Real-time Java, Part 2: Comparing compilation techniques

Issues in static and dynamic native Java code compilation

Mark Stoodley
Kenneth Ma
Marius Lut

April 17, 2007

This article, the second installment in a six-part series on real-time Java™, looks at some of the issues involved in native code compilation for the Java language. Neither dynamic (Just-in-time) nor static (Ahead-of-time) compilation alone can meet the requirements for all Java applications. The authors compare the two compilation technologies in various execution environments and show how they complement each other's strengths.

View more content in this series

Java application performance has sometimes been a source of heated debate in the development community. Because the language was designed to be interpreted to support the critical goal of application portability, early Java runtimes provided performance levels significantly lower than those possible with compiled languages such as C and C++. Although such languages can perform at a higher level, the generated code can be executed on only a limited number of systems. Over the course of the last decade, Java runtime vendors have developed sophisticated dynamic compilers, commonly known as Just-in-time (JIT) compilers. JIT compilers selectively compile the most frequently executing methods to native code while programs are running. Delaying native code compilation to run time rather than compiling before the program runs, as programs written in C or C++ do, maintains the portability requirement. Some JIT compilers even compile all code without using an interpreter, but these compilers still preserve portability for Java applications by operating while the program is executing.

Thanks to many advances in dynamic compilation technology, modern JIT compilers can produce application performance that matches statically compiled performance in C or C++ for a wide variety of applications. Still, many software developers think -- from experience or anecdotal evidence -- that dynamic compilation can significantly interfere with program operation because the compiler must share the CPU with the application. Some developers stridently call for static compilation for Java code with the firmly held belief that it will solve this performance problem. It's true that for some applications and execution environments, static compilation can tremendously
help Java performance or is the only practical option. But many complexities are involved in achieving good performance when compiling a Java application statically. The average Java developer may not fully appreciate the advantages of dynamic JIT compilers.

This article looks at some of the issues involved in compiling the Java language both statically and dynamically, with a focus on implications for real-time (RT) systems. It briefly describes how Java language interpreters operate and then describes the advantages and drawbacks of native code compilation performed by modern JIT compilers. It introduces the AOT compilation technology IBM® has released in WebSphere® Real Time and covers some of its advantages and disadvantages. It then compares and contrasts the two compilation strategies and points out several application areas as well as execution environments where AOT compilation is probably the better approach. The main point is that these two compilation technologies are not mutually exclusive: both have advantages and drawbacks that impact the kinds of applications for which each technology is most effective.

**Executing a Java program**

A Java program is initially compiled through the Java SDK's **javac** program into a native platform-neutral format known as class files. This format can be viewed as the Java platform because it defines all the information needed to execute a program written in the Java language. The execution engine for Java programs, also known as the Java Runtime Environment (JRE), includes a virtual machine that implements the Java platform for a particular native platform. For example, the Linux®-based Intel® x86 platform, the Sun Solaris platform, and the IBM System p™ platform running on the AIX® operating system each has a JRE. These JRE implementations implement all the native support needed to execute correctly a program written for the Java platform.

In fact, the operand stack’s size has a practical limit, but it would be extremely rare for a programmer to write a method that exceeded that limit. The JVM provides a safety check that notifies any programmer who does manage to create such a method.

One important piece of the Java platform program representation is a sequence of bytecodes that describe the operations that each method in a Java class performs. Bytecodes describe calculations using a theoretically infinitely large operand stack. This stack-based program representation provides platform neutrality because it doesn't depend on the number of registers available in any particular native platform's CPU. The operations that can be performed on the operand stack are all defined independently of any native processor's instruction set. Execution of these bytecodes is defined by the Java Virtual Machine (JVM) specification (see Related topics). When executing Java programs, any JRE for any particular native platform must adhere to the rules set out by the JVM specification.

Because few native platforms are stack-based (the Intel X87 floating-point coprocessor is one notable exception), most native platforms can't execute Java bytecodes directly. To address this problem, early JREs executed Java programs by interpreting the bytecodes. That is, the JVM operates in a loop that repeatedly:

1. Fetches the next bytecode to execute.
2. Decodes it.
3. Fetches the required operands from the operand stack.
4. Performs the operation according to the JVM specification.
5. Writes any result back on the stack.

