Static Interprocedural Optimizations in Java

Zoran Budimlic, Ken Kennedy

CRPC-TR98746
August 1998

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted August 1998

Static Interprocedural Optimizations in Java

Zoran Budimlic

Ken Kennedy

Center for Research on Parallel Computation

Rice University

Abstract

Interprocedural optimizations are important in Java because the object-oriented programming style encourages the use of many small methods. Unfortunately, such optimizations are difficult because of the nature of language structure and its security restrictions. A particular problem is the difficulty of knowing the entire program at any time prior to execution.

This paper presents new approaches to cloning and inlining that can be profitably used even in a single class. These optimizations are of particular interest for concurrent environments, since the correctness of their application is insensitive to concurrency. Results from our preliminary implementation are presented and ways to extend these methods are described. These methods can be thought of as strategies for *almost whole-program* analysis and optimization.

1.0 Introduction

Because Java encourages a programming style that uses many small methods, interprocedural optimizations are important for achieving high performance. One of the most effective interprocedural optimizations in object-oriented languages is *inlining* of method invocations, in which the body of the method is substituted for the invocation itself. Our experimental investigations have shown that inlining is particularly effective on *scientific* codes written in Java [2].

However, there are significant impediments to inlining in Java. Foremost among these is the difficulty of obtaining the entire program so that it can be determined whether a given method invocation can be safely applied. There are two issues that must be considered before inlining a method invocation:

1. At a given method invocation, it may not be clear which method is being invoked. This is due to the dynamic dispatch or virtual function mechanism in Java. When a method of a given object is invoked, the run-time system dispatches the version associated with the class in which that object was originally instantiated. This means that it is difficult to determine the exact method invoked for objects passed as parameters or extracted from data structures. Java shares this problem with all object-oriented languages. In such languages, an interprocedural type analysis can often precisely determine the instantiation class of an object at every point of use in the program [1, 7, 11, 18]. However, it is difficult to use such an analysis in Java, because the entire program is not known until the Java Virtual Machine (JVM) is started.
2. Even if the class in which the object was originally instantiated is known, it may not be possible to inline a method of that object without violating the security restrictions of the Java Virtual Machine [16]. For example, if the method accesses its object's private members and the method into which it is to be inlined is not permitted access to those members, the specific inlining would be precluded. In another language, where the entire program was known at compile time, the compiler could simply change the protection of variables during the code generation process, making the inlining possible. In Java, however, this would open a security hole because non-final classes might be refined through subtyping any time up until execution—any variables whose protection was relaxed to make inlining possible might be compromised by code added later.

These problems are illustrated by the example in Figure 1 below. In the method `work()` of class `Foo`, method `dec()` of class `Goo` cannot be inlined because it would violate encapsulation (problem 2). On the other hand, we cannot inline method `inc()` of class `Foo` because `inc()` might be overridden later, as it is in class `FooFoo`.

FIGURE 1. Inlining problems in Java

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

   public void inc(){
      x++;
}

   public void work(){
      Goo goo = new Goo();
      for(int i = 0; i<10; i++){
        goo.dec();       // inlining violates encapsulation
        x--;
        inc();           // inc may be overridden (see FooFoo)
      }
   }
}

class Goo{
   private int y = 0;
   public void dec(){
      y--;
   }
}
```

Potential refinement of `Foo`:

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

While it is tempting to overcome these problems by building a compiler for whole programs only, this approach would severely limit the usefulness of the resulting programs in the Ja va world, as they could never be even partially extended. On the other hand, it may be tempting to implement interprocedural analyses and optimizations in the Java Virtual Machine, extending existing JIT compilation strategies. However, the expense of interprocedural analysis, especially type analysis, makes it unlikely that this approach will prove practical.

Our research has focused on ways to get some of the benefits of interprocedural optimization without giving up on e xtensibility. The strategy we are pursuing might be called *almost whole-program* optimization. The basic idea is to analyze and optimize the portion of the program that is known to be fixed. In the extreme, this would mean class-by-class analysis. In places where we cannot see parts of the program, the compiled code would incur performance penalties relative to what was possible with whole-program compilation. In Section 5.0 we discuss features under consideration for Java that might make this strategy more powerful by permitting the developer to fix large parts of a code.

We will illustrate the strategy through two code optimizations:

1. *code specialization*, which is designed to overcome problem 1 described above, and
2. *object inlining*, which overcomes problem 2 in cases where the instantiation of an object occurs in the same method where a method of that object is invoked.

We proposed both of these approaches in our previous paper [2], but we had not yet implemented them in a compiler. In this paper we report on e xperience with implementations in the Ja vaSoft optimizer and discuss ways to extend their applicability.

2.0 Code Specialization

A simple example illustrating the impact of dynamic dispatch on interprocedural optimization is given in Figure 2 below

FIGURE 2. An example for code specialization.

