RICE UNIVERSITY

Improving Object Inlining for High-Performance Java Scientific Applications

by

Mackale Joyner

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
MASTER OF SCIENCE

APPROVED, THESIS COMMITTEE:

Ken Kennedy, Co-Chair
John and Ann Doerr Professor of
Computer Science

Zoran Budimlic, Co-Chair
Research Scientist

Keith Cooper
Professor of Computer Science,
Department Chair

Richard Tapia
Noah G. Harding Professor of
Computational and Applied Mathematics

Houston, Texas
April, 2005
ABSTRACT

Improving Object Inlining for High-Performance Java Scientific Applications

by

Mackale Joyner

Java is a popular programming language that enables many developers to achieve high productivity. Previous work in Java improved runtime performance by using object inlining. This thesis extends prior object inlining work by both analyzing the code and performing optimizations to further improve application runtime performance. Two impediments to object inlining and to increased runtime performance are object and array aliasing and binary method invocations. This thesis 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. The improvements made to the compile-time object inlining optimization should increase the scientific community's acceptance of the Java programming language in the development of high-performance scientific applications by decreasing the performance
gap between Java and accepted languages such as C and Fortran.
Acknowledgments

I would first like to thank God for giving me the strength and wisdom to complete this thesis. I would also like to thank my advisors for their guidance and advice as well as my thesis committee for their invaluable feedback. I am grateful for the many people that have contributed to this project including Rui Zhang and Jim Van Fleet. Last, but not least, I would like to thank my wife, Jamil, for the support, balance, and unique perspective she brings to my life.
# Contents

Abstract .................................................. ii

Acknowledgments .......................................... iv

List of Figures ............................................. vii

List of Tables .............................................. ix

1 Introduction .............................................. 1

2 Motivation ................................................ 3

2.1 Object Inlining ......................................... 3

2.2 Preliminary Object Inlining Study .................. 5

2.2.1 Scientific applications ............................. 5

2.2.2 Safety Conditions Preventing Object Inlining .. 6

2.2.3 Preliminary Study Results ......................... 8

3 Improving Performance by Inlining Aliased Objects and Arrays .... 14

3.1 Overview of Object and Array Inlining in the JaMake Compiler ... 15

3.2 Object and Array Aliasing in Java ................... 22

3.3 Object Alias Algorithm ............................... 24

3.4 Array Alias Algorithm ............................... 30

3.5 Performance Results ................................... 34
4 Improving Performance by Inlining Binary Method Arguments 38

4.1 Preliminary Results of Prominent Binary Methods Prohibiting Object-Inlining 39

4.2 Preliminary Results Discussion 42

4.3 Binary Method Call Transformation Algorithm 44

4.4 Binary Method Transformation Requirements 45

4.5 Transforming Binary Methods 47

4.6 Performance Results 55

5 Related Work 60

6 Conclusions and Future Work 62

References 66
List of Figures

2.1 Object inlining example showing instance field myFoo being inlined. . 4

3.1 Example of object p1 being inlined. . . . . . . . . . . . . . . . . . . 17

3.2 Example of creating a synthetic method for method foo so that an
inlined version of object p2 can be passed to the method call. . . . 18

3.3 Example of object p1Array being inlined. . . . . . . . . . . . . . 20

3.4 Example of creating a synthetic method for method foo so that an
inlined version of array p2Array can be passed to the method call. . 21

3.5 Incorrect result of inlining an aliased object. . . . . . . . . . . . 23

3.6 Example of when copies can and cannot be folded. . . . . . . . . . 27

3.7 Pseudo code for the first pass of the object alias strategy, object alias
analysis. This pass determines which copies can be folded. . . . . . 28

3.8 Pseudo code for the second pass of the object alias strategy, the object
alias transformation. This pass folds all copies that the analysis allows. 29

3.9 This example shows the difference between inlining aliased arrays and
aliased objects. In the case of the objects, only inlined g1 gets the
update to x while both inlined arrays g1Array and g2Array see their
value updated. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.10 Pseudo code for the array alias algorithm to create aliased array sets.

Members of an alias array set are all aliases of one another. When an array cannot be inlined due to violating a safety restriction, object inline analysis marks the inlineable flag for its alias array set as false. As a result, object inlining will not inline an array in the set.

4.1 Object being passed to a binary method

4.2 Example of what the application and JaMake library look like before the binary method transformation.

4.3 Example of what the application and JaMake library look like after the binary method transformation.

4.4 Example of what the application and JaMake library look like after specialized library methods have been created. Although the original library methods still exist, they are excluded from the example. JM is used as an abbreviation for the JaMakeContainer class type.

4.5 Example of what the application looks like after object inlining.

4.6 Pseudo code for the first pass of binary method transformation, binary method analysis.

4.7 Pseudo code for the second pass of binary method transformation, library method insertion.
# List of Tables

2.1 The table includes the number of files, methods, and inlineable objects for each scientific application. It also shows the percentage of objects that did not violate any safety condition. ............................... 7

2.2 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. ........................................... 9

2.3 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. ......... 11

2.4 This table shows the total number of times each safety condition was violated when no condition actually eliminated an object from being inlined. ................................................................. 12

3.1 Application runtime performance in ms with object and array alias algorithms. ................................................................. 36
3.2 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. ................................. 36

3.3 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 object and array alias strategies. ................................. 37

4.1 Top 10 binary methods eliminating objects from inlining for each scientific application. ................................................................. 41

4.2 Application runtime performance with aliasing and binary method transformation algorithms. ................................................................. 58

4.3 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. ................................................................. 58
4.4 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.
Chapter 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 slow runtime performance compared to languages such as C and Fortran. Improvements in Java translation have made the language more widely accepted [3,18,19,20,29].

This thesis describes optimizations added to the JaMake compiler [6], a compiler environment that uses the front and back-ends of Sun’s prototype compiler for Java Development Kit 1.5. This thesis discusses improvements 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 [18]. Studies have shown object inlining to be a viable optimization in improving scientific application performance [6,7,23]. The idea of object inlining dates back to boxing and unboxing in languages with polymorphic typing such as ML [22]. 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 objects within containers [18]. Budimlic's object inlining transformation implemented for Java inlines global objects as well as local objects found in methods [3]. The transformation also inlines arrays of objects which are important for scientific high-performance applications. This thesis discusses improvements to object inlining that in some cases yield performance improvements of up to 20%.
Chapter 2
Motivation

2.1 Object Inlining

Object inlining \([3,19]\) is an optimization that improves an application's runtime 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. Inlining an object removes the object's declaration. It replaces the object with a collection of variables in which each variable represents an instance field of the removed object. The collection of variables represents the removed object. Object inlining replaces a subsequent use of the removed object with a use of the collection of variables. Object inlining replaces a subsequent modification of the removed object's field with an assignment to the inlined field variable. As a result, the transformation eliminates indirect object-field accesses and inserts direct accesses using inlined field variables, thereby increasing runtime performance. Figure 2.1 shows an inlining example.