The advantage of this approach is simplicity: The JRE developers need only write the code to handle each type of bytecode. And because fewer than 255 bytecodes are available to describe operations, the implementation cost is low. The drawback, of course, is performance: a problem used by many to condemn the Java platform in its early days despite its many other advantages.

Addressing the performance gap with languages such as C or C++ meant developing native code compilation for the Java platform in such a way that portability wasn't sacrificed.

**Compiling Java code**

Despite anecdotal evidence that Java programming's write-once-run-everywhere mantra might not be strictly true in all cases, it does work for a wide variety of applications. Native compilation, on the other hand, is by its very nature platform-specific. So how does the Java platform achieve native compilation performance without sacrificing platform neutrality? The answer is, and has been for a decade, dynamic compilation in the form of a JIT compiler (see Figure 1):

**Figure 1. JIT compiler**

With a JIT compiler, Java programs are compiled one method at a time as they execute into the native processor's instructions to achieve higher performance. The process involves generating an internal representation of a method that's different from bytecodes but at a higher level than the target processor's native instructions. (The IBM JIT compiler uses a sequence of expression trees to represent the method's operations.) The compiler performs a sequence of optimizations to improve quality and efficiency and finally a code-generation step to translate the optimized internal representation to the target processor's native instructions. The generated code relies on a runtime environment to perform activities such as ensuring a type cast is legal or allocating certain types of objects that are impractical to perform directly in the code itself. The JIT compiler operates on a
compilation thread that's separate from the application threads so that the application doesn't need to wait for a compilation to occur.

Also depicted in Figure 1 is the profiling framework that observes the executing program's behaviour by periodically sampling the threads to find frequently executing methods. It also provides facilities for specialized profiling versions of methods to store dynamic values that might not change in this execution of the program.

Because this JIT compilation procedure occurs while the program executes, platform neutrality is maintained: the neutral Java platform code is still the form of distribution. Languages such as C and C++ lack this advantage because their native compilation step is performed before the program executes; the native code is what's distributed to the (native platform) execution environment.

**Challenges**

Although platform neutrality is maintained with JIT compilation, it comes at a price. Because compilation happens at the same time as program execution, the time it takes to compile the code is added to the program's running time. As anyone who has ever built a nontrivial C or C++ program can relate, compilation is not usually a quick process.

To address this drawback, modern JIT compilers take one of two approaches (and, in some cases, both). The first approach is to compile all the code but without performing any expensive analyses or transformations so that the code is generated quickly. The code can be generated so quickly that the overhead observed from compilation, though noticeable, is easily hidden behind the performance improvement resulting from repeatedly executing native code. A second approach is to devote compilation resources to only a small number of methods that execute frequently, often called the **hot** methods. Low compilation overhead is maintained that can be more easily hidden behind the performance benefit from repeatedly executing the hot code. Many applications spend time executing only a small number of hot methods, so this approach effectively reduces compilation's performance cost.

A fundamental complexity for dynamic compilers is that of balancing the need to know how much a method's execution contributes to the whole program's performance with the expected benefit from compiling the code. As an extreme example, after a program executes, you'd have perfect knowledge about which methods contributed most to this particular execution, but compiling those methods has no value because the program has already completed. At the other end of the spectrum, before the program has executed, no knowledge is available about which methods are important, but the potential benefit for each method is maximized. Most dynamic compilers operate somewhere between these two extremes by balancing the need to know what's important with the expected benefit from that knowledge.

The fact that the Java language requires classes to be loaded dynamically has a significant impact on a Java compiler's design. What if code is compiled that references another class that hasn't yet been loaded? An example might be a method that reads the value of a static field for a class that hasn't yet been loaded. The Java language requires that the first execution of a reference to
a class causes that class to be loaded and resolved into the current JVM. Until the first execution, the reference is unresolved, which means there’s no address to load that static field from. How does the compiler deal with this possibility? The compiler generates code that causes the class to be loaded and resolved, if it has not yet been loaded. Once the class has been resolved, the original code location is modified in a thread-safe way to access the static field’s address directly because that address is then known.

