COMPILING JAVA JUST IN TIME

Timothy Cramer

Richard Friedman

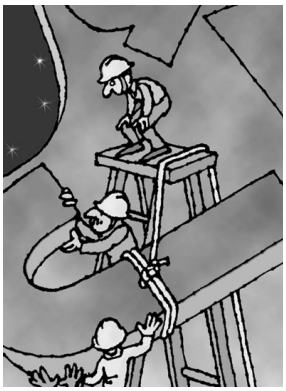
Terrence Miller

David Seberger

Robert Wilson

Mario Wolczko

Sun Microsystems, Inc.



To improve Java program performance, recent implementations use JIT compilation techniques rather than interpretation.

The Java programming language promises portable, secure execution of applications. Early Java implementations relied on interpretation, leading to poor performance compared to compiled programs. Compiling Java programs to the native machine instructions provides much higher performance. Because traditional compilation would defeat Java's portability and security, another approach is necessary.

This article describes some of the important issues related to just-in-time, or JIT, compilation techniques for Java. We focus on the JIT compilers developed by Sun for use with the JDK (Java Development Kit) virtual machine running on SPARC and Intel processors. (Access the Web at www.sun.com/workshop/java/jit for these compilers and additional information.) We also discuss performance improvements and limitations of JIT compilers. Future Java implementations may provide even better performance, and we outline some specific techniques that they may use.

Java execution model

The Java Virtual Machine (JVM) assures Java application portability and security. The JVM provides a well-defined runtime framework in which Java programs are compiled for a hypothetical instruction set architecture.[1] Programs are distributed in this abstract form, divorced from the details of any other computer architecture. Running a Java program involves either interpreting JVM instructions, compiling them into instructions of the underlying hardware, or directly executing them in a hardware implementation of the JVM.

The JVM is a stack machine. Each instruction gets its operands from the top of a stack, consuming those values and optionally replacing them with a result. The instructions themselves are encoded in a compact form of variable length, with the shortest instructions occupying 1 byte and most instructions being 1 to 3 bytes long. This form of encoding is known as bytecode. Previous systems such as the UCSD Pascal System and most Smalltalk implementations have used similar bytecodes.

A Java source-to-bytecode compiler, such as the javac program of the JDK, compiles the classes that constitute a Java program. The compiler translates methods in each source class into bytecode instructions and places all the bytecodes for a class together in a class file.

To run a Java program, the JVM loads the class file containing the program's entry point, and execution begins. The program may reference other class files, which are loaded in turn (possibly across a network). Hence, the final association of class files to form a running program takes place as execution proceeds. To maintain the integrity of the Java execution model, the JVM checks that a variety of semantic constraints are met, both within a class file and between class files. For example, bytecode from one class cannot access a field defined in another class unless explicitly permitted by the access specification in the field definition. As another example, when an integer is pushed onto the stack, all bytecodes that refer to that value must treat it as an integer, and not, say, as an object reference.

As part of a program's execution, the JVM must provide various services. It must manage memory, allowing programs to create objects and reclaiming objects once they are no longer required (a process known as garbage collection).[2] Java also allows interoperation with machine code created from other source languages (such as C), that is encapsulated to appear to a Java program as Java methods. Therefore, the JVM must also mediate between Java methods and these

0272-1732/97/$10.00 © 1997 IEEE

native methods, converting data representations and managing the flow of control into and out of native methods.

Why compile just in time?

Interpreting bytecodes is slow. In software, the JVM interpreter must fetch, decode, and then execute each bytecode in turn. Consider, for example, how the interpreter evaluates the expression $x = y + (2 * z)$, where x, y, and z are local variables containing integers. Figure 1 shows the Java bytecodes corresponding to this expression. The first three bytecodes push y, the integer constant 2, and z onto the operand stack. The imul bytecode multiplies the two values at the top of the stack and replaces them with the result. At that point, the stack contains y and $(2*z)$. The iadd bytecode then adds those values and stores the result on the stack. Finally, the istore instruction moves the value on the stack into local variable x.

Evaluating this simple expression involves not only performing the operations specified by the expression but also decoding six bytecodes, pushing five values onto the operand stack, and then popping them off again. It is no surprise that Java programs are slow when executed in this way.

One solution is to build a hardware implementation of the JVM. This certainly removes the overhead of decoding instructions in software. It does not, however, address the problem of portable execution on existing processors.

