SOURCE LEVEL DEBUGGING TECHNIQUES AND TOOLS FOR OPTIMIZED CODE

by

Clara Ines Jaramillo

B.S., Computer Science, University of New Orleans, 1987

M.C.S., Computer Science, Rice University, 1994

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh

2000

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Clara Ines Jaramillo

It was defended on

August 18 2000

and approved by

Dr. Mary Lou Soffa

Dr. Rajiv Gupta

Dr. Panos Chrysanthis

Dr. Thomas Gross (CMU)

Committee Chairperson

SOURCE LEVEL DEBUGGING TECHNIQUES AND TOOLS FOR OPTIMIZED CODE

Clara Ines Jaramillo, Ph.D.

University of Pittsburgh, 2000

As compilers increasingly rely on optimizations to achieve high performance, the technology to debug optimized code continues to falter. The problem of debugging optimized code is twofold because errors in an optimized program can originate in the source program or be introduced by the optimizer. Therefore, tools must be developed to help application programmers debug optimized code and optimizer writers debug optimizers from the point of view of the source program.

This dissertation first analyzes the effects of optimizations and the complexities in maintaining correspondences between the source and optimized code statements. A variety of code transformations are considered, including classical and aggressive statement level transformations, loop transformations, and inlining. A mapping technique is developed for determining the correspondences between the source and optimized code statements while code transformations are performed. The mappings capture the impact that optimizations have on statements and their instances and thus are useful for a wide range of optimizations.

Two complementary debugging techniques for optimized code are then developed and experimentally evaluated. The techniques are based on the mappings, and the effectiveness of the techniques rely on the use of both dynamic and static information. The first technique, called comparison checking, is oriented to help optimizer writers debug and validate optimizers. The technique compares values computed in both the unoptimized and optimized executions of a source program and detects semantic differences between the versions. This technique can be modified to check different levels of optimizations or tailored for specific optimizations, and in particular global register allocation. The second technique, a full reporting source level debugger for optimized code, helps application programmers find errors in source programs even though the optimized code executes. This technique reports more expected values than previously developed source level debuggers for optimized code. Both techniques are demonstrated using a compiler that performs a set of global statement level optimizations for C source programs. The techniques do not restrict the set of optimizations applied, and the optimized code is not modified, except for the setting of breakpoints. Experimental results are performed and demonstrate the approaches are effective and practical.

Acknowledgements

I am indebted to my co-advisors, Mary Lou Soffa and Rajiv Gupta, for their support, encouragement, friendship, and guidance throughout my graduate studies. Without them, this dissertation would not have been possible. I am also grateful to Panos Chrysanthis and Thomas Gross, who took the time to be on my committee and provided suggestions concerning this research.

I would like to thank my sisters, Bibi and Marisol, and my parents, Angela and Ivan, for their overwhelmingly faith in my ability to achieve my goals and for putting up with me all these years. I thank my husband Ralf, for his love and understanding of the importance of this research to me. Finally, I thank God, for giving me so many opportunities.

To my parents, Angela and Ivan.

Table of Contents

List of Figures xi

List of Tables xiv

1 Introduction 1
- 1.1 Overview of this research 5
- 1.2 Organization of this dissertation 6

2 Background and Related Work 8
- 2.1 Source level debugger 8
- 2.2 Prior work on source level debuggers 11
  - 2.2.1 Transparent behavior by limiting optimizations 12
  - 2.2.2 Transparent behavior by inserting code 12
  - 2.2.3 Transparent behavior by "undoing" optimizations 12
  - 2.2.4 Exposing the effects of optimizations 13
  - 2.2.5 Detecting and managing the effects of optimizations 14
    - 2.2.5.1 Using static information 14
    - 2.2.5.2 Using dynamic information 17
  - 2.2.6 Relative debugging 18
  - 2.2.7 Bisection debugging 18
- 2.3 Prior work on tools for debugging optimizers 18
- 2.4 Prior work and this dissertation 19

3 The Effects of Program Transformations 22
- 3.1 Terminology 23
- 3.2 Statement level optimizations 23
  - 3.2.1 The effects of modifying statements 23
  - 3.2.2 The effects of deleting statements 24
  - 3.2.3 The effects of moving statements 24
    - 3.2.3.1 Code hoisting transformations 25
    - 3.2.3.2 Code sinking transformations 27
    - 3.2.3.3 Summary of moving statements 28
- 3.3 Loop transformations 29
  - 3.3.1 Effects of duplicating loop bodies 29
  - 3.3.2 Effects of modifying the iteration space 30
  - 3.3.3 Effects of merging and splitting loops 31
  - 3.3.4 Effects of altering the index and bounds of a loop 32
  - 3.3.5 Summary of loop transformations 33
- 3.4 Inlining 33

3.5 Summary 34

4 Capturing the Effects of Program Transformations Through Mappings 36
4.1 Mappings 37
4.2 Generating mappings 40
4.2.1 Effects of statement level optimizations 41
4.2.2 Example 45
4.2.3 Effects of loop transformations 46
4.2.3.1 Effects of duplicating loop bodies 46
4.2.3.2 Effects of modifying the iteration space 48
4.2.3.3 Effects of merging and splitting loops 48
4.2.3.4 Effects of altering the index and bounds of a loop 48
4.3 Series of code transformations 49
4.3.1 Effects of inlining 54
4.4 Summary 54

5 Comparison Checking 55
5.1 Comparison checker overview 58
5.1.1 Comparison checking scheme example 60
5.2 Annotations 61
5.2.1 Supporting statement level optimizations 61
5.2.1.1 The Check $\mathbf{S_{uopt}}$ annotation 61
5.2.1.2 The Save $\mathbf{S_{opt}}$ annotation 63
5.2.1.3 The Delay $\mathbf{S_{uopt}}$ and Checkable $\mathbf{S_{uopt}}$ annotations 63
5.2.1.4 The Delete $\mathbf{S}$ annotation 64
5.2.1.5 The Check-self $\mathbf{S}$ annotation 64
5.2.2 Algorithms to place annotations for statement level optimizations . 65
5.2.2.1 Algorithm to introduce *Check*, *Delay*, and *Checkable* annotations 66
5.2.2.2 Algorithm to introduce *Check-self* annotations 67
5.2.2.3 Algorithm to introduce *Save* annotations 68
5.2.2.4 Algorithm to introduce *Delete* annotations 68
5.2.3 Supporting loop transformations and inlining 71
5.3 Implementation and experiments 72
5.4 Summary 74

6 Register Allocation Checking 76
6.1 Register allocation checker overview 78
6.2 Annotations 80
6.2.1 The Check $\mathbf{v}, \mathbf{r}$ annotation 80
6.2.2 The Register assign $[\mathbf{v},]\ \mathbf{r}$ annotation 81
6.2.3 The Load $[\mathbf{v},]\ \mathbf{r}$ annotation 81
6.2.4 The Store $\mathbf{v}, \mathbf{r}$ annotation 82
6.2.5 The Register move $\mathbf{r}, \mathbf{r}'$ annotation 82
6.2.6 Combining annotations 82
6.3 Annotation placement 83
6.4 Register allocation checker example 83
6.5 Summary 85

7 Source Level Debugger 86
7.1 Challenges of reporting expected values 88
7.1.1 Overwritten early in the optimized program 89
7.1.2 Written late in the optimized program 90
7.1.3 Computed in the unoptimized program but not in the optimized program 91
7.2 FULLDOC's approach 92
7.3 Reportability debug information 95
7.3.1 Simply reportable 95
7.3.2 Overwritten early 96
7.3.3 Written late 97
7.3.4 Never reportable because deleted along a path 99
7.3.5 Path sensitive nonreportability/reportability when deleted 101
7.3.6 Path sensitive nonreportability/reportability when written late . . . 102
7.4 Computing the reportability debug information 103
7.4.1 Determining statements that overwrite early or write late. 103
7.4.2 Computing `SaveDiscardPoints[] and EarlyAtBkpts[]`. 104
7.4.3 Computing `StopPoints[] and LateAtBkpts[]`. 105
7.4.4 Computing `AvailAtBkpts[,]`. 106
7.4.5 Computing `NotRepDelAtBkpts[]` and `NotRepLateAtBkpts[]`. 107
7.4.6 Computing `MaybeDelAtBkpts[] and MaybeLateAtBkpts[]`. 107
7.4.7 Computing `EndDelPoints[] and EndLatePoints[]`. 108
7.4.8 Computing `PotFutBkptsDel[]` and `PotFutBkptsLate[]`. 108
7.5 Supporting loop transformations and inlining 109
7.6 Implementation and experiments 110
7.7 Summary 114

8 Conclusion and future work 115
8.1 Summary of contributions 115
8.2 Future work 118

Bibliography 122

List of Figures

2.1 Source program and its unoptimized and optimized program versions example 10

2.2 Example of prior mappings 20

3.1 Constant propagation example 24

3.2 Dead code elimination example 24

3.3 Code hoisting transformation examples 25

3.4 Code hoisting transformation examples cont. 26

3.5 Code hoisting transformation examples cont. 26

3.6 Local instruction scheduling example (sinking) 27

3.7 Code sinking transformation examples 28

3.8 Loop unrolling example 30

3.9 Loop interchange example 31

3.10 Strip mining example 31

3.11 Loop distribution example 32

3.12 Loop normalization example 32

3.13 Inlining example 33

4.1 Loop invariant code motion mapping example 39

4.2 Partial dead code elimination mapping example 39

4.3 Loop interchange mapping example 39

4.4 Partial redundancy elimination mapping example 44

4.5 Mappings for unoptimized and optimized code example 45

4.6 Loop unrolling 47

4.7 Loop interchange effects on initial $\overrightarrow{one} \rightarrow \overrightarrow{one}$ mappings 47

4.8 Strip mining effects on initial $\overrightarrow{one} \rightarrow \overrightarrow{one}$ mappings 47

4.9 Loop normalization effects on initial $\overrightarrow{one} \rightarrow \overrightarrow{one}$ mappings 49

5.1 The comparison checking system 56

5.2 Program example for comparison checking 57

5.3 Comparison checker algorithm 59

5.4 Comparison checking scheme example 60

5.5 Annotated unoptimized and optimized code example 62

5.6 Types of annotations 64

5.7 Types of annotations 65

5.8 Algorithm to introduce *Check, Delay, Checkable* annotations 69

5.9 Algorithm to introduce *Check-self* annotations 69

5.10 Algorithm to introduce *Save* annotations 69

5.11 Algorithm to introduce *Delete* annotations 70

5.12 Annotated loop reversal example 71

5.13 Annotated loop unrolling example 72

6.1 Program example for register allocation checking 78

6.2 Register allocation checker algorithm 80

6.3 Annotations example 81

6.4 Register allocation checker example 84

7.1 Overwritten early example 89

7.2 Written late example 91

7.3 Not computed in the optimized program example 92

7.4 FULLDOC's strategy with respect to user inserting breakpoints 93

7.5 FULLDOC's strategy with respect to breakpoints hit 94

7.6 FULLDOC's strategy with respect to user queries 94

7.7 Debug information 95

7.8 Overwritten early example 97

7.9 EarlyAtBkpts and SaveDiscardPoints reportability debug information for overwritten early example in Figure 7.8 97

7.10 Overwritten late example 98

7.11 LateAtBkpts and StopPoints reportability debug information for overwritten late example in Figure 7.10 99

7.12 Dead code elimination example 100

7.13 NotRepDelAtBkpts reportability debug information for the example in Figure 7.12 100

7.14 NotRepLateAtBkpts reportability debug information for the example in Figure 7.10 100

7.15 MaybeDelAtBkpts, EndDelPoints, and PotFutBkptsDel reportability debug information for example in Figure 7.12 101

7.16 MaybeLateAtBkpts, EndLatePoints, and PotFutBkptsLate reportability debug information for example in Figure 7.10 102

7.17 Algorithm to compute the reportability debug information 103

7.18 Overwritten late example 106

7.19 Loop reversal example 110

7.20 Expected values not reportable 112

List of Tables

4.1 Dead code elimination effects on mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels 41

4.2 Statement level optimization effects that do not affect the mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels 41

4.3 Statement level optimizations which move statements to outer loops and affect mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels 42

4.4 Partial redundancy elimination effects on mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels 42

4.5 Statement level optimizations effects on $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$ mappings 49

4.6 Loop invariant code motion effects on mappings 50

4.7 Partial dead code elimination effects on mappings (when statements are moved across loop boundaries) 50

4.8 Partial redundancy elimination effects on mappings 51

4.9 Loop transformation effects on $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ mappings 52

4.10 Loop transformation effects on $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ mappings cont. 53

5.1 Execution times (minutes:seconds) 74

7.1 Percentage of local variables per breakpoint that are not reportable 111

7.2 Static statistics 113

7.3 Runtime statistics 113

Chapter 1
Introduction

Ever since optimizations were introduced into compilers more than 30 years ago, the difficulty of debugging optimized code has been recognized. This difficulty has grown with the development of increasingly more complex code optimizations, such as path sensitive optimizations, code speculation, and aggressive register allocation. The importance of debugging optimized code has also increased over the years as almost all production compilers apply optimizations to achieve high performance. Today's software applications are complex and consist of millions of lines of code. Optimizations are often required because of the time and memory constraints imposed on some systems. Also, current trends in processor design increasingly rely on compiler optimizations to achieve high performance. Code transformations restructure programs to reduce the number of instructions executed, exploit locality for effective use of caches and registers, and uncover parallelism at various levels of granularity for overlapping computations. For example, classical optimizations are applied for all types of architectures to reduce the number of instructions executed. Code reordering, register allocation, and loop transformations are applied for superscalar and VLIW architectures to uncover instruction level parallelism and exploit locality. Loop transformations are applied for parallel architectures to uncover loop level parallelism and exploit data locality.

Debugging optimized code is difficult because of the lack of effective debugging tools that support optimized code. If the output of the execution of the optimized code is incorrect, no tools exist to help the programmer determine the origin of the error. Programmers typically assume that if the unoptimized version of the program executes correctly but the optimized version does not, then the optimizer is responsible for the change in semantic behavior. Given an input, the semantic behaviors of an unoptimized program and its optimized program version are the same if all corresponding statements executed in both programs compute the same values. However, differences in semantic behaviors between unoptimized and optimized program versions can be caused by either (1) the application of an unsafe optimization, (2) an error in the optimizer, or (3) an error in the source program that is exposed by the optimization. For instance, reordered operations under certain conditions can cause overflow or underflow or produce different floating point values. The optimized program may crash because of instruction reordering. For example, a statement may be moved out of a loop in the optimized code, and at runtime, the program crashes because the statement divides by zero. The application of an optimization may assume that the source code being transformed follows a programming standard (e.g., ANSI standard), and if the code does not, then an error can be introduced by the optimization. The optimizer itself may also contain an error in the implementation of a particular optimization. And lastly, the execution of the optimized program may uncover an error that was not detected in the unoptimized program. For example, code transformations change the data layout of a program. This change may cause an uninitialized variable to be assigned different values in the unoptimized and optimized programs causing both program versions to behave differently. Thus, for a number of reasons, a program may execute correctly when compiled with the optimizer turned off but fail when the optimizer is turned on, and if application programmers intend to ship optimized code, then the fully optimized version should be fully debugged.

If an optimization is incorrectly implemented and thereby caused the error, then the optimizer is responsible for the error. In this situation, the programmer is the optimizer writer and the optimizer must now be debugged. The task of debugging an optimizer is difficult and tedious. The optimizer writer must first locate the incorrect code in the optimized program. Because of the lack of source level debugging tools for optimized code, optimizer writers typically resort to examining and debugging the assembly code to locate the incorrect code in the optimized program. Then the optimizer writer must determine what code transformation(s) produced the incorrect code, and subsequently locate the error(s) in the code transformation(s). Unfortunately, little work has been done to help isolate and analyze errors in the optimizer, and isolating errors in the optimizer remains an open problem.

If an error originates in the source program, then the application programmer is responsible for the error. In this case, the programmer must debug the optimized code to determine the cause of the error. However, since application programmers typically have no knowledge of the optimizations that were applied to the source program and do not understand how the application of optimizations affect a source program, examining and debugging the assembly code to locate the incorrect code in the optimized program is not an option. Instead, application programmers must rely on source level debugging tools to debug optimized code.

Unfortunately, conventional source level debuggers cannot be used to help debug and understand the execution behavior of optimized code. A conventional source level debugger allows a user to suspend the execution of a program and examine the suspended program state with respect to the source program. If the unoptimized version of the source program is debugged and executed, a debugger simply reports the actual contents of the locations of requested variables at breakpoints, as these are the values a user expects to observe. However, by debugging and executing the optimized version, the actual content of a variable's location at a breakpoint can differ from the value the user expects to observe (if instead the unoptimized version were being debugged and executed) because optimizations move, modify, insert, and delete code in the optimized program. Therefore, in such situations, the debugger can mislead the user if the actual value is reported at the breakpoint. Moreover, a conventional source level debugger allows a user to modify the values of variables during the execution of a program, but because of the effects of optimizations, variable modification is not allowed if the debugger executes the optimized code.

The inadequate support of debugging optimized code is the reason that software companies may not deliver optimized code. If the optimized version is shipped to customers, future bugs arising during customer use are difficult to correct within a reasonable amount of time. Also, if the program crashes, the resulting core file is useless since there is no way to correlate the optimized program to the source program.

Several approaches to debugging optimized code have been proposed and are aimed at either (1) avoiding the problems in debugging optimized code or (2) helping programmers understand the execution behavior of the optimized code. One approach is oriented to application programmers and has the programmer turn off optimizations during the development of the software application but turn on the optimizations for the production version to gain the performance benefits provided by optimizations. In other words, debug the unoptimized version of the program but ship its optimized version. This approach avoids the problems associated with debugging optimized code. Unfortunately, when the application, apparently free of errors, is optimized, its behavior may not be the same as the behavior of the unoptimized program. As mentioned earlier, in this situation, the application programmer is likely to assume errors in the optimizer are responsible for the change in behavior. The optimizer is then turned off, the unoptimized version is shipped, and all of the performance benefits of optimizations are lost. In fact, after a programmer experiences this situation several times, all confidence is lost in the optimizer and the programmer will typically not use the optimizer in future programs. However, the optimizer may be correct, but the unoptimized version of the program may contain an error that is exposed by the optimizer. Clearly in this case, instead of shipping the incorrect unoptimized program version, the application program should be further debugged.

Another approach, equally unsatisfactory, is to execute the optimized code and require the programmer to be sufficiently knowledgeable about optimizations. The programmer must determine what debugging query can be issued to get an expected response from the source level debugger[14]. While an optimizer writer may be able to utilize this approach, an application programmer is seldom familiar enough with optimizations to accomplish this burdensome task.

Instead of debugging optimized code from the point of view of the source code, another approach allows the programmer to debug optimized code from the point of view of a modified version of the source program, which displays the effects of optimizations[45]. However, the programmer must become familiar with a different version of a program, which can greatly differ from the source program. This approach benefits optimizer writers since the information can be used to understand how the source program has been transformed. However, this approach is too burdensome on application programmers.

The most researched approach to debugging optimized code is to use a specialized source level debugger that attempts to report expected values when they can be determined from the optimized code and also report when an expected value cannot be determined. Progress has been made in the development of debuggers that report more and more expected values. The early techniques focused on determining expected values using information computed statically [27, 21, 19, 49, 10]. Recent techniques have proposed using information collected during execution, along with the static information, to improve the reportability of values [22, 51, 50]. Despite the progress, none of the techniques are able to report all possible expected values of variables at all breakpoints in the source program, and thus, the optimizer writer and application programmers cannot easily debug optimized code from the point of view of the source code. The design of a source level debugger for optimized code that has the same debugging capabilities as for unoptimized code remains an open problem.

As optimizers continue to perform more sophisticated optimizations to exploit more demanding architectural features, the demand for tools to help optimizer writers debug and validate optimizers is ever increasing. The need for tools to help programmers determine the origin of an error in the optimized code is also increasing. Moreover, as application programmers continue to rely on optimizations to achieve high performance, the importance of debugging optimized code from the point of view of the source program increases. A programmer should be able to debug optimized code from the point of view of the source program. Also, a programmer should be able to suspend the execution of an optimized program between any two source statements and query all expected values of source variables at the suspended execution points. The optimized program that a programmer debugs should be identical to the version that is released. In other words, for debugging purposes, an optimizer should not be restricted to apply only a certain set of code transformations nor should the applicability of code transformation be restricted. Since the optimized program is generally the desired final version of a program, the optimized code should not be modified, except for the setting of breakpoints during debugging.

1.1 Overview of this research

This dissertation explores the following open problems in the area of debugging optimized code:

- debugging and validating optimizers,
- source level debuggers for optimized code, and
- determining the origin of an error in the optimized code.

This dissertation develops different source level debugging techniques and tools for optimized code so that optimizer writers can debug optimizers and application programmers can debug optimized code from the point of view of the source program. In particular, the use of dynamic information as well static information is explored to develop effective source level debugging techniques for optimized code. The use of dynamic information has benefited optimizers as dynamic information provides more opportunities to apply optimizations [15, 12]. Similarly, the use of dynamic information should increase the effectiveness of source level debugging techniques for optimized code. Since the use of dynamic information is expensive in terms of overhead, static information is used to minimize the amount of dynamic information utilized.

This dissertation considers source programs written in the C programming language. The techniques presented are demonstrated in a compiler that performs a set of global statement level optimizations for source programs. The techniques do not restrict the set of optimizations applied and the optimized code is not modified, except for the setting of breakpoints.

Before presenting the techniques and tools to debug optimized code, this dissertation analyzes the effects of optimizations and the complexities in maintaining the correspondences between the source and optimized code statements. A variety of code transformations are considered, including statement level optimizations, loop transformations, and inlining. Statement level optimizations include speculative code motion and path sensitive optimizations. A mapping technique was developed for capturing the correspondences between the source and optimized code statements while code transformations are applied. In particular, the mappings capture the impact that optimizations have on statements and their instances and thus are useful for a wide range of optimizations.

Two complementary debugging techniques for optimized code were developed, implemented, and experimentally evaluated. The first technique, called comparison checking, is aimed at helping optimizer writers debug and validate optimizers. The comparison checking technique compares values computed in both the unoptimized and optimized executions of a source program and detects semantic differences between the two versions. This technique can be modified to check different levels of optimizations (high, intermediate, or low level) or to check each optimization phase, or tailored for specific optimizations, and in particular global register allocation.

Once an optimizer is debugged and validated, errors in the optimized code are the responsibility of the application programmer. Thus, the second technique, a full reporting source level debugger for optimized code, is developed to help application programmers find errors in source programs that are optimized. The debugger reports more expected values than previously developed source level debuggers for optimized code. To report expected values that would otherwise not be reportable by previously developed debuggers, statically computed information is utilized to guide the debugger in gathering dynamic information. Using this static and dynamic information, the debugger can report expected values at breakpoints when reportability is affected because values have been overwritten early, due to code hoisting or register reuse, or written late, due to code sinking. The debugger can also report values that are path sensitive in that a value may be computed only along one path or the location of the value may be different along different paths. This dissertation does not consider the user modification of variables during a debugging session nor does it consider debugging core files.

1.2 Organization of this dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents background information on source level debuggers and explains the problems imposed on debuggers of optimized code which stem from the application of compiler transformations. This chapter describes the prior work that has been performed in debugging optimized code, including tools for debugging optimizers and source level debuggers of optimized code. Also, the relationship of this dissertation and prior work is discussed. Chapter 3 discusses the effects of optimizations and the complexities in maintaining the correspondences between the source and optimized code statements. Chapter 4 presents a technique for generating the correspondences between the source and optimized code statements that captures the effects of optimizations. In Chapter 5, the comparison checking technique for debugging optimizers is presented. Chapter 6 discusses how to tailor the comparison checker to debug and validate specific optimizations, in particular global register allocation. Chapter 7 presents a full reporting source level debugger for optimized code. Conclusions and directions for future research are discussed in Chapter 8.

Chapter 2
Background and Related Work

Since most of the previous work on debugging optimized code focused on the development of source level debuggers for optimized code, this chapter first provides background information on source level debuggers and then describes the problems inherent in source level debuggers of optimized code. Next, prior work that has been proposed in debugging optimized code, including tools for debugging optimizers and source level debuggers of optimized code, is described. Also, the relationship of this dissertation and prior work is discussed.

2.1 Source level debugger

A source level debugger is a tool that helps users understand the execution behavior of a target program in terms of the source program. The debugger allows the user to control the execution of the target program and examine the suspended program state with respect to the source program. Breakpoints are used to suspend the execution of a program. The most common form of a breakpoint is the *control breakpoint*, which specifies the breakpoint condition in terms of the source code, that is, at a specified line, function, or between any two source statements. Other types of breakpoints can suspend the execution of a program. A *conditional breakpoint* is initiated only if some location-dependent predicate evaluates to true. A conditional breakpoint is useful when the user wants to place a breakpoint in a heavily executed region of code. For example, a conditional breakpoint can be placed in a *for loop* with index i, and the condition $i = n$ can be used to initiate the breakpoint when variable i has the same value as variable n. A *data breakpoint* is initiated when a variable is referenced (i.e., read or written). A data breakpoint is useful when the user wants to suspend execution before every statement that references a particular variable without having to insert an explicit control breakpoint at every such statement. Since most of the previous work on debugging optimized code addressed source level debuggers and considered only control breakpoints, the term debugger refers to a source level debugger and the term breakpoint refers to a control breakpoint in the remainder of this dissertation.

Typically, a user starts a debugging session by inserting breakpoints in the source program and then instructing the debugger to execute the target program. When execution of the target program reaches a breakpoint, the debugger suspends the execution of the program and returns control to the user. At this point, the user can continue the execution of the program or examine the control state and/or the data state of the suspended program. Since all user commands are in terms of the source program, the debugger must convert a source level query to a target level query. Also, all debugger responses to the user are in terms of the source program. Therefore, the debugger must be able to

(1) insert a breakpoint in the target program in response to a breakpoint in the source code,

(2) determine when the current execution point of the target program corresponds to a source location at which the user has requested a breakpoint,

(3) display the contents of a storage location in the execution of the target program that corresponds to source variables at which the user has requested at the current breakpoint, and

(4) display the current execution point of the target program in terms of the source program.

To achieve these tasks, the debugger utilizes information relating the source program to the target program. The compiler provides this debug information, which includes information about the variables and statements in the source program and relates them to storage locations and instructions in the target program.

Conventional debuggers are typically designed to execute the unoptimized version of a source program. Debuggers for unoptimized code are straightforward to implement because a source program and its unoptimized version have a direct correspondence. That is, (1) source level statement boundaries are preserved, and (2) variables have unique memory storage locations. Therefore, when the execution of a program reaches a breakpoint, all statements prior to a breakpoint will have executed and all statements after the breakpoint will not have executed. When the user queries the values of source variables, the debugger simply reports the actual contents of the memory locations of requested variables at the breakpoints, as these are the values the user expects to observe. Consider the source program fragment and its unoptimized version in Figure 2.1. Suppose the user places a

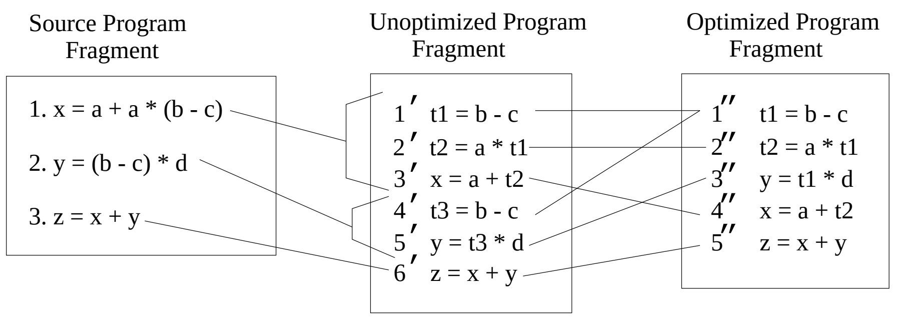


Figure 2.1: Source program and its unoptimized and optimized program versions example

breakpoint between statements 1 and 2 in the source code, and then at the breakpoint, examines the current values of x and y. Notice all of the source level statement boundaries are preserved in the unoptimized program. Statement 1 in the source program maps to statements $1'$, $2'$, and $3'$ in the unoptimized program, statement 2 maps to statements $4'$ and $5'$, and statement 3 maps to statement $6'$. Thus, when the debugger executes the unoptimized program version, the breakpoint is easily placed between statements $3'$ and $4'$ in the unoptimized program. Upon reaching the breakpoint during the unoptimized program execution, the debugger displays the expected values of both x and y, as their memory locations contain the expected and correct values.

However, if the optimized program version of the source program is debugged and executed, a conventional debugger cannot always simply report the actual content of a requested variable's location at a breakpoint because the expected value may differ from the actual value of the variable. If an unexpected value is reported, the debugger can mislead the user. This difficulty faced by debuggers in reporting the expected values of variables is caused by the effects of optimizations, which move, modify, insert, and delete statements in the optimized program. Because of the effects of optimizations, two problems surface when trying to debug optimized code from the viewpoint of the source program. The *code location* problem relates to determining the position of a breakpoint in the optimized code that corresponds to the breakpoint in the source code. The *data value* problem is the problem of reporting the values of the source variables that a user *expects* to see at a breakpoint in the source code, even though the optimizer may have reordered or deleted the statements computing the values, or overwritten the values by register allocation.

Consider the same source program fragment and its optimized version in Figure 2.1. The debugger has problems reporting the expected values of x and y, regardless of where the breakpoint is placed in the optimized code. If the breakpoint is placed between statements $2''$ and $3''$, then upon reaching the breakpoint during the optimized program execution, x

contains an unexpected value. The user expects to see the value computed by statement 1 in the source program, but since the corresponding assignment (statement $4''$) in the optimized program occurs after the breakpoint, the old value of x will be reported to the user instead. If the breakpoint is placed between statements $4''$ and $5''$, then upon reaching the breakpoint, y contains an unexpected value. The user expects to see the value of y before the assignment of y in statement 2 of the source program, but since the corresponding assignment (statement $3''$) in the optimized program was moved up in the code and occurs before the breakpoint, the future value of y will be shown to the user instead. These problems occur because the boundaries of source level statements 1 and 2 are not preserved in the optimized program. Statement 1 in the source program maps to statements $1''$, $2''$, and $4''$ in the optimized program, statement 2 maps to statements $1''$ and $3''$, and statements $1''$, $2''$, and $4''$ overlap with $1''$ and $3''$. These code location and data value problems exist when one of the aforementioned conditions does not hold in the optimized program. These problems constrain the debugging capabilities of conventional debuggers. Nonetheless, to be effective, debuggers should report the expected values of all source variables accurately.

2.2 Prior work on source level debuggers

The problem of debugging optimized code has long been recognized [48, 27, 40]. As mentioned earlier, most of the previous work focused on the development of source level debuggers for optimized code [27, 24, 52, 53, 39, 21, 26, 28, 14, 38, 9, 8, 10, 7, 20, 18, 19, 49, 22, 45, 51, 50]. Some debuggers provide *transparent behavior* [52, 53]. A debugger provides transparent behavior with respect to an optimization if responses to user queries are the same as the responses would be if the unoptimized program version is being debugged instead. Since transparent behavior is very difficult to achieve, approaches that provide transparent behavior either constrain optimizations or modify the optimized code. Instead of providing transparent behavior, other debuggers expose the effects of optimizations to the user either in terms of the source program or a different version of the source program. Finally, in an attempt to provide transparent behavior when possible, some debuggers detect and manage the effects of optimizations. Approaches that detect and manage the effects of optimizations use static information and/or dynamic information.

2.2.1 Transparent behavior by limiting optimizations

Fritzson's [24] debugging system provides transparent behavior at the expense of limiting optimizations to within a source statement. Code location and data value problems do not exist since breakpoints are placed at source statement boundaries. Therefore, full debugging capabilities are provided, but optimizations are limited.

2.2.2 Transparent behavior by inserting code

Gupta [26] considers debugging code reorganized by a trace scheduling compiler. His approach compromises debugging features for optimizations. A user must first specify monitor commands to view variables or conditions at specified points in the program. Next, any affected traces are recompiled, and monitor and renaming code is inserted into the optimized program. Then the program is executed. Code location and data value problems do not exist because monitor and renaming code are inserted into the optimized program.

Pineo and Soffa [36, 37, 38] consider debugging parallelized FORTRAN programs from a sequential source level program point of view. The parallel transformations considered are renaming, scalar expansion, loop interchange, source level spreading, global forward substitution, loop fission, loop fusion, and strip mining. These optimizations are applied on source level code. A program is converted into single assignment form to allow the tracking of values during program execution. All values can be reported except values whose computations are delayed past the breakpoint or deleted values. Thus, data value problems are partially handled. Code location problems are handled by using *syntactic breakpoints*. A syntactic breakpoint at statement S in the source program is placed at the original location of statement S in the optimized program.

2.2.3 Transparent behavior by "undoing" optimizations

Pollock and Soffa's [39] debugger inhibits compiler optimizations that affect debugging requests. They consider programs optimized with constant folding, redundant store elimination, global common subexpression elimination, copy propagation, and loop invariant code motion. Annotated DAGS represent both the unoptimized and optimized programs. They allow users to insert control and conditional breakpoints, examine and modify values of variables, single step, and edit code. Also, full debugging features are possible from specified point to point. Debugging features must first be specified before program execution. Next, the program is incrementally compiled to inhibit necessary compiler optimizations. Then the resulting program is re-executed. In some cases, the debugger can instead perform on-the-fly recovery of variables. Code location and data value problems do not exist, but the fully optimized program is not always debugged.

Hölzle, Chambers, and Ungar's [28] approach involves debugging and executing the optimized code but dynamically deoptimizing the code to provide full debugging capabilities. When full debugging features are not needed, the optimized code is executed. This switching between both program codes can occur only at interrupt points (method prologues and end of loop bodies) such that the source level state can be reconstructed. Their method applies to programs written in an object oriented language, SELF, optimized with global constant propagation, constant folding, global register allocation, inlining, customization and splitting, dead code elimination, strength reduction, global common subexpression, elimination of arithmetic expressions, loads, and stores, redundant computations that cannot cause observable side effects as arithmetic overflow, loop unrolling, and delay slot filling. This list is extensive, but the applicability of optimizations is restricted because the source level state must be reconstructed at interrupt points. Code location and data value problems do not exist, but the fully optimized program is not always debugged.

2.2.4 Exposing the effects of optimizations

Brooks, Hansen, and Simmons [14] take a different approach to debugging. They consider programs compiled with CONVEX FORTRAN and C compilers (with all levels of optimizations). The source program and the assembly level program are highlighted and animated, visually conveying the effects of optimizations on program behavior. Some optimization effects are hidden and others are not. Program stepping is provided at several levels: expression, statement, block, loop, and routine. Code location problems are partially handled. Data value problems are partially handled by using compiler generated tables of live ranges to determine if values of variables are available. Also, they can recover variables deleted due to strength reduction and induction variable elimination.

`Optview` [46] generates an optimized source program version for C programs, which conveys to the user the effects of copy propagation, constant folding, common subexpression elimination, partial redundancy elimination, dead code elimination, code hoisting and sinking, and instruction scheduling. Another related research effort is the `Optdbx` debugger [45], which displays the optimized version of the source program that is generated by `Optview`. All user commands and debugger responses to the user are with respect to the optimized source program version. Also, `Optdbx` uses invisible breakpoints to recover variables that are *evicted* from registers and determine the correct location of a variable whose location depends on the execution path[1]. Since the user debugs from the point of view of the optimized source program, data value and code location problems do not exist. However, the user must be aware of the optimizations.

