automatic memory management through the use of a garbage collector is the feature that most people think of with Java, but the use of bytecodes is another aspect that can have a significant impact on the performance of an application.

Azul has developed a new compiler used internally in the JVM, called Falcon. This replaces the existing C2 compiler, providing much greater flexibility and maintainability. Based on the open-source LLVM compiler project, Falcon uses a modular design, allowing much simpler and more rapid inclusion of new optimizations. As CPUs improve, providing more complex processing with fewer machine instructions, Falcon is able to rapidly take advantage of these features delivering better performance and throughput to Java applications, both existing and new.

Java: Write Once, Run Anywhere

The Java language has a human-readable syntax that enables the quick and easy development of applications ranging from the simple to the very complex. In order to run the application, the *source code* must first be compiled into a form that can be understood by the machine where the application will run.

Traditional languages like C and C++ use what is called ahead of time (AOT) compilation. A target machine, where the code will be deployed, is selected and a compiler for that machine architecture is used to compile the source code. Typically, the source code is compiled on the same machine architecture where it will be deployed but, that is not mandatory, as cross-compilation can be used. The resulting compiled code contains instructions that are specific to the target CPU architecture and operating system. This ties the compiled code to the deployment architecture so that an executable generated for Windows running on the x86 CPU architecture will not run on an ARM-based machine running Linux.

The Java platform takes a different approach to compilation. When a Java source file is compiled, a *class file* is generated. Rather than the class file being specific to the target machine configuration, it contains *bytecodes*. Bytecodes are low-level instructions but for a *virtual machine*, specifically the Java Virtual machine (JVM).

JVM Design

The Java Virtual Machine is an abstract representation of a computer that can run Java applications. Like a real computer system, it has an instruction set and manipulates various memory areas at run time.

The Java Virtual Machine knows nothing about the Java programming language, only the class file binary format. A class file contains JVM instructions (or bytecodes) and a symbol table, as well as other additional information.

Like real computer instruction sets, bytecodes consist of an opcode specifying the operation to be performed, followed by zero or more operands providing values to be operated on.

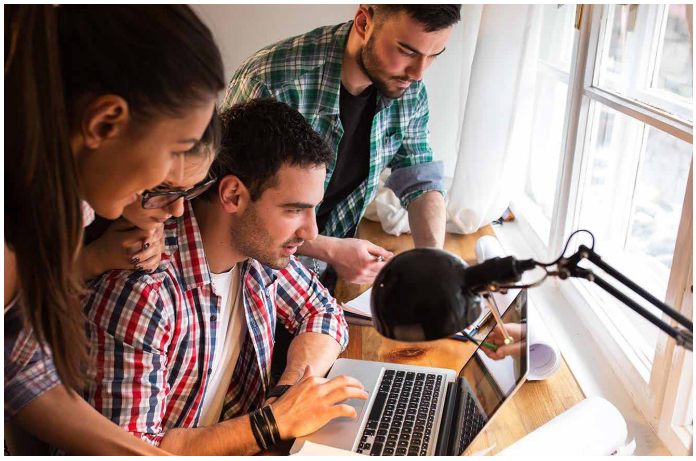
The JVM currently implements 149 bytecodes; three others are reserved for internal use and are not valid in a class file.

Execution Of Bytecodes

Since bytecodes are not specific to the platform where the application is being run, the JVM must convert the bytecodes to the correct set of instructions for the target

TABLE OF CONTENTS

hardware architecture and operating system. The original design of the JVM just took the bytecodes, in sequence, and converted them to the relevant machine-specific instructions. For some instructions this is a one-to-one mapping, for example, the iand bytecode can be converted to an *AND* instruction on an x86-based processor (other processors have similar instructions that have different names). For more complex bytecodes, such as invoking a method, a sequence of instructions is generated. This approach to executing the bytecodes is referred to as interpreted mode. Since bytecodes must be converted every time they are used this leads to significantly reduced performance compared to AOT compiled languages.