```
class Foo{
        int x = 1;

        public void work(){
          for (int i = 1; i<100; i++){
            inc();
          }
        }

        public void inc(){
          x++;
        }
}
```

In this example, the compiler cannot conclude anything about the target of the method call to `inc()` in `work()`, since the implementation of `inc()` is dynamic by default, i.e., any subclass of

`Foo` that overrides `inc()` but inherits `work()` from `Foo` would have the call dispatched to the overriding method `inc()`.

If the compiler is restricted to looking at only one class at the time, it must be overly conservative and assume that the call to `inc()` is dynamically dispatched and that it could refer to an unknown method. Thus, an interprocedural optimization such as inlining would be precluded.

To overcome this problem, we have implemented a version of procedure cloning, called *code specialization*. The main idea behind this optimization is to generate a specialized version of the class method that assumes static binding between the method calls inside the class and eliminates dynamic dispatches. A run-time test is inserted to distinguish between the cases when the method is called as a member of the exact class that is compiled, or as a member that some subclass has inherited from the compiled class. Applying this optimization, the example from Figure 2 is transformed into the code shown in Figure 3.

FIGURE 3. Class `Foo` from Figure 2 after Code Specialization

```
class Foo{
        int x = 1;

        public void work(){
          if (this.getClass() == Foo.class){
            work$$SPEC();
          else{
            for (int i = 1; i<100; i++){
              inc();
            }
          }
        }

        // A specialized version of the method work()
        final private void work$$SPEC(){
          for (int i = 1; i<100; i++){
            inc(); // Static binding can be assumed here
          }
        }

        public void inc(){
          x++;
        }
}
```

Here, a specialized version of the method `work()`, called `work$$SPEC()` has been created, and all calls from this method to other methods of class `Foo` can be assumed to be statically bound. In particular, the method call to `inc()` can be inlined at the calling site in `work$$SPEC()`. Furthermore, the call to `work$$SPEC()` from within `work()` can also be inlined along with its contained call to `inc()`.

We are also eliminating unnecessary synchronization: specialized versions of method do not have to be synchronized if the original methods are; if both `work()` and `inc()` are declared to be synchronized on the same object and for some reason (recursion for example) `inc()` cannot be inlined in `work$$SPEC()`, a call to `inc$$SPEC()` is inserted instead, eliminating a double synchronization check.

The run-time test of the exact class type that is performed at the beginning of the modified method could be quite expensive relative to the benefits the knowledge of the static binding of calls may give to the compiler. An additional speed-up is achieved by performing the call only once on the initialization of the object, and caching the result in an instance variable. Then only an instance variable lookup is needed to determine if the e xact class type of the class being e xecuted is the type of the class that is specialized. This of course adds space o verhead of an additional boolean variable per each object for each superclass that is specialized, which could be significant, so an appropriate heuristic based on the size of the specialized object versus the additional space for test caches may be needed.

We have implemented code specialization as described abo ve, followed by inlining of the resulting static method calls. Given that this is the only interprocedural optimization we have applied, it is not surprising that the performance improvements are modest. For most of the test cases we, the running time was unchanged or slightly longer (due to the added run-time tests). Of the rest, most achieved a modest speedup of 5-6%, while several cases saw speedups of 20-30%.

Until the implementation of interprocedural optimizations that will take a full adv antage of the static information now available to the compiler, as well as local optimizations that will take advantage of increased basic-block size and local information available after inlining, the impact of this optimization cannot be fully evaluated. Our previous experience with interprocedural optimizations and inlining gives us reason to believe that overall, this optimization will be quite beneficial.

3.0 Object Inlining

Inlining is one of the simplest and the most effective interprocedural optimizations. 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, at the cost of increased code size and the corresponding increase in compilation time. 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.

Java presents another impediment to inlining, in addition to the unavailability of the exact type that is discussed in the previous section. Under the assumption that the exact type is determined, either by interprocedural analysis of the part of the program that is available or by ensuring the exact type of the object using code specialization, inlining could still be precluded, as it is illustrated by the sample code fragment in Figure 4.

FIGURE 4. An example for object inlining

