Optimizing Java

Theory and Practice

Zoran Budimlic
Ken Kennedy

Rice University Computer Science Department
6100 South Main, Houston TX 77005
zoran@cs.rice.edu, ken@cs.rice.edu

Abstract
The enormous popularity of the Internet has made an instant star of the Java programming language. Java's portability, reusability, security and clean design, has made it the language of choice for Web-based applications and a popular alternative to C++ for object-oriented programming. Unfortunately, the performance of the standard Java implementation, even with just-in-time compilation technology, is far behind the most popular languages today. The need for an aggressive optimizing compiler for Java is clear.

Building on preliminary experience with the JavaSoft bytecode optimizer, this paper explores some the issues that arise in building efficient implementations of Java. A number of interesting problems are presented by the Java language design, making classical optimization strategies much harder to implement. On the other hand, Java presents the opportunity for some new optimizations, unique for this language.

1. Introduction

In this paper we present the results of a summer project on optimizing Java done at JavaSoft Division of Sun Microsystems, along with preliminary results of continuing research at Rice University. Many interesting compilation problems emerge when trying to optimize Java, including:

- unavailability of the complete program at compilation time, eliminating opportunities for interprocedural optimization,
- the exception mechanism, which limits code movement optimizations, and
- the high abstraction level of the Java Virtual Machine instruction codes (bytecodes), which hides many machine dependent optimization opportunities from the compiler.

In this paper, we discuss those problems, propose some solutions, and suggest areas for further research.

This paper is organized into several sections. Section 2 describes possible approaches to optimizing Java. Section 3 presents the design of our current research compiler, including the optimizations implemented and advantages and limitations of the compilation model it employs. Section 4 describes an approach to Java optimization that relaxes the compilation model and presents some new optimizations with preliminary results. Section 5 describes an aggressive strategy for Java optimization, discussing its potential and applicability. Section 6 addresses the Java exception mechanism, which presents a problem for optimization regardless of the strategy employed. The final section draws conclusions from this preliminary work and present some directions for future research.

2. Optimization Strategies

There are three distinct strategies for optimizing Java compilation. each having advantages and disadvantages for particular applications and situations. In this paper, we will describe each of them and discuss their applicability to different problems for which Java might be used.

The first approach is to stay within the Java compilation model defined by Sun Microsystems—code is generated for the Java Virtual Machine (VM), the compiler processes one class at a time, and the generated code is completely independent of the platform on which it is executed. This means that the compiler cannot make any assumptions about the machine on which the VM code will be executed and the VM cannot make any assumptions about the compiler that is being used to generate VM bytecodes.

This model is used in a project begun at the JavaSoft that is discussed in Section 3 of the paper. This approach has an obvious advantage: no changes are made to the current

Java execution environment, so the portability, security, and functionality that are the hallmarks of the Java technology are preserved. Unfortunately, that compilation model, along with the combination of the Java features, leaves only limited room for performance improvement. However, for applications where absolute portability and security are crucial, the compiler must stay within the boundaries of this model. Given that requirement, it is nevertheless worthwhile to exploit every opportunity for performance improvement no matter how small.

An approach at the opposite end of the spectrum would be to sacrifice the portability and security of the Java Virtual Machine, concentrating instead on the construction of the highly sophisticated optimizing native code compiler. Although this would sacrifice the portability of the VM object code and the security required for execution over the Internet, the source code would remain portable and it could always be recompiled using the first compilation strategy for use over the Internet. This approach is suggested for the very high performance server applications where Fortran and C++ are currently being used. Some important aspects of this strategy are discussed in Section 5.

A third approach would combine features of the first two, compromising some portability and functionality for better performance. As we will show in Section 4, relaxing the constraints on the way the Java code is currently compiled and executed can impact the performance significantly. This model would still keep all of the convenience and power of execution over the Internet and the Java Virtual Machine security model would still be enforced. The compiler and the VM would know about each other, weakening the absolute portability model, so that the cross knowledge could be used to achieve higher performance. Under this scheme the compilation process would not be as simple as in the JavaSoft model. This model would be suitable for applications that require execution over the net, with all the security of the Java VM instruction set, but which are willing to be limited to the specific compilers and VMs that are supporting this model.

As already mentioned, each of this strategies has its advantages and disadvantages, and the right model should be chosen to meet the needs of the intended application.

3. Standard Model

In this section, we describe the first approach, which is used in our current compiler. During the design and implementation of this project, which was done while one author was a summer student intern at JavaSoft, a number of interesting problems in optimizing Java surfaced, revealing some limitations of the compilation model that has been used to date.

Our research compiler builds on the distributed Sun Java compiler by including a standalone, bytecode-to-bytecode optimizer. This structure was chosen for the effort for two reasons. First, when the effort was begun at JavaSoft, the optimizer was easier to develop and debug because it was independent of the current build of JavaSoft's **javac** compiler. Second, the structure also makes the optimizer independent of *any* compiler, so it can be used to optimize code produced by other Java compilers, as well as any other language compiled to the Java VM. Third, it can be used as a optimizing prepass to many just-in-time (JIT) run-time compilers, which are now coming into widespread use.