2.2.5 Detecting and managing the effects of optimizations

Most debugging techniques focus on detecting the effects of optimizations and providing expected behavior when possible. Some debugging techniques focus on determining expected values using information computed statically [27, 21, 20, 18, 19, 49, 9, 8, 10, 7]. Other techniques have proposed using information collected during execution, along with the static information, to improve the reportability of values [22, 51, 50] or to handle code location problems [52, 53].

2.2.5.1 Using static information

Hennessy [27] considers debugging programs, written in a subset of PASCAL, whose optimizations are applied at the intermediate code level. Annotated DAGS represent both the unoptimized and optimized programs. By statically analyzing the annotated DAGS, Hennessy classifies variables at breakpoints as *current* or *noncurrent* by determining if variables are *endangered.* A variable is endangered at a breakpoint b if there is a path to b such that the variable may not have the correct value at b, due to a program transformation. A variable is current at b if it is not endangered on any path to b. A variable is noncurrent at b if it is endangered on all paths to b. Variables that are classified as current at breakpoints are reportable by the debugger, as these variables contain expected values at the breakpoints. Setting breakpoints and examining values of source variables are the debugging features considered. Data value problems are partially handled by conservatively detecting and recovering noncurrent variables. Algorithms are developed to detect and recover noncurrent variables in programs with local optimizations: common subexpression elimination, redundant store elimination, and code reordering. However, recovering values of noncurrent variables is not always possible. Techniques are described to detect and recover noncurrent and endangered variables in programs with global optimizations: code motion from loops to preheader, induction variable elimination, and global dead store elimination. These techniques use data flow analysis to detect endangered variables. Hennessy's technique for local optimizations does not modify the optimized code, unlike his technique for global optimizations, which inserts flag instructions to determine dynamic

[1]A variable v is evicted from a register if v is assigned to the register and then a value that is not from an assignment to v is stored in the register.

program flow. Code location problems are handled by restricting placements of breakpoints and code generation.

The work of Coutant, Melloy, and Ruscetta [21] develops a symbolic debugger for C programs optimized with global register allocation, induction variable elimination, constant and copy propagation, and instruction scheduling. Debugging features include inserting breakpoints and examining values of variables. Code location problems are handled by using syntactic breakpoints. Data value problems are partially handled by using compiler generated tables and recovering eliminated variables due to strength reduction and induction variable elimination. A live range table is used to determine if a variable's value is available. When values are not available, partial information is provided so that the user can try recomputing the data. Also, another table is used to determine if a variable was modified early or late.

Adl-Tabatabai and Gross [9] consider the effects of global register allocation and assignment on the residency problem, which determines if a variable will be in its assigned register at a breakpoint. Data flow analysis is used to determine whether a variable is evicted from a register or uninitialized. They [8] also detect and recover endangered variables caused by local instruction scheduling. Endangered variables are classified as noncurrent or *suspect*. A variable is suspect at a breakpoint b if the debugger is not able to determine whether the variable's actual value is the expected value at b. To determine if a variable is endangered at a breakpoint, the intermediate representation is annotated with the effects caused by local instruction scheduling. Their solutions [9, 8] are implemented using the `iWarp C` compiler with global optimizations, including register allocation and assignment, branch optimizations, constant folding, and unreachable code elimination, and local optimizations, including common subexpression elimination, value propagation, and instruction scheduling. Code location problems do not exist, and data value problems are partially handled. They are able to detect endangered variables and recover values by interpreting instructions. However, full recovery of values is not achieved because they do not attempt recovery of rolled back variables, as rolled back variables are difficult to recover [27]. They do not interpret function calls since they transfer control out of the current basic block and such interpretation would be difficult. They do not interpret loads since the values in memory may be endangered.

Copperman [20, 18, 19] takes an approach more general than the previous work in dealing with the data value problem. He uses data flow analysis on a single graph, which represents both the unoptimized and optimized programs, to determine whether variables are current, noncurrent, or endangered at breakpoints. Thus, data value problems are partially handled. Source programs can be optimized with local and global common subexpression elimination, constant and copy propagation, constant folding, dead code elimination, dead store elimination, cross-jumping, local and global instruction scheduling, strength reduction, code hoisting, partial redundancy elimination, induction variable elimination, loop unrolling, inlining, and other optimizations that do not change the order in which basic blocks are entered. For example, loop interchange is not handled. This list is extensive and applicable only at the intermediate code level. Code location problems are handled by using syntactic breakpoints.

Wismueller's [49] approach is similar to Copperman's approach but more general and gives the correct answer in the few circumstances where Copperman does not. For the static analysis, loops are unrolled in the unoptimized and optimized control flow graphs to distinguish among different instances of definitions. Then data flow analysis is performed on the unrolled unoptimized and optimized program control flow graphs to determine whether variables are current or noncurrent at breakpoints. Thus, data value problems are partially handled. His algorithms are implemented on a C compiler, which applies global common subexpression, global copy and constant propagation, global dead store elimination, loop invariant code motion, composite breaking, and register allocation. The optimized program is not modified. The modified control flow graphs are only used during static analysis. Code location problems are not considered, but his work is applicable with syntactic or *semantic* breakpoints. A semantic breakpoint at statement S in the source program is placed at the point at which the action specified by statement S occurs in the optimized program.

Another approach similar to Copperman's approach is that of Adl-Tabatabai and Gross [10]. Variables are classified as current, noncurrent, or suspect at breakpoints by using data flow analysis on the intermediate representation, which is annotated with the effects caused by optimizations. Nonresident and uninitialized variables are not considered. They assume code is not moved arbitrarily in the program. That is, they assume a computation cannot be introduced into a path where it did not exist before. Their approach was implemented using the `cmcc` compiler, which applied loop unrolling and peeling, induction variable expansion, constant propagation and folding, assignment propagation, dead assignment elimination, strength reduction, global register allocation, local instruction scheduling, linear function test replacement, induction variable simplification, induction variable elimination, partial dead code elimination, partial redundancy elimination, branch optimizations, and register coalescing. Software pipelining and loop transformations, such as loop interchange, are not considered. Code location problems are not considered. Data value problems are partially handled. Some recovery techniques are described. Adl-Tabatabai [7] extends his previous techniques by handling code location problems and recovery of values. He also describes how to handle speculative code motion with respect to data value problems.

2.2.5.2 Using dynamic information

Zellweger's [52, 53] debugging system Navigator handles Cedar (an Algol-like language) programs optimized with inline procedure expansion and cross-jumping. The debugging features she considers are inserting breakpoints, viewing procedure tracebacks, and examining values of variables. Data value problems do not exist because variables are always current. Code location problems are partially handled. In reference to breakpoints, transparent behavior is partially provided by using compiler generated tables and invisible breakpoints to collect an execution-history. In some cases, Zellweger's system does modify the optimized program to collect information about the execution path.

Recent work has focused on utilizing dynamic information along with static information to improve the reportability of values. Wu et al. [51, 50] selectively take control of the optimized program execution and emulate instructions in the optimized code in the order that mimics the execution of the unoptimized program. This execution reordering enables the reporting of some of the expected values of variables that are otherwise not reportable by other debuggers. Code location problems are avoided by altering the execution of the optimized program. However, altering the execution of the optimized program masks certain user and optimizer errors [51]. Data value problems are partially handled. The emulation technique does not track paths and cannot report values whose reportability is path sensitive. Their approach was implemented using the `IMPACT` compiler [17], which applied instruction scheduling, register allocation, and classical local and global optimizations such as induction variable optimizations, strength reduction, common subexpression elimination, constant folding, copy propagation, loop invariant code motion, and store/copy optimizations.

Dhamdhere et al. [22] developed a dynamic currency determination technique that can also report some values of variables that are not reportable by other debuggers. They create a minimal unrolled graph of a program and timestamp basic blocks to obtain a partial history of the execution path, which is used to precisely determine what variables are reportable at breakpoints. However, values that are overwritten early by either code hoisting or register reuses are not always reportable. Thus, data value problems are partially handled. Code location problems are not considered.

2.2.6 Relative debugging

Guard, a relative debugger, is similar to one of the debugging approaches advocated in this research in that two programs are executed and the values generated are compared [44, 5, 2, 4, 3]. Using Guard, users can compare the execution of one program, the reference program, with the execution of another program, the development version. Guard requires the user to formulate assertions about the key data structures in both versions and specify the locations at which the data structures should be identical. The relative debugger is then responsible for managing the execution of the two programs and reporting any differences in values. Guard is implemented using a debugging platform, Dynascope [41, 42, 43], which provides an interface for process control, state access, and breakpoint handling. Guard is not designed to debug optimized programs and could not be easily extended. Some optimizations are low level and the user would have to formulate assertions on assembly level statements. Guard has been extended to implement a parallel relative debugger [6], but the user must still formulate assertions. The primary difference between Guard and this research is that the latter scheme, which compares the executions of the unoptimized and optimized programs, is transparent to the user.

2.2.7 Bisection debugging

The concept of a bisection debugging model also has as its goal the identification of semantic differences between two versions of the same program, one of which is assumed to be correct [25]. The bisection debugger attempts to identify the earliest point where the two versions diverge. However, to handle the debugging of optimized code, all data values problems must be solved at all breakpoints.

2.3 Prior work on tools for debugging optimizers

Not much work has focused on developing tools to help debug optimizers. Bugfind [16] was developed to help debug optimizers by pinpointing which functions produce incorrect code. This tool also helps application writers by compiling each function to its highest level of correct optimization. To achieve these tasks, functions must be placed in separate files.

Boyd and Whalley[13] developed two tools to help debug optimizers. The first tool, `vpoiso`, identifies the first transformation during optimization that causes the output of the execution to be incorrect. In addition, the tool can identify the location and instance the offending transformation is applied. To aid in identifying the error in the implementation of an optimization, a graphical optimization viewer, `xvpodb`, was developed and allows users to view the state of the generated instructions before and after each application of transformations. However, if the optimizer writer cannot conclude which specific instructions in the optimized code produce incorrect results, using `xvpodb` will be tedious since the user has to potentially view the states of all instructions that are affected by the offending transformation.

More recent work [34] statically compares the intermediate form of a program before and after a compilation pass and verifies the preservation of the semantics. This work symbolically evaluates the intermediate forms of the program and checks that the symbolic evaluations are equivalent. This translation validation system was demonstrated in the context of the GNU C `gcc` compiler performed between each optimization phase. Optimizations include branch optimization, local and global common subexpression elimination, loop unrolling, loop inversion, induction variable optimizations, local and global register allocation, instruction scheduling, procedure integration, and tail-recursion elimination. The work used the `gcc` version 2.7.2.2, which is known to exhibit bugs in the register allocator and loop unrolling. In both cases, this translation validation system was able to detect these bugs. However, the system also detects false alarms that are not necessarily errors. This system does not obviate the need for extensive compiler testing suites. Also, this system is not careful about instructions that might raise exceptions. Thus a statement that is moved out of a loop and divides by zero at runtime remains undetected.

2.4 Prior work and this dissertation

None of the previous work successfully handles the code location and data value problems faced by source level debuggers for optimized code. Prior work does not ensure that the execution behaviors of the unoptimized and optimized programs are the same with respect to the behavior of the source level program and the given input. Also, prior work cannot help automatically pinpoint errors in the optimized code.

Another difference between the proposed work and previous work is the tracking of the unoptimized program with the optimized program version. Mappings are used to track information between the source or unoptimized program and its optimized program version. The mappings developed in this dissertation are extensions of mapping techniques previously developed for source level debuggers of optimized code, which capture only the correspondences between statements in the unoptimized and optimized programs and not statement instances. Most prior work statically analyzes the mappings to determine if variables are resident, nonresident, current, noncurrent, or endangered. The results are

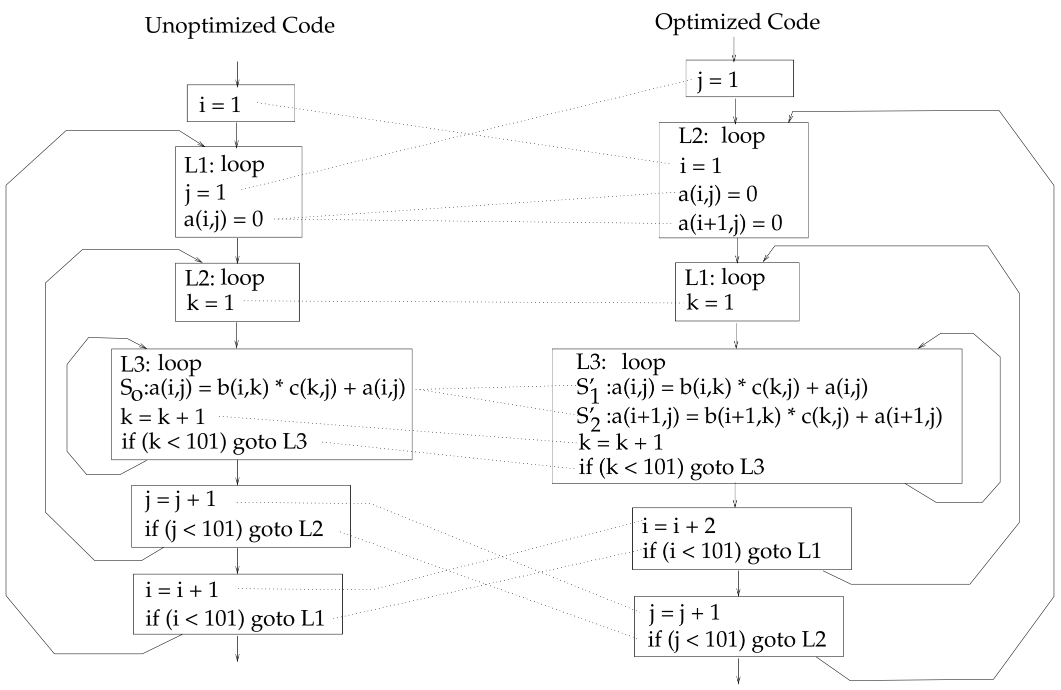


Figure 2.2: Example of prior mappings

conservative because dynamic information is not utilized and the correspondences between statement instances in both programs are not captured. More recent work utilizes some dynamic information to report more expected values [51, 50, 22].

Consider the example in Figure 2.2, in which the loops are interchanged, unrolled, and jammed in the optimized code, and as a result, some instances of statements are reordered and deleted. If only mappings of statements are used by a source level debugger that executes optimized code, the debugger is ineffectual. For any breakpoint placed within a loop, all variables (except k) inside the loop are considered noncurrent and their values cannot be reported. The debugger does not have knowledge of what values can be reported because it does not have information about loop iterations and statement instances. For example, if the user places a breakpoint after statement S_o in the innermost loop of the unoptimized program and requests the value of $a(i,j)$, the debugger cannot report the expected value of $a(i,j)$ regardless of where the breakpoint is placed in the optimized code. The debugger does not know if the value has been computed because the instances of S_o have been reordered and split amongst statements S'_1 and S'_2 and it has no information of how the instances of S_o correspond with instances of S'_1 and S'_2. Also, with these mappings, it is difficult to understand the optimized program, even with the original unoptimized program.

This dissertation develops a mapping technique that captures the effects of code transformations by capturing the correspondences between statement instances. Since the mappings track the instances of statements, the mappings are able to capture the effects of transformations, including loop transformations. This extra information is needed to develop more powerful source level debugging tools for optimized code that utilize both static and dynamic information.

Chapter 3
The Effects of Program Transformations

Source level debugging tools for optimized code allow a user to debug optimized code from the point of view of the source program. To develop such tools, a correspondence between the source and optimized code must be established. Establishing a correspondence between a source program and the optimized code requires determining the effects of the applied program transformations. The class of program transformations considered in this work include statement level optimizations, loop transformations, and inlining. Statement level optimizations include speculative code motion and path sensitive optimizations. These transformations move, modify, insert, and delete statements in the program code and affect program statements in a number of ways. A transformation can affect the position of a statement. A statement can be moved to an earlier/later position in the optimized program. In this case, the statement would execute in the optimized code before/after it does in the unoptimized version of the source code. A transformation can affect the number of times a statement executes in the optimized code. A statement moved in the optimized code may execute more or less times than it does in the unoptimized code. In fact, a statement may execute in the unoptimized program but not in the optimized program, and vice versa. Finally, a transformation can affect the order in which multiple instances of a statement are executed.

Since a source program has a direct correspondence with the unoptimized version of the program, the remainder of this chapter determines the effects of transformations that must be captured so that a correspondence between the unoptimized and optimized code can be established. The effects of transformations are determined by analyzing how the position, number, and order of instances of a statement can change, given a particular context, for statement level optimizations, loop transformations, and inlining.

3.1 Terminology

The correspondence between the unoptimized and optimized versions of a source program is actually the correspondences between statements and their instances in both the unoptimized and optimized programs.

Definition 3.1 An execution of a statement S is called an *instance* of S.

Definition 3.2 Let S be a statement in the unoptimized version of a source program and S' be a statement in the optimized version, which was derived from S by program transformations. If there exists some instance i of S and some instance j of S', denoted by S_i and S'_j, such that they should compute the same value, then there is a *correspondence* between S and S' and S_i corresponds with S'_j.

If a statement is moved across a branch or loop boundary, the correspondence between instances of the statement in the unoptimized and optimized programs depends on the execution path taken.

Definition 3.3 Let S be a statement in the unoptimized version of a source program and S' be a statement in the optimized version such S and S' correspond. If S and S' have different control dependences, then corresponding instances of S and S' are *path sensitive.*

3.2 Statement level optimizations

Statement level optimizations (e.g., constant propagation, loop invariant code motion, dead code elimination, and partial redundancy elimination) operate on individual statements. These optimizations modify, delete, and move statements in the program code, and may affect the position and the number of instances of a statement in the program code. Therefore, they may affect the correspondences between statements in the unoptimized and optimized code.

3.2.1 The effects of modifying statements

Some optimizations simply modify statements for efficiency purposes and do not affect the correspondences between statements in the unoptimized and optimized programs. For example, the constant propagation optimization propagates a constant that is assigned to a variable by replacing the uses of the variable with the constant. In the unoptimized code in Figure 3.1, the variable a is assigned the constant 5, which can be propagated to statement S. After constant propagation is applied, the corresponding statement S' in the optimized code uses the constant 5 as one of its operands instead of the variable a, but the correspondence between S and S' is not affected.

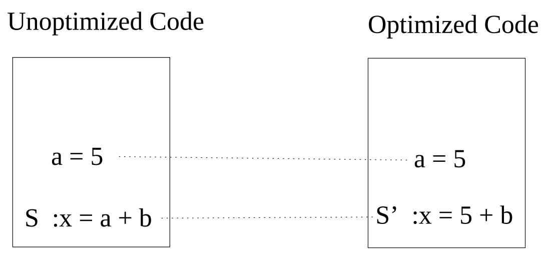


Figure 3.1: Constant propagation example

3.2.2 The effects of deleting statements

If an optimization removes a statement from the program code, the deleted statement would not execute in the optimized code and the deleted statement has no correspondence in the optimized code. For example, the application of dead code elimination removes statements that are never used in a program. As illustrated in Figure 3.2, the application of dead code elimination removes statement S. Thus, S has no correspondence in the optimized code and would not execute in the optimized code.

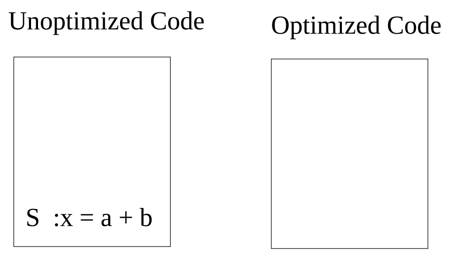


Figure 3.2: Dead code elimination example

3.2.3 The effects of moving statements

Transformations that move statements to different positions in the program code are code motion transformations, which include simple reordering of statements in a straight-line code segment, moving statements across branch boundaries, and moving statements across loop boundaries. The change in the position of a statement causes the statement to execute earlier/later in the optimized code as compared to the unoptimized program.

The change in the position of a statement can also cause its number of instances to increase or decrease and therefore, the correspondence between statement instances can be affected. Furthermore, the correspondence between statement instances may be path sensitive. Lastly, a statement in the unoptimized code can correspond to several statements in the optimized code (and vice versa).

3.2.3.1 Code hoisting transformations

Code motion transformations that move statements to earlier positions in the program are code hoisting transformations. Statements that are moved to earlier positions in the program will execute earlier than in the unoptimized program. For example, in Figure 3.3(a), local instruction scheduling has moved statement S up in the optimized code. As a result, the instance of S corresponds with the instance of S', and S' will execute earlier than S.

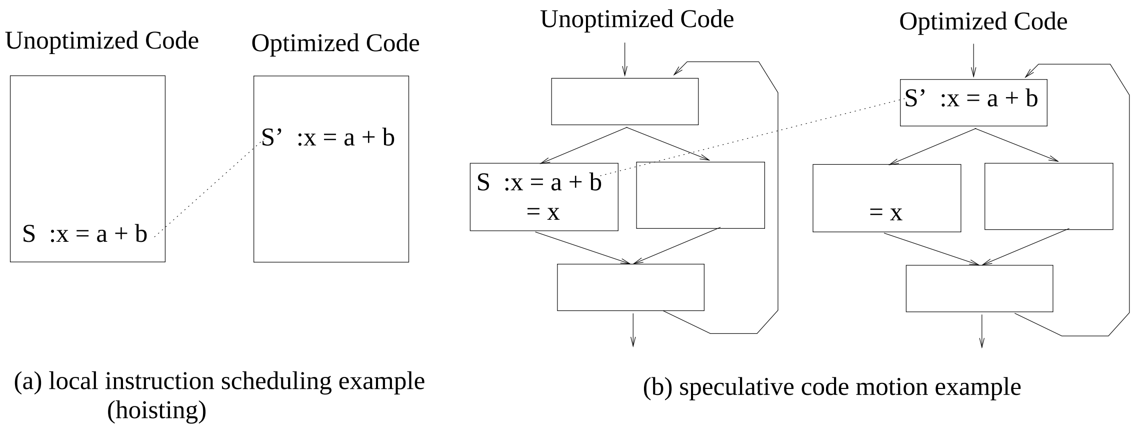


Figure 3.3: Code hoisting transformation examples

Code hoisting transformations can also move statements across branch boundaries. The effects of hoisting code across branch boundaries depend upon the nature of the transformations. For example, speculative code motion moves statements across branches (as well as loop boundaries). In Figure 3.3(b), statement S is moved across the conditional in the optimized code. Although S and S' correspond with each other, the number of times statement S' executes may be more than the number of times S executes. The correspondence between instances of S and S' is path sensitive due to the difference in control dependencies of S and S' and therefore depends on the path taken during execution. If both S and S' execute during corresponding loop iterations, then the statement instances correspond to each other. On the other hand, if S' is executed and S is not executed during corresponding loop iterations, then the instance of S' has no correspondence in the unoptimized code. The examples in Figure 3.4 illustrate more effects of code hoisting transformations. In Figure 3.4(a), S is hoisted into both conditionals. As a result, S corresponds with two statements in the optimized code and the correspondences between instances of S and the instances of S' and S'' are path sensitive due to the difference in control dependencies. Similarly, in Figure 3.4(b), S and T are hoisted above the conditional. Both S and T correspond to the same statement in the optimized code, and the correspondences between the instances of S and T and the instances of S' are path sensitive.

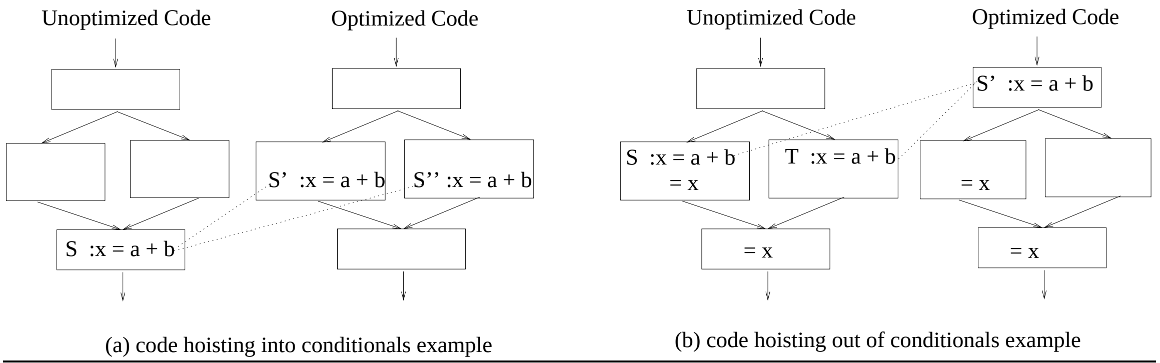


Figure 3.4: Code hoisting transformation examples cont.

Code hoisting transformations can also move statements across loop boundaries. Consider the loop invariant code motion optimization (LICM), which moves loop invariant statements out and above loops. In Figure 3.5(a), statement S is moved out and above loop $L2$ by LICM. The number of times statement S executes in the unoptimized code is greater than the number of times the corresponding statement S' executes in the optimized code, and in each iteration of $L1$, all instances of statement S correspond to an instance of S' in the optimized code.

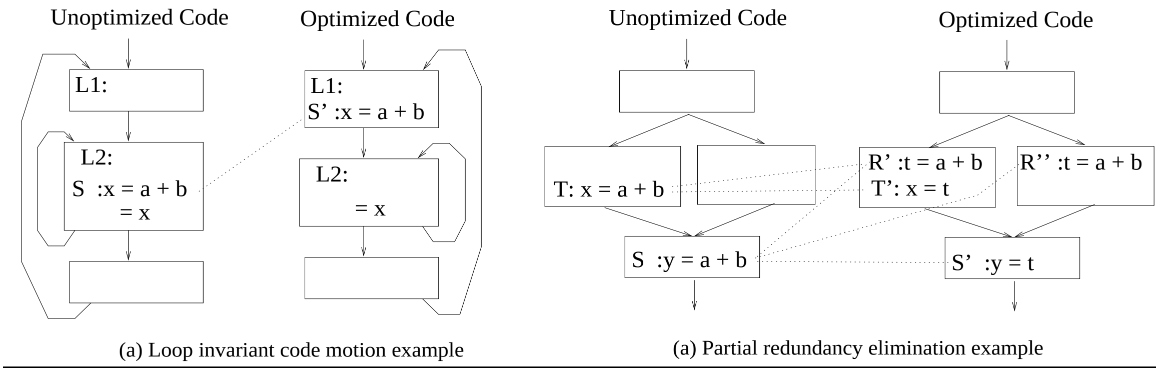


Figure 3.5: Code hoisting transformation examples cont.

Partial redundancy elimination (PRE) is also a code hoisting transformation. PRE moves and modifies computations in such a way that after the application of the transformation, the occurrences of such computations are minimized along paths. The statements that are inserted in the optimized code correspond with existing statements in the unoptimized code, and the correspondences between instances of such statements depend on the positions of the statements and therefore may be path sensitive. For example, in Figure 3.5(b), the computation $a + b$ in statement S is partially redundant with statement T in the unoptimized code. The application of PRE creates a new statement R' that assigns the partial redundant expression $a + b$ to a temporary t and uses the temporary t instead of recomputing the partial redundant expression in statements S' and T'. After PRE is applied, the correspondence between S and S' and the correspondence between T and T' are not affected. However, S corresponds with R' and R'', and the correspondences between the instances of S and R' and the instances of S and R'' are path sensitive. Similarly, T corresponds with R'.

3.2.3.2 Code sinking transformations

Code motion transformations that move statements to later positions in the program are code sinking transformations. Statements moved to later positions execute later during execution and the correspondences between statements may be affected. For example, in Figure 3.6, local instruction scheduling has moved statement S down in the code. As a result, the instance of S corresponds with the instance of S', and S' executes later than S.

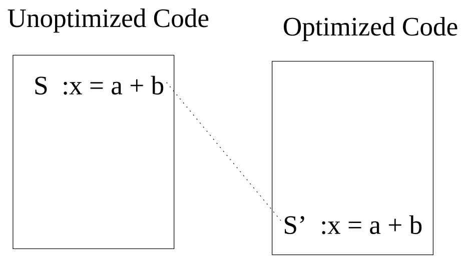


Figure 3.6: Local instruction scheduling example (sinking)

Partial dead code elimination (PDE) sinks a statement that is dead on one path but may not be dead on other paths. When statements are moved across branch boundaries, a statement in the unoptimized program corresponds to one or more statements in the optimized program. In Figure 3.7(a) the application of PDE moves statement S inside the conditional. As a result, the number of times statement S' executes may be less than the number of times S executes. The correspondence between instances of S and S' is path sensitive. If both S and S' execute, then the statement instances correspond to each other.

On the other hand if S is executed and S' is not executed, then the transformation has resulted in the deletion of the instance of S in the execution.

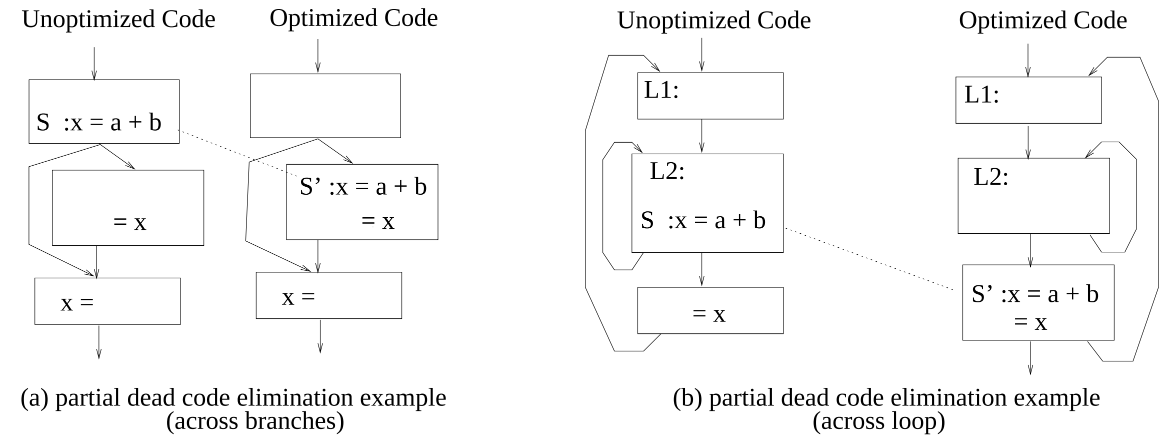


Figure 3.7: Code sinking transformation examples

In Figure 3.7(b) statement S is moved out and below loop $L2$ by PDE. As a result, statement S' executes a fewer number of times than the corresponding statement S. During each iteration of loop $L1$, only the last instance of statement S has a corresponding instance in the optimized program, which is the instance of S'. All earlier instances of statement S have been deleted by PDE.

3.2.3.3 Summary of moving statements

In summary, the application of code motion transformations affects the correspondence between the unoptimized and optimized versions of a program in the following ways:

- Since statements can be reordered, a statement in the unoptimized code can execute before/after its corresponding statement executes in the optimized code.
- New correspondences can be established to the statements in the unoptimized code. Thus, a statement in the unoptimized code can correspond to one or more statements in the optimized code.
- New correspondences can be established to the statements in the optimized code. Thus, a statement in the optimized code can correspond to one or more statements in the unoptimized code.
- The number of instances of a statement in the unoptimized code can increase or decrease in the optimized program. Thus, the correspondence between statement instances can change. One or more instances of a statement in the unoptimized code

can correspond to one instance in the optimized code. Instances of a statement in the unoptimized code can have no correspondence in the optimized code.

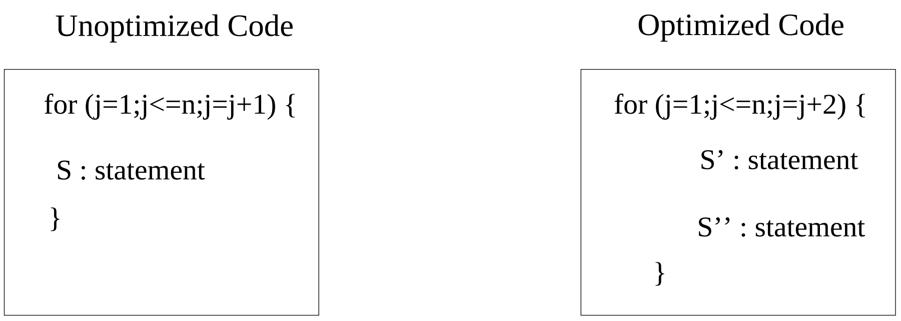
- Two corresponding statements can have different control dependences. Thus, the correspondence between instances of a statement in the unoptimized and optimized programs can be path sensitive. Instances of a statement in the unoptimized code may or may not have a correspondence in the optimized code, and vice versa. The establishment of corresponding instances of statements may only be established at runtime.

3.3 Loop transformations

Loop transformations operate on loops as a unit and have the same effects as statement level optimizations as well as other effects. The application of loop transformations can duplicate loop bodies, modify the iteration space, merge and split loops, and alter the index and bounds of a loop As a result, the correspondences between statements in the unoptimized and optimized programs are affected because the instances of a statement in the unoptimized program can be reordered and distributed among several statements in the optimized program. Also, during execution of the unoptimized and optimized programs, corresponding statements may execute earlier/later and in a different order, and the number of their instances may differ.

3.3.1 Effects of duplicating loop bodies

Loop transformations that duplicate bodies of loops affect the correspondences between statements in the unoptimized and optimized programs (e.g., loop peeling, loop unrolling, and software pipelining). Since statements in the loop body in the unoptimized program are replicated in the optimized code, the instances of each such statement in the loop body in the unoptimized program are divided among several statements in the optimized program. Also, the loop in the unoptimized program executes more iterations than the corresponding loop in the optimized program. For example, loop unrolling replaces a loop body by several copies of the loop body. The number of copies is called the unrolling factor, and the loop increment is adjusted to increment by the unrolling factor. Consider the example in Figure 3.8 where the loop is unrolled two times and the loop header is modified to iterate half the time. Notice the statement in the loop body in the unoptimized code corresponds to two statements in the optimized code: the odd instances of the statement in the loop body in the unoptimized code correspond to the instances of one statement in the optimized code, and the even instances of the statement in the loop body in the unoptimized code correspond to the instances of the other statement in the optimized code. Notice both loop initializations correspond. The loop tests correspond in that the odd instances of the loop test in the unoptimized code correspond to the instances of the loop test in the optimized code, and the last instances of both loop tests correspond. The loop increments correspond in that the even instances of the loop increment in the unoptimized code correspond to the instances in the optimized code.



```
for (j=1;j<=n;j=j+1) {
 S : statement
}
```

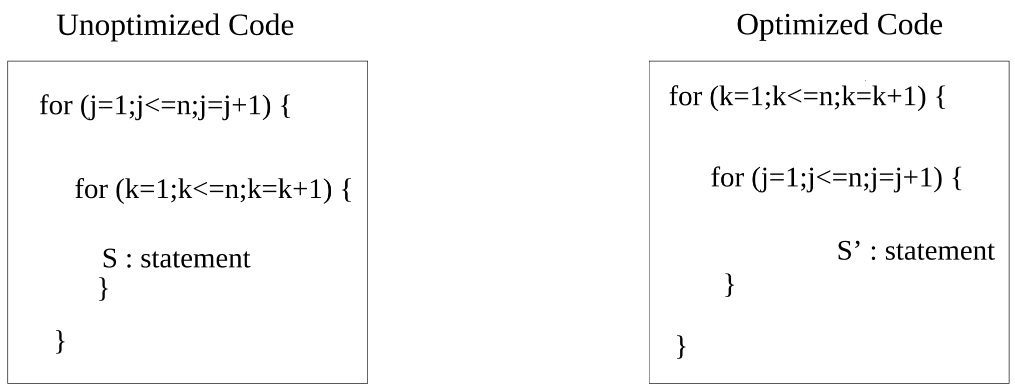
```
for (j=1;j<=n;j=j+2) {
        S' : statement

        S'' : statement
    }
```

Figure 3.8: Loop unrolling example

3.3.2 Effects of modifying the iteration space

Loop transformations can reorder the instances of statements within loops (e.g., loop reversal, loop interchange, strip mining). As a result, the iteration space of statements in the unoptimized program can differ from that of the optimized program. For example, loop interchange exchanges the positions of two loops in a loop nest, which changes the order of loop iterations in the optimized code. In Figure 3.9, the two loops in the unoptimized program are interchanged in the optimized program. Although the statements within the loop bodies remain within the same loops in both programs and therefore execute the same number of times, the execution order of their instances differ in both programs. Also, the loop headers of both loops are affected. The corresponding loop headers of the interchanged loops appear in different loop nest levels and thus, the number of instances of the corresponding loop headers differ in both programs. The statement instances of the header of the outer loop in the unoptimized code will execute more often in the optimized code because they are now in an inner loop in the optimized code. The statement instances of the loop header of the inner loop in the unoptimized code will execute less often in the optimized code because they are now in an outer loop in the optimized code.