Considerable effort has gone into the IBM JIT compiler to use safe but efficient code-patching techniques so that, after the class has been resolved, the native code that executes simply loads the field’s value as if the field had been resolved at compile time. The alternative is to generate code that always checks to see if the field is resolved before finding out where the field is and then loading the value. For unresolved fields that become resolved and frequently accessed, this naive procedure can be a huge performance problem.

Benefits of dynamic compilation

Compiling a Java program dynamically has important benefits that permit even better code generation than is typically possible for statically compiled languages. Modern JIT compilers often insert hooks into generated code to collect information about how the program is behaving so that if the methods are selected for recompilation, that dynamic behaviour can be better optimized.

A good example of this approach is collecting the length of a particular arraycopy operation. If the length is found to be mostly constant every time it executes, then specialized code for that most frequently used arraycopy length can be generated, or a sequence of code better tuned for that length can be invoked. Because of the nature of memory systems and instruction-set designs, the best generic routine to copy memory is rarely as fast as code written to copy a particular length. For example, copying 8 bytes of aligned data might require one or two instructions to copy directly compared to perhaps as many as 10 instructions to copy those same 8 bytes using a general copy loop capable of handling any number of bytes with any alignment. Even if such specialized code is generated for one particular length, however, the generated code must also correctly perform copies for other lengths. The code is simply generated to be faster for the commonly observed length so that, on average, performance is improved. This type of optimization is often impractical for most statically compiled languages because lengths that are constant for all possible executions are more rare than lengths that are constant in one particular program execution.

Another important example of this kind of optimization is class-hierarchy-based optimization. A virtual method invocation, for example, involves looking at the class of the receiver object for the call to discover which actual target implements the virtual method for the receiver object. Research has shown that most virtual invocations have only a single target for all receiver objects, and JIT compilers can generate more-efficient code for a direct call than for a virtual invocation. By analyzing the class hierarchy’s state when the code is compiled, the JIT compiler can find the single target method for a virtual invocation and generate code that directly calls the target method rather than performing the slower virtual invocation. Of course, if the class hierarchy changes and a second target method becomes possible, then the JIT compiler can correct the originally generated code so that the virtual invocation is performed. In practice, these corrections are rarely
required. Again, the potential need to make such corrections makes performing this optimization statically troublesome.

Because dynamic compilers typically focus compilation effort on only a small number of hot methods, more-aggressive analyses can be performed to generate even better code so that the payback for compilation is much higher. In fact, most modern JIT compilers also support recompiling methods that are found to be very hot. These frequently executed methods can be analyzed and transformed with even extremely aggressive optimizations usually found in static compilers (which have lower emphasis on compilation time) to generate even better code and higher performance.

The combined effect of these improvements, and others like them, is that for a large number of Java applications, dynamic compilation has bridged the gap and, in some cases, even surpasses the performance possible with static native compilation for languages such as C and C++.

**Drawbacks**

Nonetheless, dynamic compilation does have some drawbacks that make it a less than ideal solution in some situations. For example, because it takes time to identify frequently executed methods as well as to compile those methods, applications typically go through a warm-up period in which performance has not yet reached its peak. This warm-up period can be a performance issue for a number of reasons. First, the large number of initial compilations can directly impact application start-up time. Not only do these compilations delay the application reaching a stable state (imagine a Web server going through an initialization phase before reaching the point of being able to perform useful work), but the methods executing frequently during this warm-up phase might not contribute significantly to the application's steady-state performance. Performing JIT compilations that delay start-up yet do not significantly improve the application's long-term performance are particularly wasteful. Although all modern JVMs perform tuning to mitigate start-up penalties, the problem can't be completely eliminated in all cases.

Second, some applications simply cannot tolerate the delays associated with dynamic compilation. An interactive application such as a GUI interface is an example. In this case, compilation activity can adversely affect the user's experience without substantially improving the application's performance.

