Improving Compilation of Java Scientific Applications*

Zoran Budimlić† Mackale Joyner† Ken Kennedy†

Abstract

Java is a high productivity object-oriented programming language that is rapidly gaining popularity in high-performance application development. One major obstacle to its broad acceptance is its mediocre performance when compared to Fortan or C, especially if the developers use object-oriented features of the language extensively. Previous work in improving the performance of object-oriented, high-performance, scientific Java applications consisted of high level compiler optimization and analysis strategies, such as class specialization and object inlining. This paper extends prior work on object inlining by improving the analysis and developing new code transformation techniques to further improve the performance of high performance applications written in high-productivity, object-oriented style. Two major impediments to effective object inlining are object and array aliasing and binary method invocations. This paper implements object and array alias strategies to address the aliasing problem while utilizing an idea from Telescoping Languages to address the binary method invocation problem. Application runtime gains of up to 20% result from employing these techniques. These improvements should further increase the scientific community's acceptance of the Java programming language in the development of high-performance, high-productivity, scientific applications.

1 Introduction

Java has become a popular language over the past decade, enabling many application developers to experience high productivity and efficiency. Java encourages an object-oriented style of programming that models, for many people, the way they naturally think. As a result, scientists can become more efficient at developing applications, especially high-performance scientific applications. Java was initially not accepted in the scientific community because of its mediocre runtime performance compared to languages such as C and Fortran. Improvements in Java translation have made the language more widely accepted [3, 18, 19, 20].

This paper describes improvements to the analysis and trasformation techniques previously reported in the JaMake compiler [6]. The improvements are made to object inlining [9, 18]. an optimization that improves runtime performance by eliminating field accesses, reducing object allocation cost, and enabling better memory management [7, 18]. Studies have shown object inlining to be a viable optimization in improving scientific application performance [6, 8, 24]. The idea of object inlining dates back to boxing and unboxing in languages with polymorphic typing such as ML [23]. Optimal data representations were statically determined to take advantage of the underlying hardware, thereby increasing application performance. Dolby's object inlining transformation implemented for C++ inlines global objects within containers [18]. Budimlić's object inlining transformation for Java inlines global objects as well as local objects found in methods, and most importantly, arrays of objects which are crucial for scientific high-performance applications [3]. This paper discusses improvements to Budimlić's object inlining, yielding performance improvements of up to 20%.

*This work is sponsored by Hewlett-Packard and LACSI.

†Computer Science Department, Rice University, Houston TX 77005. {zoran, mjoyner, ken}@cs.rice.edu

```
//original version
class Foo {
   int x;
   double y;
}
class Bar {
   Foo myFoo;

   public int getX() {
      return myFoo.x;
   }

   public double getY() {
      return myFoo.y;
  }
}
```

```
//inlined version
class Foo {
   int x;
   double y;
}
class Bar {
   int myFoo_x, myFoo_y;

   public int getX() {
      return myFoo_x;
   }

   public double getY() {
      return myFoo_y;
   }
}
```

Fig. 1. *Object inlining example showing instance field myFoo being inlined.*

1.1 Object Inlining

Object inlining [3, 19] is an optimization that improves performance by inlining an object, thereby reducing the costs of accessing the object's fields, avoiding dynamic dispatch when invoking its method, as well as decreasing the cost of memory allocation and deallocation. *Figure 1* shows an inlining example.

Object inlining studies have shown that object inlining can drastically increase the runtime performance of an application [7, 8, 24]. There is a number of safety conditions that an object must satisfy before being inlined. These conditions ensure that the object inlining transformation does not change the semantic behavior of the application. This paper extends Budimlić's work by first analyzing the safety conditions to determine the prevalent violated conditions preventing objects from being inlined. By relaxing the constraints of the prevalent safety conditions, we enable more objects to be inlined, thereby improving runtime performance while preserving the observed behavior of the original application. The following section discusses the methodology used to determine which safety conditions eliminate the most objects from inlining, a necessary first step in extending previous technique.

1.2 Preliminary Object Inlining Study

We performed a preliminary object inlining study to understand which object inlining conditions prohibit the most objects from being inlined. We conducted the study using 10 open-source Java scientific applications from the *SourceForge* repository of open-source applications. *Table 1* includes the number of source files, methods, and percentage of inlineable objects for each scientific application. The scientific applications were taken from a variety of university and corporate affiliates. The 10 applications are the following:

- JQuantity - A project that allows end users and programmers to make scientific calculations in exact quantities or to obtain quantities with a known bounded error.
- Jmol - A molecule viewer.
- JAMA - A basic linear algebra package that provides classes for constructing and manipulating dense matrices.
- Jsky - A suite of components for use in astronomy.
- Jcckit - A library and framework for creating scientific charts.
- Biomer applet - A molecule modeling program.

- Jmat - The Java matrix tool package that provides Matlab-like functions.
- Ostermiller - Libraries for working with scientific numbers.
- SSH tools - A suite of Java ssh applications.
- JGraph - A powerful, lightweight graph component.