Another loop transformation that affects the iteration space is strip mining, which converts a serial loop into several loops (strips). The strips are essentially a series of

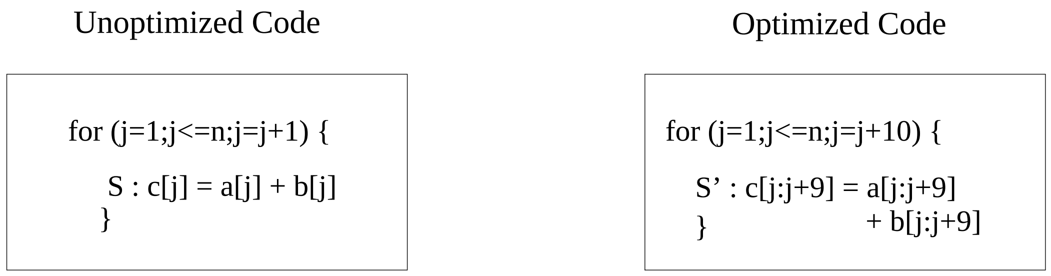


```
for (j=1;j<=n;j=j+1) {

    for (k=1;k<=n;k=k+1) {

        S : statement
        }

    }
```

```
for (k=1;k<=n;k=k+1) {

    for (j=1;j<=n;j=j+1) {

                    S' : statement
      }

}
```

Figure 3.9: Loop interchange example

vector operations. The application of strip mining affects the correspondences between statements in the unoptimized and optimized programs in that instances of statements in the unoptimized code are grouped together into one instance in the optimized code. For example, in Figure 3.10, the loop has been strip mined with a strip size of 10. The loop initializations in both programs correspond. Every tenth instance of the loop test and increment in the unoptimized code correspond with each instance of the loop test and increment in the optimized code. Also, the last instances of the loop tests in both programs correspond. Finally, every 10 instances of the assignment to c in the unoptimized code correspond with one instance of the vector assignment to c in the optimized code.



```
for (j=1;j<=n;j=j+1) {

    S : c[j] = a[j] + b[j]
    }
```

```
for (j=1;j<=n;j=j+10) {

    S' : c[j:j+9] = a[j:j+9]
    }                + b[j:j+9]
```

Figure 3.10: Strip mining example

3.3.3 Effects of merging and splitting loops

Loop transformations can split loops and merge loops together. Although the control flow changes, the number and order of the instances of the statements in the loop bodies are not affected. Loop distribution divides a loop into two or more loops with the same loop headers. The statements in each loop in the optimized code enclose a subset of the statements in the loop in the unoptimized code. Also the loop header in the unoptimized code corresponds with both loop headers in the optimized code. For example, in Figure 3.11, the loop in the unoptimized code has been distributed in the optimized code. Notice each instance of the loop headers in the unoptimized code corresponds to two instances in the optimized code.

Unoptimized Code

```
for (j=1;j<=n;j++)
     a[j] = x * j
     b[j] = y * j
     c[j] = a[j] + b[j]
 }
```

Optimized Code

```
for (j=1;j<=n;j++)
     a[j] = x * j
     b[j] = y * j
  }
for (j=1;j<=n;j++)
     c[j] = a[j] + b[j]
  }
```

Figure 3.11: Loop distribution example

3.3.4 Effects of altering the index and bounds of a loop

Loop transformations can alter the loop headers of a loop and yet the number and order of the instances of the statements in the loop bodies remain the same. For example, loop normalization changes the loop header of a loop so that the loop's index is initially 1 and is incremented by 1 on each iteration. In Figure 3.12, the loop in the unoptimized code has been normalized in the optimized code. The statements within the loop bodies execute the same number of times and in the same order. The loop initializations and the loop increments have no correspondences, and although the loop test has changed in the optimized code, there is a correspondence between the instances of the loop tests in both programs.

Unoptimized Code

```
for (j=init;j<=limit;j=j+step) {
   S : statement
 }
```

Optimized Code

```
for (j=1;j<= ⌊(limit - init + step)/step⌋; j=j+1  ) {
   S' : statement
 }
```

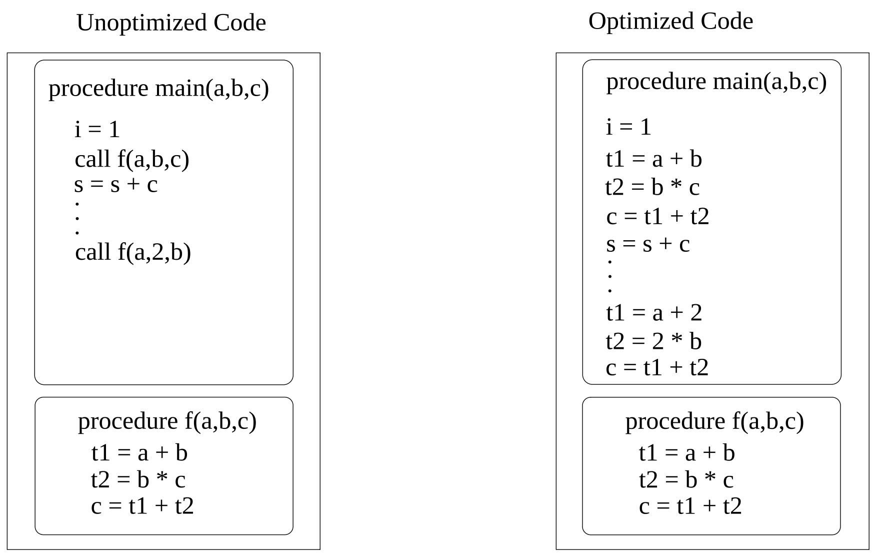
Figure 3.12: Loop normalization example

3.3.5 Summary of loop transformations

The effects of loop transformations affect the correspondence between the unoptimized and optimized versions of a program the same way as that of statement level transformations. Moreover, the correspondences between statement instances can change as a result of reordering the instances of a statement and dividing the instances of a statement in the unoptimized code among several statements in the optimized code.

3.4 Inlining

Function inlining replaces calls to a function in the unoptimized code by the bodies of the function in the optimized code. The instances of the statements in the function in the unoptimized code correspond with the instances of the statements that were inlined in the optimized code, and the instances of the call sites in the unoptimized code that were inlined in the optimized code have no correspondences in the optimized code. For example, in Figure 3.13, function *f* has been inlined two times. Each statement in the function in the unoptimized code corresponds with two statements in the optimized code, and the statements in the function in the unoptimized code execute the same number of times as the corresponding statements in the optimized code.



```
procedure main(a,b,c)
  i = 1
  call f(a,b,c)
  s = s + c
  .
  .
  .
  call f(a,2,b)

procedure f(a,b,c)
  t1 = a + b
  t2 = b * c
  c = t1 + t2
```

```
procedure main(a,b,c)
  i = 1
  t1 = a + b
  t2 = b * c
  c = t1 + t2
  s = s + c
  .
  .
  .
  t1 = a + 2
  t2 = 2 * b
  c = t1 + t2

procedure f(a,b,c)
  t1 = a + b
  t2 = b * c
  c = t1 + t2
```

Figure 3.13: Inlining example

3.5 Summary

This chapter described the effects of transformations that impact the correspondences between statements in the unoptimized and optimized programs. The effects of transformations were established by analyzing how the position, number, and order of instances of a statement can change, given a particular context, for statement level optimizations, loop transformations, and inlining. The application of these transformations affects the correspondence between the unoptimized and optimized versions of a program in the following ways.

- Statements and statement instances may have no correspondences.
- A statement in the unoptimized code can be relatively positioned before/after its corresponding statement in the optimized code. Therefore, a statement in the unoptimized code can execute before/after its corresponding statement executes in the optimized code.
- New correspondences can be established. Thus, a statement in the unoptimized code can correspond to one or more statements in the optimized code, and a statement in the optimized code can correspond to one or more statements in the unoptimized code.
- Since the number of instances of a statement in the unoptimized code can increase or decrease in the optimized program, the correspondences between statement instances in the unoptimized and optimized programs are not necessarily a one-to-one correspondence.
- The instances of a statement in the unoptimized code can be reordered in the optimized code.
- The instances of a statement in the unoptimized code can be divided among several statements in the optimized code.
- The correspondence between instances of a statement in the unoptimized and optimized programs can be path sensitive. Therefore, the instances of a statement in the unoptimized code may or may not have a correspondence in the optimized code, and vice versa, and the establishment of corresponding instances of statements may only be established at runtime.

To establish a correspondence between the unoptimized and optimized versions of a program, these effects are captured through mappings, which are discussed in the next chapter.

Chapter 4

Capturing the Effects of Program Transformations Through Mappings

The previous chapter described the effects of program transformations that affect the correspondence between the unoptimized and optimized versions of a program. In this chapter, these effects are captured through mappings to establish the correspondences between the statements in the unoptimized and optimized versions of a program. Mappings are established as transformations are applied. Since a number of transformations may be applied and in any order, the mappings reflect the combined effects of transformations. The mappings do not record the individual transformations applied nor the order in which they were applied. Instead, the mappings between the unoptimized and optimized programs at any time during optimization summarize the effects of all previously applied transformations.

Since program transformations can change the correspondences between statements and instances of statements in the unoptimized and optimized programs, the mappings associate corresponding statements and corresponding instances of statements in the unoptimized and optimized programs. The mappings can associate a statement in one program with zero, one, or more statements in the other program. Similarly, the mappings can associate an instance of a statement in one program with zero, one, or more instances of a statement in the other program. Since program transformations can divide the instances of a statement in the unoptimized code among several statements in the optimized code, the mappings can identify sequences of instances of a statement. Also, since program transformations can reorder the instances of a statement, the mappings can identify ordered sequences of instances of a statement. Although program transformations can change the relative position of a statement, the mappings do not explicitly capture this change of position. Instead, the mappings and the control flow graphs of the unoptimized and optimized programs can be analyzed to determine the relative positions of corresponding statements.

Similarly, the mappings and the control flow graphs can be analyzed to determine the correspondences between instances of statements that are path sensitive.

4.1 Mappings

Mappings are represented by labeled edges between corresponding statements in the unoptimized and optimized programs. Labels identify the instances in the unoptimized program and the corresponding instances in the optimized program. Thus, a mapping has two components: an *association of a statement* in the unoptimized code with a corresponding statement in the optimized code and an *association of instances* of the statements. A mapping of a statement S in the unoptimized program and S' in the optimized program is of the form:

ordered sequence of instances of $S \rightarrow$ *ordered sequence of instances of* S'.

The ordered sequences in the mappings express the correspondences between instances of two statements. The number of elements in the two sequences may be the same or may differ. For example, if there is an one-to-one correspondence between the instances, then the number would be the same. Corresponding instances may appear in the same order or different order (e.g., reverse order). If the number of instances is not the same, a consecutive subsequence of instances in one sequence corresponds to a single instance in the other. It should be noted that corresponding statement instances are computed statically but the mappings are between all potential dynamic instances. All of the instances in both sequences may not execute, but for the instances that do execute, the mappings capture the dynamic correspondences.

To refer to one or more instances of a statement, the loop iterations in which the instances execute are specified, as the number of instances of a statement executed is governed by the loops enclosing the statement and these instances are ordered by the order of the iterations of the loops. Therefore, each statement in the program is viewed with respect to the looping structure in which it is enclosed. Without loss of generality, a program is assumed to be enclosed within a loop of one iteration, denoted by L_0. A statement S is identified as being nested within a loop nest $L = L_0, L_1, \ldots, L_n$ where L is a collection of loops enclosing S, numbered successively from the outermost to the innermost loop, and $n+1$ is the number of loop nest levels. Each iteration of loop nest L uniquely identifies instances of statement S, and instances of statement S are ordered by the order of iterations of loop nest L. An ordered sequence of instances of a statement within loop nest $L = L_0, L_1, \ldots, L_n$ is specified by an $(n+1)$-dimensional vector. Each element in the vector is subscripted such that an element with subscript i represents an ordered sequence of iterations of loop L_i. The order in which vector elements are specified determines the order of instances in the sequence.

An element i $(0 \leq i \leq n)$ in a vector is of the following form:

"*one*" denotes *an instance* of S that executes in *each iteration* of loop L_i.

"*all*" denotes *all instances* of S that execute in *all of the iterations* of loop L_i.

"*last*" denotes *the instance* of S that executes in the *last iteration* of loop L_i.

"c" denotes the instance of S that executes in the c^{th} iteration of loop L_i, where c is a constant.

"$\{lower, upper, step\}$" and $step \geq 0$ denotes the instances of S that execute in the increasing sequence of iterations $(lower, lower + step, lower + 2 * step, \ldots, end)$ where $((end \leq\ upper$ and $(end + step) > upper))$ of loop L_i.[1]

Let S represent a statement in the unoptimized program, S' a statement in the optimized program, and $S_{(i)}$ and $S'_{(i)}$ denote instance i of statement S and S' respectively. Examples of mapping labels generated from transformations are:

- Code reordering: $(one_0, one_1) \rightarrow (one_0, one_1)$ indicates that for each iteration (i, j) of a loop nest, $S_{(i,j)}$ corresponds with $S'_{(i,j)}$.
- Loop reversal: $one_0, \{10, 1, -1\}_1 \rightarrow one_0, one_1$ indicates that for each iteration i, j of a loop nest, $S_{i,(11-j)}$ corresponds with $S'_{i,j}$.
- Loop invariant code motion: $(one_0, one_1, all_2) \rightarrow (one_0, one_1)$ indicates that for each iteration (i, j, k) of a loop nest of L_0, L_1, and L_2, $S_{(i,j,k)}$ corresponds with $S'_{(i,j)}$. In Figure 4.1, $(one_0, one_1, \mathbf{all_2})$ instances of S map to (one_0, one_1) instances of S', indicating that in each iteration of L_0 and L_1, **all** instances of S correspond to the **one** instance of S'. The vector describing the instances of S has three dimensions because programs are assumed to be implicitly enclosed within a loop of one iteration, L_0.
- Partial dead code elimination: $(one_0, one_1, last_2) \rightarrow (one_0, one_1)$ indicates $S_{(i,j,last)}$ corresponds with $S'_{(i,j)}$. In Figure 4.2, $(one_0, one_1, \mathbf{last_2})$ instances of S map to (one_0, one_1) instances of S', indicating that in each iteration of L_0 and L_1, the **last** instance of S corresponds to the **one** instance of S'.

[1]A decreasing sequence can also be denoted similarly.

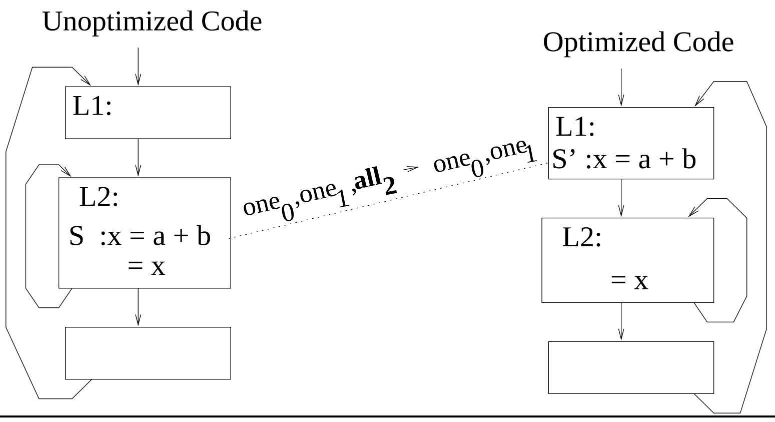


Figure 4.1: Loop invariant code motion mapping example

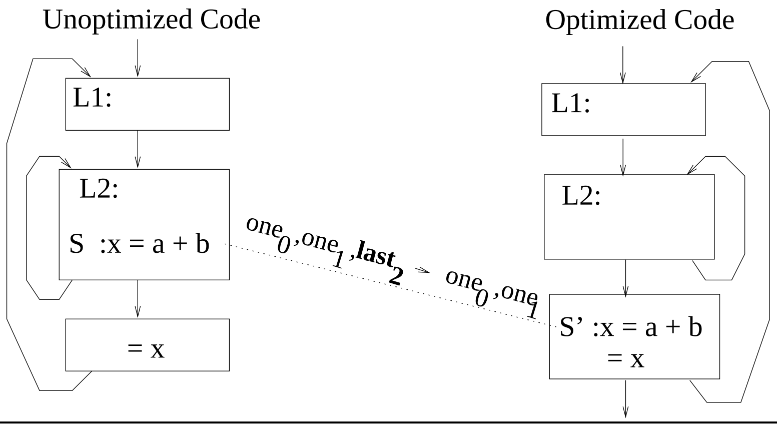


Figure 4.2: Partial dead code elimination mapping example

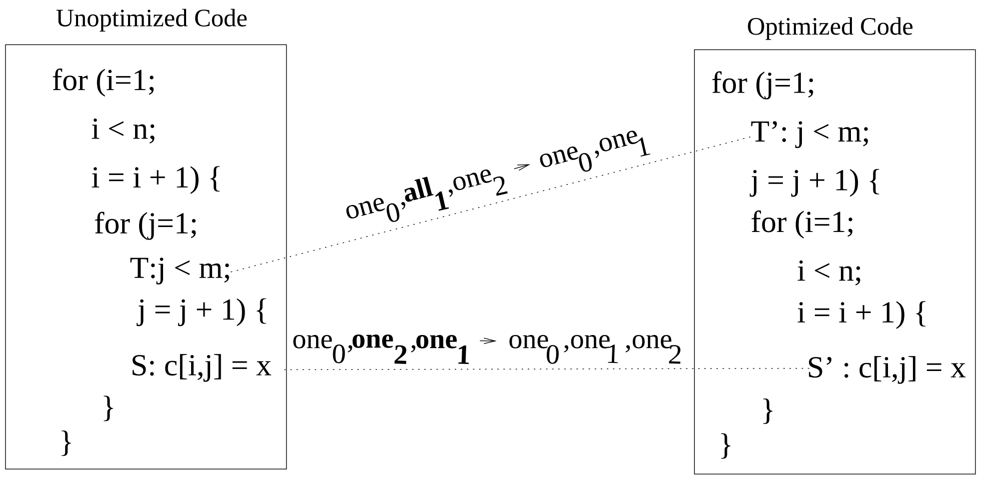


Figure 4.3: Loop interchange mapping example

- Loop interchange (on the loop body):
  $(one_0, one_2, one_1) \rightarrow (one_0, one_1, one_2)$ indicates $S_{(i,k,j)}$ corresponds to $S'_{(i,j,k)}$. In Figure 4.3, $(one_0, \mathbf{one_2}, \mathbf{one_1})$ instances of S map to $(one_0, \mathbf{one_1}, \mathbf{one_2})$ instances of S', indicating that in each iteration of L_0, L_1, and L_2, $S_{i,\mathbf{k},\mathbf{j}}$ corresponds to instance $S'_{i,\mathbf{j},\mathbf{k}}$.
- Loop interchange (on the loop header):
  $(one_0, all_1, one_2) \rightarrow (one_0, one_1)$ indicates $S_{(i,j,k)}$ corresponds with $S'_{(i,k)}$, as illustrated in Figure 4.3.

For readability and ease of explanation, the following notations are used to refer to vectors in the rest of this chapter. When all of the elements of an $n+1$-dimensional vector are of the same form, the shorthand vector notation $\rightarrow$ can be used. For example, $(one_0, one_1, \ldots, one_n)$ can be denoted by $\overrightarrow{one}$. This notation is used when the loop nesting level of a vector is not important for the explanation of how mappings are generated. In some cases, a consecutive sequence of elements of a vector are of the same form. In this case, a subscript range $m..n$ can be used to refer to these elements. For example, $(one_0, one_1, \ldots, one_n)$ can be denoted by $one_{0..n}$. Finally, in some cases, when the sequence of instances that an element i of a vector refers to is not important for the explanation of how mappings are generated, $\langle * \rangle_i$ is used to refer to the sequence of instances of element i.

The optimized program initially starts as an identical copy of the unoptimized program with initial mappings between corresponding statements in the two programs. Initially, all of the mappings have $\overrightarrow{one} \rightarrow \overrightarrow{one}$ labels because corresponding statements are enclosed by the same loops. These mappings change as code transformations are applied. The mappings for individual transformations are determined by using the semantics of those transformations with respect to the unoptimized program. From the mappings of individual transformations, the mappings for any series of transformations are determined. As a subsequent code transformation is applied, the mappings are changed to reflect the composition of the previous mappings (the effects of all previously applied transformations) by the effects of the current transformation.

4.2 Generating mappings

Code transformations can be applied in any order and as many times as desired and applicable. After a code transformation is applied, the label of a mapping may change and/or a new mapping may be established. The label of a mapping depends on the applied code transformation, positions of corresponding statements, and the mapping of the affected statement. This section describes the effects on mappings after a single (initial) transformation is applied. The subsequent section describes the effects on mappings after a series of transformations are applied.

4.2.1 Effects of statement level optimizations

The next several tables show the effects on mappings as a result of applying an initial statement level optimization to a statement S in the unoptimized program. The initial mapping of S to a corresponding statement S' in the optimized program is of the form $one_{0..n} \rightarrow one_{0..n}$.

Table 4.1: Dead code elimination effects on mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels

Transformation	Resulting mapping of S
dead code elimination	delete

Table 4.2: Statement level optimization effects that do not affect the mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels

Transformation	Resulting mapping label of S
code reordering within basic block	$one_{0..n} \longrightarrow one_{0..n}$
speculative hoisting in an acyclic scheduler	$one_{0..n} \longrightarrow one_{0..n}$
constant propagation and folding	$one_{0..n} \longrightarrow one_{0..n}$
copy propagation	$one_{0..n} \longrightarrow one_{0..n}$
partial dead code elimination (S within same loop)	$one_{0..n} \longrightarrow one_{0..n}$
partial redundancy elimination (S within same loop)	$one_{0..n} \longrightarrow one_{0..n}$

Table 4.1 displays the effects on mappings as a result of applying dead code elimination. This transformation causes the removal of mappings because corresponding statements in the optimized program are deleted.

The application of the statement level optimizations displayed in Table 4.2 do not affect the mappings. The effects of the applications of code reordering within a basic block, speculative hoisting in an acyclic scheduler, constant propagation and folding, copy propagation, partial dead code elimination (where corresponding statements are within the same loops), and partial redundancy elimination (where corresponding statements are within the same loops) do not change the mappings nor the labels because statements are

Table 4.3: Statement level optimizations which move statements to outer loops and affect mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels

Transformation	Resulting mapping label of S
loop invariant code motion (S is in an inner loop)	$one_{0..n-1}, \mathbf{all_n} \longrightarrow one_{0..n-1}$
partial dead code elimination (S is in an inner loop)	$one_{0..i}, \mathbf{last_{i+1..n}} \longrightarrow one_{0..i}$
partial redundancy elimination (S is in an inner loop)	$one_{0..i}, \mathbf{all_{i+1..n}} \longrightarrow one_{0..i}$

not moved across loop boundaries and new mappings that are established do not have corresponding statements positioned across loop boundaries.

Table 4.4: Partial redundancy elimination effects on mappings of S with $one_{0..n} \rightarrow one_{0..n}$ labels

Partial redundancy elimination	Resulting mapping label of S
(S is in an outer loop)	$one_{0..n} \longrightarrow one_{0..n}, \mathbf{last_{n+1..m}}$, where $m > n$
(S is in a different loop nest)	$one_{0..i}, \mathbf{all_{i+1..n}} \longrightarrow one_{0..i}, \mathbf{last_{i+1..m}}$

Table 4.3 displays the labels of mappings generated as a result of applying statement level optimizations that can move statements to outer loops or create correspondences with statements in outer loops. The application of loop invariant code motion moves a statement from a loop at loop nest level n to an outer loop at nesting level $n-1$. After loop invariant code motion is applied to statement S, in each iteration of this outer loop, *all* instances of S correspond to *one* instance of S'. Thus, the label of the mapping between S and S' is changed to $one_{0..n-1}, \mathbf{all_n} \rightarrow one_{0..n-1}$. The application of partial dead code elimination moves statements across branch boundaries or to an outer loop at nesting level i. When partial dead code elimination moves a statement S to an outer loop at nesting level i, in each iteration of this outer loop, the *last* instance of S corresponds to *one* instance of S'. In this case, the label of the mapping between S and S' is changed to $one_{0..i}, \mathbf{last_{i+1..n}} \rightarrow one_{0..i}$. The application of partial redundancy elimination modifies a statement S to use a temporary instead of recomputing an expression. Afterwards, statement S has a correspondence with statement R' (R' assigns the redundant expression to a temporary). When R' is in an outer loop with loop nest level i with respect to S, as illustrated in Figure 4.4(a), then in each iteration of this loop, *all* instances of S correspond with *one* instance of R'. Thus, the label of the mapping between S and R' is changed to $one_{0..i}, \mathbf{all_{i+1..n}} \rightarrow one_{0..i}$.

Table 4.4 displays the remaining possible labels of mappings generated as a result of applying partial redundancy elimination. If R' is in an inner loop with loop nest level m with respect to S, as illustrated in Figure 4.4(b), then in each iteration of the innermost loop enclosing S, *one* instance of S corresponds with the *last* instance of R'. In this case, the label of the mapping between S and R' is changed to $one_{0..n} \rightarrow one_{0..n}, \mathbf{last_{n+1..m}}$. Finally, if S and R' are in different loop nests, as illustrated in Figure 4.4(c), then in each iteration of the innermost loop enclosing both S and R', *all* instances of S correspond to the *last* instance of R'. Thus, the label of the mapping between S and R' is changed to $one_{0..i}, \mathbf{all_{i+1..n}} \rightarrow one_{0..i}, \mathbf{last_{i+1..m}}$ where i is the loop nest level of the innermost loop enclosing both S and R', n is the loop nest level of the innermost loop enclosing S, and m is the loop nest level of the innermost loop enclosing R'.

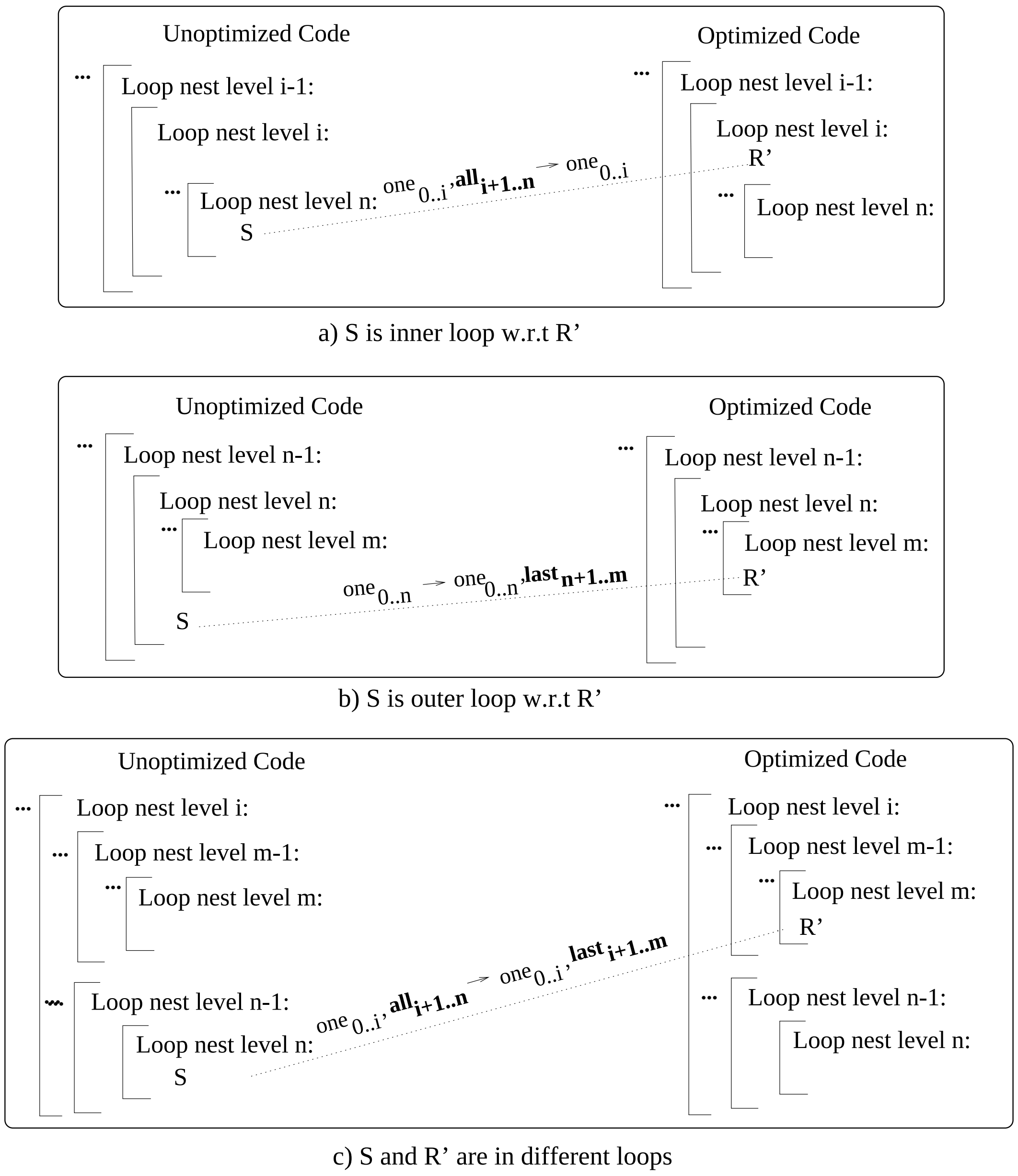


Figure 4.4: Partial redundancy elimination mapping example

4.2.2 Example

In Figure 4.5, the mappings of an unoptimized program and its optimized version are shown. This example will be used as a running example in subsequent chapters. The mappings are illustrated by labeled dotted edges between corresponding statements in both programs.

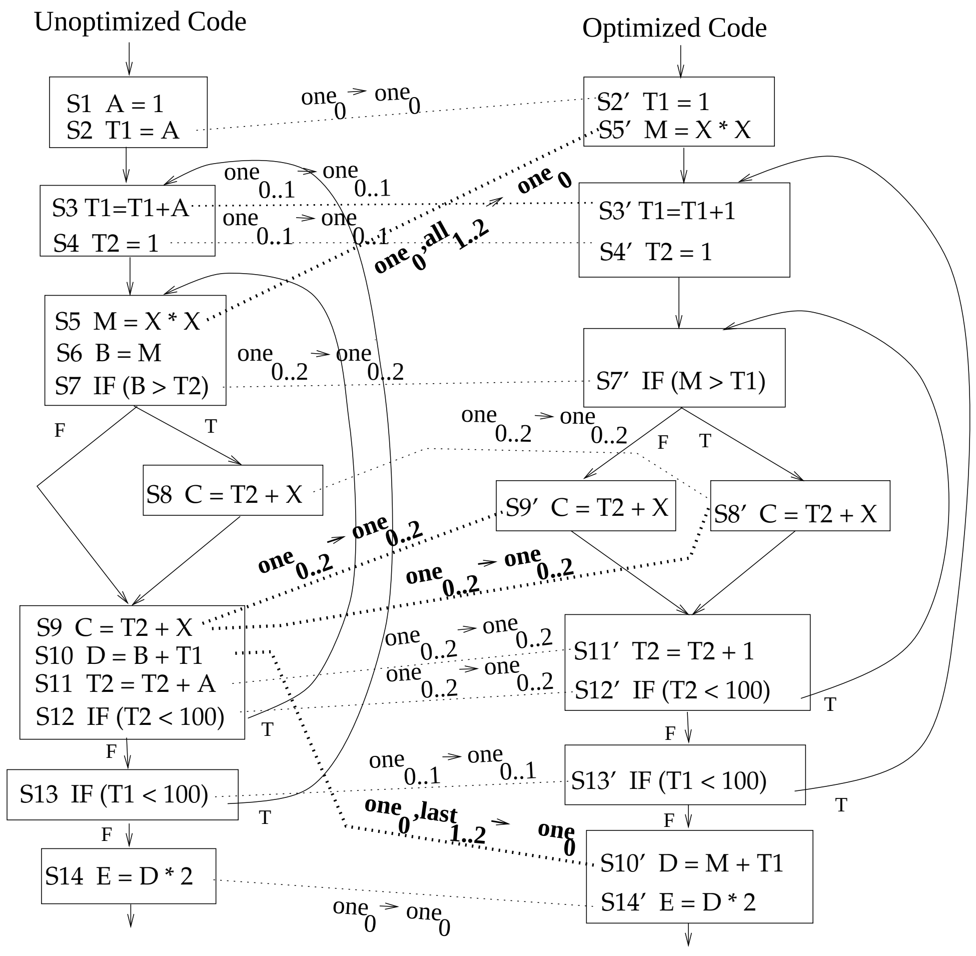


Figure 4.5: Mappings for unoptimized and optimized code example

The following optimizations were applied to the code in Figure 4.5.

- constant propagation - the constant 1 in $S1$ is propagated, as shown in $S2'$, $S3'$, and $S11'$.
- copy propagation - the copy M in $S6$ is propagated, as shown by $S7'$ and $S10'$.
- dead code elimination - $S1$ and $S6$ are dead after constant and copy propagation and thus the mappings of $S1$ and $S6$ are removed.

- loop invariant code motion - $S5$ is moved out of the doubly nested loop and thus **all** the instances of statement $S5$ in the loops in the unoptimized code must map to **one** instance of statement $S5'$ in the optimized code.

- partial redundancy elimination - $S9$ is partially redundant with $S8$, and thus, a mapping is created between $S9$ and $S8'$. Notice $S9$ now has two mappings.

- partial dead code elimination - $S10$ is moved below the outer loop and thus only the **last** instance of statement $S10$ in the loops in the unoptimized code is mapped to **one** instance of statement $S10'$ in the optimized code.

4.2.3 Effects of loop transformations

Loop transformations operate on loops as a unit, and therefore, their application affects the mappings of statements within the loops, including statements in the loop headers as well as the loop bodies. This section describes the kinds of mapping generated as a result of applying an initial loop transformation where the initial mappings are of the form $\overrightarrow{one}\rightarrow\overrightarrow{one}$.