Object inlining studies have shown that object inlining can increase the runtime performance of an application \([6,7,23]\). In the object inlining research performed by Budimlic, every object that did not violate Budimlic's object inline safety conditions was inlined. Budimlic's object inline safety conditions ensure that the object inlining transformation does not change the semantic behavior of the application. This thesis
class Foo {
    int x;
    double y;
}

class Bar {
    Foo myFoo;

    /*** get x field ***/
    public int getX() {
        return myFoo.x;
    }

    /*** get y field ***/
    public double getY() {
        return myFoo.y;
    }
}

/*** original version *****/
class Foo {
    int x;
    double y;
}

class Bar {
    Foo myFoo;

    /*** get x field ***/
    public int getX() {
        return myFoo.x;
    }

    /*** get y field ***/
    public double getY() {
        return myFoo.y;
    }
}

Figure 2.1 Object inlining example showing instance field myFoo being inlined.

extends Budimlic'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, I enable more objects to be inlined, thereby improving runtime performance while preserving the observed behavior of the original application. By implementing optimizations to transform the application's source code, I reduce the total number of objects prohibited from inlining. As a result, the application in general runs faster. The following section discusses the methodology used to determine which safety conditions eliminate the most objects
from inlining, a necessary first step in extending Budimlic’s work.

2.2 Preliminary Object Inlining Study

I performed a preliminary object inlining study to understand which object inlining conditions prohibit the most objects from being inlined. Recall that these conditions are necessary to prevent the object inlining transformation from producing code that could be exploited to yield incorrect results. I conducted the study using 10 open-source Java scientific applications. The goal of the study is to identify the prominent object inlining conditions prohibiting inlining of objects so that techniques can be developed to enable inlining of some of these objects. As a result, I expect better application runtime performance.

2.2.1 Scientific applications

I collected 10 open-source Java scientific applications from a repository of open-source applications to determine the safety conditions eliminating the most objects from inlining. All of the applications were written in Java. Table 2.1 includes the number of source files, methods, and inlineable objects for each scientific application. The table also shows the percentage of inlineable objects that did not violate an inline safety condition. These objects can be inlined. 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. JMAT also provides utilities to create plots and histograms.

• Ostermiller - Libraries for working with scientific numbers.

• SSH tools - A suite of Java ssh applications.

• Jgraph - A powerful, lightweight graph component.

2.2.2 Safety Conditions Preventing Object Inlining

There are several object inlining conditions that object inline analysis uses to determine when an object cannot be inlined such as an object being passed to a
<table>
<thead>
<tr>
<th>Filename</th>
<th>Files</th>
<th>Methods</th>
<th>Initially Inlinable Objects</th>
<th>Inlineable Objects</th>
<th>% Inlineable</th>
</tr>
</thead>
<tbody>
<tr>
<td>JQuantity</td>
<td>118</td>
<td>1374</td>
<td>648</td>
<td>39</td>
<td>6.0%</td>
</tr>
<tr>
<td>Jmol</td>
<td>167</td>
<td>2303</td>
<td>656</td>
<td>105</td>
<td>16.0%</td>
</tr>
<tr>
<td>JAMA</td>
<td>9</td>
<td>118</td>
<td>169</td>
<td>165</td>
<td>97.6%</td>
</tr>
<tr>
<td>Jsky</td>
<td>363</td>
<td>5505</td>
<td>2061</td>
<td>363</td>
<td>17.6%</td>
</tr>
<tr>
<td>Jckit</td>
<td>78</td>
<td>371</td>
<td>352</td>
<td>150</td>
<td>42.6%</td>
</tr>
<tr>
<td>Biomer</td>
<td>51</td>
<td>641</td>
<td>440</td>
<td>181</td>
<td>41.1%</td>
</tr>
<tr>
<td>JMAT</td>
<td>78</td>
<td>1042</td>
<td>548</td>
<td>90</td>
<td>16.4%</td>
</tr>
<tr>
<td>Ostermiller</td>
<td>43</td>
<td>540</td>
<td>118</td>
<td>101</td>
<td>85.6%</td>
</tr>
<tr>
<td>SSH tools</td>
<td>427</td>
<td>3812</td>
<td>1389</td>
<td>197</td>
<td>13.8%</td>
</tr>
<tr>
<td>JGraph</td>
<td>42</td>
<td>922</td>
<td>560</td>
<td>105</td>
<td>18.8%</td>
</tr>
</tbody>
</table>

Table 2.1  The table includes the number of files, methods, and inlinable objects for each scientific application. It also shows the percentage of objects that did not violate any safety condition.

method whose source code is unavailable or an assignment of two objects. The results of the study conducted show which conditions eliminate the most objects from inlining using the suite of scientific applications. My results revealed that in practice, only 7 safety conditions are violated. The rationale for why some of these safety conditions exist to ensure correct code will be visited in greater detail in the next chapter. The 7 safety conditions are:

1. An object was passed to a method whose source code was unavailable. As a result, the object 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 inlinable.

5. An array of objects was aliased to a non-inlinable array.

6. An object or array of objects was assigned to the result of a method call whose return object was not inlinable.

7. An object was a return object in a method whose result is not inlinable.

2.2.3 Preliminary Study Results

Using the 10 open-source scientific applications introduced in section 2.1, I 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. The safety conditions are not separable. Disabling one condition can affect the number of objects counted as violating a different condition. There are two ways this can occur. First, an object can violate more than one condition. In this case, the number of objects violating a particular condition may increase when turning off other safety conditions since the objects no longer violate the disabled conditions.
Second, there are some conditions that depend on other safety conditions being violated first before they can prohibit inlining of objects. Consequently, disabling a condition in this case can cause a reduction in the number of objects another safety condition prohibits from inlining. As a result, it can be hard to get exact numbers for all conditions. However, the estimated numbers still provide insight on the prominent safety conditions prohibiting inlining of objects. Table 2.2 shows how many objects each safety condition prohibited from inlining because it was the first condition violated by the object.

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

**Table 2.2** 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.
These results show how many objects were eliminated per safety condition. However, these results do not show the relationship some safety conditions have with other safety conditions, thereby being slightly misleading. Some of the safety conditions are dependent on others. These conditions depend on other conditions to be violated first before they can prohibit an object from being inlined. For example, an object can be eliminated from inlining if it was assigned to the result of a method invocation that was not inlinable. The reason the method call’s return object was not inlinable could have been because it was passed to a method whose source code was unavailable, thereby violating the binary method safety condition. As a result, safety conditions may appear to be more important than they actually are. 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 2.3 shows that the number of objects eliminated for violating a dependent condition decreases. The number of objects eliminated from inlining for violating the parameter not inlinable safety condition decreased because it relied on other safety conditions to be first violated before it eliminated an object. The return not inlinable condition is also a dependent condition. The reason its numbers only slightly drop is because when a method returns a non-inlinable object, the statistics increment the return not inlinable count.