Filename	Files	Methods	Initially Inlineable Objects	Inlineable Objects	% Inlineable
JQuantity	118	1374	648	39	6.0%
Jmol	167	2303	656	105	16.0%
JAMA	9	118	169	165	97.6%
Jsky	363	5505	2061	363	17.6%
Jcckit	78	371	352	150	42.6%
Biomer	51	641	440	181	41.1%
JMAT	78	1042	548	90	16.4%
Ostermiller	43	540	118	101	85.6%
SSH tools	427	3812	1389	197	13.8%
JGraph	42	922	560	105	18.8%

Table 1

Number of files, methods, and inlineable objects for each scientific application, as well as the percentage of objects that did not violate any safety conditions

1.3 Safety Conditions Preventing Object Inlining

There are several object inlining conditions that object inline analysis uses to determine when an object cannot be inlined. For example, an object being passed to a method whose source code is unavailable cannot safely be inlined. The results of the study conducted show which conditions prevent the most objects from inlining in our scientific applications suite. Our results revealed that in practice, some of the safety conditions are much more frequently violated than the others. The rationale for why some of these safety conditions exist to ensure correct code will be visited in greater detail in the next section. These safety conditions include:

1. An object was passed to a method whose source code was unavailable. As a result, the object potentially escapes the compiler scope and can be modified, preventing object inlining.
2. An object was aliased. Inlining objects with multiple references in Java can produce incorrect results.
3. An object that is an element of an array was aliased. When a reference to any element in the array is stored, the entire array cannot be inlined.
4. An object or an array of objects was passed to a method call whose formal parameter at that position was not inlineable.
5. An array of objects was aliased to a non-inlineable array.
6. An object or array of objects was assigned to the result of a method call whose return object was not inlineable.
7. An object was a return object in a method whose result is not inlineable.

Filename (# of files)	Binary Method	Object Aliased	Array Element Aliased	Parameter Not Inlineable	Array Aliased	Return Not Inlineable	Return Object
JQuantity (79)	34	67	0	181	2	38	7
Jmol (167)	77	85	5	46	6	82	11
JAMA (9)	0	3	0	0	0	2	0
Jsky (363)	221	309	5	487	29	192	25
Jcckit (78)	20	41	0	0	0	6	0
Biomer (51)	19	153	7	48	11	9	0
JMAT (78)	7	44	0	111	1	39	0
Ostermiller (43)	2	3	0	5	2	6	0
SSH tools (427)	96	204	1	214	2	154	4
JGraph (42)	64	73	2	117	0	7	4

Table 2

The table shows the number of objects each safety condition eliminated. There is a large disparity in the number of objects eliminated by each safety condition.

1.4 Preliminary Study Results

Obtaining accurate information about the impact of individual safety conditions on object inlining effectiveness is not a trivial task. For example, if a single object gets passed to a binary method ten times, should the 'Binary Method' condition be counted ten times, or only once? Using the 10 open-source scientific applications, we analyzed the number of objects eliminated from inlining for violating each condition. An object is eliminated from inlining the first time it violates any safety condition. Unfortunately, the safety conditions are not separable. Disabling one condition can affect the number of objects counted as violating a different condition. *Table 2* shows how many objects each safety condition prohibited from inlining because it was the first condition violated by the object. This information is useful, but insufficient.

These results show how many objects were eliminated per safety condition, if that condition was the first one eliminating that object. However, these results do not show the relationship some safety conditions have with others, thereby being slightly misleading. Some of the safety conditions depend on other conditions to be violated first before they can prohibit an object from being inlined. We can remove these dependent results in each condition by allowing an object to be eliminated from inlining only if it violates the given safety condition. *Table 3* shows that for example the number of objects eliminated for violating the 'Parameter Not Inlineable' condition decreases.

¿From the results, we conclude that two of the primary conditions preventing object inlining are object aliasing and objects being passed to methods whose source code is unavailable. The array alias condition is also important because although only a small number of arrays are eliminated from inlining, each array consists of many potentially inlineable objects. The next section will discuss the object and array aliasing conditions and the techniques employed to increase the number of inlined aliased objects and arrays of objects.

2 Improving Performance by Inlining Aliased Objects and Arrays

Aliasing [22] occurs when there is more than one reference to a memory location. One of the ways an alias can be created is through a reference assignment. As the result of aliasing, modifying a value at a memory location that has multiple references to it causes the value to be seen as modified through all references. Precisely determining what aliases exist in an application is a difficult task [13, 22]. For example, an application with multiple execution paths may have a different set of aliases depending upon which path is taken.

Filename (# of files)	Binary Method	Object Aliased	Array Element Aliased	Parameter Not Inlineable	Array Aliased	Return Not Inlineable	Return Object
JQuantity (79)	44	85	1	2	4	31	17
Jmol (167)	88	134	13	2	14	87	15
JAMA (9)	1	3	0	0	0	3	0
Jsky (363)	301	354	18	8	34	162	27
Jcckit (78)	20	41	0	0	0	6	0
Biomer (51)	19	157	23	0	27	9	0
JMAT (78)	7	46	0	0	1	16	2
Ostermiller (43)	6	3	1	0	2	6	0
SSH tools (427)	114	219	1	0	4	140	8
JGraph (42)	77	77	3	0	0	8	5

Table 3

Object inlining elimination results with one safety condition at a time enabled. The number of objects eliminated from inlining decreased for the Parameter Not Inlineable safety condition because it relied on other violated safety conditions to eliminate an object first.