To improve performance on an existing processor, one can compile the bytecodes into efficient instruction sequences for the underlying machine. Performing this compilation prior to runtime and loading the program as native code would eliminate the portability and security of Java programs. On the other hand, compiling at runtime preserves these important properties. The program can still be distributed in platform-independent class files, and the bytecodes can still be verified prior to compilation. As long as the compiler includes the essential runtime security tests, the compiled program is as secure as if it were interpreted.

One potential drawback of this approach is that time spent compiling a method is time that could have been spent interpreting its bytecodes. For an overall gain in speed, we need to amortize the cost of compilation over the subsequent executions of the method in its faster, machine code form. Therefore, compilation speed is crucial. This is a very different situation from that facing a traditional, static compiler. Typically, having a fast compiler is a convenience to the programmer, but a compiler's speed is immaterial to the user. Programmers will wait for slow, optimizing compilers if their users benefit from faster execution. In contrast, the user pays the price of compilation when it occurs at runtime.

To minimize overhead, we can avoid compiling a method until it is certain that it will be executed. (Not all methods in a class file will execute; typically less than half execute in any run of a program.) The obvious strategy is to compile methods only when they are first executed, in other words, just in time. Deutsch and Schiffman pioneered this form of JIT compilation in a highly successful implementation of Smalltalk.[3] A JIT compiler must be fast enough to recoup the time spent in compilation in as few executions of the code as possible. Compilation speed is also important for minimizing start-up delays. If the compiler is slow, compilation will dominate the starting phase of the program, and the user must wait.

```
iload y
iconst 2
iload z
imul
iadd
istore x
```

Figure 1. Bytecodes for $x = y + (2 * z)$.

Compiling Java to native code

Compared to traditional static compilers, JIT compilers perform the same basic task of generating native code, but at a much faster rate. Static compilers can afford to spend much more time performing optimizations. The challenge for JIT compilers is to find ways to generate efficient code without incurring the expense of traditional optimization techniques. Performance is not the only issue: The compiled code must correctly implement the behavior required by the JVM specification.[1] To follow that specification exactly requires special consideration in several aspects of JIT compilation.

Minimizing compilation overhead. Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process. When compiling from Java bytecodes, however, we can eliminate that overhead. The bytecodes themselves are an IR. Because they are primarily designed to be compact and to facilitate interpretation, they are not the ideal IR for compilation, but they can easily be used for that purpose.

A compiler IR should preserve all the source-level information relevant for optimization and native code generation, while hiding irrelevant syntactic details. Java bytecodes combined with other information from Java class files satisfy these criteria. Java class files retain almost all the information from Java source files, as evidenced by the quality of the output from class file decompilers and by the emergence of tools to obscure the class file information. Only a few kinds of information are lost in the translation to bytecodes. For example, the JVM does not contain a Boolean type, so Boolean variables cannot be distinguished from integers.

The bytecodes also do not guarantee preservation of the structured control flow of Java programs. Relying on structured control flow can often simplify the implementation of many compiler optimizations. The possibility of unstructured control flow in bytecodes may complicate the task of a JIT compiler, but that is a relatively minor problem. For the most part, Java bytecodes provide exactly the information necessary for effective compilation.

The stack-oriented nature of bytecodes is well suited for interpretation but not for efficient execution on register-based processors. Explicit use of a stack would introduce runtime overheads and would fail to take advantage of the processor registers. So, rather than treating bytecodes as literal descriptions of the code to be executed, a JIT compiler can use them as implicit encodings of expressions.

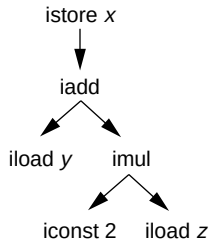


Figure 2. DAG representation of $x = y + (2 * z)$

Traditional compilers often represent expressions as directed acyclic graphs (DAGs).[4] Each vertex in a DAG specifies a single operation with the operand values specified by the outgoing edges. For example, Figure 2 shows the DAG for $x = y + (2 * z)$. (This example uses every value exactly once, so the DAG is actually a tree. The Java bytecode called dup allows intermediate results to have multiple uses so that the expressions correspond to DAGs instead of trees.) Each DAG edge corresponds to a particular intermediate result of a computation, which can be stored in a register, a temporary variable, or, in the case of Java bytecodes, on an operand stack.

