INTERACTIVE SOURCE-LEVEL DEBUGGING OF OPTIMIZED CODE

BY

LE-CHUN WU

B.S., National Taiwan University, 1989
M.S., National Taiwan University, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

INTERACTIVE SOURCE-LEVEL DEBUGGING OF OPTIMIZED CODE

Le-Chun Wu, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2000
Wen-mei W. Hwu, Advisor

With an increasing number of executable binaries generated by optimizing compilers today to fully utilize advanced architecture features, it has become a necessity to support debugging optimized code. One of the most difficult problems in debugging globally optimized code is to recover the expected variable values at source breakpoints. To solve this problem, the debugger not only has to stop the execution at appropriate places to preserve necessary program state, but also needs to be able to correctly associate storage locations with source variables.

In this dissertation, a new framework for debugging globally optimized code is proposed. This framework consists of a novel breakpoint implementation scheme and a new data location tracking mechanism. In the proposed breakpoint implementation scheme, the debugger takes over the control of execution early and executes instructions under a new forward recovery model. This enables the debugger to recover the expected behavior of a program even in the presence of optimization. Also the source breakpoints are reported to the user in the order specified by the original source program and the behavior of exceptions meets what the user expects.

The new data location tracking scheme keeps track of variable definition information during optimization. A data-flow analysis based on the definition information preserved is performed to collect data that is then used to generate the run-time data location

information. With this data location information, a debugger incorporating the proposed breakpoint implementation scheme can determine if the expected value of a variable is available at a source breakpoint and how to recover it.

The debugging framework has been prototyped in both the IMPACT compiler and an experimental debugger. Experiments conducted on several integer benchmark programs have yielded encouraging results. The overhead in executable file size and compile time incurred by this framework is reasonable. Compared with previous work, the proposed approach is much more effective in the recovery of the expected variable values.

DEDICATION

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Wen-mei Hwu, for providing an excellent environment in which to learn and carry out research, for his insight, guidance and support during my studies, and for the opportunities he has provided. My future career will benefit greatly from the lessons I have learned.

Next, I would like to extend my gratitude to the other members of my dissertation committee, Professor David Padua, Professor Constantine Polychronopoulos, and Professor Andrew Chien. Their comments, questions, and suggestions improved the quality of this work immensely. I would also like to thank Professor Jane Liu, for her guidance and support during the first two years of my graduate studies here.

This research would not have been possible without the support of the members of the IMPACT research group, both past and present. The group members were always willing to provide assistance, including research discussions, practice talks, and software enhancements. Special thanks to Ben-Chung Cheng, John Gyllenhaal, David August, and Brian Deitrich for answering numerous IMPACT compilation questions and providing bug fixes over the years. Many thanks to Sabrina Hwu for creating such an enjoyable work atmosphere in the group.

Also, thanks to Kuo-Feng Ssu, Hewijin Jiau, Liang-chuan Hsu, Tai-Yi Huang, Li-Pen Yuan, Yi-Kan Cheng, Fu-Chiarng Chen, Chien-Wei Li, Ben-Chung Cheng, and the members in my volleyball team for their steadfast friendship throughout my graduate

studies here, and for providing much needed mental breaks. They made my life in Champaign/Urbana a lot easier and much more enjoyable.

I would like to thank my parents, Chi Wu and Lien Hung, for their love and encouragement throughout my life. They provided a firm foundation for me at home and in my education, and have always offered assistance when I needed it. I would like to thank my late grandmother for her love and for taking such good care of me during my childhood. I would also like to thank my sister, Yueh-Yun Wu, and my brother, Le-Shin Wu, for their love and support. They did much more than their share in taking care of our grandmother and parents back home while I was away for so long. I owe them a lot.

Finally, I would like to thank my wife, I-Wen Dzou, for her love, patience, caring, and companionship. She has been a constant source of joy in my life and has helped me through the difficult times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A source-level interactive debugger is a useful software tool that helps a programmer control, examine, and monitor a running program interactively at the source level. Traditionally, the standard software development paradigm has been to compile the program with little or no optimization during the debugging phase. When the program is to be shipped to the users, it is sometimes compiled with optimizations enabled. However, with the compiler optimizations becoming increasingly critical for today's high-performance computer systems such as *EPIC* architecture [1] and an increasing number of executable binaries generated by optimizing compilers, the traditional paradigm has become inconvenient or even unacceptable for several reasons [2], [3], [4]:

1. Optimization has become a default process or an integral part in modern compilers. Some transformations, such as register allocation which appears to be a kind of optimization to the debugger, may still be performed by the compiler even if the optimization option has been turned off.

2. Programs may be too large to run on the target machine (especially in the embedded systems) without memory space optimizations such as tail merging, memory location (or stack frame slot) reuse, etc.

3. Without optimization, programs might run several orders of magnitude slower at debug time, which in turn makes the debugging process very ineffective.

4. Some bugs might only appear in optimized code, even when the compiler optimization modules are correct. Examples illustrating this kind of bugs due to differences in memory layout between optimized and unoptimized code can be found in Reference [5].

5. Due to the need to reverify the software after recompilation, many software vendors would like to be able to debug the shipped version of the programs.

Therefore, it has become a necessity to provide a clear and correct source-level debugger for programmers to debug optimized code.

However, debugging optimized code is difficult. There are two primary aspects associated with code optimization that make debugging difficult [2]. First, it complicates the mapping between the source code and the object code due to code duplication, elimination, and reordering. This problem is called *code location problem*. Second, it makes reporting values of source variables either inconsistent with what the user expects or simply impossible. Because of code reordering and deletion, assignments to user variables might take place earlier or later than expected. Also register allocation algorithms which reuse registers or memory locations may make the run-time locations of variables varying or non-existent at different points of execution. This problem is referred to as *data value problem*.

In general, there are two ways for an optimized code debugger to present meaningful information about the debugged program [2]. The debugger provides *expected behavior* of the program if it hides the optimization from the user and presents the program behavior consistent with what the user expects from the source code. It provides *truthful behavior* if it makes the user aware of the effects of optimizations and warns of surprising outcomes when the expected answers to the debugging queries cannot be provided. Although it is not always possible to recover the program behavior to what the user expects without constraining the optimization performed or inserting some instrumentation code [3], it is desirable for the user to see as much expected program behavior as possible. Therefore, in this dissertation I propose a new debugging framework designed to recover expected behavior, whenever possible, which addresses both code location and data value problems.

As one of the most frequently used functionalities of a source-level debugger is for the user to set breakpoints and examine variables' values at these points, the proposed debugging framework focuses on how to support these activities while debugging optimized code. The framework consists of a novel breakpoint implementation scheme and a new data location tracking scheme [6], [7]. The breakpoint implementation scheme includes a new code location mapping mechanism and a new run-time debugger strategy. Under this breakpoint implementation scheme, the program state can be properly preserved at debug time and source breakpoints and program exceptions are reported to the user in a manner consistent with what the user expects. With the preserved program state and the information generated by the data location tracking scheme, the debugger in the proposed framework can determine if the expected value of a variable is available

at a source breakpoint and how to recover it. Although the goal of this new debugging framework is to make optimization effects as transparent to the user as possible, however, since the proposed approach is *non-invasive* (that is, the debugged program is not modified or inserted additional instructions by the compiler for the purpose of debugging support [3]), the transparent debugging simply can not always be achieved. In my proposed framework, whenever the expected behavior can not be recovered, the truthful behavior will be presented so that the user will not be confused or misled.

The debugging framework was originally motivated by the observation that in order for the debugger to provide expected variable values, the program states changed by the out-of-original-source-order instructions have to be tracked by the debugger. To do this for a breakpoint at source statement S, the debugger suspends execution before executing any of the instructions that are expected to happen after S. The object code location where the debugger suspends the normal execution is referred to as the *interception point*. It then moves forward in the instruction stream executing instructions (basically single-stepping through the instructions) using a new *forward recovery* technique which keeps track of program states. When the debugger reaches the farthest extent of the instructions that should happen before S, referred to as the *finish point*, it begins to answer the user's inquiries. When reporting the value of a variable, it uses the preserved source-consistent program state to recover the expected values.

The basic idea of the approach can be illustrated by the example in Figure 1.1. If the user sets a breakpoint at source statement $S2$, since instruction $I3$ originates from source statement $S3$, the debugger suspends execution at $I3$. The debugger keeps

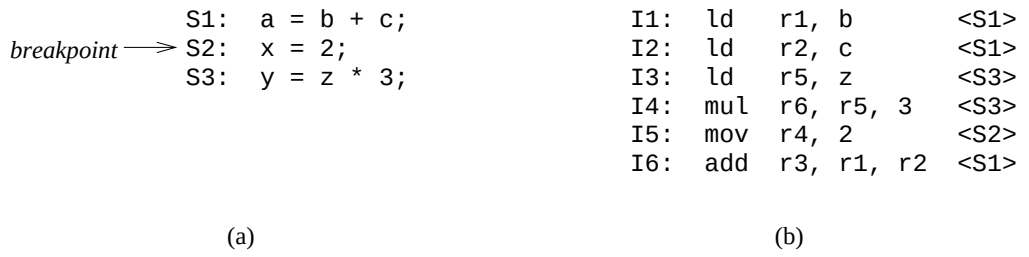


Figure 1.1 An example program (a) C-style source code (b) Optimized assembly code.

executing instructions under the forward recovery model until instruction $I6$ is executed because it originates from source statement $S1$ which should be executed before the breakpoint. The debugger then hands over the control to the user and starts taking user's requests. During forward recovery execution, the original contents of the registers which are updated prematurely are preserved to provide the user with the expected variable values at $S2$.