Aliasing is a problem for object inlining. When inlining an object, compiler creates inlined fields that serve as a placeholder for the data that was contained within an object that doesn't exist anymore. These inlined fields can only be referenced directly, and consequently, it is not possible to have more than one reference to an inlined object. Thus, object inlining may produce incorrect results if it inlines aliased objects. The overly conservative approach of the previous object inlining work solves this problem by eliminating all aliased objects and object arrays from inlining [3]. Although eliminating aliased objects ensures code correctness, we miss opportunities to improve runtime performance when aliased objects appear on the execution path, which occurs in practice. Our work addresses this problem by implementing an object aliasing strategy that uses copy folding [4]. This strategy eliminates as many assignments as possible to produce fewer aliased objects.

Aliased arrays are a special case. They can be inlined if none of the arrays aliased to one another violate any other safety condition. Even though arrays of objects are aliased, we can still inline them, because the Java implementation of arrays as objects gives us an additional level of indirection. Even though references cannot be shared to inlined individual objects, references *can* be shared to inlined arrays. Every array is a Java object, even arrays of primitive data, so the reference to that object can be shared. We implemented an array alias strategy that uses a set-based union-find [28] algorithm with rank and path compression to model aliased arrays and to determine a set of aliased arrays that can all be inlined.

2.1 Overview of Object and Array Inlining in the JaMake Compiler

Object inlining is an optimization performed after compiler analysis determines that a given object or array of objects is safe to inline [9]. An object is safe to inline if the object does not violate any of the safety conditions detailed in the previous section. Our object inlining transformation replaces an inlineable object with inlined fields and an inlineable array with arrays of inlined fields. The transformation creates a synthetic method when encountering an inlineable object or array passed to a method call. It then replaces the original method call with a call to the synthetic method. *Figure 2* shows an example of a synthetic method being created to pass an inlined array to it. The next section will discuss aliasing in Java and its effect on object inlining in more detail.

```
// original version
class Bar {
  // main
  void main(String[] args){
     Point[] p2Array;
     p2Array = new Point[5];
     p2Array[0] = new Point();
     // original call
     foo(p2Array);

  }
  // original method
  void foo(Point[] pArray){
     int i = 0;
     int j =  pArray[i].x;
  }
}
class Point {
   int x, y;
   public Point(){x = 8; y = 8;}
 }
```

```
// inlined version
class Bar {
   void main(String[] args){
     // inlined array
     int[] p2Array_x, p2Array_y;
     p2_x = new int[5];
     p2_y = new int[5];
     p2_x[0] = 8; p2_y[0] = 8;
     // synthesized method call
     foo_Bar_A_Pt_X(p2Array_x, p2Array_y);
   }
  // synthetic method
  void foo_Bar_A_Pt_X(int[] p_x,int[] p_y){
     int i = 0;
     int j = p_x[i];
  }
}
```

Fig. 2. *Example of creating a synthetic method for method foo so that an inlined version of array p2Array can be passed to the method call.*

2.2 Object Aliasing in Java

There are several ways object aliasing can occur. Assignment of one reference to a different reference is one way to introduce aliasing. Both references now point to the same memory location. As a result, when modifying an aliased object's field, the value is seen as updated through both references. In contrast, when an inlined field of an aliased object is modified, the value is seen only as updated through one reference (the direct reference to the inlined data).

Inlining an aliased object can potentially produce incorrect code. Before object inlining, changing a field of an aliased object may or may not affect other objects, depending on the execution path. However, after object inlining, because each reference has its own set of inlined data, changing the object's field will *not* affect any other objects aliased to it.

Fortunately, this does not mean that every reference copy in the original code must automatically remove the involved objects from consideration for inlining, as it was done before [3]. If the compiler can prove that two references *always* point to the same physical object, or *never* point to the same physical object, object inlining can still be done.

2.3 Object Alias Strategy

Our object alias strategy uses copy folding to reduce the number of object aliases and to increase the number of objects that can be inlined. Copy folding is a code transformation designed to remove variable assignments in an application [4]. The object alias algorithm uses a simpler copy folding algorithm by removing only assignments where the left-hand-side is a local variable. This restriction ensures that the scope of the left-hand-side object cannot live longer than the scope of the right-hand-side object. One of the benefits of copy folding is that it can create more opportunities for other compile-time optimizations such as object inlining. To be able to eliminate a copy, one must prove that it is safe to eliminate the assignment. It is safe to eliminate the assignment $a = b$ if the assignment dominates every subsequent use of a. An assignment dominates all subsequent uses if the assignment is executed on all paths leading to each use [4, 15].

