INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI
RICE UNIVERSITY

Compiling Java for High Performance and the Internet

by

Zoran Budimlić

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

[Signatures]

Dr. Ken Kennedy, Chair
Ann and John Doerr Professor in
Computational Engineering

[Signature]

Dr. Keith D. Cooper
Professor in Computer Science

[Signature]

Dr. George N. Phillips
Professor of Biochemistry and Cell Biology

Houston, Texas

January, 2001
To my parents Petar and Leposava, may they rest in peace.
Compiling Java for High Performance and the Internet

Zoran Budimlić

Abstract

Java is the first widely accepted language that addresses heterogeneous resources, security, and portability problems, making it attractive for scientific computation. It also encourages programmers to use object-oriented techniques in programming. Unfortunately, such object-oriented programs also incur unacceptable performance penalties. For example, using a polymorphic number hierarchy in a linear algebra package resulted in a code that is four times shorter, more extensible and less bug-prone than the equivalent Fortran-style code, but also many times slower.

To address the poor performance problem, this dissertation introduces several new compilation techniques that can improve the performance of scientific Java programs written in a polymorphic, object oriented style to within a factor of two of the equivalent hand-coded Fortran-style programs. These techniques also maintain an acceptable level of Java bytecode portability and flexibility, thus rewarding, rather than penalizing, good object-oriented programming practice.

This dissertation first discards the typical one-class-at-a-time Java compilation model for a whole-program model. It then introduces two novel whole-program optimizations, class specialization and object inlining, which improve the performance of high-level, object-oriented, scientific Java programs by up to two orders of magnitude, effectively eliminating the penalty of object-oriented design.

Next, this dissertation introduces a new Almost-whole-program compilation model. This model improves the flexibility of the generated code, while still permitting whole-program optimizations and incurring only modest performance penalties. It enables the
programmer balance performance and flexibility of the program after the development phase, instead of compromising the design for performance.

Furthermore, this dissertation reduces the restrictions that Java imposes upon classical optimization techniques by introducing exception hiding and SSA conversion algorithms. Exception hiding transforms the code to create exception-free zones, in which code motion transformations can move the code without restraint. The new, nearly linear-time SSA-to-CFG conversion algorithm considerably reduces the number of copies inserted in the conversion process, improving the effectiveness of classical optimizations.

Finally, this dissertation lays the groundwork for further research, particularly for fast register allocation, precise type analysis, coordinated compilation, and exception recovery.
Acknowledgments

This thesis is the culmination of many years of my personal and academic education. Many people have made significant contributions to this process and deserve my special thanks.

First and foremost, I am eternally indebted to my parents, Petar and Leposava Budimlić. They have sacrificed everything they had to provide resources they did not have to enable my education. They taught me the two most important things in life: honesty and hard work. Tragically, they did not live to see the outcome of their sacrifice. I know they are proud at this moment, wherever they are, and I dedicate this thesis to them.

I thank God for protecting me all these years and helping me to come back to the right track whenever I wandered off.

My wife Vesna was always there to support me, both through the good and the bad times. She endured and helped me survive the hard periods of my academic life. Without her constant and unconditional (and often undeserved) love I would never have completed this work, and for that I thank her candidly.

My sister Milanka, brother in law Milinko, and nephews Aca, Dule and Milica always had the words of encouragement from the other end of the globe. I thank them earnestly for their help and support throughout these years.

I also thank my advisor Ken Kennedy. His patience and wisdom guided me through the winding roads of academic research and thesis writing. He taught me how to conduct independent research and his invaluable advice helped me conclude the work on this thesis. I also thank my other committee members, Keith Cooper and George Phillips, for their patience, assistance and priceless input on my research and thesis.

Our undergraduate student programmers: Jeff Piper, Eliot Anshelevich, Amar Pai, Brian Webster, Nancy He, Ziquiang Zhou. Ashvin George and Ryan Culpepper made con-
siderable contributions to our JaMake project. They helped me gain insights into some of the problems and implementation details described in this thesis and I thank them for that.

During my life in Houston I made many lifelong friends. Dan Thompson, Cormac Flanagan, Kris Schouterden, Margaret Soller and Olena Sinkevitch made my life in Houston enjoyable and fun. My Serbian friends Paco, Dejo, Žuti, Miro, Dejan and Goran always believed in me, a long time before I started to. I thank all of them for helping my work by enriching my life and providing constant support and comfort.

Last but not least, I thank my thesis writing instructor Jan Hewitt. Her guidance through the writing process and rigorous reviews of the drafts of this thesis immensely helped me to reach the final version that is ahead of you.
# Contents

Abstract ii
Acknowledgments v
List of Illustrations ix
List of Tables xii

1 Introduction 1

2 Motivation and Related Work 6
  2.1 Sources of Java Inefficiency ............................. 8
  2.2 Preliminary Experiments ................................ 10
  2.3 Discussion of the Performance Data ..................... 12
  2.4 Related Work ......................................... 15

3 Interprocedural Optimizations 22
  3.1 Class Specialization ................................... 23
  3.2 Object Inlining ......................................... 25
    3.2.1 An Example Of Object Inlining ...................... 25
    3.2.2 Safety Analysis For Object Inlining ............... 30
    3.2.3 The Algorithm For Object Inlining ................ 41
    3.2.4 Performance Benefits Of Object Inlining .......... 49
  3.3 Experimental Results ................................ 51

4 Enabling Classical Optimizations in Java 55
  4.1 Exception Hiding ....................................... 56
4.2 Reconstruction of the Static Single Assignment Form ................. 62
  4.2.1 Static Single Assignment Form ...................................... 62
  4.2.2 Theory ........................................................................ 64
  4.2.3 Copy Insertion Algorithm ............................................. 68

5 Almost-Whole-Program Compilation ........................................... 77
  5.1 Motivation ....................................................................... 77
  5.2 Almost-Whole-Program Framework ..................................... 81
    5.2.1 Private Classes ........................................................... 82
    5.2.2 Non-extensive Classes ............................................... 82
    5.2.3 Public Classes .............................................................. 87
  5.3 Performance Results ......................................................... 89

6 Conclusions and Future Research .............................................. 91
  6.1 Conclusions ................................................................... 91
  6.2 Future Directions .............................................................. 94

Bibliography ........................................................................... 100

A OwlPack ............................................................................. 107

B The JaMake Project ............................................................. 114
## Illustrations

2.1 Sparc Ultra 5 slowdown graph (normalized to JDK 1.2 JIT Fortran style). showing a factor of 10 or more of performance penalty for OO version of OwlPack ......................................................... 13

2.2 Pentium Pro slowdown graph (normalized to Microsoft VM Fortran style). showing a factor of 60 on average of performance penalty for OO version of OwlPack ......................................................... 14

3.1 Class specialization of NMatrix class .............................................. 24

3.2 Code of the contained and the container before object inlining ............ 27

3.3 Code for the container class after object inlining ............................. 29

3.4 Analysis for object inlining ......................................................... 36

3.5 Single method analysis algorithm for object inlining ....................... 38

3.6 Local method analysis for object inlining ..................................... 39

3.7 Object inlining algorithm ............................................................. 42

3.8 Object inlining algorithm for a single method ............................... 44

3.9 An example showing the process of recursive method cloning .......... 48

3.10 Code for statement LDouble t = ooaM.getElem(4.4), after object inlining 53

4.1 An example of loop invariant code motion, showing the shorter loop body after code motion ........................................ 59

4.2 Removing barriers by inserting run-time tests ............................... 60

4.3 Removing barriers by peeling the first iteration of the loop ............. 61
4.4 Removing barriers by inserting dummy instructions ........................................... 61
4.5 CFG to SSA conversion .................................................................................. 63
4.6 The “Lost Copy” problem, showing the incorrect and correct placement of
copies for o nodes ............................................................................................ 69
4.7 Dependencies between o nodes, showing the incorrect placement of
copies for o nodes ............................................................................................ 69
4.8 “Swap” problem ............................................................................................. 70
4.9 “Virtual Swap” problem ................................................................................ 71
4.10 Inserting copies for virtual swap problem .................................................... 72
4.11 Dominance Forest Construction Algorithm .................................................. 73
4.12 Copy Insertion Algorithm .............................................................................. 74

5.1 Performance versus flexibility diagrams, showing the trade-off between
performance and flexibility in different systems ................................................. 80
5.2 The development-to-distribution transformation of private classes ............... 83
5.3 The development-to-distribution transformation of a part of the class
hierarchy ............................................................................................................... 84
5.4 Original class Foo, marked as non-extensive ................................................ 85
5.5 Generated auxiliary abstract class AFoo ......................................................... 86
5.6 Generated class Foo ....................................................................................... 87

A.1 Matrix class hierarchy, reflecting the “Lite” OO programming style .............. 108
A.2 NMatrix class hierarchy, reflecting the full OO programming style ............. 109
A.3 Fully polymorphic LNumber class hierarchy. .............................................. 109
A.4 OO style determinant computation ............................................................... 111
A.5 “Lite” OO style determinant computation .................................................... 112
A.6 Fortran style determinant computation ......................................................... 113
B.1 The structure of the JaMake project ........................................ 116
B.2 The flow of the source code through the JaMake compiler .......... 117
Tables

2.1 Pentium Pro execution times (in seconds) for preliminary experiments . . . 11
2.2 Sun Ultra 5 execution times (in seconds) for preliminary experiments . . . 12

3.1 Sun Ultra 5 execution times, showing the 5-10 times speedup of the
optimized code over the original code . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

5.1 Sun Ultra 5 execution times, showing the overhead for
almost-whole-program transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
Chapter 1

Introduction

This dissertation introduces several new compilation techniques that enable coding of high performance, object-oriented, scientific programs in Java — a very popular trend in the scientific community — without the performance loss usually associated with such practice. The ultimate result of this dissertation is the development of compiler technologies that reward, rather than penalize, good object-oriented programming practice.

When Java is considered as a platform for scientific applications, performance becomes the principal issue. Currently, Java implementations are not on par with the Fortran or C code compiled and optimized for a specific target machine. In addition, Java programs incur overhead as a result of portability, security, and multi-threading [45]. Finally, as an object-oriented language, Java encourages programmers to use an object-oriented style, thus incurring further performance penalties. Although Java implementations have matured considerably, they still fall short on programs that use the full power of Java’s object-oriented features.

A great deal of progress has been made in addressing the first two problems, but very little on the last one. Non object-oriented, Fortran-style Java programs perform within a factor of four of compiled and optimized Fortran code. However, Chapter 2 shows that Java compilers (both static and JIT) are not yet up to the task of effectively optimizing away the overhead of an object-oriented style. A “good” programming style — the one that emphasizes reuse, extensibility and maintainability of the software — results in a bad performance. Polymorphism, memory allocation and garbage collection, and the added indirection due to encapsulation can have an effect of up to two orders of magnitude on the performance.
This dissertation presents an analysis of the costs associated with polymorphic, object-oriented, scientific programming in Java, and evaluates compiler strategies that reduce the overhead of such programming style. The study introduces new compiler strategies for whole-program and almost whole-program compilation that reduce the overhead of polymorphic, object-oriented design.

Chapter 2 describes some preliminary experiments that motivate the research described in this thesis. It introduces an object-oriented, polymorphic, scientific benchmarking package developed as part of this thesis that serves as the experimental platform for validating the techniques this thesis introduces. It analyzes the performance of a set of scientific algorithms — the LINPACK package — implemented in Java using several different programming styles. The results demonstrate that the choice of programming style can affect the performance of scientific programs by up to two orders of magnitude. Unfortunately, the most desirable programming style — extensive, object-oriented and polymorphic — produces the worst performance for scientific Java programs. Chapter 2 concludes that state-of-the-art Java compilers and run-time environments are not up to the task of optimizing away the overhead introduced by such a programming style. It discusses the reasons for performance deviations for different programming styles and analyzes the insights into research directions that these results present, providing the motivation for the rest of this thesis.

Chapter 3 introduces two novel whole-program optimization techniques: Class Specialization and Object Inlining. These techniques help bridge the performance gap between Java programs written in a high-level, object-oriented, polymorphic style and the equivalent low-level, Fortran-style programs.

These optimizations for the first time address several important aspects of scientific programming in Java, such as optimization of arrays and program extensibility and portability. They overcome the restrictions posed to the source code compiler by the Java bytecode execution model and maintain full security and portability of the Java bytecode.
Class specialization clones a class containing polymorphic fields based on the possible subtypes of those fields. It generates several specialized versions of the original class, exposing the subtype distinctions in the revised class hierarchy [19, 28]. The specialized classes are monomorphic with respect to the selected fields, enabling subsequent optimizations such as object inlining. This transformation extends the classical notion of cloning [30, 31], where a function is cloned based on the values or types of its parameters, by allowing a class to be cloned based on the exact type of the polymorphic data it contains [19, 28]. Class specialization results in only modest performance benefits: its main result is that it enables subsequent object inlining.

Object inlining is a novel program transformation that converts Java objects into inlined objects, a new data representation for the objects, and transforms one program into another that operates on this new representation. This transformation eliminates the indirection in object accesses by “flattening” the data structures in the program. It enables method inlining on the inlined objects by eliminating the privacy restrictions of the original code. Moreover, it improves the memory hierarchy performance by increasing the locality of the data. It reduces the overhead of dynamic memory management by allocating some objects on the stack and reducing the total number of objects in the program. In addition, it reduces the memory footprint of the program by reducing the number of objects among which the program data is distributed. Finally, it creates more space for local compiler optimizations by performing these transformations. Object inlining can improve the performance of polymorphic, object-oriented, scientific Java programs by up to two orders of magnitude.

If Java is to be used for high performance computing, some of the well-understood code-moving optimizations must also be implemented. As Chapter 4 demonstrates, implementation of these optimizations is not straightforward in Java. In particular, the Java exception mechanism prevents or seriously limits most of the code motion optimizations. Chapter 4 introduces exception hiding, a program transformation that enables more efficient code motion. This technique uses loop peeling [7] and guard insertion to create “exception-
free zones” for code motion. Code motion transformations can then freely move the code within these zones, without concern for exceptions.

Static Single Assignment (SSA) form, the standard intermediate representation for programs in a compiler, enables efficient implementation of many classical code optimization techniques [34, 12]. The SSA versions of many traditional optimization algorithms are both more efficient and easier to implement than their non-SSA equivalents. An efficient SSA representation is needed to improve the implementation of classical optimizations. Chapter 4 presents a novel algorithm for converting SSA form into a control flow graph (CFG). This is a very fast algorithm — it is nearly linear in time, while inserting many fewer copies than the standard SSA-to-CFG conversion algorithm. This chapter first proves various properties of the SSA form and then uses those properties to develop this fast algorithm. This algorithm has a much wider applicability than just in compiling of Java for high performance, since it improves the compiler technology in general.

Chapter 5 introduces Almost-whole-program compilation. This is a novel compilation strategy in which the compiler assumes a static class hierarchy at compile time [19] and the programmer specifies the classes that would be publicly visible. This strategy uses Java visibility rules, novel implementation techniques, and novel class packaging techniques to allow for extensive program optimization in Java. These techniques allow both good design and good performance by dividing the compilation process into two fundamentally different processes: development and distribution.

In the development phase, it provides the programmers with freedom to develop flexible, extensible object-oriented programs, without worrying about the performance. This enables the programmers to concentrate on the overall design and extensibility of the program without forcing them to introduce performance-design trade-offs.

Whole-program compilation [37, 27] is too restrictive for Java. Such a model produces programs that are either non-portable (by generating machine code at the client side) or completely inflexible (by generating closed programs at the server side). The almost-whole-program compilation model eliminates this problem by introducing a distribution
phase in the compilation process. This phase is completely controlled by the programmer, allowing a balance between the performance of the generated program and its flexibility from the end-user viewpoint. This way, the programmer can first decide on the acceptable level of flexibility that the end-user needs (or solicit this information from the end-user), and then transform the program to allow for maximum performance under those flexibility restrictions. The programmer can completely restrict the flexibility of the produced program and thus enable full whole-program optimizations on one extreme, completely restrict the whole-program optimizations and thus enable full program flexibility to the end-user on the other extreme, and fine-tune the program to anywhere in between these two extremes in general. Regardless of the choice, the code that almost-whole-program compilation produces is always fully portable and verifiable Java bytecode.

Formulated in a single sentence, the thesis of this dissertation is:

**Whole and almost-whole program optimization techniques can improve the performance of many scientific Java programs written in a polymorphic, object-oriented style to within a factor of two of the equivalent hand-coded Fortran-style programs, while maintaining an acceptable level of Java bytecode portability and flexibility.**
Chapter 2

Motivation and Related Work

This chapter describes some preliminary experiments that have helped in motivating the research described in this thesis. It introduces a benchmarking package developed as part of this thesis, describes some performance measurements using the state-of-the-art compilers and Java virtual machines, and discuss the research directions these results have encouraged. It also describes the research that has been done in the areas of object-oriented language compilation, functional languages and classical compilers, the work which forms a background for the techniques described later in this thesis.

Since the introduction of the Java programming language, there has been widespread interest in the use of Java for high performance scientific computing. As evidenced by the emergence of workshops and conferences that have, as a primary subject, high-performance Java (such as Java Grande) and the growth in their attendance, the interest in using Java for high-performance computing is significant and expanding. There are numerous scientists in the research community who are attempting to use Java for their scientific applications.

However, one major impediment to such use of Java is the performance penalty paid relative "classical" languages, such as C/C++ or Fortran. It has been shown in various studies that Java performance lags at about a factor of four behind that of Fortran or C. If Java is to be used for scientific programming, significant advancements in compiler technologies are needed for overcoming this performance penalty. To better understand the impact that programming style has on performance of scientific Java programs, we have concentrated a part of our efforts on development of a benchmarking suite. This suite reflects a style of programming most likely to be used by a trained Java programmer that devotes most of his attention to writing the most elegant, easily maintainable, and highly extensible code.
This effort produced a collection of scientific Java programs that uses the full power of object-oriented design. The emphasis on these programs is both 'scientific' and 'object-oriented'. For the scientific part, this suite chooses the LINPACK library as one of the most prominent examples of "classical" scientific programming. Although there are existing implementations of LINPACK in Java, most of these are produced by direct translation from Fortran; as such they do not reflect the style of programming that a good object-oriented programmer would use in Java. Thus, we have developed our own object-oriented version of the LINPACK library, called OwlPack (Objects Within Linear algebra PACKage). Owlpack dramatically improves the style of programming over the original LINPACK by using a polymorphic number abstraction. A detailed description of OwlPack is given in Appendix A.

The code for OwlPack is about four times shorter than for the equivalent Fortran-style version. This illustrates the fact that high performance computing is not exempt from the need for extensible programming techniques. High performance scientific programs can be as big and as hard to maintain and extend as any other programs, and the techniques that enhance their reuse, extensibility and maintainability are highly desirable.

This thesis investigates the performance penalty incurred by using the pure polymorphic, object-oriented style for scientific programs. To support this effort, we developed two object-oriented versions of LINPACK in Java, a true polymorphic version and a Lite version designed for higher performance by eliminating the polymorphism through code replication. We performed a detailed performance analysis using several leading Java compilers and virtual machines, comparing the performance of our two object-oriented versions of the benchmark with a version produced by direct translation from Fortran. As the second part of this chapter shows, even though commercial and research Java VM implementations have made great strides in improving the performance of Java, they still fall to achieve acceptable performance on programs that use the full power of Java's object-oriented features.
2.1 Sources of Java Inefficiency

Discussion of Java inefficiency has recently been a favorite topic in the research community. Three main reasons for Java programs not to achieve high performance by comparison with Fortran and C are:

- Java compilers and execution environments are not yet on par with the traditional optimizing compilers. Although there has been a significant advancement in this area lately [18, 37, 38], especially with run-time compilation and optimization techniques [67, 3, 2], Java systems still have to incorporate many optimizations to be able to compete with the traditional compilers.

- The non object-oriented features of Java add significant overhead. Bytecode portability requires that major optimizations be delayed until run time. Garbage collection [3], synchronization [2], and the exception mechanism [53] all require additional overhead for their implementation. Java security measures force the virtual machine implementation to examine the code for security holes before execution, extending the execution time. All these Java features, important as they are, reduce the performance of Java programs at run time.