```
class Foo{
        public void work(){
          Goo goo = new Goo();
          for (int i = 1; i<100; i++){
            goo.inc();
          }
        }
}

class Goo{
        private int j = 1;
        public void inc(){
          j++;
        }
}
```

In this hypothetical code, method `work()` of the class `Foo` instantiates an object of the type `Goo`. It would be quite easy for the compiler to see that the variable `goo` is indeed of the type `Goo`, and that all references to its methods (including the one to `inc()` inside the loop) are statically determinable at the compile time. However, compiler cannot inline the call to `inc()`. Unlike C++, a 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 [16]. In particular, bytecodes have to respect the privacy of object fields. In our example, the code from method `work()` cannot directly access the variable `j` from the class `Goo`, in either the source or in the bytecodes. Most Java Virtual Machine implementations would reject programs that violate the privacy rules.

The idea behind 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 its private data and make it possible to directly inline all the calls to that object's methods. There are multiple benefits emerging from this approach:

- The variables that were private to the class `Goo` become local variables for the procedure `work()`, thus enabling direct access to them, and enabling inlining of the call to `inc()`
- In addition to eliminating method calls due to inlining, access to the object's variables are also faster. Instead of using a field access to an instance variable in the inlined method, a simple reference to a local variable is used.
- The object itself is eliminated, thus avoiding the need for dynamic allocation of the space used by the object, as well as potential garbage collection when the object is not needed anymore.

The sample code from Figure 4 after object inlining is shown on Figure 5.

FIGURE 5. Example from Figure 4 after object inlining

```
class Foo{
      public void work(){
        // Goo goo = new Goo();       // eliminated
        goo$$j = 1;                   // replaced the above line
        for (int i = 1; i<100; i++){
          // goo.inc();               // eliminated
          goo$$j ++;                  // replaced the above line
        }
      }
}
class Goo{
      private int j = 1;
      public void inc(){
        j++;
      }
}
```

Performance tests on our implementation of object inlining confirmed the results that are earlier obtained by performing this optimization by hand [2]. The statistical data is shown on Table 1.

TABLE 1. Performance data for object inlining.

Benchmark	Interpreted	Object Inlining	% Gain	JIT	JIT + Object Inlining	% Gain
Max	20.95	12.558	67%	0.951	0.761	25%
Matrix	42.24	19.348	118%	1.843	0.761	142%
Complex	53.787	46.287	16%	13.94	13.540	3%
Iterator	32.937	8.332	295%	1.402	0.25	461%

We performed our tests on the same set of benchmarks as in our earlier work [2]. The Interpreted, Object Inlining, JIT and JIT+Object Inlining columns represent the running times in seconds for interpreted bytecodes, interpreted bytecodes with object inlining performed, bytecodes executed on Symantec JIT 210.063 and bytecodes with object inlining on the Symantec JIT, respectively. All tests were done on a 200 MHz Pentium Pro with 64 MB of memory running Windows NT Workstation 4.0, with optimization flag -O turned on during compilation.

Note that the only disappointing performance improvement was on the Complex benchmark. This benchmark makes extensive use of arrays of objects (complex data types), which are not currently inlined by our implementation. The next section contains a discussion of the problems associated with this form of inlining, which our previous experiments indicate will be highly successful [2].

4.0 Inlining Arrays of Objects

Additional problems arise when attempting to inline arrays of objects that are locally allocated inside a method. This case is of particular interest when compiling scientific programs that operate on arrays and matrices of complex numbers. Some of these problems are illustrated in Figure 6.

FIGURE 6. An example for using an array of objects