To provide better performance versions of the JVM since JDK 1.2 use an *adaptive compilation* technique. When an application starts, the JVM works in interpreted mode, as described above. However, it also profiles the code being executed, keeping statistics of how many times methods are called. When a method is called a pre-defined number of times, the method is considered to be a hot spot (hence the name of the JVM). At this point, a compiler is used to convert the bytecodes into native instructions that can be cached and reused for subsequent calls to the method. Since the bytecodes are similar to the P-code intermediate representation used by some compilers, the back-end code generator part of the compiler can be used in conjunction with other optimization techniques. This is called *Just In Time*, or JIT compilation.

JIT compilation was not something that was invented just for Java. The concept was first used with the LISP programming language in 1960. It is a technique that has been used by many languages both before and after Java was launched. SmallTalk pioneered a number of new JIT concepts, and both JavaScript and Microsoft's .NET Common Language Runtime (CLR) use it.

The OpenJDK JVM uses two separate JIT compilers called C1 and C2, sometimes referred to as *client* and *server*. The reason for having two is due to the way they work and the code they produce:

C1 JIT: This uses a basic code generator, which is designed to compile code as quickly as possible but does not have the ability to apply significant optimizations to the code that it produces. The idea is to eliminate the overhead of bytecode interpretation as quickly as possible for frequently used methods. Also, the generated code includes more profiling instructions that can gather more detailed statistics about how the code is executing.

C2 JIT: The compiler for C2 is much more sophisticated and can use a much larger number of optimizations to improve the efficiency of the code it generates. The profiling data gathered whilst the C1 produced code executes provides input to C2 so that it has a clear picture of the flow of execution of the application code. More aggressive optimization techniques necessitate more work on the part of the C2 JIT resulting in it taking longer to generate code. The benefits of C2 are typically much more apparent for longer running, server-side applications.

One of the techniques that the C2 JIT can use, given the profiling data from C1, is speculative optimizations. The idea is to optimize code based on how the application has behaved so far, not necessarily using all the original source code.

Let's look at a example of this using the following method:

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

Assume that the situation where *val* is greater than 10 is a very rare condition and that whilst the JVM was running with C1 JIT compiled code it never happened. Using the collected profiling data, the C2 JIT can speculate that *val* will not exceed 10 in the future so can eliminate the unnecessary (and time-consuming) code. In this case, C2 will compile the code as if it were written this way:

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

No computation of the log function is required, as long as val never exceeds 10 so the code will be heavily optimized for performance. If this speculation turns out to be wrong at some point, the uncommonTrap() method is called. Since the compiled code cannot execute correctly for this situation, the compiled code is invalidated, and execution returns to interpreted mode so the correct calculation can be performed. As profiling continues the code may be recompiled by C2 but with the additional computation included.

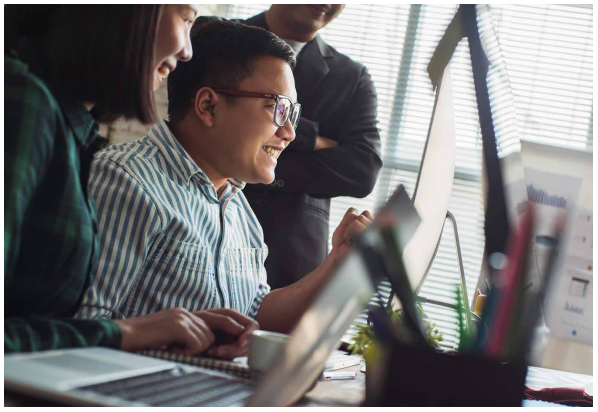
The OpenJDK JVM supports five tiers of execution:

Tier 0: Interpreted mode
Tier 1: C1 with full optimization (no profiling)
Tier 2: C1 with basic counters.
Tier 3: C1 with full profiling
Tier 4: C2

The typical use case for the tiers is 0 followed by 3 then 4, as explained earlier. Tier 1 is used when a method is determined to be trivial, in which case the code generated by C1 and C2 will be the same so recompiling with C2 will have no effect. Tier 2 is used when the queue of methods waiting to be compiled by C2 becomes too long, which will delay optimized compilation of the method. Using Tier 2, the method can execute approximately 30% faster until C2 can process the profiling data and recompile the method. Whilst running in tier 2 the JVM monitors how many times a method is invoked as well as whether it is executing a high iteration loop. It makes a lot of sense to compile a method that has a loop that executes the same set of instruction many times, even if the method running the loop does not get called frequently.