In addition to showing a decrease in the number of objects eliminated from inlining
<table>
<thead>
<tr>
<th>Filename (# of files)</th>
<th>Binary Method</th>
<th>Object Aliased</th>
<th>Array Element Aliased</th>
<th>Parameter Not Inlineable</th>
<th>Array Not Aliased</th>
<th>Return Not Inlineable</th>
<th>Return Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>JQuantity (79)</td>
<td>44</td>
<td>85</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>31</td>
<td>17</td>
</tr>
<tr>
<td>Jmol (167)</td>
<td>88</td>
<td>134</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>JAMA (9)</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jsky (363)</td>
<td>301</td>
<td>354</td>
<td>18</td>
<td>8</td>
<td>34</td>
<td>162</td>
<td>27</td>
</tr>
<tr>
<td>Jcckit (78)</td>
<td>20</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Biomer (51)</td>
<td>19</td>
<td>157</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JMAT (78)</td>
<td>7</td>
<td>46</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Ostermiller (43)</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>SSH tools (427)</td>
<td>114</td>
<td>219</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>140</td>
<td>8</td>
</tr>
<tr>
<td>JGraph (42)</td>
<td>77</td>
<td>77</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.3  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.
by dependent safety conditions, Table 2.3 also shows an increase in the number of objects eliminated by non-dependent conditions. The reason for the increased numbers by these safety conditions is due to the safety conditions eliminating objects that previously violated another safety condition first. Table 2.4 shows the total number of times each safety condition was violated when no safety condition prevented an object from being inlined.

<table>
<thead>
<tr>
<th>Filename (# of files)</th>
<th>Binary Method</th>
<th>Object Aliased</th>
<th>Array Element Aliased</th>
<th>Parameter Not Inlineable</th>
<th>Array Not Inlineable</th>
<th>Return Not Inlineable</th>
<th>Return Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>JQuantity (79)</td>
<td>121</td>
<td>114</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>Jmol (167)</td>
<td>106</td>
<td>148</td>
<td>17</td>
<td>3</td>
<td>20</td>
<td>72</td>
<td>15</td>
</tr>
<tr>
<td>JAMA (9)</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jsky (363)</td>
<td>470</td>
<td>393</td>
<td>37</td>
<td>12</td>
<td>44</td>
<td>180</td>
<td>27</td>
</tr>
<tr>
<td>Jcckit (78)</td>
<td>20</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Biomer (51)</td>
<td>36</td>
<td>254</td>
<td>142</td>
<td>0</td>
<td>40</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>JMAT (78)</td>
<td>8</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>Ostermiller (43)</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>SSH tools (427)</td>
<td>155</td>
<td>233</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>127</td>
<td>8</td>
</tr>
<tr>
<td>JGraph (42)</td>
<td>107</td>
<td>88</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 2.4** This table shows the total number of times each safety condition was violated when no condition actually eliminated an object from being inlined.
From the results, I 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 inlinable objects. The number of objects that violate the parameter not inlinable safety condition will decrease as I reduce the number of objects violating other safety conditions. The next chapter will discuss the object and array aliasing conditions and the techniques employed to increase the number of inlined aliased objects and arrays of objects.
Chapter 3
Improving Performance by Inlining Aliased Objects and Arrays

Aliasing [21] occurs when there is more than one reference to a memory location. One of the ways an alias can be created is through an 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 [13,21]. For example, an application with multiple execution paths may have a different set of aliases depending upon which path is taken.

Aliasing is a problem for object inlining. When inlining an object, inlined fields are created such that only one reference sees the inline field value change when the inline field is redefined. However, when modifying an aliased object’s field, multiple references see the object’s new field value. Consequently, object inlining may produce incorrect results if it inlines aliased objects. As a result, the object inlining transformation avoids producing incorrect code by eliminating all aliased objects and object arrays from inlining. 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. To address the problem, I implemented the copy folding optimization [4]. Copy folding eliminates as many
assignments as possible to produce fewer aliased objects.

Previously, object inlining eliminated all aliased arrays from inline consideration [3]. However, aliased arrays can be inlined if none of the arrays aliased to one another violate any other safety condition. I implemented the set-based union-find [27] algorithm with rank and path compression to model aliased arrays and to determine when a set of aliased arrays could all be inlined. I provide details of the two optimizations after defining the aliasing problem with respect to Java. The first section in this chapter gives an overview of how objects and arrays were previously inlined in the JaMake compiler [6]. The second section discusses the aliasing problem with respect to the Java programming language. The third section details how aliasing prohibits array inlining and object inlining in the JaMake compiler. The remaining sections in this chapter detail the object and array alias strategies that I implemented to allow more objects and arrays of objects to be inlined. Performance results conclude this chapter.

3.1 Overview of Object and Array Inlining in the JaMake Compiler

Object inlining is an optimization performed in a series of steps after object inlining analysis determines that a given object 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 preliminary object inlining study in chapter 2. The first step in object inlining is to
remove the object variable from the application. For each field of the removed object, a variable representing that field is created. The type of this newly created variable is the same type as the type of the field that it represents. Subsequently, wherever the removed object’s field is accessed, that access is replaced with an equivalent access to the field that now represents it. Figure 3.1 illustrates an object being inlined.

There are two steps that occur when the removed object is passed as an actual argument to a method call. The first step is to create a semantically equivalent method to replace the method call. This synthetic method becomes a member of the method call’s owner class. The synthetic method’s formal parameter types match the types of the variables representing the removed object’s field types. As a result, object inlining passes the original object’s inlined fields to a method instead of the original object. The second step is to replace the original method call in the source code with a method call to the new synthetic method. Object inlining passes the new variables representing the inlined fields of the removed object as actual arguments to the new synthetic method. Figure 3.2 shows the result of passing an inlineable object to a method invocation.
/***** original version *****/

class MatrixA {
    Point p1;

    /** constructor ***/
    public MatrixA() {
        p1 = new Point();
    }

    /** main ***/
    public static void main(String[] args){
        System.out.println("start");
    }
}

/***** inlined version *****/

class MatrixA {
    /** inlined point ***/
    int p1_x;
    int p1_y;

    /** constructor ***/
    public MatrixA() {
        /** inlined point constructor call***/
        p1_x = 8;
        p1_y = 8;
    }

    /** main ***/
    public static void main(String[] args){
        System.out.println("start");
    }
}

// Figure 3.1  Example of object p1 being inlined.
class Point {
    int x;
    int y;
    public Point() {
        x = 8;
        y = 8;
    }
}

class Bar {

    public static void main(String[] args) {
        Point p2 = new Point();

        /// pass p2 to method call ***/
        foo(p2);
    }

    public static void foo(Point myPt) {
        int i = 5 + myPt.x;
    }
}

class Bar {

    public static void main(String[] args) {
        /// inlined point ***/
        int p2_x;
        int p2_y;
        /// inlined point constructor call ***/
        p2_x = 8;
        p2_y = 8;
        /// created synthetic method call ***/
        foo_Bar_L_Point(p2_x, p2_y);
    }

    public static void foo(Point myPt) {
        int i = 5 + myPt.x;
    }

    /// synthetic method ***/
    public static void foo_Bar_L_Point(int p_x, int p_y) {
        int i = 5 + p_x;
    }
}

Figure 3.2 Example of creating a synthetic method for method foo so that an inlined version of object p2 can be passed to the method call.
Object inlining uses similar steps to inline an array of objects. To inline an object, object inlining replaces an object with a group of variables representing the removed object’s fields. However, when inlining an array of objects, object inlining removes the declared array of objects and replaces the individual array with a group of arrays. Each array in the group represents an inlined field array.