While the basic idea of the new debugging scheme appears straightforward for straight-line code, there are several challenging issues the scheme must address when dealing with globally optimized code. I use the example shown in Figure 1.2 to illustrate how global optimization complicates the problem. Figure 1.2(a) shows the control flow graph of an unoptimized program where instruction $I1$ is from statement $S1$, $I2$ is from $S4$, $I3$ is from $S2$, $I4$ and $I5$ are from $S3$, and $I3'$ is from $S5$. Figure 1.2(b) shows an optimized version of the program where instruction $I5$ is moved out of loop, instruction $I4$ is hoisted to basic block B, and $I3$ and $I3'$ are merged and sunk to basic block E. Basic block C becomes empty and is therefore removed. Suppose a source breakpoint is set at statement $S3$ by the user. The problems which need to be addressed by the proposed framework include:

1. *How to calculate all the possible interception points and finish points?* With code being reordered globally, instructions which should be executed after a source breakpoint might be hoisted above the breakpoint on different paths leading to the breakpoint. For example, in Figure 1.2(b), instruction $I4'$ and $I5'$ are the instructions which should be executed after the breakpoint but were hoisted. We can see that when the control first reaches basic block A, the debugger should suspend the execution at $I5'$, while when the control reaches basic block B through the back edge, the debugger should suspend the execution at $I4'$. Therefore $I5'$ and $I4'$ should both be interception points of $S3$. Similarly, $I3$ should be executed before the breakpoint but was sunk to basic block E. The debugger needs to be able to identify where $I3''$ is and continue its forward recovery until $I3''$ is executed. Hence it is necessary to devise a set of systematic algorithms to calculate all the possible interception points and finish points.

2. *How does the debugger confirm a source breakpoint?* To preserve the required program state, the debugger has to suspend the execution early at an interception point. However, reaching an interception point of a source breakpoint does not necessarily mean the breakpoint should be reported to the user. Consider Figure 1.2(b). After taking over control at instruction $I5'$, which is an interception point of $S3$, the debugger should report the breakpoint only when basic block C is reached. Otherwise, it should continue the normal execution without reporting the breakpoint. However, in this case, basic block C is removed after optimization.

We can see that there is no single object location which by itself can be used by the debugger to decide if statement $S3$ will be reached or not. Thus, a set of object locations and possibly some branch conditions will need to be incorporated into the mapping scheme to help the confirmation of a breakpoint.

3. *How does forward recovery work?* A run-time (debug-time) technique which maintains data structures to keep track of the program states changed during the forward recovery needs to be devised. This forward recovery technique also needs to ensure that all the source breakpoints and exceptions are reported to the user in the order prescribed by the source program for globally optimized code.

4. *Where are the locations of user variables at run-time?* The run-time locations of user variables may be altered by optimization. The variable value may be in different places (constant, register, or memory) at different points of execution. Or it may not exist at all. To allow the user to access the value of a variable at breakpoints, the debugger has to determine if the variable value exist or not, and if it does, where or how to obtain it.

The aforementioned problems are addressed in the proposed debugging framework and discussed in this dissertation.

1.1 Contributions

The major contributions of my dissertation work are discussed below.

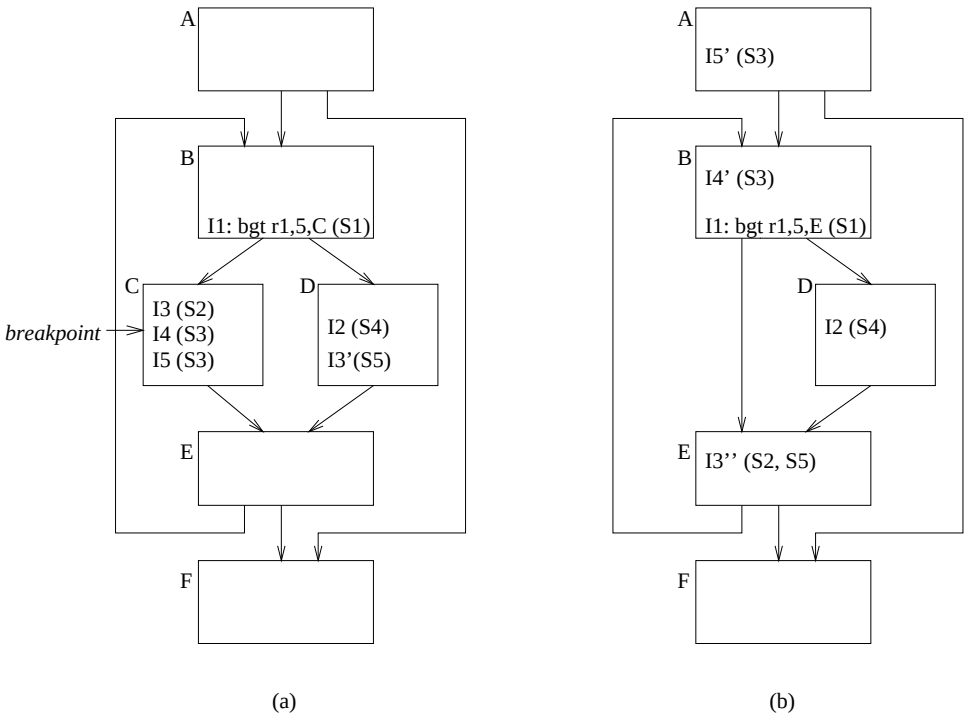


Figure 1.2 A control flow graph example (a) Original program (b) After code hoisting and tail merging.

- A general, non-invasive, and practical solution to debugging globally optimized code is proposed. In particular, I have proposed and developed

  1. a novel breakpoint implementation scheme, under which source breakpoints behave the way consistent with what the user expects and the program state can be properly preserved, and

  2. a new data-location tracking scheme that generates the information with which the debugger can unambiguously determine if the expected value of a variable is available at a source breakpoint and where to obtain it.

The framework addresses both code location and data value problems and is capable of recovering the expected program behavior for programs globally optimized by techniques involving code duplication, deletion, and instruction-level reordering. It does not require special run-time system support, nor does it need to insert special instrumentation code at compile time. Practical algorithms and theoretical foundations for the framework are devised and developed.

- A prototype debugger and necessary compiler support are implemented and evaluated. While a lot of the previous approaches have been developed and evaluated only on the compiler side, I have implemented not only the compiler support within the IMPACT compiler, but also an experimental debugger which incorporates the proposed debugging strategy to prove the concept and feasibility of the framework. Based on the fully-implemented prototype debugger and compiler support, the overhead and effectiveness of the proposed approach are quantitatively evaluated. This implementation experience and evaluation can be valuable and helpful to the future research in the area of debugging optimized code.

1.2 Overview

This dissertation is composed of seven chapters. Chapter 2 reviews various previous approaches to debugging optimized code and compare my approach with them.

Chapter 3 discusses the code location mapping scheme. A new instruction source order tracking method at compile time is described. The algorithms and the theoretical foundations for constructing and calculating source-to-object mappings are presented.

Chapter 4 describes the forward recovery model in the novel breakpoint implementation scheme. The data structures and methods used for preserving program state and ensuring source-consistent breakpoint and exception behavior are discussed in detail.

Chapter 5 presents the new data location tracking scheme. The approach to tracking variable definition information during optimization is described. A data-flow analysis based on the variable definition information is presented and explained. The algorithm using the data-flow information to generate run-time data location information is also presented.

Chapter 6 provides empirical results and evaluation based on the prototype. The experimental framework used for generating the results is first described. The overhead in compile time and executable file size due to the proposed debugging framework is presented. The cost incurred in setting and reporting source-level breakpoints under the new debugging framework as well as the effectiveness of the approach are quantitatively evaluated.

Conclusions and directions for future work are presented in Chapter 7.

CHAPTER 2

APPROACHES TO DEBUGGING OPTIMIZED CODE

Most of the previous approaches in debugging optimized code have focused on providing truthful program behavior [8], [9], [10], [5], [11], [12], [13], [14], [15], [16]. Hennessy [8] first introduced the concept of *currency*. A variable is *current* at a breakpoint if its value is consistent with what the user expects from the original source program at this breakpoint. A variable is *endangered* if it might not be current. In his paper, Hennessy provided algorithms to detect non-current and endangered variables caused by selected local and global optimizations.

Coutant, Meloy and Ruscetta at HP [10] modified an existing C compiler and a source-level symbolic debugger to support optimized code debugging. The optimizations they addressed are global register allocation, induction variable elimination, copy propagation, and instruction scheduling. The most noticeable part of their work is their solution to the problem of tracking data locations. Based on the live range information of variables, their compiler builds *range* data structure which keeps track of run-time locations of variables in different address ranges of the binary program. With the range information, the debugger can decide if there exists *any* source-level value for a source variable at an

object code location. A similar data structure is adopted in my dissertation work for the compiler to communicate to the debugger the location information for variables.

Range information calculated based on the live ranges of variables is considered conservative because of the fact that a variable is not live does not necessarily mean its value should be unavailable. Adl-Tabatabai and Gross [13] later proposed a framework using data-flow analysis to extend the range of a value location to the points where the variable value is killed. Both works done by Coutant et al. and Adl-Tabatabai et al. provide early implementation experience valuable to this dissertation work and are further discussed in Chapter 5.

Works done by Copperman [5], [11] and Wismuller [12] are similar. Both of their works focused on data value problem. Their approaches used data-flow analysis to determine and inform the user if a variable is current. Adl-Tabatabai and Gross [14], [15], [3] also proposed algorithms to detect endangered variables caused by local and global optimizations. Their approach provides more precise classifications of variables by further determining if an endangered variable is absolutely *non-current* or just *suspect*.

Tice and Graham [16] proposed an approach to display the effects of compiler optimizations at the source level by presenting a modified version of the source program. While my proposed framework is not suitable for optimizations that reorder loop iteration spaces such as loop interchange, loop fusion, loop skewing, etc., their approach can handle this kind of optimization better by providing a transformed source to the user.