- Java is an object-oriented language, and as such it encourages programmers to use object-oriented style when writing scientific programs. It is far more natural for the programmers to think of matrices, vectors and complex numbers as objects and pass them around and use encapsulation and code reuse when performing operations on them, than to perform all the operations directly on the Fortran-style arrays.

Even though significant efforts have been made to address the first two topics, very little thought has been given to the third. Static and run-time compilation techniques have been developed to eliminate or lessen the problems following from the first two described issues. We believe that effect of the programming style on the performance of scientific Java programs is the most important issue of the three that are discussed, and will show
that it can produce the most drastic performance variations. Section 2.2 shows that Java compilers (both static and JIT) are not yet up to the task of effectively optimizing away the overhead resulting from using the object-oriented style of programming.

If designed properly, scientific object-oriented programs fast to develop, easy to maintain, and unconstrained to extend. The right solution is to build compiler systems that minimize the penalties for fully utilizing the features of the language.

Effective research on Java compiler systems must be driven by experimental methods. Without good benchmarks on which to conduct these experiments, it is difficult to validate the compiler strategies proposed by researchers. To verify the strategies proposed in this thesis, we developed a benchmarking suite that can readily expose the performance differences incurred by utilization of different programming styles.

The majority of benchmarks available for evaluation of the cost of using Java in high performance scientific computing are not suitable for modern compiler evaluation. These programs are either micro-benchmarks or benchmarks obtained by direct translation from Fortran (either automatic [26] or manual [43]. Neither of these closely resemble the object-oriented programs that Java programmers would prefer to write. The need for a benchmark that would closely reflect the “real world” scientific computation in Java is clear. Although there have been some reports of scientific applications implemented in Java that could be easily converted to serve as benchmarks, evidently many of these have been translated to Java without a corresponding conversion to true object-oriented programming style. A good example is the Java version of the LINPACK Benchmark [26], which strongly resembles the Fortran version.

To address this issue and to help foster more research on Java compilation, we have designed and implemented OwlPack, an object-oriented version of the LINPACK linear algebra library [21]. We used OwlPack to perform a detailed analysis of the performance of Java programs written in different programming styles [20]. Specifically, we compared the performance of the object-oriented version of the benchmark with a version written in a style closer to Fortran. We have analyzed the results of these experiments to illuminate
important issues that must be addressed by Java compiler and run-time systems research if we are to improve the performance of scientific programs written in Java. A detailed overview of the design issues in OwlPack is given in Appendix A.

2.2 Preliminary Experiments

We performed an experiment to evaluate our predictions on the performance penalties associated with using the pure object-oriented style in high performance scientific computing.

This experiment compared OwlPack with the partial Java version from FPL Statistics Group [43], obtained by straight-line transformation from the Fortran source code (referred to as Fortran style). The tables 2.1 and 2.2 include only the timings for the routines that have been implemented in the FPL version: factorization and solving the positive definite matrix, LU and QR decomposition and solving of the full matrix, inverse and determinant computation and singular value decomposition. These routines were available only for double precision floating point numbers, so the experiment included the comparison of the running times of their equivalents OwlPack: *DPoFull* and *DFull* classes handle these functions in our Lite OO version, while *NPoFull* and *NFull* classes instantiated with *LDouble* numbers handle them in our OO version. A short description of these programs is given below:

- *dpofa* factors a 300x300 random generated positive definite matrix
- *dposl* solves the equation \( A \times x = B \), where \( A \) is the matrix factored by *dpofa* and \( B \) is a random-generated vector of 300 numbers
- *dpodi* computes the determinant and the inverse of the 300x300 positive definite matrix
- *dgefa* performs an LU factorization of a 200x200 random-generated full matrix
- *dgesl* solves \( A \times x = B \), where \( A \) is factored full matrix, \( B \) is a vector size 200
- dgedi computes the determinant and the inverse of a full 200x200 matrix
- dqrdc performs QR decomposition with pivoting on a 300x300 random full matrix
- dqrs1 solves \( A \times x = B \), where \( A \) is QR decomposed matrix, and \( B \) is a vector
- dsvdc performs the singular value decomposition on a random 100x100 matrix.

This experiment included the tests on Sun Ultra 5, with 64MB of memory, running Solaris 2.6, with the jdk 1.1.5 from JavaSoft for the interpreter tests, and the jdk 1.2 Production Release for Solaris for the JIT tests.

The experiment included PC tests on a 200 MHz Pentium Pro with 64MB of memory, running Windows NT Workstation 4.0, using the jdk 1.1.6 from JavaSoft for the interpreter tests, and the Symantec JIT 3.00.029 that comes with Symantec Visual Cafe for the JIT tests. It also included measurements of the execution times for the Microsoft VM 4.79.2405 that comes with Microsoft Java SDK 3.0.

| Table 2.1 : Pentium Pro execution times (in seconds) for preliminary experiments |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                                 | Forrun style    | "Lite"          | OO style        | Forrun style    | "Lite"          | OO style        |
|                                 | "OO style"      | "OO style"      | "OO style"      | "OO style"      | "OO style"      | "Win32"         | "native"        |
| \textit{dpoja}                  | 0.511           | 0.741           | 75.849          | 0.721           | 0.711           | 52.966          | 0.210           |
| \textit{dpost}                  | 6.91            | 7.13            | 102.307         | 6.69            | 7.011           | 40.278          | 0.781           |
| \textit{dpoa}                   | 1.292           | 1.763           | 226.275         | 1.792           | 1.943           | 164.547         | 0.601           |
| \textit{dgefa}                  | 0.411           | 0.551           | 69.039          | 0.541           | 0.581           | 41.420          | 0.311           |
| \textit{dgest}                  | 0.510           | 0.681           | 20.870          | 0.531           | 0.561           | 8.912           | 0.481           |
| \textit{dgedi}                  | 0.841           | 1.031           | 124.730         | 1.051           | 1.101           | 68.509          | 0.441           |
| \textit{dqrdc}                  | 2.854           | 4.046           | 437.149         | 3.725           | 4.477           | 275.226         | 1.211           |
| \textit{dqrs1}                  | 1.782           | 1.983           | 155.674         | 2.123           | 2.103           | 67.307          | 0.711           |
| \textit{dsvdc}                  | 0.952           | 2.233           | 170.586         | 1.052           | 2.734           | 91.542          | 0.421           |

All tests were run as single processes on quiescent machines with no other processes running. All times are an average of three runs. All the classes were compiled using javac
Table 2.2: Sun Ultra 5 execution times (in seconds) for preliminary experiments

<table>
<thead>
<tr>
<th></th>
<th>jdk 1.1.5 Interpreter</th>
<th>jdk 1.2 JIT</th>
<th>F90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fortran style</td>
<td>&quot;Lite&quot; OO</td>
<td>OO style</td>
</tr>
<tr>
<td>dpoia</td>
<td>4.669</td>
<td>5.548</td>
<td>115.155</td>
</tr>
<tr>
<td>dposl</td>
<td>8.82</td>
<td>10.362</td>
<td>127.392</td>
</tr>
<tr>
<td>dpodi</td>
<td>12.736</td>
<td>14.315</td>
<td>297.346</td>
</tr>
<tr>
<td>dgefa</td>
<td>3.226</td>
<td>3.511</td>
<td>81.981</td>
</tr>
<tr>
<td>dgesl</td>
<td>4.079</td>
<td>4.640</td>
<td>35.868</td>
</tr>
<tr>
<td>dgedi</td>
<td>6.234</td>
<td>6.971</td>
<td>161.868</td>
</tr>
<tr>
<td>dqrdc</td>
<td>21.197</td>
<td>25.139</td>
<td>538.53</td>
</tr>
<tr>
<td>dqrsd</td>
<td>14.504</td>
<td>15.757</td>
<td>162.044</td>
</tr>
<tr>
<td>dsvedc</td>
<td>9.008</td>
<td>15.439</td>
<td>226.495</td>
</tr>
</tbody>
</table>

from jdk 1.1.6 with the -O option. For reference, we performed the same measurements of the Fortran version of the LINPACK, using the Fortran 90 native compiler for Solaris and the Digital PowerStation F90 compiler for the Windows NT machine. In general, the execution times for Fortran style Java LINPACK on the jdk 1.2 Production Release and on the Microsoft VM were within the factor of four of native Fortran code.

2.3 Discussion of the Performance Data

The data presented in the Section 2.2 provides some interesting and somewhat surprising results. The substantial performance penalty associated with the object-oriented design was expected, although the magnitude of this cost exceeded our expectations. In general, the Lite OO version of OwlPack performed somewhat more slowly than the Fortran-style version, with differences ranging from 7% faster for dposl on the Symantec JIT, to around 2.7 times slower for dsvedc on the Microsoft VM. The main reason for this performance degradation is additional indirection in the innermost loops of our routines. The Fortran-style library has all the matrices passed by reference to the target routine, and the argument of the routine contains the direct reference to the matrices involved in the computation. The
**Figure 2.1**: Sparc Ultra 5 slowdown graph (normalized to JDK 1.2 JIT Fortran style), showing a factor of 10 or more of performance penalty for OO version of OwlPack

Lite OO library treats matrices as objects, with the reference to the actual array that contains the matrix data stored as an instance variable. The most natural way to reference the matrix data is through the instance variable, i.e. $Mat[i][j]$, but this generates an additional field access to obtain $Mat$ from this reference, followed by the usual array access instructions. In most, but not in all cases, the JIT compilers were able to optimize this extra reference away. This problem could be easily eliminated even in a static compiler with some form of loop invariant code motion or scalar replacement.

The most surprising timings were the ones recorded for the OO version of the OwlPack. The object-oriented version was substantially slower in all of our tests, ranging from around 4 times slower than the Fortran-style version for dqrs1 and dgesl on jdk 1.2 on Solaris, to more than 250 times slower for dqrdc on the Symantec JIT. The average slowdown was from a factor of 19 for the Solaris JIT to a factor of 140 for the Symantec JIT.

Although the Lite OO version of OwlPack has an enormous performance advantage over the OO version on current Java implementations, the polymorphic object-oriented
Figure 2.2: Pentium Pro slowdown graph (normalized to Microsoft VM Fortran style), showing a factor of 60 on average of performance penalty for OO version of OwlPack.

style results in a code that is much smaller in size—roughly a factor of 3 for OwlPack. In addition, it is easier to maintain because algorithmic changes need to be made only once. Compiler technologies that we are proposing in this thesis can automatically transform the OO style OwlPack code into a code that approaches the Lite OO version in appearance, and more significantly, in performance.

We believe that the main reasons for the dramatically inferior performance of the OO version can be categorized as follows:

- Every number that is a part of a computation is allocated on the heap as a separate object, requiring additional overhead for instantiation and garbage collection.

- Numbers that are elements of a matrix are scattered over the heap, effectively eliminating the cache performance benefits of spatial locality in standard matrices used in the Fortran and OO Lite versions.
• Since all numbers are abstracted in the $LNumber$ class, all operations on numbers are done through method calls to the corresponding objects. This restriction incurs additional overhead for the method invocation and the dynamic dispatch required to determine which method is being invoked.

• There is a greater memory requirement for the OO style, because every number takes up memory associated with object representation in addition to memory for the data itself.

• The presence of objects and method calls prevents some forms of local compiler optimization, such as common subexpression elimination, that are possible in the Fortran and Lite OO versions.

As we will show in this thesis, it is possible to eliminate most of these performance penalties through advanced compiler optimizations. A form of specialization or procedure cloning [Section 3.1] is needed to convert the NMatrix class hierarchy into four specialized hierarchies based on the four types of the numbers that could be used as elements of the matrix ($LDouble$, $LFloat$, $LCDouble$, $LCFloat$). This optimization alone does not bring significant performance gains, but coupled with object inlining [Section 3.2] it generates a class structure very similar to the $Matrix$ hierarchy in OwlPack. These techniques can be applied in Java environment without sacrificing the portability and security of the Java VM.

The next section describes the relation of this dissertation to the work that has been done in the areas of object-oriented language compilation, functional languages and classical compilers. That work forms a background for the techniques described later in this thesis.

2.4 Related Work

This thesis is largely based on combining the work that has been done on compiler optimizations for object-oriented languages [1, 28, 27, 28, 36, 37, 55, 64], functional languages [42, 54, 56, 66, 68], and compilers for vector and parallel machines [40, 48, 69], as
well as some techniques from classical scalar compilers [14, 30]. The unique paradigm - compiling for the Internet - that Java has imposed on compiler research has allowed us to explore these techniques, modify, extend and apply them in this new environment. We will now address some of the work in these areas that is closely related to this thesis.

Object-Oriented Languages Compilation

Dean et al. [36, 37, 27] have developed a whole program optimization framework for object-oriented languages. They have proposed several optimization and analysis techniques for compilation of such languages when the whole program is known at the compile time. This thesis is concentrated on further developing some of their techniques and some new ones in the context of Java programming language and the unique problems created by Java's execution model. This research is also more focused on compilation of high-performance scientific programs written in Java rather than on common, everyday programs. It also exploit the impact the almost-whole program optimization framework has on applicability of their techniques.

Object inlining was introduced simultaneously by Budimlić and Kennedy [18, 19] and Dolby and Chien [38, 39]. Both these research groups present a similar idea: inlining whole objects to eliminate indirection and improve the performance of the generated code. There are some major differences, however. Dolby and Chien focus only on whole program optimization of C++ programs. Whole-program compilation is a significant restriction for Java programs, since it reduces their flexibility and ultimately their usability. In addition, our technique is more complex, since in Java objects can be inlined only with some additional restrictions, which are trivial to overcome through type-unsafe casting and pointer arithmetics in C++, as we discuss in section 3.2.2. The most important advantage of our technique is handling arrays of objects, which is essential for transformation of high performance, object-oriented, scientific programs. Furthermore, Dolby and Chien inline only objects inside other objects, reducing heap allocation cost since only one allocation for the resulting object is required instead of separate allocations for all contained objects, while
our technique also converts objects into local variables, essentially changing heap to stack allocation, which results in a much better performance.

**Functional Languages**

A significant amount of work in functional languages community has been done on unboxing. Leroy [56] introduces new constructs to core ML language: wrap and unwrap, which are inserted in the code to handle boxing and unboxing of the objects. To our knowledge, this thesis is the first work to apply a similar technique to a non-functional language. Our technique is also more complex, since it achieves this without introducing any new constructs to the language. The techniques presented in this thesis have to do a more extensive safety analysis since all the transformations are one-way (unwrapping) — there is no notion of reconstructing objects from inlined objects (wrapping).

Hindley and Milner [59], Kranz et al. [54], Shao and Appel [66] and Tarditi et al. [68] have all looked into developing type inference algorithms for functional languages. The work of Cartwright et al. on soft typing [24, 71] for Scheme is also closely related to this problem. All this research has been a good starting point for Agesen [1], who has done some substantial progress in inferring concrete types for object-oriented languages and their usage for compilation purposes. His methods should prove highly useful for providing necessary analysis information needed for class specialization and object inlining. Others have also looked into the problem of type inference for object-oriented languages, most significant being the work of Plevyak and Chien [64], Palsberg and Schwartzbach [62, 63] and Elfrig et al. [41].

Based on the set-based analysis of Heintze [49], Flanagan and Felleisen [42] have developed a modular set-based analysis that is capable of inferring the sets of values that program variables can have at a given program point. Their analysis is fairly precise, executes in practically linear time and is a good alternative to type analysis methods mentioned above for obtaining the program information needed for the optimizations presented here. This thesis uses Java Spidey, an implementation of the set-based analysis technique to ob-
tain precise type information needed for object inlining. Even though Java Spidey produces quite precise information, some refinements of this technique are still necessary to infer the information that would be usable for object inlining and class specialization.

**Procedure Cloning and Specialization**

Cooper, Hall and Kennedy [30, 31] pioneered the work on procedure cloning, that laid the grounds for code specialization. Their approach uses procedure cloning to make copies of chosen procedures based on the values of their parameters. Call sites are partitioned and procedure copies are then individually optimized based on the restricted number of possible call sites for those procedures and the known value of some of their parameters.

Chambers et al. [28] and Dean et al. [35] extended this idea further to clone procedures based on the type of the parameter(s) passed, thus creating more precise call graph information in the presence of polymorphism. This has enabled them to apply more traditional compiler optimizations (such as inlining). This thesis takes a similar approach and relax the requirement for the whole program, thus preserving some of the Java bytecode flexibility.

**Exception Analysis**

Hennessy [50] describes modifications to data-flow analysis in order to handle programs with exceptions. Unfortunately, his approach is not directly applicable to object-oriented languages since he assumes that the call graph of the program is known at the analysis time. The exception hiding transformation from this thesis does not make this assumption. Also, exception hiding attempts to reduce the impact of exceptions on the data flow analysis as much as possible by using set-based analysis information to prove that some instructions will never raise an exception. Hennessy has not reported any implementation experience with his method.

Aiken et al. [5] use denotational semantics to describe valid program transformations (transformations that preserve the meaning of the program) in the presence of errors in a given context. By introducing an annotation, 'Safe', that marks parts of the program that
cannot produce errors, they are able to prove validity of various program optimizations. While effective on a pure functional programming language (FL), there is no evidence that their method is powerful enough to formally describe interesting code-movement transformations in real world object-oriented languages such as Java and prove the validity of those transformations. Authors state that “it remains to be proven that Safe enables large scale optimization in practice”.

**Classical Compilers**

To fully exploit the benefits of class specialization and object inlining, some of the classical compiler optimizations have to be employed afterward. The most appealing of these include dead code elimination [17], constant propagation [70] and value numbering [14]. The effects of these optimizations will be similar to their effects on procedural programs, once the techniques proposed in this thesis transform object-oriented programs into a style similar to 'old' procedural programs.

Code motion techniques, such as loop invariant code motion [4, 33], partial redundancy elimination [60] and strength reduction [8] play an important role in every optimizing compiler. They require special attention when adopted for Java. The exception hiding transformation described in this thesis enables implementation of these techniques, although in somewhat limited form. The performance improvements due to these techniques cannot be expected to be as high as in the classical compilers for procedural languages, but are still significant enough to justify their implementation and the development of the exception hiding.

Havlak and Kennedy [48] and Triolet et al. [69] have developed regular section analysis techniques to efficiently describe parts of arrays that are affected by particular loops in the program. While these techniques were mainly targeting high performance compilation for parallel machines, their simplified versions can be used to help isolate arrays and parts of arrays that are amenable to object inlining, guaranteeing the correctness of applying this
optimization to those arrays. This approach enables more extensive object inlining that can involve only parts of the arrays, further improving the performance.

Hall and Kennedy [46] have developed an efficient call graph analysis algorithm for Fortran in the presence of functional arguments. The problem addressed by this approach is similar to the one we use type analysis algorithms: find the exact target of all the method calls in the program. In object-oriented languages, this problem is created by dynamic dispatches of method calls, and in Fortran by passing functions as parameters to other functions. Transformation of the call graph analysis algorithm for Java has the potential of supplying the information needed for object inlining and class specialization transformations at a much lower cost than type analysis, a feature which makes it an attractive alternative for providing the information for program transformation described in this thesis.

Cytron et al. [34] have proposed the SSA form as an advanced intermediate compiler representation. Their algorithm is still widely used for conversion from CFG into SSA and back. Many improvements to their approach have been suggested since, including semi-pruned and pruned SSA forms [16, 12], the gated SSA form [47], and the array SSA form [52]. However, none of these techniques considers reducing the number of copies needed after SSA is used for optimizations that require copy insertion for o node instantiation. The SSA-to-CFG conversion algorithm described in this thesis improves the SSA form further by reducing the number of copies that are inserted for the o nodes.

Briggs et al. [16, 12] have written detailed reports on implementation of the SSA algorithm by Cytron et al. [34]. They describe many improvements to the original design, as well as some implementation difficulties, such as the lost copy problem and the swap problem. They do not consider reducing the number of copies for o node instantiation either.

Traditional copy propagation algorithms [4] do not take advantage of the SSA form and the dominance information which is available when the code is converted into SSA form. Rather, it relies on computing the DEF-USE and USE-DEF information in the CFG, as
well as solving the data-flow problem for reaching copies, essentially yielding the $O(n^2)$ algorithmic complexity.