Finally, applications designed to function in real-time environments with strict task deadlines may not be able to tolerate either the nondeterministic performance effects of compilations or the memory overhead of the dynamic compiler itself.

So, although JIT compilation technology has been developed to the point where it can provide performance at a level comparable to or even better than static language performance, dynamic compilation is simply not the right fit for some applications. In these scenarios, Ahead-of-time (AOT) compilation for Java code may be the right solution.
AOT Java compilation

In principle, native compilation for the Java language should be a straightforward application of the compilation technologies developed for traditional languages such as C++ or Fortran. Unfortunately, the dynamic nature of the Java language itself introduces additional complexities that impact the quality of statically compiled code for Java programs. But the basic idea is still the same: Generate native code for Java methods before the program executes so that the native code can be used directly once the program is run. The goal is either to avoid the JIT compiler's run-time performance or memory cost or to avoid the interpreter's early performance overhead.

Challenges

Dynamic class loading, which is a challenge for the dynamic JIT compiler, is an even more significant issue for AOT compilation. A class can't be loaded until the executing code makes a reference to that class. Because AOT compilation occurs before the program executes, the compiler can't make any assumptions about which classes have been loaded. That means the compiler can't know the address of any static field, the offset of any instance field of any object, or the real target of any invocation, even for direct (that is, nonvirtual) calls. Making an assumption about any of this information that turns out to be false when the code executes means the code is incorrect and Java conformance has been sacrificed.

Inlining

Inlining is a technique aiming to generate code that avoids the overhead of prologs or epilogs at run time by inserting the function's call code into the caller's function. But perhaps inlining's greatest benefit is that the scope of code visible to the optimizer grows, enabling higher-quality code generation. Here's an example of code before inlining: int foo() { int x=2, y=3; return bar(x,y); } final int bar(int a, int b) { return a+b; } If the compiler can prove that this bar is the one that's invoked inside foo(), then the code from bar can replace the call to bar() in foo(). In this case, the bar() method is final, so it must be the one called in foo(). Even in virtual cases, a dynamic JIT compiler can often speculatively inline the target method's code and get it right in the vast majority of cases. The compiler generates this code: int foo() { int x=2, y=3; return x+y; }

In this example, an optimization called value propagation followed by simplification could generate code that simply returns 5. Without inlining, this optimization can't be performed, and much lower performance results. If the bar() method is unresolved, as is the case with static compiling, this optimization isn't possible, and the code must perform the virtual invocation. At run time, a different bar that multiplies rather than adds the two numbers might be the invocation's actual target. For this reason, inlining can't be naively performed during static compilation of Java programs.

Because the code can execute in any environment, the class files might not be the same as when the code was compiled. For example, one JVM instance might load a class from a particular location from disk, and a subsequent instance might load that class from a different location or even over the network. Imagine a development environment where bug fixes are being made: A class file's contents can change from one program execution to the next. Moreover, the Java code
might not even exist until the program runs: Java reflection services, for example, often generate new classes at run time to support the program’s activities.

The lack of knowledge about statics, fields, classes, and methods means that most of the optimization framework in a Java compiler is severely hampered. Inlining, which is probably the most important optimization applied by static or dynamic compilers, can no longer be applied because the compiler has no information about what the target method of an invocation is.

AOT code must therefore be generated with every static, field, class, and method reference unresolved. On execution, every single one of these references must be updated with the correct values for the current run-time environment. This process can have a direct penalty to first-execution performance because all references are resolved on that first execution. Subsequent executions, of course, will benefit from the result of patching the code so that the instance or static field, or method target, is more directly referenced.

On top of that, the generated native code for a Java method typically requires values that can be used only in a single JVM instance. For example, the code must call certain runtime routines in the JVM runtime to perform specific actions, such as looking up an unresolved method or allocating memory. These runtime routines’ addresses can be different each time the JVM is loaded into memory. AOT-compiled code, therefore, needs to be bound into the JVM’s current execution environment before it can be executed. Other examples are addresses of strings and the internal locations of constant pool entries.