There also have been several research works using different strategies to provide expected program behavior [17], [8], [3], [14], [18], [19], [20]. Zellweger's work [17] concen-

trated on code location problem. She proposed and implemented a method to handle programs optimized by *function inlining* and *cross jumping (tail merging)* where a source statement may have more than one instances, or a sequence of machine instructions might correspond to two or more statements. Her method can correctly map a source breakpoint to every object code location corresponding to the breakpoint, and can also determine whether to report a breakpoint in a merged area by inserting hidden breakpoints to the program. However, her work does not generalize to other optimizations, while my code location mapping scheme can handle optimizations involving code duplication, deletion, and instruction-level reordering in general. Data value problem is not addressed in her approach.

In his thesis [3], Adl-Tabatabai proposed to use branch conditions to help the debugger to confirm a source breakpoint when there is no single object location for the debugger to map the breakpoint to. This idea is similar to the *anchoring condition* mechanism in my code location mapping scheme (see Chapter 3). However, he did not provide any in-depth discussion on this topic, nor did he provide algorithms to keep track of the required branch conditions during compilation.

Hennessy [8] and Adl-Tabatabai et al. [14] proposed techniques to recover the expected values of variables. Their approaches are similar in concept. They recover the value of a variable by reconstructing and interpreting the original assignment of the variable. The expected value of the variable can be recovered successfully as long as the source operands of the assignment are still available at the object breakpoint. Since both of their approaches are based on a traditional source-to-object mapping scheme, the debugger

does not always suspend the execution early enough to preserve the values of the original source operands. Therefore their recovery approaches can be improved by my breakpoint implementation scheme.[1] Also Adl-Tabatabai did not address the recovery of variable values in globally optimized code, while Hennessy only briefly mentioned some extensions to support a limited set of global optimizations in his paper and did not address the problem of tracking run-time locations of variable values.

Gupta [19] proposed an approach to debug trace scheduled code. The user has to specify monitoring commands before compilation. These commands will be compiled into the program and later on used by the debugger to report the monitored information to the user. The major problem with this invasive approach is that adding extra code to the debugged program might change the program behavior and consequently introduce new bugs.

Holzle, Chambers and Ungar [20] proposed an approach in their *SELF* programming environment [21] to debug globally optimized code. By dynamically *deoptimizing* code on demand, their debugger can provide full expected behavior. In their approach, the debugger can be invoked only at pre-defined *interrupt points* where the program state is guaranteed to be consistent with what the original program would have. This constraint implies that the optimization can only be performed so that its effects either do not reach an interrupt point or can be undone at that point. Once the debugger is invoked, the function containing the interrupt point is deoptimized so that the debugging requests

[1] The effectiveness of my framework in the recovery of expected variable values is compared with that of Hennessy's approach quantitatively in Chapter 6.

can be carried out. With the function deoptimized, the program can be stopped at any source point within the function and almost all the typical debugging operations can be supported. In this deoptimization scheme, the user is actually debugging *unoptimized* code, whereas in my scheme it is the optimized code that is being debugged.

CHAPTER 3

CODE LOCATION MAPPING

A debugger usually uses two kinds of code mappings [2],[3]: the *object-to-source* mapping which the debugger uses to report the faulty statement when an exception occurs, and the *source-to-object* mapping which the debugger uses to determine where to suspend the normal execution and decide if a source breakpoint should be reported. Since I am only interested in the implementation of user breakpoints in this dissertation, the proposed scheme only focuses on source-to-object mapping. The object-to-source mapping, nonetheless, can easily be built from the source ordering information preserved during compilation (see Section 3.2.1).

To solve the code location mapping problem in debugging optimized code, there have been different source-to-object mapping schemes proposed such as *semantic breakpoint mapping* [2] which maps a source statement to the object code location that performs the operations specified by the statement, *syntactic breakpoint mapping* [2] which preserves the position of a statement with respect to its neighboring statements, and *statement label mapping* [3, 10] which usually maps a statement to the first instruction originating from the statement. Each of these mapping schemes maps a source breakpoint to a different place in the object code to preserve different kind of source code properties. However, since all of them map a source breakpoint to a single object location, only the

program state of a single point is available once the execution is halted by the debugger. Therefore, optimized code debuggers that adopt a traditional breakpoint implementation scheme usually have problems reporting the expected values of the variables which are updated either too early or too late. When the values of these variables are requested, the user will be informed that the expected values are not available at this point. The availability of the variable values decreases when the code is optimized by increasingly aggressive techniques which usually cause more code re-organization.

Unlike the previous source-to-object mapping schemes where a source statement is mapped to a single object location, my approach maps a statement to a set of object locations which can be classified into four categories with different functionalities: *anchor points*, *interception points*, *finish points*, and *escape points*. Interception points are the object locations where the debugger should suspend the normal execution and start forward recovery. Finish points are the object locations where the debugger should stop forward recovery and begin to take the user's requests. Escape points are used for the debugger to determine that a source breakpoint should not be reported. Anchor point information is the base for deriving interception, finish, and escape points, and needs to be constructed and maintained by the compiler. Interception points, finish points, and escape points can be derived from the anchor point information at debug time. I will discuss each of these object locations in the following sections.

3.1 Anchor Points

In a traditional mapping scheme, a source breakpoint at statement S is mapped to a single object location (usually the first instruction of S). Without optimization, reaching this object location at run time means statement S is reached (providing the compiler is correct) and the debugger should report the breakpoint to the user.

Optimization, however, leaves this simple scheme insufficient. During optimization, the first instruction of a statement (or even the whole statement) might be deleted or moved away from its original place. Reaching the first instruction of a statement S does not necessarily mean S will be reached in the original source program. Sometimes the compiler cannot even find a single object location in the optimized code to correctly map a source statement to, as illustrated by the example shown in Figure 1.2 (b).

In order for the debugger to be able to correctly confirm a source breakpoint for globally optimized code, each source statement is associated with *anchor point* information. An anchor point of a source statement is an object code location (an instruction). Each anchor point comes with a boolean condition referred to as the *anchoring condition*. When an anchor point of a source statement is reached during execution and its anchoring condition is true, the breakpoint set at that source statement should be reported.

Anchor point information for each source statement is constructed and maintained by the compiler. Before any optimization is performed, the anchor point of a source statement S is set to the first instruction of S and the anchoring condition is set to boolean value 1 (true).

During the process of code optimization, when code duplication optimization such as loop unrolling, function inlining, and loop peeling is performed, if an anchor point of statement S is contained in the duplicated code, the anchor point information is also duplicated. When an instruction I which is an anchor point of statement S is deleted or moved away from its original place, the compiler will modify the anchor point information of S using the algorithm shown below.

case 1 If I has an immediate succeeding instruction J in the same basic block, J replaces I to become an anchor point of S and the anchoring condition is boolean value 1.

case 2 else if I has an immediate preceding instruction J in the same basic block, J replaces I to become an anchor point of S and the anchoring condition is boolean value 1.

case 3 else, all of I's immediate preceding instructions, J_1, J_2, ..., J_k (where $k \geq 1$), jointly replace I to become anchor points of S. If J_i is a conditional branch instruction, the condition under which J_i will branch to I becomes the anchoring condition. Otherwise, the anchoring condition is boolean value 1.

Note that the algorithm is based on the assumption that conditional branches will not be removed (assuming no predicated code). Thus any instruction I which is being removed is never a conditional branch and its anchoring condition is always 1. If the condition of a branch is a constant, our method allows the branch to be treated as an unconditional jump and thus allows it to be removed.

Refer back to Figure 1.2(b), where the whole basic block C is removed due to optimization. Based on our algorithm, instruction $I1$ becomes the new anchor point of $S3$ (and $S2$) with the anchoring condition of $r1 > 5$.[1]

Using the above algorithm, the anchor point(s) identified for each source statement preserves the original source order. That is, if statement $S2$ follows statement $S1$ in source order, the anchor point(s) of $S2$ will not be reached earlier than the anchor point(s) of $S1$ (in the same iteration) during execution.

3.1.1 Proof of correctness

To prove that the anchor point information maintained by the compiler using the algorithm shown in Section 3.1 is correct, I will show that the debugger can unambiguously decide if a source breakpoint should take effect based on the anchor point information. I first introduce the concept of *reaching condition*.

Definition 1 *The reaching condition of an instruction I, RC_I, is a boolean expression comprising program variables and intermediate results so that when the condition is true, instruction I will be reached from the function entry point.*

Note that I assume that conditional branches will not be removed during optimization, therefore the reaching condition of an instruction remains the same during optimization as long as the instruction itself is not moved or deleted.

[1] In practice, the direction of the branch (taken or fall-through) instead of the actual boolean expression is used as the anchoring condition.

There might be more than one path which can lead to an instruction. For an instruction to be reached through a specific path, on this path every branch condition under which the path will be taken has to be true. Therefore, the *single-path reaching condition* of instruction I through a specific path P, $RC_{I,P}$, is the conjunction of every branch condition on P under which P is taken. That is,

Definition 2 *$RC_{I,P} = \bigwedge_{i=1}^{n} C_i$, where C_i is the condition of branch i under which P is taken, and n is the number of branches on P.*

Since an instruction can be reached through multiple paths, the operational definition of the reaching condition of instruction I, RC_I, is the disjunction of all the I's single-path reaching conditions. That is,

Definition 3 *$RC_I = \bigvee_{i=1}^{n} RC_{I,Pi}$, where Pi is the ith path leading to I and n is the number of different paths leading to I.*

The reaching condition of an instruction can also be derived from that of its predecessors as the following lemma shows:

Lemma 1 *The reaching condition of the first instruction I of a basic block, RC_I, can be expressed as*

$$\bigvee_{i=1}^{n} (RC_{Ji} \wedge BC_{Ji,I})$$

, where $\{Ji\}$ is the set of immediate preceding instructions of I, RC_{Ji} is the reaching condition of instruction Ji, $BC_{Ji,I}$ is the branch condition under which Ji will branch to I, and n is the number of I's immediate preceding instructions.