Leung and George [57] have developed an algorithm for inserting a minimal number of copies during the conversion of the SSA into CFG for machine code, which on a first glance seems like a competing algorithm to the algorithm presented in this thesis. However, their work is concentrated on preserving the references to the machine registers while the program is in SSA form. Their algorithm inserts the minimal number of copies necessary to preserve the machine register requirements. In their paper they are not concerned with reducing the number of copies inserted to instantiate $o$ nodes, and we are assuming that they are inserting all the copies, as in [34, 16].
Chapter 3

Interprocedural Optimizations

This chapter describes two novel optimization techniques: *Class Specialization* and *Object Inlining*. These techniques help bridge the performance gap between Java programs written in high-level, object-oriented, polymorphic style and the equivalent low-level Fortran style programs.

Even though similar optimizations have been proposed before in functional [56] and highly dynamic languages [28], none of these techniques has been employed in production Java implementations. This thesis describes optimizations that are effective on a real-world language, in a real-world compiler, on real-world scientific applications. These optimizations address, for the first time, several important aspects of scientific programming in Java, such as optimization of arrays and program extensibility.

- *Class specialization*, described in the next section, clones a class containing polymorphic fields based on the possible subtypes of those fields. It generates several specialized versions of the original class, exposing the subtype distinctions in the revised class hierarchy [19, 28]. The specialized classes are monomorphic with respect to the selected fields, enabling subsequent optimizations such as object inlining. This transformation extends the classical notion of cloning [30, 31], where a function is cloned based on the values or types of its parameters, by allowing a class to be cloned based on the exact type of the polymorphic data it contains [19, 28].

- *Object inlining* identifies objects that are instantiated and used entirely within a known portion of the program and expands them in line [18, 19]. This transformation eliminates indirection in data representations, enables subsequent code inlining, improves cache performance and reduces pressure on the garbage collector.
3.1 Class Specialization

In procedure cloning [30, 31], distinct copies of a procedure are made for call chains with distinct parameter values. For example, if two different call chains deliver different procedure values to a procedure argument, cloning can disambiguate those call chains. Another example is constant propagation. If there are call chains for which the value of a parameter is a known constant, cloning permits that constant value to be used in optimizations in the clone specialized to that value. Typically performance improvements derive from later interprocedural optimizations, such as dead code elimination and constant folding, rather than from cloning itself.

For object-oriented languages, the concept of cloning has has been been further generalized. For those languages, the optimization is known as customization [28, 55] or specialization [35]. Specialization clones methods based on the instantiation type of the object on which these methods are executed—the object referred to by "this". Similar to cloning, the main contribution of specialization is generation of a more precise call graph and elimination of dynamic dispatches that could appear at every call site.

This thesis takes this approach even further with class specialization. Instead of specializing a method only on the exact type of the this argument, the whole class is specialized based on the exact types of the polymorphic data it contains. This transformation generates several specialized versions of the original class, exposing the subtype distinctions in the revised class hierarchy.

Figure 3.1 shows the result of the process of class specialization on a part of OwlPack code. The results of specializing the class on Figure 3.1a are the classes on Figures 3.1b and 3.1c. Class NMatrix contains variable Mat, which is a reference to an array of LNumber objects. LNumber class is polymorphic, so according to its class hierarchy (please refer to Figure A.3 in Appendix A) four specialized versions of the class NMatrix are created: LFloat_NMatrix, LDouble_NMatrix, LCDouble_NMatrix and LCFloat_NMatrix that contain arrays of objects that have a corresponding exact type. For simplicity, Figure 3.1 shows only two of the generated classes.
class NMatrix{
    LNumber[][] Mat;
    int rows, cols;
    int[] pivot;
    void NMatrix(LNumber[][] F)
    {
        rows = F.length;
        cols = (F[0]).length;
        Mat = F;
        pivot = new int[cols];
    }
    ...
}

a. Original class

class LDouble_NMatrix{
    LDouble[][] Mat;
    int rows, cols;
    int[] pivot;
    void LDouble_NMatrix
    (LDouble[][] F)
    {
        rows = F.length;
        cols = (F[0]).length;
        Mat = F;
        pivot = new int[cols];
    }
    ...
}

b. Specialized for type LDouble

class LCFloat_NMatrix{
    LCFloat[][] Mat;
    int rows, cols;
    int[] pivot;
    void LCFloat_NMatrix
    (LCFloat[][] F)
    {
        rows = F.length;
        cols = (F[0]).length;
        Mat = F;
        pivot = new int[cols];
    }
    ...
}
c. Specialized for type LCFloat

Figure 3.1: Class specialization of NMatrix class
A whole-program type inference \cite{1, 64} or a set-based analysis \cite{42} is needed to provide necessary information in order to perform class specialization. The creation points and exact types of the variables declared as \texttt{NMatrix} have to be determined. All the references to those variables are then changed to refer to the corresponding exact type.

Unfortunately, a naïve use of this transformation results in a combinatoric code growth in the worst case. Heuristics must be applied to isolate objects most profitable for inlining using the optimization described in Section 3.2 prior to performing class specialization only on their classes. For example, our heuristics may determine that most objects we are interested in inlining are instantiated as \texttt{LDouble}. In this case, only one specialized version of the \texttt{NMatrix} class would be generated (\texttt{LDouble\_NMatrix}), and would be used wherever \texttt{NMatrix} was used with \texttt{LDouble} numbers. All other uses of \texttt{NMatrix} would remain untouched.

### 3.2 Object Inlining

This section presents \textit{object inlining}, a novel program transformation that converts Java objects into \textit{inlined objects}, a new data representation for the objects, and transforms programs so that they operate on this new representation. This transformation eliminates the indirection, enables method inlining on the inlined objects, improves the memory hierarchy performance, reduces the overhead of dynamic memory management, reduces the memory footprint of the program and creates more space for local compiler optimizations. This section begins with an example of object inlining transformation, then describes in detail the safety analysis necessary to perform object inlining, continues with a formal algorithm for program transformations that create inlined objects and finally discusses the performance benefits of object inlining.

#### 3.2.1 An Example Of Object Inlining

We begin this section with an example of object inlining. The example illustrates how the safety analysis validates the transformations, and what kind of program transformations
need to be done to ensure observational equivalence of the original and the transformed program.

Three Java constructs form a target for object inlining:

- **Local Objects.** These are objects that are created in the program: a reference to them is stored in a local variable. Object inlining of local objects converts these heap-allocated objects into stack data.

- **Global Objects.** These are objects that are created in the program: a reference to them is stored in a field of another object. Object inlining of global objects converts the original heap structure (container object contains a reference to the contained object) into a container object that contains the data of the contained.

- **Arrays of Objects.** These are the arrays of objects that are created in the program: a reference to them is stored either in a local variable or in a field of an object. Object inlining converts these arrays into arrays of data that the elements of the original arrays contained.

Figure 3.2a shows an example of a class of an object that can be inlined. It is a simple (and incomplete) definition of a complex number. Figure 3.2b shows a simple example of a class that contains an object o and an array a of type Complex, and a method run() that has local objects 11 and 12 of type Complex. This method initializes these objects and then performs some simple operations on them.

Figure 3.3 shows the transformed class Example after object inlining has been performed on o, a, 11 and 12. Object inlining replaced an object o with the original data that the object contained. It replaced the array of objects a with arrays of data a_r and a_i, and the local variables 11 and 12 with the data original that objects contained.

Initialization of the inlined objects is quite simple: the transformation directly inlines constructor calls in the code. While inlining the constructor call, the transformation replaces all references to this within the constructor to the object being inlined, and then
all the field accesses o.r and o.i with the corresponding inlined data accesses o_r and o_i.

class Complex{
    private float r;
    private float i;
    public Complex(float r, float i){
        this.r = r;
        this.i = i;
    }
    public Complex add(Complex y){
        return new Complex(real() + y.real(),
                           imag() - y.imag());
    }
    public float real(){
        return r;
    }
    public float imag(){
        return i;
    }
    a. Class of the object to be inlined
}

class Example{
    Complex o;
    Complex[] a = new Complex[5];
}

void run(){
    for(int i=0; i<5; i++)
        a[i]=new Complex(i,i);
    Complex 11,12;
    11 = new Complex(5,6);
    12 = 11.add(11);
    11 = a[0].add(a[1]);
    a[3]=a[2].add(a[1]);
    a[0]=a[0].add(a[1]);
}

...
Object inlining transforms the second method call \( \text{ll} = \text{a[0].add(a[1])} \) into two simple instructions. We can note here that, even though the end result is very simple, the mechanism by which this transformation generates the code from Figure 3.3 has to devote attention to an additional issue: handling of the elements of arrays of inlined objects. When an inlined object that is an element of an array is passed to a method, the object inlining transformation has to ensure that the indices of the array are passed to the method as well. It generates an intermediate clone of the add method:

```java
public static void ll_add(Example t, int ti, Example Cy, int yi){
    ll_r=t.a_r[ti]+Cy.a_r[yi];  ll_i=t.a_i[ti]+Cy.a_i[yi];
}
```

and changes the call site to:

```java
ll_add(this,0,this,1);
```

This intermediate method takes as arguments references to the containing objects (due to the fact that more than one class can have inlined arrays of objects of type Complex) and the indices needed to access array elements. If the arrays were multidimensional, all the indices would have to be passed to the cloned method. When generating `ll_add`, object inlining recursively replaces all the uses of the original arguments with corresponding array accesses: `y.real()` becomes a call to a clone `real(Cy, yi)` which in turn gets inlined to `Cy.a_r[yi]`, and similarly for calls to `y.imag()`, `this.real()` and `this.imag()`.

The code above illustrates another important issue with object inlining: `ll_add` is not a valid Java method, since it references the local variables `ll_r` and `ll_i` from the method `run`. `ll_add` is an intermediate representation in object inlining implementation and is shown here only to illustrate the mechanisms behind the scenes of object inlining. After generating this method, object inlining immediately inlines it at the call site, yielding the two simple instructions in Figure 3.3.

Finally, the last method call illustrates the issues described above by showing the resulting code where the algorithm chose not to inline the method call. This is the “intermediate”
class Example{
    float o_r, o_i;
    float[] a_r = new float[5];
    float[] a_i = new float[5];
    void run(){
        for(int i = 0; i < 5; i++){
            //***inlined: a[i]=new Complex(i, i)***
            a_r[i] = i; a_i[i] = i;
            float l1_r, l1_i, l2_r, l2_i;
            //*****inlined: l1 = new Complex(5, 6)***
            l1_r = 5; l1_i = 6;
            //*****inlined: l2 = l1.add(l1);***
            l2_r = l1_r+l1_r; l2_i = l1_i+l1_i;
            //*****inlined: l1 = a[0].add(a[1!])***
            l1_r = a_r[0]+a_r[1]; l1_i = a_i[0]+a_i[1];
            a_add(this, 3, this, 2, this, 1);
        }
    }
}

public static void a_add(Example r, int ri, Example t, int ti, Example Cy, int yi){
    r.a_r[ri]=t.a_r[ti]+Cy.a_r[yi]; r.a_i[ri]=t.a_i[ti]+Cy.a_i[yi];
}
...

Figure 3.3: Code for the container class after object inlining

result that object inlining generates before doing method inlining. Since there are no local objects involved, there is no reason why this particular method call should not be inlined, but such a situation may arise in complex programs. If the cloned method is called from multiple places, a heuristic in the object inlining algorithm might decide that the code growth resulting from inlining this method call outweighs the performance benefits.

We can immediately note an optimization of this scheme. If a method does not modify the fields of an object passed to it, it would be sufficient to pass an object’s fields to the
method instead of passing the object itself. This is equivalent to passing the object in question by-value, instead of by-reference. This transformation eliminates the necessity for cloning of the method based on the container type and on the name of the object being passed to this method. In our example, instead of the method add() from Figure 3.3, we can define the following method:

```java
public static void a_add(Example r, int ri, float t_r, float t_i,
                        float y_r, float y_i){
    r.a_r[ri]=t_r+y_r;  r.a_i[ri]=t_i+y_i;
}
```

This eliminates the necessity for cloning the method add based on the this and y arguments. It is sufficient to clone this method based only on the result argument.

Method calls that involve locally inlined objects that are modified within the method body of the callee have to be inlined, since there exists no mechanism to access the inlined data outside of the caller. This restriction could be avoided in a language such as C++ by passing the address of the inlined data to the callee.

### 3.2.2 Safety Analysis For Object Inlining

In order to perform object inlining, an extensive analysis is needed to prove that this transformation preserves the meaning of the program. Depending on what the program does with objects, some objects in the program cannot be inlined.

The previous section leads to a crucial observation: object inlining in Java is not a reversible optimization — once an object is transformed into an inlined object, the notion of the original object disappears and it cannot be recreated. Some operations that are valid on objects in their original form cannot be applied to inlined objects. Safety analysis has to check the program for such operations and prevent object inlining for objects that are involved in those operations. This section presents and explains an algorithm for finding all the objects in the program which can be safely inlined.
Object Comparison

Inlined objects cannot be directly compared. Binary Java operators == and != are storage-dependent and cannot be used on inlined objects. After object inlining of an object o, there is no “object” per se, only the data that represents the former object. Operators == and != compare the place where objects are stored, and since it is not possible to get this information in Java, == and != comparisons are not allowed for inlined objects. Safety analysis eliminates the objects that are involved in comparisons from the list of candidates for object inlining.

It is possible to allow direct comparison of inlined objects in some type-unsafe object-oriented languages, such as in C++. It is sufficient to convert the comparison \((o1 == o2)\) into the address comparison of their inlined data (assuming the analysis proves that both \(o1\) and \(o2\) are non-null): \((\&o1.data == \&o2.data)\).

It should be noted that “deep” comparisons, using the method equals() can still be done on the inlined objects. Since these comparisons are simple method calls, only the method call restrictions apply to them.

Aliasing

Two object variables may never refer to the same inlined object. The reason is quite simple: object inlining is a storage transformation, and as such, it prohibits the use of storage-dependent operations such as aliasing. The storage dependence of object inlining can be illustrated with the following example:

```java
Foo o = new Foo(); // can this be object inlined?
Foo o2 = new Foo(); // how about this one?
if (...) {
    o = o2;
}
o.data++; // which storage does this refer to!!??
    // o.data or o2.data??
```
Our implementation eliminates aliasing of inlined objects by eliminating the source of aliasing: assignments. No object-to-object assignments are allowed for inlined objects. Only new instances, or method calls that return new instances, can be assigned to inlined objects.

Even an assignment to a variable of a result of a method call that \textit{does} return a new instance can induce aliasing. If the variable to which the newly created object is assigned is passed as an argument to the method, and the method has a return variable which is instantiated within the method and returned as the result, aliasing can occur between the assigned-to variable and the return variable. Safety analysis has to check for this case as well. The following code illustrates that:

```java
Foo o = new Foo(); // this can be object inlined
...
o = foo(o);
...

Foo foo(Foo arg) {
    Foo ret = new Foo();
    ...
    ret.data++; // arg and ret are aliased in this
    arg.data++; // part of the code!
    ...
    return ret;
}
```

Our implementation of the safety analysis tests for this case, even though it is not shown in the pseudo-code on Figure 3.5, which shows the algorithm for analyzing a single method. Assignment to an inlined object of the result of a method call, where assigned-to object is passed as an argument to the method, the corresponding formal argument is in the set \( MOD(m) \), and \( m \) is returning a new instance through a return variable is not allowed by the safety analysis.

By using a more precise alias analysis similar to that of Cooper and Kennedy [29], it would be possible to allow more objects to be inlined. However, the alias analysis would
have to be augmented to produce something akin to the *must-alias* information [9]. If a
*must-alias* b, then a can be object inlined if and only if b is object inlined.

We can note a future research direction here: in a type-unsafe object-oriented language
such as C++, it would be possible to eliminate the aliasing problem for inlined objects.
All references to the original objects can be converted to pointers to the first field of the
inlined data, and all operations on the objects converted to operations on these pointers.
However, this transformation would require that all the objects data be accessed through
these pointers, using type casts and pointer arithmetics, thus greatly reducing the effect of
this optimization. For example, let us assume that the original code was

class Foo {
    int d1;
    float d2;
    ...
};
...
Foo o, o2;
...
o = o2; //aliasing!!
...
o.d2++;

After the object inlining of o, and pointer transformation, the new code will be:

...
int o_d1; float o_d2;
int o2_d1; float o2_d2;
int* o = &o_d1;
int* o2 = &o2_d1;
...
o = o2; //aliasing;
...
(*(float*)(o+1))++;

It could even be possible to allow aliasing of inlined objects with non-inlined objects
if the difference between the values of a pointer to the object and a pointer to the objects’
first field is a known constant for a given implementation. Implementation-specific pointer arithmetic and padding the beginning of the inlined objects' data could be used to produce transformations similar to the ones above, allowing object inlining in those cases.

Unfortunately, the described transformations would eliminate most of the performance benefits of object inlining that come from eliminating indirection. However, the effects of object inlining on memory hierarchy performance would still be present, so this direction is still worth pursuing in future research.

Passing an Object To a Native Method

An inlined object cannot be passed to a native method, so the safety analysis disables object inlining for objects that are passed to a native method. There are two reasons for this:

- Native methods are not available for analysis. Generally, the source code is not available for native methods, and when it is, it is usually in a different language. Moreover, there would be no way to verify that the object code of the native method actually corresponds to the source code being analyzed.

- Object inlining requires modifications to the code. It is generally not possible to modify the native code directly to handle inlined objects properly. Even if the source code is available, obstacles will be the same as above.

If we were to adopt the copy-in, copy-out semantics, passing objects to native code would be possible. Such a scheme would create a new object and fill in its fields with the data from the inlined object before the call to native method. It would read the data from the passed object and fill in the fields of the inlined object after the call. Of course, some sort of heuristic would have to be implemented to ensure that this added overhead does not outweigh the benefits of inlining the object in the first place. Unfortunately, Java does not support the copy-in, copy-out semantic. These speculations fall outside of the scope of this thesis and are a basis for future research.
There is no fundamental difference between Java bytecodes and Java source codes, and both can be (in principle) treated equally. Conceptually, the techniques described in this thesis can be equally applied both to source code and to bytecode. However, to simplify implementation, our infrastructure handles only Java source code. Classes and methods in bytecode form are not analyzed and are treated as native. It is important to note that this is not the shortcoming of the technology described in this thesis, but rather of our specific implementation.

Next section describes the algorithm we developed for performing safety analysis that determines the set of objects that can be inlined.

The Algorithm

Figure 3.4 shows the pseudo-code for our whole program analysis algorithm which determines which global and local objects in the program can be safely inlined. The main procedure takes as arguments “entry” methods of the program, i.e. the methods from which the control flow of the whole program can start. It also needs the precise call graph information for the program, which can be derived from the concrete type information. The result of the computation are

- a set of “global” objects $O$, which contains all the objects in the program that are contained within other objects, and that can be object inlined. In the beginning, all objects that are contained inside other objects are put in this set, which is later refined by calls to Analyze, and

- a set of local objects $LO$ for every method $m$, that contains all the local objects from $m$ that can be object inlined.

Figure 3.5 shows the pseudo-code for a function Analyze, a single method analysis algorithm. The algorithm on Figure 3.4 simply calls Analyze on all the methods in the “entry” set. Function Analyze then does most of the work for computing the analysis
inputs: $IN$ (Precise call graph and concrete type information for the program)

(Set $Entry$ of all entry methods in the program)

($MOD(m)$ and $FMOD$ information for all methods)

outputs: $OUT$ (Set of objects $O$ and local objects $LO$ that can be inlined)

begin
    for $\forall c. c$ is a class in the program
        for $\forall o. o$ is a field of $c$
            if $o$ is an object $O = O \cup \{o\}$
        endfor
    endfor
    for all methods $m$ in the program
        LocalAnalyze($m$)
    endfor
    change = true
    while (change)
        change = false
        for $\forall m$ in the program
            $Visited(m) = false$
        endfor
        for $\forall m \in Entry$
            Analyze($m$)
        endfor
    endwhile
end

Figure 3.4: Analysis for object inlining

information. The information that algorithm from Figure 3.5 computes for each method is as follows:

- modified set of globally inlineable objects $O$. When an object from $O$ is determined to be non-inlineable, it is removed from $O$;

- a set of local objects $LO$ for given method $m$, that contains all the local objects from $m$ that can be object inlined;
• a set $FO$, of arguments to $m$, that contains all the formal arguments of $m$ that can be object inlined;

• a set $MOD \subset FO$, of arguments to $m$, that contains all the formal arguments of $m$ that can be object inlined and whose fields are modified by $m$ or by methods called by $m$.