```
public void sumParts(){
  Complex[] array = new Complex[N];
     for (int i = 0; i < array.length; i++){
      array[i] = new Complex(i*2, i*3);
  }
  Complex result = new Complex(0,0);
  for (i = 0; i < array.length; i++){
      result.setRe(array[i].re() + result.re());
      result.setIm(array[i].im() + result.im());
  }
  System.out.println(result);
}

class Complex{
   double re,im;
   Complex(double re, double im){
      this.re = re;
      this.im = im;
   }
   Double re(){return this.re;}
   Double im(){return this.im;}
   void setRe(double re){this.re = re;}
   void setIm(double im){this.im = im;}
   void print(){
      System.out.println("Real : " + re);
      System.out.println("Imaginary : " + im);
   }
}
```

The goal of inlining in this case is to inline the *entire array of objects*, rather than just individual objects within the array. This means that we would replace the instance variables of objects in the array with arrays of variables. In the example from Figure 6, we would replace the instance variables `re` and `im` with arrays `array$re` and `array$im` to yield the code shown in Figure 7 below. Note that in this version, the complex object `result` has been inlined using the standard object inlining approach described in Section 3.0.

To apply this optimization, we must ensure that two conditions are met:

1. Every array element that is actually used in the body of the procedure where the inlining is to take place must be initialized to an object of the same known class (e.g., the `Complex` class). This restriction insures that we can carry out the inlining by creating arrays of the instance variables and that a single inlined version of each method can be used in loops.

FIGURE 7. Optimized method `sumParts` from Figure 6