4.2.3.1 Effects of duplicating loop bodies

When a body of a loop is duplicated, the number of instances of a statement in the loop body in the unoptimized code is divided among several statements in the optimized code. Thus, a statement in the unoptimized code corresponds to several statements in the optimized code, and the mappings are updated to reflect the new correspondences between the instances of these statements. For example, the application of loop unrolling in Figure 4.6 has unrolled the loop at nesting level i two times in the optimized code. The instances of statement S within the loop body at nesting level i in the unoptimized code are divided as follows. The odd instances of S, denoted by $\{\mathbf{1}, \mathbf{n}, \mathbf{2}\}$, correspond to the instances of S' in the optimized code, and therefore, the label of the mapping of S and S' are changed from $\overrightarrow{one}\rightarrow\overrightarrow{one}$ to $one_{0..i-1}, \{\mathbf{1}, \mathbf{n}, \mathbf{2}\}_{\mathbf{i}}, \overrightarrow{one}\rightarrow\overrightarrow{one}$. The even instances of S, denoted by $\{\mathbf{2}, \mathbf{n}, \mathbf{2}\}$, correspond to the instances of S'', and therefore, a mapping is created between S and S'' with label $one_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{2}\}_{\mathbf{i}}, \overrightarrow{one}\rightarrow\overrightarrow{one}$.

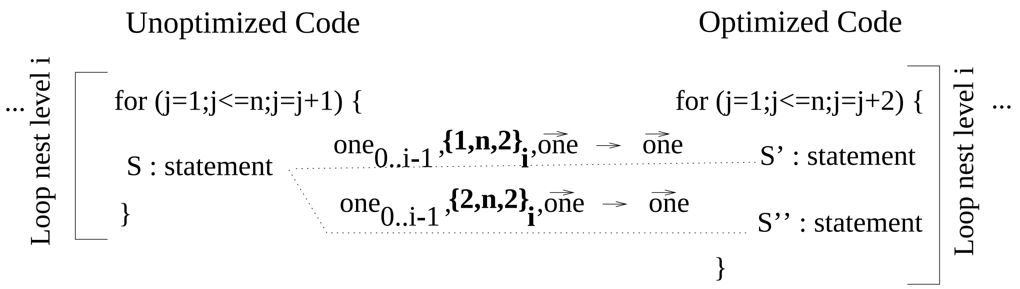


Figure 4.6: Loop unrolling

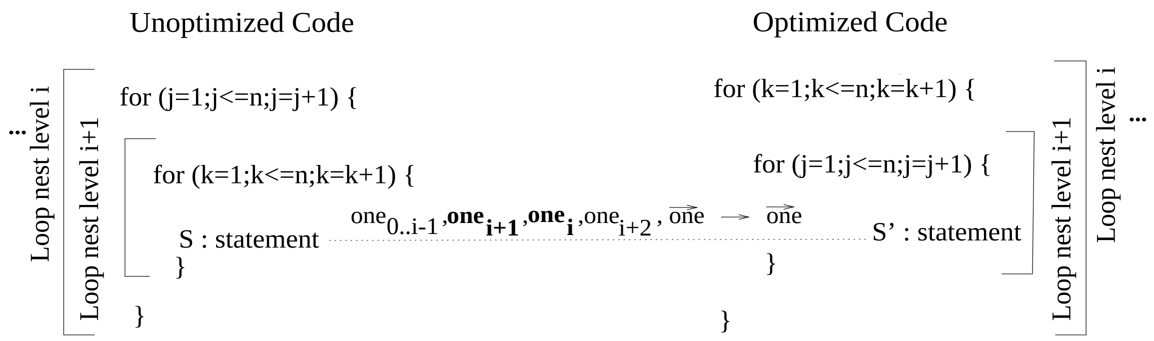
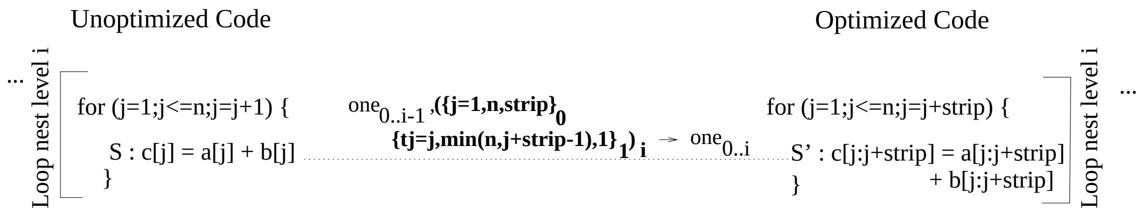


Figure 4.7: Loop interchange effects on initial $\overrightarrow{one} \rightarrow \overrightarrow{one}$ mappings



Unoptimized Code
Optimized Code
Loop nest level i
for (j=1;j<=n;j=j+1) {
S : c[j] = a[j] + b[j]
}
one0..i-1,({j=1,n,strip}0
{tj=j,min(n,j+strip-1),1}1)i → one0..i
for (j=1;j<=n;j=j+strip) {
S' : c[j:j+strip] = a[j:j+strip]
+ b[j:j+strip]
}
...

Figure 4.8: Strip mining effects on initial $\overrightarrow{one} \rightarrow \overrightarrow{one}$ mappings

4.2.3.2 Effects of modifying the iteration space

When the iteration space of a loop is reordered, the instances of a statement in the loop body in the unoptimized code are reordered in the optimized code, and the mappings are updated to reflect the new correspondences between the instances of these statements. For example, the application of loop interchange in Figure 4.7 has interchanged the loop at nesting level i with the loop at nesting level $i+1$, and thus, the instances of S are reordered in the optimized code. This reordering effect is captured by permuting the elements of the vector specified in the label of the mapping between S and S' to reflect the ordering in the optimized code. Therefore, the label of the mapping between S and S' is changed from $\overrightarrow{one} \rightarrow \overrightarrow{one}$ to $one_{0..i-1}, \mathbf{one_{i+1}}, \mathbf{one_i}, one_{i+2}, \overrightarrow{one} \rightarrow \overrightarrow{one}$.

In Figure 4.8, the loop at nesting level i has been stripped from the application of strip mining. The instances of statement S within the loop body at nesting level i in the unoptimized code are divided as follows. Instances 1 through n are divided into strips of size *strip*, and each strip of instances of S corresponds to one instance of S'. Therefore, the label of the mapping of S is changed from $\overrightarrow{one} \rightarrow \overrightarrow{one}$ to $one_{0..i-1}, (\{\mathbf{j = 1, n, strip}\}_\mathbf{0}$ $\{\mathbf{tj = j, min(n, j + strip - 1), 1}\}_\mathbf{1})_\mathbf{i} \rightarrow one_{0..i}$. The indices in the sequences reflect the dependence of the inner loop limits on the outer loop index.

4.2.3.3 Effects of merging and splitting loops

Merging or splitting loops does not reorder and split the instances of statements within the loops. Subsequently, the mappings of the statements are not affected. However, the instances of statements of the loops are reordered with respect to other instances of statements in the loops.

4.2.3.4 Effects of altering the index and bounds of a loop

Altering the index and bounds of a loop does not reorder and split the instances of statements within a loop body. Subsequently, the mappings of the statements are not affected. For example, the application of loop normalization, as illustrated in Figure 4.9, does not affect the mappings of the statements of the loop bodies and thus, the label of the mapping between S and S' remains $\overrightarrow{one} \rightarrow \overrightarrow{one}$.

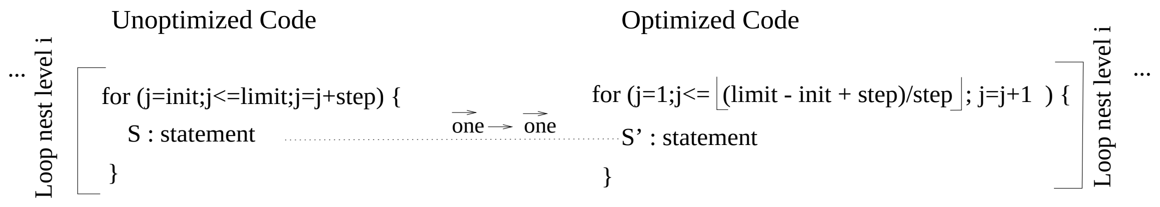


Figure 4.9: Loop normalization effects on initial $\overrightarrow{one}\rightarrow\overrightarrow{one}$ mappings

4.3 Series of code transformations

Code transformations can be applied in any order and as many times as desired and applicable. As transformations are applied on statements, a mapping's label is changed to reflect the composition of the previous mapping (the effects of all previously applied transformations) by the effects of the current transformation. Table 4.5 shows the effects of statement level optimizations on mappings of statements with label $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$. The application of dead code elimination eliminates mappings while the application of code reordering within a basic block, speculative hoisting in an acyclic scheduler, constant propagation and folding, and copy propagation does not affect the label of the mappings, and thus the label remains $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$.

Table 4.5: Statement level optimizations effects on $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$ mappings

Transformation	Resulting mapping label
dead code elimination	mapping delete
code reordering within a basic block	$\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$
speculative hoisting in an acyclic scheduler	$\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$
constant propagation and folding	$\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$
copy propagation	$\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$

Tables 4.6 and 4.7 show the effects of loop invariant code motion and partial dead code elimination (when statements are moved across loop boundaries) on the mapping labels. When a statement S' is moved in the optimized program, the mapping of S' and corresponding statement S (in the unoptimized program) is updated to reflect the composition of the previous mapping by the effects of the current transformation. Row two assumes statements S and S' are within the same loop before the transformation is applied. Row three assumes statement S is in an inner loop with respect to S', and row four assumes either statement S is in an outer loop with respect to S' or both are in different loop nests. When statements are moved out of loops after an application of loop

invariant code motion, the last element $\langle * \rangle_m$ in the vector for S' is removed. If the loop of this element also encloses S, then *all* is applied to the current element $\langle * \rangle_m$ in the vector of S. Similarly, when statements are moved across loop boundaries to a loop at nesting level j after the application of partial dead code elimination, elements $\langle * \rangle_{j+1..m}$ in the vector for S' are removed. If the loop enclosing element $j+1$ of S' also encloses S, then *last* is applied to the current elements $\langle * \rangle_{j+1..n}$ in the vector of S. When statements are moved within the same loop for partial dead code elimination, the labels of the mappings are not affected.

Table 4.6: Loop invariant code motion effects on mappings

Initial mapping label	Resulting mapping label
S is in same loop as S': $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..m-1}, all(\langle * \rangle_m)_m \rightarrow \langle * \rangle_{0..m-1}$
S is in an inner loop: $\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..m-1}, all(\langle * \rangle_m)_m, \langle * \rangle_{m+1..n} \rightarrow \langle * \rangle_{0..m-1}$
S is in an outer or different loop nest: $\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..m-1}$

Table 4.7: Partial dead code elimination effects on mappings (when statements are moved across loop boundaries)

Initial mapping label	Resulting mapping label
S is in same loop as S': $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..j}, last(\langle * \rangle_{j+1})_{j+1}..last(\langle * \rangle_m)_m \rightarrow \langle * \rangle_{0..j}$
S is in an inner loop: $\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..j}, last(\langle * \rangle_{j+1})_{j+1}..last(\langle * \rangle_n)_n \rightarrow \langle * \rangle_{0..j}$
S is in an outer or different loop nest: $\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..m}$	$\langle * \rangle_{0..n} \rightarrow \langle * \rangle_{0..j}$ (if S is still in outer or different loop nest) $\langle * \rangle_{0..j}, last(\langle * \rangle_{j+1})_{j+1}..last(\langle * \rangle_n)_n \rightarrow \langle * \rangle_{0..j}$ (if S is now in an inner loop)

Similarly, for partial redundancy elimination, after statement S is modified to use a temporary instead of recomputing an expression, statement S corresponds with S' and R' (R' assigns the redundant expression to a temporary). Therefore, a mapping is created between S and R', and the mapping of S and S' is utilized to create the mapping between S and R'. Table 4.8 displays the labels of the mapping between S and R', given the previous $\langle * \rangle_{0..m} \rightarrow \langle * \rangle_{0..n}$ mapping of S and S'.

Finally, the tables shown in Figures 4.9 and 4.10 display the labels of mappings generated as a result of applying a loop transformation where the mapping labels are of

Table 4.8: Partial redundancy elimination effects on mappings

	Resulting mapping label of S and R''
S is in same loop	$\langle * \rangle_{0..n} \longrightarrow \langle * \rangle_{0..m}$
S is in an inner loop	$\langle * \rangle_{0..i}, \mathbf{all}(\langle * \rangle_{\mathbf{i+1}})_{\mathbf{i+1}} .. \mathbf{all}(\langle * \rangle_{\mathbf{n}})_{\mathbf{n}} \longrightarrow one_{0..i}$
S is in an outer loop	$\langle * \rangle_{0..n} \longrightarrow one_{0..n}, \mathbf{last}_{\mathbf{n+1..m}}$, where $m > n$
S is in a different loop nest	$one_{0..i}, \mathbf{all}(\langle * \rangle_{\mathbf{i+1}})_{\mathbf{i+1}} .. \mathbf{all}(\langle * \rangle_{\mathbf{n}})_{\mathbf{n}} \longrightarrow one_{0..i}, \mathbf{last}_{\mathbf{i+1..m}}$

the form $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$. The mappings of statements within the loop bodies as well as the loop headers may be affected.

Table 4.9: Loop transformation effects on $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ mappings

Loop peeling (one time before loop)	
loop body	$\langle * \rangle_{0..i-1}, \mathbf{1}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ $\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{i}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	**delete**
loop test	$\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$ $\langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$
loop increment	$\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$

Loop unrolling (unroll factor = 2)	
loop body	$\langle * \rangle_{0..i-1}, \{\mathbf{1}, \mathbf{n}, \mathbf{2}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ $\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{2}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	$\langle * \rangle_{0..i-1} \rightarrow \langle * \rangle_{0..i-1}$
loop test	$\langle * \rangle_{0..i-1}, \{\mathbf{1}, \mathbf{n}, \mathbf{2}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$ $\langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$
loop increment	$\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{2}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$

Software pipelining (number of stages = 2)	
loop body:	
stage 1	$\langle * \rangle_{0..i-1}, \mathbf{1}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ $\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
stage 2	$\langle * \rangle_{0..i-1}, \{\mathbf{1}, \mathbf{n-1}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ $\langle * \rangle_{0..i-1}, \mathbf{n}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	**delete**
loop test	$\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$ $\langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$
loop increment	$\langle * \rangle_{0..i-1}, \{\mathbf{2}, \mathbf{n}, \mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$

Loop unswitching at nesting level i	
loop body:	
conditional	$\langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}$
other statements	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	$\langle * \rangle_{0..i-1} \rightarrow \langle * \rangle_{0..i-1}$
loop increment	$\langle * \rangle_{0..i} \rightarrow \langle * \rangle_{0..i}$
loop test	$\langle * \rangle_{0..i} \rightarrow \langle * \rangle_{0..i}$

Table 4.10: Loop transformation effects on $\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$ mappings cont.

Loop reversal at nesting level i	
loop body	$\langle * \rangle_{0..i-1}, \{\mathbf{n}, \mathbf{1}, -\mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	**delete**
loop test	$\langle * \rangle_{0..i-1}, \{\mathbf{n}, \mathbf{1}, -\mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$ $\langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$
loop increment	$\langle * \rangle_{0..i-1}, \{\mathbf{n}-\mathbf{2}, \mathbf{1}, -\mathbf{1}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$

Loop interchange at nesting levels i and $i+1$	
loop body	$\langle * \rangle_{0..i-1}, \langle * \rangle_{\mathbf{i+1}}, \langle * \rangle_{\mathbf{i}}, \langle * \rangle_{i+2}, \overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	$\langle * \rangle_{0..i-1} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$ $\langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i-1}$
loop test	$\langle * \rangle_{0..i} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \langle * \rangle_{i+1}$ $\langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \langle * \rangle_{i+1} \rightarrow \langle * \rangle_{0..i}$
loop increment	$\langle * \rangle_{0..i} \rightarrow \langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \langle * \rangle_{i+1}$ $\langle * \rangle_{0..i-1}, \mathbf{all}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}, \langle * \rangle_{i+1} \rightarrow \langle * \rangle_{0..i}$

Loop distribution and loop jamming	
loop body	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop increment	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop test	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$

Strip mining loop at nesting level i	
loop body	$\langle * \rangle_{0..i-1}, (\{\mathbf{j}=\mathbf{1}, \mathbf{n}, \mathbf{strip}\}_{\mathbf{0}}$ $\{\mathbf{tj}=\mathbf{j}, \mathbf{min}(\mathbf{n}, \mathbf{j}+\mathbf{strip}-\mathbf{1}), \mathbf{1}\}_{\mathbf{1}})(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$
loop initialization	$\langle * \rangle_{0..i-1} \rightarrow \langle * \rangle_{0..i-1}$
loop increment	$\langle * \rangle_{0..i-1}, \{\mathbf{strip}, \mathbf{n}, \mathbf{strip}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$
loop test	$\langle * \rangle_{0..i-1}, \{\mathbf{1}, \mathbf{n}, \mathbf{strip}\}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}$ $\langle * \rangle_{0..i-1}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}} \rightarrow \langle * \rangle_{0..i}, \mathbf{last}(\langle * \rangle_{\mathbf{i}})_{\mathbf{i}}$

Loop normalization	
loop body	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$
loop initialization	**delete**
loop increment	**delete**
loop test	$\overrightarrow{\langle * \rangle} \rightarrow \overrightarrow{\langle * \rangle}$

4.3.1 Effects of inlining

The mappings can be extended to support function inlining, which replaces calls to a function in the unoptimized code by bodies of the function in the optimized code. Since each inlined body may be optimized differently, each inlined call site has its own set of mappings. That is, for each inlined call site, separate mappings are maintained between the statements in the function in the unoptimized code and the inlined copy in the optimized code. When the function is examined or executed at runtime, the appropriate set of mappings are selected and utilized by using the knowledge of the call site encountered during program execution.

4.4 Summary

This chapter described a technique to automatically identify statement instance correspondences between unoptimized and optimized code and generate mappings reflecting these correspondences as code improving transformations are applied. The mappings reflect the effects of transformations and are established by analyzing how the position, number, and order of instances of a statement can change in a particular context when transformations are applied. No restrictions are placed on the order or number of transformations. The mappings support statement level optimizations, inlining, as well as loop transformations.

Now that the correspondence between the unoptimized and optimized versions of a program can be established through mappings, source level debugging techniques for optimized code can be developed. The remainder of this dissertation focuses on developing such techniques to help optimizer writers debug optimizers and application programmers to debug optimized code from the point of view of the source program. The mappings are utilized at compile time as well as runtime. Statically, the mappings along with the control flow graphs of the unoptimized and optimized programs are analyzed to determine how corresponding statements are relatively positioned with respect to each other and determine the corresponding statements that are path sensitive. Also, the mappings are used to generate *annotations* for the unoptimized and optimized programs, which can guide the actions of source level debugging tools for optimized code. At runtime, the mappings are utilized to determine how statements that execute in the optimized program relate to the unoptimized program version.

Chapter 5
Comparison Checking

A novel technique called comparison checking is presented in this chapter that utilizes the mappings presented in the previous chapter. This technique helps users validate and debug optimizers by verifying, for given inputs, that the semantics of a program are not changed by the application of optimizations. The comparison checking technique determines if the semantics of the optimized version differ from that of the unoptimized program by comparing the internal execution behavior using values that are computed by the unoptimized program with the corresponding values computed by the optimized program for given inputs.

The comparison checking scheme, as illustrated in Figure 5.1, automatically orchestrates the executions of both the unoptimized and optimized versions of a source program, for given inputs, and compares values computed by corresponding executed statements from both program versions. The mappings described in Chapter 4 specify the statements corresponding in the unoptimized and optimized programs. If the semantic behaviors are the same and correct with respect to the source program, the optimized program can be run with high confidence. On the other hand, if the semantic behaviors differ, the comparison checker displays the statements responsible for the differences and the optimizations applied to these statements. The optimizer writer can use this information to locate the incorrect code in the optimized program and determine what transformation(s) produced the incorrect code.[1]

The semantic behavior of an unoptimized or optimized program with respect to the source program is characterized by the outputs and values computed by source level statements in the unoptimized or optimized program for all possible inputs. Therefore, the semantic behaviors of the unoptimized and optimized programs with respect to the source program are compared by checking that (1) the same paths are executed in both programs, (2) corresponding source level assignments compute the same values and reference (i.e.,

[1]The checker can also be used to detect certain errors in a source program.

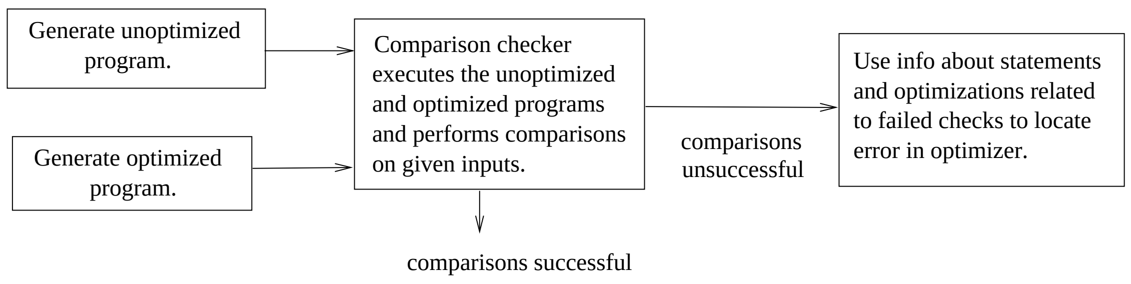

Generate unoptimized program.
Generate optimized program.
Comparison checker executes the unoptimized and optimized programs and performs comparisons on given inputs.
comparisons unsuccessful
Use info about statements and optimizations related to failed checks to locate error in optimizer.
comparisons successful

Figure 5.1: The comparison checking system

read, write) the corresponding locations, and (3) the outputs are the same. The outputs and the values computed by source level assignment statements and branch predicates for given inputs are compared. In addition, for assignments through arrays and pointers, checking is done to ensure the addresses to which the values are assigned correspond to each other. All assignments to source level variables are compared with the exception of those dead values that are not computed in the optimized code. This level of checking allows the comparison checking system to locate the *earliest* point where the unoptimized and optimized programs differ in their semantic behavior with respect to the source program. That is, the checker detects the earliest point during execution when corresponding source level statement instances should but do not compute the same values. Therefore, the checker can detect statements that are incorrectly optimized and subsequently compute incorrect values.

Consider the unoptimized C program fragment and its optimized version in Figure 5.2. Assume the unoptimized program is correct and the optimizer is turned on. The optimizer moves the assignment of x to the outside of the loop as a result of applying loop invariant code motion. When the optimized program executes, it returns incorrect output. The optimizer writer must debug the optimizer by first determining the cause of the error in the optimized code. Using the checker, a difference is detected in the internal behavior of the unoptimized and optimized programs at line 5 in the unoptimized program during the second iteration of the loop and at line 4 in the optimized program. The checker indicates that the value 4 is assigned in the unoptimized program and the value 3 is assigned in the optimized program. The checker also indicates that loop invariant code motion was applied to statement 5. The optimizer writer can examine the unoptimized and/or optimized versions of the program and then determine that loop invariant code motion was applied incorrectly. The optimizer writer can fix the error in the implementation of the loop invariant code motion optimization and rerun the checker on the unoptimized and optimized versions of the program.

```
Unoptimized Program                      Optimized Program
     Fragment                                 Fragment

1) int i, n, x, y, z;                    1) int i, n, x, y, z;
   ...                                      ...
2) y = 1;                                2) y = 1;
3) z = 2;                                3) z = 2;
4) for (i=1; i<=n; i=i+1){               4) x = y + z;
5)     x = y + z;                        5) for (i=1; i<=n; i=i+1){
6)     y = y + 1;                        6)     y = y + 1;
7)   }                                   7)   }
8) print x,y,z                           8) print x,y,z
```

Figure 5.2: Program example for comparison checking

The merits of a comparison checking system are as follows.

- When a comparison fails, the earliest place where the failure occurred and the optimizations that are involved are reported. Information about where an optimized program differs from the unoptimized version benefits the optimizer writer in debugging the optimizer.

- Since the internal values computed in the optimized code are compared to that of the unoptimized program, a finer level of testing is provided than just comparing the outputs. This level of checking can find errors in the optimized code that do not cause the output of the program to be incorrect.

- The optimizer writer has greater confidence in the correctness of the optimizer.

- A wide range of optimizations including classical optimizations, register allocation, loop transformations, and inlining can be handled by the technique.

- Optimizations can be performed at the source, intermediate, or target code level.

- The comparison checker is language independent. The technique is applicable to a variety of programming languages.

- The optimized code is not modified except for breakpoints, and thus no recompilation is required.

The comparison checking scheme is generally applicable to a wide range of optimizations from simple code reordering transformations to loop transformations. This chapter focuses mainly on statement level optimizations. The end of the chapter describes how to extend the comparison checking scheme to handle loop transformations and inlining.

The rest of this chapter is organized by presenting an overview of the comparison checker in Section 5.1. Section 5.2 describes the annotations used by the comparison checker to guide the checking of values and describes the algorithms to place the annotations. Section 5.3 presents experimental results.

5.1 Comparison checker overview

The comparison checker scheme compares values computed by both the unoptimized and optimized program executions to ensure the semantic behaviors of both programs are the same. To automate this scheme, the comparison checker must (1) determine which values computed by both programs need to be compared with each other, (2) determine where the comparisons are to be performed in the program executions, and (3) perform the comparisons. To achieve these tasks, three sources of information are utilized. First, *mappings* between corresponding instances of statements in the unoptimized and optimized programs, which are described in Chapter 4, are utilized to determine which values computed by both programs should be compared. These mappings are generated as optimizations are applied. Also, the statements that are affected by optimizations are marked with the optimization applied so that the checker can report the optimizations that are applied to statements. Second, after code is optimized and generated by the compiler, the mappings are used to automatically generate *annotations* for the unoptimized and optimized programs, which guide the comparison checker in comparing corresponding values and addresses. When a program point in either program version that has annotations is reached, the actions associated with the annotations at that point are executed by the comparison checker. Annotations identify program points where comparison checks should be performed. Third, since values to be compared are not always computed in the same order in the unoptimized and optimized code, a mechanism saves values that are computed early. These values are saved in a *value pool* and removed when no longer needed. Annotations are used to indicate if values should be saved in the value pool or discarded from the value pool.

A high level conceptual overview of the comparison checker algorithm is given in Figure 5.3. To avoid modifying the unoptimized and optimized programs, breakpoints are used to extract values from the unoptimized and optimized programs as well as activate

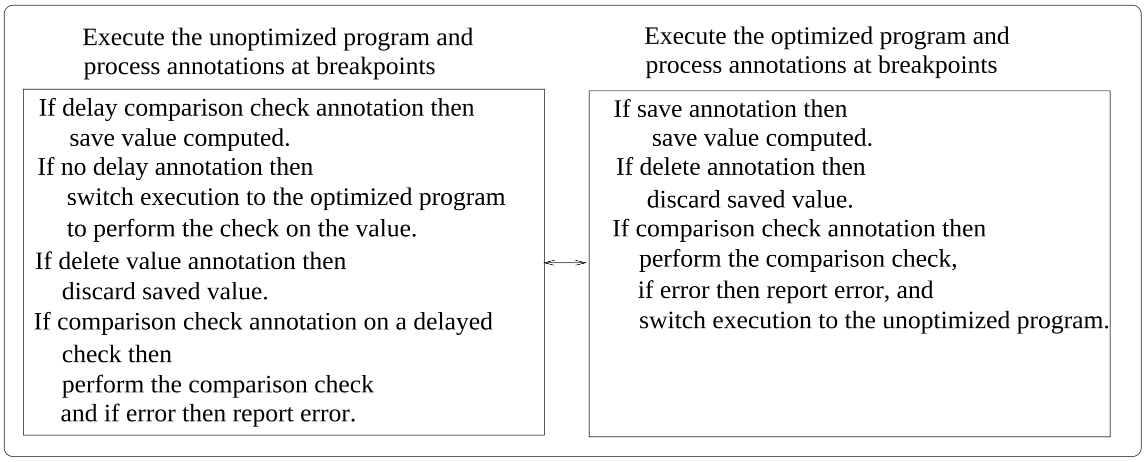

Execute the unoptimized program and process annotations at breakpoints
If delay comparison check annotation then save value computed.
If no delay annotation then switch execution to the optimized program to perform the check on the value.
If delete value annotation then discard saved value.
If comparison check annotation on a delayed check then perform the comparison check and if error then report error.
Execute the optimized program and process annotations at breakpoints
If save annotation then save value computed.
If delete annotation then discard saved value.
If comparison check annotation then perform the comparison check, if error then report error, and switch execution to the unoptimized program.

Figure 5.3: Comparison checker algorithm

annotations that are associated with program points in the unoptimized and optimized programs. The execution of the unoptimized program drives the checking and the execution of the optimized program. Therefore, execution begins in the unoptimized code and proceeds until a breakpoint is reached. Using the annotations in the unoptimized code, the checker can determine if the value computed can be checked at this point. A breakpoint at a program point in the unoptimized code that has a *comparison check* annotation indicates that a value computed by a statement should be checked. Also, by default, a breakpoint at a program point in the unoptimized code that has no associated annotation indicates that the value computed by the most recently executed statement should be checked. If a value should be checked, the optimized program executes until the corresponding value is computed (as indicated by a comparison check annotation), at which time the check is performed on the two values. During the execution of the optimized program, any values that are computed "early" (i.e., the corresponding value in the unoptimized code has not been computed yet) are saved in the value pool, as directed by the *save* annotations. If a *delay comparison check* annotation is encountered, indicating the checking of the value computed by the unoptimized program cannot be performed at the current point, the value is saved for future checking. The checker continues to alternate between executions of the unoptimized and optimized programs. Annotations also indicate when values that were saved for future checking can finally be checked and when the values can be removed from the value pool. Values computed by statement instances that are deleted by an optimization are not checked.

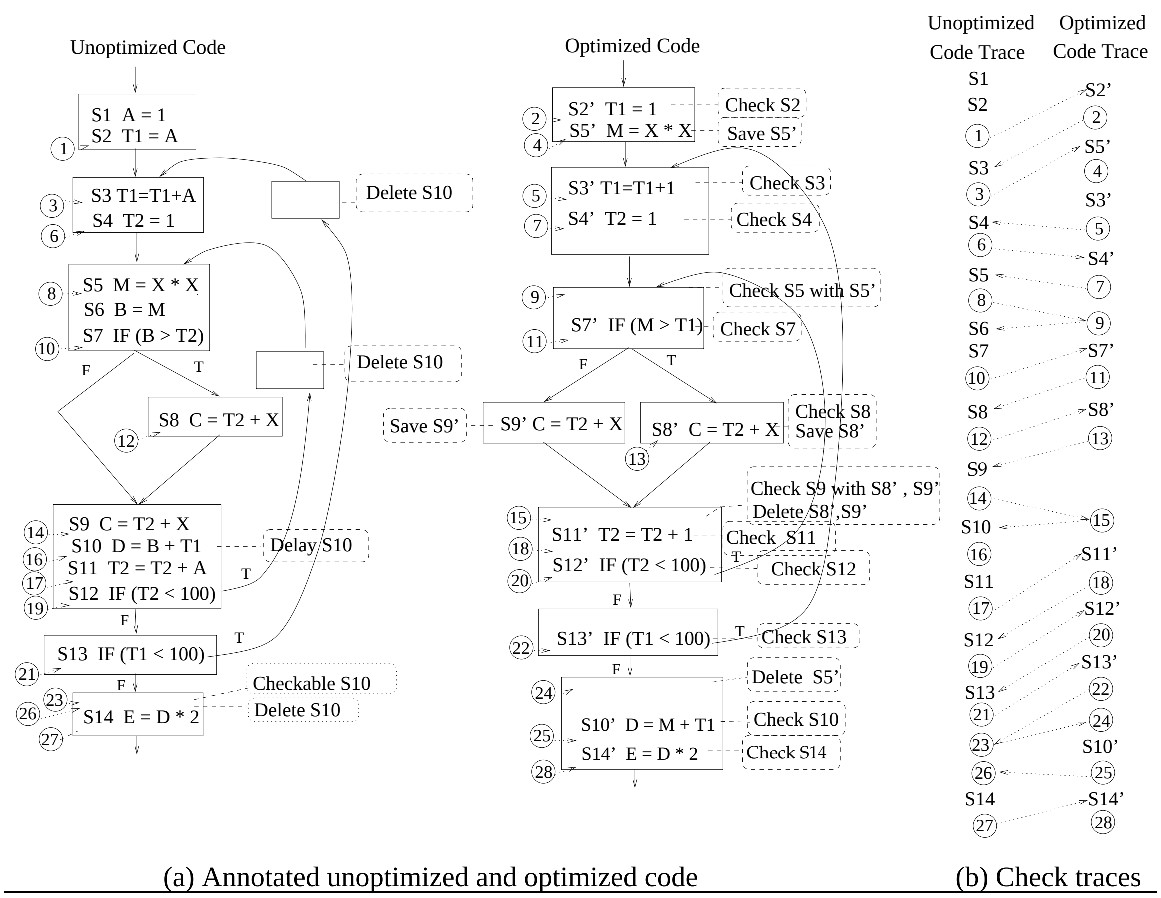


Figure 5.4: Comparison checking scheme example

5.1.1 Comparison checking scheme example

Consider the annotated unoptimized and optimized code segments in Figure 5.4(a), which illustrates the same unoptimized and optimized code example given in Figure 4.5 in Chapter 4. Assume all the statements shown are source level statements and loops execute for a single iteration. Breakpoints are indicated by circles. Breakpoints have been placed at program points in the unoptimized and optimized code that are associated with annotations as well as program points in the unoptimized code where comparison checks should be performed. The switching between the unoptimized and optimized program executions by the checker is illustrated by the traces in Figure 5.4(b). The traces include the statements executed as well as the breakpoints (circled) where annotations are processed. The arrows indicate the switching between programs.

The unoptimized program starts to execute with $S1$ and continues executing without checking, as $S1$ was deleted from the optimized program. After $S2$ executes, breakpoint 1 is reached and the checker determines from the annotation that the value computed can be checked at this point and so the optimized program executes until *Check* $S2$ is processed, which occurs at breakpoint 2. The values computed by $S2$ and $S2'$ are compared. The unoptimized program resumes execution and the loop iteration at $S3$ begins. After $S3$ executes, breakpoint 3 is reached and the optimized program executes until *Check* $S3$ is processed. Since a number of comparisons have to be performed using the value computed by $S5'$, when breakpoint 4 is reached, the annotation *Save* $S5'$ is processed and consequently, the value computed by $S5'$ is stored in the value pool. The optimized code continues executing until breakpoint 5, at which time the annotation *Check* $S3$ is processed. The values computed by $S3$ and $S3'$ are compared. $S4$ then executes and its value is checked. $S5$ then executes and breakpoint 8 is encountered. The optimized program executes until the value computed by $S5$ can be compared, indicated by the annotation *Check* $S5$ *with* $S5'$ at breakpoint 9. The value of $S5$ saved in the value pool is used for the check. The programs continue executing in a similar manner.

5.2 Annotations

Code annotations guide the comparison checking of values computed by corresponding statement instances from the unoptimized and optimized code. Annotations (1) identify program points where comparison checks should be performed, (2) indicate if values should be saved in a value pool so that they will be available when checks are performed, and (3) indicate when a value currently residing in the value pool can be discarded since all checks involving the value have been performed. The set of annotations is complete, and the placement of the annotations go hand in hand with the comparison checking algorithm.