For any two instructions in the same basic block, when control reaches one instruction, it will definitely reach or have reached the other. Therefore,

Lemma 2 *All the instructions in the same basic block have the same reaching condition.*

As I mentioned earlier, a source breakpoint will be reported only when any of its anchor points is reached and the corresponding anchoring condition is true. The condition for the debugger to report a source breakpoint is referred to as *breakpoint confirmation condition.*

Definition 4 *The breakpoint confirmation condition of a source statement S, BCC_S, is*

$$\bigvee_{i=1}^{n} (RC_{Ii} \wedge AC_{Ii,S})$$

, where $\{Ii\}$ is the set of anchor points of S, $AC_{Ii,S}$ is the anchoring condition of Ii with respect to S, and n is the number of S's anchor points.

Before any optimization is performed, the proposed scheme will map the anchor point of statement S to the first instruction, say I, of S and set the anchoring condition to 1. Assuming the compiler is correct, it is true that for the unoptimized code the breakpoint set at S should be reported if and only if I is reached. That is, before any optimization is performed, the breakpoint confirmation condition of S is a sufficient and necessary condition for the breakpoint set at S to be reported. Therefore if I can prove that the breakpoint confirmation conditions before and after the algorithm in Section 3.1 is applied are the same, the algorithm is correct.

Lemma 3 *When an instruction I is removed due to optimization, its reaching condition, RC_I, is equal to*

$$\bigvee_{i=1}^{k} (RC_{Ji} \wedge AC_{Ji,I})$$

, where $J_1, J_2, ..., J_k (k \geq 1)$ are the instructions calculated using the algorithm in Section 3.1, and $AC_{Ji,I}$ is the anchoring condition of Ji with respect to I.

Proof : When the algorithm in Section 3.1 is applied, if either step 1 or step 2 is true, there will be only one instruction returned (i.e. $k = 1$) and the anchoring condition is 1. Assuming J_1 is the instruction returned, since J_1 and I are in the same basic block, according to Lemma 2, $RC_{J_1} = RC_I$.

Thus, $\bigvee_{i=1}^{1}(RC_{Ji} \wedge AC_{Ji,I}) = RC_{J_1} \wedge 1 = RC_I \wedge 1 = RC_I$.

If step 3 is applied, all the I's immediate preceding instructions, $J_1, J_2, ..., J_k$ will be returned and the branch condition of J_i, $BC_{Ji,I}$, under which J_i will branch to I becomes the anchoring condition of J_i, $AC_{Ji,I}$.

Hence, $\bigvee_{i=1}^{k}(RC_{Ji} \wedge AC_{Ji,I}) = \bigvee_{i=1}^{k}(RC_{Ji} \wedge BC_{Ji,I})$.

From Lemma 1, we know $RC_I = \bigvee_{i=1}^{k}(RC_{Ji} \wedge BC_{Ji,I})$, where $BC_{Ji,I}$ is the branch condition under which Ji will branch to I.

Therefore, $RC_I = \bigvee_{i=1}^{k}(RC_{Ji} \wedge BC_{Ji,I}) = \bigvee_{i=1}^{k}(RC_{Ji} \wedge AC_{Ji,I})$

Theorem 1 *When an instruction I, which is an anchor point of source statement S, is removed due to optimization, the breakpoint confirmation conditions of S before and after the algorithm in Section 3.1 is applied are the same.*

Proof : Assuming S has k anchor points, $I1$, $I2$, ..., Ii, ..., Ik, where $k \geq 1$ and $Ii = I$, according to Definition 4, we have

$$BCC_{S,before} = (RC_{I1} \wedge AC_{I1,S}) \vee ... \vee$$

$$(RC_I \wedge AC_{I,S}) \vee ... \vee (RC_{Ik} \wedge AC_{Ik,S}) \tag{3.1}$$

After instruction I is removed and the algorithm is applied, assuming m new anchor points, $J1$, $J2$, ..., Jm, are calculated in place of I, according to Definition 4, we have

$$BCC_{S,after} = (RC_{I1} \wedge AC_{I1,S}) \vee ... \vee$$

$$(RC_{J1} \wedge AC_{J1,I}) \vee ... \vee (RC_{Jm} \wedge AC_{Jm,I})$$

$$\vee ... \vee (RC_{Ik} \wedge AC_{Ik,S})$$

From Lemma 3, we know $RC_I = \bigvee_{i=1}^{m}(RC_{Ji} \wedge AC_{Ji,I})$. Also, based on the assumption of the algorithm that only non-conditional-branch instructions can be removed, we know that the anchoring condition of I (with respect to S) is 1 (i.e. $AC_{I,S} = 1$).

Therefore, $RC_I \wedge AC_{I,S} = RC_I = \bigvee_{i=1}^{m}(RC_{Ji} \wedge AC_{Ji,I})$.

Thus, replacing $RC_I \wedge AC_{I,S}$ with $\bigvee_{i=1}^{m}(RC_{Ji} \wedge AC_{Ji,I})$ in Equation (3.1), we have $BCC_{S,before} = BCC_{S,after}$.

3.2 Interception Points and Finish Points

When the user sets a breakpoint, the debugger needs to first identify the *interception points* and *finish points* corresponding to the source breakpoint so that it knows where the

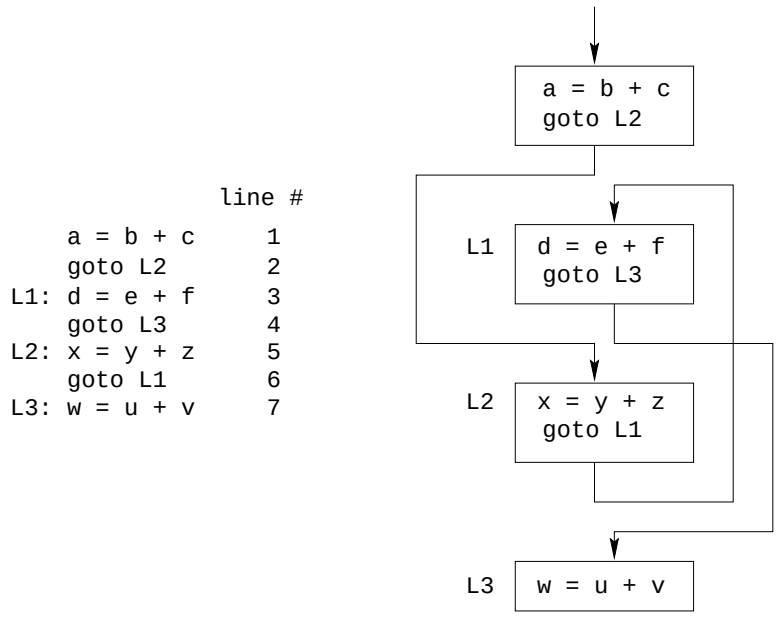
normal execution should be suspended and where the forward recovery should stop. Note that reaching an interception point of a source breakpoint does not necessarily mean the breakpoint should be reported to the user. Only when an anchor point of the breakpoint is reached during forward recovery can the debugger report the breakpoint.

To calculate the interception points and finish points, information about the original source ordering of instructions has to be constructed and preserved during compilation. In the following subsections, an instruction source-order tracking method is presented, and the algorithms to calculate the interception points and finish points are described.

3.2.1 Instruction source order

I propose an instruction source order tracking method which determines the original execution order of all the instructions and maintains this information during compilation. In my scheme, I do not distinguish execution order between instructions originating from the same source statement. The reason for this is because I am focusing on source-level breakpoints which can only be set at statement boundaries.

To determine the source order of instructions, one would intuitively think about using source line numbers and column numbers, and annotating each instruction with this information. Although the line number and column number information can determine the execution order of the instructions in the same basic block, it is not sufficient to track the execution order of the instructions across basic blocks. As we can see from Figure 3.1, although statement L1 has a smaller line number (line 3) than statement L2 (line 5), L2 will always be executed before L1 in the dynamic execution flow as shown



(a) (b)

Figure 3.1 Example program (a) Source program with line numbers (b) Control flow graph.

---

in Figure 3.1(b). Therefore it is necessary to incorporate control flow information to the source ordering information.

In my proposed scheme, each basic block is assigned an integer number, called *sequence number*, which reflects the dynamic execution flow. Sequence numbers are assigned to basic blocks of the original program such that basic block A has a smaller sequence number than basic block B if and only if each path from B to A involves a back edge.

The compiler computes sequence numbers by duplicating nodes to make the flow graph reducible, removing back edges, and then topologically sorting the resulting acyclic graph. Note that the sequence number assignment might not be unique, but there is only

one relative execution order between two basic blocks where the execution control can reach one from the other without traversing the back edges.

Theorem 2 *In a reducible acyclic control flow graph, there is always a well-defined execution order between two basic blocks where execution control can reach one from the other.*

Proof : We prove the theorem by contradiction.

> Suppose there is a basic block A which can be reached both before and after another basic block B in the control flow graph. There must be a path from A to B and back to A, which makes the graph cyclic and contradicts our assumption. Thus, there is only one execution order between A and B.

Having the sequence number, line number, and column number information associated with each instruction, a simple comparison of the numbers can determine the original execution order of instructions.

Before any optimization is performed, sequence numbers will be assigned, along with the line number and column number information, to each instruction. During an optimization phase, the source ordering information associated with each instruction remains the same as long as there is no code duplication or code creation optimization performed.

When code duplication optimization which duplicates basic blocks is performed, maintaining the sequence number information depends on if the duplicated code is introduced to a new context. In optimizations such as loop unrolling, function inlining, and loop

peeling, the duplicated basic blocks are introduced to a context different from their original one. Their original sequences numbers may no longer be valid in the new context, so it is necessary to dynamically adjust the sequence numbers of the duplicated code and the affected instructions in the surrounding new context to reflect the new execution order. For optimizations such as tail duplication where the duplicated code remains in the old context, the original sequence number information is kept.