The compiler itself is organized into a series of phases that are loosely connected with one another:

1. parse the input class and load it into memory;
2. convert the stack machine bytecodes into traditional expression trees and construct the *control flow graph* (CFG);
3. convert the CFG into *static single assignment* (SSA) form.
4. perform some standard optimizations—dead code elimination and constant propagation—on the SSA.
5. restore the CFG from the SSA, renaming the local variables if necessary.
6. perform local optimizations on the CFG itself—local common subexpression elimination, copy propagation, and register allocation; and
7. generate the final program as Java VM bytecodes, performing peephole optimizations (stack allocation, replacement with the cheaper operator) along the way.

In the next several sections we will discuss some of these operations.

3.1. High Level Source Recovery

Java **bytecodes** are not a particularly suitable intermediate representation for optimization. Most optimizations in the literature apply to expression-like code, with variables and registers instead of a stack. The most common intermediate representations are sequences of three-address and two-address instructions resembling assembly language [Aho et al. 1986].

The Java VM is a stack machine, which makes data flow analysis complicated. The need to track the usage of the variables to and from the operator stack is the most common difficulty we encountered. This motivated us to develop a pass to translate Java VM bytecodes into expression trees, a representation more suitable for analysis.

The conversion procedure is straightforward. Basic blocks are identified by scanning the Java VM bytecodes for . branches, identifying branch targets in the process. When the structure of the CFG has been established in this way, we proceed with the conversion of the bytecodes in each basic block to the expression trees.

The conversion to expression trees is done by traversing the basic block simulating the behavior of the virtual machine using an *expression stack*. There are three cases:

- when a *constructing* bytecode—one that creates a constant or pushes a local variable on the stack—is encountered, the corresponding constant or variable is pushed onto an expression stack;
- When an arithmetic operation is simulated, the appropriate number of operands are popped from the expression stack, the resulting expression is constructed, and it is pushed back on the expression stack.
- A bytecode that stores the value on top of the stack in a local variable will be simulated by popping the top expression from the expression stack and constructing an assignment.

Figure 1 shows an example of the bytecodes and the resulting expression.

Bytecode	Stack	Code
iconst_1	1	
iload_1	R1 1	
iadd	(R1+1)	
istore_1		R1=R1+1

Figure 1

After construction of the expression trees for each basic block, the expression stacks have to be merged at the each basic block entry. The expression tree constructor assigns arbitrary "register" names to the local variables when constructing expressions. If two or more blocks have non-empty expression stacks at their exits, and they merge at the same point in the CFG, the expression stacks must be unified. This is done by unifying each of the expressions on the stack with the appropriate expressions on other stack(s). A complex expression that must be unified is converted into an assignment to a temporary variable and then the variable is unified with other expressions. The code that does the unification assignments is inserted at the beginning of the block that succeeds the merging point. A similar merging operation is done for the splitting points in the CFG, with code inserted (if necessary) at the end of the block before the split.

3.2. SSA Construction and Usage

The fastest known algorithms for many compiler optimizations use SSA form to represent the program being optimized. That was the primary reason we selected it as an intermediate representation in our compiler. SSA construction and reconstruction algorithms are described in detail elsewhere [Cytron et al. 1991][Briggs et al. 1995].

One issue in SSA construction and reconstruction for Java programs is yet to be resolved. Our current implementation converts only the method's local variables into SSA names, ignoring all instance variables. This is because some of the code moving optimizations (loop invariant code motion, for example) require the SSA reconstruction algorithm to rename variables when restoring the CFG. This is, of course, unacceptable for instance and global variables in Java, since that would change the class interface. We believe that the omission of instance variables from the SSA (and thus from optimizations on it) causes a significant decrease in the quality of code generated. We plan to use techniques developed by Briggs, Schillingburg and Simpson [Briggs et al. 1995], which employ tags for memory locations to deal with instance variables. All of the optimizations on SSA would be aware of the instance variable tags and would act accordingly so the class interface would not be changed after the reconstruction of the CFG.

3.3. Dead Code Elimination and Constant Propagation

Dead code elimination is a important and well-understood optimization in traditional languages [Kennedy 1981], [Briggs et al. 1993]. In applying this optimization to Java, it is important to understand the impact of impact of the Java exception mechanism. Since many Java instructions can potentially cause an exception, a prepass of the dead code elimination algorithm must mark all of those instructions as critical (e.g., undeletable), thus limiting opportunities for dead code elimination. Some approaches to dealing with exceptions are discussed later in this paper.

For constant propagation we use the well-known Wegman-Zadeck algorithm [Wegman Zadeck 1985].

3.4. Local Common Subexpression Elimination

In our implementation of local common subexpression elimination, we use a well known value numbering algorithm [Simpson 1996],[Briggs et al. 1996]. Value numbering provides a very natural framework for common subexpression elimination. All of the variables in the program are numbered, with two variables having the same number only if they have the same value. Two expressions have the same value (and get the same value number) only if they have identical structure and their operands have the same value number. After discovering the value numbers, it is trivial to discover the common subexpressions, store the value once computed for the given subexpression and replace all the subsequent computations with simple loads.