In WebSphere Real Time, AOT native code compilation is performed with a tool called jxeinajar (see Figure 2). This tool either applies native code compilation to all methods of all classes in a JAR file or applies it selectively to the methods of interest. The results are stored into an internal format known as a Java eXEcutable (JXE), but could just as easily be stored into any persistent container.

**Figure 2. jxeinajar**

You might think that compiling all the code statically is the best approach because it results in the largest amount of native code executing at run time. But several trade-offs could be made here. The more methods compiled, the more memory the code occupies. Compiled native methods are roughly 10 times larger than the bytecode: native code is itself less dense than the bytecodes, and the additional metadata about the code must be included so that the code can be bound into the JVM and properly executed when exceptions occur or stack traces are requested. The JAR
files that make up an average Java application typically contain many methods that are rarely executed. Compiling those methods carries a memory penalty with little expected benefit. This size penalty carries with it associated costs to store the code on disk, to bring the code off disk and into the JVM, and to bind the code into the JVM. Unless the code is executed several times, these costs might not be offset by the performance benefit of native code versus interpretation.

Acting against the size issue is the fact that calls between compiled and interpreted methods (that is, when a compiled method calls an interpreted method or vice versa) can be more expensive than calls from interpreted method to interpreted method or from compiled method to compiled method. A dynamic compiler mitigates this cost by eventually compiling all the interpreted methods that are frequently called by JIT compiled code, but without a dynamic compiler, this cost can't be hidden. So if methods are selectively compiled, care must be taken to minimize transitions from compiled methods to methods that aren't compiled. Selecting the right set of methods to avoid this problem for all possible executions can be difficult.

Benefits

Although AOT-compiled code has the drawbacks and challenges we've outlined, compiling Java programs ahead of time can yield performance benefits, particularly in the environments where dynamic compilers are not always an effective solution.

You can accelerate application start-up by carefully using AOT-compiled code because this code, although typically slower than JIT-compiled code, can be many times faster than interpretation. Furthermore, because the time to load and bind AOT-compiled code is typically less than the time to detect and dynamically compile an important method, you can achieve that performance earlier in a program's execution. Similarly, interactive applications can benefit from native code performance quickly without the cost of dynamic compilation that causes poor responsiveness.

RT applications can also derive an important benefit from AOT-compiled code: more-deterministic performance that exceeds interpreted performance. The dynamic JIT compiler that WebSphere Real Time uses has been specially adapted for use in RT systems. It makes the compilation thread operate at a lower priority than RT tasks and is tuned to avoid generating code with severely nondeterministic performance effects. In some RT environments, however, even the presence of the JIT compiler is unacceptable. Such environments typically require the most strict control of deadline management. In these cases, AOT-compiled code can provide better raw performance than interpreted code without impacting the degree of determinism that can be achieved. Eliminating the JIT compilation thread eliminates even the performance impact of preempting it when a higher-priority RT task must be initiated.

The scorecard

Dynamic (JIT) compilers support platform neutrality and generate high-quality code by exploiting the dynamic behaviour of an application's execution and knowledge about loaded classes and their hierarchy. But JIT compilers have only a limited compile-time budget and can impact the program's run-time performance. Static (AOT) compilers, on the other hand, sacrifice platform neutrality and code quality because they can't exploit the program's dynamic behaviour or have any knowledge about loaded classes or the class hierarchy. AOT compilation has an effectively unlimited compile-
time budget because AOT compilation time has no run-time performance impact, though in practice developers won't wait forever for the static compilation step.