To show how the compiler adjusts the sequence number information, I use a function inlining example. Figure 3.2(a) shows an example C program with two functions, *foo* and *bar*, where *bar* calls *foo*. Each statement is annotated with the execution order information *(sequence number, line number, column number)*. After inlining, statement $S8$ is replaced with a set of statements duplicated from function *foo* as shown in Figure 3.2(b). In order to maintain the correct relative execution order among instructions originating from function *foo* and function *bar*, we need to change the sequence numbers of all the new statements coming from *foo* and the sequence numbers of the statements which should be executed after $S8$. In Figure 3.2(b), we can see the sequence numbers of the duplicated statements are all changed to 2 (their original sequence number plus 1, the original sequence number of the function call) and the sequence number of $S9$ becomes 3.

Sequence number adjustment for loop unrolling and loop peeling can be done in a similar fashion.

For optimizations which involve creating new code such as common subexpression elimination, the newly-inserted instructions are treated as if they are from one of the statements involved in the optimization and assign them the same source ordering infor-

```
    foo(int a, int b)
    {
       int t;
S3:    t = a;       (1, 4, 4)
S4:    a = b + 4;   (1, 5, 4)
S5:    b = t;       (1, 6, 4)
    }

    bar()
    {
       int x, y;
S6:    x = 2;       (1, 12, 4)
S7:    y = 3;       (1, 13, 4)
S8:    foo(x, y);   (1, 14, 4)
S9:    y = x + 1;   (1, 15, 4)
    }
```

(a)

```
    foo(int a, int b)
    {
       int t;
S3:    t = a;       (1, 4, 4)
S4:    a = b + 4;   (1, 5, 4)
S5:    b = t;       (1, 6, 4)
    }

    bar()
    {
       int x, y;
S6:    x = 2;       (1, 12, 4)
S7:    y = 3;       (1, 13, 4)
       {
         int a,b,t;
S1':     a = x;     (2, 1, 4)
S2':     b = y;     (2, 1, 4)
S3':     t = a;     (2, 4, 4)
S4':     a = b + 4;(2, 5, 4)
S5':     b = t;     (2, 6, 4)
       }
S9:    y = x + 1;   (3, 15, 4)
    }
```

(b)

Figure 3.2 Sequence number adjustment for function inlining (a) Original C source code (b) Functions after inlining. Each statement is annotated with (sequence #, line #, column #).

mation as the other instructions of the statement. For example, Figure 3.3(b) shows an optimized program after common subexpression elimination, where $S3$ is newly created code. The source ordering information of $S1$ is assigned to $S3$ because $S3$ is treated as if it originates from $S1$.

3.2.2 Interception points

With regard to a breakpoint at source statement S, all the instructions in the function can be divided into two groups based on the source ordering information:

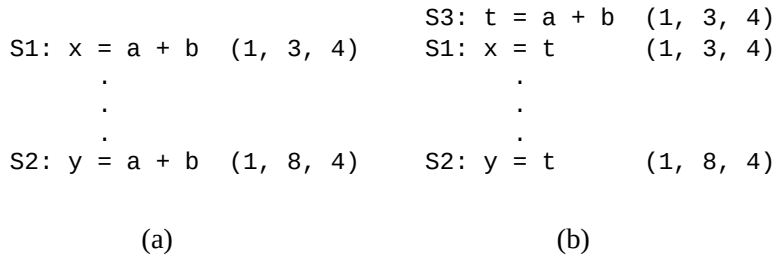


Figure 3.3 Execution order information maintenance (a) Original C source code (b) Program after common subexpression elimination. Each statement is annotated with (sequence #, line #, column #).

Pre-breakpoint instructions are the instructions which have a source execution order smaller than S, and

post-breakpoint instructions are the instructions which have a source execution order equal to or larger than S.

Instructions which can neither reach S nor be reached from S without traversing back edges may be classified as either pre-breakpoint or post-breakpoint based on their source ordering information. These instructions are irrelevant as long as they are not moved to a place which can reach S or be reached from S during optimization. Otherwise, the values of the variables affected will be denoted as unavailable to avoid providing misleading information (see Chapter 5).

Interception points for a breakpoint set at S, assuming instruction I is an anchor point of S, are calculated using:

- every path from the function entry point to I without traversing back edges, and

- every path starting from the loop header to I without traversing back edges for each loop which I resides in.[2]

Along each path mentioned above, the first post-breakpoint instruction encountered is an interception point of S.

Note that the above algorithm assumes that all the loops in the optimized code are *monotonic*. A loop in the optimized code is called *monotonic* if all the instructions in iteration $i+1$ of the loop are supposed to be executed after any instruction in iteration i in terms of the original program execution order, as opposed to *non-monotonic loops* such as modulo scheduled loops [23, 24] where instructions from different iterations of the original loop are mixed together in the same iteration of the new loop. In this dissertation I only base the discussion of the proposed approach on the assumption that all the loops in the optimized code are *monotonic loops* (such as unrolled loops).

Referring back to Figure 1.2(b), assuming $I1$ is the only anchor point of $S3$, there are two paths leading to $I1$ which the debugger needs to consider: path $P1 = < A, B(I1) >$ and path $P2 = < B(I1) >$. Assuming $I5'$ is the earliest post-breakpoint instruction along $P1$, $I5'$ is an interception point of $S3$. Also, assuming $I4'$ is the earliest post-breakpoint instruction along $P2$, $I4'$ is another interception point of $S3$.

An algorithm using backward data-flow analysis to systematically calculate all the interception points with regard to an anchor point is presented in the following.

In the control flow graph G of the function, suppose an anchor point I of statement S is in basic block D and the function entry block is E. To find out the interception

[2]The definitions of the loop header and the back edge can be found in Reference [22].

points of S with regard to I, D needs to be first split into two basic blocks $D1$ and $D2$, where

1. $D1$ is the top portion of D including instructions from the first instruction of D up to the one at I.

2. $D2$ contains the bottom portion of D including instructions from the one immediately following I to the last one.

3. All the D's predecessors become $D1$'s predecessors.

4. All the D's successors become $D2$'s successors.

5. There is no edge directly from $D1$ to $D2$.

Let V be the set of basic blocks which can reach $D1$ without traversing back edges in graph G (including $D1$).[3] For each basic block B in graph G, $gen[B]$ and $kill[B]$ are defined as follows:

- If B is in V,

  $gen[B]$ = A one-element set containing the first post-breakpoint instruction in basic block B, if there is any. An empty set, otherwise.

  $$kill[B] = \begin{cases} out[B] & \text{if } gen[B] \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

- If B is not in V,

  $gen[B] = kill[B] = \phi$

[3] V can be obtained through a simple backward depth-first search from $D1$.

```
for each block B in graph G do
    if B is in V then
       in[B] = gen[B];
    else
       in[B] = φ;
    endif
end for
while changes to any of the in sets occur do
    for each block B in V do
        out[B] = ⋃_{S is a successor of B} in[S];
        in[B] = gen[B] ∪ (out[B] − kill[B]);
    end for
end while
```

Figure 3.4 An iterative algorithm for interception point calculation.

The data-flow equations for in and out sets of B are:

$$
\begin{aligned}
out[B] &= \bigcup_{S \text{ is a successor of } B} in[S] \\
in[B] &= gen[B] \cup (out[B] - kill[B])
\end{aligned}
$$

A standard iterative algorithms for solving data-flow equations [22] to derive the $in[B]$ for each basic block B is presented in Figure 3.4. The union of $in[E]$ and $in[D2]$ is the set of all the interception points of S with regard to I.

3.2.3 Finish points

To identify finish points, an issue about function calls needs to be first addressed. Due to the surrounding pre-breakpoint instructions, if a post-breakpoint function call which

performs some I/O operations such as printing messages to the display screen is executed by the debugger, the user can be confused because the breakpoint was supposed to be reached before the function call. Therefore, the debugger in the proposed framework is not allowed to execute those post-breakpoint function calls while it does forward recovery. Instead, the finish point will be set before any of function call instructions. This way the ability of the debugger to recover the expected values of some variables might be reduced because the debugger does not always execute all the pre-breakpoint instructions, but at least it does not confuse the user. Since most compilers have very limited abilities to move code across I/O function calls, this will not be a serious issue in practice.

Suppose I is an anchor point of a source statement S. For each different path from I to the function exit point without traversing back edges, the finish point of S on this path is either the instruction immediately preceding the earliest post-breakpoint function call or the last pre-breakpoint instruction, depending on which one is encountered first. An algorithm using data-flow analysis to calculate all the finish points with regard to an anchor point is designed. The algorithm is similar to the one shown in Section 3.2.2 and is presented in Appendix A.

3.3 Escape points

Conceptually, the debugger confirms a source breakpoint when an anchor point of the breakpoint is reached during forward recovery and the anchoring condition is true. However, before the debugger can be sure that no anchor point of the statement is going

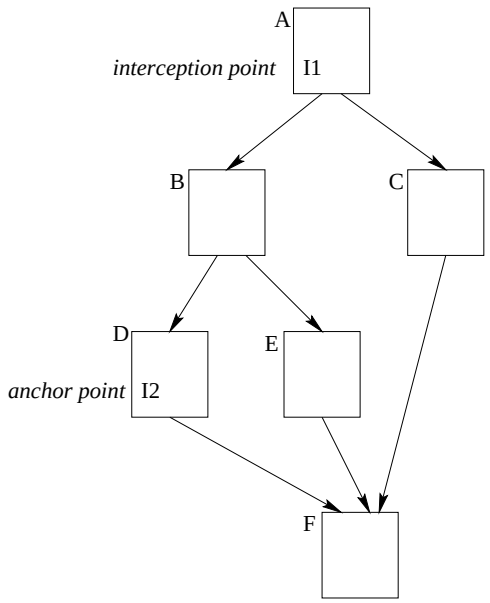


Figure 3.5 A control flow graph example.

to be encountered, it will have to scan forward in the binary all the way to the end of a function. In order to allow the debugger to resume normal execution as soon as possible when the breakpoint should not take effect, another set of object locations (referred to as the *escape points*) which are derived from anchor point information is proposed. An *escape point* of a source breakpoint is an object location such that when it is reached during forward recovery, its corresponding breakpoint should not be allowed to take effect and the normal execution is resumed.