The bytecodes for an expression encode the same information represented in a DAG. For example, the bytecodes shown earlier in Figure 1 evaluate the same expression shown as a DAG in Figure 2. A JIT compiler could easily convert the bytecodes to DAGs, but it is more efficient to generate code directly from the bytecodes.

Generating efficient code. There are three major issues associated with transforming bytecodes directly into efficient machine instructions:

- The use of the operand stack constrains the order of evaluation specified by the bytecodes. More efficient orderings may be possible with a register-based processor.
- The best choice of machine instructions for a particular bytecode may depend on how the result of that bytecode is used.
- The bytecodes may include redundant operations.

There may be more than one possible order of evaluation for a given DAG, and the bytecode ordering may not be the optimal choice. Other orderings may require less storage to hold intermediate results. For example, reordering the bytecodes in Figure 1 to load variable *y* after performing the multiplication reduces the maximum stack size from three to two. In native code, that may increase performance by allowing more values to be stored in processor registers. Changing the evaluation order requires some care, though. The Java language specifies a particular evaluation order for expressions so that a compiler can only reorder operations in ways that conform to the specification.

Some bytecodes are best translated to machine instructions in combination with the bytecodes that use their results. For example, the iconst bytecode pushes an integer constant onto the operand stack. Depending on how that constant value is used, it may be possible to specify the value as an immediate operand of a machine instruction. This is more efficient than storing the constant into a register. The same idea applies to several other bytecodes.

Bytecodes often include redundant computations within a single basic block. For example, bytecodes may load the same value from the constant pool several times. Also, a local variable may be pushed onto the operand stack more than once, and so on. Eliminating redundancies is an important optimization. Finding redundancies within a basic block is quite inexpensive and often provides a noticeable improvement in performance. However, detecting redundant computations across basic blocks can be more expensive, and it is not clear that a JIT compiler should attempt it.

Some redundancies are due to array bounds checking. Java requires that all array accesses be checked to ensure that the indices are within the bounds of the array. Performing those checks at runtime can significantly degrade performance of programs that use many array accesses. If the same array element is accessed more than once in a basic block, only the first access requires bounds checking. We remove the redundant bounds checks for the other accesses of that element. Future compilers may perform more advanced analyses to eliminate unnecessary bounds checks.

The code generation technique used in Sun's JIT compilers addresses each of these three issues. The basic idea is simple. The compiler steps through the bytecodes in a basic block one at a time. For bytecode operations whose order of evaluation is constrained by the Java specification, the compiler emits native code. For other bytecodes, however, it merely records the information necessary to generate native code. It delays producing the code until it reaches the bytecode using the result. This delayed emission of code allows the compiler to reorder computations to reduce the amount of temporary storage required. It also makes it easy for the code generator to take into consideration how the result values are used. To remove redundant operations, the compiler keeps track of which operations have already been performed at each point in a basic block.

As the compiler steps through the bytecodes, it essentially simulates their execution, keeping information about the values on the operand stack at every point. The compiler uses its own stack, which mirrors the runtime status of the operand stack. Instead of holding runtime values, the compile-time stack records information about the intermediate results of the computation. If the code to produce a value has not yet been emitted, the entry on the compile-time stack records the operations that need to be performed. Otherwise, it specifies the register or memory location that holds the value.

As an example, consider the process of generating SPARC instructions for the bytecodes from Figure 1. As the compiler steps through the first three bytecodes, it pushes three entries onto the compile-time stack. Figure 3 shows the contents of the stack at that point, assuming that local variables *y* and *z* have been allocated to registers L2 and L3. When the compiler encounters the multiplication bytecode, it emits code to perform that operation. The top two entries on the compile-time stack specify the operands. The constant integer 2 can be included as an immediate operand. A register, L0 in this case, must be selected to hold the intermediate result. The compil-

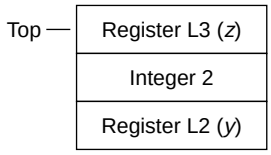


Figure 3. Compile-time stack before the multiplication.

Figure 4. Compile-time stack after the multiplication.

er than emits the instruction `smul L3, 2 → L0` results and updates the compile-time stack, as shown in Figure 4.

