A JAVA PROGRAMMING BASED ON HIGH PERFORMANCE

Shachindra Kumar Dubey
Faculty of Computer Science and Engineering Department
Jabalpur Engineering College, Jabalpur (M.P), India

Abstract — The standard technique for Java execution is interpretation, which provides for extensive portability of programs. A Java interpreter dynamically executes Java bytecodes, which comprise the instruction set of the Java Virtual Machine (JVM). Engineering a program in the context of performance aspect exhibits a crucial part, as it minimizes the time and space complexity. If the space complexity in a program is minimized, the application runs on minimum memory. Many JAVA developers write JAVA programs without consideration towards optimized execution. This paper elucidates miscellaneous techniques to escalate JAVA application program performance and can serve as an optimization tool for the JAVA application programmer i.e. Time and Space complexities are enhanced.

Keywords — Java based application, Java developers, Java optimization, Space complexity, Time complexity.

I. INTRODUCTION

The performance of a Java bytecode compiled Java program depends on how optimally its given tasks are managed by the host Java virtual machine (JVM), and how well the JVM exploits the features of the computer hardware and operating system (OS) in doing so. Thus, any Java performance test or comparison has to always report the version, vendor, OS and hardware architecture of the used JVM. In a similar manner, the performance of the equivalent natively compiled program will depend on the quality of its generated machine code, so the test or comparison also has to report the name, version and vendor of the used compiler, and its activated compiler optimization directives. In software development, the programming language Java was historically considered slower than the fastest 3rd generation strongly typed languages such as C and C++.

The Java programming language that evolved out of a research project started by Sun Microsystems in 1990 [Arnold and Gosling 1996; Gosling et al. 1996] is one of the most exciting technical developments in recent years. Java combines several features found in different programming paradigms into one language. Features such as platform independence for portability, an object-
orientation model, support for multithreading, support for distributed programming, and automatic
garbage collection, make Java very appealing to program developers.

To support portability, Java source code is translated into architecture neutral bytecodes that can be
executed on any platform that supports an implementation of the Java Virtual Machine (JVM). Most
JVM implementations execute Java bytecodes through either interpretation or Just-In-Time (JIT)
compilation. Since both interpretation and JIT compilation require runtime translation of bytecodes,
they both result in relatively slow execution times for an application program.

While advances with JIT compilers are making progress towards improving Java performance,
existing Java execution techniques do not yet match the performance attained by conventional
compiled languages. Of course, performance improves when Java is compiled directly to native
machine code, but at the expense of diminished portability.

II. JAVA EXECUTION
The Java source code files (.java files) are translated by a Java compiler into Java bytecodes, which
are then placed into .class files. The bytecodes define the instruction set for the JVM which actually
executes the user’s application program.

III. JAVA VIRTUAL MACHINE
The JVM is said to be virtual since, in general, it is implemented in software on an existing hardware
platform. The JVM must be implemented on the target platform before any compiled Java programs
can be executed on that platform. The ability to implement the JVM on various platforms is what
makes Java portable.

The instructions operate on byte, short, integer, long, float, double, char, object, and return address
data types. The JVM’s instruction set defines 200 standard opcodes, 25 quick variations of some
opcodes (to support efficient dynamic binding), and three reserved opcodes. The opcodes dictate to
the JVM what action to perform. Operands provide additional information, if needed, for the JVM to
execute the action.

The JVM can be divided into the five basic components shown in given Figure. Each of the
registers, stack, garbage-collected heap, methods area, and execution engine components must be
implemented in some form in every JVM. The registers component includes a program counter and
three other registers used to manage the stack. Since most of the bytecode instructions operate on the
stack, only a few registers are needed. The bytecodes are stored in the methods area. The program
counter points to the next byte in the methods area to be executed by the JVM. Parameters for
bytecode instructions, as well as results from the execution of bytecode instructions, are stored in the
stack. The stack passes parameters and return values to and from the methods. It is also used to
maintain the state of each method invocation, which is referred to as the stack frame. The optop,
frame, and vars registers manage the stack frame.

IV. GARBAGE COLLECTION
The garbage collection process imposes a time penalty on the user program. Consequently, it is
important that the garbage collector is efficient and interferes with program execution as little as
possible. From the implementor’s point of view, the programming effort required to implement the
garbage collector is another consideration. However, easy-to-implement techniques may not be the
most execution time efficient. For example, conservative garbage collectors treat every register and
word of allocated memory as a potential pointer and thus do not require any additional type
information for allocated memory blocks to be maintained. The drawback, however, is slower
execution time. Thus, there are trade-offs between ease of implementation and execution-time performance to be made when selecting a garbage collection technique for the JVM implementation.