For a breakpoint set at source statement S, there are two sets of escape points corresponding to it. The first set includes those instructions which can be reached from any of S's interception point(s) but does not lead to any of S's anchor point(s) without traversing back edges. The escape points in this set are calculated in the following way:

step 1 Using a simple backward depth-first search to find out all the basic blocks which can reach any of the anchor points of S without traversing back edges. Let D be the set containing all the basic blocks found in this step.

step 2 Using a simple forward depth-first search to find out all the basic blocks which can be reached by any of the interception points of S. Let E be the set containing all the basic blocks found in this step.

step 3 $F = D \cap E$. Set F will then contain all the basic blocks which can be reached from one of the interception points and lead to one of the anchor points without traversing back edges.

step 4 For each basic block in F which itself does not contain an anchor point of S, the first instruction of its immediate successor which is not in set F is an escape point of S.

Figure 3.5 shows a control flow graph of an example program in which $I2$ is the only anchor point of a source breakpoint and $I1$ is the only interception point. We can see that there is only one path from $I1$ to $I2$, which is $< A, B, D >$. We find that basic blocks C and E are the immediate successors of A and B, and they are not on any path leading to $I2$. Therefore the first instructions of C and E are the escape points corresponding to anchor point $I2$.

The second set of the escape points includes those anchor points with anchoring conditions other than boolean constant 1. When an anchor point is reached, if its anchoring

condition is false and it does not lead to any of S's other anchor point(s) without traversing back edges, the anchor point itself becomes an escape point.

CHAPTER 4

FORWARD RECOVERY SCHEME

After the debugger takes over control at an interception point of a source breakpoint, the forward recovery process begins. During the forward recovery, the debugger must ensure the program state required for the recovery of expected variable values is properly preserved and the source breakpoints behave the way prescribed by the source program. In this chapter, I first briefly describe an intuitive forward recovery scheme and discuss some of the issues it has to address. I then describe in detail my proposed forward recovery model that avoids the problems faced by the intuitive scheme and is adopted in my prototype debugger.

4.1 Selective Emulation Model

To preserve source-consistent program state, an intuitive forward recovery model is to execute (emulate) only pre-breakpoint instructions and skip the post-breakpoint instructions. Refer back to the simple example in Figure 1.1. If this recovery model is used, after taking over control at $I3$ (the interception point), the debugger skips instruction $I3$, $I4$, and $I5$, and emulate only instruction $I6$ before it hands over control to the user. Once the user resumes execution, the debugger will go back to emulate those skipped instructions. It then writes out the emulated state to the debuggee, and resumes the de-

```
S1: a = 1            S1: a = 1            I1: r1 = 1
S2: c = 0            S3: b = a + 1        I2: r2 = r1 + 1
S3: b = a + 1        S2: c = 0            I3: r1 = 0
```

(a) (b) (c)

Figure 4.1 (a) Original code (b) Optimized code after instruction scheduling (c) Optimized code after register allocation.

---

buggee in native execution mode. This forward recovery model is referred to as *selective emulation* [6], [25].

In general, it is safe to emulate pre-breakpoint instructions before post-breakpoint instructions because the optimizer must respect the data dependencies of the source program when reordering instructions. However, since false dependencies might be introduced in the optimized code when register allocation is performed after instruction scheduling, a naive instruction reordering at debug time does not work. For example, Figure 4.1(a) shows the original code of a sample program. Figure 4.1(b) shows the optimized code after instruction scheduling. Figure 4.1(c) shows the optimized code after register allocation where both `a` and `c` are mapped to the same register `r1`. If a breakpoint is set at source statement $S3$, then `r1 = 0` must be emulated before `b = r1 + 1`. However, at the point when the debugger emulates `b = r1 + 1` (when the user continues from the breakpoint), the value of `r1` used for instruction $I2$ must be the value that it had before it was overwritten by `r1 = 0`.

One way to accomplish this is to represent the emulated state of the debuggee as a list of change records, one for each emulated instruction [6]. A change record stores the modified values of registers and memory. Although the instructions are emulated out

of binary order, the list is ordered by the binary ordering of the emulated instructions. When emulating an instruction, the debugger finds the instruction's position in binary order, and only use the *preceding* change records to construct the machine state needed for emulation. Consider the example shown in Figure 4.1 again. When the user continues from the breakpoint set at $S3$ and the debugger comes to emulate instruction $I2$, only the change record for instruction $I1$ is used to find the value of `r1` because $I1$ is the only instruction preceding $I2$ in binary order.

The selective emulation scheme is further complicated by the presence of function calls and post-breakpoint branches in the emulated region. Special run-time strategies need to be devised to address these problems [6]. Also this forward recovery model requires the debugger to incorporate an emulator of the target processor.

4.2 Proposed Forward Recovery Model

In this dissertation work I propose another forward recovery model which avoids the aforementioned issues faced by the selective emulation model. Under this forward recovery model, every instruction between the interception point and the finish point (or the escape point) is executed in the binary order and the values overwritten prematurely are saved. This model does not require a full-fledged emulator of the target machine in the debugger as all the instructions are natively executed. Therefore the debugging process is more efficient.

In this model, two important data structures need to be maintained by the debugger during the forward recovery. The first one is the *data history buffer* which keeps track of all the old contents of the destinations of the post-breakpoint instructions executed between the interception point and the finish point. These old values are necessary to recover the expected values of user variables. An entry in the data history buffer comprises a destination location, which may be a register number or a memory address, and one or more value information records. A value information record of a destination includes the address of the corresponding instruction and the old content of the destination.

The other data structure is called *instruction history buffer* which contains the addresses of the instructions executed between the interception point and the finish point in their dynamic execution order. Each address in the buffer might be annotated with some other information.

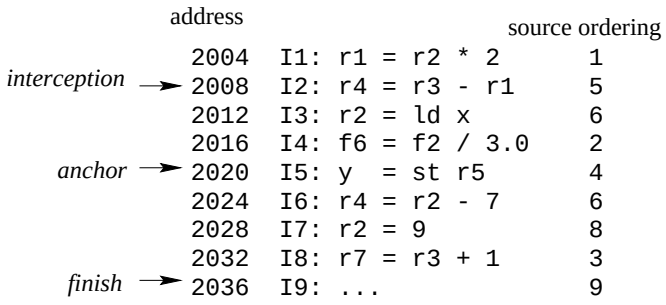
The instructions of the debugged program will be executed in either *normal mode* or *forward recovery mode*. The program starts running in normal mode. When an interception point of a source breakpoint is reached during normal execution, the debugger takes over control and the forward recovery mode is entered. From this point on, each instruction will be executed one by one (*single-stepped*) under the debugger's supervision until one of the finish points or escape points is reached.

In the forward recovery mode, before an instruction I is executed, the current content of I's destination along with the address of I will be saved in the data history buffer if I is a post-breakpoint instruction. The address of I is also saved in the instruction history buffer (regardless of whether I is pre-breakpoint or post-breakpoint). If I happens to be

an interception point of other breakpoint(s), I's entry in the instruction history buffer will be annotated with this information.

Figure 4.2(a) shows an optimized program example. For simplicity, I use a single number as the source ordering information in this example. Assuming the anchor point of a source breakpoint set by the user is at instruction $I5$. The interception point and the finish point will be set at $I2$ and $I9$ respectively. Once the debugger takes over control at $I2$, each instruction is executed in forward recovery mode until $I9$ is reached. Figure 4.2(b) and (c) show the resulting instruction history buffer and the data history buffer.

Because some instructions such as load and floating-point operations might cause exceptions during execution, handling the exceptions so that they behave in the way the user expects is very important. If an exception is caused by a post-breakpoint instruction and is posted immediately, the users might be confused because the exception should have occurred *after* the breakpoint. In order not to confuse the user, the debugger should suppress the exceptions caused by post-breakpoint instructions while executing them, and post the exceptions to the user later on. One way to achieve this is for the debugger to provide its own exception handling routine. When an exception occurs in the forward recovery mode, the handling routine provided by the debugger takes over. If the exception is caused by a post-breakpoint instruction, it will be suppressed and the debugger will annotate the entry of the instruction in the instruction history buffer with the exception information so that the exception can be signaled later on. Referring back to the example in Figure 4.2, if an exception occurs at instruction $I3$, since it is a post-



(a)

Instruction History Buffer

address	annotation
2008	
2012	
2016	
2020	
2024	
2028	
2032	

(b)

Data History Buffer

destination	old value	instr. addr
r4	8	2008
	6	2024
r2	-2	2012
	7	2028
M(y)	0	2020

(c)

Figure 4.2 (a) Optimized code example (b) Instruction history buffer (c) Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

---

breakpoint instruction, the exception will be suppressed and the entry corresponding to $I3$ in instruction history buffer will be annotated as shown in Figure 4.3.

When a finish point is reached, the debugger stops the forward recovery, reports the source breakpoint to the user, and starts answering user's requests. The information preserved during forward recovery, as well as the data location information (see Chapter 5), is used by the debugger to provide the expected variable values.

