IBM High Performance Compiler for Java

By Ven Seshadri

IBM has developed an optimizing native code compiler for Java. This article describes the differences between static and Just-In-Time compilation, the basic structure of the AIX compiler, some implementation highlights, performance results, and the current status of the system.

Most Java Virtual Machines contain a Just-In-Time (JIT) compiler to improve performance. JIT compilers turn Java™ bytecodes into native (object) code on-the-fly as they are loaded into the virtual machine. The virtual machine then executes the generated object code directly instead of interpreting bytecodes. Because this translation occurs at execution time, the speed requirements for compilation limit the performance optimization of a JIT compiler.

An optimizing native code compiler for Java compiles Java bytecodes directly into native (object) code just as compilers for C/C++, FORTRAN, or COBOL. Unlike JIT compilers, this static compilation occurs only once—before execution time. Thus, traditional resource-intensive optimization techniques (such as dataflow analysis or interprocedural optimization) can be applied to improve the performance of the generated code.

Since the static native code compiler adheres to the machine-independent bytecode specification, statically compiling a Java application to multiple platforms results in object code that produces identical results on each platform. However, some important distinctions exist between statically compiled Java and dynamically interpreted (or JIT-compiled) Java, which we discuss in the next section.

Early beta-level versions of the optimizing native code compiler of Java for both AIX® and Windows NT™ are freely available at IBM's alphaWorks™ Web site (http://www.alphaworks.ibm.com).

Static vs. Just-In-Time Compilation

The Java programming language and Application Programming Interfaces (APIs) contain several dynamic language features, the most important being dynamic bytecode loading and Release-to-Release Binary Compatibility (RRBC). These features require late (load-time) binding capabilities, which are easily supported within a virtual machine. A Just-In-Time compiler for Java supports these features simply by using the late binding facilities of the Java Virtual Machine.

A static compiler uses the bytecodes for a Java application to generate either a set of shared objects or Dynamic Link Libraries (DLLs), or a statically bound executable before the application executes. Although

Ven Seshadri

our current implementation does not support dynamic bytecode loading or RRBC, these features can be implemented in a statically compiled context with some additional overhead.

If the runtime system includes a JIT compiler, dynamic bytecode loading can be implemented within a static compilation framework. The JIT compiler would be invoked to create a DLL from the bytecodes being loaded. Then the runtime system can load this DLL like any other application DLL. For this to work correctly, the bytecodes (or equivalent information) for classes referenced directly and indirectly by the dynamically loaded class must be present at runtime. This allows the compiler to generate the appropriate references to instance data and method tables in those classes.

The RRBC problem in object-oriented languages like Java arises because instance fields and class methods are referenced from other classes indirectly, usually by base+offset addressing. A static or JIT compiler determines the layout of instance method tables and object instance data of a class at compile time. It also generates code in client classes of that class, which contains inline offsets into these structures.

If a class changes in a way that causes the layout of the instance method tables or instance data to change (for example, adding an instance method or deleting a field), all of its client classes must be recompiled. This recompilation is unnecessary for a JIT compiler because the layouts of instance data and instance method tables are determined at class load time, which is the same as JIT compile time. References to instance fields and methods of other classes in the bytecodes are by name. These are turned into base+offset addressing after the class is loaded.

A statically compiled framework offers several ways to support RRBC. One approach is to extend the system linker and loader to understand special relocations for offsets into instance data and instance method tables. The compiler would generate these special relocations in the object code instead of inline offsets, which would then cause the linker or loader to generate the appropriate inline offset. This approach would require support from various operating system facilities and also increase load time.

An optimizing native code compiler for Java compiles Java bytecodes directly into native (object) code.

Another approach is to use object-oriented runtime support systems, such as IBM's System Object Model (SOM) which provides language-neutral object services such as RRBC. This method would use pre-existing services, but these services tend to impose a noticeable performance penalty.

A third approach would be to generate all references to instance method tables and instance data through an extra level of indirection—the compiler would generate temporaries to hold the actual offsets used in base+offset addressing. The compiler would then generate DLL initialization code to set appropriate values in these temporaries. This approach would not require special support from the operating system, but would impose a runtime penalty with increased load time and an additional level of indirection to access method tables and instance data.

Basic System Structure for the AIX Compiler

Figure 1 shows the basic structure of the compiler. The input is a single Java bytecode (`.class`) file, which contains a single class in the application. A translator processes the bytecodes to produce an internal compiler Intermediate Language (IL) representation of the class. The common back end from IBM's XL family of compilers for the RS/6000™ is then used to turn this intermediate representation into an object module (`.o` file) that is linked with other object modules from the application and libraries to produce an executable program.

The libraries implement garbage collection, the Java APIs, and various system routines to support object creation, threads,

exception handling, and application startup and termination. The compiler will also accept Java source code as input; if so, it invokes the AIX Java Development Kit's Java source-to-bytecode compiler (javac) to first produce bytecodes.

Currently, the compiler has a simple command-line interface. However, it is easier to develop optimized native code by using the IBM VisualAge™ for Java application development tool, with extensions to support this compiler (beta availability for these extensions is expected in the future).