The object alias strategy has 2 passes. The first pass analyzes the application's source code and determines which objects, defined in assignments, have all their subsequent uses dominated by every definition of the object. This requirement ensures that it is possible to determine the exact

```
// original version
class Foo {
  // original method
  public static void bar(){
     goo();
  }
  // original method
  public static void goo(){
     Bar b1 = new Bar();
     Bar b2 = new Bar();
     // copy can be folded
     b1 = b2;
     // b1.x will become b2.x
     b1.x = 6;

     Bar b3 = new Bar();
     Bar b4 = new Bar();
     // assume value of b1.x
     // cannot be determined
     if (b1.x > 5){
        // cannot fold copy
        b3 = b4;
     }
     b3.x = 5;
  }
}
class Bar(){
  int x;
  public Bar(){x = 4;}
}
```

```
// copy fold version
class Foo {
// method stays the same
public static void bar() {
   goo();
}
// copy gets folded
public static void goo(){
   // b1,b2 can be inlined
   int b1_x;
   int b2_x;
   // inlined Bar constructor
   b1_x = 4;
   b2_x = 4;
   // copy was folded
   ;
   // b1.x becomes b2_x
   b2_x = 6;
   // cannot inline b3,b4
   Bar b3 = new Bar();
   Bar b4 = new Bar();
   // b3.x stays
   if (b2_x > 5){
      // copy still exists
      b3 = b4;
   }
   b3.x = 5;
 }
}
```

Fig. 3. *Example of when copies can and cannot be folded.*

definition of all the subsequent uses of removed assignments at compile-time.

The second pass of the object alias algorithm transforms the code. Upon encountering an assignment, if the defined variable is in the list of variables whose assignment can be removed, the algorithm removes the assignment. Then, upon encountering a variable use, the object alias algorithm replaces the use with the most recently assigned value. *Figure 3* shows an example of a case when a copy can be folded and a case when the copy cannot be folded. *Figure 4* provides pseudo code for the object alias algorithm.

2.4 Array Alias Strategy

In previous object inlining work, no aliased objects or arrays have been inlined [3]. The problem with inlining an aliased object is that once the object is inlined, the inherent relationship the object shared with its aliases is lost. Their references no longer point to the same memory location. As a result, when redefinition of an inlined object field occurs, the value is seen as updated through only one inlined object.

Arrays are different, however, because they have an additional level of indirection. To acces a field of an object that is a part of an array of objects, the processor must dereference the array reference first, then index into the array, then dereference the object reference. After object inlining, the object references disappear, but the array references are still there, and it is possible to use them to enable the same behaviour as before object inlining, even for aliased arrays.

For an array to be inlineable, all its aliases have to be inlineable as well. The compiler strategy we are using to ensure this is to create equivalence classes of inlineable arrays. All the arrays in the same class are either all inlineable, or are all not inlineable. We use union-find algorithm with path

```
Object Alias Analysis:
for all block b ∈ basic blocks do
  add basic block id bid to open scope list
  for all statement s ∈ b do
    if s instanceof assignment then
      add current block id to lhs object o defs list
      add lhs object o to copy fold list
    end if
    if s instanceof identifier then
      for all block id bid ∈ def list of s do
        if bid ∉ open scopes list then
          add s to preserve assignments list
        end if
      end for
    end if
  end for
  remove b from open scope list
end for
```

```
Object Alias Transformation:
for all block b ∈ basic blocks do
  add basic block id bid to open scope list
  for all statement s ∈ b do
    if s instanceof assignment then
      if lhs object o ∉ preserve assignment list then
        add rhs to value list in current scope
        remove assignment
      end if
    end if
    if s instanceof identifier then
      if s ∉ preserve assignment list then
        rid = most recent s value in open scope list
        replace s with rid
      end if
    end if
  end for
  remove b from open scope list
end for
```

Fig. 4. *Pseudo code for the object alias strategy to eliminate aliases by removing assignments.*

compression and rank to implement this strategy [28].

A set will consist of members where each member is an array of objects. Each member of a set is aliased to every other member in the set. A union of sets occurs when a member of one set becomes aliased to a member of a different set. As a result, the sets are unioned together to form one set using the ranks of the sets.

Each set has an "inlineable" flag initially set to true. If object inline analysis prohibits any member in the set from being inlined, object inline analysis marks the inlineable flag for that set as false. Consequently, object inlining cannot occur for any other member in the set. *Figure 5* gives pseudo code for the array alias strategy.

3 Improving Performance by Inlining Binary Method Arguments

Binary methods are methods whose source code is unavailable at the compile time, and thus the compiler is unable to perform any analysis or optimizaiton on them. Because there is no guarantee to what the binary method might do with its arguments, the compiler must assume a conservative strategy and prevent inlining of any object that is being passed as an argument to a binary method. Section 1.2 shows that in practice many objects do get eliminated from inlining in such a way. This

```
Array Alias Strategy:
for all method m ∈ reachable method list do
  for all statement s ∈ m do
    if s instanceof array assignment then
      lset = findSet(lhs)
      rset = findSet(rhs)
      if lhs and rhs ∈ inlineable set then
        union lset and rset
      else
        lset and rset inlineable flag is false
      end if
    end if
    if s instanceof array var definition then
      sset = createSet(s)
      if init ∈ variable set then
        initset = findSet(init)
        union sset and initset
      end if
    end if
  end for
end for
```

Fig. 5. *Pseudo code for the array alias strategy to create aliased array sets. Members of an alias array set are all aliases of one another.*

section introduces a strategy that allows some of these objects to be inlined.