This information is computed on-demand. When $Analyze$ is called, it first checks whether it already has all the information computed for given method $m$. If so, it returns the computed information. If not, it proceeds with the analysis and stores the computed information for sub-sequential calls. This ensures that the analysis is done exactly once per every method in the call graph, which guarantees the linear asymptotic complexity of the analysis algorithm.

To avoid special treatment of this argument in method calls, and this expression in the method body, the notation in the algorithm on Figure 3.5 assumes that this argument is explicitly passed to the called method. Thus, a method call $o.foo(args)$ in the algorithm on Figure 3.5 is treated as a method call $foo(args_{1})$, where $args_{1} = \{this\} \cup args$.

The function $returnsNewInstance(expr)$ from Figure 3.5 deserves an explanation. This function always returns true if $expr$ is a direct instantiation of a new object (i.e. a new expression). If $expr$ is a method call, this function analyses the method to discover what are the values that the method returns. All return statements in the method have to return a new instance (a recursive definition), or to return a locally ininlineable object. Otherwise, $returnsNewInstance(expr)$ returns false. This is a minor analysis information that our algorithm computes in the same manner that it computes $LO(m)$, $FO(m)$ and $MOD(m)$. We chose not to include it in the formal algorithm on Figure 3.5 for simplicity reasons.

A natural implementation would have to test for occurrences of "forbidden" operations on inlined objects, namely aliasing, object comparisons and passing objects to binary methods. To simplify the coding, we have designed the algorithm on Figure 3.5 somewhat differently. Rather than searching for all the cases that are not allowed, this algorithm catches all the cases that are allowed: assignment of a new instance to an inlined object, and
function Analyze:
inputs: \( IN \) (Method \( m \) to analyze)
outputs: \( INOUT \) (Set of objects \( O \) that can be inlined)
  (Set of formal arguments \( FO(m) \) that can be inlined in \( m \))
  (\( MOD(m) \subseteq FO(m) \), arguments whose fields are modified in \( m \))
\( OUT \) (Set of local objects \( LO(m) \) that can be inlined in the method \( m \))
begin
  If (\( Visited(m) \)) return \( LO(m) \), \( FO(m) \), \( MOD(m) \)
  \( Visited(m) = true \)
  for all expressions \( r \in m \), in evaluation order
    case of \( r \)
      declaration of local object \( o \):
      \( LO = LO \cup \{ o \} \)
      method call \( n(args) \):
      Analyze\( (n) \), get \( FO(n) \)
      for \( \forall o, o \in args \), \( o \) is an object
        let \( f \) be the formal argument of \( n \) corresponding to \( o \)
        if \( f \notin FO(n) \)
          if \( (o \in FO(m)) \) \( FO(m) = FO(m) \setminus \{ o \} \), change = true
          if \( (o \in O) \) \( O = O \setminus \{ o \} \)
          if \( (o \in LO(m)) \) \( LO(m) = LO(m) \setminus \{ o \} \)
        endif
        if \( f \in MOD(n) \) and \( o \in FO(m) \)
          \( MOD(m) = MOD(m) \cup \{ o \} \)
        endif
      endfor
    assignment \( o = expr, o \in O \):
    if \( \neg \text{return} \text{New} \text{Instance}(expr) \) \( O = O \setminus \{ o \} \)
    assignment \( o = expr, o \in LO \):
    if \( \neg \text{return} \text{New} \text{Instance}(expr) \) \( LO = LO \setminus \{ o \} \)
    assignment \( o = expr, o \in FO: FO(m) = FO(m) \setminus \{ o \} \), change = true
    assignment \( o.x = expr, o \in FO: MOD(m) = MOD(m) \cup \{ o \} \), change = true
    free appearance of \( o, o \in FO(m) \):
    \( FO(m) = FO(m) \setminus \{ o \} \), change = true
    free appearance of \( o, o \in O \cup LO(m) \):
    \( O = O \setminus \{ o \} \), \( LO(m) = LO(m) \setminus \{ o \} \)
  endcase
end

Figure 3.5 : Single method analysis algorithm for object inlining
function LocalAnalyze:

inputs:  \( IN \) (Method \( m \) to analyze)
\( OUT \) (Set of formal arguments \( FO(m) \) that can be inlined in \( m \))
\( MOD(m) \subseteq FO(m) \), arguments whose fields are modified in \( m \)

begin
\( FO(m) = \text{args}(m) \); \( LO(m) = \{ \} \); \( MOD(m) = \{ \} \)
for all expressions \( e \in m \), in evaluation order
\[ \text{case of } e \]
assignment \( o = expr, o \in FO: \)
\( FO(m) = FO(m) \setminus \{ o \} \)
assignment \( o . x = expr, o \in FO: \)
\( MOD(m) = MOD(m) \cup \{ o \} \)
free appearance of \( o \in FO(m): \)
\( FO(m) = FO(m) \setminus \{ o \} \)
endcase
endfor
end

Figure 3.6 : Local method analysis for object inlining

passing inlined objects to methods whose corresponding arguments can be inlined. Using
the Visitor pattern [44] the expressions are traversed recursively, and if an inlined object is
not used in an allowed context, it will eventually appear as a free variable in the traversal.
and removed from the set of inlined objects.

There are a couple of important issues that the algorithm from Figure 3.4 has to address
in order to compute the required information properly. First one is the problem that program
recursion poses to the analysis.

Handling Recursion

Naïve implementation of our algorithm would result in a code that could not handle recur-
sion loops in the call graph of the program. If all the information is computed on demand
whenever a method call is encountered, a recursion loop in the call graph would cause the
algorithm to loop indefinitely.
We opted for an off-line solution to this problem. As Figure 3.4 shows, our implementation has a pre-pass which computes the local information for each method. This pass analyses the methods in the program are visited independently and computes the $LO$, $FO$ and $MOD$ sets without traversing the call graph. Figure 3.6 presents the pseudo-code for this phase of the algorithm.

Our algorithm traverses the call graph during the second (interprocedural) phase of the algorithm. Figure 3.5 presents the pseudo-code for this phase. The interprocedural traversal sets a flag $Visited$ the first time a method call is visited in the call graph traversal. If the algorithm encounters the same method again, it will only use the currently computed information for the method. This ensures that each method is visited exactly once, regardless of the shape of the call graph. The algorithm then iterates this second phase until a fixed point is reached on the $LO$, $FO$ and the $MOD$ info.

**Handling Arrays of Objects**

Another concern for the safety analysis are arrays of objects. Arrays are crucial in scientific programming since practically all scientific programs involve some usage of single or multidimensional arrays. To enable scientific programming in polymorphic, object-oriented style, special attention has to be given to arrays. This section describes our treatment of the arrays in scientific programs.

In addition to safety conditions described in the previous section, an array of objects has to satisfy some more in order to be object inlined:

1. All the objects in the array have to have a single concrete type;

2. There is no aliasing between different parts of a multidimensional array;

3. Multidimensional arrays are rectangular.

These constraints can be relaxed using regular section analysis [48] techniques. Only regular sections of an array would be required to satisfy these conditions, and the analysis would have to prove that those sections are used consistently throughout the program. This
would allow even more arrays of objects to be inlined, but falls outside of the scope of this thesis and into future work.

3.2.3 The Algorithm For Object Inlining

Figure 3.7 shows the pseudo-code for the algorithm that performs object inlining transformation of the program. The algorithm operates in the same way as the analysis algorithm from Figure 3.4. It starts with a set of entry methods, performs the transformation on them and recursively traverses the call graph by calling the single method transformation algorithm from Figure 3.8. The algorithm from Figure 3.8 contains the main steps of the object inlining transformation.

To simplify the discussion of the algorithm, we will treat the assignments of the result of a method call to an inlined object as a method call that has an additional argument to which the cloned methods assigns the return expression. The safety analysis described in the previous section ensures that the return expression (or variable) of the cloned method is a new instance. For example, the method call to $o = \text{foo()}$ where $\text{foo()}$ is defined as

```java
public O foo(){
    //some computation
    return new O(args1);
    ...
    O l = new O(args2);
    ...
    return l;
}
```

will be treated as

```java
public void foo(){
    //some computation
    o = new O(args1);
    ...
    o = new O(args2);
    ...
    //return l; removed!
}
```
inputs: IN (Program P)
(Setting O of all the global objects and arrays of objects to inline)
(Setting LO(m) of all local objects to inline in all methods m)
(Setting Entry of all entry methods in P)
(Setting MOD(m) information for all methods m)

outputs: OUT (Program P with all the objects from O and LO(m) inlined)

begin
  for ∀o. o ∈ O
    remove o from its class C
    for ∀f. f is a field of o. of type T
      add a field o_f of type T to C
  endfor
endfor

for ∀m in the program
 Visited(m) = false
endfor

for all methods m. m ∈ Entry
  M(m) = {}
  ObjectInline(m)
endfor

end

Figure 3.7: Object inlining algorithm

Of course, this is not a legal Java code, since it contains a reference to a free variable o. This example only illustrates the idea of how are the assignments to inlined objects treated, the discussion below explains how does the object inlining algorithm further transform this example to generate legal and correct Java code. If the method f00 contains a locally inlined and instantiated object 1 that is sometimes returned as a result of f00, this local object is renamed to the variable that is receiving the result of the method call, as in the example above. Object inlining converts all the return statements into assignments to
the return variable, through formal-to-actual argument mapping $M(m)$, computed in the algorithm on Figure 3.8.

From this point, to simplify the discussion, when the text refers to an object that is passed to a method, it will mean one of the following:

- the object is passed as a regular argument to the method.
- the object is passed as a this argument to the method, i.e. the method is called on the object in question, such as $o\cdot foo()$, or
- the result of the method is assigned to an inlined object, as in the example above.

To further simplify the discussion, this section will discuss only method calls to methods with a single argument. Naturally, the complete implementation handles all the method calls with arbitrary number of arguments. The discussion below easily extends to method with multiple arguments, when a context “method is cloned under such and such conditions” is replaced with “for each argument position, method is cloned under such and such conditions”.

If more than one class in the program contains inlined objects of type O, object inlining clones the calls that pass the inlined object to the method based on the type of the container class. It will generate a method that accepts the container object as an argument instead of the inlined object. For example, object inlining will transform the call to $foo(o)$ into a call to $o\cdot foo(Co)$, where Co is the object of type CO that contains the inlined object o.

Object inlining will create a clone $c\cdot foo(CO, Co)$ of the original method $foo(O, arg)$ with all the references to fields of o (such as $arg.x$) changed into references to inlined fields contained in Co (such as $Co\cdot o_x$).

If a method does not modify the fields of the inlined object that is passed to it, the object inlining algorithm does not need to create a clone based on the containers type. The analysis algorithm from Figure 3.4 computes this information and stores it into the $MOD(m)$ set. In this case, object inlining transforms the method call into a pass-by-value call by passing the fields of the object instead of the object itself. For example, it
function ObjectInline:
inputs: IN (Method m to transform, sets O, LO(m) and FO(m))
(Mapping M(m) of formal to actual arguments of m)
outputs: OUT (m with all objects from O and LO(m) inlined)
begin
  if (Visited(m)) return. else Visited(m) = true
  for all expressions e in m, in evaluation order
  case of e
  declaration D of local object o, o \in LO(m): remove D from the code
  for (\forall f. f is a field of o, of type T): add a declaration of o.f of type T
  method call n(args):
    signature = signature of n
    for \forall o, o \in args, o is an object
      let f be the formal argument of n corresponding to o
      if (f \in FO(n), f \notin MOD(n))
        for (\forall fld, fld is a field of o, of type T): add T to signature.
        continue
      endif
    if (o \in FO(m)): add M(m, o) to signature. add (f, M(m, o)) to M(n)
    if (o \in LO(m)): add o to signature. add (f, o) to M(n)
    if (o \in O)
      let Co be the container of o, of type CO
      add Co to signature. add (f, Co, o) to M(n)
  endif
  if (o is an array access, o \in O \cup LO(m) \cup FO(m))
    for (\forall i, i is the index of the array access of o): add int to signature.
    clone = Hashtable(signature)
    if (clone == null)
      clone = Clone(n, signature). Hashtable(signature) = clone
      ObjectInline(clone)
    else convert the call to n to call to clone
  endfor
  assignment o = n(args), o \in O \cup LO(m): process the call n(o, args), as above
  field access o.x, o \in FO(m) \cup O \cup LO(m):
    if (o \in FO(m)): let a = M(m, o). convert o.x to a.x
    else convert o.x to o.x
  endcase
  endfor
end

Figure 3.8: Object inlining algorithm for a single method
will transform a method call to `foo(o)`, where object `o` contains integer fields `x` and `y`, into a call to `foo(o.x, o.y)`. In this case, object inlining will create only one clone `foo(int x, int y)` of the method `foo(O arg)` for the whole program. It will transform all the references in the body of the clone from `arg.x` and `arg.y` into `x` and `y`, respectively. The algorithm will change all the method calls to `foo` in the program that pass an inlined object as an argument to this *pass-by-value* form. This single clone accepts all inlined objects, regardless of whether they are local, global or elements of local or global arrays of inlined objects.

When a locally inlined object is passed to a method that *does* modify the object's fields, object inlining will create a new clone. For example, it will transform the call to `foo(l)`, where `l` is the locally inlined object, and the method is declared as `foo(O arg)` into a call to `l_foo()`.

Cloning will transform all the references in the body of the clone from `arg.x` and `arg.y` into `l.x` and `l.y`, respectively. The generated clone is not a valid Java method since it contains references to free variables `l.x` and `l.y`. Thus, this clone exists only in the intermediate form, the final pass of the algorithm inlines all the calls to it.

When an element of an array of inlined objects is passed to a method, object inlining will create a clone with additional arguments for passing array indices. It will also convert the old method call into a call to the newly created clone. For example, it will convert the call

```
foo(o[idx1][idx2]);
```

to a method `foo(O arg)` into a call

```
o_foo(Co, idx1, idx2);
```

to a clone `o_foo(CO Co, int ix1, int ix2)`. This process will transform all the references to fields of the formal argument `arg` (such as `arg.x`) into references to the inlined fields of the array element (`Co.o_x[idx1][idx2]`).

The example above shows the treatment of method calls when an element of an array of globally inlined objects is passed to a method. Object inlining applies a very similar
transformation when an element of an array of locally inlined objects is passed to a method, except that it will not generate the container argument. It converts the call

```c
foo(l[idx1][idx2]);
```

to a method `foo()` into a call

```c
l_foo(idx1,idx2);
```

to a clone `l_foo(int ix1, int ix2)`. All the references to `arg.x` in the clone turn into `l_x[ix1][ix2]`. Again, the generated clone can exist only temporarily since it has references to free variables, and object inlining will inline all the calls to it.

Let us show, as a final example, a method call that illustrates all of the issues discussed:

```c
foo(c1.o1, c2.a1[idx1], l1, la[idx2]);
```

where `o1` is a globally inlined object inside the container `c1` of type `C1`. `a1` is a globally inlined array of objects inside the container `c2` of type `C2`. `l1` is a locally inlined object and `la` is a locally inlined array of objects. The signature of the method `foo` is `foo(O1 a1, O2 a2, O3 a3, O4 a4)`. Let us assume that classes `O1`, `O2`, `O3` and `O4` have two integer fields each: `x1`, `y1`, `x2`, `y2`, `x3`, `y3`, and `x4` and `y4`, respectively. `foo` does not modify the fields of `a3`. Object inlining will create a clone of `foo` with signature

```c
o1_a1_la_foo(C1 Cal, C2 Ca2, int a2_i1, int a3_x3, int a3_y3, a4_i1)
```

Cloning will change all the references to fields of the arguments of `foo` into corresponding inlined data references: `a1.x1` into `Cal.a1.x1`, `a2.x2` into `Ca2.a2.x[a2_i1]`, `a3.x3` into `a3.x3` and `a4.x4` into `la.x4[a4_i1]`, and similar for the 'y' fields of inlined objects. The algorithm will always inline the converted method call, since it involves a locally inlined object (`la`) that is passed by reference to the clone.

To prevent redundant cloning of the methods (and to ensure the termination of our algorithm), our implementation stores all the intermediate clones into a hash table indexed
by the method signature which is described above. When a need for a particular clone arises, it first checks the hash table to see if the method has already been generated. It changes all the method calls to call the corresponding clones. Then, in a separate pass over the code, it inlines all those method calls that have to be inlined (those that involve local objects that are not passed by value), those method calls to a clone that has only one call site (improving performance and leaving the code size unchanged) and those method calls to a clone that has more than one call site, but the heuristics have shown that they are profitable to inline despite the code growth. As a final step, only the clones that still have call sites associated with them (i.e. that are not inlined everywhere they were called) are inserted in the code.

Whenever a new clone is created, object inlining recursively traverses its body with the knowledge of the bindings between the formal arguments of the clone and the actual argument that triggered the cloning. This ensures that the algorithm correctly transforms the subsequent method calls within the body of the clone where a formal argument is passed to the callee. It will clone the callee (if necessary) based on the info for the actual argument passed to the clone. The example on Figure 3.9 illustrates this case.

Figure 3.9a shows the original code, that has a class Example containing an object o of type Complex and a couple of methods foo and goo that perform simple operations. After inlining the object o and cloning the call to foo, object inlining will generate the intermediate code from Figure 3.9b. This code contains the clone o._foo. During the transformation of the method o._foo, object inlining will encounter the call to goo. This call involves one of the formal arguments of the original method foo that has been object-inlined, so object inlining has to create a clone of goo as well. Figure 3.9c shows the code that object inlining generates after cloning the call to goo inside o._foo. In addition to the code from Figure 3.9b, it has a method o._goo. Methods o._foo and o._goo might get inlined in the later phase of the algorithm, while foo and goo might get eliminated (if they are not called from anywhere else), leaving only a straight-line code.
class Example{
    float o_r, o_i;
    ...
}

class Example{
    float o_r, o_i;
    ...
    o_foo(this);
    foo(Complex o){
        ...
        o_r++;
        goo(o);
    }
    foo(Complex o){
        ...
        o_r++;
        o_foo(Example Co){
            ...
            goo(o);
            Co.o_r++;
            o_goo(Co);
        }
    }
    foo(Complex o){
        ...
        o_r++;
        o_foo(Example Co){
            ...
            goo(o);
            Co.o_r++;
            goo(Complex o){
                ...
                o.i++;
            }
        }
    }
    goo(Complex o){
        ...
        o.i++;
    }
}

a. Original code
b. After cloning the call to foo()
c. After cloning the call to goo()

Figure 3.9: An example showing the process of recursive method cloning

The handling of the recursion within object inlining raises an important concern: algorithm termination. We need to prove that the described algorithm will not generate new clones indefinitely and that it will eventually terminate.

**Theorem 3.2.1** The algorithm from Figure 3.7 terminates.

**Proof:** The algorithm from Figure 3.7 creates at most one method clone per call site. recursively traverses it, and then continues with the transformation of the code after the call site. Each new clone is traversed exactly once. To show that the algorithm will eventually terminate, it is sufficient to show that there is only a limited number of possible clones that can be created. Let \( m \) be the number of methods in the original program, and \( arg \) the
maximal number of arguments of those methods. A method can be cloned on a particular argument position at most $|O| + \sum_{i=1}^{m} |LO(i)|$ times. So the total number of clones is at most:

$$m \left( |O| + \sum_{i=1}^{m} |LO(i)| \right)^{arg}$$

Since this is a finite value for any given $m, O, LO$ and $arg$, the algorithm will generate a finite number of clones and terminate after that.

### 3.2.4 Performance Benefits Of Object Inlining

Object inlining has several significant impacts on the program performance:

- **Eliminating indirection.** Globally inlined objects have one less level of indirection, since one only needs to traverse the reference to the container to get to the inlined data. In the original program, accessing the data requires traversing both the reference to the container and the reference to the object. Locally inlined objects completely eliminate the indirection — all the data is available in the local variables.

- **Improving memory hierarchy performance.** Globally inlined object and its container are allocated in a continuous space in the heap, thus improving the spatial locality. Moreover, arrays of inlined objects have by far the biggest impact on memory hierarchy performance. Object inlining transforms arrays of objects, where the data is scattered throughout the heap, into arrays of inlined objects, where the data occupies continuous memory locations. This is especially important in high performance scientific applications that depend significantly on memory hierarchy performance.