Object inlining still creates a synthetic method when the removed array is passed to a method. Object inlining again replaces the original method call with a synthetic method call. The group of newly declared arrays representing the inlined field arrays of the removed array become the actual arguments passed to the synthetic method. *Figure 3.3* illustrates the inlining of an object array and *Figure 3.4* illustrates the inlining of an object array passed as an argument to a method. The next section will discuss object aliasing and array aliasing in Java and why it prevents object and array inlining.
class MatrixB {
    Point[] p1Array;

    /** constructor ****/
    public MatrixB() {
        p1Array = new Point[5];
    }

    /** main ***/
    public static void main(String[] args) {
        System.out.println("start");
    }
}

class Point {

    /** constructor ***/
    public Point() {
        x = 8;
        y = 8;
    }
}

class MatrixB{
    int[] p1Array_x;
    int[] p1Array_y;

    /** constructor ***/
    public MatrixB() {
        /** inlined array***/
        p1Array_x = new int[5];
        p1Array_y = new int[5];
    }

    /** main ***/
    public static void main(String[] args) {
        System.out.println("start");
    }
}

Figure 3.3  Example of object p1Array being inlined.
```java
class Bar {

    public static void main(String[] args) {
        Point[] p2Array = new Point[5];

        // pass p2Array to method call
        foo(p2Array);
    }

    public static void foo(Point[] pArray, int i) {
        int i = 5 + pArray[i].x;
    }
}

class Point {
    int x;
    int y;
}
```

```java
class Bar {

    public static void main(String[] args) {
        int[] p2Array_x;
        int[] p2Array_y;
        int[] p2.Team_1:
        int[] p2.Team_2:
        int[] p2.Team_3:
        int[] p2.Team_4:
        int[] p2.Team_5:

        // created synthetic method
        foo_Bar_A_Pt_X_int(p2Array_x, p2Array_y);
    }

    public static void foo(Point pArray, int i) {
        int i = 5 + pArray[i].x;
    }

    public static void foo_Bar_A_Pt_X_int(int[] pA_x, int[] pA_y, int i) {
        int i = 5 + pA_x[i];
    }
}
```

**Figure 3.4** Example of creating a synthetic method for method foo so that an inlined version of array p2Array can be passed to the method call.
3.2 Object and Array Aliasing in Java

There are several ways object or array 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 declaring a primitive type variable, another reference to the memory location cannot exist. Primitive types in Java include boolean, double, float, int, char, long, and short. Consequently, no aliasing occurs when assigning two primitive type variables to one another. As a result, redefining the value of a declared primitive type variable will not modify any other variable. Consequently, if inlining of an aliased object occurs followed by an object field modification, object inlining produces incorrect code due to not updating the fields of the aliases. Figure 3.5 shows an example of this case. Because it can be unsafe to inline aliased objects, the JaMake compiler did not inline any aliased objects. Using the copy folding algorithm, I was able to inline some aliased objects by removing the assignments in which the objects were involved. As a result, those objects were no longer aliased.
class Matrix {
    Point p1;
    Point p2;

    constructor {
        p1 = new Point();
        p2 = new Point();
    }

    prints out 5 ***/
    public void computePoint() {
        p1 = p2;
        p1.x = 5;
        System.out.print(p2.x);
    }
}

class Point {
    int x;
    int y;

    constructor {
        x = 8;
        y = 8;
    }
}

//**** inlined version ****/

class Matrix {
    int p1_x, p1_y, p2_x, p2_y;

    constructor {
        p1_x = 8;
        p1_y = 8;
        p2_x = 8;
        p2_y = 8;
    }

    incorrectly prints out 8 ***/
    public void computePoint() {
        p1_x = p2_x;
        p1_y = p2_y;
        p1_x = 5;
        System.out.print(p2_x);
    }
}

Figure 3.5 Incorrect result of inlining an aliased object.
In addition to not inlining aliased objects, the JaMake compiler did not inline aliased arrays of objects. However, arrays leave an additional level of indirection that allows us to handle aliasing. Object inlining handles array inlining by replacing the original array with newly created field arrays. Modifications to the original aliased arrays of objects can be properly simulated with the new inlined array versions. As a result, as long as all the original aliased arrays are inlineable, they can all be safely inlined. Using the array alias strategy, I was able to determine when a group of arrays that were all aliases of one another were all inlineable. The next sections give descriptions of both the object and array alias strategies implemented to inline aliased objects and arrays. Performance results conclude the chapter.

3.3 Object Alias Algorithm

The object alias algorithm 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 elimi-
nate 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].

I implemented an object alias strategy that 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 definition of all the subsequent uses of removed assignments at compile-time. An application's source code is broken into basic blocks. Each basic block is straightline code that has no control-flow [17]. Each basic block is defined as a scope and tagged with a block id.

Using the top-down approach by starting from the entry point into the basic block, each reachable line of code is traversed. The analysis creates a block id for each basic block and places the id on a stack of currently open scopes. When a new block is encountered, such as in the body a for statement, the object alias algorithm again creates a block id and places the block id on the stack of open scopes. Upon completion of the block, the algorithm pops the block id off the stack of open scopes.

When encountering an assignment, the object alias algorithm adds the current block id to the defined variable's definition list using the unique variable id. Then, when subsequently encountering a variable use, the algorithm obtains the variable's definition list. The next step is to make a check to determine if each block in the
variable's definition list is currently an open scope. If one of the block ids is not on the open scope stack, the definition in that block does not dominate the use. Consequently, there are multiple definitions that can reach the use. As a result, the variable is placed on a list that prevents any of its definitions from being removed. However, if all the block ids are on the open scope stack, the variable is placed on a list that allows its definitions to be removed.