3.1 Preliminary Results of Prominent Binary Methods Prohibiting Object-Inlining

We expect to improve runtime performance by enabling inlining of some of the objects passed to binary methods. *Figure 6* illustrates a reason why an object passed to a binary method cannot be inlined. In performing this study, we were able to determine which particular binary methods were, in practice, preventing objects from being inlined. Using the same application suite, we counted the frequency with which each binary method eliminated an object from inlining.

```
class Foo {
   Matrix mat;
   public void init() {
       LinkedList list1 = new LinkedList();
       v1.add(mat);// mat is passed to a binary method
       mat.m = 8;  // multiple references to mat could exist, can't inline it
   }
}
class Matrix {
   int m, n; double A;
}
```

Fig. 6. *Object being passed to a binary method*

The results from the study show that it's the standard Java utility methods often prevent objects from being inlined. In particular, there were several Java utility container classes that invoked common methods which performed similar functions on the container. These container classes included List, LinkedList, ArrayList, Stack, and Vector. The common functions invoked by these classes included adding, retrieving, removing or copying objects from the container. Because

the semantic behavior of these methods is well-defined, object inlining could be allowed to inline objects passed to these methods. Procedure specialization [12, 25] can be used to enable inlining of objects passed to these well-known methods .

In addition to instance methods of binary containers, there are two additional standard Java binary methods preventing objects from being inlined. The first binary method is the *System.arraycopy* method. We transform the *System.arraycopy* method into the semantically equivalent *JaMakeContainer.arraycopy* with the same argument types. Object inlining then creates a specialized library method version based on the concrete types passed to the original call. As a result, arrays passed to *arraycopy* can now be inlined.

The second binary method is *PrintStream.println(o)*. We transform this method call into the semantically equivalent *PrintStream.println(o.toString())*. As a result of the transformations, these two binary methods no longer eliminate objects from being inlined.

```
//original
class Foo {

   // main
   public void main(){
      Bar b1;
      bar b1 = new Bar();
      Vector v1 = new Vector();
      v1.addElement(b1);
  }
}

class Bar {
    int x;

    // constructor
    public Bar(){
       x = 5;
    }
}
```

```
// JaMake library
class JaMakeContainer {
    Object[] _elems, _resA;
    int _size, _num_elems;
    // constructor
    public JaMakeContainer() {
      _size = 20, _num_elems = 0;
      _elems = new Object[20];
    }
    // add object
    public void addObj(Object o){
       if(_num_elems==_size)
          resize();
       _elems[_num_elems++] = o;
    }
    // resize list
    public void resize(){
       _size *= 2;
       _resA = new Object[_size];
       for (int i=0;i<_elems.length;i++){
          _resA[i] = _elems[i];
       }
       _elems = _resA;
    }
}
```

Fig. 7. *Example of what the application and JaMake library look like before the binary method transformation.*

3.2 Binary Method Transformation Strategy

We enable object inlining of objects passed to binary methods by implementing a transformation that converts the binary method calls into semantically equivalent method calls whose source code is available to the JaMake compiler. These semantically equivalent methods perform the same function as the original binary methods while ensuring that the object passed to them does not violate any safety condition inside the method's body. As a result, the object may now be inlined. Furthermore, because the source code of the method call replacing the binary method is available to our JaMake compiler and its method body code size is small, inlining the method call can further improve the runtime performance of the application.

The JaMake compiler has a list of all the binary methods that can be replaced by equivalent methods. The set of these equivalent methods resides in the JaMake library. They are all instance methods of the JaMake container except the static method *arraycopy*. No object passed to any

of these methods will violate a safety condition in the body of the method. All the tranformable binary methods perform a function on a container that holds a collection of objects. They either add, retrieve, copy, or remove an object from a collection of objects. The next section will go into detail on how our technique transforms binary methods into JaMake library methods.

```
// original
class Foo {

   // main
   public void main(){
      Bar b1;
      init_Bar(b1);
      JaMakeContainer jm1;
      init_JM(jm1);
      addObj_JM_Bar(b1,jm1);
   }
 }
class Bar {
    int x;

    // constructor
    public Bar(){
       x = 5;
    }
}
```

```
// JaMake library
class JaMakeContainer {
   Object[] _elems, _resA;
   int _size, _num_elems;
   // constructor
   public void init_JM(JM j) {
      j._size = 20;
      j._num_elems = 0;
      j._elems = new Object[20];
   }
   // add object
   public void addObj_JM_Bar(Bar b,JM j){
      if(j._num_elems==j._size)
         resize_JM(j);
      j._elems[j._num_elems++] = o;
   }
   // resize list
   public void resize_JM(JM j){
      j._size *= 2;
      j._resA = new Object[j._size];
      for (int i=0;i<j._elems.length;i++){
         j._resA[i] = j._elems[i];
      }
      j._elems = j._resA;
   }
}
```