The compiler processes the addition in a similar manner. If local variable *x* was allocated to a register (for example, L1), the result of the addition can be placed directly in that register. The result in that case is the instruction `add L2, L0 → L1`. This example demonstrates how a JIT compiler removes interpretation overhead, reducing the entire expression to as few as two SPARC instructions.

So far, we have focused on the compilation process within a single basic block; handling block boundaries requires some additional effort. Besides holding the intermediate values in an expression evaluation, the operand stack may also hold values across basic block boundaries. To avoid generating code with explicit stack operations, a JIT compiler must determine the contents of the operand stack at the entry to every basic block. Java bytecodes are constrained in such a way that this is fairly easy to do. Because of security concerns, the operand stack must always have the same configuration every time a particular bytecode executes. That is, the stack must have the same height and must contain the same kinds of values. The JVM verifier checks that this constraint is always satisfied. One quick pass over the bytecodes in a method is sufficient for the JIT compiler to find the stack configuration at the start of each basic block. The values on the stack can then be allocated to registers or memory.

Correctness. Every JVM implementation must provide the correct behavior required by the JVM specification. The presence of a JIT compiler does not alter that requirement. In many ways, a Java compiler is no different from any other compiler in this regard. However, to ensure the portability of Java programs across different JVM implementations, the JVM specification includes many details that other language specifications may omit. We mention only a few examples here.

- When a thread runs out of stack space, the JVM must throw a special Java exception to report the problem.
- The user-specified initialization code for a class must execute when a class is first referenced. This adds significant complexity to a JIT compiler. (See the box on constant pool resolution.)

Constant pool resolution

A class file contains more than just bytecodes. One of the more important additional structures is the constant pool. In addition to holding numeric values that are too big to fit into bytecodes, the constant pool holds references to fields and methods in other classes.

A constant pool reference to another class initially just identifies the class by name. When the interpreter first uses that constant pool entry, it resolves the entry to directly reference an internal data structure in the JVM that describes the other class. If that class has not been previously used, resolving the constant pool entry causes it to be loaded and initialized.

The constant pool resolution mechanism serves an important role in the execution of a Java program. A JVM implementation is not allowed to execute a class's initialization code until that class is actually used. The interpreter resolves entries in the constant pool the first time it evaluates a bytecode referencing that entry. This ensures that classes are initialized at the proper times. After a particular bytecode has been evaluated once, there is no need for subsequent evaluations to resolve the constant pool entry. To avoid the overhead of doing so, the interpreter replaces the original bytecode with a "quick" version of the same operation. Later, when evaluating a quick version of a bytecode, the interpreter can safely assume that the constant pool entry has already been resolved.

Compiled Java code must address this same issue. When the compiled code first uses a particular constant pool entry referring to another class, it must call a routine in the JVM to resolve the entry. Resolving the entry at compilation time is not legal: It must happen when the compiled code executes. Programs that rely on proper initialization behavior might not run correctly if the constant pool entries are resolved too early. The compiler can easily generate a call to resolve the constant pool entry, but we only want to execute that call once. Our solution is analogous to the interpreter's use of "quick" bytecodes. We use self-modifying code to remove the call after it first executes.

- Java requires that floating-point arithmetic use the IEEE-754 specification with 64-bit "double" values and 32-bit "float" data types. On Intel processors, which use 80-bit floating-point arithmetic, each intermediate floating-point result must be truncated properly.

Interactions with the JVM

Code compiled by a JIT compiler must interact with the JVM in a number of ways:

- Some bytecodes, such as those involving memory allocation or synchronization, are translated into calls to JVM routines.
- Method calls in compiled code may invoke the JVM interpreter, either directly by calling methods that for

some reason are not compiled, or indirectly by calling native methods that in turn invoke the interpreter.
- Entries in the constant pool must be resolved when they are first referenced. This may involve extensive work in the JVM to load and initialize new classes. (See the box.)
- In a number of situations, the JVM needs to examine the execution stack for Java methods. Exception handling and garbage collection are primary examples. The JVM must locate the stack frames for compiled methods and know the format of the data within each frame.

The main issue in these interactions is reducing the overhead of method invocations. The calling convention used by the interpreter is too inefficient for compiled code. A JIT compiler can use its own calling convention as long as it continues to support all the essential interactions with the JVM.