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. Again, the object alias algorithm maintains a list of currently open scopes. The algorithm adds each assignment to the end of the most recently created open scope, called the current scope. The object alias algorithm has previously added the current scope to the end of an open scopes list. Then, upon encountering a variable use, the algorithm traverses the open scopes list starting from the end to determine the most recently defined variable value. The most recently assigned value replaces the use. Figure 3.6 shows an example of a case when a copy can be folded and a case when the copy cannot be folded. Figure 3.7 and figure 3.8 provide pseudo code for the object alias algorithm. The next section gives a description of the array alias algorithm.
class Foo {

    public static void bar() {
        goo();
    }

    public static void goo() {
        Bar b1 = new Bar();
        Bar b2 = new Bar();
        b1 = b2;
        b1.x = 6;

        Bar b3 = new Bar();
        Bar b4 = new Bar();
        if (b1.x > 5) {
            b3 = b4;
        }
        b3.x = 5;
    }
}

class Bar {
    int x;

    public Bar() {
        x = 4;
    }
}

Figure 3.6   Example of when copies can and cannot be folded.
Object Alias Analysis:
  for each basic block b
    add basic block id to list of open scopes
  for each statement s in b
    switch s
      case assignment:
        add current block id to defs list for lhs object o
        add lhs object o to copy fold list
      end case

      case identifier:
        for each def block id in def list of s
          if block id in open scope list
            def dominates s use
          else
            def does not dominate s use
            add s to preserve assignments list
            break for
          end if
        end for
      end case
    end switch
  end for
  remove basic block id from list of open scopes
end for

Figure 3.7  Pseudo code for the first pass of the object alias strategy, object alias analysis. This pass determines which copies can be folded.
Object Alias Transformation:
  for each basic block b
    add b id to open scope list
    for each statement s in b
      switch s
        case assignment:
          if lhs object is not in preserve assignment list
            add rhs to value list in current scope
            remove assignment
          end if
        end case

        case identifier:
          if s not in preserve assignment list
            replace_id = most recent s value in open scope list
            replace s with replace_id
          end if
        end case
      end switch
    end for
    remove b from open scope list
  end for

Figure 3.8  Pseudo code for the second pass of the object alias strategy, the object alias transformation. This pass folds all copies that the analysis allows.
3.4 Array Alias Algorithm

In previous object inlining work, Budimlic prohibited all aliased objects and arrays from being 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. However, because arrays have an additional level of indirection with respect to object access and object inlining maintains the additional level of indirection when inlining the array, aliased arrays shouldn’t be immediately prohibited from inlining. In fact, if an array and all its aliases are inlineable, then all the arrays can be inlined. The implemented union-find algorithm with path compression and rank is well suited to represent this case [27].

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. Every array of objects is a member of some set. If the
set contains only one member, the array has not been aliased and can be inlined. In this case, the set’s rank will have a value of 1 indicating that the array of objects has not been aliased. Figure 3.9 shows an example of the difference between inlining an object and an array. Figure 3.10 gives pseudo code for the array alias strategy.
/***** original version *****/
class Foo {

    /*** main ***/ 
    public void main() {
        Goo g1;
        Goo g2;
        g1 = new Goo();
        /*** aliased objects ***/ 
        g2 = g1;
        g1.x = 9;
        Goo[] g1Array;
        Goo[] g2Array;
        g1Array = new Goo[5];
        /*** aliased arrays ***/ 
        g2Array = g1Array;
        g1Array[0].x = 9;
    }
}

class Goo {

    int x;

    /*** constructor ***/ 
    public Goo(){
        int x = 5;
    }
}

/***** inlined version *****/
class Foo {

    /*** main ***/ 
    public void main(){
        int g1_x;
        int g2_x;
        /*** inlined constructor ***/ 
        g1_x = 5;
        /*** inlined object alias ***/ 
        g2_x = g1_x;
        g1_x = 9;
        int[] g1Array_x;
        int[] g2Array_x;
        g1Array_x = new int[5];
        /*** inlined array alias ***/ 
        g2Array_x = g1Array_x;
        g1Array_x[0] = 9;
    }
}

Figure 3.9 This example shows the difference between inlining aliased arrays and aliased objects. In the case of the objects, only inlined g1 gets the update to x while both inlined arrays g1Array and g2Array see their value updated.
Array Alias Algorithm:
for each reachable method m
  for each statement s in m
    switch s
      case assignment:
        if array assignment
          l_set = find lhs set
          r_set = find rhs set
          if rhs and lhs are inlineable
            union l_set and r_set
          end if
        else
          r_set.inlineable = false;
          l_set.inlineable = false;
        end if
      end case
      case var definition:
        if s is an array
          s_set = create s set
          if (init is a variable)
            init_set = find init set
            union s_set and init_set
          end if
        end if
      end case
    end switch
  end for
end for

Figure 3.10  Pseudo code for the array alias algorithm to create aliased array sets. Members of an alias array set are all aliases of one another. When an array cannot be inlined due to violating a safety restriction, object inline analysis marks the inlineable flag for its alias array set as false. As a result, object inlining will not inline an array in the set.
3.5 Performance Results

The experimental results were obtained on an 798 MHz Athlon AMD(tm) processor with 384 MB of RAM. Table 3.1 shows the runtime performance results of some of the applications in the test suite. The performance gain is measured against object inlining (OI) without the new object (OA) and array (AA) alias strategies. A few of the applications were not included in the tables because there were no opportunities for the object and array alias algorithms to improve performance. Table 3.2 shows that for some scientific applications, the object and array alias strategies reduce the number of objects and arrays prohibited from inlining for violating an alias safety restriction. Table 3.3 shows that for some scientific applications, the object and array alias strategies reduce the total number of objects and arrays prohibited from inlining.

The results show that the object alias algorithm was responsible for performance gains of up to 20%. One case that the object alias algorithm improves occurs when a local variable is created with a precise type and assigned to a casted variable. This assignment was normally seen after the `instanceof` operation was called on the casted variable. This case occurred several times in the `JQuantity` application. The common case the object alias strategy does not catch is when global objects are assigned to formal parameters inside constructors. Almost all the aliases in `Ssh KeyGen` fell in this category. Those assignments cannot be folded. To catch this case, other constraints will have to be relaxed.
There were only a few cases where arrays were eliminated from inlining due to aliasing. The array alias algorithm did enable inlining of a few aliased arrays. The array alias algorithm is important even if it only eliminates a small number of arrays from aliasing because each array consists of many potentially inlineable objects.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Application Runtime Performance in ms</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>2493</td>
<td>2494</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>JQuantity</td>
<td>2284</td>
<td>2293</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>481</td>
<td>471</td>
</tr>
<tr>
<td>Approximation</td>
<td>441</td>
<td>440</td>
</tr>
</tbody>
</table>

Table 3.1 Application runtime performance in ms with object and array alias algorithms.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>93</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>1</td>
</tr>
<tr>
<td>JQuantity</td>
<td>19</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>1</td>
</tr>
<tr>
<td>Approximation</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 3.2 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.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Total Number of Objects Prohibited from Inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total non-inlineable</td>
</tr>
<tr>
<td></td>
<td>objects with</td>
</tr>
<tr>
<td></td>
<td>conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>1035</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>11</td>
</tr>
<tr>
<td>JQuantity</td>
<td>317</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>21</td>
</tr>
<tr>
<td>Approximation</td>
<td>325</td>
</tr>
</tbody>
</table>