Why replace the existing C2 compiler?

The existing C2 compiler was aging poorly. As we'll explain later one of the key performance features of modern processor design is vectorization, which enables a CPU to execute a single instruction on multiple data (SIMD). Since C2 was initially developed in the late 1990s, this feature was added much later and did not integrate well with the core design. In addition, C2 has a very complex code base which makes it very hard to either fix bugs or test the compiler in isolation.



What was required was an entirely new replacement. When approaching the development of Falcon, the engineers had two primary goals:

To outperform the code generated by the competition. The OpenJDK source code uses a different version of C2, and we wanted to make sure that our new compiler would be better than that. The C2 OpenJDK JIT sets a high bar for performance, being one of the best JITs that is available today.

The velocity of performance improvements. Given that CPU architectures are continuing to evolve and adding significant new features to improve application performance it was essential that these could be exploited as quickly and efficiently as possible.

The LLVM Project

Rather than developing an entirely new JIT compiler, the team at Azul looked around to see if there was any existing work that they could use as a starting point. The team's research led them to an open source project called LLVM.

The LLVM compiler infrastructure project is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends. The project was started in 2000 at the University of Illinois at Urbana–Champaign. Originally, LLVM was an acronym for Low-Level Virtual Machine. This has since been dropped, as the project now has much broader scope including compiler front ends, intermediate representation, backends, a linker, and debugger.

Profiling data from the C1 JIT is critical to optimizations that can be exploited by the C2 JIT. LLVM provides an extensive set of metadata and attributes, which provide the semantics necessary for the optimizer code.

The LLVM project is supported with significant contributions from a number of highly respected IT companies. Amongst these are Intel, Apple, ARM, NVidia, IBM, and Microsoft. Being able to take advantage of such an effectively large engineering team was one of the major

Azul's engineers also extended LLVM to enable it to, in effect, question the JVM during optimization via callback methods. This allows for lazy querying as the optimizer runs, which is more efficient during JIT compilation.

factors in Azul's decision to use LLVM. In addition, LLVM's stability and widespread adoption, as well as its support for new micro-architectures made it a clear choice.

Although LLVM provided a great starting point for Falcon, considerable work was required to integrate it into the JVM. Much of this work was in the area where the JVM handles memory management automatically. The JIT compiler needs to work closely with the garbage collector (GC) to ensure that it uses safepoints correctly. A thread is said to be at a safepoint when the thread's representation of it's Java machine state is well described, and can be safely manipulated and observed by other threads.

Azul's engineers also extended LLVM to enable it to, in effect, question the JVM during optimization via callback methods. This allows for lazy querying as the optimizer runs, which is more efficient during JIT compilation.

Azul has subsequently contributed back much of this work to the LLVM project.

Falcon Optimizations For Modern Processors

With the basics of LLVM in place, more advanced optimizations could be used in compiling methods.

As mentioned earlier, one of the more recent advances in CPU performance related features is in the area of *vectorization*. These are supported by Intel and AMD processors through extensions to the x86 instruction set, referred to as Advanced Vector Extensions (AVX). These were first introduced in 2011 as part of the Sandy Bridge range of processors from Intel and later in the Bulldozer range of processors from AMD.

AVX instructions have developed over time, starting with Streaming SIMD Extensions (SSE), first introduced in the Pentium III in 1999 followed more recently by AVX, AVX2, and AVX-512. This is one of the critical challenges for JIT compilers, to use the right instruction set architecture (ISA) to ensure the highest level of optimization.

Vectorization exploits the idea of Single Instruction, Multiple Data (SIMD) processing, which implements data level parallelism. CPUs that support this feature include one or more very wide registers (currently either 256 or 512 bits). These registers can hold multiple data values, for example, a 256-bit wide register can hold eight 32-bit single-precision floating point numbers or four 64-bit double-precision floating point numbers.