At a source breakpoint, the user might set additional breakpoints or delete existing breakpoints. When a new source breakpoint is set, besides setting the necessary object

Instruction History Buffer

address	annotation
2008	
2012	exception
2016	
2020	
2024	
2028	
2032	

Figure 4.3 Instruction history buffer.

breakpoints at the interception points, the debugger also needs to check if any of the instructions in the instruction history buffer is an interception point of the newly-set source breakpoint. If so, the debugger will annotate the corresponding entry in the instruction history buffer with the interception point information so that the newly-set breakpoint can be properly handled later on when the user continues from the current breakpoint. Similarly, when the user deletes an existing source breakpoint, the debugger needs to check if any of the instructions in the instruction history buffer is an interception point of the deleted source breakpoint. If so, the debugger needs to remove the interception point annotation from the corresponding entry in the instruction history buffer.

Once the user resumes execution, the debugger will go through the instruction history buffer to check if there is any annotated information and will update both the instruction history buffer and data history buffer. Until an instruction denoted as an interception point of another breakpoint (or other breakpoints) is encountered or the whole instruction history buffer has been processed, the debugger will visit each instruction I in the buffer with the following actions:

1. If the instruction is annotated with exception information, the debugger will signal the exception.

2. The value information record of this instruction's destination (if there is any) is removed from the data history buffer.

3. The entry of this instruction in the instruction history buffer is removed.

If the debugger has visited every instruction in the instruction history buffer without running into another interception point, the normal execution will resume from the finish point.

If an instruction visited is an interception point of another breakpoint, the debugger will continue going through the instruction history buffer in the following way:

1. If an instruction is annotated with exception information, the debugger will first determine the new type (pre-breakpoint or post-breakpoint) of the instruction with regard to the new breakpoint because a post-breakpoint instruction for the old breakpoint might become a pre-breakpoint instruction for the new breakpoint. If the instruction becomes pre-breakpoint, the debugger signals the exception and removes the annotation. Otherwise, the exception remains suppressed.

2. If the type of the instruction is changed (from post-breakpoint to pre-breakpoint), the value information record of this instruction's destination is removed from the data history buffer.

Since the finish points of the new breakpoint might be different from those of the old breakpoint, after having gone through the instruction history buffer, the debugger might need to execute more instructions in forward recovery mode until a finish point or an escape point is hit.

To understand the visiting process, refer to the example in Figure 4.2 again. If $I3$ is an interception point of another outstanding breakpoint (whose anchor point is at $I6$), its entry in the instruction history buffer will be annotated with this information as shown in Figure 4.4(a). When the user wants to resume execution from the current breakpoint, the debugger goes through the instruction history buffer. Since the instruction at address 2008 is not annotated with anything, the debugger removes it from the instruction history buffer and deletes its corresponding entry in data history buffer as shown in Figure 4.4. The debugger visits the next entry in the instruction history buffer and finds out the instruction at address 2012 is an interception point of another breakpoint. The debugger will keep going through the rest of the instruction history buffer without deleting any entry. Since $I5$ (at address 2020) becomes a pre-breakpoint instruction with regard to the new breakpoint, its entry in the data history buffer is deleted.

Similarly, when an escape point is reached in forward recovery mode, which means the breakpoint should not take effect, the debugger will go through the instruction history buffer in the same way described above and does not report the source breakpoint to the user.

Instruction History Buffer

address	annotation
~~2008~~	interception point
2012	
2016	
2020	
2024	
2028	
2032	

(a)

Data History Buffer

destination	old value	instr. addr
r4	~~8~~	~~2008~~
	6	2024
r2	-2	2012
	7	2028
~~M(y)~~	~~0~~	~~2020~~

(b)

Figure 4.4 (a) Instruction history buffer (b) Data history buffer (the old values in the data history buffer are given arbitrarily in the example).

4.2.1 Function calls in forward recovery

A source of complication for the recovery scheme arises from the function calls in the forward recovery region. As described earlier, during forward recovery, instructions are *single-stepped* (that is, the debugger takes over control immediately after each instruction is executed). Therefore executing instructions in forward recovery mode are considerably slower than in normal mode. To increase the performance of the proposed breakpoint implementation scheme, it is desirable to execute function calls in normal mode during forward recovery.

On seeing a function call in forward recovery mode, the debugger first sets an object breakpoint at the return point of the function so that it can regain control once the function is finished. Since there might be source breakpoints set in the called function, in order to avoid messing up the instruction and data history buffers of source breakpoints from different functions, the debugger also needs to push the current instruction and

data history buffers into stacks. The debugger then lets the function execute in normal mode.

While executing the function call in normal mode, if the debugger stops at an interception point of a source breakpoint in the called function, a new forward recovery process starts and the source breakpoint is handled the same way as described earlier. Note that all the source breakpoints set in the called function will be reported to the user before the original source breakpoint in the calling function. This behavior is consistent with what the user expects because all the function calls encountered during forward recovery are pre-breakpoint instructions (based on the finish point calculation algorithm described in Section 3.2.3), and therefore any source breakpoint in the called function is expected to happen before the original source breakpoint in process.

After the debugger regains control at the function return point, the original instruction and data history buffers are popped out from the stacks and the original forward recovery process is resumed.

4.2.2 Loops in forward recovery

Another problem occurs when the debugger has to execute all the iterations of a loop in forward recovery mode before it reaches a finish point or an escape point. There are two cases where this problem will happen:

1. There is a loop lying between the interception point and the anchor point. The interception point may or may not be in this loop, but the anchor point is not in the loop. Figure 4.5(a) shows an example of this case. After taking over the control

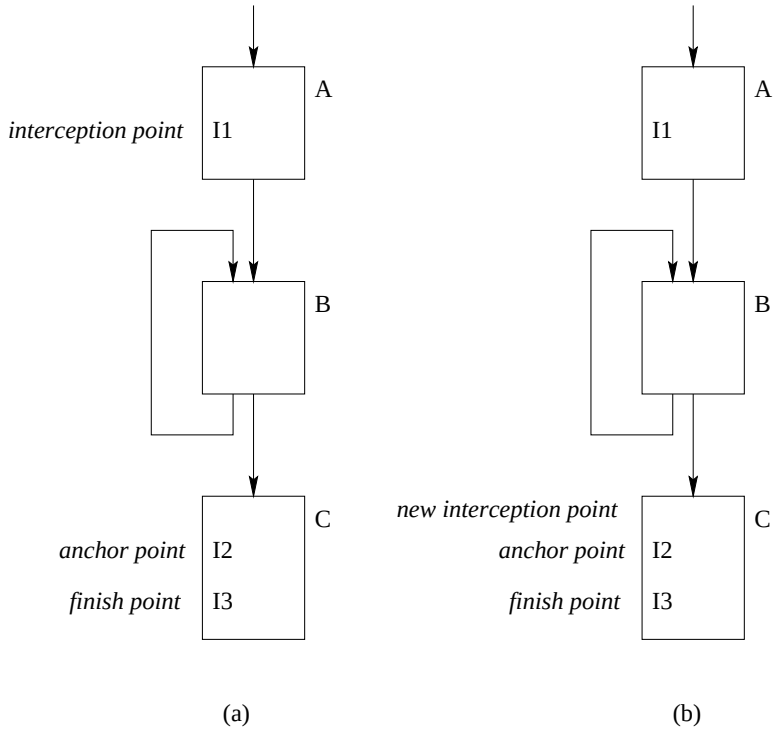


Figure 4.5 A control flow graph with (a) old interception point (b) new interception point.

---

of execution at $I1$, the debugger has to execute every iteration of the loop in the forward recovery mode before it can reach $I3$.

2. There is a loop lying between the anchor point and the finish point. The anchor point may or may not be in the loop, but the finish point is not in the loop. Figure 4.6(a) shows an example of this case. After taking over the control of execution at $I1$, the debugger has to execute every iteration of the loop in the forward recovery mode before it can reach $I3$.

Since the number of loop iterations is non-deterministic, the instruction history buffer and data history buffer might potentially explode.

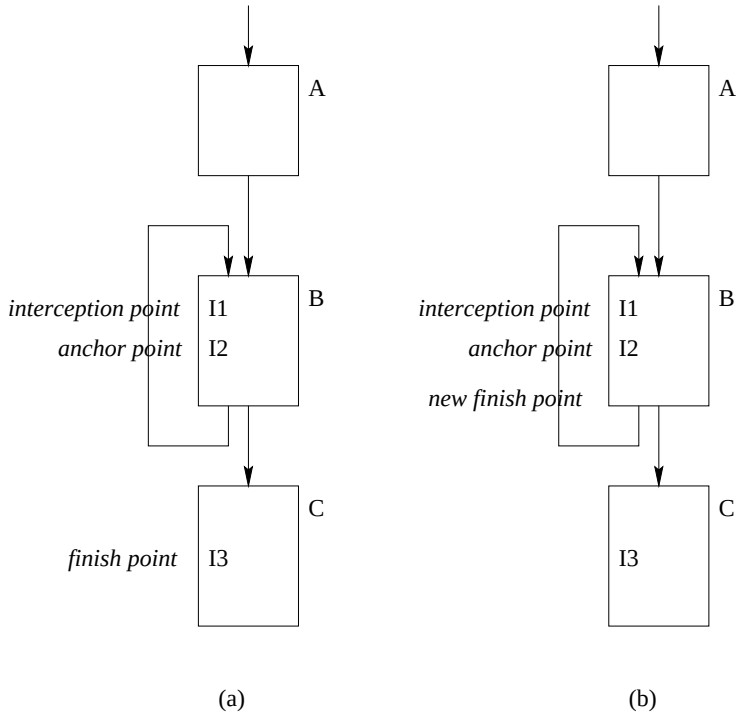


Figure 4.6 A control flow graph with (a) old finish point (b) new finish point.

To solve this problem, the debugger can adopt a less aggressive approach which trades the availability of the values of some variables for the feasibility of the proposed scheme. The algorithms for finding the interception points and finish points can be modified to avoid these cases. For example, in the case shown in Figure 4.5(a), the interception point can be set at the beginning of block C as shown in Figure 4.5(b). Similarly, in the case shown in Figure 4.6(a), the finish point can be set at the end of block B as shown in Figure 4.6(a). Some of the variable values might be unavailable at the breakpoint because of this modification.