Table 3.3  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 object and array alias strategies.
Chapter 4
Improving Performance by Inlining Binary Method Arguments

Binary methods are methods whose source code is unavailable to the JaMake compiler. Binary methods present a problem to object inlining. When an application invokes a binary method and passes an object as an argument to that binary method, object inline analysis immediately eliminates the object from inlining. Since the invoked method’s source code is not available, object inlining cannot analyze the method to determine if the argument passed to the method violates an inline safety condition. For example, a reference to the object could be stored in the invoked method body, allowing subsequent modifications outside of analyzed code, preventing the object from being inlined. For the sake of safety, object inlining has to be conservative and assume that all objects passed as arguments to binary method invocations violate a safety condition, thereby prohibiting the objects from being inlined.

Studying the open-source application test suite study in the motivation chapter revealed that many objects are eliminated from inlining because they are passed to binary methods. This chapter introduces a strategy that allows some of these objects to be inlined. The sections in this chapter are as follows: section 4.1 discusses the prominent binary methods prohibiting object inlining, section 4.2 provides insight on the preliminary results, section 4.3 describes the strategy used to transform bi-
nary method invocations, section 4.4 discusses in detail the requirements necessary for binary method invocation transformation, section 4.5 details the transformation process with examples. Performance results conclude the chapter.

4.1 Preliminary Results of Prominent Binary Methods Prohibiting Object-Inlining

I expect to improve runtime performance by enabling inlining of some of the objects passed to binary methods. Figure 4.1 illustrates a reason why an object passed to a binary method cannot be inlined. In performing a preliminary study, I was able to determine which particular binary methods were, in practice, preventing objects from being inlined. Using the 10 open-source scientific applications, I counted the frequency with which each binary method eliminated an object from inlining. Table 4.1 shows, for each application, the top 10 binary methods eliminating objects from inlining. The binary methods were ranked according to the number of objects they eliminated from inlining.
import java.util.*;

class Foo {
    Matrix mat;

    public void init() {
        Vector v1 = new Vector();
        /** mat is passed to a binary method, *
         * eliminating it from being inlined
         ***/
        v1.addElement(mat);

        /** there could be multiple references to mat, *
         * we can't inline it
         ***/
        mat.m = 8;
    }
}

class Matrix {
    int m;
    int n;
    double[] A;
}

Figure 4.1 Object being passed to a binary method
<table>
<thead>
<tr>
<th>JQuantity</th>
<th>Jmol</th>
<th>Java</th>
<th>Jcckit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class.isInstance</td>
<td>Vector.addElement</td>
<td>Vector.addElement</td>
<td></td>
</tr>
<tr>
<td>PrintStream.println</td>
<td>AbstractButton.addActionListener</td>
<td>Vector.setElement</td>
<td></td>
</tr>
<tr>
<td>Stack.push</td>
<td>DefaultTreeModel.reload</td>
<td>Vector.removeElement</td>
<td></td>
</tr>
<tr>
<td>Object.equals</td>
<td>Window.addWindowListener</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashMap.put</td>
<td>Hashtable.put</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vector.copyInto</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.arraycopy</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SwingUtilities.invokeLater</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JSlider.addChangeListener</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JOptionPane.showMessageDialog</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jsky</th>
<th>Biomer</th>
<th>Jmat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container.add</td>
<td>System.arraycopy</td>
<td>Vector.add</td>
</tr>
<tr>
<td>Vector.add</td>
<td>Graphics.drawImage</td>
<td>Vector.remove</td>
</tr>
<tr>
<td>Printstream.println</td>
<td>Container.add</td>
<td>Component.addComponentListener</td>
</tr>
<tr>
<td>fireVetoableChange</td>
<td>setConstraints</td>
<td>showSaveDialog</td>
</tr>
<tr>
<td>Window.addWindowListener</td>
<td>ImageProducer.</td>
<td>Vector.copyInto</td>
</tr>
<tr>
<td></td>
<td>startProduction</td>
<td></td>
</tr>
<tr>
<td>JComponent.firePropertyChange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List.add</td>
<td></td>
<td>Container.add</td>
</tr>
<tr>
<td>JFileChooser.addChoosableFileFilter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashtable.put</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ostermiller</th>
<th>SSH tools</th>
<th>JGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashMap.put</td>
<td>List.remove</td>
<td>Map.put</td>
</tr>
<tr>
<td>Vector.add</td>
<td>Vector.add</td>
<td>DefaultMutableTreeNode.add</td>
</tr>
<tr>
<td>Map.put</td>
<td>ActionMap.put</td>
<td></td>
</tr>
<tr>
<td>Container.add</td>
<td>Vector.addElement</td>
<td>List.add</td>
</tr>
<tr>
<td>JOptionPane.showMessageDialog</td>
<td></td>
<td>List.add</td>
</tr>
<tr>
<td>HashMap.put</td>
<td>UndoableEditSupport.postEdit</td>
<td></td>
</tr>
<tr>
<td>List.add</td>
<td>Vector.add</td>
<td></td>
</tr>
<tr>
<td>SAXParser.parse</td>
<td>List.remove</td>
<td></td>
</tr>
<tr>
<td>AbstractButton.addActionListener</td>
<td>JComponent.firePropertyChange</td>
<td></td>
</tr>
<tr>
<td>AbstractAction.putValue</td>
<td>List.toArray</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1** Top 10 binary methods eliminating objects from inlining for each scientific application.
4.2 Preliminary Results Discussion

The results from the analysis show that often the standard Java utility methods 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 objects to the container, retrieving objects from the container, removing objects from the container, and copying objects from one container to another container. Because the semantic behavior of these methods is well-defined, object inlining should be able to inline objects passed to these methods. An idea from Telescoping Languages can be used to enable inlining of objects passed to these well-known methods [12,24].

The concept of procedure specialization [12] in Telescoping Languages revolves around the idea that by using conceptual and context information for a method invocation, one can create a semantically equivalent specialized version. The specialized method invocation will then replace the original method invocation to improve the runtime performance of the application. I create specialized versions of binary methods by first creating generic, semantically equivalent library methods that work only with generic objects. Then, when object inlining determines the precise element type passed to the binary containers via the well-known binary methods, object inlining creates a specialized version of the library method. Subsequent sections in this
chapter provide details of this process.

The binary method study also revealed other prominent binary methods preventing objects from being inlined. Most of them fall into the category of methods commonly used to build a graphical user interface such as \texttt{awt.Window.addWindowListener}, \texttt{awt.Graphics.drawImage}, and \texttt{swing.JOptionPane.showMessageDialog}. There were two reasons why I did not create semantically equivalent JaMake library classes for methods that I categorized as gui building methods.