5.2.1 Supporting statement level optimizations

Five different types of annotations are needed to implement the comparison checking strategy. In the example in Figure 5.5, which is the same example as in Figure 5.4, annotations are shown in dotted boxes. The annotations used by the checker and their actions follow. In the description, S_{uopt} indicates a statement in the unoptimized code and S_{opt} a statement in the optimized code.

5.2.1.1 The Check $\mathbf{S_{uopt}}$ annotation

The *Check* annotation is associated with a program point in the optimized code to indicate a check of a value computed by statement S_{uopt} is to be performed. The corresponding value to be compared is the result of the most recently executed statement in the optimized code. For example, in Figure 5.5, the annotation *Check* $S2$ is associated with

$S2'$. The *Check* annotation is used to indicate a check should be performed on values of a statement in the unoptimized code whose corresponding statement in the optimized code remains in its original positions.

A variation of this annotation is the **Check** S_{uopt} **with** $S_i, S_j, \ldots$ annotation, which is associated with a program point in the optimized code to indicate a check of a value computed by statement S_{uopt} is to be performed with a value computed by one of S_{uopt}'s corresponding statements $S_i, S_j, \ldots$ in the optimized program. The corresponding value to be compared is either the result of the most recently executed statement in the optimized code or is in the value pool. For example, in Figure 5.5, the annotation *Check* $S5$ *with* $S5'$ is associated with the original position of statement $S5$ in the optimized code. The *Check with* annotation is used to check statements in the unoptimized code whose corresponding statements in the optimized code have been moved.

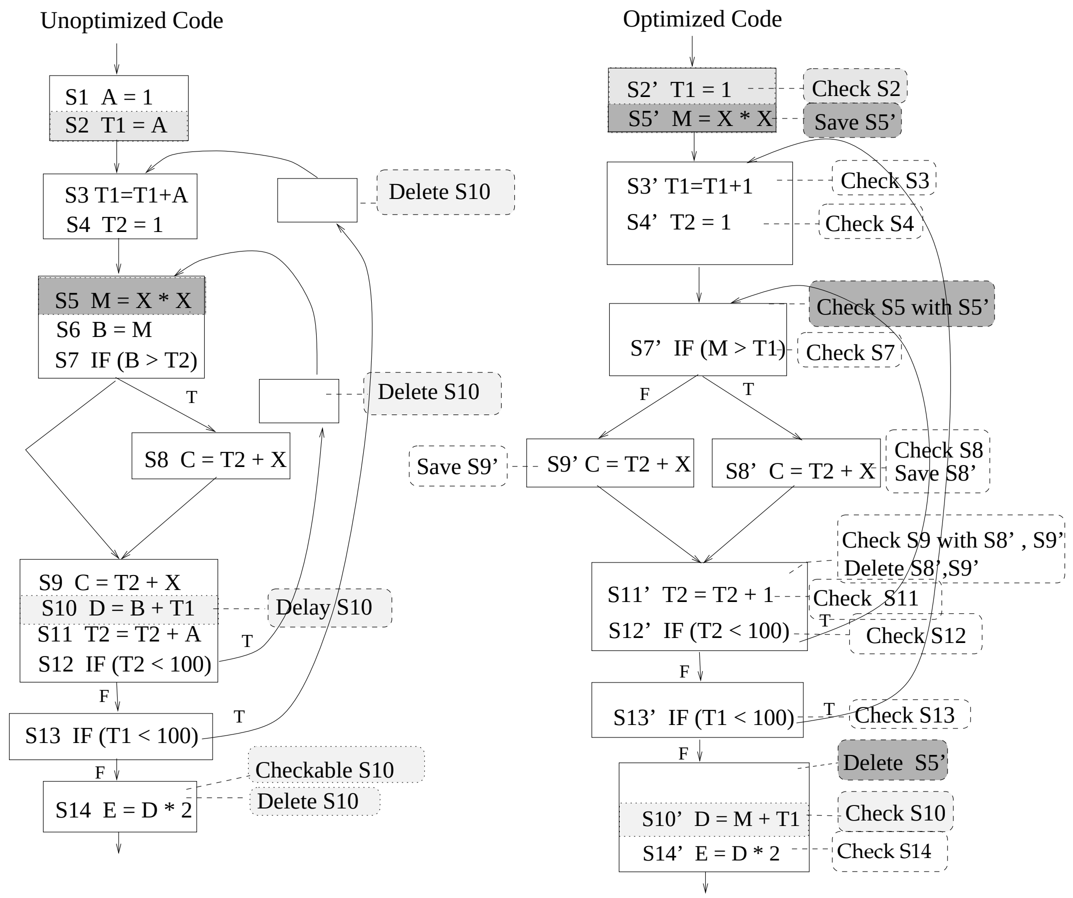


Figure 5.5: Annotated unoptimized and optimized code example

5.2.1.2 The Save $\mathbf{S_{opt}}$ annotation

If a value computed by a statement S_{opt} cannot be immediately compared with the corresponding value computed by the unoptimized code, then the value computed by S_{opt} must be saved in the value pool. In some situations, a value computed by S_{opt} must be compared with multiple values computed by the unoptimized code. Therefore, it must be saved until all those values have been computed and compared. The annotation *Save* S_{opt} is associated with S_{opt} and indicates that the value computed by S_{opt} is to be saved. In Figure 5.5, the statement $S5$ in the unoptimized code, which is moved out of the loops by invariant code motion, corresponds to statement $S5'$ in the optimized code. The value computed by $S5'$ cannot be immediately compared with the corresponding values computed by $S5$ in the unoptimized code because $S5'$ is executed prior to the execution of $S5$. Thus, the annotation *Save* $S5'$ is associated with $S5'$.

5.2.1.3 The Delay $\mathbf{S_{uopt}}$ and Checkable $\mathbf{S_{uopt}}$ annotations

If the value computed by the execution of a statement in the unoptimized code, S_{uopt}, cannot be immediately compared with the corresponding value computed by the optimized code because the correspondence between the values cannot be immediately established, then the value of S_{uopt} must be saved in the value pool. The annotation *Delay* S_{uopt} is associated with S_{uopt} to indicate the checking of the value computed by S_{uopt} should be delayed, saving the value in the value pool. The point in the unoptimized code at which checking can finally be performed is marked using the annotation **Checkable** S_{uopt}.

In some situations, a delay check is needed because the correspondence between statement instances cannot be established unless the execution of the unoptimized code is further advanced. In Figure 5.5, statement $S10$ inside the nested loop in the unoptimized code is moved after the loops in the optimized code by partial dead code elimination. In this situation, only the value computed by statement $S10$ during the last iteration of the nested loops is to be compared with the value computed by $S10'$. However, an execution of $S10$ corresponding to the last iteration of the nested loops can only be determined when the execution of the unoptimized code exits the loops. Therefore, the checking of $S10$'s value is delayed.

There is another situation in which a check is delayed for efficiency reasons. Consider the example in Figure 5.6(a) in which the computation of x's value is moved from before the loop to after the loop. In this case, after x has been computed by the unoptimized code, the execution of the optimized code is advanced to the point after the loop and the value of x is checked. However, all values of y that are computed inside the loop would have to be saved, resulting in potentially a large value pool. To avoid the creation of a large pool, the checking of the value of x can be delayed until after the loop, as shown in Figure 5.6(b).

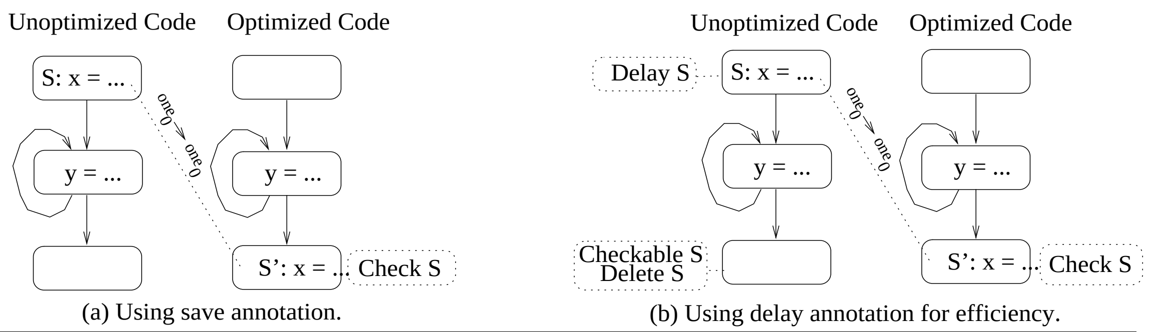


Figure 5.6: Types of annotations

5.2.1.4 The Delete S annotation

The *Delete* annotation is associated with a program point in the unoptimized/optimized code to indicate a value computed previously by S and stored in the value pool can be discarded. Since a value may be involved in multiple checks, a delete annotation must be introduced at a point where all relevant checks would have been performed. In Figure 5.5, the annotation *Delete* $S5'$ appears after the loops in the optimized code because at that point, all values computed by statement $S5$ in the unoptimized code would have been compared with the corresponding value computed by $S5'$ in the optimized code.

5.2.1.5 The Check-self S annotation

The *Check-self* annotation is associated with a program point in the unoptimized/optimized code and indicates that values computed by S must be compared against each other to ensure the values are the same. The annotation is used when a mapping refers to *all* instances of statement S in the optimized/unoptimized program. As the loop enclosing S begins to execute, the first value computed by S is saved in the value pool. Subsequent values computed by S are compared with the saved value. Since this annotation causes a value of S to be saved in the value pool, a *Delete* S annotation is used to later discard the value from the value pool.

In Figure 5.7, the mapping of statement S' refers to *all* instances of S'. The annotation *Check-self* S' is associated with S' and the annotation *Delete* S' is introduced after the enclosed loop. The checking of S and S' is as follows. After S executes, the checking of S is delayed and its computed value is saved in the value pool. The first time

S' executes, its computed value is saved in the value pool. Subsequent values computed by S' are compared with the saved value. Once the unoptimized program execution reaches the program point at *Checkable* S, the value computed by S can now be compared and the optimized program executes until *Check* S *with* S' is encountered. The values computed by S and S' are compared at this point, and then the values of S and S' are discarded from the value pool.

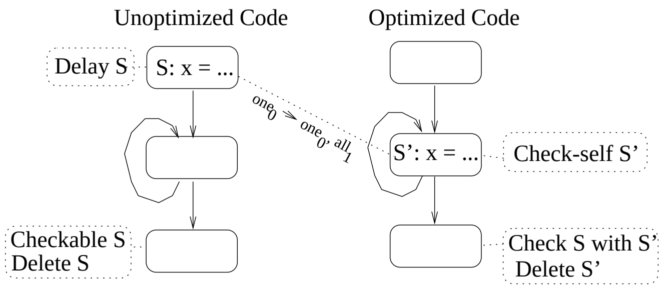


Figure 5.7: Types of annotations

Depending on the position of a statement in the unoptimized code that has a mapping referring to *all* instances, a *Check-self* annotation is not necessary, as illustrated in Figure 5.5. Although the mapping of statement $S5$ refers to *all* instances of $S5$, as illustrated in Figure 4.5 in Chapter 4, a *Check* $S5$ *with* $S5'$ annotation can be used to perform all of the comparison checks on $S5$.

5.2.2 Algorithms to place annotations for statement level optimizations

The selection and placement of annotations are independent of particular optimizations and depend only on which and how statement instances correspond and the relative positions of corresponding statements in both the unoptimized and optimized programs. The mappings and data flow analysis, including reachability and postdominance, are used to determine where and what annotations to use. Annotations are placed after all optimizations are performed and target code has been generated, and therefore, the code to emit the annotations can be integrated as a separate phase within a compiler.

Four algorithms annotate the unoptimized and optimized programs. Three algorithms, which introduce *Check*, *Delay*, *Checkable*, *Check-self*, and *Save* annotations, can be consolidated and applied at the same time. The fourth algorithm, which introduces *Delete* annotations, is applied after the other three algorithms are applied. For ease of explanation, the algorithms use separate control flow graphs, G_{unopt} and G_{opt} for the unoptimized and optimized programs, respectively. However, these algorithms can easily be modified to handle the representation of both programs within a single control flow graph. Also, some of the algorithms use the following terminology.

Suppose a mapping exists between statement S in the unoptimized program and statement S' in the optimized program, which is denoted by $S - S'$. Statement S' is either *rolled forward*, *rolled back*, or neither rolled forward nor rolled back with respect to S. To determine if S' is rolled backward with respect to S (i.e., S' is relatively positioned before S), the original position of S and the actual position of S' are compared in G_{opt}. Let $ORHead(S)$ denote the corresponding original position of S in G_{opt}. S' is rolled backward with respect to S if a path P in G_{opt} exists from S' to $ORHead(S)$ such that P does not include backedges of loops enclosing both S' and $ORHead(S)$. The backedge restriction on P ensures that only the positions of the same instance of S' before and after optimization are considered. An enclosing loop would cause the incorrect examination of instances of S' from two successive iterations of the loop. This restricted notion of a path is captured by the $SimplePath$ predicate.

Definition 5.1 The predicate $SimplePath(x,y)$ is true if $\exists$ path P from program point x to program point y in G_{opt} and P does not include backedges of loops enclosing both x and y.

Using the $SimplePath$ predicate, statement S' with respect to S is rolled back if $SimplePath(S',ORHead(S))$ is true. Statement S' with respect to S is rolled forward if $SimplePath(ORHead(S),S')$ is true. In Figure 5.5, statement $S5'$ is rolled back with respect to $S5$, statement $S10'$ is rolled forward with respect to $S10$, and statement $S2'$ is neither rolled forward nor rolled back with respect to $S2$.

5.2.2.1 Algorithm to introduce *Check*, *Delay*, and *Checkable* annotations

The *Check*, *Delay*, and *Checkable* annotations direct the comparison checker as to when comparisons between values computed in both the unoptimized and optimized programs are to be performed. Since the execution of the unoptimized program drives the checking and the execution of the optimized program, these annotations are introduced for statements in the unoptimized program. The algorithm, shown in Figure 5.8, takes as input a statement S from the unoptimized program, the mappings of S, and the unoptimized and optimized control flow graphs G_{unopt} and G_{opt}. The algorithm is applied to every statement in the unoptimized program that has a mapping.

All of the mappings of statement S are examined because the types of annotations introduced depend on the relative positions of the corresponding statements of S as well as the instances of S to be checked. When the corresponding statement S' in the optimized code is rolled forward, its check is delayed for efficiency reasons. Also, when a mapping of $S - S'$ refers to a *last* instance for either S or S', its check is delayed. In this case, the delay is necessary because a *last* loop iteration can only be determined after the unoptimized/optimized code further executes and exits the appropriate loop. If a check is delayed, then a *Delay* annotation is introduced for S. *Check* and *Checkable* annotations are also introduced so that comparisons will be later performed on S. Otherwise, if the checking of S is not delayed, then only *Check* annotations are introduced for S.

Check annotations are carefully placed to ensure that the number of *Check* annotations encountered during runtime execution equals the number of comparisons that need to be performed. For each mapping of S such that the corresponding statement in the optimized program is rolled forward or a *last* instance is referred to in the mapping, the first two conditions in the algorithm must be satisfied. This ensures that instances of S to be checked will not have extraneous nor too little *Check* annotations processed on its behalf during the optimized program execution. For the remaining mappings of statement S, a *Check* annotation is associated with the program point in the optimized code that represents the original position of statement S.

5.2.2.2 Algorithm to introduce *Check-self* annotations

The *Check-self* annotation directs the comparison checker to check instances of a statement computed in a loop against each other. The algorithm to introduce *Check-self* annotations is shown in Figure 5.9. It is applied to all statements in the unoptimized and optimized programs that have mappings and takes as input a statement S from the unoptimized or optimized program, the mappings of S, and the unoptimized and optimized control flow graphs, G_{unopt} and G_{opt}.

Statements in the optimized programs whose instances are referred to as *all* in one or more mappings have *Check-self* annotations associated with them. Statements in the unoptimized code whose instances are referred to as *all* in one or more mappings and have at least one of the corresponding statements rolled forward in the optimized code have *Check-self* annotations associated with them. Also, statements in the unoptimized code whose instances are referred to as *all* and *last* in one or more mappings have *Check-self* annotations associated with them.

5.2.2.3 Algorithm to introduce *Save* annotations

Save annotations direct the comparison checker to save values that will be used in future comparison checks. Statements in the optimized program whose instances are referred to as *last* in one or more mappings or are rolled back with respect to corresponding statements in the unoptimized program have *Save* annotations associated with them. The algorithm to introduce these annotations is shown in Figure 5.10. It takes as input a statement S' from the optimized program, S''s mappings, and the unoptimized and optimized control flow graphs.

5.2.2.4 Algorithm to introduce *Delete* annotations

Delete annotations direct the comparison checker to discard values from the value pool. Any statements in the unoptimized and optimized programs whose computed values are saved in the value pool as a result of *Delay*, *Save*, and *Check-self* annotations use *Delete* annotations. The algorithm, shown in Figure 5.11, is applied after all of the previous algorithms are applied and takes as input the mappings and the annotated unoptimized and optimized control flow graphs.

For statements in the unoptimized program, *Delete* annotations are introduced in the unoptimized program, and similarly, for statements in the optimized program, *Delete* annotations are introduced in the optimized program.

Delete annotations are carefully placed to ensure that appropriate values are safely discarded. *Delete* annotations are introduced where no more comparisons involving the value (to be deleted) will be performed.

```
For each mapping $S - S'$ such that $S'$ is rolled forward with respect to $S$ or
the mapping $S - S'$ refers to a $last$ instance
    Associate $Delay\ S$ annotation on $S$ (if not already delayed)
    Associate $Check\ S\ with\ S'$ annotations on points $P$ in $G_{opt}$ such that
        (i) points $P$ postdominate $S'$ and the original position of $S$
        (ii) points $P$ are in the innermost loop $L_i$ enclosing $S'$ such that
        the element representing $L_i$ in the vector of $S'$ is $one$
    Associate $Checkable\ S$ on the same points $P$ in $G_{unopt}$
End For
With the remaining mappings $S - S'_1, S - S'_2, \ldots, S - S'_n$
    Associate $Check\ S\ with\ S'_1, S'_2, \ldots, S'_n$ annotation at the original position of $S$ in $G_{opt}$
    If $S$ has a $Delay$ annotation then
      Associate $Checkable\ S$ at the position of $S$ in $G_{unopt}$
    End if
```

Figure 5.8: Algorithm to introduce *Check, Delay, Checkable* annotations

```
If $\exists$ mapping that refers to $all$ instances of $S$ then
    If $S$ is a statement in $G_{opt}$ then
      Associate $Check\text{-}self\ S$ annotation with statement $S$
    Else /* $S$ is in the unoptimized program */
      If the corresponding statement of $S$ is rolled forward
        Associate $Check\text{-}self\ S$ annotation with statement $S$
      Else if $\exists$ mapping that refers to $last$ instances of $S$ then
        Associate $Check\text{-}self\ S$ annotation with statement $S$
    End if
End if
```

Figure 5.9: Algorithm to introduce *Check-self* annotations

```
If $\exists$ mapping that refers to a $last$ instance of $S'$ or
  ($\exists$ mapping $S - S'$ and $S'$ is rolled back with respect to $S$) then
    Associate $Save\ S'$ annotation with statement $S'$
End if
```

Figure 5.10: Algorithm to introduce *Save* annotations

```
For each statement $S$ in the unoptimized program
    For each mapping that refers to a $last$ instance of $S$
        Associate $Delete\ S$ on the back edges of each loop $L_i$ in $G_{unopt}$ enclosing $S$
        such that the element representing $L_i$ in the vector of $S$ is $last$
    End for
    If a $Delay\ S$ was introduced for $S$ then
        Associate $Delete\ S$ on points $P$ in $G_{unopt}$ such that $P$ postdominates
        all $Checkable\ S$ annotations
    End if
End For
For each statement $S'$ in the optimized program
    If a $Save\ S'$ annotation was introduced for $S'$ then
        For each mapping that refers to a $last$ instance of $S'$
            Associate $Delete\ S'$ on the back edges of each loop $L_i$ in $G_{opt}$ enclosing $S'$
            such that the element representing $L_i$ in the vector of $S'$ is $last$
        End for
    End if
    For each $Save\ S'$ or $Check\text{-}self\ S'$ introduced for $S'$
        Associate $Delete\ S'$ on points $P$ in $G_{opt}$ such that
            (i) $P$ postdominates $S'$
            (ii) $\forall$ mappings $S - S'$,
                $P$ postdominates all $Check\ S$ annotations introduced for mapping $S - S'$
            (iii)If $\exists$ mapping that refers to $last$ or $all$ instances of $S'$ then
                    The $Delete\ S'$ annotation is in the innermost loop $L_i$ enclosing $S'$
                    such that $\forall$ the mappings of $S'$, the elements representing $L_i$ in the
                    vectors of $S'$ are $one$
                Otherwise the $Delete\ S'$ annotation is in the same loop nest as the
                    original position of statement $S$
                End if
    End for
End for
```

Figure 5.11: Algorithm to introduce *Delete* annotations

5.2.3 Supporting loop transformations and inlining

The previous sections support statement level optimizations. To support loop transformations, the annotations are extended with suffixes to handle the instances of a statement in the unoptimized code that are (1) reordered in the optimized code and (2) divided among several statements in the optimized code. The suffixes describe specific instances and sequences of instances as described by the mappings of such statements. The placement of annotations is extended as follows. If the instances of a statement S in the unoptimized code are reordered in the optimized code, the checking of such instances are delayed. A mapping suffix is added to the *Delay* annotation to indicate the instances whose checks are to be delayed and the order in which the values computed by these instances of S are saved in the value pool by the comparison checker. This ordering reflects the ordering of the corresponding instances of S in the optimized code, which is the order in which the comparison checks are performed. For each *Checkable* annotation, a suffix indicates the instances of a statement referred by the annotations. For example, in Figure 5.12, the loop in the optimized code has been reversed and thus, the instances of statements in the unoptimized code are reordered in the optimized code. *Delay* annotations are placed for the instances of statements $S2, S3$, and $S4$. *Checkable* annotations are placed after the loop to indicate all of the instances of statements $S2, S3$, and $S4$ (whose checks were delayed) are now ready to be checked.

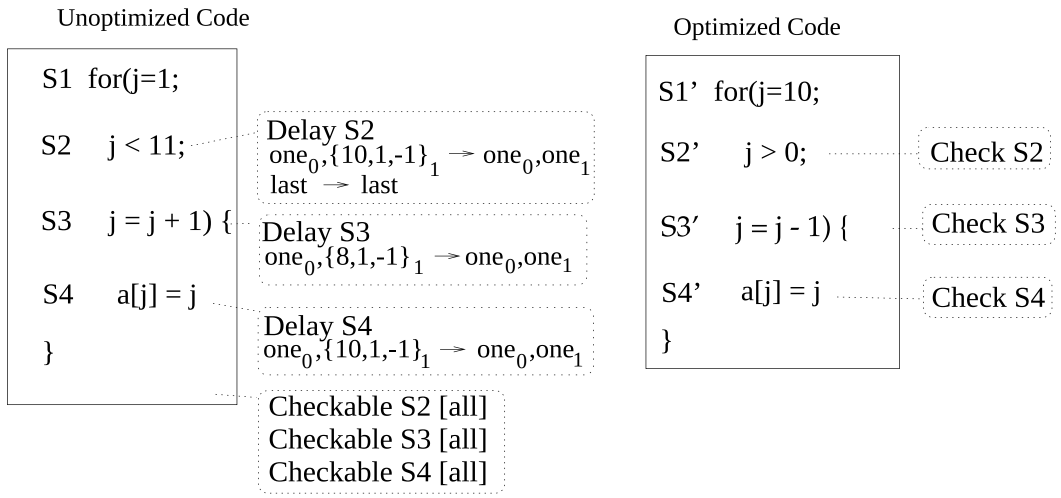


Figure 5.12: Annotated loop reversal example

If the instances of a statement S in the unoptimized code are divided among several statements in the optimized code, a separate check or delayed check is utilized for each set of divided instances of S. A mapping suffix is added to the *check* annotation to indicate the instances of the statement in the unoptimized code that will be checked. For example, in Figure 5.13, the loop in the optimized code has been unrolled, and thus the instances of statements in the loop in the unoptimized code are divided among two statements in the optimized code. For statement $S4$, two *check* annotations are placed in the optimized code. The first annotation refers to the odd instances of $S4$ and the second annotation refers to the even instances of $S4$. For statement $S2$ (the loop test), the *check* annotation indicates the checking of the odd instances and the last instance of $S2$. For statement $S3$ (the loop increment), the *check* annotation indicates the checking of the even instances of $S3$.

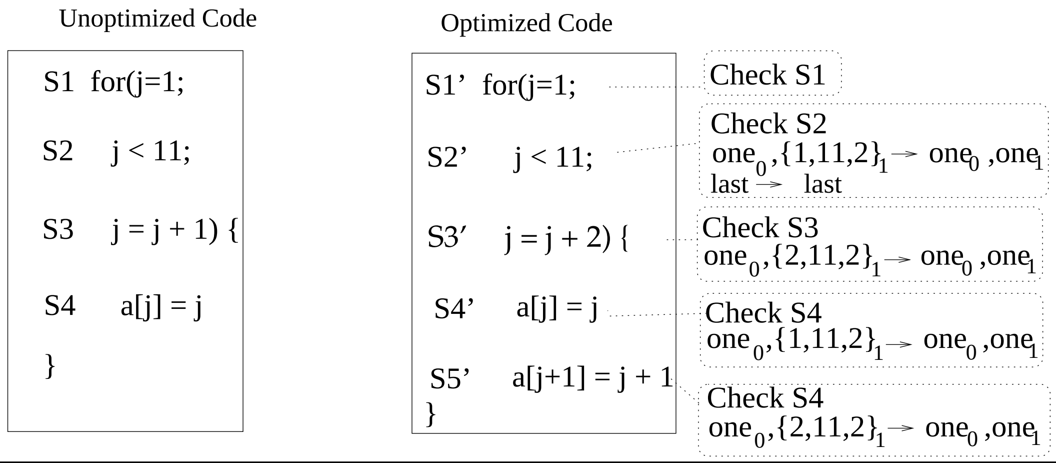


Figure 5.13: Annotated loop unrolling example

Function inlining, which replaces calls to a function in the unoptimized code by bodies of the function in the optimized code, can also be supported. For each call site, a separate mapping is maintained between the statements in the function in the unoptimized code and the inlined copy in the optimized code. By analyzing the mappings corresponding to each call site, a set of annotations is computed. At runtime, when the function is executed, the comparison checker selects and follows the appropriate set of annotations by using the knowledge of the call site encountered during program execution.

5.3 Implementation and experiments

The **C**omparison checker for **OP**timized code, COP, was implemented, including the instruction mapping, annotation placement, and checking. `Lcc` [23] was used as the compiler for an application program and was extended to include a set of optimizations, namely loop invariant code motion, dead code elimination, partial redundancy elimination, register allocation, copy propagation, and constant propagation and folding. On average, the optimized code generated by the optimized `lcc` executes 16% faster in execution time than the unoptimized code.

As a program is optimized, mappings are generated. Besides generating target code, `lcc` was extended to determine the mappings between the unoptimized and optimized code, breakpoint information, and annotations that are derived from the mappings. The code to emit breakpoint information and annotations is integrated within `lcc` through library routines. Thus, compilation and optimization of the application program produce the target code for both the unoptimized program and optimized program as well as auxiliary files containing breakpoint information and annotations for both the unoptimized and optimized programs. These auxiliary files are used by the checker. Breakpoints are generated whenever the value of a source level assignment or a predicate is computed and whenever array and pointer addresses are computed. Breakpoints are also generated to save base addresses for dynamically allocated storage of structures (e.g., `malloc()`, `free()`, etc.). Array addresses and pointer addresses are compared by actually comparing their offsets from the closest base addresses collected by the checker. Floating point numbers are compared by allowing for inexact equality. That is, two floating point numbers are allowed to differ by a certain small delta [44]. Breakpointing is implemented using fast breakpoints [32].

Experiments were performed to assess the practicality of COP. The main concerns were usefulness as well as cost of the comparison checking scheme. COP was found to be very useful in actually debugging the optimizer implemented for this work. Errors were easily detected and located in the implementation of the optimizations as well as in the mappings and annotations. When an unsuccessful comparison between two values was detected, COP indicated which source level statement computed the value, the optimizations applied to the statement, and which statements in the unoptimized and optimized assembly code computed the values.

In terms of cost, the slow downs of the unoptimized and optimized programs and the speed of the comparison checker are of interest. COP performs *on-the-fly* checking during the execution of both programs. Both value and address comparisons are performed. In the experiments, COP ran on an HP 712/100 and the unoptimized and optimized programs on separate SPARC 5 workstations instead of running all three on the same processor as described in Section 3. Messages are passed through sockets on a 10 Mb network. A buffer is used to reduce the number of messages sent between the executing programs and the checker. Some of the integer Spec95 benchmarks as well as some smaller test programs were used as test cases.

Table 5.1 shows the CPU execution times of the unoptimized and optimized programs with and without annotations. On average, the annotations slowed down the

Table 5.1: Execution times (minutes:seconds)

Program	Source length (lines)	Unoptimized Code (CPU)	Unoptimized Code annotated (CPU)	Optimized Code (CPU)	Optimized Code annotated (CPU)	COP (CPU)	COP (response time)
`wc`	338	00:00.26	00:02.16	00:00.18	00:01.86	00:30.29	00:53.33
`yacc`	59	00:01.10	00:06.38	00:00.98	00:05.84	01:06.95	01:34.33
`go`	28547	00:01.43	00:08.36	00:01.38	00:08.53	01:41.34	02:18.82
`m88ksim`[1]	17939	00:29.62	03:08.15	00:24.92	03:07.39	41:15.92	48:59.29
`compress`[1]	1438	00:00.20	00:02.91	00:00.17	00:02.89	00:52.09	01:22.82
`li`[1]	6916	01:00.25	05:42.39	00:55.15	05:32.32	99:51.17	123:37.67
`ijpeg`[1]	27848	00:22.53	02:35.22	00:20.72	02:33.98	38:32.45	57:30.74

[1] Spec95 benchmark test input set was used.

execution of the unoptimized programs by a factor of 8 and that of the optimized programs by a factor of 9. The optimized program experiences greater overhead than the unoptimized program because more annotations are added to the optimized program.

Table 5.1 also shows the CPU and response times of COP. The performance of COP depends greatly upon the lengths of the execution runs of the programs. Comparison checking took from a few minutes to a few hours in terms of CPU and response times. These times are clearly acceptable if comparison checking is performed off-line (i.e., non-interactively). The performance of the checker was found to be bounded by the processing platform and speed of the network. A faster processor and 100 Mb network would considerably lower these times. In fact, when COP executes on a 333 MHz Pentium Pro processor, the performance is on average 6 times faster in terms of CPU time. Access to a faster network was not possible. The pool size was measured during experiments and was fairly small. If addresses are not compared, the pool size contains less than 40 values for each of the programs. If addresses are compared, then the pool size contains less than 1900 values.

5.4 Summary

A novel approach to debug optimizers is presented. In the technique presented, both the unoptimized and optimized versions of an application program are executed, and computed values are compared to ensure the behaviors of the two versions are the same under the given input. If the values are different, the comparison checker displays where in the application program the differences occurred and what optimizations were involved. The optimizer writer can utilize this information to debug the optimizer. The automation of the comparison checking scheme relies on the mappings described in chapter 4 and annotations described in this chapter. The comparison checking scheme was implemented and executes the unoptimized and optimized versions of C programs. Experimental results demonstrate the approach is effective and practical.

In this chapter, the comparison checking technique compared the execution behavior of an unoptimized version of a program with the optimized version of the program. However, the comparison checking technique can also be utilized to check different levels of optimizations. That is, the checking can be performed in phases just as optimizations are often phased. For example, checking can be performed after loop optimizations are applied, after statement level optimizations are applied, and after low level optimizations are applied. This phase checking can reduce the cost of checking as well as help optimizer writers debug the optimizations that were applied in the phase that is to be checked. Furthermore, the comparison checking technique can be tailored to help optimizers writers debug and validate specific optimizations. In the next chapter, the comparison checking technique is tailored to global register allocation.

Chapter 6
Register Allocation Checking

The comparison checking technique described in the previous chapter compared the execution behaviors of an unoptimized version of a program with the optimized version of the program. When the semantic behaviors differ, the comparison checker displays the statements responsible for the differences and the optimizations applied to these statements. This technique can locate the earliest point during execution when corresponding source level statement instances should but do not compute the same values and can indicate the optimizations that were applied to the statement. However, this information is not always helpful in that many optimizations may have been applied to the statement and detailed information about the optimizations may not be available. To provide more effective information, the comparison checking technique can be tailored to help optimizer writers debug and validate specific optimizations. In particular, the comparison checking technique can be tailored to help debug and validate an implementation of register allocation, which is a code transformation that can be very tedious and difficult to debug, especially when errors are intermittent. In this chapter, a register allocation checker is developed that extends the comparison checking technique in that the checker can detect errors in a register allocator implementation and determine the possible cause(s) of the errors. This register allocation checker can be incorporated into the comparison checker or can be used as a standalone tool.

The register allocation checker is similar to the comparison checker in that the semantic behaviors of the unoptimized and optimized programs are compared. However, the register allocation checker saves different kinds of information and utilizes a different set of annotations to track information about the variables that are assigned to registers and verify that the expected values of these variables are used throughout the optimized program execution. This level of checking and tracking allows the register allocation checker to locate the earliest execution point where the unoptimized and optimized programs differ in their semantic behaviors and display to the user the actual cause(s) of the differences.

When a register allocation technique is implemented incorrectly, the incorrect behavior can include

- using a wrong register,
- evicting a value from a register but not saving it for future uses,
- failing to load a value from memory, and
- using a *stale* value. A stale value of a variable is used in the optimized program when a value of a variable is computed in a register, but instead of using the value of the variable in the register, the optimized program uses the old value in memory.

The register allocation checker can determine when a register allocator exhibits these types of behavior. Consider the unoptimized C program fragment and its optimized version in Figure 6.1. Assume the unoptimized program is correct and the register allocation is turned on. The register allocator assigns variables x, y, z, and a to registers $r3$, $r1$, $r4$, and $r5$, respectively, in the optimized program, and copies the value of y in register $r1$ to register $r2$. Assume the optimized program returns incorrect output. Using the checker, a difference is detected in the internal behavior of the unoptimized and optimized programs at line 3 in the unoptimized code and line 7 in the optimized code. The checker indicates that the values used by y in the unoptimized and optimized programs differ and indicates that $r1$ is used inconsistently as y was evicted from $r1$ earlier during the execution of the optimized program. The checker also indicates that the expected value of y is in register $r2$ and thus, $r2$ should have been used instead of $r1$. The optimizer writer can then use this information to debug the implementation of register allocation.

```
Unoptimized Code          Optimized Code

1) x = 2 * y              1) load r1,y
     ...                  2) load r2,2
2) z = x + 5
     ...                  3) mul r3,r1,r2
3) a = y + y                   ...
                          4) move r1,r2

                          5) load r1,5

                          6) add r4,r3,r1
                               ...

                          7) add r5,r1,r1
                          8) store r5,a
```

Figure 6.1: Program example for register allocation checking

6.1 Register allocation checker overview

The register allocation checking scheme is similar to the comparison checking scheme in that values computed in both the unoptimized and optimized programs are checked, but the register allocation checking scheme also compares values of variables that are used in both programs and tracks and verifies information about the variables that are assigned to registers throughout the optimized program execution. To automate this scheme, the register allocation checker must

(1) determine which values are computed by both programs and need to be compared with each other,

(2) determine where the comparisons are to be performed in the program executions,

(3) perform the comparisons,

(4) track information about variables assigned to registers in the optimized program execution, and

(5) verify that the expected values of these variables are used throughout the optimized program execution.