3.5. Register and Stack Allocation

The Java Virtual Machine is a stack machine, so it does not have the notion of registers in the usual sense. Furthermore, it does not have a notion of memory, just a potentially huge number of local variables to store temporary values and a stack for operations. However, there is a substantive difference in the speed when accessing the stack versus the local variables.

Four local variables (0-3) have a special codes for load and store operations assigned to them that are shorter than normal. This results in shorter code and faster execution when those variables are used. Also, the first 64 local variables can fit into the register cache on some Java microprocessors currently under development. It is reasonable to expect that most of the specialized Java microprocessors, as well as the run-time environments for conventional processors will cache some of the local variables in their registers. Although the speed differences are not dramatic enough, relative to access times in registers vs. cache vs. memory on conventional processors, to justify the implementation of a full graph-coloring allocation algorithm [Briggs et al. 1994], they are significant enough to prompt us to implement a simple, but fast and effective heuristic to exploit the described asymmetry. The register allocation algorithm is as follows:

- if the compiled method is large enough, the input values (originally stored in local variables starting from 0) are taken into account for register allocation and relocated, else they are left in their original location
- for each basic block, make a list of available registers
- traverse each block, assigning the lowest available register to the variable that has been assigned the value, and replacing the usage of the local variables with their assigned registers
- if the current instruction uses the variable for the last time, put the corresponding register on the free list
- after assigning registers for each basic block, registers are assigned to the global values. Every local variable that is live across the basic block is considered global and is assigned a unique register for the whole method. The registers are assigned sequentially to those variables, starting with the maximum register number allocated in the basic block register allocation

Although not nearly as aggressive and effective as graph coloring allocators, this simple strategy proved quite effective in practice, especially for short methods, which are very common for object oriented programming style.

As we already noted, the operator stack is usually the fastest memory in Java virtual machine. Using the stack as much as possible for storing the intermediate values would speed-up the code substantially. We use a very simple pass over the code to discover the possibilities for reusing the stack:

- if the value stored in some local variable is used exactly once and in the very next instruction, we eliminate the store and subsequent load, leaving the value on the stack. This is the most common case, because it results from breaking up large expressions into subexpressions and computing them separately.
- if the value stored in the local variable is used twice by the next two instructions, store and two later loads are replaced with the duplication of the value on the stack. This case is a common result of the unification algorithm described in section 3.1. and of the common subexpression elimination algorithm.

Again, this is a very simple method, and although there are more effective algorithms for code generation for stack machines that use the stack optimally [Bruno Lassagne 1975], we found this one quite appropriate for our needs.

3.6. Performance

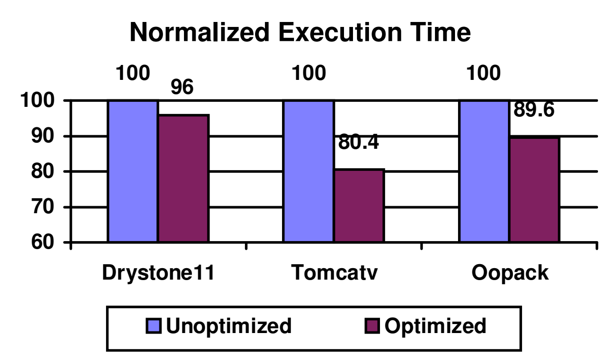

Figure 2.

The performance improvements under this model are presented on Figure 2 and Figure 3. These experiments were performed on a Sparc 5 workstation, with 32MB of memory using the Sun Java interpreter.

The performance results are quite interesting because actual execution time improvement is larger than the instruction count decrease. This is primarily a consequence of local common subexpression elimination, which replaces many array accesses with register references, and peephole optimizations, which replace existing instructions with the cheaper ones that do the same work.

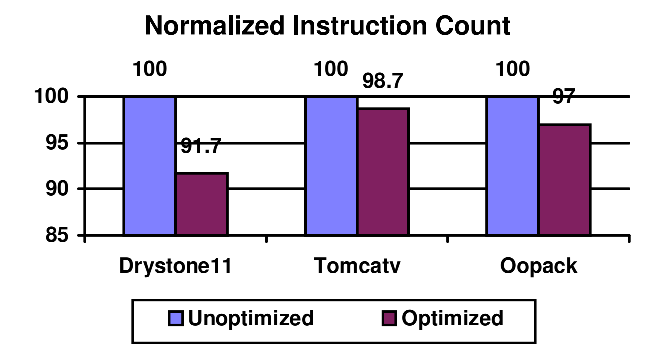

Figure 3.

In our opinion, the total performance gain possible under this compiling model is modest. Some additional optimizations can be implemented and some improvements can be made to the current ones, but unless the compilation model changes, we do not believe that big improvements are possible beyond what can be delivered by just-in-time compilers. For more significant improvements, at least a part of the current Java compilation and execution model will need to be sacrificed.

4. Relaxed Model

A more effective approach to optimizing Java would be possible if we could relax some rules of the current Java compilation model. In particular, given the standard Java object-oriented coding style—frequent method calls, many short methods—interprocedural optimization techniques such as aggressive method inlining could lead to substantive performance gains.