Fig. 8. *Application and JaMake library after specialized library methods creation. JM is an abbreviation for the JaMakeContainer class type. The JaMake library method calls and the arguments to those methods will all be inlined.*

3.3 Transforming Binary Methods

The transformation of a binary method to a specialized library method occurs in two passes. The first pass is the binary method invocation analysis pass. The binary method invocation analysis pass determines which binary containers can be translated into the *JaMakeContainer* class. Analysis marks binary containers that cannot safely be translated as "contaminated". The second pass performs the actual transformation. The second pass converts all eligible binary methods, whose binary containers have not been marked "contaminated", into semantically equivalent JaMake library methods. *Figures 7* and *8* show part of the binary method transformation process. *Figures 9* and *10* gives pseudo code for the binary method transformation strategy.

4 Performance Results

The experimental results were obtained on an 798 MHz Athlon AMD(tm) processor with 384 MB of RAM. *Table 4* shows the runtime performance results of some of the applications in the test suite. The performance gain is measured against conservative object inlining(OI). Some of the applications were omitted because there were no opportunities for the aliasing and binary method strategies to improve performance. The object alias strategy was responsible for performance gains of up to 20%. *Table 5* shows that aliasing strategies reduce the number of objects and arrays prohibited from inlining for violating an alias safety restriction.

```
Binary Method Analysis, First Pass:
for all method m do
  for all statement s ∈ m do
    if s ∈ replaceable binary method call set then
      create BinaryContainer bc for target variable v of s
    end if
  end for
end for
Second Pass:
for all method m do
  for all statement s ∈ m do
    if s instanceof replaceable binary method call then
      if target variable v of s has BinaryContainer bc and ∃ arg of s ∉ inlineable set then
        mark bc as contaminated
      end if
    end if
    if s instanceof assignment then
      if lhs object o modified after placed in a BinaryContainer bc then
        mark bc as contaminated
      end if
    end if
  end for
end for
for all BinaryContainers contambc ∈ contaminated list do
  for all object o ∈ contambc do
    for all BinaryContainers cleanbc ∉ contaminated list with member o do
      place cleanbc in contaminated list
    end for
  end for
end for
```

FIG. 9. *Pseudo code for the binary method analysis. Binary method analysis determines which binary methods can be transformed into JaMake libary methods*

```
Library Method Insertion:
for all BinaryContainers cleanbc ∉ contaminated list do
  map JaMakeContainer jc to cleanbc
end for
for all method m do
  for all statement s ∈ m do
    if s instanceof replaceable binary method call then
      replace statement s with equivalent library method lm
      replace target t with mapped JaMakeContainer jc
    end if
    if s instanceof var definition and maps to JamakeContainer jc then
      replace var definition s with var definition jc
    end if
  end for
end for
```

FIG. 10. *Pseudo code for the libary method insertion pass of binary method transformation.*

There were only a few cases where arrays were eliminated from inlining due to aliasing. The array alias strategy did enable inlining of a few aliased arrays. The array alias strategy is important even if it only eliminates a small number of arrays from aliasing because each array consists of many potentially inlineable objects.

Table 6 shows that for some scientific applications, the binary method strategy actually increases

the number of objects and arrays prohibited from inlining for violating the binary method safety restriction. This results from the binary method transformation enabling the inlining of objects passed to binary methods. In addition, objects eliminated because of a secondary or dependent condition may become temporarily inlineable. However, these objects can become prohibited again from inlining for violating another condition such as the binary method safety condition. This is what happens in the *JQuantity* and *Approximation* applications.

Table 7 shows that the aliasing and binary method strategies can reduce the total number of objects and arrays prohibited from inlining. For the *Ssh KeyGen* application, the total number of objects prohibited from inlining decreases the most when object inlining uses only the aliasing strategies. This is due to an increase in number of objects being prohibited from inlining to break cycles of must inline methods calls to properly simulate pass-by-reference of inlineable local objects in a pass-by-value environment.

Binary method transformation is a promising approach because it is reducing the number of objects being eliminated from inlining. Because many of the objects in our application suite are still being eliminated for other reasons, binary method transformation alone is not showing significant performance improvements. We expect the binary method transformation to become more effective once constraints on other conditions for allowing object inlining are relaxed.

To illustrate this point, we have performed an experiment by hand on a code snipped from the *Ssh KeyGen* application. *Figure 11* shows a loop referencing an object that we inline by hand to illustrate the potential benefit of the binary method transformation. We have modified the loop to include only statements referencing objects whose source code is available, to prevent statements that perform functions such as writing text to the console from dominating the execution time inside the loop. As a result of inlining objects inside the loop by hand, we reduce the execution time of the loop in *figure 11* by 91.5%. The reduction in execution time is mainly attributed to the conversion of a vector add method to a JaMake library method tuned for performance. This is an encouraging result that suggests that further refinement of our techniques may result in significant performance improvements.