Reducing call overhead. Each thread in the JVM has two separate stacks. The thread stack holds the stack frames for native methods and the routines within the JVM itself. The Java stack contains the stack frames for interpreted methods. Java stacks consist of a number of noncontiguous segments. (For platforms without virtual memory, dividing the Java stacks into segments allows them to start small and expand as necessary.)

For efficiency, JIT-compiled code uses the thread stack rather than the Java stack. Because the Java stack is divided into noncontiguous segments, creating a new Java stack frame requires an extra check for sufficient space in the current stack segment. In contrast, adding a stack frame on the thread stack typically requires a single instruction to adjust the stack pointer, relying on the virtual memory system to detect stack overflows.

Besides avoiding the cost of handling noncontiguous segments in the Java stack, using the thread stack for compiled code avoids another significant source of overhead. The JVM expects each frame on the Java stack to contain a number of fields to facilitate operations such as exception handling and garbage collection. The runtime cost of initializing those fields when creating a new stack frame is not significant compared to the overhead of interpretation. For compiled code, however, it is relatively expensive. By using the thread stack, the compiled code can use its own stack frame layout with minimal initialization cost.

Compiled code also uses a different calling convention than interpreted methods. The interpreter relies on use of the Java stack to pass arguments. Outgoing arguments are pushed onto the operand stack. The new frame for the callee overlaps with the portion of the operand stack containing the arguments, so that the incoming arguments appear at the beginning of the callee's frame. When the callee returns, it stores the return value onto the caller's operand stack. Since the compiled code does not use the Java stack and since frames on the thread stack do not overlap, some other means of passing arguments is necessary. Moreover, for systems where the native calling convention supports passing arguments and returning values in registers, the compiled code should take advantage of this to improve the performance of method invocations. Our solution is to use the native calling convention for each platform.

To allow compiled and interpreted methods to coexist, we must translate between the different calling conventions at every transition between the interpreter and compiled code. Even without a JIT compiler, a similar situation arises when calling native methods. The solution for native methods is to insert stub routines between the interpreter and the native code. A native method stub reads the incoming arguments from the Java stack and places them in registers or on the thread stack according to the native calling convention. When the native method returns, the stub stores the return value back onto the Java stack. The stubs for compiled methods perform the same functions. An advantage of using the native calling convention for compiled methods is that transitions from compiled code to native methods can be much more efficient. Since they both use the same calling convention, only a very minimal stub routine is necessary.

Garbage collection. The JVM uses automatic memory management to relieve programmers of the burden of providing explicit deallocation. Whenever additional memory is required, the JVM releases storage by reclaiming objects that are no longer needed. The basic approach of most garbage collectors is to trace through all the objects that are reachable. The storage occupied by unreachable objects can then be reused.

The use of a conservative garbage collector in the current JDK virtual machine greatly simplifies support for garbage collection. The garbage collector scans the thread stacks and the Java stacks, searching for any values that could possibly be references to objects in the Java heap. It does not use information about the particular layout of stack frames on the thread stack, and therefore cannot know which locations contain object references and which contain raw data (such as integers). Whenever it sees a bit pattern that may represent a reference to an object, it must assume that it is a reference. Thus, the JIT compiler does not need to inform the garbage collector of the layout of the stack frames for compiled methods.

Exception handling. The situation is not so simple for exception handling. When an exception occurs, the JVM first searches for a handler in the current method. If one cannot be found, it pops the last frame off the stack and searches for a handler in the next method on the stack. It continues this stack "unwinding" until it finds a handler or there are no more frames on the stack. When interpreted and compiled methods are mixed together, there are two separate stacks to unwind: the Java stack and the thread stack. Both may contain frames for Java methods. The JVM must interleave the unwinding of these two stacks.

Beyond simply unwinding the stacks, the exception-handling code must be able to obtain certain information from the stack frames. For each frame, it needs to know the corresponding method and the current point in the execution of that method to determine if there is a relevant exception handler at that point. For interpreted methods, that information resides in the Java stack frames. For compiled code, the necessary information can be determined from the return addresses recorded in the thread stack. This avoids the overhead of recording the information in the stack frames, but it slows the exception handling. The relative

Table 1. JIT compiler microbenchmark results.

Benchmark	Speedup over interpretation
CM/Loop	42.5
CM/Graphics	1.0
UCSD/GC	1.0
UCSD/Exception	0.5

Table 2. Speedups for complete programs.