```
public void sumParts(){
  double[] array$re = new double[N];
  double[] array$im = new double[N];
     for (int i = 0; i < array$re.length; i++){
      array$re[i] = i*2;
      array$im[i] = i*3;
  }
  double result$re = 0;
  double result$im = 0;
  for (i = 0; i < array$re.length; i++){
      result$re = array$re[i] + result$re;
      result$im = array$im[i]+ result$im;
  }
  System.out.println("Real : " + result$re);
  System.out.println("Imaginary : " + result$im);
}
```

2. The instantiation and uses of the array must all take place within the class and neither the array nor any object in it may be passed outside the class where inlining takes place. This ensures that no attempt will be made to access any object of the array using the defined methods, which would not be possible because we will never instantiate any object in the array. Section 5.0 will discuss ways to relax this restriction.

Standard scalar and array data flow analysis techniques can be used to ensure that these conditions are met. For example the analysis of scalar v alues [3,13,17] can permit the determination that the range of instantiation includes the range of use for the array. This analysis is similar to array kill analysis in parallelization [12]. Alias analysis is necessary to prove that parts of the allocated array are not aliased to other arrays that escape the scope of the method where it is being inlined. These analyses are widely discussed in the literature and pose no obstacles, aside from programming effort, to the approach we propose here.

5.0 Expanding the Range of Applicability

5.1 Object Reconstruction

Object inlining, as described, could be applied only to the objects that are instantiated locally inside the method. If the method passes a reference to the inlined object to some other method (either by invoking it directly or via a return statement), the object could not be inlined with the proposed methodology. This is a serious restriction, because it is quite common to construct an object inside a method, work on it, and return it. Can we extend the method to handle those cases as well?

An obvious approach to this problem would be to reconstruct the object at the end of the method, before returning it. This could be accomplished in either of two ways: one would be to analyze the inlined object's constructors and determine if any of them (or any combination of them) is suitable for setting all of the object's fields to the desired values. Although this would not work in every case, we expect that most small objects could be reconstructed in this way. Another problem is presented by the profitability of such a scheme—the cost of reconstructing objects might outweigh the benefits of the inlining.

Alternatively, we could add an extra constructor to the class that is object-inlined, which would take arguments for all the fields of the object and set them to the desired v alues. This raises some security considerations: Under the current security model in Java, a compiler is not allowed to add publicly-accessible methods or fields to the class that is being compiled. It is easy to imagine a scenario where the internal consistency of the object is maintained by the implementation of the class's methods (for example, complex object for some performance reasons may have both orthogonal and radial representation internally, that are kept in consistency by the implementation of the methods that the class implements). Although the constructor is not a "regular" method, in the sense that it is not possible to call it directly, adding a constructor that would allow someone to create object with arbitrary and thus possibly inconsistent state may violate security.

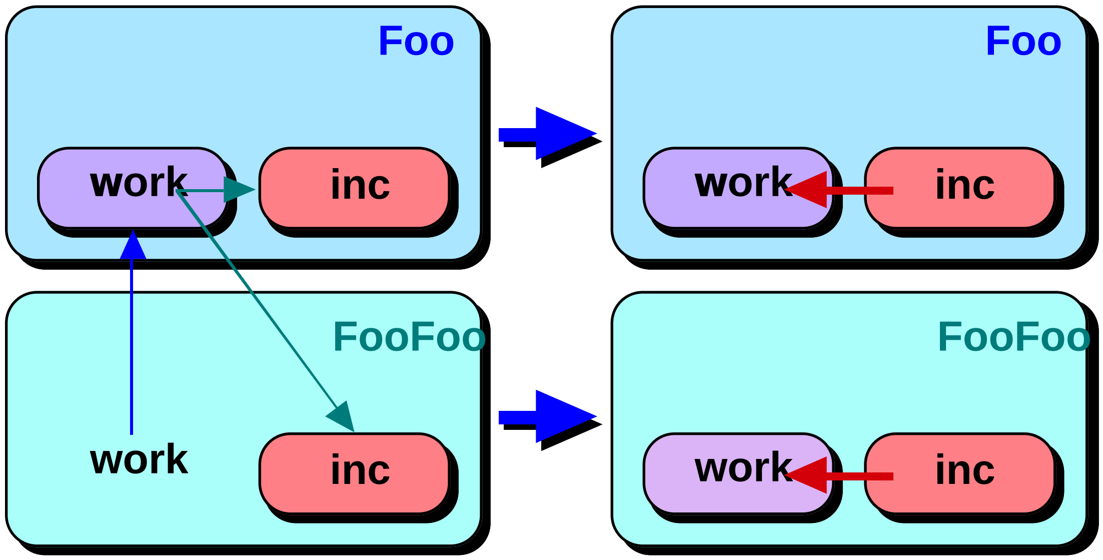
5.2 Signed Archive Files

The impediments to optimization due to security are already recognized by JavaSoft, and they are in process of developing a somewhat relaxed security model, that would be based on *JAR (JavaARchive)* packages. Under this scheme, classes would be packaged into JAR files, where the security mechanisms would be loosened. A JAR file can be signed, so no one can add or delete classes or methods from it. This model would help the implementation of many interprocedural optimizations, not only object inlining. Although this model still doesn't solve the problem in general (only a model where the whole program would be available at the compile time would allow full range of interprocedural optimizations), we expect that for a significant number of scientific programs much better optimization could be achieved for programs that were almost complete. The e xtensions discussed in the follo wing subsections are based upon our speculations about an ideal relaxed security model for signed JAR files.

5.2.1 Extending Code Specialization

Let us examine how some of the methods we have described here could be improved if we had substantially more of the program available for analysis. Classes that are defined locally in the same file as the class that uses them could be modified freely, since they are not accessible outside the file where they are defined. If the entire program can be packaged into a single file, standard whole-program analysis techniques can be used to optimize these procedures. Let us begin by considering the case of *code specialization*. If no class in the tree is known outside the file, code specialization takes a particularly simple form, illustrated in Figure 8:

FIGURE 8. Graphic view of extended code specialization.



1. First preprocess the class definition where inlining is desired and the definition of all classes that extend it, connecting each definition of a method at a le vel in the deri vation tree with every use of that method in the same class or in one that it is derived from. Note that a method definition `inc()` can be used in another method `work()` only if `work()` is associated with a class above or in the class containing the definition of `inc()` in the hierarchy tree and no intermediate class contains an overriding definition of `work()`.

2. Starting at the top of the hierarchy tree, at each point where inlining is desired, see if more than one definition of the method to be inlined might actually be used due to dynamic dispatch. Suppose one definition is at the same level and all others are at lower levels.
3. Insert a copy of the method in which inlining is desired into each class at a lower level containing a definition that might have been used. Then inline each definition in the cop y at the same level.

In the example above, method `inc()` is defined in both class `Foo` and an extension, `FooFoo`. There is only a single definition of method `work()`. If `inc()` is to be inlined, method `work()` must be cloned. In this case however, we do not need to insert a test of any kind, as each class will get its own specialized version of `work()`.

This extended version of code specialization fails to work if some class in the hierarchy is known outside the file being optimized. In this case, we can use the procedure described above and then apply the version of code specialization described in Section 2.0 for all classes that may be extended by code outside the JAR file.

5.2.2 Extending Object Inlining

If we know the entire program, global type analysis combined with cloning can often be used to substitute static invocation for dynamic invocation, permitting inlining everywhere. When the program consists of more than one file, two problems can arise:

1. The class of the object is defined in a different file from the one in which it is used, a common organizational practice in object-oriented languages. This problem can be overcome by an extended form of object inlining for objects of the class that are instantiated and used entirely in one file. In the case of a file that contains more than a single class definition, a copy of the original class definition can be inserted in file where the usage occurs and all reference to objects of the original class can be changed to reference the new class. Now methods from this class can be freely inlined in the file. Even if the object is passed out of the file, it may be possible to construct an object of the original class by field copying as described in Section 5.1.
2. Some objects may be instantiated in one file and passed into another file. Object inlining can again be used if, as above, we define a copy of the object's class that would be known only internally within the file where the object is used and if we can find some way to construct an object of the new class from one of the original class. In this case, we insert constructors at each point where an object of the original class is passed in and we have reduced the problem to single-file inlining.

Clearly these extended versions of code specialization and object inlining could be extremely effective in large JAR files. It remains to be seen, however, whether some of the object construction and reconstruction techniques can be widely applied.

6.0 Related Work

The term "object inlining" was introduced simultaneously by Dolby [9] and Budimlic and Kennedy [2]. They both present a similar idea: inlining whole objects to eliminate indirection and improve the performance of the generated code. There are some major differences, however. Dolby focuses on whole program optimization, which is a significant restriction for Java programs in a given context. Additionally, Dolby only inlines objects inside other objects, reducing heap allocation cost since only one allocation for the resulting object is required instead of separate allocations for all contained objects, while our technique converts objects into local variables, essentially changing heap to stack allocation.

There has been a significant amount of work in functional languages community done on *unboxing*. Leroy [15] introduces new constructs to core ML language: *wrap* and *unwrap*, which are inserted in the code to handle boxing and unboxing of the objects. Our technique is similar to his, with the main difference being that we are applying it to an object oriented language (Ja va) and we are not introducing any new constructs to the language itself.

Cooper, Hall and Kennedy [6] pioneered the work on *procedure cloning*, that laid the grounds for code specialization. Their approach uses procedure cloning to make copies of chosen procedures based on the values of their parameters. Call sites are partitioned and created copies of procedures are then individually optimized based on the restricted number of possible call sites for those procedures and the known value of some of their parameters.