- **Improving memory management performance.** By replacing objects with their inlined data, object inlining reduces the total number of objects in the program. There is a smaller number of bigger objects throughout the execution of the program. This transformation thus reduces the amount of work for the dynamic memory allocator.
and the garbage collector. Object inlining completely eliminates the need for dynamic memory management of locally inlined objects, since it essentially transforms them from heap-allocated into stack-allocated.

- **Reducing the memory footprint of the program data.** Object that contains fields `a` and `b always` requires more memory for storage than just what fields `a` and `b` occupy. Every object has memory requirements in addition to data storage: usually a pointer to the class object, hash code, locks, etc. The amount of memory overhead really depends on the implementation, but it is at least a couple of words. Relative impact of this overhead is greater on small objects: an array of `int` occupies a lot more memory than an array of `Integer` of the same length. This benefit of object inlining is especially important for scientific programs that operate on large data sets.

- **Improving the impact of local optimizations.** By inlining objects, object inlining enables (and requires) more method inlining, which in turn increases the average size of a method and of a basic block. This gives more "operating space" to classical local compiler optimizations and increases their effectiveness. For example, a method call within a loop might prevent loop invariant code motion in the original program, while the transformed loop with the method call inlined allows for the unrestricted code motion. Exceptions, as it will be discussed in greater detail in Chapter 4, significantly limit the effect of code motion of field accesses. By inlining local objects, our transformation converts some of these field accesses into local variables which do not pose a problem to code motion.

- **Increasing the code size.** This is a negative performance impact of object inlining. In fact, na"ïve object inlining is not a very practical solution since there is a potential for exponential code growth. In addition to that, some JIT compilers might encounter problems compiling huge methods that can result from extensive inlining, and revert to interpretation of the code (as we have experienced with the JDK 1.2 production release VM for Windows platform), which eliminates the effectiveness of the opti-
mization. A careful application of object inlining with limited total code expansion and moderate method size increase gives the best practical results.

These qualitative evaluations need to be validated in practice. Next section describes some experimental results that confirm the weight of these claims.

### 3.3 Experimental Results

This section presents the performance results for experiments that validate the claims from the previous section, and show that object inlining can virtually eliminate the overhead introduced by object-oriented, polymorphic style of programming in high performance scientific applications.

<table>
<thead>
<tr>
<th></th>
<th>jdk 1.1.5 interpreter</th>
<th>jdk 1.2 JIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fortran style</td>
<td>“Lite” OO style</td>
</tr>
<tr>
<td>dpodi</td>
<td>12.736</td>
<td>14.315</td>
</tr>
<tr>
<td>dgefa</td>
<td>3.226</td>
<td>3.511</td>
</tr>
<tr>
<td>dgsef</td>
<td>4.079</td>
<td>4.640</td>
</tr>
<tr>
<td>dqrdc</td>
<td>21.197</td>
<td>25.139</td>
</tr>
<tr>
<td>dqrsf</td>
<td>14.504</td>
<td>15.757</td>
</tr>
<tr>
<td>dsvec</td>
<td>9.008</td>
<td>15.439</td>
</tr>
<tr>
<td>average</td>
<td>1</td>
<td>1.202</td>
</tr>
</tbody>
</table>

Table 3.1 shows the execution times for our standard collection of OwlPack benchmarks. The times for Fortran style, “Lite” object-oriented style and full object-oriented
style are the same that Chapter 2 reports, and we are repeating them here for comparison purposes. Fourth and eighth column show the execution times for the full object-oriented style version of OwlPack, optimized with the JaMake compiler. We have performed the class specialization of the OO code by hand, while the object inlining is performed automatically by JaMake.

The last row in the table shows the average execution times of "Lite" OO, full OO and optimized OO versions of the code, relative to the Fortran style version on the same platform. The table shows that on the average, the "Lite" OO version is 20-25% slower than the Fortran style version.

Optimized object-oriented version shows tremendous improvement over the object-oriented version. Optimized code is 5 to 10 times faster than the original, around 6 times faster on average when interpreted and around 9 times faster when executed on a VM with a JIT.

Even though the optimized object-oriented version approaches the Fortran style and "Lite" OO style in performance, there is still a significant performance difference. Optimized code is still two times slower than the Fortran style code on a JIT VM.

There is a very simple reason for this disparity. To ensure the correctness of the transformation without complicating the implementation of the object-inlining transformation, the JaMake compiler inserts many extraneous instructions in the code. It handles method inlining by inserting copies of the actual arguments into formal arguments of the method, followed by the method body. It handles returns by assigning the value to the return variable and breaking out of the method body. This results in many unnecessary instructions that should be very easy to eliminate in a back end of an optimizing compiler.

The following example illustrates this problem. Let us look at a simple statement from the object-oriented version of the OwlPack:

```java
LDouble t = ooAM.getElem(4,4);
```

Both `t` and `ooAM` are object inlined. `t` is of concrete type `LDouble`, and `ooAM` is of type `NFull`, specialized for `LDouble`. Method `getElem(i,j)` returns a copy of the
object in the i-th row and j-th column of the matrix. The JaMake compiler will generate
the code on Figure 3.10 from this statement.

double t_data = 0.0;
double[][] R10 = ooaM_Mat_data;
int R11 = ooaM_cols;
int[] R12 = ooaM_pivot;
int R13 = ooaM_rows;
int R14 = 4;
int R15 = 4;
INL_4: {
    CL16: {
        double R17 = R10[R14][R15];
        INL_5: {
            CL18: {
                double R19 = R17;
                INL_6: {
                    t_data = R19;
                };
                break CL18;
            }
        break CL16;
    }
}
break CL16;
}

Figure 3.10: Code for statement LDouble t = ooaM.getElem(4,4). after object inlining

Compiling this code with javac -O generates 24 Java bytecode instructions that occu-
py 40 bytes. It is obvious that application of a collection of simple, straightforward classical optimizations, such as copy propagation, dead code elimination, constant propagation
and dead branch elimination would reduce the above code into a single statement:

double t_data = ooaM_Mat_data[4][4];
Compiling this code with `javac -O` generates only 6 Java bytecode instructions that are 7 bytes long. Unfortunately `javac` even with the optimizations turned on does not recognize these optimization opportunities. Even the JIT compilers, as evidenced by Table 3.1 do not fully eliminate this overhead, although they are doing a better job than just the straightforward interpretation.

Although the Lite OO version of OwlPack has an enormous performance advantage (up to two orders of magnitude) over the OO version on current Java platforms, the polymorphic object-oriented style results in a code that is much smaller in size - a factor of 4 for our implementation of OwlPack. In addition it is easier to maintain because algorithmic changes need only to be made once. Compiler technologies that we are proposing in this thesis are already able to automatically transform the OO style OwlPack code into a code that comes within a factor of two of the Lite OO version in performance. With a full range of local optimizations performed on the output of JaMake, we expect considerable reduction in execution times and code size, rivaling and even bettering those of the Lite OO version of the code.
Chapter 4

Enabling Classical Optimizations in Java

For Java to be even considered for high-performance computing, Java compilers have to implement some of the well-known classical optimizations. The transformations described in Chapter 3 convert a high-level, object-oriented Java program into a program that resembles a procedural style of programming that one might use in Fortran or C, but that is not enough. State-of-the-art classical compilers for C, C++ and Fortran all implement a wide array of classical optimizations. Loop invariant code motion, strength reduction, constant propagation, dead code elimination, and partial redundancy elimination are only a few examples of code transformations performed in the classical compilers. These optimizations yield significant performance improvements for scientific programs and must be implemented in any compiler that seeks to be competitive with the current compilers for procedural languages.

Unfortunately, Java presents some formidable obstacles to the implementation of some of these classical techniques. Presence of the exception mechanism in Java may prevent code motion techniques from moving the code across the points that can raise an exception. Most of the optimization techniques mentioned above use the Static Single Assignment (SSA) [16] form of the program. Code motion techniques, as well as copy folding, can force variable renaming during the conversion of the SSA into the Control Flow Graph (CFG), including the field names of the classes in the program. This field renaming might be unacceptable for some Java programs since it would change the interface of the generated program (the generated bytecode is a legitimate unit for further compilation), and it might induce incorrect program behavior in some programs that use multithreading and synchronization. The classical SSA-to-CFG conversion technique [16] requires copy in-
sertion all $\emptyset$ nodes in the SSA program; the inserted copies can often outweigh the benefits of the applied optimizations.

This chapter improves the optimization of high performance scientific Java programs by addressing two of the most important issues in enabling the implementation of classical optimizations: presence of exceptions and SSA reconstruction. Section 4.1 introduces a technique called Exception Hiding that extends the effectiveness of code motion optimizations by expanding the segments of the code in which the code motion can be applied freely. Section 4.2 presents a novel SSA-to-CFG conversion algorithm that inserts a much smaller number of copies than the classical algorithm and performs this in practically linear time. This algorithm is more precise (it eliminates more copies) than the standard copy insertion algorithm [16] (which also runs in linear time). It also eliminates a similar number of copies as the interference-based algorithm does but it has a better asymptotic complexity.

Next section describes the exception hiding transformation and its effect on code motion in Java.

### 4.1 Exception Hiding

One of the main reasons for poor performance of Java systems relative to the more traditional Fortran and C/C++ implementatons is the presence of the exception mechanism. While this mechanism greatly improves the language by enabling the programmer to handle errors and exceptional situations in the program in an elegant and non-intrusive way, a liberal use of the exceptions for programming purposes can greatly interfere with the optimizations the compiler is trying to perform. While it is hard to imagine and even harder to justify the usage of exceptions other than that for unusual (erroneous) situations in the program, compilers nevertheless have to take the conservative route and account for the possibility that the programmer uses the exception mechanism for control flow. Some values in the program can escape their context and could be reused somewhere else in the program through exception raising and handling.
The exception mechanism in Java presents an obstacle to compiler optimization regardless of which framework is used in the compiler implementation. This section analyzes the problems that arise due to Java’s exception model and propose some solutions that reduce the impact that possible exceptions have on traditional compiler optimizations.

Java has an exact exception model [45]. If an exception occurs, the visible state of the program must be as if all the instructions in the original source code before the one causing the exception, and none of the instructions after it, have been executed. This behaviour must be preserved regardless of the optimizations performed in the compiler. In another words, if an exception occurs in the program, a user should not be able to tell by examining the state that optimizations have been performed.

This restriction greatly reduces the freedom of the compiler to move code within the program. None of the instructions that change the user-visible state of the program can be moved across an instruction that can cause an exception. In addition to data dependency constraints, code-motion optimizations must observe the constraint that those instructions that can trigger an exception cannot be interchanged or moved arbitrarily around the control flow graph.

When analysed independently, most Java instructions can cause an exception. For example, a field access \texttt{x.foo} can potentially induce an exception since the variable \texttt{x} might have a value \texttt{null}. The naive implementation of code motion optimizations—marking all of the instructions that can raise an exception and prohibiting the optimizer from moving instructions past a marked instruction—would therefore virtually eliminate all the possibilities for performance enhancement by code movement.

Fortunately, although most Java VM instructions can cause an exception, in the normal program execution most of them will not do so. In our example, if \texttt{x} has been assigned a non-null value someplace before the field access \texttt{x.foo}, the compiler can conclude that this field access will not raise a null dereference exception. By exploiting the fact that most instructions will not raise an exception during the program execution, it is possible to achieve most of the performance of program written languages without exceptions (e.g.,
Fortran). If the compiler can prove that an instruction will not cause an exception in the given program context, optimizations can freely move the code around it.

Our approach uses the same set-based analysis approach [42] that was used for inferring the concrete types in Chapter 3. This analysis identifies the instructions that will not raise an exception by proving the facts about the instruction in question. These instructions are then marked as "safe". Consecutive "safe" instructions form "safe zones" of the code, in which the code-motion optimizations can operate freely. Instructions that cannot be proven to be safe are marked as "unsafe" and present barriers for code motion.

One can think of the original program with no analysis as having many barriers and very few safe zones in the code. An analysis then eliminates some of these barriers, thus increasing the size of safe zones. Unfortunately, even an extensive (and expensive) whole-program analysis such as set-based analysis will leave many barriers in the program. What is worse, even if there are only a few barriers in the program, they might be at the most unfortunate places, preventing the code-motion optimizations from performing the most profitable transformations. For example, a single barrier in the loop nest will prevent loop invariant code motion.

The Exception Hiding technique presented in this section takes the relaxation of barriers one step further by using code transformations that artificially create "safe zones" in the parts of the code that are the most profitable for code-motion optimizations.

As already noted, this approach is completely dependent on the optimizations attempted. The basic idea behind these transformations is to insert in front of the safe zone we are trying to create some instructions that simulate the exception behavior of the barriers within the zone. This eliminates the barriers within the zone and enables free code movement inside it. The instructions that are inserted in front of the safe zone can be

- Explicit tests for exception conditions.
- Useful instructions from some other parts of the code,
- Dummy instructions to trigger the exceptions.
A. A loop example

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

B. Code from A after loop invariant code motion

```java
for(i = k; a[i] < 0; i++){
    sum = sum + a[i];
}
```

Figure 4.1: An example of loop invariant code motion, showing the shorter loop body after code motion

These techniques are easiest to explain in an example. Consider the code fragment in Figure 4.1.

Since \( k \) is loop invariant, the first assignment in the loop body is loop invariant, so loop invariant code motion would seek to move that assignment out of loop. However, since the loop header contains a barrier (let us assume that the analysis has been able to prove that \( a[] \) is not null, but not that the value of \( k \) at the entry to the loop is less than the length of \( a \)) the optimizer cannot move this assignment. If the first iteration of the loop header caused an exception, the value of \( x \) would not have changed in the original program, but it would be changed in the optimized program. If there is an exception handler that catches the raised \( ArrayOutOfBoundsException \) exception and tests the value of \( x \), the program will behave incorrectly.

On figure 4.2, run-time tests are inserted that guard the entire loop. These tests insure that no exceptions can occur within the loop, effectively moving the code-motion barriers from the loop header into the guarding tests. This technique performs a cost-estimate heuristic that ensures that the added cost (both in execution time and in code size) of guarding this loop with run-time tests for exceptions is warranted by the performance benefit of the loop invariant code motion.
if (a != null && k < a.length) {
    for (i = k; a[i] < 0; i++) {
        this.x = 3*k;
        sum = sum + a[i];
    }
}

Figure 4.2 : Removing barriers by inserting run-time tests

In this particular case, we can solve the problem even more easily by simply peeling off the first iteration of the loop, as shown in Figure 4.3. There is no added cost in execution time, but the size of the transformed code is roughly doubled. In both approaches, after the code transformation, loop invariant code motion can move the assignment to this.x outside the loop. In the latter case of removing barriers using loop peeling, this assignment is later completely eliminated by a dead code elimination pass.

Another approach is to insert dummy instructions that simulate the exception behaviour. As illustrated in Figure 4.4, a dummy assignment int dummy = a[k] is inserted in front of the loop. This code will raise the ArrayOutOf Bounds exception if it was going to occur in the original code. This technique can sometimes be less expensive than inserting the explicit tests, but a careful implementation of the dead code elimination is necessary to ensure that the inserted instructions do not get removed by the dead code elimination pass.

Exception hiding greatly improves the effectiveness of code motion optimizations. However, code motion in SSA form requires [16] that the conversion process of the SSA form back into the CFG form inserts copies for all the o nodes in the SSA form. Without an efficient algorithm for minimizing the number of inserted copies, the benefits of code motion in combination with exceptin hiding would be greatly reduced. The next section addresses this problem by introducing and describing such an algorithm.
i = k;
if (a[i] < 0){
    this.x = 3*k;
    sum = sum + a[i];
}
for(i = k+1; a[i] < 0; i++){
    this.x = 3*k;
    sum = sum + a[i];
}

Figure 4.3: Removing barriers by peeling the first iteration of the loop

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

Figure 4.4: Removing barriers by inserting dummy instructions
4.2 Reconstruction of the Static Single Assignment Form

This section presents a new algorithm for conversion of the SSA form into the CFG that inserts a much smaller number of copies for $\phi$ node instantiation than does the traditional algorithm. First, it defines the SSA form and the CFG-to-SSA conversion process, then it presents the theoretical foundation for the conversion algorithm, describes some implementation peculiarities followed by the description of the algorithm itself and its asymptotic complexity.

4.2.1 Static Single Assignment Form

SSA form, the standard intermediate representation of programs in a compiler, enables efficient implementation of many classical code optimization techniques [34, 12]. By converting the Control Flow Graph(CFG) representation of a program into a form that has each program variable assigned to once and exactly once in the program text, compilers are able to assess many program properties just from the variable names and locations. The SSA versions of many classical optimization algorithms are both more efficient and more easily implemented than their non-SSA equivalents.

An example of CFG-to-SSA conversion is given in Figure 4.5. All variables are renamed to "original name + suffix" form. The key to enabling the construction of the SSA form is the introduction of a special form of node, called the $\phi$ node. $\phi$ nodes are new variable assignments that have a special ($\phi$) function as their right hand side. $\phi$ nodes are inserted at the join points in the CFG, where joining of two different values of an original variable of the CFG is needed. For example, variable $x$ on Figure 4.5a is assigned twice in the program. When this is converted into the SSA form, these two assignments generate two distinct variables, $x_1$ and $x_2$. However, these two values flow into a single variable $x$ after the join point on Figure 4.5a, and $x$ is used later on.

To enable this kind of behavior, a $\phi$ node $x_3 = \phi(x_1, x_2)$ is introduced. $\phi$ functions act as if they "know" where the control flow in the program came from and return the value on the corresponding argument position. Note that the $\phi$ functions are abstractions
that incorporate knowledge of the flow of control in the program; one cannot write an actual subprogram (for example \texttt{int phi(int x1, int x2)}) that behaves as the $\sigma$ function does. A $\sigma$ node behaves as if there were "invisible" copies of the arguments to the $\sigma$ function to the target variable on the corresponding incoming edges, and all the copies of all the $\sigma$ nodes of a basic block are done in parallel.

![Diagram](image)

\textbf{a. Original Code} \hspace{2cm} \textbf{b. SSA Form}

Figure 4.5: CFG to SSA conversion

The CFG-to-SSA construction algorithm is described in great detail elsewhere [16]. It relies on constructing the dominator information for the basic blocks in the program and inserting $\sigma$ nodes for variables defined in the block into the dominance frontier of that block.

One problem with the SSA form is that its conversion back into the CFG is not always trivial. If some sort of code motion or copy folding optimization is performed on the program in the SSA form, simply renaming the subscripted variables back to their original names and removing the $\sigma$ nodes might result in an incorrect program. For example, in Figure 4.5b, if some form of partial redundancy elimination [60, 15] moves the assignment to $x_2$ above the test of $p$, then $x_1$ and $x_2$ would have live ranges that overlap and cannot be simply renamed back to $x$. 
If a code motion or copy folding optimization has been performed on the SSA form, the algorithm that converts SSA into CFG proceeds with the nontrivial case. Briggs et al. describe this algorithm in great detail [16]. It consists of systematically replacing the $\circ$ nodes with copies of the arguments of the $\circ$ function to the target variable on the corresponding incoming edges, with some special cases that have to be taken into account. These special cases, the "lost copy" problem and the "swap" problem, are discussed in section 4.2.3.

The problem is that the traditional CFG-to-SSA conversion algorithm described in literature [34, 16, 12] inserts all the copies for all arguments of all $\circ$ nodes in the program, which can be far more than is necessary for correct program transformation.

This section presents an algorithm that inserts a small number of copies necessary for correct $\circ$ node instantiation. It does not achieve the absolute minimum number of copies in some cases, because it has to use some heuristics when choosing which variables to insert the copies for. In practice, however, the number of copies inserted is very close to the minimum, and far smaller than with the traditional algorithm.

For simplicity, we will assume that the original program is converted from the CFG form into the SSA form while folding copies on the fly [16] and that no other optimizations are performed on the SSA. This SSA form is then converted back into the CFG form using the algorithm presented here. Any other optimizations can be performed using the following sequence: CFG-to-SSA conversion, optimization, SSA-to-CFG conversion using the traditional approach, CFG-to-SSA conversion with copy folding, SSA-to-CFG conversion using our approach. A production compiler implementation should eliminate unnecessary conversions and implement our SSA-to-CFG conversion algorithm in a single pass regardless of the optimizations performed.