The first reason is because it is not clear that an equivalent library method can be created for these gui methods with the criterion that its method body does not violate an inline safety condition, thereby still prohibiting inlining of the object passed as an argument.

Second, the library methods added to the library are short. This property is important because the specialized library methods are inlined. Again, it is not obvious if the library methods created to replace the gui binary methods would be small in code size. However, that is not to say library methods could not also be created for these methods. A performance heuristic would be ideal in determining if creating library methods for these gui methods would increase application performance. This topic is discussed more in the conclusions and future work chapter.

There were 2 additional standard Java binary methods preventing objects from being inlined that were not instance methods of binary containers.
The first binary method is the `System.arraycopy` method. I transform the `System.arraycopy` method into the semantically equivalent `JaMakeContainer.arraycopy` with the same argument types. Object inlining then will create 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)`. I transform this method call into the semantically equivalent `PrintStream.println(o.toString())`. As a result of the transformations, these 2 binary methods no longer eliminate objects from being inlined. The next section gives an overview of the binary container algorithm implemented to enable inlining of some of the objects passed to the common Java utility container functions.

4.3 Binary Method Call Transformation Algorithm

I 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 functions as the original binary method calls 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 reside 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 transformable 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 the binary container requirements imposed to ensure safety when inlining all object members of a binary container.

4.4 Binary Method Transformation Requirements

Most of the binary methods that my transformation handles are instance methods of a binary container. A binary container is a Java utility class whose source code is unavailable to our compiler. The binary containers of these allowed binary methods include `ArrayList`, `Vector`, `Stack`, and `LinkedList`. The primary function of these containers is to store and retrieve objects. The following requirements are necessary to ensure that any element of these containers can be safely inlined:

- An object cannot be modified after it is placed in a binary container via an allowed binary method. Modifying an object after it has been added to the binary container creates an aliasing problem because there exists an object in a container that has been modified. The inlined object in the container may
not be properly updated when the reference to the object is later modified.

- A container must be homogenous with respect to its element type. This rule ensures that each inlined field array of the container has only one type.

- There cannot exist a non-inlineable object in the container. The underlying structure in the binary containers used to store objects is an array. In order to create an inlined version of the array, every element must be inlineable.

- The binary container cannot invoke a binary method that is not mapped to a JaMake library method. This requirement prevents objects from escaping the context of the container. Binary methods that aren't transformed could store a reference to an object element which creates an alias, thereby violating the aliasing safety condition.

- The binary container itself cannot be assigned to any other container and must be instantiated as an empty container. This requirement ensures that we can determine if any element of the binary container violated a safety condition.

- The binary container cannot be passed as a argument to any binary method. A reference could again be stored to an element of the container creating an alias which violates the aliasing safety condition.
4.5 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 a JaMakeContainer. This first pass creates a BinaryContainer object for each declared variable that invokes a binary method that can be translated into a JaMake library method. The pass stores the declared variable id in this BinaryContainer object. If the variable violates any of the requirements in section 4.2, the binary method analysis pass marks the BinaryContainer object representing that variable as "contaminated". As a result, object inlining cannot inline any object in the variable's object collection.

The second pass performs the actual transformation. The second pass begins by creating a mapping for every "non-contaminated" binary container to a JaMake container. This pass replaces every occurrence of a "non-contaminated" binary container variable with a variable having the JaMake container class type, including variable declarations. The transformation also replaces every occurrence of a binary method invoked by a "non-contaminated" binary container with a JaMake library method invoked by a variable with the JaMake container class type. Figures 4.2-4.5 show the transformation process starting from the binary method transformation through object inlining. Figures 4.6 and 4.7 give pseudo code for the binary method transfor-
mation algorithm. Performance results conclude the chapter.
Figure 4.2 Example of what the application and JaMake library class look like before the binary method transformation.
Figure 4.3 Example of what the application and JaMake library class look like after the binary method transformation.
Figure 4.4  Example of what the application and JaMake library look like after specialized library methods have been created. Although the original library methods still exist, they are excluded from the example. JM is used as an abbreviation for the JaMakeContainer class type.
**** inline version ****
class Foo {

    /** main ***/
    public void main(){
        /** inline Bar ***/
        b1_x;
        /** inline constructor ***/
        b1_x = 5;
        /** inline specialized jamake container ***/
        int[] jml elems_x;
        int[] jml_resA_x;
        int jml_size;
        int jml_num elems;
        /** inline init method ***/
        jml_size = 20;
        jml_num elems = 0;
        jml elems_x = new int[20];
        /** inline specialized library addObj method ***/
        if (jml_num elems == jml_size){
            /** inline specialized library resize method ***/
            jml_size *=2;
            jml_resA_x = new int[jml_size];
            int i = 0;
            for(i<jml elems_x.length;i++){
                jml_resA_x[i] = jml elems_x[i];
            }
            jml elems_x = jml_resA_x;
        }
        jml elems_x[jml_num elems++] = b1_x;
    }
}

class Bar {
    int x;

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

Figure 4.5  Example of what the application looks like after object inlining.
Binary Method Analysis:
first pass:
   for each method body mb
      for each statement s in mb
         if s is a method call
            if s is a replaceable binary method
               create BinaryContainer for target var v of s
            end if
         end if
      end for
   end for
second pass:
   for each method body mb
      for each statement s in mb
         switch s
            case method call:
               if s is a replaceable binary method and
                  target var v of s has a BinaryContainer
                  if all object arguments equal inlineable
                     BinaryContainer is clean
                  else
                     BinaryContainer is contaminated
               end if
            end if
         end case
         case assignment:
            if object o modified after placed in a BinaryContainer
               BinaryContainer is contaminated
            end if
         end case
      end switch
   end for
for each contaminated BinaryContainer c
   for each object o in c
      for each clean BinaryContainer d with member o
         d is contaminated
         place d on contaminated BinaryContainer list
      end for
   end for

Figure 4.6 Pseudo code for the first pass of binary method transformation, binary method analysis.
Library Method Insertion:
for each clean BinaryContainer c
  map a JaMakeContainer jm to c
end for

for each method body mb
  for each statement s in mb
    switch s
      case var definition:
        if s maps to a JaMakeContainer j
          replace var def s with var def j
        end if
      end case
      case method call:
        if s is a replaceable binary method
          replace s with equivalent library method lm
          if target t
            replace target t with mapped JaMakeContainer j
          end if
        end if
      end case
    end switch
  end for
end for

Figure 4.7 Pseudo code for the second pass of binary method transformation, library method insertion.
4.6 Performance Results

The experimental results were again obtained on an 798 MHz Athlon AMD(tm) processor with 384 MB of RAM. Once again, some of the applications compiled with the JaMake compiler were left out of the result table because there were no opportunities for the binary method transformation. Table 4.4 shows the runtime performance gains when compared to object inlining(OI) with the object alias(OA), array alias(AA), and binary method(BM) tranformation algorithms. Once again, the object alias algorithm was mostly responsible for the performance gains over conservative object inlining. This result was expected in every application except for Ssh KeyGen. The binary method transformation algorithm was not able to transform as many binary methods as expected in this application due to the many constraints that must hold for the binary container not to become "contaminated". These constraints must be relaxed as many of the transformable binary methods appear to be on the execution path of a couple of the applications in the result table. This explains why there was only a small speedup in the Ssh KeyGen application when the binary method transformation was turned on.