Benchmark	Speedup over interpretation
Richards	9.0
Tomcatv	7.3
Compress	6.8
RayTracer	2.2
DeltaBlue	2.1

infrequency of exceptions in most Java programs justifies this trade-off.

The compiled code for an exception handler needs to locate the current values of local variables. This is easy when each local variable resides in memory at a fixed position in the stack frame. It is not always so easy when variables are allocated to registers. If the compiled exception handler needs the value of a local variable in a particular register, the exception-handling code must restore the proper value of that register before transferring control to the handler.

The register windows on SPARC processors make it easy to find values stored in registers. Each compiled method essentially has its own set of registers that can be flushed to known locations in the stack frames when an exception occurs. For processors without register windows, the calling convention can have a significant effect on register allocation. Because of exception handling, we do not allocate variables to callee-saved registers across method invocation sites. The interface between compiled code and the JVM is written in C, so if an exception is thrown from within that C code, we cannot restore the callee-saved registers to the values they held in the compiled Java code. We have no way to determine which locations the C code used to save those values. Consequently, we cannot globally allocate many variables to registers for Intel and PowerPC systems, where many of the registers are callee-saved.

Performance

The performance improvements of JIT compilers vary widely across different programs. As mentioned earlier, translating to native code avoids interpretation overhead. Programs that would otherwise spend almost all their execution time in the JVM interpreter speed up dramatically. A JIT compiler, however, does not address the other components of a JVM implementation.

To illustrate this point, we have profiled the execution of the JDK virtual machine. When interpreting the javac program with a large compiler written in Java as input, only 68% of the execution time was spent interpreting bytecodes. No matter how fast the compiled code, at least one third of the interpreted execution time will remain. Therefore, a JIT compiler cannot speed up the program by more than a factor of three. The rest of the time is spent in synchronization (18%), allocating memory and collecting garbage (13%), and running native methods (1%). We expect future JVM implementations to improve the performance of these operations, but that is outside the scope of JIT compilation.

We have evaluated the performance of our JIT compilers on many benchmarks. Table 1 lists the results for some microbenchmarks that illustrate the performance of particular kinds of operations. We collected these numbers with the JDK 1.0.2 virtual machine on an UltraSPARC I system, running at 167 MHz under Solaris 2.5.1 (taking the best of three runs).

The first two microbenchmarks are components of the CaffeineMark benchmark.[5] The Loop test consists of nested loops that repetitively increment local variables. As evidenced by the large speedup, JIT compilers work very well for this code style, in which the interpretation overhead is the dominant component of the interpreted execution time. This is an extreme case. The Graphics test, which primarily executes code in native methods, shows the other extreme, where the JIT compiler has no visible effect.

The last two benchmarks in Table 1 illustrate other limitations of JIT compilers. These microbenchmarks are components of the UCSD benchmarks for Java.[6] The GC benchmark measures the performance of the JVM garbage collection. As expected, the JIT compiler has no effect. The Exception test measures the performance of exception handling. In this case, the JIT compiler actually makes the performance worse. This reflects our decision to optimize for the common case where exceptions are not thrown, making exception handling considerably more expensive.

The performance with the JIT compiler is somewhat more consistent for complete programs. Table 2 shows the speedups measured for several benchmarks using the system just described. These are typical of the speedups that we have observed for many programs.

The Richards benchmark is an operating system simulation program that has been translated to a variety of programming languages. This version, written in a C-like style, has few virtual calls, and the JIT compiler can speed it up by almost an order of magnitude. Tomcatv and Compress are Java versions of the corresponding SPEC benchmarks. Both of these programs spend most of their execution time in small loops with frequent array accesses. Our compiler removes a number of the redundant array bounds checks in Tomcatv. It is unable to do so for Compress because the array accesses are in separate methods, but the overall speedup is still almost as good as for Tomcatv. RayTracer renders an image using ray tracing. DeltaBlue is a constraint solver. These programs make extensive use of virtual calls, and the speedups are somewhat smaller than for the other benchmarks described here.

We measured the speed of the JIT compilers and found that they typically require about 700 machine cycles per byte of input for the SPARC compiler and about 1,300 per byte of

input for the Intel compiler (on a 200-MHz Pentium Pro).