4.2.2 Theory

This section proves some properties of the programs in SSA form, creating a basis for the development of the copy insertion algorithm.
For presentation simplicity, we will consider only variables that are live across basic blocks. Variables that are defined in a basic block and used only within that block can be handled trivially with a separate pass that folds copies locally within a basic block.

**Definition 4.2.1** *Strict program* is a program in which for every variable \( v \) in the program, and for every use of \( v \), every possible path from the root of the CFG of the program to that use of \( v \) goes through at least one definition of \( v \).

We will consider only strict programs in our further discussions, which may seem like an important restriction on the first glance. This is not an issue in Java, since all legal Java programs are strict. In other languages, programs that are not strict either rely on uninitialized values of variables to produce results or have dead code that uses uninitialized values. In either case, such programs are not interesting from a compiler research perspective. Therefore, the results of this section are applicable to the compiler technology in general, not only to Java compilation.

**Definition 4.2.2** A *regular program* is a program in the SSA form, where \( o \)-nodes are modeled as copies on the corresponding edges, and where definition of every variable \( v \) dominates all its uses.

The next theorem establishes a link between the variables in a program and the dominance information.

**Theorem 4.2.1** SSA conversion of a strict program results in a regular program.

*Proof.* Let us look at the SSA form of a strict program. Let \( S \) be the starting point of the program, \( v \) the name of a variable in the original strict program, \( v_i \) the name of a corresponding SSA variable, \( d \) a definition of \( v \), and \( u \) use of \( v_i \). We need to prove that \( d \) always dominates \( u \). There are three possible cases:

1. \( d \) is a non \( o \)-node assignment, and \( u \) is in a non \( o \)-node expression. This use had to be in the original strict program (since SSA construction only adds \( o \)-nodes). By
definition of a strict program, all paths from $S$ to $u$ have to contain at least one
definition of $v$. Since SSA form requires a single static assignment, $d$ is the only
assignment to $v$; therefore all paths from $S$ to $u$ go through $d$, so $d$ dominates $u$.

2. $u$ is in a $\sigma$-node. According to SSA construction algorithm, $\sigma$-nodes are inserted in
the dominance frontier of the assignment; therefore $d$ dominates the edge by which $u$
flows into the $\sigma$-node. Since $\sigma$-nodes are modeled as copies on the incoming edges,
$d$ dominates $u$.

3. $d$ is a $\sigma$-node, and $u$ is in a non $\sigma$-node expression. Let us assume that $d$ does not
dominate $u$. That means that there is a path $P$ from $S$ to $u$ that does not go through $d$. By definition of a strict program, there has to be a definition $d_k$ of a unique SSA
variable $v_k$ on $P$. If $d_k$ does not dominate $u$, it cannot dominate $d$ and by SSA
construction there has to be a $\sigma$-node inserted in the dominance frontier of $d_k$ that
defines $v_j$ ($j \neq i$, $j \neq k$), and $v_i$ would have been replaced with $v_j$. If $d_k$ dominates
$u$, $v_i$ would have been replaced with $v_k$.

**Definition 4.2.3** Two variables interfere if they are both live at some point in the SSA.

**Lemma 4.2.1** If two variables interfere, the definition point of one will dominate the defi-
nition point of the other.

*Proof:* From Theorem 4.2.1 it follows that a definition of a variable $x$ in SSA dominates its
every use. Since variable can be live only on a path from its definition to its use, it follows
that a definition of $x$ dominates every point where $x$ is live. Let $p$ be the point where both
$x$ and $y$ are live. Both definitions of $x$ and $y$ dominate $p$. Since there is only one path from
the root of the dominator tree to $p$, that path has to go through the definitions of $x$ and $y$, so
one has to dominate the other.

**Lemma 4.2.2** If two variables interfere, they will be in the live-out set of the block that
contains the dominated definition of the two.
Proof. Let \( x \) and \( y \) be variables that are simultaneously live at point \( p \), let the definition of \( x \) dominate the definition of \( y \) by Lemma 4.2.1, and let \( B \) be the block that contains the definition of \( y \). Since all the paths from the start of the program to \( p \) go through \( B \), and \( x \) and \( y \) cannot be killed from \( B \) to \( p \), and \( x \) and \( y \) are live at \( p \), by live information propagation algorithm, \( x \) and \( y \) must be live at the end of \( B \) and will be in its live-out set \( \Box \)

Lemmas 4.2.1 and 4.2.2 allow us to determine a single point in the SSA where we need to check for variable interference:

**Theorem 4.2.2** Two variables can interfere if and only if the definition of one dominates the definition of the other and they are both in the live-out set of the dominee.

Proof. The “only if” part follows directly from Lemmas 4.2.1 and 4.2.2. The “if” part is trivial: If the definition of \( x \) dominates the definition of \( y \), and \( x \) and \( y \) are live at the end of the block \( B \) that contains the definition of \( y \), then \( x \) and \( y \) interfere at the end of \( B \). \( \square \)

Now we need to define an appropriate data structure that our algorithm uses:

**Definition 4.2.4** Let \( S \) be a set of SSA variables in the program such that no two variables in \( S \) are defined in the same block. Let \( \succ \) be a strict dominance relation. Let \( v_i \) be a variable in \( S \), and \( B_i \) the block in which \( v_i \) is defined. Dominance forest \( DF(S) \) is a graph in which the nodes are the blocks \( B_i \) such that \( v_i \in S \), and there is an edge from \( B_i \) to \( B_j \) if and only if \( B_i \succ B_j \), and \( v_k \in S \), \( v_i \neq v_k \neq v_j \) such that \( B_i \succ B_k \succ B_j \).

In plain English, a dominance forest represents a mapping of a set of variables to the dominator tree, where each variable corresponds to a basic block where that variable is defined, and the paths in the dominator tree are collapsed.

The next lemma establishes that the algorithm needs to check only the edges in the dominance forest:

**Lemma 4.2.3** If a variable \( v_i \) represented with a block \( B_i \) in the dominance forest does not interfere with the variable \( v_j \) represented with a block \( B_j \) which is a successor of \( B_i \),
in the dominance forest, \( v_i \) cannot interfere with any of the variables represented by \( B_j \)'s descendants in the dominance forest.

**Proof.** Let \( v_k \) be the variable represented with \( B_k \), which is a descendent of \( B_j \) in the dominance forest. Let us assume \( v_i \) interferes with \( v_k \). From Theorem 4.2.2 it follows that \( v_i \) has to be live at the end of \( B_k \). Since \( B_i \) dominates \( B_j \), which dominates \( B_k \), \( v_i \) has to be live at the end of \( B_j \) as well. Since \( v_j \) is live at the end of \( B_j \), it follows that \( v_i \) and \( v_j \) have to interfere.

The next section describes implementation details for some unusual situations in the code and then formally presents the copy insertion algorithm and its complexity.

### 4.2.3 Copy Insertion Algorithm

One has to be careful when inserting copies for \( o \) node instantiation because naïve copy insertion may produce incorrect code. Two such problems, the “lost copy” problem and the “swap” problem, are described in detail by Briggs et al. [16] and are reiterated here. The “virtual swap” problem is a case similar to the “swap” problem for which the naïve copy insertion produces the correct result, but it has to be addressed with particular attention when attempting to produce the minimal number of copies. We will first describe the lost copy problem.

**Lost Copy Problem**

Figure 4.6 illustrates an example where naïve insertion of copies produces incorrect code. This particular case is called the *Lost Copy* problem, because the copy into \( i \) gets killed by the inserted copy. This can happen only if the critical paths are not split and if the variable that is assigned within a loop is live at the exit of the loop. The algorithm for instantiation of \( o \) nodes has to detect when a copy insertion would assign a variable that is live at the exit of the block and to save the value of the variable being assigned to into a temporary. Note that if the critical edges have been split, the copy \( x_2 = x_3 \) would have gone into the back edge instead, and then temporary insertion would have been necessary.
Figure 4.6: The “Lost Copy” problem, showing the incorrect and correct placement of copies for φ nodes.

The “Swap” Problem

Figure 4.7: Dependencies between φ nodes, showing the incorrect placement of copies for φ nodes.

When instantiating φ nodes by inserting copies, attention must be given to satisfying the dependencies between the sources and the targets of the copies. In the SSA form, each variable is defined exactly once, so there is no possibility for an output dependence to appear among φ nodes. The φ node semantic is modeled as parallel execution of copies on
corresponding incoming edges; thus all the inputs are read before any of the outputs of the \( \phi \) nodes are generated. Indeed, if the inserted copy instructions on Figure 4.8b were executed in parallel, the resulting code would be correct. This behavior can be achieved by using parallel copy instructions to execute the problematic copies on a machine that supports parallel instruction issue. However, this is not the case in general. We have to detect the dependencies in the \( \phi \) nodes and insert copies in the order that is imposed by those dependencies. The only dependencies that we have to worry about are antidependencies. This is illustrated on Figure 4.7. If a \( \phi \) node uses a name that another \( \phi \) node defines, the copy of the use has to be scheduled before the copy for the definition. On Figure 4.7, the copy \( b_2 = a_2 \) has to be scheduled before the copy \( a_2 = a_3 \) otherwise the value of \( a_2 \) that needs to be copied into \( b_2 \) would be killed by the inserted copy \( a_2 = a_3 \). A detailed algorithm is presented in [16].

These dependencies impose an order on the copies that are inserted to instantiate \( \phi \) nodes. Sometimes dependencies can create a circle which cannot be resolved by simple scheduling. Figure 4.8 illustrates such an example where naive insertion of copies produces incorrect code. We call this particular case a \textit{Swap} problem, for obvious reasons. Both
φ nodes define a name that the other φ node uses. In this case we have to introduce a temporary variable to break the cycle of dependencies and produce correct code.

In our algorithm, we handle this problem by creating lists of pending copies for each edge in the SSA. When the algorithm makes a decision to insert a copy in the SSA, it puts it in the list of pending copies instead of inserting it directly into the code. After completing φ node instantiation, the algorithm sorts all the lists to eliminate dependencies, inserting temporaries to break cycles.

The “Virtual Swap” Problem

![Diagram of code examples]

Figure 4.9 : “Virtual Swap” problem

Figure 4.9 illustrates an example where naive insertion of copies still produces correct code. However, a careful implementation is needed when the copies are inserted into the SSA in our algorithm. The naïve algorithm inserts all copies for the φ nodes, while our algorithm attempts to insert as few copies as possible. When the φ nodes are analyzed, it is determined that the variables a1 and b1 are simultaneously live at the end of the first block and cannot be folded together. The algorithm then picks one of them and inserts copies for it. The left side of the Figure 4.10 shows the example from Figure 4.9 with a1 being picked.
for copy insertion. You can see that our algorithm inserts only three copies as opposed to the naïve algorithm that inserts four.

After the copies have been inserted, the last pass of the algorithm scans through the SSA and renames the variables as needed. This is the point where some additional dependencies can be introduced and some additional copies can be needed. For example, in Figure 4.10 b, all appearances of $r2$ have been replaced with $b1$. This introduces a dependence between the first and the second $o$ node, which forces insertion of a copy on Figure 4.10 c.

The next section presents a formal description of the copy insertion algorithm, along with an informal discussion of it.

The Algorithm

In order to perform efficient copy insertion, the algorithm needs the dominance forest information. Figure 4.11 presents the algorithm for dominance forest construction. This algorithm actually creates a tree by adding a $VirtualRoot$ to the result to simplify the construction process, and then it removes $VirtualRoot$, possibly creating a forest. It starts with the depth-first traversal of the dominator tree. The algorithm then labels all nodes in
the tree with their order of traversal. On the way up in the DFS traversal, this algorithm also computes the maximum DFS number of the descendants for each particular node. This number helps the algorithm to identify the antecedent-descendent information from the dominator tree in constant time. This whole DFS numbering process is done only once for the whole SSA.

The algorithm then sorts the variables that need to be mapped to dominator tree by increasing order of their $dfn$.

```
function DominanceForest:
    inputs: IN (dominator tree DT)
            (set of variables $S$)
    outputs: OUT (dominance forest DF)
    begin
        for depth-first order over dominator tree nodes $v$
            $dfn(n) =$ the order of first visit
            $maxdfn(v) =$ the largest $dfn$ of $v$'s descendants (computed on the way up)
        endfor
        Sort $S$ by increasing $dfn$, using Radix Sort
        $maxdfn(VirtualRoot) = MAX$
        $CurrentParent = VirtualRoot$
        stack.Push(VirtualRoot)
        for all the variables $v$ in $S$ in sorted order
            while $dfn(v) > maxdfn(CurrentParent)$
                stack.Pop()
                $CurrentParent = stack.Topp()$
            endwhile
            make $v$ a child of the $CurrentParent$
            stack.Push($v$)
            $CurrentParent = v$
        endfor
        Remove $VirtualRoot$ from $DF$
    end
```

Figure 4.11 : Dominance Forest Construction Algorithm
inputs: \( IN \) (dominator tree \( DT \))

(\( P \) in SSA form)

outputs: \( OUT \) (Program \( P \) with copies inserted for \( \phi \) nodes)

begin

for all variables \( v \) in \( P \):
\[ \text{find}(v) = \{ v \} \]

for all \( \phi \) nodes \( r_i = \phi(a_1, a_2, ..., a_k) \) in \( P \)

\[ \text{find}(a_1) \cup \text{find}(a_2) \cup ... \cup \text{find}(a_k) \]

endfor

for all distinct \( \text{find}_i \),

\[ DF_i = \text{DominanceForest}(\text{find}_i) \]

endfor

for all dominance forests \( DF_i \),

for all edges \((p, c)\) in the forest \( DF_i \),

if variable \( v \) corresponding to the block \( p \) is in the \( \text{LiveOut} \) set of block \( c \)

Add copies for \( v \) to lists of pending copies in \( SSA \)

Rename uses of \( v \) in \( \phi \) nodes with the name of the copy destination

Remove \( p \) from \( DF_i \)

endfor

endfor

for all \( \phi \) nodes \( v_i = \phi(v_j, v_k, ..., v_n) \)

Rename all variables in \( \phi \) to \( v_i \)

endfor

for all lists \( l \) of pending copies in the \( SSA \)

Sort \( l \) to satisfy antidependencies

Add temporaries to \( l \) to break any cycles

Insert all copies from \( l \) in the \( SSA \)

endfor

end

Figure 4.12 : Copy Insertion Algorithm
The algorithm then enters the loop which is the core part of the algorithm. The loop traverses all the variables in sorted (increasing \textit{dfn}) order. Within the loop, a \textit{CurrentParent} variable is maintained, which holds the reference to the root of the subtree that is currently constructed. There is an edge between \textit{CurrentParent} and the current variable \(v\) if the \textit{dfn} of \(v\) is less or equal than the \textit{maxdfn} of the \textit{CurrentParent} (which means that \textit{CurrentParent} dominates \(v\)). Traversing the variables in increasing order of their \textit{dfn} ensures that no edges are inserted prematurely (if \(a\) dominates \(b\) which dominates \(c\), it will be \(\text{dfn}(a) < \text{dfn}(b) < \text{dfn}(c)\), which will ensure that only the edges \((a, b)\) and \((b, c)\) are inserted and not the edge \((a, c)\)).

Figure 4.12 presents the algorithm for inserting copies for \(\phi\) nodes. It starts by constructing sets of variables that are joined at \(\phi\) nodes. These sets represent variables that had a single name in the original CFG, or are folded together during the CFG-to-SSA conversion. If no two variables from the set interfere anywhere in the program, all variables in the set can be replaced with a single variable, and the \(\phi\) node removed.

The algorithm checks for interference by constructing the dominance forest information for each of these variable sets. As Section 4.2.2 shows, only the edges in the dominance forest need to be checked for interference. If the algorithm detects the interference, it inserts copies for the parent variable and removes it from the dominance forest.

\textbf{Algorithmic Complexity}

The dominance forest construction algorithm is linear in the size of the join set. It starts with a depth-first traversal of the dominator tree, which is a linear algorithm (linear in size of the dominator tree, but it is done only once for the whole SSA). It then uses the radix sort [32] to sort the variables in the set, which is linear as well (remember that the number of variables in the join set cannot be greater than the number of basic blocks in the CFG). Since the radix sort has a large constant, the actual implementation of the algorithm tests the size of the join set and performs a bubble sort instead if the set is small enough. Each of the variables in the set is visited exactly once in the loop (it is pushed on and popped from
the stack exactly once). So the complexity of the dominance forest construction algorithm is \( O(|S|) \).

The copy insertion algorithm begins with constructing join sets for variables in the \( o \) nodes of the graph. This can be done in a time practically linear in the number of variables in the \( o \) nodes, using the union-find algorithm [32]. The algorithm then constructs the dominance forests for these sets (which are disjoint), which is linear in the total number of variables in \( o \) nodes. For each dominance forest, the algorithm visits all the edges, which is linear in the number of nodes in the forest. At the end, all \( o \) nodes are visited and the variables that are the arguments of the \( o \) functions are renamed into a single variable name. The total complexity of this algorithm is \( O(n) \) in practice, where \( n \) is the total number of arguments in all the \( o \) nodes in the SSA.

One cannot hope to achieve better algorithmic complexity than what is presented here, since all the \( o \) nodes and all of their arguments have to be visited at least once in the SSA-to-CFG conversion.
Chapter 5

Almost-Whole-Program Compilation

This chapter presents the motivation and techniques for developing, and results of applying a new \textit{almost-whole-program} compilation framework for Java. This framework allows the programmer to write extensible object-oriented code while developing the program, and still enable extensive whole-program optimizations described in Chapter 3. Section 5.1 discusses the motivation for development of this framework. Section 5.2 describes the techniques used to achieve this goal, and Section 5.3 presents and reviews the experimental results that demonstrate the effectiveness of this framework.

5.1 Motivation

Current Java implementations adopt a simple compilation model: every class is compiled one at a time, and the compiler cannot assume anything about the other classes that are being compiled. This gives users great flexibility in the ways they can use the program — who can add, remove and replace classes in the package at will, without the need for re- compilation. Unfortunately, this system prohibits many optimizations: the compiler cannot assume anything about other classes and therefore cannot perform many interprocedural optimizations.

On the other hand, a whole-program-compilation model assumes that the whole program is available at the compile time and that the programmer will not add, remove or modify any classes without recompiling the whole program. This allows the compiler to be most aggressive and perform a wide range of interprocedural optimizations, including the object inlining and class specialization that Chapter 3 describes. Unfortunately, this model
provides no flexibility to the user, who can only run the compiled program and nothing else.

In the real world, programs are written with intentions that lie somewhere in between these two models. Some parts of the program are fixed and the user is not expected to modify them or add anything to them, while some parts should be available to the user for extending. Unfortunately, there are currently no mechanisms available to the programmers to control this: they can only write the program, compile it and distribute it to the user. Thus, they are forced to make some decisions that improve performance but hurt the overall flexibility of the program and also make some choices that amend the flexibility but impair the performance.

The source of this tradeoff problem lies in the fact that a single process (compiling the program using the Java compiler) is used for two very distinct purposes: development of the program and its distribution. These two phases of writing a program often have conflicting goals.

While developing the program, programmers want maximal code flexibility, so that they can change, extend and debug it. Also, they want their code to be easily upgraded in later versions of the program. Consequently, programmers use object-oriented techniques such as encapsulation and polymorphism when designing their systems. Most of the classes in such a system will be public (enabling their later extension), most of the methods virtual (enabling their overriding) and public.

On the other hand, the main goal during the distribution process is maximum performance. In the context of whole-program optimization, this requires that the whole program be fixed and known at compile time. In the Java framework, this requirement means that all of the classes have to be made private and final, and that all of the methods are final and/or static. Only the main class and main method of the program should be made visible to the user to allow running the program. Such formulation of the program allows the static analysis to achieve maximal effectiveness, but it overly restricts the possible uses of the program.
The programmers can choose to perform manual "optimization" of the code. When the development phase is finished and the program is debugged and ready for shipping, the programmer can manually create another version of the program that converts as many as possible of the classes and methods into final and private. This transformation enables the compiler to perform a better job of optimizing the whole program. Such a process is tedious and error-prone, even though not uncommon in the software industry. This is evidenced by the different Java JDK versions, in which many of the classes and methods that were public and extensible in the earlier versions of the JDK are private and final in the later versions. A significant amount of programmer time has been spent to make this transformation manually in order to enable javac to inline these modified methods, yet yielding only modest performance improvements.