V. JAVA COMPILERS

Another technique to execute Java programs is with a compiler that translates the Java bytecodes into native machine code. Like traditional high-level language compilers, a direct Java compiler starts with an application’s Java source code (or, alternatively, with its bytecode) and translates it directly into the machine language of the target processor. The JIT compilers, on the other hand, are dynamically invoked compilers that compile the Java bytecode during runtime. These compilers can apply different optimizations to speed up the execution of the generated native code.

Due to the unique features of the JVM, such as the stack architecture, dynamic loading, and exception handling, Java native code compilers need to apply different types of optimizations than those used in traditional compilers. Some commonly implemented optimizations include mapping stack variables to machine registers, moving constant stack variables into native instructions, increasing the efficiency of exception handling, and inlining of methods.

VI. BYTECODE OPTIMIZATION

One technique for improving the execution time performance of Java programs is to optimize the bytecodes. Converting Java bytecodes to JavaIR is very expensive, however, requiring an order of magnitude more time than the time required to perform the optimizations themselves. In an on-line version of Briki [Cierniak and Li 1997b], the same optimizations are performed while recovering only as much of the structure information as needed and using faster analysis techniques than those used in traditional compilers. This on-line version of Briki is integrated with the Kaffe JIT compiler [Wilkinson, Kaffe v0.10.0], using Kaffe to generate an IR. The compiler analyzes and transforms the Kaffe IR into an optimized IR which the Kaffe JIT backend then translates into native code.

The High Performance Java project [Bik and Gannon 1997; Bik and Gannon 1998] exploits implicit parallelism in loops and multiway recursive methods to generate parallel code using the standard Java multithreading mechanism. The JAVAR [Bik and Gannon 1997] tool, which is a source-to-source restructuring compiler, relies on explicit annotations in a sequential Java program to transform a sequential Java source code into a corresponding parallel code. The transformed program can be compiled into bytecodes using any standard Java compiler.

VII. IMPROVED JVM FEATURES

The underlying implementation of various JVM features, such as thread synchronization, RMI support, and garbage collection, often has a significant effect on the execution time performance of Java application programs. While these features make Java more powerful as a programming language, they tend to add delays to a program’s execution time. Hence, it is important to efficiently implement these features to improve Java performance. To support numerical applications, Java should include features that allow efficient execution of floating point and complex numbers.

Thread synchronization is a potential performance problem in many Java programs that use multithreading. Since Java libraries are implemented in a thread-safe manner, the performance of even single threaded applications may be degraded due to synchronization. In Java, synchronization is provided through monitors, which are language-level constructs used to guarantee mutually-exclusive access to shared data-structures.

The Java Remote Method Invocation (RMI) [Remote Method Invocation Specification] mechanism enables distributed programming by allowing methods of remote Java objects to be invoked from other JVMs, possibly on different physical hosts. A Java program can invoke methods on a remote
object once it obtains a reference to the remote object. This remote object reference is obtained either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. RMI uses object serialization to marshal and unmarshal parameters.

The current design and implementation of Java does not support high-performance numerical applications. To make the performance of Java numerical programs comparable to the performance obtained through programming languages such as C or Fortran, the numeric features of the Java programming language must be improved.

To directly address these limitations in Java, IBM has developed a special library that improves the performance of numerically-intensive Java applications. This library takes the form of the Java Array package and is implemented in the IBM HPCJ. The Array package supports such FORTRAN 90 like features as complex numbers, multidimensional arrays, and linear algebra library. A number of Java numeric applications that used this library were shown to achieve between 55% and 90% of the performance of corresponding highly optimized FORTRAN codes.

Most JVM implementations use conservative garbage collectors that are very easy to implement, but demonstrate rather poor performance. Conservative garbage collectors cannot always determine where all object references are located. As a result, they must be careful in marking objects as candidates for garbage collection to ensure that no objects that are potentially in use are freed prematurely. This inaccuracy sometimes leads to memory fragmentation due to the inability to relocate objects.

VIII. CONCLUSIONS

Java, as a programming language, offers enormous potential by providing platform independence and by combining a wide variety of language features found in different programming paradigms, such as, an object-orientation model, multithreading, automatic garbage collection, and so forth. The same features that make Java so attractive, however, come at the expense of very slow performance. The main reason behind Java’s slow performance is the high degree of hardware abstraction it offers. To make Java programs portable across all hardware platforms, Java source code is compiled to generate platform-independent bytecodes. These bytecodes are generated with no knowledge of the native CPU on which the code will be executed, however. Therefore, some translation or interpretation must occur at run-time, which directly adds to the application program’s execution time.

The choice of a particular Java execution technique must be guided by the requirements of the application program as well as the performance offered by the technique. However, as was pointed out earlier, the performance evaluation of these various execution techniques is incomplete due to the lack of a standardized set of Java benchmark programs. While Java has tremendous potential as a programming language, there is a tremendous amount yet to be done to make Java execution-time performance comparable to more traditional approaches.

REFERENCES