This tracking information includes maintaining at each program point of the execution of the optimized program the

(1) current locations of values of variables,

(2) variables whose memory locations hold stale values,

(3) variables whose values in registers have been evicted, and

(4) variables that are currently assigned to registers.

To achieve these tasks, mappings and annotations are utilized. The mappings of the comparison checker are extended to include the correspondences between the *uses of variables* in the unoptimized and optimized intermediate programs. These mappings are generated before register allocation is applied because the correspondence between the two program versions is not changed by the application of register allocation. The mappings capture only the effects of register allocation and not other optimizations because the checking is performed on a program before registers are allocated and on a program after registers are allocated. However, the unoptimized program can include the application of other optimizations, which are assumed to be correct. After register allocation is applied and code is generated by the compiler, the mappings are used to automatically generate *annotations* for the optimized program, which guide the register allocation checker in comparing corresponding values and addresses and tracking and verifying information about the variables that are assigned to registers throughout the program execution. When a targeted program point in the optimized code is reached, the actions associated with the annotations at that point are executed by the register allocation checker.

A high level conceptual overview of the register allocation checker algorithm is given in Figure 6.2. This algorithm is similar to that of the comparison checker. Breakpoints are used to extract values from the unoptimized and optimized programs as well as activate annotations. Annotations guide the actions of the register allocation checker. However, since values that are assigned to variables in the optimized code are the values that are tracked and verified, the execution of the optimized program drives the checking and the execution of the unoptimized program. Therefore, execution begins in the optimized code and proceeds until a breakpoint is reached. Depending on the annotation, the checker may track variables assigned to registers, evicted variables, and stale variables, and/or determine if a value computed or used should be checked at the current point of execution of the optimized code. When a value should be checked at the current point, the unoptimized program executes until the corresponding point of execution is reached, at which time the check is performed on the two values. The checker continues to alternate between executions of the unoptimized and optimized programs. If the values that are compared differ, then the checker informs the user of the possible causes of the difference. Also, as values are tracked, the checker informs the user of any inconsistencies (e.g., a stale value is loaded, unexpected value is stored to a memory location, etc.). Once an inconsistency of a value of a variable is detected, the inconsistency is propagated through the uses of the value.

```
Do
   Execute the optimized program and process annotations at breakpoint
   If Check annotation then
          Execute the unoptimized program until the equivalent execution
          point (of the optimized program) is reached
          Perform the comparison check
          If error then report the error and the cause of the error
   If Register Assign annotation or Load annotation or Store annotation
       or Register Move annotation then
          Update register/variable information
          Verify the loaded or stored value is the expected value
          Inform user of any inconsistencies
   End if
While the optimized program has not finished executing
```

Figure 6.2: Register allocation checker algorithm

6.2 Annotations

Similar to the comparison checking technique, code annotations guide the checking of values in the unoptimized and optimized code. Code annotations are also used to verify and track values of variables. Annotations (1) identify program points where comparison checks should be performed and (2) indicate what values of variables/registers should be tracked and verified in the optimized code. Five types of annotations are needed to implement the register allocation checking strategy. In the example in Figure 6.3, which illustrates the same unoptimized and optimized code example given in Figure 6.1, annotations are shown in dotted boxes.

6.2.1 The Check $\mathbf{v}, \mathbf{r}$ annotation

The *check* v, r annotation is associated with a program point p in the optimized code to indicate a check of a value of variable v in register r is to be performed. The register allocation checker will execute the unoptimized program until the equivalent program point p is reached. The corresponding value to be compared is the current value of v in the

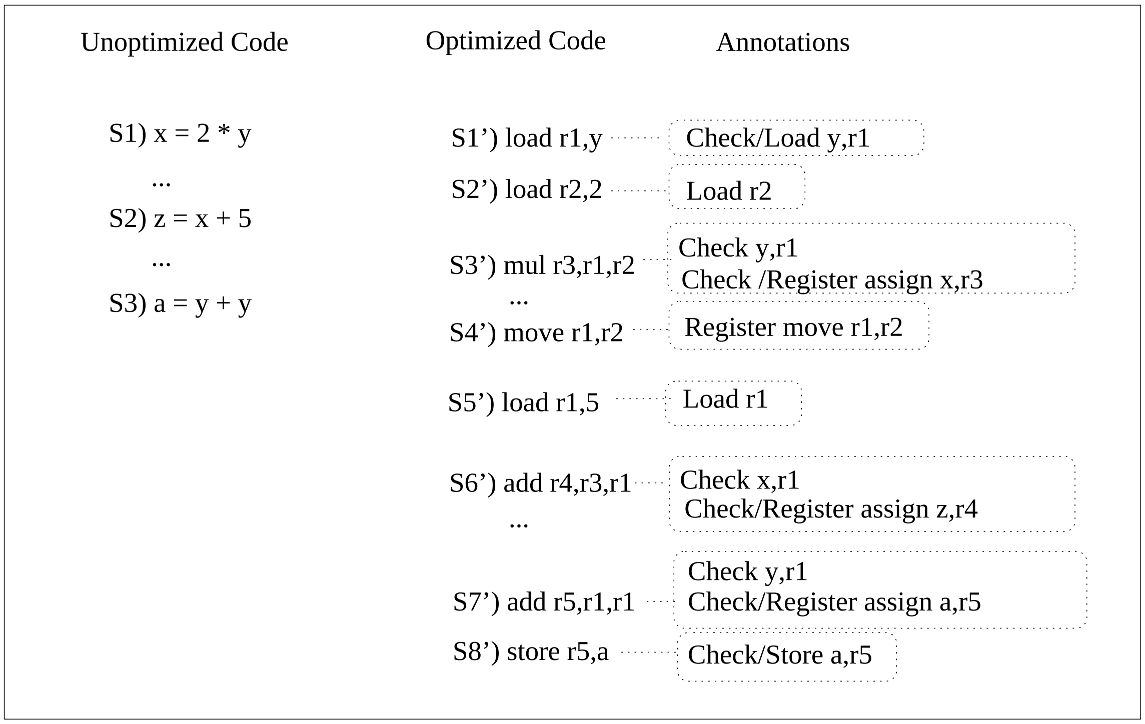


Figure 6.3: Annotations example

optimized code. For example, in Figure 6.3, a *check* annotation is associated with statement $S1'$ in the optimized code so that the contents of $r1$ in the optimized code is compared with the value of y in the unoptimized code. *Check* annotations are used to check register loads, stores, uses, and assignments.

6.2.2 The Register assign [v,] r annotation

The *register assign* annotation is associated with a program point in the optimized code to indicate the tracking information for register r should be updated. The register allocation checker records that the previous variable assigned to r is evicted. If v is specified, the register allocation checker updates its information to indicate that r holds variable v, v is currently stored in r, the memory location of v holds a stale value, and any other values of v currently in registers are evicted. For example, in Figure 6.3, a *register assign* annotation is associated with statement $S3'$ in the optimized code so that the variable x is tracked with register $r3$ in the optimized code.

6.2.3 The Load [v,] r annotation

The *load* annotation is associated with a *load* instruction in the optimized code and is used to track and verify the load information for register r. The register allocation checker records that the previous variable assigned to r is evicted. If v is specified, the register allocation checker records that r holds variable v, v is currently stored in r, and any other values of v currently in registers are evicted. Using the tracking information, the checker verifies if the loaded value is stale, and if so, records this information, informs the user of the stale value of v, and informs the user of the current location of the expected value of v, if it exists. For example, in Figure 6.3, a *load* annotation is associated with statement $S1'$ in the optimized code to track and verify the information in register $r1$.

6.2.4 The Store $\mathbf{v}, \mathbf{r}$ annotation

The *store* annotation is associated with a *store* instruction in the optimized code to track and verify the store information for register r. Using the tracking information, the checker verifies if r does not hold the expected value of v, and if so, informs the user that r does not hold the expected value of v and informs the user of the current location of v, if it exists. Also, the register allocation checker records that the memory location of v holds the current value. For example, in Figure 6.3, a *store* annotation is associated with statement $S8'$.

6.2.5 The Register move $\mathbf{r}, \mathbf{r}'$ annotation

The *register move* annotation is associated with a *move* instruction in the optimized code to track the information in register r'. The register allocation checker duplicates the information pertaining to register r for that of register r'. For example, in Figure 6.3, a *register move* annotation is associated with statement $S4'$ in the optimized code to track the information in register $r2$.

6.2.6 Combining annotations

When a *Check* annotation is associated with a *Store* annotation, the checker verifies that the value in the register stored in the memory location of the variable at a program point in the optimized program matches the value of the variable at the equivalent program point in the unoptimized program. If the values do not match, then if the register currently holds the variable, then the checker informs the user why the value in the optimized code is incorrect. Either the value is stale, uninitialized, or wrong (possibly because the correct value was evicted and now the register contains the wrong value that will be stored). If the register does not currently hold the variable, the checker informs the user (1) if the expected value of the variable resides in another register, (2) the last location of the variable, and (3) that either the wrong register or address was supplied in the instruction, the expected value was evicted earlier and not saved, or the memory value already has the expected value (because of an earlier store). A *Check* annotation associated with a *Load* annotation is treated in a similar manner.

When a *Check* annotation is associated with a *Register assign* annotation, the checker verifies that the value assigned to the register at a program point in the optimized code matches the value of the variable at the equivalent program point in the unoptimized code. If the operands were incorrect, the checker will have already notified the user of the uses that have unexpected values. Otherwise, incorrect code was generated.

6.3 Annotation placement

Annotations are placed in the optimized program as follows. Using the mappings, *Check/Register assign* annotations are placed on every variable assignment in the optimized code and *Check* annotations are placed on every variable use in the optimized code. Next, at every instruction in the optimized code that stores to a register, *Register assign* annotations are placed, except at the program points where *Check/Register assign* annotations have been placed. At every instruction in the optimized code that loads a variable into a register, *Check/Load* annotations are placed. At all other load instructions in the optimized code, *Load* annotations are placed. Similarly, at every instruction in the optimized code that stores to a memory location of a variable, *Check/Store* annotations are placed. At all other store instructions in the optimized code, *Store* annotations are placed. Finally, at every move instruction in the optimized code, *Register move* annotations are placed.

6.4 Register allocation checker example

Consider the annotated unoptimized and optimized program segments in Figure 6.4, which illustrate the same unoptimized and optimized code example given in Figure 6.1. Breakpoints are indicated by circles. Annotations are shown in dotted boxes. The optimized program starts to execute with $S1'$, and breakpoint 1 is reached. The checker determines from the annotation that the value loaded into register $r1$ should be compared with the value of y in the unoptimized code at the equivalent program point in the unoptimized code. Thus, the unoptimized program executes until breakpoint 2 is reached, at which time the checker compares the value of y in the unoptimized program with the value of $r1$ in the optimized code. If the values are the same, the checker determines from the *Load* annotation that information regarding y and $r1$ should be tracked. The checker records that $r1$ now holds the value of y and that y is currently stored in $r1$. If y is stored in any other register, the checker records that y is evicted from these other registers. Also, if the loaded value is stale, the checker informs the user of the stale value and the location of the expected value of the variable (if it exists).

The optimized program continues execution and breakpoint 3 is reached. The checker processes the *Load* annotation by recording that the latest variable in $r2$ is now evicted. The optimized and unoptimized programs continue executing in a similar manner.

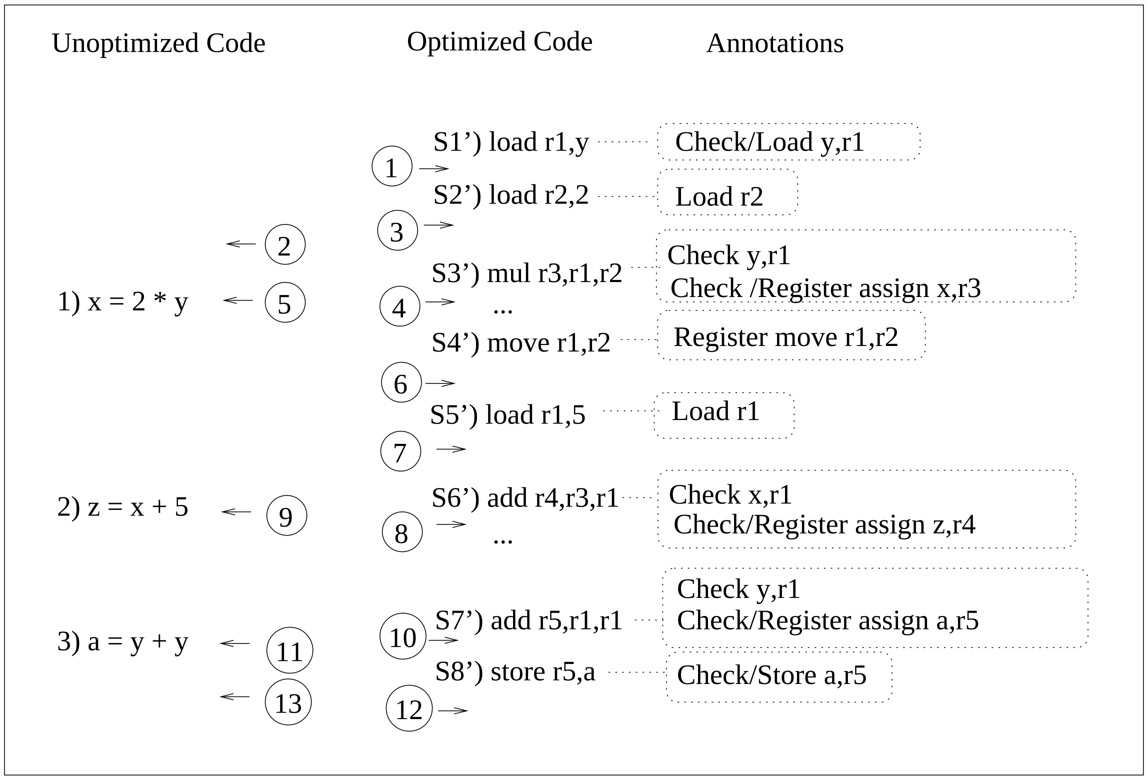


Figure 6.4: Register allocation checker example

Notice that when breakpoint 6 is reached, the checker processes the *Register move* annotation and records that $r2$ holds the value of y and y is stored in $r2$. At breakpoint 7, the checker processes the *Load* annotation and records that y is evicted from $r1$. At breakpoint 10, the checker processes the Check annotation by executing the unoptimized program until breakpoint 11 is reached, at which time the value of y in the unoptimized code is compared with $r1$. The values differ and the checker informs the user that y was evicted from $r1$ and the expected value of y in the optimized code is in $r2$.

6.5 Summary

The register allocation checker provides a finer level of checking, which helps an optimizer writer debug and validate an implementation of register allocation. This level of checking and tracking enables the checker to locate the earliest execution point where the unoptimized and optimized programs differ in their semantic behavior and display to the user the actual cause(s) of the differences. For example, the register allocation checker can inform the user when a stale value is used, a wrong register is used, and when a value is evicted from a register but not saved for future uses. The register allocation checker can be incorporated into the comparison checker or can be used as a standalone tool that is used after optimizations are applied.

Chapter 7
Source Level Debugger

In this chapter, the mappings described in Chapter 4 are utilized to develop a source level debugger of optimized code that extends the class of reportable expected values of previous work by reporting all expected values that are computed in the optimized program. That is, every value of a source variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. Expected values at breakpoints are reportable even though reportability is affected because values have been overwritten early or written late. Expected values at breakpoints are also reportable even though values are path sensitive in that a value may be computed only along one path or the location of the value may be different along different paths. The only values that are not reportable are those that are deleted on a path by an optimization. However in these cases, the debugger reports the value has been deleted. This level of reporting is considered "full reporting" and thus the debugger developed in this dissertation is called FULLDOC, a **FULL** reporting **D**ebugger of **O**ptimized **C**ode.

The design of the source level debugger is more complex than that of the comparison checker for a number of reasons. The comparison checker uses expected values that are computed in the unoptimized and optimized programs only at certain program points in order to perform comparison checks, but the debugger needs to be able to report expected values that are computed at all breakpoints within their reportable ranges. Next, the comparison checker delays the comparison checking of those values computed in the unoptimized program whose corresponding values are computed later in the optimized code. For the case of the debugger, if a user queries a variable at a breakpoint whose expected value is computed later in the optimized code, the debugger should not delay reporting the expected value. Finally, the comparison checker utilizes all of the annotations to automate the checking since all values computed in both programs are compared, but to minimize the runtime overhead of the debugger, the debugger should only deal with reportability of expected values that are affected at current user breakpoints.

FULLDOC extends the class of reportable expected values by judiciously using both static and dynamic information. The overall strategy is to determine, by static program analysis, those values that the optimizer has placed in a precarious position in that their values may not be reportable. The reportability of these values may depend on runtime and debugging information, including the placement of the breakpoints and the paths taken in a program's execution. Thus, during execution, invisible breakpoints are employed to gather dynamic information that aids in the reporting of precariously placed values. Three schemes, all transparent to the user during a debugging session, are employed to enable full reporting. To report values that are overwritten early with respect to a breakpoint either because of code motion or register reuse, FULLDOC saves the values before they are overwritten and deletes them as soon as they are no longer needed for reporting. FULLDOC only saves the values if they are, indeed, the expected values at the breakpoint. To report values that are written late with respect to a breakpoint because of code sinking, FULLDOC prematurely executes the optimized program until it can report the value, saving the values overwritten by the roll ahead execution so that they can be reported at subsequent breakpoints. When reportability of a variable at a breakpoint is dependent on the execution path of the optimized code, FULLDOC dynamically records information to indicate the impact of the path on the reportability of a value, and thus is able to report values that are path sensitive either because the computation of the value or the location is dependent on the path.

FULLDOC's technique is non-invasive in that the code that executes is the code that the optimizer generated. Also, unlike the emulation technique [51], instructions are not executed in a different order and thus the problem of masking user and optimizer errors is avoided.

The capabilities of FULLDOC are as follows.

- Every value of a source variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. Therefore, FULLDOC can report more expected values that are computed in the optimized program execution than any existing technique [27, 21, 14, 37, 19, 49, 9, 8, 10, 7, 22, 51, 45]. Values that are not computed in the optimized program execution are the only values that are not reported. However, FULLDOC can incorporate existing techniques that recover some of these values.

- Runtime overhead is minimized by performing all analysis during compilation. FULLDOC utilizes debugging information generated during compilation to determine the impact of reportability of values at user breakpoints and to determine the invisible breakpoints that must be inserted to report affected values.
- The techniques are transparent to the user. If a user inserts a breakpoint where the reportability of values is affected at the breakpoint or a potential breakpoint, FULLDOC automatically inserts invisible breakpoints to gather dynamic information to report the expected values.
- User breakpoints can be placed between any two source level statements, regardless of the optimizations applied.
- The optimized program is not modified except for setting breakpoints.
- Breakpoints in the source code are syntactically mapped in the optimized code.

While a wide range of optimizations from simple code reordering transformations to loop transformations are supported, this chapter focuses mainly on statement level optimizations that hoist and sink code, including speculative code motion, path sensitive optimizations (e.g., partial redundancy elimination), and register allocation.

The rest of this chapter is organized by Section 7.1 describing the challenges of reporting expected values using examples. Section 7.2 describes FULLDOC's approach. Sections 7.3 and 7.4 describe the debug information as well as how the debug information is computed and used. Section 7.5 describes how to extend FULLDOC to support loop transformations and inlining. Section 7.6 presents experimental results.

7.1 Challenges of reporting expected values

The reportability of a variable's value involved in an optimization is affected by

1. register reuse, code reordering, and code deletion,
2. the execution path, including loop iterations, and
3. the placement of breakpoints.

This section considers the effect of optimizations that can cause a value of a variable to be overwritten early, written late, or deleted. Within each of these cases, the impact of the path and the placement of breakpoints is considered. This section also demonstrates how FULLDOC handles these cases. In the figures, the paths highlighted are the regions in which reportability is affected. Reportability is not affected in the other regions.

7.1.1 Overwritten early in the optimized program

A value *val* of a variable *v* is *overwritten early* in the optimized program if another value *val′* prematurely overwrites *v*'s value. The application of a code hoisting optimization and register reuse can cause values to be overwritten early. For example, consider the unoptimized program and its optimized version in Figure 7.1(a), where X^n refers to the n^{th} definition of X. X^2 has been speculatively hoisted, and as a result, the reportability of X is affected. Regardless of the execution path of the optimized code, a debugger cannot report the expected value of X at a breakpoint b along region ① by simply displaying the *actual* contents of X. The *expected* value of X at b is the value of X^1, but since X^2 is computed early, causing the previous value (i.e., X^1) to be overwritten early, the actual value of X at b is X^2.

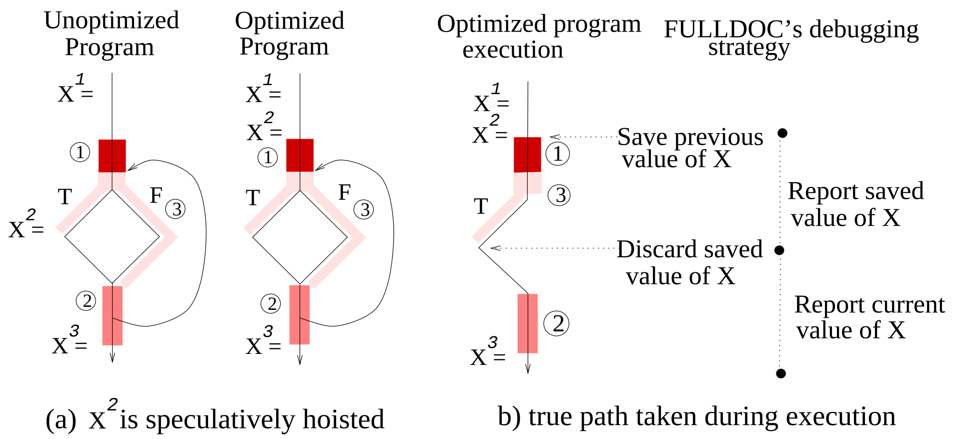

Unoptimized Program
Optimized Program
Optimized program execution
FULLDOC's debugging strategy
X^1=
X^2=
X^3=
①
②
③
T
F
Save previous value of X
Report saved value of X
Discard saved value of X
Report current value of X
(a) X^2 is speculatively hoisted
b) true path taken during execution

Figure 7.1: Overwritten early example

The path can also affect reportability. Assume now that a breakpoint b is placed in region ②. The expected value of X at b is either X^2, if the true path is taken, or X^1, if only the false path is taken within each loop iteration. However, since X^2 is computed before the branch, the actual value of X at b in the optimized code is X^2. Thus, when execution follows the true path, the expected value of X at b can be reported, but when only the false path is taken, its value cannot be reported.

The number of loop iterations can also affect reportability. The expected value of X at a breakpoint b along region ③ depends not only on whether the true path was taken but also on the current loop iteration. During the first loop iteration, the expected value is X^1. On subsequent loop iterations, the expected value is either X^2 (if the true path is taken) or X^1 (if only the false path is taken on prior loop iterations). However, since X^2 is computed before the loop, the actual value of X at b in the optimized code is X^2. When execution follows the true path, the debugger can report the expected value of X at b on subsequent loop iterations. Otherwise, the debugger cannot report the expected value of X.

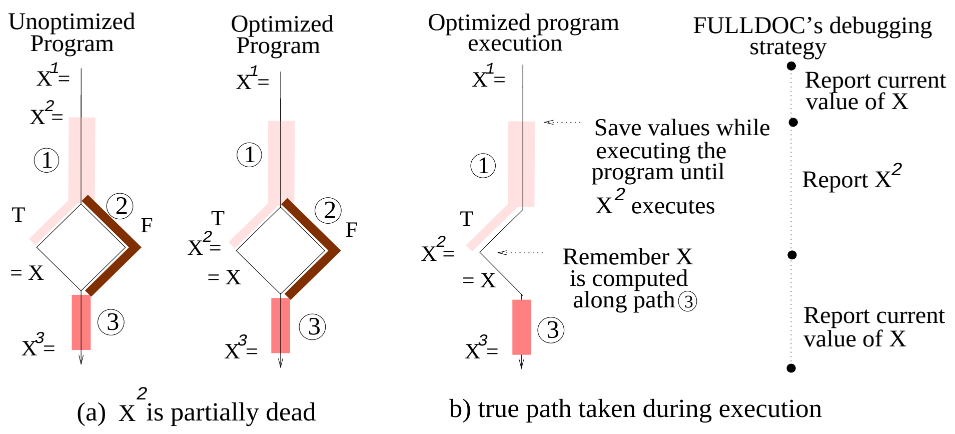
Using only dynamic currency determination [22], the expected value of X at breakpoints along region ① cannot be reported because the value has been overwritten. The emulation technique [51] can report the expected value of X along region ① and along the true path of region ③ , but since the technique is not path sensitive, the expected value cannot be reported along region ② and along the false path of region ③ due to iterations.

FULLDOC can report all of these expected values. During the execution of the optimized code, if a value is overwritten early **with respect to a breakpoint**, FULLDOC saves the value in a *value pool.* FULLDOC only saves what is necessary and discards values when they are no longer needed for reporting. Figure 7.1(b) illustrates FULLDOC's strategy when the optimized program in Figure 7.1(a) executes along the true path, assuming the loop executes one time. FULLDOC saves X^1 before the assignment to X^2 and reports the saved value X^1 at breakpoints along regions ① and ③. FULLDOC discards the saved value when execution reaches the original position of X^2. At breakpoints along the non-highlighted path and region ②, FULLDOC reports the current value of X. Notice that values are saved only as long as they could have been observable at a breakpoint in the source program, and thus, the save/discard mechanism automatically disambiguates which value to report at breakpoints along region ②. If X^1 is currently saved at the breakpoint, then only the false path was executed and the saved value is reported. Otherwise if X^1 is not currently saved, then the true path was executed and the current value of X is reported. Notice that this saving strategy, as well as the other strategies, is performed with respect to user breakpoints. In other words, if a user does not insert breakpoints along the regions where the reportability of X is affected, then FULLDOC does not save the value of X.

7.1.2 Written late in the optimized program

A value *val* of a variable *v* is *written late* in the optimized program if the computation of *val* is delayed due to, for example, code sinking and partial dead code elimination. In Figure 7.2(a), suppose X^2 is partially dead along the false path and moved to the true branch. As a result, the expected value of X at a breakpoint b along regions ① and ② is not reportable in the optimized code.

Consider a breakpoint b placed in region ③. The expected value of X at b is X^2. However, the actual value of X at b in the optimized code is either X^2 (if the true path is taken) or X^1 (if the false path is taken). Thus, only when execution follows the true path,



Unoptimized Program
X^1=
X^2=
①
T
②
F
= X
③
X^3=
Optimized Program
X^1=
①
T
②
F
X^2=
= X
③
X^3=
(a) X^2 is partially dead
Optimized program execution
X^1=
Save values while executing the program until X^2 executes
①
T
X^2=
= X
Remember X is computed along path ③
③
X^3=
FULLDOC's debugging strategy
Report current value of X
Report X^2
Report current value of X
b) true path taken during execution

Figure 7.2: Written late example

can the expected value of X at b be reported. Reportability can also be affected by loop iterations, which has the same effect as for the overwritten early case.

Using only dynamic currency determination [22], the expected value of X at breakpoints along region ③ can be reported provided the true path is taken but not along regions ① and ② . Since the emulation technique [51] is not path sensitive, the expected value of X along region ③ cannot be reported. FULLDOC can report values in ① and ③ provided the true path is taken. Note that values in regions ①, ②, and ③ could possibly be reported by all schemes if recovery techniques are employed.

If a requested value is written late **with respect to a breakpoint**, FULLDOC prematurely executes the optimized code, saving previously computed values before they are overwritten (so that they can be reported at subsequent breakpoints). Figure 7.2(b) illustrates FULLDOC's strategy when the optimized program in Figure 7.2(a) executes along the true path. At breakpoints along region ①, FULLDOC reports the expected value of X by further executing the optimized code, saving previously computed values before they are overwritten. The roll ahead execution stops once X^2 executes. At breakpoints along the non-highlighted path and region ③, FULLDOC reports X^2.

7.1.3 Computed in the unoptimized program but not in the optimized program

Finally, consider the case where a statement is deleted and thus its value is not computed in the optimized code. For example, in Figure 7.3(a), suppose Y^2 is dead in the unoptimized program and deleted. The expected value of Y at a breakpoint b along region ① is Y^2, which cannot be reported in the optimized code.

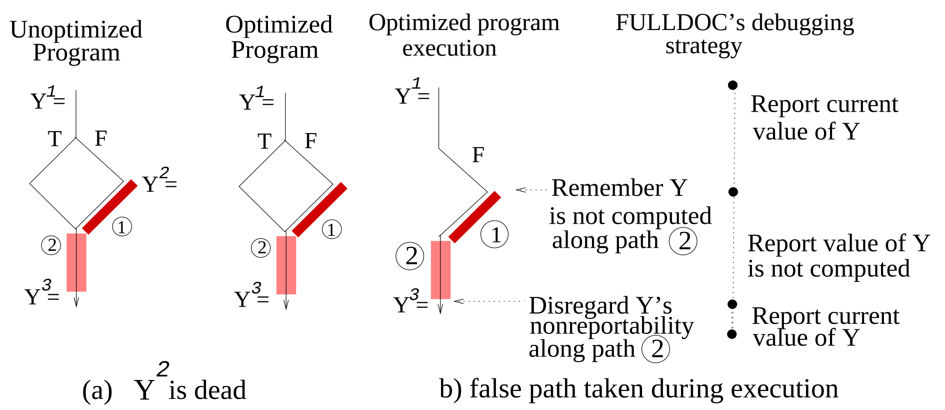

Unoptimized Program
Optimized Program
Optimized program execution
FULLDOC's debugging strategy
Y1=
T
F
Y2=
Y3=
①
②
Remember Y is not computed along path ②
Disregard Y's nonreportability along path ②
Report current value of Y
Report value of Y is not computed
Report current value of Y
(a) Y2 is dead
b) false path taken during execution

Figure 7.3: Not computed in the optimized program example

Now consider placing a breakpoint at region ② . The expected value of Y at b along region ② is either Y^1 (if the true path is taken) or Y^2 (if the false path is taken). However, since Y^2 was deleted, the actual value of Y at b in the optimized code is Y^1. Thus, along the true path, the actual value is the expected value and can be reported, but along the false path, the expected value cannot be reported.

The emulation technique [51] cannot report the expected value of Y along region ② because it is not path sensitive. Dynamic currency determination [22] as well as FULLDOC can report the expected value of Y at breakpoints along region ② if the true path is taken.

Figure 7.3(b) illustrates FULLDOC's strategy when the optimized program in Figure 7.3(a) executes along the false path. At a breakpoint along the non-highlighted paths, FULLDOC reports the current value of Y. When execution reaches the original position of Y^2, FULLDOC knows Y is not reportable along regions ① and ②, and reports the expected value of Y is not computed. When execution reaches Y^3, FULLDOC disregards the non-reportability information of Y.

7.2 FULLDOC's approach

FULLDOC uses three sources of *debug information* for its debugging capabilities. First, *mappings* between corresponding instances of statements in the unoptimized and optimized programs, which are described in Chapter 4, are generated as optimizations are applied. Second, after code is optimized and generated by the compiler, static analysis is applied to gather information about the reportability of expected values. This *reportability debug information* is used when user breakpoints are inserted, special program points are reached in the program execution, or when a user breakpoint is reached. Third, during

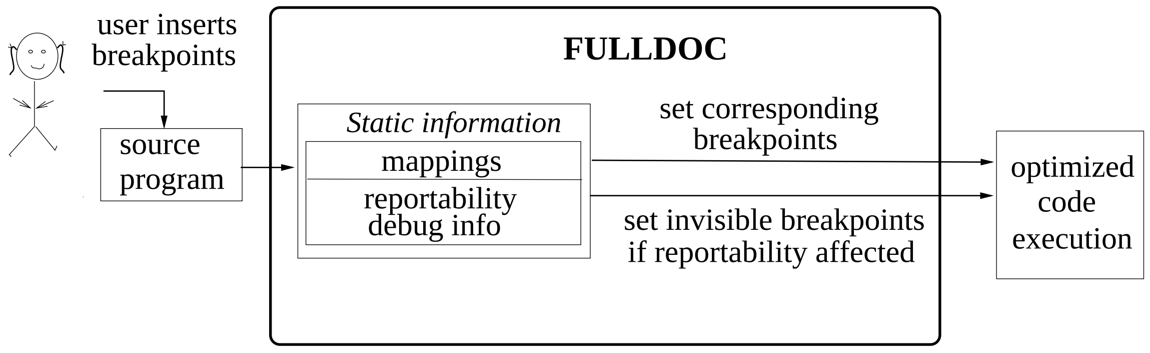


Figure 7.4: FULLDOC's strategy with respect to user inserting breakpoints

execution, *dynamic debug information* indicating that these special points have been reached is used as well as the position of the user breakpoints to enable full reporting.

Figure 7.4 illustrates FULLDOC's strategy with respect to a user inserting breakpoints. When the user inserts breakpoints either before the program executes or during program execution, FULLDOC uses the mappings to determine the corresponding breakpoints in the optimized code. FULLDOC uses the reportability debug information to determine the impact on reportability at the breakpoints and potential breakpoints:

• If a value is *overwritten early* with respect to a breakpoint, FULLDOC inserts *invisible breakpoints* [52] to *save* the value during execution as long as the value should be reportable and *discard* the value when it is no longer needed.

• If the reportability of a variable with respect to a breakpoint is path sensitive, FULLDOC inserts invisible breakpoints to update the dynamic debug information regarding the reportability of the value.

Figure 7.5 illustrates FULLDOC's strategy when either a user or invisible breakpoint is reached. If a user breakpoint is reached, FULLDOC informs the user. When invisible breakpoints are reached, FULLDOC performs the following actions. For a value that is *overwritten early*, FULLDOC *saves* the value in a *value pool* and later *discards* the value when it is no longer needed for reporting. For a value that is path sensitive, FULLDOC updates the *path sensitive info* regarding the reportability of the value depending on the execution path taken.

Figure 7.6 illustrates FULLDOC's strategy with respect to user queries at a user breakpoint. FULLDOC responds to user queries by using both static and dynamic information. When the user requests the value of a variable, FULLDOC uses the reportability debug information and dynamic debug information to determine the reportability of the value. If the value is available at the location (in memory or register) of the variable or in the value pool, FULLDOC reports the value. If the reportability of the value is path sensitive at the breakpoint, FULLDOC uses the path sensitive information to determine whether the value is reportable at the breakpoint. If the requested value is *written late* with respect to the breakpoint, FULLDOC uses the reportability debug information to *roll ahead* the execution of the optimized code, saving previously computed values before they are overwritten. It *stops* execution once the value is computed and reports the value to the user if it is computed. If the value is not computed in the execution, FULLDOC informs the user that the value is not reportable.