The need for a framework that would separate the development and distribution phases of program construction is clear. A tool described in the following section automatically performs the described transformations on the program and enables the programmer to write clear, extensible object-oriented programs. After the development phase, with minimal input from the programmer, this tool converts the program into a more specialized version, with most of the classes and methods made final and inaccessible to the end user, and passes this converted program to the whole-program framework described in Chapter 3. The whole-program framework can now obtain more precise type information from the program and perform more effective object inlining.

To extend our whole-program optimizing compiler to cope with partial programs, we have developed a framework for combining optimized partial programs with other classes [19]. This almost-whole-program compilation framework includes machinery for specifying which classes in a partial program are open (extendible) and forces all supplemental classes to obey these restrictions. We have developed a technique for expressing and enforcing constraints on supplemental classes. This technique places the compiled partial program in a single Java package, marks forbidden classes as package private and relies on Java class visibility rules and run-time checks to prevent extending the forbidden classes.
Figure 5.1: Performance versus flexibility diagrams, showing the trade-off between performance and flexibility in different systems.
Figure 5.1a. shows an intuitive relation between the flexibility of the generated code and its performance. As parts of the code are fixed to enable Java source compiler to perform more optimizations, the flexibility of the code that is shipped to the end user is reduced. This reduction in flexibility is a logical consequence and generally accepted by the end user. Unfortunately, the programmer is forced to tolerate the same reduction in flexibility since there is direct mapping from the source code to the code that is deployed to the user.

If we observe the current automatic Java compilation systems, the performance versus flexibility diagram looks more like the one on Figure 5.1b. There is a quantum drop in user flexibility when the systems abandon the class-by-class compilation model and assumes the whole-program compilation. Depending on the number and effectiveness of the optimizations applied, performance of the generated code increases, but the user flexibility stays the same. The user can only run the generated program and nothing more.

The almost-whole-program compilation model attacks this problem by bridging the gap between the whole-program and class-by-class compilation models. The user's choices are more restricted in using the program package than in the class-by-class model, but are still more flexible than in the whole-program-model. Similarly, the range and effectiveness of the interprocedural optimizations that can be applied are smaller than in the whole-program model, but far greater than in the class-by-class model. Figure 5.1c. shows this relation.

### 5.2 Almost-Whole-Program Framework

This section presents a technique that enables the almost-whole-program compilation model without changing any of the rules that are currently in effect for Java systems. There are no changes to the Java language. The Java virtual machine specification is respected, thus still allowing the generated bytecodes to be executed on any Java VM. The rules for Java names visibility are unchanged. In fact, it is the rules of Java package names visibility together with the run-time checks that enable us to perform our transformations while still generating pure Java programs.
5.2.1 Private Classes

It is easy to imagine a scenario in which some of the classes in the programmer’s project are specified as public, but the programmer does not want to allow the end-user to access them. These classes are specified as public for development reasons — some classes in other packages may need to access them. If the original project is translated directly into bytecode, the end user will be able to access those classes that are specified as public, instantiate them and even extend them.

The first goal of the almost-whole-program framework is to allow the programmer to specify which classes are inaccessible to the end-user, regardless of the original class layout in the development project. Figure 5.2 shows an example of what happens to the classes the programmer has specified as inaccessible during the transformation of the development project into the distribution package.

All classes that are written as private in the original class layout remain private in the distribution package (class top.sub1.class2 in the example in Figure 5.2). The tool converts all the classes that were public in the development package and marked as inaccessible by the programmer into package private (classes top.class1, top.sub1.class1, top.sub2.class1, top.sub2.class2 in Figure 5.2). The framework renames and repackages all the classes from the development project into a single distribution package. This transformation is necessary to allow the mutual access of the classes that were originally public and in different packages but are now package private.

Only the main class of the whole development project remains public (but final, and thus non-extensive) in the distribution package. The end-user will be able only to run the program by calling Main.main() and nothing else.

5.2.2 Non-extensive Classes

The second goal of the almost-whole-program framework is to allow the programmer to specify classes the end-user is allowed to see (instantiate and run), but not to extend. This case arises in situations in which a class should be made available to the user first to in-
stinate the class and then to call its public method, but the programmer wants to preserve the class hierarchy for performance reasons. A logical approach would be to convert these classes into final; unfortunately this is not always possible. Some of these non-extensive classes may have subclasses already defined in the development project, and the back-end compiler will not allow the compilation of the subclasses if their super-classes are declared as final.

This section presents an alternative approach. JaMake converts all the classes that the developer specifies as visible but non-extensible into private. This system then creates auxiliary public interfaces that summarize the member information of the non-extensible classes.

Figures 5.4, 5.5 and 5.6 display an example of the development-to-distribution project conversion. Foo on Figure 5.4 is a class that the developer has marked as public but non-extensible. The almost-whole-program framework converts this class into the class Foo on Figure 5.6. All the public methods and fields are converted to be package private, the same as for private classes described in Section 5.2.1.
Figure 5.3: The development-to-distribution transformation of a part of the class hierarchy.

The end-users need a mechanism for instantiation of objects of type $A\text{Foo}$. Since the class $\text{Foo}$ is package private, they cannot directly create objects of type $\text{Foo}$. Instead, the abstract class $A\text{Foo}$ provides static $\text{Create}$ methods, which serve as an abstract factory [44] for obtaining new instances of $\text{Foo}$. For each constructor of $\text{Foo}$, class $A\text{Foo}$ contains a corresponding method $\text{Create}$. The only way the end-users can create new instances of $\text{Foo}$ is by calling the $A\text{Foo}.\text{Create()}$ method. Since the generated factory methods $\text{Create}$ are static, most of the current Java VM implementations are able to perform runtime inlining of these calls, making them as fast as if the original constructors were called directly.

The almost-whole-program framework ensures the non-extensibility of the classes the developer marked as non-extensible through a run-time mechanism. When converting the project from the development to the distribution form, this tool inserts run-time tests in all the constructors of the auxiliary abstract class. On the example from Figure 5.5, all the
public class Foo extends Goo{
    public T field1;
    public static T field2;
    private T field3;
    public Foo(T arg){
        /*implementation*/
    }
    public Foo(T1 arg1, T2 arg2){
        /*implementation*/
    }
    public T Meth1(T2 arg){
        /*implementation*/
    }
    static public T Meth2(){
        /*implementation*/
    }
}

Figure 5.4: Original class Foo, marked as non-extensible

Constructors have a test of whether the object being currently instantiated is of the type Foo. If not, the constructor will throw a run-time exception. This technique prevents the end-user from instantiating his own version of subclasses of AFoo. The end-user can still write and compile subclasses of AFoo, but cannot do anything else with those classes. This restriction ensures that there cannot be any instances of AFoo that are not also instances of Foo at run-time. The whole-program optimization techniques from Chapter 3 can take advantage of this fact and optimize the code that uses Foo.

The development-to-distribution conversion implementation as described above would extract all the public methods from the class that is being converted into the auxiliary abstract. It would also generate wrapper methods for all the public static methods. Such an implementation would put less burden on developers, since they would only have to specify the classes which are non-extensible, but it would be less flexible. In contrast, our
public abstract class AFoo extends Goo {
    public T field1;
    public static T field2;
    public static final AFoo Create(T arg) {
        return new Foo(arg);
    }
    AFoo(T arg) {
        super();
        if (!(this instanceof Foo))
            throw new Error("Not Foo!");
    }
    public static final AFoo Create(T1 arg1, T2 arg2) {
        return new Foo(arg1, arg2);
    }
    AFoo(T1 arg1, T2 arg2) {
        super();
        if (!(this instanceof Foo))
            throw new Error("Not Foo!");
    }
    abstract public T Meth1(T2 arg);
    static public T Meth2() {
        /*implementation*/
    }
}

Figure 5.5: Generated auxiliary abstract class AFoo

implementation allows developers to specify precisely which methods and fields should be exported, in addition to the per-class specification. This approach allows more flexibility to developers, although requiring more effort. Also, the generated code is shorter, since only the required conversions have to be performed.

This development-to-distribution conversion completely hides the class Foo from the end-users, but still enables them to create instances of it, call its public methods and access its public fields. The end-users cannot extend the class Foo, since it is not visible outside the generated package. They can create new classes that extend the class AFoo, but cannot create instances of those new classes.
class Foo extends AFoo{
    private T field 3;
    Foo(T arg){
        /*implementation*/
    }
    Foo(T1 arg1, T2 arg2){
        /*implementation*/
    }
    T Meth1(T2 arg){
        /*implementation*/
    }
}

Figure 5.6: Generated class Foo

It is important to note that this framework does not provide air-tight protection of the inaccessible and non-extendible classes from the misuse by the end-users. The users can breach these measures by decompressing and unpacking the generated JAR file, and replacing some of these unpacked classes with his own. Alternatively, they can mimic the directory structure of the package and create their own classes with the same name as some of the classes from the package, and put their own classes before the package on the classpath. Even if their classes conform to the interfaces of the ones they have replaced and have the same functionality, the program created in such a way would still be incorrect. If the end-users wish to make such changes, they need the whole-program or almost-whole-program recompilation framework.

5.2.3 Public Classes

Finally, the developers can specify which classes from the development project are to be made completely available to the end-users. These classes are declared as public and can be used by end-users in any way regular public Java classes can be used. They can instantiate them, call their methods and (for the non-final classes) write their own classes that extend
them. The conversion process from the development project into the distribution project is trivial: these classes are simply declared as public.

However, the implementation details in the context of the set-based analysis and whole-program optimization are more complex. The JaMake system has to capture the information that these classes could be extended by the end-users and pass it to the set-based type analysis tool as well as to the optimizing back-end of the compiler.

The almost-whole-program framework achieves this behavior by inserting "unknown" classes, which represent the portion of the program that the end-user can add to the final package. Let us assume that the developer marked the class Foo as available. The almost-whole-framework will create the type UnknownFoo, which is a subtype of Foo. The set-based analysis which computes the concrete type information for the program inserts this information into the analysis. For every public method (public methods of public classes and public methods of the non-extensive classes described in the previous section) that takes an argument of the type Foo, it will add UnknownFoo to the set of possible types of that argument. The analysis then proceeds as usual.

The optimization techniques from Chapter 3 are aware of this special type and limit the transformations that they perform on the program accordingly. Class specialization can specialize the classes based only on the polymorphic data that do not contain the unknown type. It will keep a default implementation (basically the original code of the class) for the classes that do contain the unknown type. After class specialization, object inlining will inline only objects of a precise (and known) type. These restrictions ensure that the program that the end-user creates by using the distribution package and extending the available classes from it is observationally equivalent to the program in which the distribution package is replaced with the development package.

Naturally, marking too many classes as public in the distribution package severely limits the range of interprocedural optimizations from Chapter 3 and reduces their effectiveness. In the extreme case, marking all of the classes available will completely prevent global object inlining (although some limited local object inlining will still be possible).
developer has to have this fact in mind when creating the distribution package, and balance its flexibility and performance.

5.3 Performance Results

The almost-whole program transformation incurs some additional overhead over the original program. This transformation adds a level of indirection to the classes declared as non-extensive. All the method calls on the non-extensive class become virtual. The most significant effect in the performance is that every instantiation has an added constructor call (to the constructor of Foo in our example) in the chain of \texttt{super()} calls in the constructors. In addition, that constructor is performing a run-time check to determine whether the object being instantiated is of a proper type (Foo in our case).

Table 5.1 shows the execution times for our standard set of benchmarks, which Chapter 2 describes in a great detail, both for the original (development) and the almost-whole-program (distribution) package. We choose to mark the classes \texttt{LNumber} and \texttt{LDouble} as non-extensive for the distribution package. These classes are extensively used by the benchmarks from Table 5.1. All the elements of matrices in these benchmarks are of type \texttt{LNumber} and are instantiated to the type \texttt{LDouble}. Table 5.1 shows the execution times for Fortran style, "Lite" OO version and object-oriented version of the benchmarks in its first three columns, respectively. The last six columns show the execution times and percentage increase of the execution time for the distribution package versions of the same benchmarks.

Since they do not utilize \texttt{LNumber} and \texttt{LDouble} classes, the numbers for "Lite" OO style and for Fortran style computation exhibit only the measurement noise.

However, Table 5.1 shows significant performance degradation for the object-oriented version of the benchmarks in the distribution package. On average, the benchmarks in the distribution package took over 23% more time to execute than their equivalents in the development package. This is because every element of every matrix involved in the computation now has a level of indirection for all method calls and two added levels of constructor calls
(both for ALNumber and for ALDouble) for every instantiation of a new number. Since
the object-oriented version of OwlPack utilizes the copy-in, copy-out semantic for number
computation (i.e. every operation on two numbers creates a new instance), these added
constructors are called very frequently.

The results from Table 5.1 suggest that the developer should utilize the almost-whole-
program infrastructure very carefully. If a key component of the program (a class that is
involved in most of the computation and could be potentially object-inlined) is made non-
extensible, the added overhead of extra constructor calls can be quite substantial. This is
exactly what has happened with the our benchmark package. Unless the end-user really
needs to see the classes LNumber and LDouble, these classes should be left as private.

Table 5.1: Sun Ultra 5 execution times, showing the overhead for almost-whole-program
transformation

<table>
<thead>
<tr>
<th>Development package</th>
<th>Distribution package</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fortran style</td>
</tr>
<tr>
<td>dpoja</td>
<td>4.66</td>
</tr>
<tr>
<td>dposi</td>
<td>9.097</td>
</tr>
<tr>
<td>dpodi</td>
<td>10.992</td>
</tr>
<tr>
<td>dgfa</td>
<td>3.182</td>
</tr>
<tr>
<td>dgesl</td>
<td>4.329</td>
</tr>
<tr>
<td>dgedi</td>
<td>6.322</td>
</tr>
<tr>
<td>dqrde</td>
<td>21.848</td>
</tr>
<tr>
<td>dqrs1</td>
<td>14.012</td>
</tr>
<tr>
<td>dsydc</td>
<td>9.079</td>
</tr>
<tr>
<td>average %</td>
<td></td>
</tr>
</tbody>
</table>

Marking LDouble as public would have an even greater effect on performance. Should
the developer mark as public or non-extensive some additional classes that are involved
in the computation and that take LDouble as an argument, class specialization and object
inlining of the class LDouble would be completely prevented. This restriction would elim-
inate most of the performance improvements from these optimizations and make the entire
almost-whole-program transformation, whole-program optimization process obsolete.
Chapter 6

Conclusions and Future Research

This dissertation lays the groundwork for further development of Java compilation technology. It explores several research directions in this novel field, while opening the doors and establishing the paths for many more technical improvements in this field and for compiler technology in general.

This chapter summarizes the research presented in this dissertation and outlines potential future research directions. The major results of this thesis are a high performance object-oriented benchmarking framework, class specialization and object inlining — new interprocedural optimizations, almost-whole-program compilation — a novel compilation framework, and exception hiding and SSA-to-CFG conversion algorithm — new techniques for improving the effectiveness of classical optimizations.

6.1 Conclusions

The next four sections summarize the results of these techniques in greater detail.

High-Performance Object-Oriented Benchmarking Framework

This dissertation develops compiler technologies that reward, rather than penalize, good object-oriented programming practice. Before evaluating these techniques, this thesis illustrates what “good object-oriented programming practice” is.

As part of this thesis, we have developed OwlPack, a collection of scientific Java programs that uses the full power of object-oriented design. OwlPack addresses the lack of suitable high-performance benchmarks for Java, which are currently compromising the elegant and flexible design for better performance. Using a polymorphic number hierarchy.
the design of OwlPack resulted in a code that is four times shorter, more extensive and less bug-prone than the equivalent Fortran-style code. This style of code is what we believe the next generation of Java programmers is most likely and most willing to use. For comparison reasons, we have also developed an equivalent collection of programs that compromises the elegant and flexible design in order to gain better performance, and that we believe the current generation of Java programmers is most likely to use.

Evaluation of the performance of the state-of-the-art Java compilers and run-time systems shows that the luxury of elegant object-oriented design comes at a dear price: the full object-oriented version of OwlPack is up to two orders of magnitude slower than the Fortran-style version. Analysis of the reasons for this performance degradation clearly demonstrates the need for developing new compiler techniques that can eliminate performance penalties without forcing the programmers to compromise their design. This thesis introduces several such programming techniques while establishing the foundation for development of many others.

Whole-program optimizations

The major source of performance degradation in object-oriented, scientific Java programs is the current "one class at a time" compilation model. This model forces the compiler to assume nothing about other classes in the program while compiling a single class, preventing any and all interprocedural optimizations.

This thesis abolishes this compilation model and recommends a whole-program compilation model, later further enhanced into an almost-whole-program framework, allowing the compilers to look at the whole (almost-whole) program at once, instead of one class at a time.

This study also introduces two novel optimization techniques, class specialization and object inlining, that take advantage of this compilation model. These optimizations for the first time address several important aspects of scientific programming in Java, such as optimization of arrays and program extensibility and portability. Class specialization
generates several versions of a class based on the types of the objects the class contains, enabling subsequent object inlining. Object inlining converts the objects in the program into inlined objects, a new data representation, and transforms the program to operate on this new representation.

These techniques can automatically transform many object-oriented, polymorphic, scientific Java programs into code that is akin to procedural, Fortran-style code. This transformation results in performance improvement of up to two orders of magnitude, practically eliminating the performance degradation due to the object-oriented style of coding.

**Almost-whole-program Framework**

A whole-program optimization framework is effective, but in Java's case, overly restrictive. The end-user can only run a program optimized by such a framework and nothing else, which might be acceptable for traditional systems like C++ or Fortran, but it is not adequate for Java, whose major strengths are flexibility and portability. This fact forces the programmers either to abandon whole-program framework for maximum flexibility, or to make numerous compromises in the program design to allow some optimizations and some flexibility, unnecessarily complicating the design.

This thesis introduces almost-whole-program compilation, a novel compilation framework that puts the power of choosing between the flexibility and the performance into the programmer's hands. It allows programmers to concentrate on elegant, object-oriented design in the development phase while permitting the compilers to perform class specialization and object inlining in the distribution phase for maximal performance.

This framework automatically performs the necessary program transformations to enable program optimizations and to restrict the usage of the final program. It enables the programmers to improve the flexibility of the program while incurring only modest performance penalties when compared to the whole-program compilation framework. Most importantly, it allows the programmers to make this choice after the development phase, enabling them to better focus on the design during that phase.
Classical Optimizations

If Java is to be used for high performance computing, some of the well-researched code-moving optimizations must also be implemented. Unfortunately, this is not a straightforward task in Java; the Java exception mechanism seriously limits most of the code motion optimizations. Therefore, this thesis introduces exception hiding, a program transformation that enables more efficient code motion. By using loop peeling and guard insertion, exception hiding creates exception-free zones, in which the code motion transformations can move the code without restraint.

Many classical optimizations use SSA form as their intermediate program representation. Many of those optimizations (for example code motion and copy folding) force node instantiation during the SSA form into CFG conversion process, which insert a large number of copies in the code, reducing the effect of these optimizations.

To cope with this problem, this thesis introduces a new, practically linear-time algorithm for SSA-to-CFG conversion. This fast algorithm considerably reduces the number of copies inserted in the conversion process, increasing the effect of classical optimizations. This algorithm proves certain theoretical properties of the SSA form, and uses these properties to perform fast interference checking, which enables fast elimination of unnecessary copies.

The techniques described above are result of exploring several research directions in the Java compilation technology. This thesis also opens the doors for future investigations in this thriving field; the next section suggests a number of possible directions for future study.

6.2 Future Directions

This dissertation establishes a sound foundation for further development of compilation techniques for high-performance Java computing. It also forms a basis for further developments of some techniques in the programming languages theory as well as in the classical compiler field. Out of the numerous possibilities for future research resulting from this thesis, this section outlines only a few most interesting ones.
Type Analysis

Precise type analysis is crucial for the effectiveness of the optimizations described in Chapter 3. This thesis uses the set-based analysis for the determination of precise object types. This analysis has a worst-case $O(n^3)$ complexity in theory but it has performed quite well in practice. Should a faster analysis prove to be necessary, a development of a faster concrete type analysis algorithm would be required.

The simplicity of Java makes it possible to perform precise type analysis much faster than is possible for other OO languages. Two different approaches to this problem can be envisioned:

(i) a generalization of the fast call graph analysis [46] to cope with inheritance and polymorphism, and

(ii) a Hindley-Milner soft type analysis using row variables akin to the SoftScheme analyzer [71].