Table 1 summarizes several characteristics of dynamic and static compilers for the Java language as discussed in this article:

**Table 1. Comparing compilation techniques**

<table>
<thead>
<tr>
<th></th>
<th>Dynamic (JIT)</th>
<th>Static (AOT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform neutrality</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Code quality</td>
<td>High</td>
<td>Good</td>
</tr>
<tr>
<td>Exploit dynamic behaviours</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Knowledge of classes and hierarchy</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Compile-time budget</td>
<td>Limited, has run-time cost</td>
<td>Much less limited, no run-time cost</td>
</tr>
<tr>
<td>Run-time performance impact</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>What to compile</td>
<td>Needs care, handled by JIT</td>
<td>Needs care, handled by developer</td>
</tr>
</tbody>
</table>

Both technologies require careful selection of the methods to be compiled to achieve the highest performance. For dynamic compilers, the compiler itself makes this decision, whereas for static compilers, the selection is up to the developer. Having the JIT compiler choose the methods to be compiled may or may not be an advantage, depending on how well the compiler's heuristics work in a given situation. In the majority of cases, we believe it is a benefit.

Because they can best optimize a running program, JIT compilers are better at delivering the steady-state performance that matters most to a large number of production Java systems. Interactive performance is best covered by static compilation because no run-time compilation activity interferes with the user's response-time expectations. Start-up and deterministic performance can be addressed to some extent by tuning dynamic compilers, but static compilation can deliver the fastest start-up and the highest levels of determinism when it's needed. Table 2 compares the two compilation technologies in four different execution environments:

**Table 2. Where each technology is best**

<table>
<thead>
<tr>
<th></th>
<th>Dynamic (JIT)</th>
<th>Static (AOT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start-up performance</td>
<td>Tunable, but not so good</td>
<td>Best</td>
</tr>
<tr>
<td>Steady-state performance</td>
<td>Best</td>
<td>Good</td>
</tr>
<tr>
<td>Interactive performance</td>
<td>Not so good</td>
<td>Good</td>
</tr>
<tr>
<td>Deterministic performance</td>
<td>Tunable, but not best</td>
<td>Best</td>
</tr>
</tbody>
</table>

Figure 3 shows the general trend in start-up performance and steady-state performance:
Figure 3. Performance AOT versus JIT

Performance with a JIT compiler is initially very low because methods are initially interpreted. As more methods are compiled and the JIT spends less time performing compilations, the performance curve grows and finally begins to reach peak performance. AOT-compiled code, on the other hand, starts much higher than interpreted performance but is unlikely to be as high as can be achieved via the JIT compiler. Binding the static code into the JVM instance incurs some cost, so performance initially dips lower than its steady-state value. But the steady-state level is achieved much more quickly than it is with a JIT compiler.

No one native code compilation technology is suitable for all Java execution environments. Each technology is generally strong where the other is weak. For this reason, both compilation technologies are required to meet Java application developers' demands. In fact, static and dynamic compilation can be used together to deliver the broadest possible performance boost -- but only if platform neutrality, one of the Java language’s main selling points, is not an issue.

Summary

This article explored the issue of native code compilation for the Java language, focusing on the main question of whether dynamic compilation in the form of a JIT compiler or static AOT compilation is better.

Although dynamic compilers have matured dramatically in the last decade to the point where a large variety of Java applications can match or exceed performance achievable via implementation in a statically compiled language such as C++ or Fortran, dynamic compilation is still less appropriate in several types of applications and execution environments. AOT compilation, though often touted as a panacea to the drawbacks of dynamic compilation, faces several challenges to providing the full potential of native compilation because of the dynamic nature of the Java language itself.

Neither one of these technologies can solve all the requirements for native code compilation in Java execution environments but instead are tools to be used where they are each most effective. The two technologies are complementary. Runtime systems using both compilation models appropriately will yield benefits to developers and users across a tremendous spectrum of application environments.
Related topics

- **Real-time Java series**: Read the other parts in this series.
- **Java Virtual Machine Specification**: The second edition of the JVM specification is available for download.
- **JSR 1: Real-time Specification for Java**: You'll find the RTSJ at the Java Community Process site.
- **IBM WebSphere Real Time V1.0 delivers predictable response times using Java standards**: Read the product announcement for WebSphere Real Time.
- **WebSphere Real Time**: WebSphere Real Time lets applications dependent on a precise response times take advantage of standard Java technology without sacrificing determinism.
- **Real-time Java technology**: Visit the authors' IBM alphaWorks research site to find cutting-edge technologies for RT Java.

© Copyright IBM Corporation 2007

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)