These extensions will provide a seamless integration between the Integrated Development Environment (IDE) and the compiler. They enhance the IDE "Export" Smartguide so that when the Java program is ready to be deployed, the developer selects the "Export" option from the IBM VisualAge for Java IDE. Instead of exporting the Java bytecode files, they export a compiled object, providing compile-time flags as appropriate. The Smartguide will transfer the Java bytecode file(s) seamlessly to a development system running the same operating system as the production environment, and invoke the IBM High Performance Compiler™ for Java for that platform.

Object and Class Internal Structure

An object reference acts as a pointer to some data in the heap. The data consists of the following:

- A pointer to a static data structure containing the metadata for the class of that object (one per class)
- Eight bytes to implement Java's per-object locks
- Instance data of the object

The class metadata contains the following types of information:

- Class initialization status
- Method table
- Field table
- Inner classes table
- Interface table
- Access flags for the class, methods, and fields

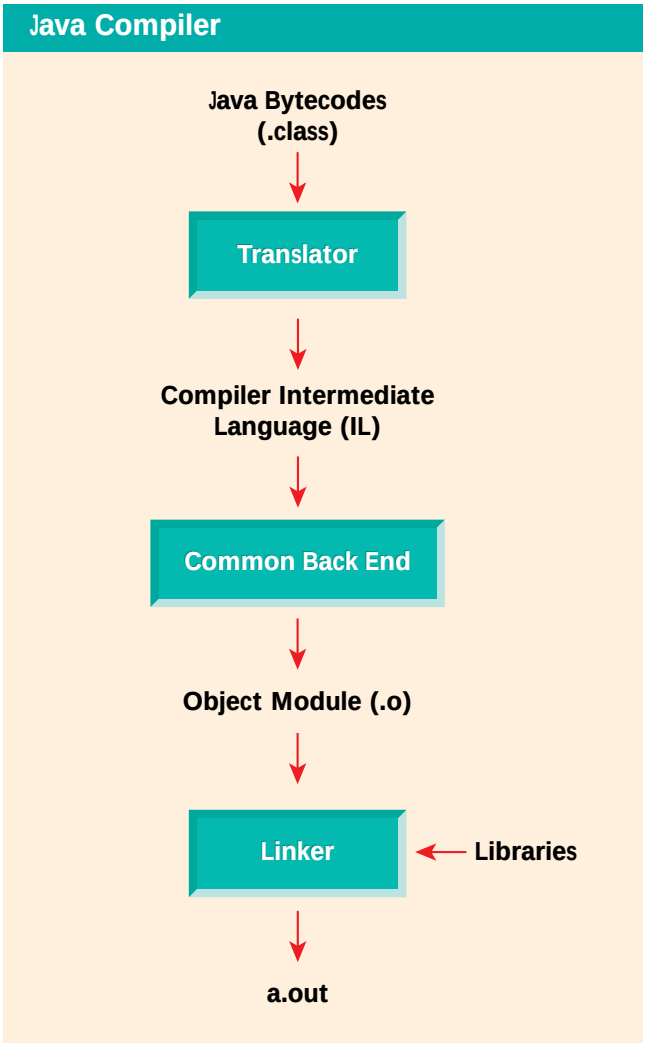

Figure 1. Basic structure of Java compiler

Much of the information contained in the class metadata supports the rich set of runtime facilities in Java, including interface method invocation, cast checking at runtime, reflection APIs, and the Java Native Interface (JNI).

All types of classes—plain, interface, and array—are represented using the same metadata format. Java requires a unique instance of `java.lang.Class` for each class in the application. The class metadata is the data for the unique instance of `java.lang.Class` for that class (thus the first twelve bytes of the class metadata for a class must correspond to the first twelve bytes of the data for an object—four bytes for the class metadata pointer and eight bytes for the lock information).

Garbage Collection

Use of the common back end is an important design decision that merits some discussion. By using the same back end that is used in IBM's C/C++, FORTRAN, COBOL, and other RS/6000 compilers, we can obtain the same high-quality, robust code optimization found in these other products. It also dictates that we use a conservative garbage collector instead of a nonconservative copying collector, since the common back end provides no special support for garbage collection (such as being able to distinguish pointers from nonpointers at runtime).

While this may appear to be a disadvantage, it is important to note that code generation for systems that support copying collection imposes runtime overhead that slows down user code. This overhead can be considerable. With some garbage collection, schemes such as generation scavenging, every pointer stored in the heap requires compare-and-branch code to determine if this location needs to be specially marked for garbage collection.

This overhead slows the execution of code that is not part of a garbage collection—simply to support garbage collection. Since time spent collecting garbage is typically far less than time spent running user code, our scheme optimizes generated user code at the expense of more costly garbage collection. We use the publicly available Boehm garbage collector, a conservative collector that has been ported to many platforms.

Code Generation Optimizations

Using the common back end from IBM's XL compilers immediately gives us a rich set of language-independent optimizations, such as instruction scheduling, common subexpression elimination, intramodule inlining, constant propagation, and global register allocation. We also perform some Java-specific optimizations in the translator.