In the generalized Hall-Kennedy fast call graph analysis, the basic strategy is to produce a completely elaborated method invocation graph, in which each edge represents a call and each vertex represents a method invoked with a unique instantiation signature. The instantiation signature is defined to be the instantiation types of the objects used as inputs to the method, along with the instantiation type of the object on which the method is invoked. Hence, a given method would have a distinct vertex for each set of input types and each variant in its inheritance hierarchy. To obtain precise type signatures, the compiler can rely on the explicit type abstractions and applications provided by GJ/NextGen [11, 25, 61].

This approach is similar to the construction of a completely precise call graph in Fortran [65, 22]. The improvement derived by Hall and Kennedy was to construct the call graph as it was needed, carefully limiting the effective cost of propagation of information at newly-discovered calls. This achieved a running time for non-polyvariant analysis that was linear in the size of the generated call graph. Of course there was loss of precision due to the absence of polyvariant analysis. However, the same strategy could be used for
co-propagating up to a fixed number of types simultaneously with an integer factor increase in running time.

We believe that these ideas can be extended into a correct, though possibly not precise, algorithm for constructing the elaborated method invocation graph in time approximately proportional to the size of that graph. Although this graph can be exponential in the number of methods in the program, it should have vertices only for signatures that can actually occur in the program when run. This may be acceptably efficient for use in a compiler. If it is not, the number of vertices can be cut by combining signatures when the optimization advantages of keeping them disambiguated are not significant.

The second approach, based on Hindley-Milner typing [59] is an attractive alternative to flow analysis because it is based on the scalable type inference algorithm used in ML and Haskell compilers to perform static type checking. Hindley-Milner inference is generally much faster than a corresponding flow analysis because it approximates subtype constraints as equality constraints augmented by parametric polymorphism, permitting constraint systems to be solved by unification in essentially linear time.

The disadvantage of Hindley-Milner analysis is a loss of precision due to the imperfect encoding of subtyping constraints as equality constraints on polymorphic row types. Some subtype constraints collapse to equality constraints, yielding less precise typing.

The encoding of subtyping as equality constraints breaks down in recursive methods because Hindley-Milner inference forces all recursive calls to have exactly the same type instantiation as the parent call. However, if the analysis is iterated, each recursive call conforms to the the polymorphic type for the procedure deduced in the preceding iteration! As a result, subtyping relationships lost in the encoding process can be recovered. To explore this research direction, one needs to implement an iterative version of the Hindley-Milner type analysis and verify the precision of the information such an analysis would generate.
Coordinated Compilation

The techniques described in this thesis form a good foundation for a development of a compiler that can transform an object-oriented program into a code that resembles procedural programming style that one can expect to see in Fortran or C. These techniques eliminate most of the overhead of the object-oriented abstractions and enable some low-level classical compiler optimizations.

However, if Java is to be used effectively for high-performance computing, some low-level machine dependent optimizations have to be performed as well. Register allocation, scalar replacement, strength reduction, instruction scheduling and similar optimizations are highly dependent on the machine on which they are being executed.

The bytecode execution model of Java presents a bottleneck for such optimizations. Bytecode is high level of abstraction that hides the underlying architecture. This enables the "write once, run anywhere" paradigm, absolute portability of the compiled code across different platforms, and a single verification algorithm on all Java VMs. However, it forces the aforementioned optimizations to be performed at the run-time, a restriction that seriously limits the choice of algorithms due to the time constraints.

A compilation scheme that coordinates the work of the static compiler and the just-in-time compiler, similar to Cartwright's code warping [23] would lessen this problem. All the optimizations are performed in a static compiler on an SSA intermediate form and are tailored to a particular target machine. The back-end of the compiler then uses a clever encoding scheme to encode the SSA form into the bytecode. The bytecode that the compiler produces is still fully verifiable and its execution produces the same result as the original program. A just-in-time compiler that is aware of the coordinated compilation scheme can decode this bytecode into a machine using a simple and very fast algorithm. This coordinate compilation approach would combine the best of the two worlds: it would enable the static compiler to perform extensive machine-dependent optimizations on the code, while preserving the security and portability of the Java bytecode mechanism.
A combination of the techniques introduced in this dissertation and the coordinated compilation scheme would enable full optimization of scientific high-performance Java programs and would bring the performance of such programs to a level comparable to Fortran or C, while maintaining the full power of object-oriented programming and of Java portability and security.

**Fast Register Allocation**

The fast SSA-to-CFG conversion algorithm presented in this thesis establishes a theoretical ground that can encourage development of a new faster register allocation algorithm. Register allocation is a crucial part of a compiler, and today's state-of-the-art compilers employ some form of graph-coloring register allocation [13]. All variations of this algorithm rely constructing an interference graph for all the variables in the program and coloring the nodes in the graph so that adjacent nodes are colored with different colors. Variables that receive the same color can then be allocated the same register.

Our algorithm essentially computes the interference information in linear time for variables that are joined at \( o \) nodes. This information can be used by the register allocator instead of computing the complete interference graph. In our scheme, only one variable can be mapped to a basic block, so some form of local register allocation within the basic blocks would have to be performed. We believe that using our discovery can enable design of a register allocation algorithm that runs in basically linear time, with some loss of precision due to the “one variable per basic block” mapping.

**Enabling More Classical Optimizations**

In addition to the exception hiding transformation described in Chapter 4, some other approaches are valuable for consideration in future research. It might be possible to apply techniques developed for optimized code debugging and for program re-execution to allow the classical optimizations to be even more effective.
One approach that can help in addressing the problem of exceptions is to emulate methods used to make it possible to debug optimized code [51]. The idea is to allow the code moving optimizations to be performed freely, without the concern about the possible exceptions, and insert a mechanism that will, should an exception occur, recover the correct state of the program. A whole method will be enclosed in a try statement, with a corresponding exception handler at the end. Thus, the handler will catch any exception that occurs in the code and use the methods from [51] to recover the correct state of the object before allowing the user's exception handling mechanism to proceed. The correct state recovery and the debugging of the optimized code problems are very similar, except for two very important details:

- In debugging optimized code, the debugger can simply inform the user if it was unable to recover the correct state of the program. In a system that uses this technique for program optimization, the program must return in the correct state, so the recovery code would have to be inserted in the program, increasing the code size.

- It is not always possible to recover the correct state, so this method would have to be restricted to optimizations that can be undone.

An approach similar to the one just described would involve code re-execution instead of the recovery of the program state. This would require insertion of checkpoints, thus increasing the execution time. Special attention would have to be given to the re-execution of I/O operations. An implementation of this method would have the same structure as the one just described, with the exception handler re-executing the non-optimized code instead of undoing the optimizations to achieve the correct program state. The re-execution of the code is described in great detail elsewhere [6, 10, 58].

All of the research directions described in this section are attractive for investigation. If pursued, they would help bring the performance of object-oriented, scientific Java programs even closer to the performance of native Fortran or C scientific programs. This thesis forms a sound basis for future research in these directions, that would ultimately result in widespread use of Java in high-performance computing.
Bibliography


Appendix A

OwlPack

In order to support our research on optimization of high-performance scientific programs in Java, we have developed OwlPack (Objects Within Linear Algebra PACKAGE). This package is an object-oriented implementation of the complete LINPACK library. This appendix describes some of the major design decisions that were made during the creation of OwlPack.

The primary goal of employing the most natural hierarchy possible while achieving the highest level of abstraction motivated the creation of a general framework for classes in Owlpack. The central design decision was how to store the elements of a matrix. Using two dimensional arrays of primitive types, like double and float, permits direct manipulation of the data and does not force translation during parameter passing to the library, since the users would be likely to have their data in this form. However, this approach suffers from the disadvantage that the class hierarchy would have to be replicated for each primitive type.

Instead, a polymorphic solution is achieved by creating a new abstract class of numbers to which any primitive type can be translated. This approach allows for a single implementation for all primitive types and can be extended to any type for which the required operations are defined. Therefore, it is easier to program, easier to maintain, and more general than specializing the code to each primitive type. In that sense, it uses the full power of the object-oriented features of the language.

As an experiment, we decided to explore both styles. We created two different matrix abstractions, Matrix and NMatrix, that represent the different storage strategies while performing the same numeric computations. The Matrix class is the superclass of all classes...
that are specialized based on primitive types and gives rise to a replicated class hierarchy parameterized by primitive element type. We call the resulting style of programming “lite” object-oriented style (Lite OO) style.

The classes that derive from NMatrix all use the abstract number class, LNumber. Thus, NMatrix is fully polymorphic and can be extended to new number types easily. We will refer to this style of programming as object-oriented (OO) style.

FMatrix, DMatrix, and CMatrix all extend Matrix and contain the BLAS (Basic Linear Algebra Subroutines) for the corresponding data type (F = float, D=double, C=complex). The CMatrix class is further specialized for double and single precision complex numbers. In our current implementation, the complex numbers used in CMatrix are objects that are an extension of our LNumber class, although consistency with the efficiency goals of the Lite OO style would dictate that CMatrix use pairs of two-dimensional arrays of primitive types—one array for the real part of the complex numbers and one array for the imaginary parts. This is a proposed change for the future version of OwlPack. All classes whose names begin with an N are extensions of NMatrix and use LNumber.

Figure A.1 shows the class hierarchy of the Matrix class, which reflects the “Lite” OO programming style.

![Matrix class hierarchy](image)

**Figure A.1 : Matrix class hierarchy, reflecting the “Lite” OO programming style**
For all classes, the next differentiation is based on the storage type, so DMatrix is extended by DBanded, DPack, DTDiag, and DFull (and similarly for FMatrix, CMatrix and NMatrix). By separating the matrices by storage form, the classes further down the hierarchy can reuse code because the internal variables are accessed the same way.

Figure A.2 : NMatrix class hierarchy. reflecting the full OO programming style.

Figure A.3 : Fully polymorphic LNumber class hierarchy.

To attain a high degree of generality, Matrix and NMatrix contain the size information, an internal pivot array and the base interface for all of the methods available for matrices. These method in general are overridden by the extended classes. Base functions throw an exception whenever the base method is not overridden because it does not always
make sense to provide every method for every extension. For example, banded matrices do not have the inverse operation, because the inverse would require more storage than that allocated for the original matrix. In the interest of abstracting these functions, our solution is to throw an exception whenever an invalid function call is made. This brings most of the interesting methods to the Matrix level where storage form and data type are not important.

In order to raise `solve()` and `determ()` to the highest level, we created a Vector class that is slightly more than an array of the primitive type being used in the corresponding Matrix. Since Matrix must account for single, double, and complex precision, but cannot see the types of the Matrix elements inside it, `solve()` and `determ()` could not return an array of the same type. Instead, they return a subclass of Vector that contains an array of the needed primitive type.

The next level of the structure was determined so that the subclasses of this level could have the most in common. Because `floats`, `doubles`, and any new number type cannot interact with each other easily, the storage type was deemed to be the most useful factor of separation between the Matrix classes. By having `FMatrix`, `DMatrix`, `CMatrix`, and `NMatrix`, we obtained a set of abstract classes that contain the BLAS for each concrete number type, which is the natural basis for such a set of routines. The doubly subscripted array that each of these classes also holds was chosen over a faster design consisting of a single array with computed subscripts because the form (arguably) better represents the way people think about matrices. An example of the differences among the OO style, the Lite OO style, and the code resulting from direct Fortran translation code is provided in Figures A.4, A.5 and A.6 by the code for computing and normalizing the determinant of a matrix.

The OO style of code works for all LNumbers, invoking methods for comparison and numeric operations. The Lite OO style allows for the direct manipulation of the information within the array so that primitive operations are performed on the elements directly. Before returning with the determinant, though, a new Vector must be formed.
for(int i=0;i<=cols;i++) {
    if(pivot[i] != i+1) Det[0].negate();
    Det[0].multTo(Mat[i][i]);
    if(Det[0].equals(0)) {
        return Det;
    } else {
        while(!(Det[0].abs()).greaterOrEqual(1)) {
            Det[0].multTo(10);
            Det[1].subTo(1);
        }
        while((Det[0].abs()).greaterOrEqual(1)) {
            Det[0].divTo(10);
            Det[1].pplus();
        }
    }
    return Det;
}

Figure A.4: OO style determinant computation.

The Fortran-style code, shown on Figure A.6, although similar in appearance to the Lite OO style which is shown on Figure A.5, follows Fortran conventions of passing in the matrix and breaking out of loops. The function containing the code from Figure A.6 also gets a reference to det as an argument and hence there is no need to return it.

DBanded, DFull, and DTDiag all extend DMatrix and are at the highest level of non-abstract Matrices. These classes contain the Gaussian (LU) decomposition routines since they are the most general routines that can be used by any subclass with the corresponding storage form. If a matrix is declared as a subclass of one of these types, it can be factored and solved with Gaussian elimination by calling its super class's factor and solve routines. DPack ed is an abstract class which does not contain the Gaussian routines. DFull also contains the singular value decomposition, qr-decomposition, and Cholesky's updating routines.

In designing NMatrix, we encountered some difficulties when dealing with LNumber, because these two classes are completely separate in their internal representation and
for(int i=0;i<cols;i++) {
    if (pivot[i] != i+1) Det[0] = -Det[0];
    Det[0] *= this.Mat[i][i];
    if (Det[0] == 0) {
        return (new DVector(Det));
    } else {
        while (Math.abs(Det[0]) < 1) {
            Det[0] *= 10;
            Det[1] /= 1;
        } while (Math.abs(Det[0]) >= 10) {
            Det[0] /= 10;
            Det[1] *= 10;
        }
    }
}
return (new DVector(Det));

Figure A.5: "Lite" OO style determinant computation.

LNumber is a fully self-supporting class. There were several instances when a new LNumber was needed inside an NMatrix, but because LNumber is an abstract class, it was impossible to create a new instance. To solve this problem, we implemented a version of the Factory [44] pattern: we made LNumber implement Cloneable and added the methods setZero() and setOne() which would set the value of the LNumber to zero or one of the appropriate exact type, respectively. An example of the intended implementation can be found in the method NFull.determ():

Det[0] = Mat[0][0].Clone();
Det[1] = Mat[0][0].Clone();
Det[0].setOne();
Det[1].setZero();

Another consideration when designing LNumber was the way in which methods should be called. For example, LNumber could have method add that takes LFloat, LDouble, LCFloat, and LCDouble and throw an exception if the method is called and not over-
for (i = 0; i < n; i++) {
    if (ipvt[i] != i) det[0] = a[i][i];
    det[0] *= a[i][i];
    if (det[0] == 0.0) break;
    while (Math.abs(det[0]) < 1.0) {
        det[0] *= ten;
        det[1] = ...
    }
    while (Math.abs(det[0]) >= ten) {
        det[0] /= ten;
        det[1] += ...
    }
}

Figure A.6: Fortran style determinant computation.

loaded for the type, much the way Number.getValue() is implemented in the standard Java library. Every class would overload the method that took its own type and perform the operation quickly because it knows precisely what the passed object is. Even though this method would have been slightly faster, we opted against it. We did not consider it appropriate for the full object oriented version, since it would introduce into LNumber a knowledge of all of the classes that extend it. Instead, we choose a slightly less efficient design that takes a LNumber and performs an explicit cast. Only one version of the method is written, and the exception is still thrown if an incompatible LNumber is passed.
Appendix B

The JaMake Project

To support the implementation and validation of the compiler techniques described in this thesis, we have developed an extensive compiler infrastructure. We call this framework JaMake (no clever acronyms here). The JaMake project is a significant part of this thesis and deserves a more detailed description. This appendix describes the implementation details of our research compiler infrastructure.

The JaMake infrastructure is written completely in Java. It consists of more than 90,000 lines of Java source code distributed over approximately 400 files with more than 500 classes. Most of the code was designed by the author of this thesis at Rice University, with some components written by the author while at Sun Microsystems, and some components modified from Sun Microsystems and Compaq.

Figure B.1 shows the structural diagram of the JaMake compiler infrastructure, with our main data structures as the nodes in the graph. Disk icons represent the data structures that are stored on the disk, while rectangle icons symbolize the internal data structures. Labeled edges in the diagram symbolize the transformations between different data structures. The unlabeled solid edges symbolize writes, and the dashed ones reads. Hexagons represent analysis and transformations that require the knowledge of more than one major data structure.

For implementation simplicity, most of the transformations in the JaMake compiler operate on data structures that are stored on the disk in the intermediate form. For example, object inlining and class specialization read the source code from the disk, create the abstract syntax trees (AST), transform the AST's and then generate the source code again. This technique eliminates many dependencies and incompatibilities (due to different ori-
gins) between the different components of the infrastructure and greatly simplifies the debugging of the components. Of course, this is not an optimal arrangement, so there are some performance penalties during the execution the JaMake compiler, but that is a small price to pay for the modularity of this design. Naturally, a production compiler would eliminate the unnecessary source-to-AST-to-source and CFG-to-SSA-to-CFG conversions and avoid storing and reading the intermediate data structures to and from the disk.

The main disk data structures are the source code for all the classes in the program, the assembler-generated bytecode, the bytecode for the classes that are not available in the source form at the compile time (or are intentionally left only in bytecode form for almost-whole-program transformation purposes), the almost-whole-program description, and the concrete type information generated by the set-based analysis. The main internal data structures are the abstract syntax trees, the control flow graph (CFG) and the static single assignment (SSA) form. The main transformations are the almost-whole-program transformation from Chapter 5, object inlining and class specialization from Chapter 3, copy minimizing conversion, code motion and exception hiding from Chapter 4, parsing, printing, AST-to-CFG conversion, CFG-to-SSA conversion, local optimizations on the SSA and CFG, and assembling.

Figure B.2 shows the flow of the source code through the JaMake compiler. Almost-whole-program infrastructure takes the source code for the almost-whole program and the program description. The program description contains the information that tells the almost-whole-program transformation component which classes are private, which are non-extensible and which are public. This component parses the source code, generates the abstract syntax trees and does its program modifications on AST's. After it handles all the classes in the development package, it generates the source code for the distribution package, as the Chapter 5 describes.

Almost-whole-program framework then passes the source code to JavaSpidey, which does the whole-program set-based analysis of the program. JavaSpidey also needs the program description to detect the classes and method that the developer has designated
as public and to augment the set-based analysis accordingly. JavaSpidey was written by Cormac Flanagan, a former Rice student, at Compaq SRC lab. It was originally meant to be used for static debugging purposes: JaMake project uses it at the front end of the compiler infrastructure. JavaSpidey has a graphic user interface that allows the user to augment the type information it has generated. This is an excellent tool for correcting the imprecisions in the analysis, as we have done in Section 3.3. JavaSpidey does not modify the source code of the program, but passes it directly to our front-end interprocedural compiler.

Since it is written completely in Java, the JaMake project executes identically on all Java platforms, with the exception of JavaSpidey. Even though JavaSpidey is written in Java as well, it depends on some native system information for getting the types of the
binary classes (such as Object.java) and as such it generates the type information only on the Solaris platforms. This limitation will be corrected in the future versions of JavaSpidey.

The interprocedural compiler performs class specialization and object inlining, as described in Chapter 3. It converts the source code of all the methods of all the classes into the abstract syntax trees and performs these optimizations directly on the AST’s. It uses a modified Visitor pattern [44] to traverse the AST’s and modify them. On the front end of this compiler is a parser from the javac compiler from Sun Microsystems, that parses the source code and generates the AST’s. After doing class specialization and object inlining, our interprocedural compiler generates the modified source for every class in the program and passes them to our mid-end, class-by-class compiler.

This mid-end compiler is again based on the javac compiler from Sun Microsystems. It compiles one class at a time. As before, it parses the source and creates an AST for all the methods in the class. It then converts the AST into the control flow graph. It also has the ability to create CFG directly from the bytecode of the class by using symbolic code interpretation of the stack machine code. This compiler then performs exception hiding, which Section 4.1 describes. It then uses the classical SSA construction algorithm [12, 34]
to generate the SSA form for all the methods in the class. It then passes the SSA form to our back-end optimizing compiler.

This back-end compiler operates on the SSA form of the code. It performs numerous classical optimizations (dead code elimination, value numbering, constant propagation, value-driven code motion, local common subexpression elimination e.t.c.). As a last step, this compiler executes copy folding and then performs our SSA-to-CFG conversion algorithm from Section 4.2 to generate the final CFG. The CFG is then passed to our assembler that generates the bytecode for each class.