Unfortunately, interprocedural analysis as described in the literature for conventional languages is not permissible with the existing Java compilation and execution model, except with **final** classes and methods. The reason is that the compiler must assume late binding of the method calls inside the same class as well as to the methods from other objects. This is a consequence of not having the whole program in hand at compilation time.

However, if we change the model, some interprocedural optimizations become possible, *Object inlining*, described in the next section, looks at all the classes that are available to the compiler and, whenever it can prove that a method invocation must be static, inlines whole objects. *Code duplication* relies on the run-time mechanism to distinguish between the optimized and non-optimized classes and to decide which one to use for instantiation and which one for inheritance.

4.1. Object Inlining

One of the simplest and the most effective interprocedural optimizations is inlining. It has two major positive effects on the compiled code: elimination of subroutine call overhead and exposure of the method body to further optimization in the context of the original invocation. The potential negative effect is explosion of code size and the corresponding increases in compilation times.

Intuitively, inlining would be most effective for code that has many subroutine calls and short subroutines. Thus, object oriented languages and programs written in object oriented style would profit the most from this optimization. Many of the current C++ compilers implement extensive inlining and achieve significant performance improvements as a result.

4.1.1. The Method

Java presents two major impediments to inlining, which are illustrated by the sample code fragment Figure 4.

In this hypothetical code, there are two Java classes, each of which declares a private integer variable along with methods that access and modify that variable. The **main()** method of the class **Foo** contains a simple loop that does some computation and calls the internal method **inc()** of **Foo** and the external method **dec()** of the object **goo**, an instance of the class **Goo**.

There are two (different) reasons why those two methods calls cannot be inlined in Java. First, we cannot inline the call to **inc()** because neither the class **Foo** nor the method **inc()** are declared **final**. Thus, someone can later produce a class that inherits the class **Foo** and overrides the method **inc()**, but not the method **main()**, as shown on Figure 5.

It is obvious that if the method call to **inc()** was inlined in **Foo**, a call to **main()** of an instance of **FooFoo** would produce an incorrect result. Thus, we cannot inline calls to the non-final methods that belong to the same class since someone can inherit the compiled class later and override some of the inlined methods. Since the overriding methods would not be correctly invoked from the method where inlining took place, the resulting program would be erroneous.

```
class Foo{
  private int x = 0;

  public void inc(){
    x++;
  }

  public void main(){
    Goo goo = new Goo();
    for(int i = 0; i<10; i++){
      inc();
      x--;
      goo.dec();
    }
  }
}

class Goo{
  private int y = 0;

  public void dec(){
    y--;
  }
}
```

Figure 4.

The call to **goo.dec()** cannot be inlined for a different reason. Unlike C++, Java compiler transforms the source code to bytecodes for the Java Virtual Machine, which have a very similar structure to Java source code, with object instantiations, method invocations and language rules that directly reflect the rules for Java source code [Lindholm Yellin 1996]. In particular, bytecodes have to respect the privacy of object fields. In our example, the code from method **main()** cannot directly access the variable **y** from the class **Goo**, in either the source or in the bytecodes. Most Java Virtual Machine implementations would reject the bytecodes that violate the privacy rules.

```
class FooFoo extends Foo{
  public void inc(){
    x--;
  }
}
```

Figure 5.

We propose two methods for handling these two problems: *code duplication*, which addresses the first problem, and *object inlining*, which addresses the second one. These will be described in the next two subsections

4.1.2. Object Inlining

The idea of object inlining is very simple: instead of simply inlining method calls, we will inline *whole objects*, including data and code. By making the whole object local to the calling procedure, we gain immediate access to it's private data and make it possible to directly inline all the calls to that object's methods. Figure 6 shows how this would affect the method **main()** from our previous example.

```
 public void main(){
   // inlined: Goo goo = new
Goo();
   int goo_y = 0;

   for(int i = 0; i<10; i++){
     inc();
     x--;

     // inlined: goo.dec();
     goo_y--;
   }
 }
```

Figure 6.

There are two obvious problems with this approach:

1. it can only be applied to the objects that are instantiated inside the current class, and
2. the inlined object cannot be passed to other objects and methods as an argument.

The first problem cannot be solved unless the compiler has access to the whole program at compilation time, which is not the case in our framework. The second problem is less serious, since we could wrap a container object around our inlined data and pass it along to the called method as a parameter, restoring the inlined data after the call.

Let us illustrate this on an example: Suppose that we are passing the object **goo** to the **System.out.println()** method inside our **main()** method. Obviously, we cannot apply object inlining as above, since we wouldn't have an object to pass to the system method. What we need is an object whose data can be set to correct values and which can be passed to the given method. Our inlined data can be restored on return if the called method changes anything. To do that, we need direct access to our helper object's data, which can be provided by inheriting the inlined object and adding accessor and modifier methods to it, as shown on Figure 7.