Memory use is another factor to consider with JIT compilation. Not only does the JIT compiler use memory itself, it also uses memory to hold the compiled code. The SPARC and Intel compilers themselves require 176 Kbytes and 120 Kbytes. On average, each byte of bytecode translates into 5 bytes of SPARC machine code and 4 bytes of Intel machine code. The extra space required to hold the compiled code may limit the use of a JIT compiler on systems with little memory. Even if there is sufficient memory, the extra space may affect performance due to increased paging and cache effects.

Beyond JIT compilation

Current JIT compilers can substantially improve performance, and future JVM implementations will use more advanced techniques to realize further performance gains. Precise garbage collectors can reduce overheads and avoid the potential inaccuracy of conservative collectors. The JVM can also selectively compile and optimize the most frequently executed methods.

Precise garbage collection. Earlier, we mentioned that the current Sun JIT compilers rely on a conservative, or imprecise, garbage collector that sweeps the whole heap to determine if any objects may be reclaimed. This time-consuming sweep causes disruptive pauses in a program's execution. It would be preferable to use a less disruptive garbage collector that does not require scanning the whole heap at each reclamation. These collectors are precise, in that they need to know the exact set of memory locations that contain object references. (The adoption of the Java Native Interface in the JDK 1.1 makes this possible by regulating how native methods can access Java objects.)

A compiler must produce extra information for a precise garbage collector. When precise garbage collection takes place, the garbage collector must scan the stacks of all executing threads checking for object references. The compiler must emit so-called stack maps to provide information to the garbage collector to identify which locations in a stack frame contain these references.

It is prohibitively expensive to store a stack map for every instruction, but fortunately there is an alternative. Instead of allowing thread suspension at arbitrary points, we can insist that a thread be suspended before a collection only at compiler-specified safe points. Only at safe points do we need to keep stack maps. To ensure that a thread is at a safe point when the collector runs, we can modify the thread scheduler. We can also use break points to advance suspended threads to safe points prior to a collection (inserting the break points immediately before collection and removing them thereafter). Every call site must be a safe point because a collection may occur before returning from the callee. Additionally, the compiler must ensure that every loop contains a safe point to guarantee that garbage collection is not delayed indefinitely.

Adaptive optimization. Ideally, we would tailor the compilation of a method to the amount of time the program actually spends in that method. A dynamic compiler can observe the program as it runs and optimize the most frequenty executed models. A simple example is deciding when to compile based on observed execution frequencies. The bytecode interpreter can count how many times a method has been invoked or a loop iterated. When the count reaches a predetermined threshold, the interpreter can invoke the compiler.

Recompiling based on observed behavior is a form of adaptive optimization. This technique was pioneered in the Self 3.0 Virtual Machine as part of a research project at Sun Microsystems Laboratories and Stanford University.[7] In an adaptively optimizing system, initial executions of a method either are interpreted or use a simple compiler. The code is self-monitoring, using execution counters to detect hot spots.

When a method is found worthy of optimization, the system can spend more time on its compilation, hoping to amortize the optimization time in future executions of the method. Such a method is recompiled with a higher level of optimization. The infrastructure required for adaptive optimization is considerably more complex than that of an interpreted system or simple JIT compiler, but it can potentially provide much higher levels of performance.

Adaptive inlining. Inlining methods (replacing calls with the actual code of the called methods) is an important optimization in any program with a high call density and small methods. Without inlining, the call and return overhead often dominates the execution of such programs. Furthermore, method calls inhibit the compiler's ability to produce efficient code because many optimizations cannot be applied across method calls.

The use of virtual calls in Java defeats traditional inlining techniques, because many target methods may exist at a virtual call site. However, as the Self system[7] demonstrated, virtual calls need not trouble a dynamic compiler. For some call sites, the compiler can determine that the call has only one possible target. For example, the target method may be declared to be final, ensuring that there are no other candidates. Alternatively, the compiler may observe that there are no subclasses overriding the target method. In this case, the runtime system must take note of any subclasses that are loaded subsequent to compilation and undo inlining decisions if the target method is overridden in any subclass.

A dynamic compiler can inline even virtual calls with more than one potential target. Typically, most virtual call sites invoke the same method repeatedly. The runtime system can note call sites that possess this behavior. The compiler can emit specialized versions of the method that optimize the common case yet still retain the ability to deal with the other cases.