An AVX enabled processor can simultaneously execute an operation on multiple data operands in a single instruction. For example, if we have a simple loop that multiplies together values from two separate arrays:

```
for (i = 0; i < size; i++) {
        a[i] = b[i] * c[i];
}
```

Without AVX each value of the arrays b and c must be loaded into registers before performing the multiplication operation. The number of operations required is equal to the size of the arrays. With AVX, assuming we have 32-bit values and 512-bit wide registers, 16 array elements can be loaded. Only a single instruction is required to multiply all 16 values (the operation happens in parallel on all data elements). This reduces the time to process the arrays by a factor of 16.

There are now three versions of AVX available with different processor versions:

AVX: The original, using 16 256-bit registers including backward compatible support for the earlier SSE instructions that used 128-bit registers. This also supports three-operand instructions, where the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form a = a + b can now use a non-destructive three-operand form c = a + b, preserving both source operands.

AVX2: This was introduced in the Haswell range of processors from Intel. In addition to the AVX features, it expands most vector integer SSE and AVX instructions to 256-bits as well as including three-operand general-purpose bit manipulation and multiply instructions.

AVX-512: This was introduced in the Knights Landing range of processors from Intel. This expands the registers used for AVX to 512-bits. It also adds some new instructions that can handle four operands, which enables conditional copying from one of two registers to a third register based on a bitmask stored in a fourth register.

Because AVX support has been available for some time the existing C2 compiler does have limited support for this. In the case of our simple example, the JIT would use the AVX instructions where available.

Falcon can exploit AVX instructions in many more places than C2 can. Given the very complex code base of C2, it is not practical to be able to integrate new opportunities to use features like AVX. Falcon's modular more open design makes this very straightforward.

Let's look at another example. Rather than using the simple multiplication of each element in two arrays let's add some more logic and change the operation to an addition:

```
for (int i = 0; i < a.length; i++) {
    if ((b[i] & 0x1) == 0) {
        a[i] += b[i];
    }
}
```

In this case, we are using a conditional to only add the value of elements in array b to a corresponding element in array a if the element in array b is even. This is definitely something that can be optimized using AVX instructions. If we look at the code generated by C2 for this, we find that it performs each operation on the arrays as a separate instruction. Falcon can identify this as an opportunity to use vectorization and generates the appropriate AVX instruction sequence.

This is just one example of optimization that is possible with Falcon that is not possible with C2. Azul's engineers are continually working on identifying more places where vectorization, as well as other low-level processor optimizations, can be utilized to deliver better overall performance for applications running on the JVM.

Vectorization is not the only technique that Falcon uses to improve the performance of code running on the JVM. There are two other areas that Falcon finds easy to optimize in comparison to code produced by the C2 JIT compiler. These are primarily relevant to newer features in Java and more modern styles of coding.

Firstly, there is field finality. Falcon can optimize these efficiently, whereas C2 does not optimize these at all. Falcon will generate correctly working code even if a developer use reflection to mutate a final value. Immutable objects have become very popular with the inclusion of functional style programming constructs like streams, introduced in JDK 8. JDK 9 includes factory methods for collections that return unmodifiable collections, and JDK 10 adds new methods to the collections API to return unmodifiable copies.

The second area is where an application uses heavily abstracted code with lots of functions and small libraries. The C2 compiler often struggles with this type of code due to the complexities of the heuristics and limits on how much code can be inlined. Falcon due to its use of LLVM does not have these limitations. This is especially useful for the case where an application uses several independently developed libraries (something that is very common in complex enterprise applications).

Summary

The Falcon JIT compiler forms part of the Zing JVM, which has been developed specifically with the purpose of delivering the best possible performance for Java applications.

Using the LLVM open source project as a starting point and leveraging the wealth of contributions from many companies and individuals has enabled an extremely high-performance JIT compiler to be developed.

For many types of server type workloads, Zing will not only radically reduce pause times associated with memory management and garbage collection but will also increase the throughput of the application.

Zing provides a very cost-effective solution to enable enterprise IT departments to meet their Service Level Agreements, both internally and externally. With more and more applications moving to the cloud, improving throughput and reducing latency is easy to calculate in reduced running costs.

Find out more

Try Zing free for 30 days, available from azul.com/zingtrial

For up-to-date Falcon benchmark results, send a note to info@azul.com

Azul Systems, Inc.
385 Moffett Park Drive, Suite 115
Sunnyvale, CA 94089 USA
+1.650.230.6500
www.azul.com