Runtime checking code (null pointer, array bounds, and zero-divide checking) accounts for a significant amount of overhead in Java. The translator simulates the bytecode stack while compiling basic blocks to determine whether such checks can be eliminated. For example, in this way, most calls to constructors that immediately follow object creation can be performed without determining that the object reference passed to the constructor is not null.

The translator also emits direct calls to instance methods that are known to be final, or to members of a final class. The usual indirect call through the instance method table is unnecessary because final classes cannot be subclassed and final methods cannot be overridden. Therefore, at compile time the translator will know which method will be invoked.

Performance

Figure 2 shows the performance of our compiler on a set of publicly available language processing applications, including the following:

- Javac: Sun's Java 1.0.2 source-to-bytecode compiler
- Jacorb: Jacorb Version 0.5, a CORBA/IDL-to-Java translator, by Gerald Bose, Berlin University
- Toba: A Java-to-C translator, University of Arizona
- JavaLex: A lexical analyzer generator for Java, Princeton University
- JavaParser: A parser for Java, Sun Microsystems®, Inc. Generated automatically by Sun's Jack tool
- JavaCup: A LALR parser generator for Java, by Scott Hudson, Georgia Institute of Technology
- Jobe: A Java obfuscator, by Eron Jokipii

The benchmarks were run on an RS/6000 with a 133 MHz PowerPC™ 604 CPU, 96 MB of memory, and a 512 KB L2 cache.

Each benchmark was run using four different compilers/virtual machines:

- AIX Java Virtual Machine 1.0.2D (without JIT compiler)
- AIX Java Virtual Machine 1.0.2D with IBM JIT compiler 1.0+
- Our compiler with full runtime checking

- Our compiler with runtime checking turned off

Compiling with runtime checking turned off is unsafe since the generated code is not Java-compliant. However, we present the results below to show the overhead introduced by runtime checking.

Our compiler was used with the optimization flag (`-O`) turned on. As shown in Figure 2, compiling to native code provides a speedup between 4 and 17 times over interpreted code, and between 3 and 13 times over the JIT compiler. This family of applications tends to exercise Java's object-oriented features, such as instance method calls, object creation, and garbage collection. Note that in this set of benchmarks, turning off runtime checking does not provide a significant benefit. However, it makes a considerable difference for codes that spend a significant amount of time in tight loops manipulating arrays, such as scientific codes.

The javac benchmark measured above, using Sun's Java 1.0.2-level source-to-bytecode compiler, is provided in compiled form in the samples subdirectory on our Web site (www.alphaworks.ibm.com).

Status

One possible usage scenario for the compiler is on a performance-sensitive intranet server-side Java application where the code is ready to roll into production. Once the application is written and debugged, the compiler can be used on the server to create a high-performance static executable.

Another scenario is for an Internet Web server to automatically invoke the compiler to accelerate existing Java servlets. The first time a servlet is called, the Web server calls the compiler to create a static executable. With subsequent invocations, the Web server calls the previously created static executable. The portability of the servlet is retained at the Java bytecode level.

An early beta-level technology demonstration version of our compiler for AIX is available on IBM's alphaWorks Web site (www.alphaworks.ibm.com). At publication time for this article, it is a Java 1.0.2-level implementation with some missing features. The missing features include the `java.awt` and `java.applet` APIs, dynamic class loading, and Java-style Release-to-Release Binary Compatibility.

Compiler Performance

Benchmark	Interpreted	JIT	Compiled	Compiled (no check)
Javac	40.2s	21.1s	3.9s	3.7s
Jacorb	47.2s	34.3s	7.0s	6.8s
Toba	67.2s	51.7s	5.7s	5.5s
JavaLex	49.4s	38.9s	2.9s	2.8s
JavaParser	60.6s	20.1s	6.1s	4.8s
JavaCup	15.7s	11.9s	1.5s	1.5s
Jobe	18.1s	13.6s	4.6s	4.2s

Figure 2. Java benchmarks

References

Boehm, H. J. "Space Efficient Conservative Garbage Collection." *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

Acknowledgments

The author would like to acknowledge the contributions of the following people to the first release of the High Performance Compiler for Java: Dick Attanasio, Tony Cocchi, Pat Gallop, Mark Mergen, Mauricio Serrano, Janice Shepherd, Steve Smith, Simon Tooke, and Dean Williams.

Ven Seshadri, IBM Canada Ltd, Software Solutions Division, 1150 Eglinton Ave. E., North York, Ontario, CANADA, M3C 1H7. E-mail: seshadri@vnet.ibm.com. Mr. Seshadri is an advisory software developer at IBM's Toronto Lab, and has been working on the design and implementation of IBM's High Performance Compiler for Java since 1996. Prior to that, he was responsible for implementing loop transformations in IBM's XL FORTRAN and XL High Performance FORTRAN compilers. Mr. Seshadri has a BASc in Engineering Science and an MASc in Electrical Engineering, both from the University of Toronto.