Of course, heuristics have to be applied at compile time to determine if the benefits of inlining the object outweigh the cost of introducing a new object and the wrapping and unwrapping that must take place around method where it is passed as an argument.

```
class GooH extends Goo{
   public int get_y(){
     return y;
   }
   public void set_y(int val){
     y = val;
   }
 }
 public void main(){
   GooH gooH = new GooH()
   int goo_y = 0;
   for(int i = 0; i<10; i++){
     inc();
     x--;
     goo_y--;
   }
   gooH.set_y(goo_y);
   System.out.println(gooH);
   goo_y = gooH.get_y();
 }
```

Figure 7.

4.1.3. Arrays of Objects

Arrays of objects can also be inlined, with some additional caution in implementation. Allocation of an array of objects will be replaced by allocation of an array of data for each field in the given object. An instantiation of an object in the array will be replaced by initialization of the inlined data, as shown on Figure 8. The lines that are commented out represent the original code, as before.

```
public void main(){
  // Goo goo = new Goo[10];
  int goo_y[] = new int[10];
  for(int i = 0; i<10; i++){
    // goo[i] = new Goo();
    goo_y[i] = 0;
  }
  for(int j = 0; j<10; j++){
    for(int k = 0; k<10; k++){
      // goo[i].dec();
      goo_y[i]--;
    }
  }
}
```

Figure 8.

4.1.4. Performance

On Figure 9, we present some performance results obtained on applying object inlining by hand on the Intel Oopack benchmark. The actual implementation of object inlining is under way, and we hope to represent more complete and more detailed benchmark results once completed.

The graph on Figure 9 shows the execution times in seconds on an IBM 6x86 P166+ at 133MHz, with 40MB of memory and running under Windows 95. The first two bars show the execution times of the nonoptimized and optimized code (only the object inlining optimization is applied) using the original Sun bytecodes interpreter from JDK 1.0. The third and fourth bar show the execution times under the Symantec Café JIT compiler, again for nonoptimized version and version optimized by object inlining.

The data shows a promising performance improvement with both interpretation and JIT compilation. The actual speedup varies from 1.5 on Max to more than 7 on Complex. This huge performance gain is not surprising since Oopack is simply Linpack written in object-oriented style with many method calls and short methods—thus it is very suitable for this kind of optimization.

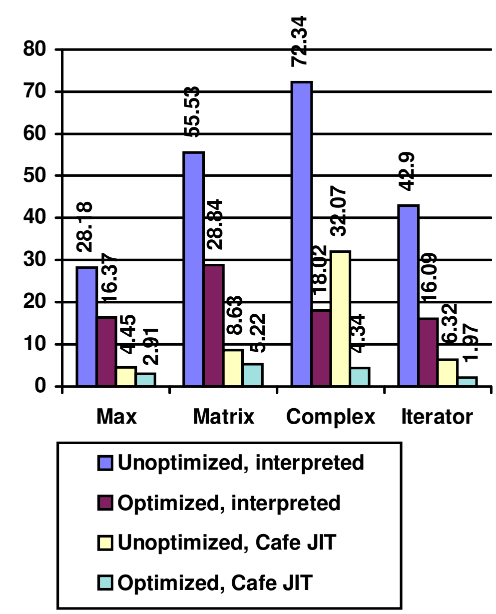

Figure 9.

4.2. Code Duplication

Code Duplication is a method for widening the spectrum of applicable interprocedural optimizations on the bytecodes, with very little sacrifice in portability and potentially big gains in the performance.

The problem we are trying to solve was described in Section 4.1.1. Let's look at the example at Figure 4 again. The compiler cannot inline the call to method **inc()** in the main loop because there is a possibility that someone may later inherit from the class **Foo** and override the method **inc()** but not **main()**, resulting in incorrect code. If the method **inc()** was declared as **final**, the inlining could be done safely (this is already done in most Java compilers). The problem arises with the methods that are not declared as **final**, as in our example.

The solution is very simple: the compiler generates two copies of the code, one under the assumption that the generated class will not be used for inheritance, thus permitting aggressive optimizations, and a second with the assumption that derived classes will be presented later. The compiler will apply to the second only optimizations that are safe under that assumption, as described in previous sections. We already showed the result of applying object inlining on the example on Figure 4 for those invocations that are *inheritance-safe*. Figure 10 shows the result of inlining under the assumption that the class **foo** will not be inherited from. Another optimization pass, such as value numbering, would eliminate both increment and decrement of **x** from the main loop.

There is obviously a problem with compatibility when this method is applied. The run-time environment must know that code duplication has been applied to the code, and act accordingly. Only minor changes to the run-time environment are required. When executing the code that is instantiating the class **Foo**, it should use the class from Figure 10, and when loading the class that is inheriting from class **Foo**, it should use the *inheritance-safe* class from Figure 6.