Table 4.5 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 addi-
tion, 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 4.6 shows that for scientific applications, the aliasing and binary method strategies reduce the total number of objects and arrays prohibited from inlining. However, 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 number of objects being prohibited from inlining to break must inline method cycles. A method must be inlined when passed an inlineable local variable that the method modifies in its body. Inlining the method ensures that we properly simulate pass-by-reference in a pass-by-value environment. In the case of Ssh KeyGen, the binary method strategy creates a different set of must inline method cycles resulting in an increase number of prohibited objects to break the cycles.

Despite these results, the binary method transformation work is still promising because it is reducing the number of objects being eliminated from inlining. However, it is not showing performance improvements yet because many of those objects are still being eliminated for other reasons. I expect the binary method transformation to become more effective once constraints on the strategy are relaxed. Improvements in other areas such as type analysis are expected to improve the effectiveness of this
strategy as well.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Application Runtime Performance in ms</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>2493</td>
<td>2494</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>JQuantity</td>
<td>2284</td>
<td>2293</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>481</td>
<td>471</td>
</tr>
<tr>
<td>Approximation</td>
<td>441</td>
<td>440</td>
</tr>
</tbody>
</table>

**Table 4.2** Application runtime performance with aliasing and binary method transformation algorithms.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>69</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>1</td>
</tr>
<tr>
<td>JQuantity</td>
<td>4</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>1</td>
</tr>
<tr>
<td>Approximation</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 4.3** 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.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Total Number of Objects Prohibited from Inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total non-inlineable</td>
</tr>
<tr>
<td></td>
<td>objects with conservative OI</td>
</tr>
<tr>
<td>Ssh KeyGen</td>
<td>1035</td>
</tr>
<tr>
<td>Mapped Matrix</td>
<td>11</td>
</tr>
<tr>
<td>JQuantity</td>
<td>317</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>21</td>
</tr>
<tr>
<td>Approximation</td>
<td>325</td>
</tr>
</tbody>
</table>

Table 4.4  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.
Chapter 5
Related Work

Procedure specialization in Telescoping Languages [12] is similar to the JaMake libraries the binary method transformation algorithm uses to replace some binary methods. The idea of procedure specialization is to produce specialized variants of library routines. The compiler selects the best variant to replace the called procedures [12]. As a result of employing the variants, performance increases. The binary method transformation replaces method calls with specialized library variants at compile time. However, unlike procedure specialization, the specialized JaMake library variants are not precompiled. 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. Similarly, in procedure cloning, a call site may be cloned if the callee can benefit from knowing about its actual arguments [1].

Object inlining [18,19] and object combining [28] seek to improve application performance by reducing the strain on the garbage collector in deallocating objects on the heap. Work being done on reordering objects improves performance by canonicalizing the code to allow garbage collection to more efficiently deallocate objects [29]. Predicate dispatching uses predicate guards for method implementations to re-
duce the costs associated with dynamic dispatch [10, 25]. Alias analysis [4, 13, 16] and alias removal [13] enable compiler optimizations such as object inlining to be more effective. Work has also been done on arrays of objects and their inlining [3, 19, 28] to reduce object allocation costs and to enable better memory management.
Chapter 6
Conclusions and Future Work

The techniques described in this thesis produce runtime performance gains of up to 20% over conservative object inlining. The object alias algorithm decreased the number of aliased objects increasing the total number of objects that can be inlined. The array alias algorithm had fewer opportunities in the test suite to enable inlining of previously prohibited aliased arrays. However, the fact that the array alias strategy enabled the inlining of only a couple of arrays in the test suite doesn’t diminish the importance of the algorithm. Inlining a couple of arrays could potentially lead to the inlining of many objects.

An idea from Telescoping Languages enabled runtime performance improvement by creating specialized library methods to replace binary method invocations [12,24]. These JaMake library methods were specialized in two ways. First, the semantically equivalent methods were created so that not only would their source code be available, but also the library method body would not cause an object to violate a safety condition, thereby preventing it from being inlined. Second, a specialized library method, derived from a general library method with general argument types, was created to enable object inlining of arguments. The library methods were created for heavily used Java utility methods. A byproduct of creating the specialized methods is
that these methods could then be inlined. The JaMake library methods replaced Java utility methods whose semantic behavior was not only well understood, but whose method body could be made small.

One of the benefits of this research 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 one cannot determine if a reference to the object is stored.

In the near future, I expect type analysis to be implemented. Type analysis would enable even more objects to be inlined. Many objects are currently not inlined because they are either passed as arguments to an abstract method, assigned the result of an abstract method, or are the targets of an abstract method call. Without type analysis [11], one cannot determine the concrete method invocation until runtime. As a result, object inlining prohibits any object falling into one of these three cases from inlining.

Also, because object inline analysis cannot determine which method is invoked at compile time, object inline analysis is conservative and assumes all possible concrete methods defining the abstract method are reachable from an entry point in the program. Some of these concrete methods are not invoked at runtime, although, they may eliminate objects from inlining because the objects violate safety conditions in the method body. Type analysis can prevent some non-reachable methods from
eliminating objects from inlining.

Currently, there is no performance benefit associated with inlining an object or a cost associated with the code transformation necessary to enable inlining of the object. Consequently, a costly code transformation could be employed to inline an object that has a small performance benefit, thereby degrading performance. An object inlining performance heuristic would assign positive and negative weights to each object. The heuristic would allow only inlining to be performed on objects with positive weights. Object inlining would alert the compiler writer when inlining was unable to inline objects with positive weights.

I plan to relax the constraints on the object alias strategy to improve its effectiveness and to reduce the number of objects prohibited from inlining for violating the object alias inline safety restriction. One of the limitations of the object alias strategy is that it only handles assignments to local objects. I will extend the strategy to also handle assignments to global objects. I also plan to extend the work on the binary method strategy. One way to accomplish this task will be to expand the number of binary methods that can be replaced with JaMake library methods. As a result, I will reduce the number of objects prohibited from inlining for violating the binary method inline safety restriction. I also plan to run experiments with scientific applications outside the test suite used to tune the strategies to validate the robustness of the alias and binary method strategies. I expect the result of extending these strategies
to be an increase in the number of inlineable objects and a decrease in application runtime.
References