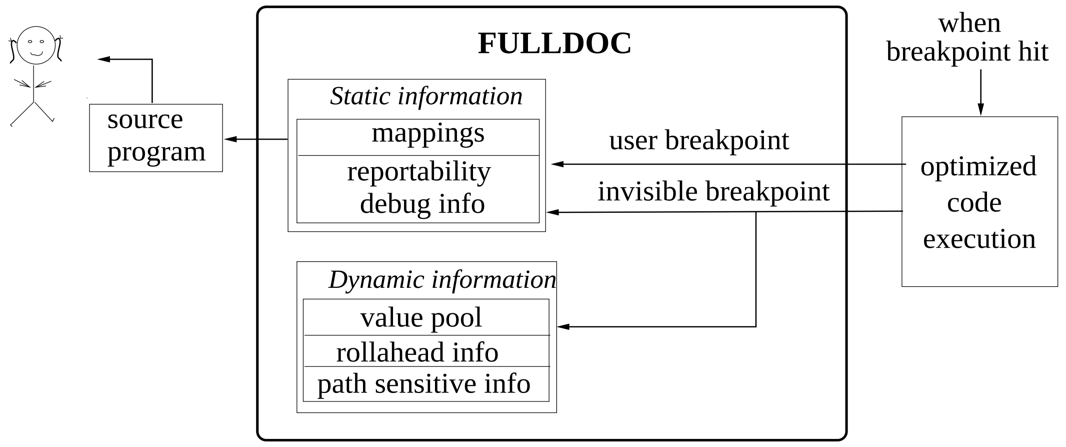


Figure 7.5: FULLDOC's strategy with respect to breakpoints hit

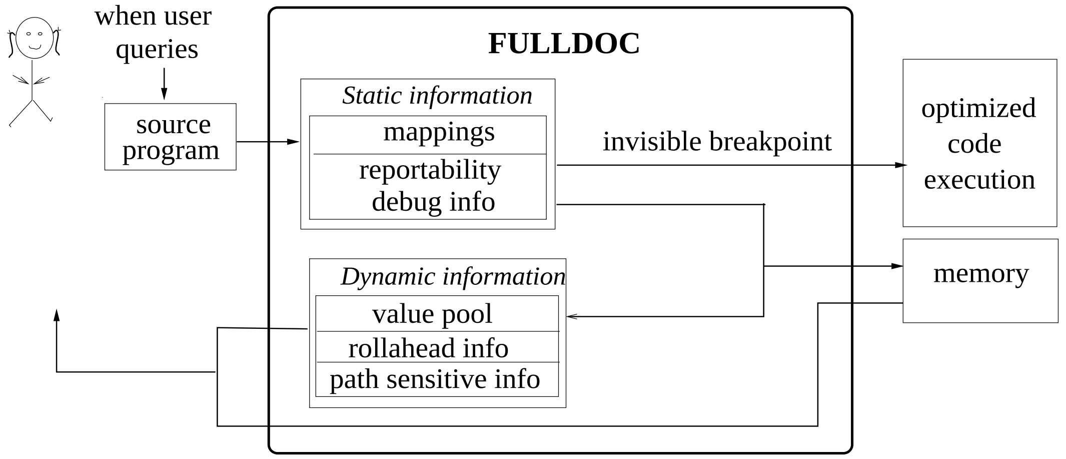
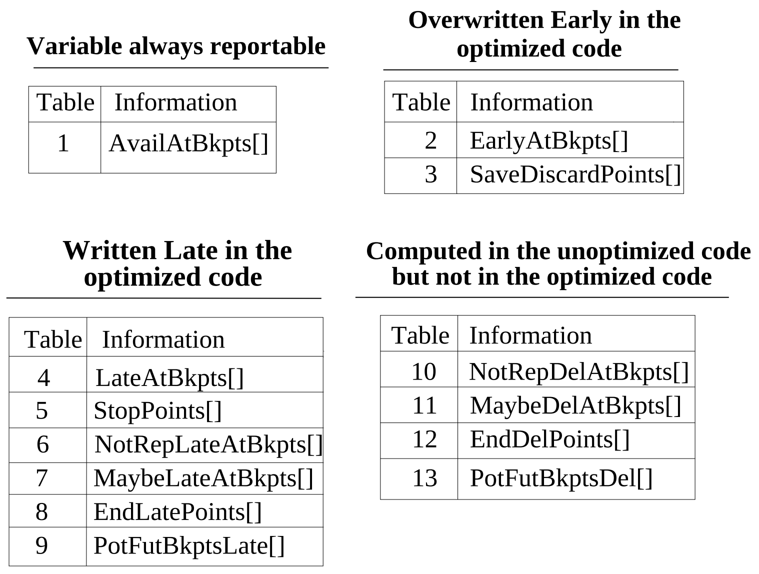


Figure 7.6: FULLDOC's strategy with respect to user queries

7.3 Reportability debug information

This section describes the reportability debug information computed through static analysis of the optimized code that is provided to FULLDOC as well as how FULLDOC employs this information at runtime and how it collects dynamic debug information in response to the user setting breakpoints and requesting values of variables at these breakpoints. The debug information is organized by tables, which are shown in Figure 7.7. Table 1 contains the debug information for the expected values of variables that are always reportable. Tables 2-3 contain the debug information for reporting the values of variables that are overwritten or may have been overwritten early. Tables 4-9 contain the debug information for reporting the values of variables that are written or may have been overwritten late. Tables 10-13 contain the debug information for determining the values of variables that are not computed in the execution of the optimized code. The rest of this section describes the tables in detail.



Variable always reportable

Table	Information
1	AvailAtBkpts[]

Overwritten Early in the optimized code

Table	Information
2	EarlyAtBkpts[]
3	SaveDiscardPoints[]

Written Late in the optimized code

Table	Information
4	LateAtBkpts[]
5	StopPoints[]
6	NotRepLateAtBkpts[]
7	MaybeLateAtBkpts[]
8	EndLatePoints[]
9	PotFutBkptsLate[]

Computed in the unoptimized code but not in the optimized code

Table	Information
10	NotRepDelAtBkpts[]
11	MaybeDelAtBkpts[]
12	EndDelPoints[]
13	PotFutBkptsDel[]

Figure 7.7: Debug information

7.3.1 Simply reportable

The `AvailAtBkpts` table indicates the program ranges in the optimized code where the expected values of source variables are always available for reporting.

$$\boxed{\texttt{AvailAtBkpts[b,v] = \{loc\} or \{(def1,loc1), (def2,loc2), ...\}}}$$

If the value of variable `v` is always reportable at breakpoint `b`, then `AvailAt-Bkpts[b,v]` provides the location (memory location or register name) where the value of `v` can be found. In case the value can always be found at the same location, no matter what execution path is taken, `loc` provides the location.

However, it is possible that the location of `v` depends on the path taken during execution because `b` is reachable by multiple definitions of `v`, each of which stores the value of `v` in a different location (e.g., a different register). In this case, the execution path taken determines the latest definition of `v` that is encountered and hence the location where the value of `v` can be found. Each of the potential definition-location pairs, `(defi,loci)`, are provided by `AvailAtBkpts[b,v]` in this case. When a breakpoint is set at `b`, the debugger *activates* the recording of the definition of `v` that is encountered from among `(def1, def2, ...)` by inserting invisible breakpoints at each of these points. When an invisible breakpoint is hit during execution, the debugger records the latest definition encountered by overwriting the previously recorded definition.

7.3.2 Overwritten early

The `EarlyAtBkpts` table indicates the program ranges in the optimized code where expected values of source variables are not available because such values may have been overwritten early.

```
EarlyAtBkpts[b] = {es:  es overwrites early w.r.t.  breakpoint b}
SaveDiscardPoints[es] = (save, {discard1, discard2, ...})
```

For FULLDOC to report such values at the effected program ranges, the `EarlyAtBkpts` and `SaveDiscardPoints` tables are used as follows. If the user sets a breakpoint at `b`, then for each statement `es` that overwrites early in `EarlyAtBkpts[b]`, the save and discard points in `SaveDiscardPoints[es]` are activated by inserting invisible breakpoints. This ensures that the values of variables overwritten early with respect to breakpoint `b` will be saved and available for reporting at `b` from the value pool in case they are requested by the user. Note that the save and discard points must be activated immediately when a breakpoint is set by the user so that all values that may be requested by the user, when the breakpoint is hit, are saved. If a discard point is reached along a path and nothing is currently saved because a save point was not reached along the same path, the debugger simply ignores the discard point.

The example in Figure 7.8 (also illustrated in Figure 7.1 of Section 7.1) where X is overwritten early is handled by this case. The highlighted regions are the regions where reportability of X is affected. At breakpoints along region ① , the reportability of X is affected, regardless of the execution path taken. At breakpoints along region ② , the

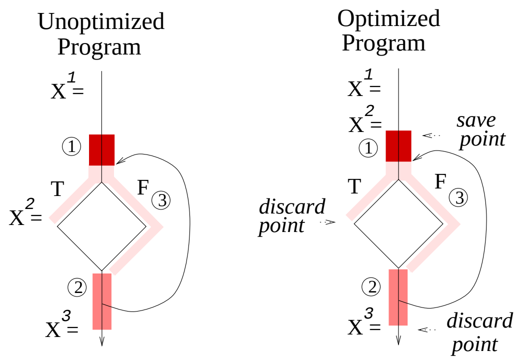


Figure 7.8: Overwritten early example

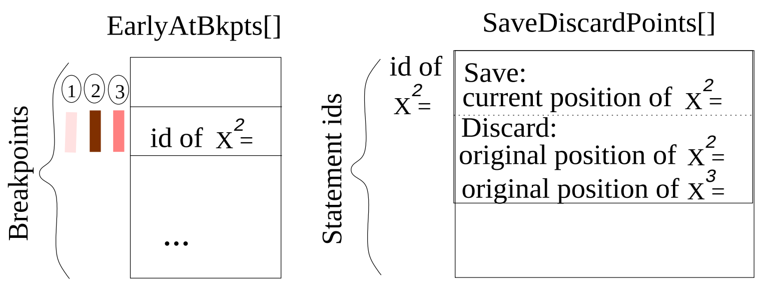


Figure 7.9: EarlyAtBkpts and SaveDiscardPoints reportability debug information for overwritten early example in Figure 7.8

reportability of X is path sensitive. At breakpoints along region ③, the reportability of X is path sensitive and depends on the current loop iteration. These three cases where the reportability of X is affected are handled by the `EarlyAtBkpts` and `SaveDiscardPoints` tables. The save and discard points are illustrated in Figure 7.8 and the `EarlyAtBkpts` and `SaveDiscardPoints` tables are displayed in Figure 7.9.

7.3.3 Written late

The `LateAtBkpts` table indicates the program ranges in the optimized code where expected values of source variables are not available because such values may be overwritten late and may still execute.

```
LateAtBkpts[b]  = {ls:  ls writes late w.r.t.  breakpoint b}
StopPoints[ls]  = {stop1, stop2, ...}
```

For FULLDOC to report such values at the effected program ranges, the `LateAtBkpts` and `StopPoints` tables are used as follows. Assume the user sets a breakpoint at `b`. Then for each statement `ls` $\in$ `LateAtBkpts[b]`, FULLDOC must first determine if `ls` is written late with respect to the next instance of the breakpoint `b`. If the original position of `ls` is reached during execution but the current position of `ls` is not reached (before the breakpoint `b` is hit), then `ls` is written late. This information is determined as follows. For each statement `ls` that is written late, FULLDOC inserts invisible breakpoints at the original and current positions of `ls` and records if the original position of `ls` is encountered during execution. When the current position of `ls` is reached during execution, the recorded information is discarded. Now, suppose execution reaches `b`, and the user requests the value of a variable `v` such that `v` is written late by a statement `ls` in `LateAtBkpts[b]`. If the original position of `ls` is currently recorded, then `v` is late at the current instance of the breakpoint `b` and the execution of the program rolls ahead until one of the stop points in `StopPoints[ls]` is encountered. At a stop point, either the value of `v` has just been computed or it is known that it will definitely not be computed (recall that sinking of partially dead code can cause such situations to arise). Unlike the overwritten early case where the save and discard points were activated when a breakpoint was set, here the stop points are activated when the breakpoint is hit and a request for a value that is written late is made.

The example in Figure 7.10 (also illustrated in Figure 7.2 of Section 7.1), where the reportability of X along region ① is affected is handled by this case. The stop points are illustrated in Figure 7.10 and the `LateAtBkpts` and `StopPoints` tables are displayed in Figure 7.11.

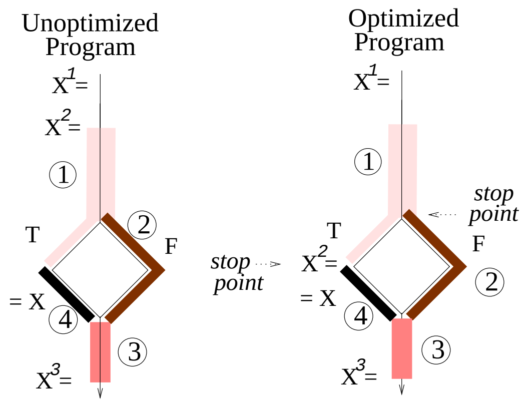


Figure 7.10: Overwritten late example

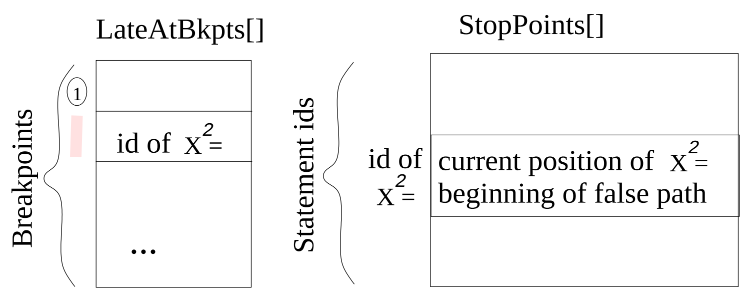


Figure 7.11: LateAtBkpts and StopPoints reportability debug information for overwritten late example in Figure 7.10

7.3.4 Never reportable because deleted along a path

When (partial) dead code removal is performed, the value of a variable defined by the deleted statement becomes unreportable. For each breakpoint `b`, the variables whose values are never reportable at `b`, no matter what execution path is taken, are recorded in `NotRepDelAtBkpts[b]` and `NotRepLateAtBkpts[b]`, for statements removed from paths by dead code elimination and partial dead code elimination, respectively.

```
NotRepDelAtBkpts[b] = {v:  v is never reportable at b (deleted)}
NotRepLateAtBkpts[b] = {v:  v is never reportable at b (late)}
```

When the user requests the value of a variable `v` at breakpoint `b`, if `v` is in `NotRepDelAtBkpts[b]` or `NotRepLateAtBkpts[b]`, FULLDOC reports to the user that the value is not reportable because the statement that computes it has been deleted along the execution path.

The example in Figure 7.12 (also illustrated in Figure 7.3 of Section 7.1), where the reportability of Y is affected along region ①, is handled by this case. The `NotRepDelAt-Bkpts` table is displayed in Figure 7.13. Also, the example in Figure 7.10, where the reportability of X is affected along region ② is handled by this case. The `NotRepLateAtBkpts` table is displayed in Figure 7.14.

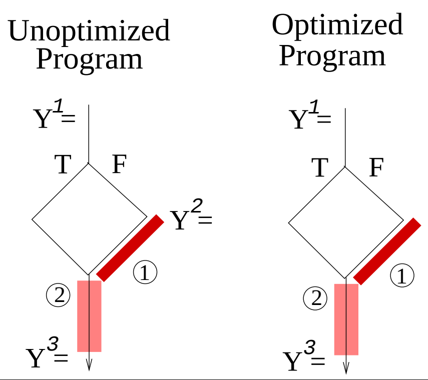


Figure 7.12: Dead code elimination example

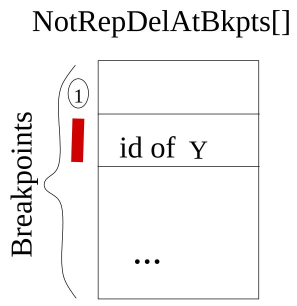


Figure 7.13: NotRepDelAtBkpts reportability debug information for the example in Figure 7.12

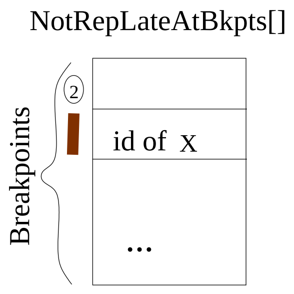


Figure 7.14: NotRepLateAtBkpts reportability debug information for the example in Figure 7.10

7.3.5 Path sensitive nonreportability/reportability when deleted

A value may be deleted on one path, in which case it is not reportable, and not deleted on another path, in which case it is reportable. In this path sensitive case, the reportability information must be updated during execution, based on the paths that are actually executed (i.e., program points reached).

```
MaybeDelAtBkpts[b] = {ds:  ds may be deleted w.r.t.  breakpoint b}
EndDelPoints[ds] = {EndDel1, EndDel2, ...}
PotFutBkptsDel[b] = {ds:  ds may be deleted at later breakpoints}
```

If a user sets a breakpoint at `b`, invisible breakpoints are set at each of the original positions of any deleted statement `ds` in `MaybeDelAtBkpts[b]` to record if one of these positions is encountered during execution. Invisible breakpoints are also set at the end of the definition range of `ds`, stored in `EndDelPoints[ds]`. When `EndDeli` in `EndDelPoints[ds]` is reached during execution, the recorded information is discarded. Now consider the case when breakpoint `b` is reached, and the user requests the value of variable `v` defined by some statement `ds` in `MaybeDelAtBkpts[b]`. If the dynamically recorded information shows that the original position of `ds` was encountered, the debugger reports that the value of `v` was not computed as `ds` was deleted. Otherwise the debugger reports the current value of `v`. The example in Figure 7.12, where the reportability of Y along region ② is path sensitive, is handled by this case. The `MaybeDelAtBkpts` and `EndDelPoints` tables are displayed in Figure 7.15.

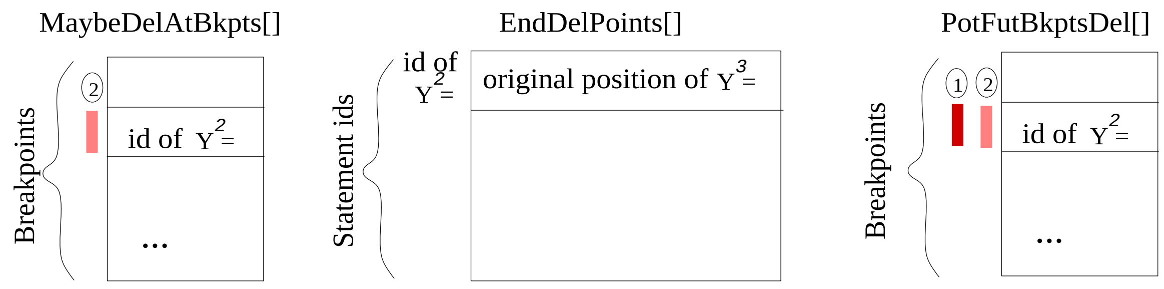


Figure 7.15: MaybeDelAtBkpts, EndDelPoints, and PotFutBkptsDel reportability debug information for example in Figure 7.12

The same strategy is used for each deleted statement in `PotFutBkptsDel[b]`, which prevents FULLDOC from setting invisible breakpoints too late. `PotFutBkptsDel[b]` holds the deleted statements where reportability could be affected at potential breakpoints even though reportability is not necessarily affected at `b`. Invisible breakpoints must now be set so that during the execution to breakpoint `b`, FULLDOC gathers the appropriate dynamic information for the potential breakpoints. The `PotFutBkptsDel` table for the example in Figure 7.12 is displayed in Figure 7.15.

7.3.6 Path sensitive nonreportability/reportability when written late

Sinking code can also involve path sensitive reporting, because a statement may be sunk on one path and not another. This case is the opposite to the previous one in that if a late statement is encountered, it is reportable.

```
MaybeLateAtBkpts[b] = {ls:  ls may be late w.r.t.  breakpoint b}
EndLatePoints[ls] = {EndLate1, EndLate2, ...}
PotFutBkptsLate[b] = {ls:  ls may be late at later breakpoints}
```

If the user sets a breakpoint at `b`, the debugger initiates the recording of the late statements in `MaybeLateAtBkpts[b]` by setting invisible breakpoints at the original and new positions of the late statements. The debugger will discard the recorded information of a late statement `ls` when a `EndLatei` in `EndLatePoints[ls]` is encountered (`EndLatePoints[ls]` holds the end of the definition range of `ls`). Now consider the case when breakpoint `b` is reached, and the user requests the value of variable `v` defined by some statement `ls` in `MaybeLateAtBkpts[b]`. If the dynamically recorded information shows that the late statement `ls` was encountered, the debugger reports the current value of `v`. Otherwise if only the original position of the late statement was encountered, the debugger reports that the value of `v` is not reportable. The example in Figure 7.10, where the reportability of X along region ③ is path sensitive, is handled by this case. The `MaybeLateAtBkpts` and `EndLatePoints` tables are displayed in Figure 7.16.

The same strategy applies for each late statement `ds` in `PotFutBkptsLate[b]`, which prevents FULLDOC from setting invisible breakpoints too late. The `PotFutBkpts-Late` table for the example in Figure 7.10 is displayed in Figure 7.16.

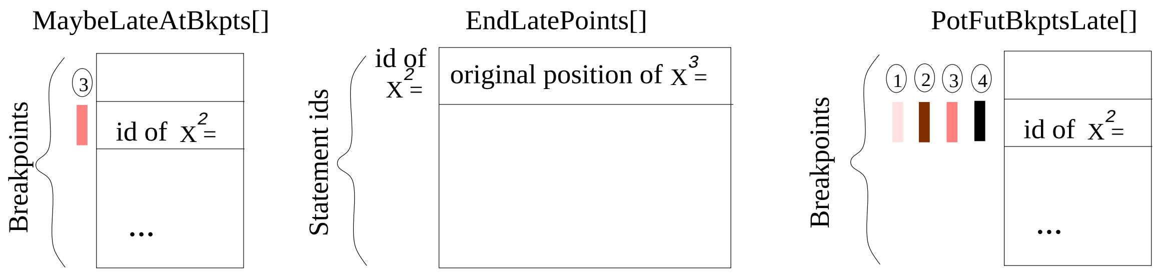


Figure 7.16: MaybeLateAtBkpts, EndLatePoints, and PotFutBkptsLate reportability debug information for example in Figure 7.10

7.4 Computing the reportability debug information

Mappings are used to compute the reportability debug information. The algorithm in Figure 7.17 gives an overview of how this debug information is computed. Lines $2-6$ determine what values are overwritten early and compute the `SaveDiscardPoints[] and EarlyAtBkpts[]` information. Lines $7-10$ determine what values are written late and compute the `StopPoints[] and LateAtBkpts[]`. Lines 11-13 determine the rest of the debug information by using data flow analysis. More details about the particular steps follow.

```
1  For each source definition $D_v$
2    If $D_v$ overwrites $x$ early then
3      Let discard1, discard2, ...   = the corresponding positions of original
       definitions of $x$ that are reachable from $ARHead(D_v)$ in the optimized code
4      SaveDiscardPoints $[D_v]$ = $(ARHead(D_v), \{$discard1, discard2,...$\}$ )
5      For each breakpoint $B$ along a path from $D_v$ to discard1, discard2,...,
6          EarlyAtBkpts[B] = EarlyAtBkpts[B]  $\cup \{ D_v \}$
7    Else If $D_v$ writes late in the optimized code then
8      StopPoints $[D_v]$ = $\{ARHead(D_v)\} \cup \{p : p$ is an earliest possible program
           point along paths from $ORHead(D_v)$ where $D_v$ will not execute$\}$
9      For each breakpoint $B$ along paths $ORHead(D_v)$ to $p \in$ StopPoints $[D_v]$ and/or
          the corresponding positions of original definitions of $x$ that are reachable
          from $ORHead(D_v)$ in the optimized code,
10        LateAtBkpts[B] = LateAtBkpts[B]  $\cup \{ D_v \}$
11 Compute AvailAtBkpts[,], NotRepDelAtBkpts[], and NotRepLateAtBkpts[]
          by comparing ranges using $ORHead(D_v)$ and $ARHead(D_v)$
12 Compute MaybeDelAtBkpts[] and MaybeLateAtBkpts[] by determining when
          deleted and late statements occur on one path and not another
13 Compute EndDelPoints[], EndLatePoints[], PotFutBkptsDel[], and
          PotFutBkptsLate[] by using reachability
```

Figure 7.17: Algorithm to compute the reportability debug information

7.4.1 Determining statements that overwrite early or write late.

To determine where values are overwritten early due to register reuse, suppose D_x is a definition of a variable x and the location of x is in register r in the optimized code. If D_x reaches an assignment to r in which r is reassigned to another variable or temporary, then x is overwritten early at the reassignment.

To determine where values of variables are overwritten early due to code hoisting optimizations, the original positions of the definitions and their actual positions in the optimized program are compared in G_{opt}. Let $ARHead(D_v)$ denote the actual position of a definition D_v and let $ORHead(D_v)$ denote the corresponding original position of D_v. The existence of a path P from $ARHead(D_v)$ to $ORHead(D_v)$ such that

P does not include backedges of loops enclosing both $ARHead(D_v)$ and $ORHead(D_v)$ is determined. The backedge restriction on P ensures that only the positions of the same instance of D_v before and after optimization are considered. This restricted notion of a path is captured by the *SimplePath* predicate as given by Definition 5.1. If $SimplePath(ARHead(D_v), ORHead(D_v))$ is true and the location of v at the program point before $ARHead(D_v)$ is the same location that is used to hold the value of D_v, then v is overwritten early at D_v in the optimized code. For example, in Figure 7.8, $SimplePath(ARHead(X^2), ORHead(X^2))$ is true, and thus, X is overwritten early at X^2.

To determine where values of variables are written late in the optimized program, the original positions of the definitions and their actual positions in the optimized program are similarly compared using G_{opt}. That is, for a definition D_v, the existence of a path P from $ORHead(D_v)$ to $ARHead(D_v)$ such that P does not include backedges enclosing both points is determined. Thus, if $SimplePath(ORHead(D_v), ARHead(D_v))$ is true, then definition D_v is written late in the optimized code. For example, in Figure 7.10, X is written late at X^2 because $SimplePath(ORHead(X^2), ARHead(X^2))$ is true.

7.4.2 Computing `SaveDiscardPoints[]` and `EarlyAtBkpts[]`.

If a value of x is overwritten early at D_v in the optimized code, then a save point is associated at the position of D_v in the optimized code, and discard points are associated at the corresponding positions of original definitions of x that are reachable from D_v in the optimized code. Reachable original definitions, which is similar to the reachable definitions problem, is determined by solving the following data flow equation:

$$ReachableOrigDefs(B) = \bigcup_{N \in succ(B)} Gen_{rod}(N) \cup (ReachableOrigDefs(N) - Kill_{rod}(N))$$

where

$$Gen_{rod}(B) = \{D_v : ORHead(D_V) = B\} \text{ and}$$
$$Kill_{rod}(B) = \{D_v : ORHead(D'_v) = B \wedge D'_v \text{ is a definition of v}\}.$$

For example, in Figure 7.8, the original definitions reachable from X^2 in the optimized code are X^2 and X^3 because $ReachableOrigDefs(ARHead(X^2)) = \{X^2, X^3\}$. Therefore, for X^2, a save point is associated at $ARHead(X^2)$, discard points are associated at $ORHead(X^2)$ and $ORHead(X^3)$, and `SaveDiscardPoints[`X^2`]` $= (ARHead(X^2), \{ORHead(X^2), ORHead(X^3)\})$.

After the save and discard points of D_v are computed, the breakpoints where reportability is affected by D_v are determined. $D_v \in$ `EarlyAtBkpts[b]` if b lies along paths from save to corresponding discard points of D_v. `EarlyAtBkpts[]` is easily computed by solving the following data flow equation on G_{opt}:

$$EarlyAt(B) = \bigcup_{N \in pred(B)} Gen_{ea}(N) \cup (EarlyAt(N) - Kill_{ea}(N))$$

where

$Gen_{ea}(B) = \{D_v : D_v$ overwrites early and a save point of D_v is at $B\}$ and
$Kill_{ea}(B) = \{D_v : D_v$ overwrites early and a discard point of D_v is at $B\}$.

Then $D_v \in$ `EarlyAtBkpts[B]` if $D_v \in EarlyAt(B)$. For example, in Figure 7.8, for a breakpoint b along regions ①, ②, and ③, `EarlyAtBkpts[`b`]` $= \{X^2\}$.

7.4.3 Computing `StopPoints[]` and `LateAtBkpts[]`.

For a definition D_v of a variable x that is written late, `StopPoints` $[D_v]$ are the earliest points at which execution can stop because either (1) the late value is computed or (2) a point is reached such that it is known the value will not be computed in the execution. A stop point of D_v is associated at the $ARHead(D_v)$. Stop points are also associated with the earliest points along paths from $ORHead(D_v)$ where the appropriate instance of D_v does not execute. That is, $p \in StopPoint(D_v)$ if

$$p = ARHead(D_v) \vee \tag{7.1}$$

$$(D_v \notin ReachableLate(p) \wedge \tag{7.2}$$

$$\nexists\, p'(SimplePath(p', p) \wedge p' \in StopPoint(D_v))). \tag{7.3}$$

Condition 1 ensures a stop point is placed at D_v. Condition 2 ensures the rest of the stop points are not placed at program points where the appropriate instance of the late statement would execute. Condition 3 ensures stop points are placed at the earliest points. $ReachableLate(p)$ is the set of statements written late that are reachable at p. *Reachable-Late*() is easily computed by solving the following data flow equation on G_{opt}:

$$ReachableLate(B) = \bigcup_{N \in succ(B)} Gen_{rl}(N) \cup (ReachableLate(N) - Kill_{rl}(N))$$

where

$Gen_{rl}(B) = \{D_v : ARHead(D_v) = B\}$ and
$Kill_{rl}(B) = \{D_v : ORHead(D_v) = B\}$.

Consider the example in Figure 7.10. `StopPoints` $[X^2] = \{ARHead(X^2)$, program point at the beginning of the false path$\}$. Notice that condition 2 ensures that a stop point is not placed along region ①.

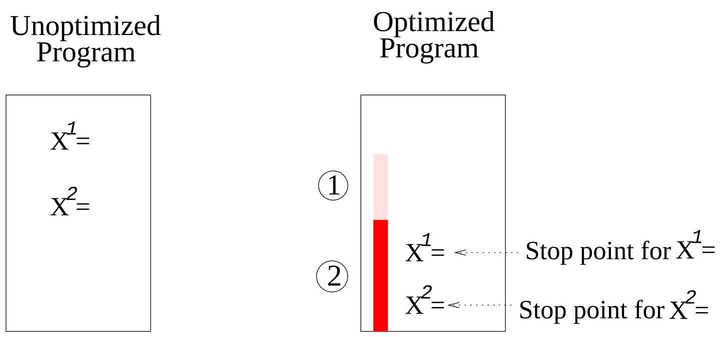


Figure 7.18: Overwritten late example

After the stop points of D_v are computed, the breakpoints where reportability is affected by D_v (because D_v may be overwritten late and may still execute) are determined. $D_v \in$ `LateAtBkpts[b]` if b lies along paths from $ORHead(D_v)$ to the stop points of D_v and the corresponding positions of original definitions of x that are reachable from D_v in the optimized code, except for the paths between the stop points and the positions of original definitions of x. The paths between the stop points and the positions of original definitions of x are excluded because it is possible that definition D_v is sunk past the position of an original definition of x and thus D_v should not be considered late with respect to breakpoints along such paths. For example, in Figure 7.18, although the stop point of X^1 is at $ARHead(X^1)$, `LateAtBkpts[b]` $= \{X^1\}$ for a breakpoint b along region ① and `LateAtBkpts[b]` $= \{X^2\}$ for a breakpoint b along region ②.

`LateAtBkpts[b]` is easily computed by solving the following data flow equation on G_{opt}:

$$LateAt(B) = \bigcup_{N \in pred(B)} Gen_{la}(N) \cup (LateAt(N) - Kill_{la}(N))$$

where

$$Gen_{la}(B) = \{D_v : ORHead(D_V) = B\} \text{ and}$$
$$Kill_{la}(B) = \{D_v : B \in StopPoints(D_v) \ \vee$$
$$(ORHead(D'_v) = B \ \wedge \ D'_v \text{ is a definition of v})\}.$$

Then $D_v \in$ `LateAtBkpts[B]` if $D_v \in LateAt(B)$. For example, in Figure 7.10, for a breakpoint b along region ①, `LateAtBkpts[`b`]` $= \{X^2\}$.

7.4.4 Computing `AvailAtBkpts[,]`.

The mappings are used to construct program ranges of a variable's value which correspond to the unoptimized code (*real* range) and the optimized code (*actual* range). By comparing the two ranges, the program ranges in the optimized code corresponding to regions where the value of the variable is always available for reporting are identified. If breakpoint B is in this program range for a variable v then `AvailAtBkpts[B,v]` is computed by performing data flow analysis to propagate the locations (memory and registers) of variables within these program ranges.

7.4.5 Computing `NotRepDelAtBkpts[]` and `NotRepLateAtBkpts[]`.

To determine the values of variables that are not reportable along a breakpoint because of the application of dead code elimination, the deleted statements are propagated where reportability is affected (regardless of the execution path taken) through the optimized control flow graph G_{opt} by solving the data flow equation:

$$NonRepDel(B) = \bigcap_{N \in pred(B)} Gen_{nrd}(N) \cup (NonRepDel(N) - Kill_{nrd}(N))$$

where

$Gen_{nrd}(B) = \{D_v : ORHead(D_v) = B \wedge D_v \text{ is deleted}\}$ and
$Kill_{nrd}(B) = \{D_v : ORHead(D'_v) = B \wedge D'_v \text{ is a definition of v}\}$.

Then for each breakpoint B, $v \in$ `NotRepDelAtBkpts[B]` if $\exists\ D_v$ such that $D_v \in NonRepDel(B)$. For example, in Figure 7.12, for a breakpoint B along region ①, $Y \in$ `NotRepDelAtBkpts`$[B]$.

`NotRepLateAtBkpts[]` is similarly computed by solving the following data flow equation on G_{opt}:

$$NonRepLate(B) = \bigcap_{N \in pred(B)} Gen_{nrl}(N) \cup (NonRepLate(N) - Kill_{nrl}(N))$$

where

$Gen_{nrl}(B) = \{D_v : ORHead(D_v) = B \wedge D_v \text{ is overwritten late}\}$ and
$Kill_{nrl}(B) = \{D_v : ORHead(D'_v) = B \wedge D'_v \text{ is a definition of v}\}$.

Then for each breakpoint B, $v \in$ `NotRepLateAtBkpts[B]` if $\exists\ D_v$ such that $D_v \in NonRepLate(B)$ and $D_v \notin LateAtBkpts(B)$. For example, in Figure 7.10, for a breakpoint B along region ②, $X \in$ `NotRepLateAtBkpts`$[B]$ because $X^2 \in NonRepLate(B)$ and $X^2 \notin LateAtBkpts(B)$.

7.4.6 Computing `MaybeDelAtBkpts[] and MaybeLateAtBkpts[]`.

To determine the values of variables that may not be reportable along a path when deleted, the data flow equation on G_{opt} is first computed:

$$MaybeDel(B) = \bigcup_{N \in pred(B)} Gen_{md}(N) \cup (MaybeDel(N) - Kill_{md}(N))$$

where

$Gen_{md}(B) = \{D_v :\ ORHead(D_v) = B \wedge D_v$ is deleted$\}$ and

$Kill_{md}(B) = \{D_v :\ ORHead(D'_v) = B \wedge D'_v$ is a definition of v$\}$.