Applications	*Application Runtime Performance in ms*				*Total*
	Original	*Conservative OI*	*OA,AA*	*OA,AA,BM*	*Gain*
Ssh KeyGen	2493	2494	2483	2474	0.8%
Mapped Matrix	200	200	200	200	0%
JQuantity	2284	2293	1893	1892	21.2%
Sparse Matrix	481	471	480	480	-1.9%
Approximation	441	440	430	430	2.3%

Table 4

Application runtime performance with object aliasing (OA), array aliasing (AA) and binary method (BM) transformation strategies.

Applications	*Optimizations*		
	Conservative OI	*OA*	*OA,AA*
Ssh KeyGen	93	91	91
Mapped Matrix	1	1	1
JQuantity	19	8	7
Sparse Matrix	1	1	1
Approximation	20	9	8

Table 5

Shows the number of objects and arrays prohibited from inlining for violating an alias safety restriction. The number of prohibited objects and arrays decreases in some cases with the addition of the object and array alias strategies.

Applications	*Optimizations*		
	Conservative OI	*OA,AA*	*OA,AA,BM*
Ssh KeyGen	69	69	69
Mapped Matrix	1	1	1
JQuantity	4	4	5
Sparse Matrix	1	1	1
Approximation	5	5	9

Table 6

Shows the number of objects and arrays prohibited from inlining for violating the binary method safety restriction. The number of prohibited objects and arrays actually increases because objects and arrays that once violated another safety restriction now violate the binary method restriction first.

Applications	*Total Number of Objects Prohibited from Inlining*		
	Total non-inlineable objects with conservative OI	*Total non-inlineable objects with alias and binary strategies*	*Total number of objects saved*
Ssh KeyGen	1035	1025	10
Mapped Matrix	11	11	0
JQuantity	317	307	10
Sparse Matrix	21	21	0
Approximation	325	315	10

Table 7

Shows the total number of objects and arrays prohibited from inlining for violating a safety restriction. The total number of prohibited objects and arrays decreases in some cases with the addition of the aliasing and binary method strategies.

```
// loop appearing inside the Ssh KeyGen application
for (int i=0;i<children.length;i++){
   log.info("Extension " + children[i].getAbsolutePath()
                    + " being added to classpath");
   ext.add(children[i];
}
```

Fig. 11. *An actual loop in the Ssh KeyGen application. All JaMake compiled objects referenced inside the loop are inlined by hand. Prior to optimizing the loop by hand, the loop was modified to include only statements referencing objects whose source code is available. In this loop, the object ext is inlined.*

5 Related Work

The creation of specialized JaMake libraries to replace binary methods is similar to Procedure specialization in Telescoping Languages [12]. The compiler selects the best variant to replace the called procedures. As a result of employing the variants, performance increases. Procedure cloning [1, 14] is related to the specialized methods created during the object inlining process. Object inlining may create a specialized method to provide actual inlineable argument information to the callee. Predicate dispatch [26] for object-oriented language reduces the costs of dynamic dispatch by determining the precise method invocation at a call site. Predicate dispatch utilizes guards to specify the conditions under which a method should be invoked.

Object inlining [7, 3, 18, 19] seeks to improve application performance by reducing the strain on the garbage collector in deallocating objects on the heap. Object combining [29] seeks to join together objects that have relatively the same life span. This technique joins objects together by appending fields from one object to another. As a result, multiple objects are (de)allocated at the

same time, thereby reducing the strain on memory management. Static write barrier removal [30] is work geared toward improving generational garbage collection. Generally, the strategy these garbage collectors employ is to remove objects from younger generations. These collectors assume that older objects remain live. Write barriers restrict younger generation object removal when there may be a reference to the object from an older generation. Static write barrier analysis can detect superfluous write barriers and remove them.

Alias analysis [4, 13, 16] and alias removal [13] enable compiler optimizations such as object inlining to be more effective. Aliasing often prohibits objects from being inlined. Precise alias analysis can limit the impact aliasing has object inlining. Work has also been done on arrays of objects and their inlining [3] to reduce object allocation costs and to enable better memory management.

6 Conclusions and Future Work

The techniques described in this paper produce runtime performance gains of up to 20% over conservative object inlining, eliminating object aliases being the most effective so far. The procedure specialization enables runtime peformance improvement by creating specialized library methods to replace binary method invocations [12, 25]. One of the benefits of this work is that by relaxing the safety conditions prohibiting object inlining, one can construct a more precise definition to characterize when an object could not be inlined. That is, an object cannot be inlined if it is truly polymorphic or the object escapes object inline analysis so that the compiler cannot determine what happens to the reference to the object.

We expect concrete type analysis [25] to enable even more objects to be inlined. We are also investigating further relaxation of the constraints on the aliasing and binary method strategies to improve their effectiveness. In addition, we will investigate the possible heuristics in deciding *when* to inline an object. Right now, every object inlining is considered beneficial, which may not be the case, as there is a small overhead assotiated with inlining an object, that can outweight the benefits in some cases.