With careful naming conventions, we can completely eliminate the compatibility problem. It is obvious that the run-time environment that understands the code on which the code duplication has been performed will have no problems executing the ordinary bytecode, so we are safe from that side. Code duplication will create a subdirectory named **$_OPT_$** in the directory where the generated code is stored under the standard Java directory conventions, and put the optimized class files in it. This way, if the run-time environment does not understand code duplication, it will instantiate the class from the usual directory, which is still correct, though not as fast. In our example, if the class **Foo** was a part of the package **foo**, and the whole class tree is in directory *Main*, the compiler will put the *inheritance-safe* version of the class **Foo,** as in Figure 6, in file *Foo.class* in the directory *Main/foo/*, and the optimized version from Figure 10 in the file *Foo.class* in the directory *Main/foo/$_OPT_$/*. Run-time environment that understands code duplication will use *Main/foo/Foo.class* for inheritance and *Main/foo/$_OPT_$/Foo.class* for instantiation, whether the run-time environment that does not understand code duplication will still use *Main/foo/Foo.class* for both inheritance and instantiation, with lesser performance, of course.

```
class Foo{
 private int x = 0;

 public void inc(){
   x++;
 }

 public void main(){
   // Goo goo = new Goo();
   int goo_y = 0;
   for(int i = 0; i<10; i++){
     //inc();
     x++;
     x--;
     goo_y--;
   }
 }
}
```

Figure 10.

The main disadvantage this method is implied by its name: it duplicates the code. This could be a serious problem when executing over the Internet, since it actually doubles the downloading time. For some applications where the downloading takes the biggest part of the execution time, this could be unacceptable, and the code duplication should be disabled. It is our goal to automate the decision on the profitability of using code duplication in light of the increase in download time to get the highest performance possible.

It should be noted here that run-time environments that do not understand code duplication will not experience any download time increase. Since they don't know about the optimized versions of the class files, they will not download them. What still remains is the increase in the disk space the code is using on the server, which is clearly a far smaller problem.

The total performance increase to be expected is dependent on the interprocedural optimizations implemented. Code duplication does not increase the speed of the code, it is simply ameliorating the problem of not having the whole program available at the compile time. Its real value is that it enables other optimizations, particularly inlining of the intra-class methods, as described on our example. It also enables the whole spectrum of interprocedural analysis and optimizations on a single class file, which goes beyond the scope of this paper.

5. High Performance Model

Although Java has risen to prominence as a language for the Internet, there are many reasons why a programmer might find it attractive for building high-performance codes that would run on a secure server. In addition to its clean object-oriented design, type safety and automatic memory management, Java offers a way to write applications that can not only run on servers but which can also be securely downloaded to run at remote platforms over the Internet, albeit with some performance degradation. Furthermore, it seems likely that Java will be increasingly used for education at both the undergraduate and K-12 levels because of its ease of use, widespread availability, and absolute portability. Many industry pundits are predicting that it will supplant C++ as the language of choice for object-oriented commercial development.

Given these factors, it is easy to imagine Java being used in a high-performance execution environment where whole programs would be compiled directly to target machine without going through the Java VM. and its associated security and portability constraints. Note that usage in such an environment does prevent the program from being downloaded over the Internet—if the program is written in standard Java, it can still be compiled to the Java VM if the user later wishes to use it in a network environment.

In this section we explore some of the implications for compilation in complete freedom from the standard Java execution environment. The only restrictions that would remain would be those imposed by the semantics of the language. Clearly some performance problems would remain—the language would still need to be garbage collected and it would require that user-managed exceptions be handled precisely (see the next section). However, there would be no restrictions on the use of procedure inlining and other classical optimizations because the entire program would be available at compile time.

5.1 Whole Program Management

For a high-performance approach to be effective, it must be undertaken in the context of a program management system that will be comfortable for programmers to use. For example, it would not be acceptable for the entire program to be recompiled each time a single change is made to a single file or package. Thus, a truly usable high-performance implementation environment will need a mechanism for managing whole-program optimizations such as inlining and transformations based on interprocedural analysis.

Our previous work on whole-program compilation for Fortran [Cooper et al. 1986] led to the development of the R^n Environment, which supported *recompilation analysis* to limit recompilation in response to a change. In other words, part of the job of interprocedural analysis was to determine which files needed to be recompiled after a source change to a part of the program. In Java, for example, a change to a single method in a public class would require recompilation of any file in which that method had been inlined.

To address the problem, the R^n Environment introduced the notion of a *program compiler* whose job was to perform interprocedural analysis and optimization, including recompilation analysis, prior to selectively invoking the source compiler on program files. Such a system, which can be thought of as an intelligent version of the Unix utility MAKE, is a near requirement for high-performance compilation of Java.

5.2 Uniprocessor Compilation

To compile for a high-performance Uniprocessor, we plan to take full advantage of the body of compilation technology that has been developed for languages such as Fortran and C. The Massively Scalar Compiler Project at Rice [http://softlib.rice.edu/MSCP/results.ps] has developed a state-of-the-art experimental optimizing compiler for RISC machines that uses a more traditional quadruple-based intermediate language called ILOC (intermediate language for optimizing compilers). Our plan is to adapt the Java front-end to generate ILOC so we can experiment directly with a high-performance compiler back end. Prior to execution of the compiler, the interprocedural optimization framework would perform aggressive inlining to make the code more amenable to optimization.

In addition, we plan to experiment with a number of Java-specific optimizations such as ameliorating the need for garbage collection through program analysis [cite Barth and others].

Although Java presents many challenges to the compiler, we believe that there is no reason that the performance of

Java programs in an unrestricted compilation environment cannot come asymptotically close to equivalent Fortran and C programs.