Then $D_v \in$ `MaybeDelAtBkpts[B]` if $\exists\ D_v$ such that $D_v \in MaybeDel(B) \wedge D_v \notin NonRepDel(B)$. For example, in Figure 7.12, for a breakpoint B along region ②, $Y^2 \in$ `MaybeDelAtBkpts`$[B]$ because $Y^2 \in MaybeDel(B) \wedge Y^2 \notin NonRepDel(B)$.

`MaybeLateAtBkpts[]` is similarly computed by solving the following data flow equation on G_{opt}:

$$MaybeLate(B) = \bigcup_{N \in pred(B)} Gen_{ml}(N) \cup (MaybeLate(N) - Kill_{ml}(N))$$

where

$Gen_{ml}(B) = \{D_v :\ ORHead(D_v) = B \wedge D_v$ is overwritten late$\}$ and

$Kill_{ml}(B) = \{D_v :\ ORHead(D'_v) = B \wedge D'_v$ is a definition of v$\}$.

Then $D_v \in$ `MaybeLateAtBkpts[B]` if $\exists\ D_v$ such that $D_v \in MaybeLate(B) \wedge D_v \notin NonRepLate(B) \wedge D_v \notin LateAtBkpts(B)$. For example, in Figure 7.10, for a breakpoint B along region ③, $X \in$ `MaybeLateAtBkpts`$[B]$.

7.4.7 Computing `EndDelPoints[]` and `EndLatePoints[]`.

For each variable v of a deleted statement `ds` $\in$ `MaybeDelAtBkpts[]`, `EndDelPoints[ds]` are the corresponding positions of original definitions of v that are reachable from $ORHead($`ds`$)$ in G_{opt}. For example, in Figure 7.12, `EndDelPoints[Y]` = the original position of Y^3, which is $ORHead(Y^3)$. Similarly, for a variable v of a late statement `ls` $\in$ `MaybeLateAtBkpts[]`, `EndLatePoints[ls]` are the corresponding positions of original definitions of v that are reachable from $ORHead($`ls`$)$.

7.4.8 Computing `PotFutBkptsDel[]` and `PotFutBkptsLate[]`.

For each deleted statement D_v in `MaybeDelAtBkpts[]`, $D_v \in$ `PotFutBkptsDel[b]` if b lies along paths from the $ORHead(D_v)$ to the corresponding positions of original definitions of v that are reachable from $ORHead(D_v)$ in the optimized code. `PotFutBkptsDel[]` is easily computed by solving the following data flow equation on G_{opt}:

$$PotBkptsDel(B) = \bigcup_{N \in pred(B)} Gen_{pbd}(N) \cup (PotBkptsDel(N) - Kill_{pbd}(N))$$

where

$Gen_{pbd}(B) = \{D_v : ORHead(D_v) = B \wedge D_v$ is deleted$\}$ and
$Kill_{pbd}(B) = \{D_v : ORHead(D'_v) = B \wedge D'_v$ is a definition of v$\}$.

Then $D_v \in$ `PotFutBkptsDel[B]` if $D_v \in PotBkptsDel(N)$. For example, in Figure 7.12, for a breakpoint B along regions ①and ②, $Y^2 \in$ `PotFutBkptsDel(B)`.

`PotFutBkptsLate[]` is similarly computed. For each statement D_v that is overwritten late in `MaybeLateAtBkpts[]`, $D_v \in$ `PotFutBkptsLate[b]` if b lies along paths from the $ORHead(D_v)$ to the corresponding positions of original definitions of v that are reachable from $ORHead(D_v)$ in the optimized code. `PotFutBkptsLate[]` is easily computed by solving the following data flow equation on G_{opt}:

$$PotBkptsLate(B) = \bigcup_{N \in pred(B)} Gen_{pbl}(N) \cup (PotBkptsLate(N) - Kill_{pbl}(N))$$

where

$Gen_{pbl}(B) = \{D_v : ORHead(D_v) = B \wedge D_v$ is overwritten late$\}$ and
$Kill_{pbl}(B) = \{D_v : ORHead(D'_v) = B \wedge D'_v$ is a definition of v$\}$.

Then $D_v \in$ `PotFutBkptsLate[B]` if $D_v \in PotBkptsLate(N)$. For example, in Figure 7.10, for a breakpoint B along regions ①, ②, ③, and ④, $X^2, \in$ `PotFutBkptsLate(B)`.

7.5 Supporting loop transformations and inlining

With loop transformations and inlining, loops that are transformed and functions that are inlined are prematurely executed. That is, the debugger rolls ahead the execution of the optimized program. The statement instances that are prematurely executed are saved in the order of their corresponding instances in the unoptimized program. The values overwritten by the roll ahead execution are also saved so that they can be reported at subsequent breakpoints. The reportability debug information is extended to indicate the (1) program ranges in the optimized code where loops have been transformed or functions have been inlined, (2) start and stop points of execution for the roll forwarding, and (3) program ranges of expected values of variables that are not reportable because they are not computed in the transformed loop or the inlined code (but should be with respect to the unoptimized program). For example, in Figure 7.19, the loop in the optimized code has been reversed. Suppose a breakpoint is placed by a user at the beginning of the loop in the unoptimized code and at each instance of the breakpoint, the user requests the value of j. The debugger will roll forward the execution of the optimized program until the entire loop executes. As values of source variables are computed, the debugger saves the values in the order they would be computed in the unoptimized code. Then the debugger returns control to the user. At each instance of breakpoint 1, the debugger reports to the user the expected value of j. At breakpoint 2, the debugger reports that the expected of value of j (i.e., 11) is not computed in the optimized program execution.

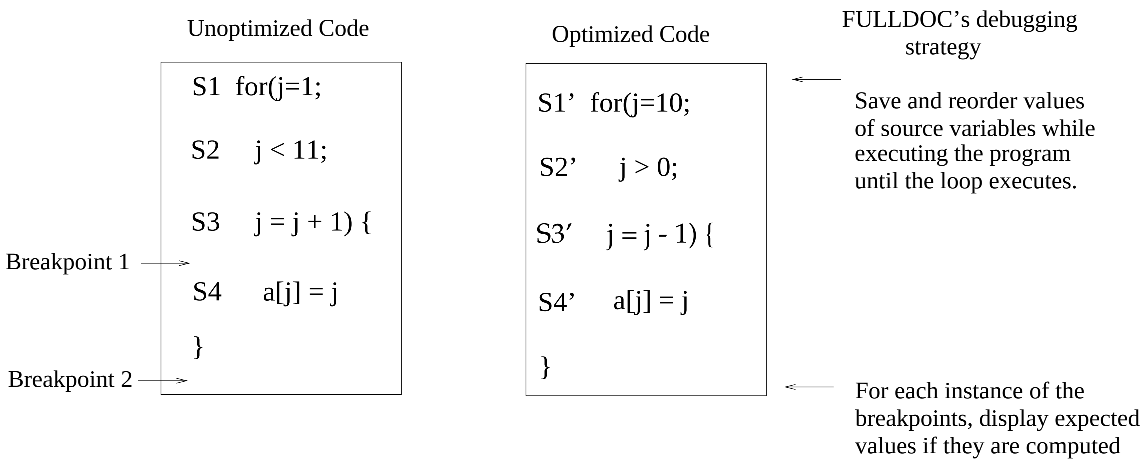

Unoptimized Code
S1 for(j=1;
S2 j < 11;
S3 j = j + 1) {
S4 a[j] = j
}
Breakpoint 1
Breakpoint 2
Optimized Code
S1' for(j=10;
S2' j > 0;
S3' j = j - 1) {
S4' a[j] = j
}
FULLDOC's debugging strategy
Save and reorder values of source variables while executing the program until the loop executes.
For each instance of the breakpoints, display expected values if they are computed

Figure 7.19: Loop reversal example

7.6 Implementation and experiments

FULLDOC was implemented by first extending `lcc` [23], a compiler for C programs, with a set of optimizations, including (coloring) register allocation, loop invariant code motion, dead code elimination, partial dead code elimination, partial redundancy elimination, copy propagation, and constant propagation and folding. `Lcc` was also extended to perform the analyses needed to provide the debug information to FULLDOC, given in the previous section. FULLDOC was then implemented, using the debug information generated by `lcc`, and fast breakpoints [32] for the implementation of invisible breakpoints.

Experiments were performed to measure the improvement in the reportability of expected values for a suite of programs, namely YACC and some SPEC95 benchmarks. For the purpose of evaluation, a user breakpoint was placed at every source statement, and the improvement in reportability of FULLDOC over a technique that uses only static information was determined. Also, for each breakpoint, the reasons why reportability is affected is reported, and thus the improvement of FULLDOC's technique over techniques that cannot report overwritten values or path sensitive values can be compared.

Table 7.1 shows for each benchmark, the percentage of values that could not be reported by (1) using only statically computed information and (2) FULLDOC. The first row gives the percentages of values that were deleted along all paths, and are thus not reportable in FULLDOC (as noted, FULLDOC could recover some of these values, as other debuggers can [27]). The next two rows give the percentages of values whose reportability is affected because they are overwritten early, either because of code hoisting (row 2) or a register being overwritten early (row 3). If a debugger does not include some mechanism for "saving" values overwritten early, it would not be able to report these values. The next three rows give the percentages of values whose reportability is affected because the statements that computed the values were affected by partial dead code elimination. Row 4 indicates the percentages of values that are not reportable along paths before the sunk values. Row 5 indicates the percentages of values that are not reportable along paths where the sunk values are never computed. Row 6 indicates the percentages of values that are not reportable along paths because the reportability of the values sunk is path sensitive. If a debugger does not include some mechanism to "roll ahead" the execution of the optimized program, it would not be able to report these values. The next two rows give the results when reportability is affected by path sensitive information. The seventh row gives the percentages that were not reportable for path sensitive deletes. In this case, the values may have been deleted on paths that were executed. The eighth row gives the results when the location of a value is path sensitive. A technique that does not include path sensitive information would fail to report these values. The last row gives the total percentages that could not be reported. On average, FULLDOC cannot report 8% of the local variables at a source breakpoint while a debugger using only static information cannot report 30%.

Table 7.1: Percentage of local variables per breakpoint that are not reportable

Problems	yacc		compress		go		m88ksim		ijpeg	
	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC	static info	FULL DOC
deleted-all paths	0.96	0.96	15.03	15.03	0.75	0.75	1.87	1.87	10.42	10.42
code hoisting	0.19	0.00	0.34	0.00	0.30	0.00	0.14	0.00	4.15	0.00
reg overwrite	42.65	0.00	17.24	0.00	9.44	0.00	1.83	0.00	15.87	0.00
code sinking (rf)	0.19	0.00	0.64	0.09	1.40	0.39	0.57	0.07	1.79	0.09
del on path	0.00	0.00	0.02	0.02	0.10	0.10	0.06	0.06	0.28	0.28
path sens late	0.00	0.00	0.18	0.09	0.51	0.18	0.41	0.37	0.58	0.39
path sens delete	8.27	6.07	0.18	0.00	2.25	0.74	0.00	0.00	2.36	1.20
path sens location	3.95	0.00	0.07	0.00	1.14	0.00	0.32	0.00	1.43	0.00
total	56.21	7.03	33.70	15.23	15.89	2.16	5.20	2.37	36.88	12.38

Figure 7.20 shows for each benchmark, the percentage of values that could not be reported by (1) using only statically computed information, (2) the timestamping technique, (3) the emulation technique, and (4) FULLDOC. FULLDOC can report 31% more values than techniques using only statically computed information. FULLDOC can report at least 28% more values than the emulation technique [51] since neither path sensitivity nor register overwrites were handled. Finally, FULLDOC can report at least 26% more values than the dynamic currency determination technique [22] since early overwrites were not preserved and no roll ahead mechanism is employed.

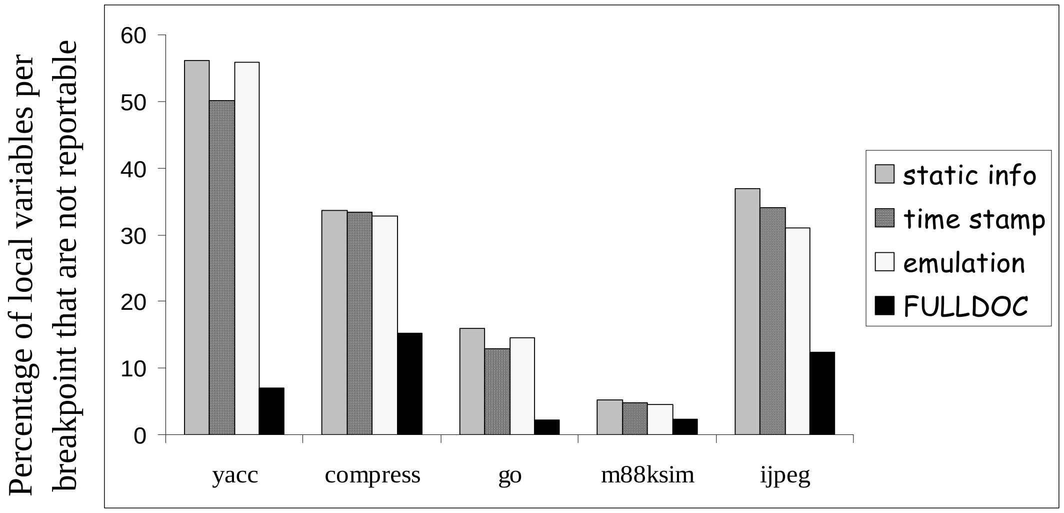


Figure 7.20: Expected values not reportable

Table 7.2 presents statistics from the static analysis for FULLDOC. The first two rows show the number of source statements and the percentage of source statements whose reportability is affected by optimizations. The next 6 rows give the number of entries in each of the tables generated for use at runtime. It should be noted that the largest table is for register overwrites. The last row shows that the increase in compilation for computing all the debug information averaged only 10.9%.

Table 7.3 shows the average number of invisible breakpoints per source code statement that was encountered during execution. These numbers are shown for each of the various types of invisible breakpoints. These numbers indicate that not much overhead is incurred at runtime for invisible breakpoints. The last three rows display the overhead imposed by the roll ahead execution of the optimized program. On average, 9.7% of the source assignment statements were executed during the roll aheads. The maximum number of statements executed during a roll forward ranges from 5 to 4102 values, which means at most 5 to 4102 number of values are saved from the roll ahead at any given moment. The average roll ahead of source assignment statements ranges from 2 to 7 statements. The size

Table 7.2: Static statistics

		yacc	compress	go	m88ksim	ijpeg
no. source statements		168	354	10876	5778	8214
% statements affected		85	57	59	52	56
number of table entries	code hoisting	10	77	1502	987	2374
	reg overwrite	517	234	11819	3961	9655
	code sinking (rf)	13	177	5355	1839	3745
	path sens late	0	117	2912	1203	1833
	path sens delete	66	37	1785	397	1452
	path sens location	48	59	1937	301	1447
% increase compile time		12.1	8.8	11.0	9.6	13.1

of the value pool holding values that are overwritten early was small with the maximum size ranging from 8 entries to 77 entries, indicating that optimizations are not moving code very far.

Table 7.3: Runtime statistics

		yacc	compress	go	m88ksim	ijpeg
% breakpoints where reportability affected		94	95	67	21	92
avg. no. invisible breakpoints per source statement	code hoisting	0.12	0.03	0.04	0.05	0.35
	reg overwrite	1.03	0.13	0.26	0.02	0.35
	code sinking (rf)	0.03	0.03	0.07	0.03	0.12
	path sens late	0.10	0.05	0.13	0.04	0.23
	path sens delete	0.09	0.00	0.03	0.01	0.23
	path sens location	0.07	0.02	0.02	0.00	0.05
	overall	1.44	0.26	0.56	0.18	1.37
	(duplicates removed) overall	0.56	0.14	0.37	0.17	0.43
% source assignments executed for roll forwards		1.33	4.11	17.39	6.01	19.8
maximum roll forward length		5	60	314	4102	1482
average roll forward length		2	4	7	5	4

Thus, the experiments show that the table sizes required to hold the debug information and the increase in compile time to compute debug information are both quite modest. The runtime cost of FULLDOC's technique, which is a maximum of less than one fast breakpoint per source level statement if all possible values are requested by the user at all possible breakpoints, is also reasonable. The payoff of FULLDOC's technique is substantial since it reports at least 26% more values than the best previously known techniques.

The presence of pointer assignments in a source program can increase FULLDOC's overheads because the strategies rely on determining the ranges in which the reportability of variables are affected. For control equivalent code motion (assignments are not introduced into new paths nor removed from paths), the ranges in which reportability of values are affected even in the presence of pointer assignments can be statically determined. For the case when the reportability of a value of a variable is affected and the end of its reportable range is possibly at a pointer assignment (because of code deletion and non-control equivalent code motion), FULLDOC's strategy has to dynamically track the range in which the reportability of the value of the variable is affected.

7.7 Summary

This chapter presents FULLDOC, a **FULL** reporting **D**ebugger of **O**ptimized **C**ode that reports all expected values that are computed in the optimized program. That is, every value of a source variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. Experimental results show that FULLDOC can report 31% more values than techniques relying on static information and at least 26% more over existing techniques that limit the dynamic information used. FULLDOC's improvement over existing techniques is achieved by statically computing information to guide the gathering of dynamic information that enables full reporting. The only values that FULLDOC cannot reported are those that are not computed in the optimized program execution, either because they are deleted along all paths or a path that is executed.

Chapter 8
Conclusion and future work

Compilers apply optimizations to improve the performance of programs, but their application creates challenges in debugging the optimized code. Debugging optimized code is necessary because almost all production compilers apply optimizations to achieve high performance, and current trends in processor design increasingly rely on compiler optimizations to achieve high performance. In many cases, only the optimized version of a program can execute or execute within a reasonable amount of time. Finally, if application programmers intend to ship optimized code, then the optimized code must be debugged. Otherwise, errors may be masked in the version of the program that is instead debugged.

Most prior work on debugging optimized code focused on the development of source level debuggers for optimized code. However, the problem of debugging optimized code is twofold because errors in an optimized program can be caused by errors in the original source program or introduced by the optimizer. The optimizer may apply an unsafe transformation or an error may exist in the implementation of an optimization. Therefore, source level debugging techniques must be developed to help application programmers debug optimized code from the point of view of the source program and to help optimizer writers debug optimizers, which are becoming more complex pieces of software and tedious to debug. Moreover, source level debugging techniques that are developed should (1) not modify the optimized code, except for setting breakpoints, (2) be transparent to the user, and (3) support more aggressive optimizations.

8.1 Summary of contributions

Developing source level tools to debug optimized code is hampered by the difficulties in establishing the correspondence between the source and optimized code. This dissertation has analyzed the effects of optimizations and the complexities in maintaining a correspondence between the unoptimized and optimized code. The scope of this research covers a variety of code transformations, including statement level optimizations, loop transformations, and inlining. Statement level optimizations include speculative code motion and path sensitive optimizations. Optimizations such as dynamic memory management optimizations are not considered. A mapping technique [29] was developed for tracking the correspondences between the unoptimized and optimized code statements while code transformations are performed. The mappings capture the impact that optimizations have on statements and their instances and thus are useful for a wide range of optimizations.

This dissertation has explored the use of dynamic information to develop source level debugging techniques that help optimizer writers and application programmers debug optimized code. The mappings developed in this dissertation have been used to develop two complementary source level debugging tools for optimized code, which have been implemented and experimentally evaluated. These techniques can support more aggressive optimizations, including speculative code motion, path sensitive optimizations, and loop transformations, than previously developed techniques.

The first technique developed to help debug optimizers, called comparison checking [30], is an approach that compares values computed in both the unoptimized and optimized executions of a source program and detects semantic differences between the two versions. When a comparison fails, the earliest place where the failure occurred and the optimizations that are involved are reported. Thus, the optimizer writer can utilize this information to debug the optimizer, and the optimizer writer can have greater confidence in the correctness of the optimizer. Moreover, since the internal values computed in the optimized code are compared to that of the unoptimized program, a finer level of testing is provided which can find errors in the optimized code that do not cause the output of the program to be incorrect. The automation of the comparison checking scheme relies on the mappings developed in this dissertation and annotations to guide the actions of the comparison checker. This technique does not restrict the set of optimizations applied and the optimized code is not modified, except for the setting of breakpoints. The comparison checking scheme was implemented and executes the unoptimized and optimized versions of C programs. Experimental results demonstrate the approach is effective and practical. In fact, this scheme proved very useful in debugging the optimizer that was implemented for this work.

The comparison checking technique can be modified to check different levels of optimizations. Just as optimizations are often phased, the checking can be performed in phases. For example, checking can be performed after loop optimizations are applied, after statement level optimizations are applied, and after low level optimizations are applied.

This phase checking can reduce the cost of checking as well as help optimizer writers debug the optimizations that were applied in the phase that is to be checked.

Furthermore, the comparison checking technique can be tailored to help optimizers writers debug and validate specific optimizations. This dissertation described how to tailor the comparison checker to global register allocation. The register allocation checker can detect errors in a register allocator implementation and determine the possible cause(s) of the errors. For example, the register allocation checker can inform the user when a stale value is used, a wrong register is used, and when a value is evicted from a register but not saved for future uses. The register allocation checker scheme compares values computed and used by both the unoptimized and optimized program executions and tracks and verifies information about the variables that are assigned to registers throughout the program execution. The register allocation checker can be incorporated into the comparison checker or can be used as a standalone tool.

The second technique, a full reporting source level debugger for optimized code called FULLDOC, is used by application programmers to find errors in source programs [31]. This debugger can provide more of the expected program state than previously developed source level debuggers for optimized code. That is, every value of a source variable that is computed in the optimized program execution is reportable at all breakpoints in the source code where the value of the variable should be reportable. FULLDOC's improvement over existing techniques is achieved by statically computing information to guide the gathering of dynamic information that enables full reporting. This technique is demonstrated in a compiler that performs a set of global statement level optimizations for C source programs. The technique does not restrict the set of optimizations applied and the optimized code is not modified, except for the setting of breakpoints. The techniques are transparent to the user. If a user inserts a breakpoint where the reportability of values is affected at the breakpoint or a potential future breakpoint, FULLDOC automatically inserts invisible breakpoints to gather dynamic information to report the expected values. Experimental results show that FULLDOC can report 31% more values than techniques relying on static information and at least 26% more over existing techniques that limit the dynamic information used. The only values that FULLDOC cannot reported are those that are not computed in the optimized program execution, either because they are deleted along all paths or a path that is executed.

8.2 Future work

There are a number of open interesting research problems. Although the mappings were used in this dissertation to develop a comparison checker and source level debugger for optimized code, the mappings can also benefit a number of applications. Other problems for future research include extending the debugging techniques to provide more debugging features and to support more aggressive optimizations and other programming languages.

1. The mappings developed in this dissertation can be used to develop tools that benefit a number of other applications.

   It is very difficult to understand or inspect optimized code in isolation of the unoptimized code. Even if the unoptimized code is available, it is still difficult, especially when there is no knowledge of what transformations were applied and/or someone else wrote the program. The mappings can be used to develop a tool that enables the understanding and inspection of optimized code by indicating which statement instances in the unoptimized code correspond to statement instances in the optimized code.

   The mappings can enhance programming environment tools. For example, interactive programming environment tools have been developed to assist users in parallelizing programs. These tools help users decide how to restructure programs by analyzing and performing transformations to detect and exploit parallelism. Integrating the mappings gives users a better understanding of the effects of the applied transformations by visually seeing the differences between both the new version and the original program version. This extra information can help users make better informed decisions and verify that their decisions are indeed correct.

   The mappings can be used to design other program development tools. For example, profile-based performance analysis tools can identify performance bottlenecks. However, the results should be conveyed to users in terms of the unoptimized program. Therefore, for the case of profiling optimized code, the mappings can be used to convey the results of profiled optimized code in terms of the unoptimized program.

2. Although this dissertation covers more aggressive optimizations than previously developed source level debugging techniques for optimized code, there is still the issue of optimization coverage. For example, as more production compilers produce predicated code [47, 11, 35] and/or code is dynamically optimized [15, 12], the need to develop source level debugging tools to support such optimizations has to be addressed. The

support of dynamic optimizations would require the dynamic updating of the mappings between the unoptimized and optimized code and the dynamic analysis of the mappings, unoptimized code, and optimized code. In terms of a source level debugger, the predicated code would require invisible breakpoints to capture the execution control flow of the optimized code.

3. The source level debugger developed in this dissertation does not support the modification of variables, nor does it effectively support asynchronous breakpoints and core files. Moreover, the debugger cannot report expected values that have been deleted from the optimized code. Future research would extend the source level debugger techniques to include such debugging features. The design of a source level debugger for optimized code that has the same debugging capabilities as for unoptimized code remains an open problem.

4. This dissertation focused on C source programs. It would be interesting to consider the impact of features of other programming languages such as Java in developing source level debugging tools for optimized code. Perhaps some of the problems encountered in the C language can be avoided in Java. For example, memory can be uninitialized in C but not in Java, and optimizations can reorder floating point operations in C but not in Java.

5. The comparison checker was tailored to help users find errors in the implementation of global register allocation. It would be interesting to see how to tailor the comparison checker to help users find errors in other optimizations.

6. The comparison checker could be embedded in a debugger. In this case, the response time of the checker must be appropriate for use in a debugger. Therefore, the number of comparisons would have to be reduced. The amount of checking can be limited by checking certain regions of the source level code, as specified by the user. This strategy is useful when it is unnecessary to check an entire program. For example, during testing, programs are typically executed under different inputs and checking the entire program under every input may be redundant and thus unnecessary. Alternatively, a region can be defined by the statements affected by the application of code transformations. This approach is optimization dependent. Analysis techniques would need to be designed to determine what values may be affected by the program changes.

   Another approach, which is optimization independent, is to check only those values that cannot be guaranteed to always be the same in both the unoptimized and optimized program executions. Static analysis techniques would need to be developed to analyze both versions of the program and determine what values are always the same under any execution and thus do not need to be checked. The values that cannot be guaranteed would be checked. Of course, a conservative approach would be taken. Only those checks where the analysis can guarantee the same value could be eliminated.

7. Techniques can be explored to effectively develop a comparison checking technique for parallelized code. This is a much harder problem than that of optimized code because the execution behavior of sequential code is now compared with that of parallelized code. The program executions cannot be easily orchestrated because the parallel version would be required to perform a serial execution. Extracting values from shared memory is another problem as values from memory must be extracted before the values are overwritten by any one of the processes. With distributed memory, the communication of values to the comparison checker may have to be optimized for practicality purposes.

8. With the need to gather dynamic information for both the source level debugger and comparison checker, more support from hardware should be provided to access the program state and to control the execution of the program. For example, more hardware assisted breakpoints should be provided. Also, perhaps a new mechanism should be developed to allow a debugger that does not execute within the same context as the debugged program to set breakpoints and extract the program state without going through the operating system.

9. Currently the debug information supplied by the optimizing compiler to the debugger has not been incorporated into existing debug formats such as stabs [33] and DWARF [1] debug formats. These debug formats would have to be extended.

10. Although the comparison checking technique developed in this dissertation is designed only for unoptimized and optimized versions of a program, the comparison checking technique can possibly be extended to handle different program versions. That is one version is derived form the other version either by programming edits or other programming tools such as source to source translators. Mappings would have to be developed to establish the relationship between both program versions.

Bibliography

Bibliography

[1] *DWARF Debugging Information Format.* Industry Review Draft, Unix International Programming Language Special Interest Group (SIG), 1993.

[2] Abramson, D. A., Foster, I., Michalakes, J., and Sosic, R. Relative Debugging and its Application to the Development of Large Numerical Models. In *Proceedings of IEEE Supercomputing 1995*, December 1995.

[3] Abramson, D. and Sosic, R. A Debugging Tool for Software Evolution. *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, pages 206–214, July 1995.

[4] Abramson, D. and Sosic, R. A Debugging and Testing Tool for Supporting Software Evolution. *Journal of Automated Software Engineering*, 3:369–390, 1996.

[5] Abramson, D., Foster, I, Michalakes, J., and Sosic, R. A New Methodology for Debugging Scientific Applications. *Communications of the ACM*, 39(11):69–77, November 1996.

[6] Abramson, D., Sosic. R., and Watson, R. Implementation Techniques for a Parallel Relative Debugger. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[7] Adl-Tabatabai, A. *Source-Level Debugging of Globally Optimized Code.* PhD dissertation, Carnegie Mellon University, 1996. Technical Report CMU-CS-96-133.

[8] Adl-Tabatabai, A. and Gross, T. Detection and Recovery of Endangered Variables Caused by Instruction Scheduling. In *Proceedings ACM SIGPLAN'93 Conf. on Programming Languages Design and Implementation*, pages 13–25, June 1993.

[9] Adl-Tabatabai, A. and Gross, T. Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging. In *Proceedings 20th POPL Conference*, pages 371–383, January 1993.

[10] Adl-Tabatabai, A. and Gross, T. Source-Level Debugging of Scalar Optimized Code. In *Proceedings ACM SIGPLAN'96 Conf. on Programming Languages Design and Implementation*, pages 33–43, May 1996.

[11] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of Control Dependence to Data Dependence. In *Proceedings 10th POPL Conference*, pages 177–189, January 1983.

[12] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings ACM SIGPLAN'2000 Conf. on Programming Languages Design and Implementation*, pages 1–12, June 2000.

[13] Boyd, M.R. and Whalley, D.B. Isolation and Analysis of Optimization Errors. In *Proceedings ACM SIGPLAN'93 Conf. on Programming Languages Design and Implementation*, pages 26–35, June 1993.

[14] Brooks, G., Hansen, G.J., and Simmons, S. A New Approach to Debugging Optimized Code. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 1–11, June 1992.

[15] Burke, M.G., Choi, J., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., and Whaley, J. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings Java'99*, pages 129–141, June 1999.

[16] Caron, J.M. and Darnell, P.A. Bugfind: A Tool for Debugging Optimizing Compilers. *Sigplan Notices*, 25(1):17–22, January 1990.

[17] Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.W. IMPACT: An architectural framework for multiple-instruction-issue processor. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[18] Copperman, M. *Debugging Optimized Code Without Being Misled.* PhD dissertation, University of California, Santa Cruz, 1993. Technical Report UCSC-CRL-93-21.

[19] Copperman, M. Debugging Optimized Code Without Being Misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, 1994.

[20] Copperman, M. and McDowell, C.E. Detecting Unexpected Data Values in Optimized Code. Technical Report 90-56, Board of Studies in Computer and Information Sciences, University of California at Santa Cruz, October 1990.

[21] Coutant, D.S., Meloy, S., and Ruscetta, M. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings ACM SIGPLAN'88 Conf. on Programming Languages Design and Implementation*, pages 125–134, June 1988.

[22] Dhamdhere, D.M. and Sankaranarayanan, K.V. Dynamic Currency Determination in Optimized Programs. *ACM Transactions on Programming Languages and Systems*, 20(6):1111–1130, November 1998.

[23] Fraser, C. and Hanson, D. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.

[24] Fritzson, P. A Systematic Approach to Advanced Debugging through Incremental Compilation. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 130–139, 1983.

[25] Gross, T. Bisection Debugging. In *Proceedings of the AADEBUG'97 Workshop*, pages 185–191, May, 1997.

[26] Gupta, R. Debugging Code Reorganized by a Trace Scheduling Compiler. *Structured Programming*, 11(3):141–150, 1990.

[27] Hennessy, J. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[28] Holzle, U., Chambers, C., and Ungar, D. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 32–43, June 1992.

[29] Jaramillo, C., Gupta, R., and Soffa, M.L. Capturing the Effects of Code Improving Transformations. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 118–123. Springer Verlag, October 1998.

[30] Jaramillo, C., Gupta, R., and Soffa, M.L. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering and European Software Engineering Conference*, pages 268–284. Springer Verlag, September 1999.

[31] Jaramillo, C., Gupta, R., and Soffa, M.L. FULLDOC: A Full Reporting Debugger for Optimized Code. In *7th International Static Analysis Symposium*, pages 240–259. Springer Verlag, June/July 2000.

[32] Kessler, P. Fast Breakpoints: Design and Implementation. In *Proceedings ACM SIGPLAN'90 Conf. on Programming Languages Design and Implementation*, pages 78–84, June 1990.

[33] Menapace, J., Kingdon, J., and MacKenzie, D. *The "stabs" Debug Format.* Free Software Foundation, Inc., Contributed by Cygnus Support, 1993.

[34] Necula, G. Translation Validation for an Optimizing Compiler. In *Proceedings ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pages 83–94, June 2000.

[35] Park, J.C.H., Schlansker, M.S. On Predicated Execution. Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[36] Pineo, P.P. *The High-Level Debugging of Parallelized Code Using Code Liberation.* PhD dissertation, University of Pittsburgh, April 1993. Technical Report 93-07.

[37] Pineo, P.P. and Soffa, M.L. Debugging Parallelized Code using Code Liberation Techniques. *Proceedings of ACM/ONR SIGPLAN Workshop on Parallel and Distributed Debugging*, 26(4):103–114, May 1991.

[38] Pineo, P.P. and Soffa, M.L. A Practical Approach to the Symbolic Debugging of Parallelized Code. *Proceedings of International Conference on Compiler Construction*, 26(12):357–373, April 1994.

[39] Pollock, L.L. and Soffa, M.L. High-Level Debugging with the Aid of an Incremental Optimizer. In *21st Annual Hawaii International Conference on System Sciences*, volume 2, pages 524–531, January 1988.

[40] Seidner, R. and Tindall, N. Interactive Debug Requirements. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 9–22, 1983.

[41] Sosic, R. Dynascope: A Tool for Program Directing. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 12–21, June 1992.

[42] Sosic, R. A Procedural Interface for Program Directing. *Software Practice and Experience*, 25(7):767–787, July 1995.

[43] Sosic, R. Design and Implementation of Dynascope, a Directing Platform for Compiled Programs. *Computing Systems*, 8(2):107–134, Spring 1995.

[44] Sosic, R. and Abramson, D. A. Guard: A Relative Debugger. *Software Practice and Experience*, 27(2):185–206, February 1997.

[45] Tice, C. *Non-Transparent Debugging of Optimized Code.* PhD dissertation, University of California, Berkeley, 1999. Technical Report UCB-CSD-99-1077.

[46] Tice, C. and Graham, S.L. OPTVIEW: A New Approach for Examining Optimized Code. *Proceedings of ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, June 1998.

[47] Towle, R.A. *Control and Data Dependence for Program Transformations.* PhD dissertation, University of Illinois, Urbana, IL, 1976.

[48] Warren, H.S. and Schlaeppi, H.P. Design of the FDS Interactive Debugging System. Technical Report RC7214, IBM Yorktown Heights, Yorktown Heights, N. Y., July 1978.

[49] Wismueller, R. Debugging of Globally Optimized Programs Using Data Flow Analysis. In *Proceedings ACM SIGPLAN'94 Conf. on Programming Languages Design and Implementation*, pages 278–289, June 1994.

[50] Wu, L. *Interactive Source-Level Debugging of Optimized Code.* PhD dissertation, University of Illinois, Urbana-Champaign, 2000.

[51] Wu, L., Mirani, R., Patil H., Olsen, B., and Hwu, W.W. A New Framework for Debugging Globally Optimized Code. In *Proceedings ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pages 181–191, May 1999.

[52] Zellweger, P.T. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.

[53] Zellweger, P.T. *Interactive Source-Level Debugging of Optimized Programs.* PhD dissertation, University of California, Berkeley, May 1984. Published as XEROX PARC Technical Report CSL-84-5.