Compilation and optimization of high-level, high-productivity languages for high-performance applications is a difficult task. Many features found in such languages disable or greatly limit the compiler's ability to generate fast code. This paper builds on the already remarkable advancements in compiler technology and presents several techniques that further improve the effectiveness of such high level compiler optimizations. It is a stepping stone to further compiler developments that will one day enable unrestricted high-level high-productivity programming for high-performance application, without any loss in performance when compared to traditional languages.

References

[1] A. Ayers, R. Gottlieb, R. Schooler. Aggressive Inlining. *In Proceedings of the 1997 ACM Sigplan Conference on Programming Language Design and Implementation*, pp. 134-145, 1997.

[2] C. Barton, P. Zhao, R. Niewiadomski, J. Amaral. Identifying Opportunities for Automatic Remote Field Cloning. *In Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pp. 124-134, 2004.

[3] Z. Budimlić. *Compiling Java for High Performance and the Internet*. PhD thesis, Rice University, 2001.

[4] Z. Budimlić, K. Cooper, T. Harvey, K. Kennedy, T. Oberg, and S. Reeves. Fast Copy Coalescing and Live-Range Identification. *In Proceedings of the 2002 ACM Sigplan Conference on Programming Language Design and Implementation*, Berlin, Germany, pp. 25-32.

[5] Z. Budimlić and K. Kennedy. Almost-Whole-Program Compilation. *In Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, WA, November 3, 2002, pp 104-111.

[6] Z. Budimlić and K. Kennedy. JaMake: A Java Compiler Environment. *In 3rd International Conference on Large Scale Scientific Computing*, pp. 201-209, 2001.

[7] Z. Budimlić and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445-463, June 1997

[8] Z. Budimlić and K. Kennedy. Prospects for Scientific Computing in Polymorphic, Object-Oriented Style. *In the Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999.

[9] Z. Budimlić and K. Kennedy. Static Interprocedural Optimizations in Java. *Technical Report CRPC-TR98746*, Rice University, December 1998.

[10] C. Chambers, W. Chen. Efficient Multiple and Predicate Dispatching. *ACM SIGPLAN Notices , In Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 238-255, 1999.

[11] C. Chambers, D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *In Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, pp. 150-164, 1990.

[12] A. Chauhan, K. Kennedy. Optimizing Strategies for Telescoping Languages: Procedure Strength Reduction and Procedure Vectorization. *In Proceedings of the 15th international conference on Supercomputing*. pp. 92-101, 2001.

[13] K. Cooper. Analyzing Aliases of Reference Formal Parameters. *In Proceedings of the 12th Symposium on the Principles of Programming Languages*, pp. 281-290, 1985.

[14] K. Cooper, M. Hall, K. Kennedy. Procedure Cloning. *In Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, 1992.

[15] K. Cooper, T. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. *Software - Practice and Experience*.

[16] K. Cooper, K. Kennedy. Fast Interprocedural Alias Analysis. *Proceedings of the 16th Symposium on Principles of Programming Languages*, pp. 49-59, 1989.

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

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

[19] J. Dolby, A. Chien. An Automatic Object Inlining Optimization and its Evaluation. *In Proceedings of the 2000 ACM Sigplan Conference on Programming Language Design and Implementation*, pp. 345-357, 2000.

[20] S. Ghemawat, K. Randall, D. Scales. Field Analysis: Getting Useful and Low-cost Interprocedural Information. *In Proceedings of the 2000 ACM Sigplan Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada.

[21] M. Joyner. Improving Object Inlining for High Performance Java Scientific Applications. *Master's Thesis*, Rice University, 2005.

[22] W. Landi and B. Ryder. Pointer-induced Aliasing: A Problem Classification. *Proceedings of the 18th Symposium on Principles of Programming Languages*, pp. 93-103, 1991.

[23] X. Leroy. Unboxed objects and polymorphic typing. *In Proceedings of the 19th Symposium on the Principles of Programming Languages*, pp. 177-188, 1992.

[24] O. Lhotak, L. Hendren. Run-time Evaluation of Opportunities for Object Inlining in Java. *In Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 175-184, 2002.

[25] C. McCosh, E. Allen, K. Kennedy, W. Taha. Static Type Inference for Specialization in a Telescoping Compiler. *Technical Report TR04-439*, Rice University, 2004.

[26] T. Millstein. Practical Predicate Dispatch. *In Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 345-364, 2004.

[27] M. Naik, R. Kumar. Efficient Message Dispatch in Object-Oriented Systems. *ACM SIGPLAN Notices*, 35(3):49-58, March 2000.

[28] R. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215-225, April 1975.

[29] R. Veldema, C. Jacobs, R. Hofman, H. Bal. Object Combining: A New Aggressive Optimization for Object Intensive Programs. *In Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 165-174, 2002.

[30] K. Zee, M. Rinard. Write Barrier Removal by Static Analysis. *In Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 191-210, 2002.