5.2 Compiling to Parallel Machines

Most high-performance servers that Java might run on will be multiprocessors. In the case of most scientific applications and some business applications, the need for performance will make a *scalable* parallel system a requirement. Because they employ many processors with complex memory hierarchies, almost all scalable parallel systems will put a premium on using lots of parallelism while maintaining a high degree of locality in the computation.

Although Java supports explicit parallel programming through its threads mechanism, it is unlikely that most programmers will use this extensively for high degrees of parallelism. Furthermore, there is no facility for explicit locality management in Java. For these reasons we will experiment with directive-style hints, such as those found in High Performance Fortran (HPF) [Koelbel et al. 1993] for specifying both parallelism and data location. As an example consider class instantiation. It may be possible to use a directive to establish a home processor for each instantiation, thus ensuring that classes that will be used together will be allocated together. Furthermore, array classes can be distributed in the same way as arrays in HPF. Just as in HPF, class methods would be executed on the processor where the class is instantiated, thus implicitly introducing parallelism. Of course, the compiler would be free to migrate code whenever performance could be improved as a result.

With these and similar methods, we believe that it will be possible to build truly high performance applications in Java and run these with high efficiency on scalable parallel systems.

6. Handling Exceptions

The exception mechanism in Java presents an obstacle to compiler optimization regardless of which framework is used in the compiler implementation. In this section we discuss the problems that arise due to the Java exception model, and propose some solutions.

Java has an exact exception model [Gosling et al. 1996]: if the exception occurs in the program, the program state at the moment of exception must be visible to the user and no matter how many optimizations are performed, the state at an exception must be indistinguishable from the state that would result if all the instructions in the original source code before the one causing the exception, and none of the instructions after it, have been executed. In another words, if the exception occurs in the program, user should not be able to tell by examining the state that optimizations have been performed.

This, of course, greatly reduces the freedom of the compiler to move code within the program. None of the instructions that change the user-visible state of the program can be moved across an instruction that can cause an exception. In other words., those instructions that can trigger an exception cannot be interchanged or moved arbitrarily around the control flow graph, even if all of the data dependencies remain satisfied.

Because most Java instructions can cause an exception, the naïve approach—marking all of the instructions that can cause an exception and prohibiting the optimizer from moving instructions past a marked instruction—would effectively eliminate all the possibilities for performance enhancement by code movement.

Fortunately, although most Java VM instructions *can* cause an exception, in the normal program execution most of them will *not* do so. By exploiting this fact, it may be possible to achieve most of the performance of program written languages without exceptions (e.g., Fortran). If the compiler can prove that an instruction will not cause an exception in the given program context, optimizations can freely move the code around it.

There is ongoing research within the Rice University programming languages research effort on type analysis for the object oriented languages [Flanagan Felleisen 1996]. This work is highly applicable to the problem described, and we expect it to be able to identify most of the instructions that are not going to cause an exception, thus greatly easing the constraints on the code movement. The problem with this analysis is that it is potentially slow ($O(n^3)$ in the worst case), although it is very fast in practice. Some faster, though less powerful algorithms do exist however [Steensgaard 1996].

There are a number of other approaches to attacking this problem that could be used apart or in combination with aggressive exception analysis. Many of these are little tricks that are optimization dependent. Consider for example the code fragment in Figure 11.

```
for(i = k; a[i] < 0; i++){
 this.x = 3*k;
 sum = sum + a[i];
}
```

Figure 11.

Since **k** is loop invariant, the first assignment in the loop body is loop invariant, so loop invariant code motion would seek to move that assignment out of loop. However, since the loop header can cause an exception—the loop test involves an array access (let's assume that the exception analysis has been able to prove that **a[]** is not null, but not that the value of **k** at the entry to the loop is less than the length of a)—the optimizer cannot move this assignment. If the first iteration of the loop header caused an exception, the value of **x** would not have changed in the original program, but it would be changed in the optimized program.

In this case, we can solve the problem by simply peeling off the first iteration of the loop, as shown on Figure 12. Loop invariant code motion can now move the assignment to **this.x** outside the loop, where can be later completely eliminated.

```
i = k;
if (a[i] < 0){
  this.x = 3*k;
  sum = sum + a[i];
}
for(i = k+1; a[i] < 0; i++){
  this.x = 3*k;
  sum = sum + a[i];
}
```

Figure 12.

As already noted, this approach is completely dependent on the optimizations attempted. The basic idea is to insert in front of the code block we are trying to optimize some dummy instructions that would simulate the exception behavior of the instructions in the block and leave the program in the correct state if an exception occurs. This would eliminate the concern about the exceptions in a particular code block and enable free code movement inside it. For example, if a block of code is using a particular array, and exception analysis is unable to prove that the array reference is not null, a simple dummy access of the array before the given block of code will raise the exception if it is going to occur anyway.