JIT COMPILERS CAN PROVIDE dramatic performance improvements for programs where the vast majority of execution time would otherwise be spent interpreting bytecodes. Because JIT compilers do not address the performance of other aspects of a virtual machine, programs that include extensive use of synchronization, memory allocation, and native methods may not run much faster. While a JIT compiler is essential for high-performance JVM implementations, it is not a complete solution.

Java performance will continue to improve in the future. Better garbage collection techniques and faster synchronization will decrease those components of the execution time that are not addressed by compilation. JIT compilers will evolve to

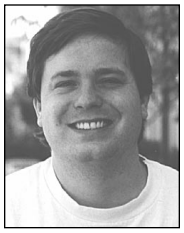
incorporate adaptive optimization and inlining. With these and other techniques, performance is unlikely to remain an obstacle to using Java in the vast majority of applications.

Acknowledgments

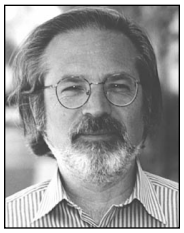
Others at Sun Microsystems contributing to this work include Boris Beylin, Dave Cox, Steve Dever, Bruce Kenner, Ross Knippel, and Dave Spott. We also acknowledge the earlier work by Dan Grove and Nand Mulchandani. Java and SPARC are trademarks of Sun Microsystems Inc.

References

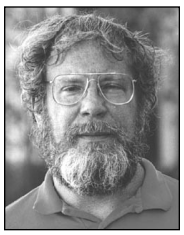
1. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Addison-Wesley, Reading, Mass., 1996.
2. R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management,* John Wiley & Sons, New York, 1996.
3. L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th ACM Symp. Principles of Programming Languages,* Assoc. Computing Machinery, New York, 1984, pp. 297-302.
4. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1988.
5. *CaffeineMark(TM) Version 2.01*, Pendragon Software, www.webfayre.com/pendragon/cm2/.
6. W.G. Griswold and P.S. Phillips, *UCSD Benchmarks for Java*, www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.
7. U. Hölzle and D. Ungar, "A Third-Generation Self Implementation: Reconciling Responsiveness with Performance," *Proc. ACM OOPSLA (Object-Oriented Programming Systems, Languages, and Applications) 94 Conf.,* ACM, 1994, pp. 229-243.



Timothy Cramer, a staff engineer at Sun, currently works on just-in-time compilers for Solaris/NT Intel and Solaris SPARC. He holds a BS degree in math and computer science from the University of Wisconsin, Eau Claire. He is a member of the ACM.

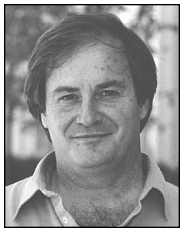


Richard Friedman is a senior technical writer at Sun Microsystems and is project lead for developer products documentation. Specializing in programming languages and supercomputing, he holds a BS in mathematics from the Polytechnic Institute of Brooklyn, and is a member of the ACM SIGDOC, IEEE Computer Society, and the Society for Technical Communication.



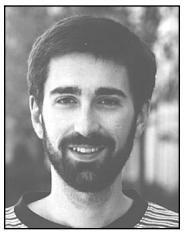
Terrence Miller is a senior staff engineer at Sun Microsystems and is project lead for JIT compiler development. His technical interests include processor architecture, compilation technology, and development environments.

Miller holds a PhD in computer science from Yale University and is a member of the IEEE and the ACM.



David Seberger is a manager at Sun Microsystems, where he currently manages the JIT compiler efforts. Technical interests include compiler optimizations for retargetable multilanguage systems.

Seberger holds an MS in computer science from the University of California at Davis.



Robert Wilson is an engineer at Sun Microsystems. He is currently working on high-performance Java virtual machines and on completing his PhD dissertation at Stanford University. He is a member of the ACM.

Mario Wolczko is a senior staff engineer at Sun Microsystems Laboratories. His technical interests include software and hardware implementations of object-oriented languages.

Wolczko holds a PhD in computer science from the University of Manchester and is a member of the IEEE and the ACM.

Direct questions concerning this article to David Seberger, Sun Microsystems, Inc., MS UMPK16-303, 2550 Garcia Avenue, Mountain View, CA 94043; seberger@enc.sun.com.