Chambers et al. [5] and Dean et al. [8] extended this idea further to clone procedures based on the *type* of the parameter(s) passed, thus creating a more precise call graph information in the presence of polymorphism. This has enabled them to apply more traditional compiler optimizations (such as inlining). Our approach in code specialization is similar to theirs, main difference being that we do not require knowledge of the whole program to perform our optimization, though the main benefits of our optimization can be experienced if the code that is available at the compile time is indeed the major portion of the code that is being executed.

7.0 Conclusions and Future Research

We have described two optimization strategies:

1. *code specialization*, a variant of method cloning, and
2. *object inlining*, a general analogue of method inlining that also inlines instance variables.

These two strategies are used to overcome barriers to interprocedural optimizations, especially inlining, that are caused by dynamic dispatch and the security features of Java. Both strategies have been implemented in an optimizing compiler that is developed at JavaSoft, and is largely based on the JDK 1.1 `javac` compiler. Preliminary experiments have shown results that are at worst promising and at best spectacular.

The main problem with these methods are the limits of their applicability. In this paper, we have shown how object inlining can be safely extended to arrays of objects and to objects that are passed to methods of other classes. In addition, we have described how to generalize these methods to take advantage of forthcoming features of the Java environment, specifically JAR files, that would permit classes to be collected into signed files within which security restrictions were relaxed. We plan to implement these extensions and show their efficacy for large codes written in Java, particularly scientific codes.

The Java language presents both opportunities and challenges to the compiler optimizer. The opportunity is to work with a clean, pointer-free language that should be much easier to analyze. The challenge is to structure the compiler and optimizations so that Java security and extensibility features, treasured aspects of the language, need not be compromised. Interprocedural optimizations offer the most promise for improvement while presenting the greatest challenges to maintaining security and extensibility. The methods described here represent a small first step toward meeting those challenges.

8.0 References

[1] O. Agesen. *Concrete type inference: delivering object-oriented applications.* Ph.D. thesis, Stanford University, 1995.

[2] Z. Budimlic and K. Kennedy. *Optimizing Java: theory and practice.* Concurrency: Practice and Experience 9(6), 445-463, 1997.

[3] P. Briggs, K. Cooper, and L. T. Simpson. *Value numbering*. Technical report CRPC-TR95517-S, Center for Research on Parallel Computation, Rice University, November 1994.

[4] D. Callahan and K. Kennedy. *Analysis of interprocedural side effects in a parallel programming environment*. In Proceedings of the First International Conference on Supercomputing. Springer-Verlag, Athens, Greece, June 1987.

[5] C. Chambers and D. Ungar. *Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented programming language*. In Proceedings of the ACM SIGPLAN '89 Conference on PLDI, 24(7):146-160, July 1989.

[6] K. Cooper, M. Hall and K. Kennedy. *Procedure cloning*. Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, 96-105, April 1992.

[7] J. A. Dean. *Whole program optimization of object-oriented languages.* Ph.D. thesis, University of Washington, 1996.

[8] J. Dean, C. Chambers, and D. Grove. *Selective specialization for object-oriented languages*. In Proceedings of the ACM SIGPLAN '95 Conference on PLDI, pages 93-102, June 1995.

[9] J. Dolby. *Automatic Inline Allocation of Objects*. In Proceedings of ACM SIGPLAN conference on POPL, Las Vegas, Nevada, June 1997.

[10] K. Driesen, U. Hölzle and J. Vitek. *Message dispatch on pipelined processors*. In Proceedings ECOOP'95, Aarhus, Denmark. Springer-Verlag, August 1995.

[11] C. Flanagan and M. Felleisen. *Modular and polymorphic set-based analysis: theory and practice*. Technical Report TR96-266, Rice University, 1996.

[12] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. *Interprocedural analysis for parallelization: A case study.* In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 1995.

[13] P. Havlak. *Interprocedural symbolic analysis*. Ph.D. thesis, Rice University, Dept. of Computer Science, May 1994.

[14] P. Havlak and K. Kennedy. *An implementation of interprocedural bounded regular section analysis*. IEEE Transactions on Parallel and Distributed Systems, 2(3):350 360, July 1991.

[15] X. Leroy. *Unboxed Objects and Polymorphic Typing*. In Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on POPL, 177-188, Albequerque, New Mexico, January 1992.

[16] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification.* Reading, Mass., Addison-Wesley, 1996.

[17] R. Metzger and S. Stroud. *Interprocedural constant propagation: an empirical study. A*CM Letters on Programming Languages and Systems, 1(3), December 1992.

[18] J. Plevyak and A. Chien. *Precise concrete type inference for object-oriented languages*. In Proceedings of the Ninth Annual ACM Conference on OOPSLA, 324-340, 1994.

[19] R. Triolet, F. Irigoin, and P. Feautrier. *Direct parallelization of call statements*. In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices 21(7), pages 176-185. ACM, July 1986.