Another approach that can help in addressing the problem of exceptions is to emulate methods used to make it possible to debug optimized code [Hennessy 1982]. These two problems are very similar, except for two very important details: in debugging optimized code, the debugger can simply inform the user if it was unable to recover the correct state of the program, but our system *must* return in the correct state, so the recovery code would have to be inserted in the executable program itself instead of the debugger, thus increasing the original code size. The idea is as follows: a whole method is enclosed in a **try** statement, with corresponding exception handler at the end. Thus, the handler would catch any exception that occurs in the code, and use the methods from [Henessy 1982] to recover the correct state of the object before allowing the user's exception handling mechanism to proceed. Of course, it is not always possible to recover the correct state, so this method would have to be restricted only to optimizations that can be undone.

An approach similar to the one just described would involve code reexecution instead of the recovery of the program state. This would require insertion of checkpoints, thus increasing the execution time. Special attention would have to be given to the reexecution of I/O operations. This method has the same structure as the one just described, with the exception handler reexecuting the non-optimized code instead of un-doing the optimizations to achieve the correct program state. The rollback and reexecution is well researched and applied in interactive environments [Archer et al 1981], distributed databases [Long 1994], and other fault tolerance systems [Alewine 1995].

7. Conclusions and Future Research

Over the next several years, the Java phenomenon should continue unabated. Increasingly, Java will be used as a general-purpose language rather than just a language for the Internet. This paper addresses some preliminary techniques for making Java more efficient. We believe that, using these techniques and others that are yet to be discovered, Java performance can rival that of languages like C and Fortran, even on scalable parallel systems.

Most of the approaches described in this paper have yet to be implemented. The project we have embarked upon at Rice will explore these methods in the context of a sophisticated compilation environment, with the goal of making Java the language of choice for high-performance server applications as well as those that execute over the Internet.

References

[Alewine 1995] Alewine, N. J., 1995. *Compiler-assisted multiple instruction rollback recovery using a read buffer.* NASA contractor report, NASA CR-199703.

[Archer at al. 1981] Archer, J. E. Jr., Conway, R. W., Schneider, F. B., 1981. *User Recovery and Reversal in Interactive Systems.* Technical Report, Cornell University, Computer Science Department, TR81-476.

[Aho et al. 1986] Aho, V. A., Sethi, R., and Ullman, J. D. 1986. *Compilers Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley.

[Briggs et al. 1993] Briggs, P., Shillingsburg, R., and Simpson, T., October 1993. *Dead Code Elimination.* Technical Report, Rice University. Available via anonymous ftp.

[Briggs et al. 1995] Briggs, P., Harvey, T., and Simpson, T., July 1995. *Static Single Assignment Construction.* Technical Report, Rice University. Available via anonymous ftp.

[Briggs et al. 1996] Briggs, P., Cooper, K. D., and Simpson, L. T., 1996. *Value Numbering.* Software - Practice and Experience.

[Bruno Lassagne 1975] Bruno, J., and Lassagne, T., 1975. *The generation of optimal code for stack machines.* Journal of ACM 23:3, 382-396.

[Cooper et al. 1986] Cooper, K., Kennedy, K., and Torczon, L., October 1986. *The impact of interprocedural analysis and optimization in the* R^n *programming environment*. ACM Transactions on Programming Languages and Systems, 491-523.

[Cytron et al. 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., October 1991. *Efficiently computing static single assignment form and the control dependence graph.* ACM Transactions on Programming languages and Systems, 13(4):451-490.

[Flanagan Felleisen 1996] Flanagan, C., Felleisen, M., 1996. *Modular and Polymorphic Set-Based Analysis: Theory and Practice.* Technical Report TR96-266, Rice University.

[Gosling et al. 1996] Gosling, J., Joy, B., and Steele, G. 1996. *The Java™ Language Specification.* Reading, Mass.: Addison-Wesley.

[Hennessy 1982] Hennessy, J., July 1982. *Symbolic debugging of optimized code*. ACM Transaction on Programming Languages and Systems, 4:3:323-344.

[Kennedy 1981] Kennedy, K., 1981. *A survey of data flow analysis techniques.* In Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis: Theory and Applications, Prentice-Hall.

[Koelbel et al. 1993] Koelbel, C., Loveman, D., Schreiber, R., Steele, G., and Zosel, M., 1993. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA.

[Lindholm Yellin 1996] Lindholm, T., Yellin, F. 1996. *The Java™ Virtual Machine Specification.* Reading, Mass.: Addison-Wesley.

[Long 1994] Long, J., 1994. *Checkpoint-based forward recovery using lookahead execution and rollback validation in parallel and distributed systems.* NASA contractor report, NASA CR-195760.

[Simpson 1996] Simpson, L. T., September 1996. *Value Numbering.* Technical Report, Rice University. Available via anonymous ftp.

[Steensgaard 1996] Steensgaard, B., January 1996. *Points-to Analysis in Almost Linear Time*. In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida.

[Wegman Zadeck 1985] Wegman, M. N., Zadeck, F. K., January 1985. *Constant Propagation with Conditional Branches.* Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages: 291-299.

Acknowledgments

We gratefully acknowledge the support of JavaSoft and the project leaders for the effort that started this work: Frank Yellin and Tim Lindholm. We also thank Tin Quan from The University of Illinois at Urbana-Champaigne, who was co-implementor of the bytcode-to-bytecode optimizer.