4.2.3 Beyond the function scope

The discussion of the forward recovery scheme has been so far limited to within the function where the source breakpoint is set. When there are instructions moved across the call site of this function in the caller function, the scheme is further complicated. Figure 4.7(a) shows an example with two functions where function *bar* calls function *foo*. Figure 4.7(b) shows the program after the optimizer hoists the statement $S2$ (which is an assignment of global variable y) above statement $S1$. If a breakpoint is set at statement $S3$ and the forward recovery is only applied within function *foo*, the user won't be able to get the correct value of variable y. Apparently the debugger will have to take over the control of execution as early as at $S2$ in the optimized code in order to solve this problem. To do so, the debugger has to keep track of all the call sites for each function. When the user sets a breakpoint in function *foo*, the debugger will need to go to every caller of *foo* and see if there are instructions moved across the call site. If so, the debugger will have to set the interception points, finish points, and escape points in caller function as if there is a breakpoint set immediately after the call site. If there are multiple levels of functions calls where $F1$ calls $F2$, $F2$ calls $F3$, ..., and Fi calls *foo*, the interception points and finish points should set in the outer-most level of functions where there are instructions moved across the call site.

The problem with this solution is that the instruction history buffer and data history buffer might explode when there is a long chain of function calls. Also the time it takes to

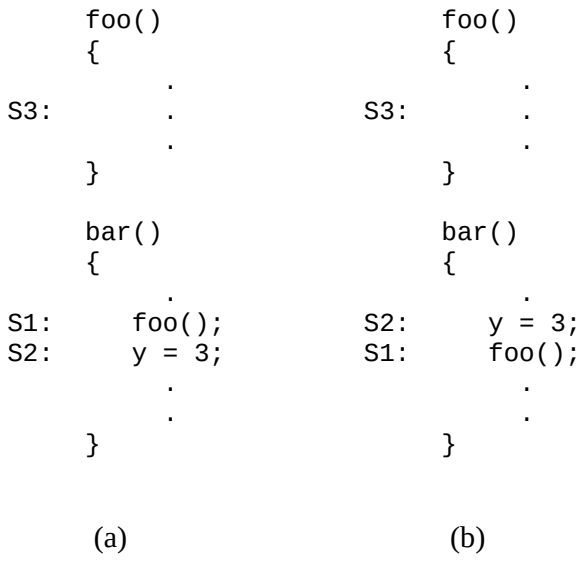


Figure 4.7 (a) Original program (b) Optimized program.

run to the breakpoint may be extremely long because lots of the instructions are executed in the forward recovery mode.[1]

Therefore, another more practical but less aggressive solution to this problem is to still use the forward recovery technique within a function. As for those variables whose values are updated either too early or too late due to the code movement in the caller functions, the debugger does not try to recover them but just inform the user that the values of those variables are not available at this point because of the optimization. By using this approach, the availability of some variable values is traded for the efficiency and feasibility of the method.

[1]The run-time strategy I discussed in Section 4.2.1 to handle function calls cannot be applied in this scheme because the debugger will not be able to preserve the values modified by the post-breakpoint instructions in the inner functions if the inner functions are executed in normal mode.

4.2.4 Proof of correctness

As I mentioned in Chapter 1, to provide the expected program behavior, it is essential for the debugger to report the source breakpoints in the order consistent with what the user expects. The new breakpoint implementation scheme handles and reports the source breakpoints in the order of their corresponding interception points being reached. To show that the source breakpoint behavior provided by my approach is correct, I need to prove that the interception points of source breakpoints are always reached in the order conforming to the original execution order of their corresponding source statements.

Theorem 3 *For any two source statements $S1$ and $S2$ such that $S1$ has a smaller execution order than $S2$ (that is, $S1$ should be executed before $S2$ if execution flow can reach $S2$ from $S1$), on any path which contains interception points of both $S1$ and $S2$, the interception point of $S2$ will never be reached before the interception point of $S1$.*

Proof : I prove the theorem by contradiction.

Suppose there is a path P contains an interception point of $S1$'s, $I1$, and an interception point of $S2$'s, $I2$, such that $I2$ is reached before $I1$ along P. From the definition of interception point, we know $I1$ is the earliest post-breakpoint instruction on path P with regard to $S1$.

Suppose $I2$ originates from statement B. B should be executed after statement $S2$. Since $S2$ should be executed after $S1$, B should be executed after $S1$. Therefore, $I2$ is also a post-breakpoint instruction with regard to $S1$. Because $I2$ is reached before $I1$ on path P, $I2$ is an earlier post-breakpoint instruction than $I1$ on P with

regard to $S1$, which contradicts the assumption that $I1$ is the interception point of $S1$ on path P.

CHAPTER 5

DATA LOCATION TRACKING SCHEME

Under the proposed breakpoint implementation scheme, the storage locations (memory and register file) updated prematurely by the out-of-order instructions can be properly preserved. However, because the run-time locations of user variables may vary at different points of the optimized program, in order to provide the user with expected variable values, the debugger still needs to be able to correctly associate the storage locations with user variables.

Coutant et al. [10] proposed a data structure called *range* to communicate to the debugger the location information for variables in different ranges of the binary program. A variable has a set of range records, each of which consists of an address range and a storage location (register, memory, or constant). By comparing the address of an object code location with each range record, the debugger can decide if there exists *any* source-level value for the variable at this object code location. If the address is not in any one of the range records, no source-level value of the variable is available at this object point. Range information is calculated based on the live ranges of variables. Adl-Tabatabai and Gross [13] later proposed a data-flow algorithm to extend the range of a value location to the points where the value is killed.

The aforementioned techniques have focused on presenting *truthful program behavior*. Therefore, while the data location information generated by these techniques provides a good foundation for the debugger to determine and warn the user if the source-level value of a variable can be found in any place at an object code location,[1] it becomes insufficient when the debugger attempts to recover the expected variable values at breakpoints under the proposed breakpoint implementation scheme, as illustrated in Figure 5.1. Figure 5.1 (a) and (b) show the original source code and the unoptimized assembly code of a sample program, respectively. Figure 5.1(c) shows the optimized code where instruction $I5$ (a definition of variable a) is moved down and instruction $I7$ (a definition of variable b) is moved up. The range records constructed for variable a and b using previous approach are depicted in Figure 5.1(d). Suppose a breakpoint is set at statement $S6$. If the user is interested in the value of variable b, based on the source program, b's value at the breakpoint should come from the definition at $I3$ (in register $r2$). In order for the debugger to correctly associate b with register $r2$ using the range records shown in Figure 5.1(d), the debugger has to map the source breakpoint to instruction $I7'$. However, this mapping becomes problematic if the user requests for a's value. When the debugger uses the address of $I7'$ to compare with a's range records shown in Figure 5.1(d), the debugger would think a's value is in register $r1$, while in fact the expected value of a should be in $r3$. Mapping the source breakpoint to instruction $I8$ instead of $I7'$ does not solve the problem either as we will encounter the same problem the other way around.

[1] If the source-level value of a variable can be found in any place at an object code location, the variable is *resident* at this object location [13].

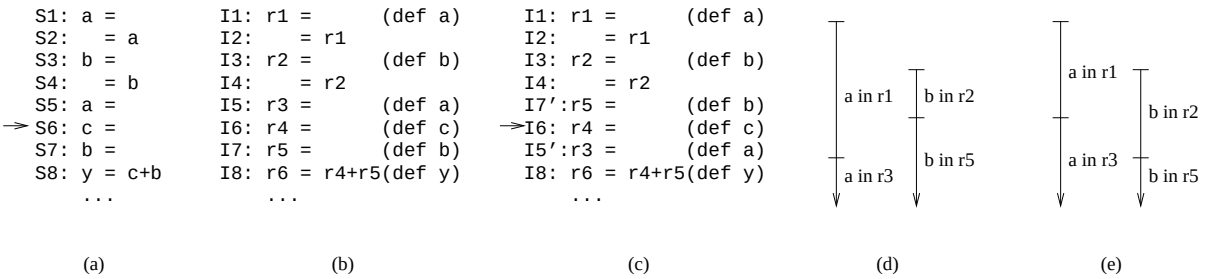


Figure 5.1 (a) Original source code (b) Unoptimized code (c) Optimized code (d) Range records for variable a and b using previous techniques (e) Range records for variable a and b desired by the proposed framework.

In my breakpoint implementation scheme, the *anchor point* of the source breakpoint (instruction $I6$ in this example) will be used to compare with the range records because the anchor point information preserves some important properties of the source program. We can see that the new breakpoint implementation scheme faces the same problem of incorrectly associating storage locations with both variable a and b using the range information shown in Figure 5.1(d).

To solve the problem mentioned above and provide data location information more fitting to my breakpoint implementation framework, a new data location tracking scheme which is extended from the previous techniques is proposed in this paper. The new scheme keeps track of information about the definitions of source variables during optimization and register allocation. A data-flow analysis is proposed to collect information that is used for calculating the range information which will correctly reflect the original execution order relationship between the assignments of a variable and source breakpoints. Refer back to the example in Figure 5.1. The range information constructed by the proposed data location tracking scheme will be the one depicted in Figure 5.1(e).

By comparing the anchor point of a source breakpoint with this range information, the debugger can unambiguously determine if the expected value of a variable is available at the source breakpoint and where to obtain it under my breakpoint implementation scheme.

Recovery of the expected values of variables whose assignments are deleted is also handled by the new data location tracking scheme. Most of the time when an assignment of a variable is deleted, the value assigned to the variable can still be found somewhere. To exploit this fact, I have extended the location information of a variable to be an expression which can be a constant, a register number, a memory location, or an arithmetic expression. The new scheme tracks where or how to obtain the value of a variable when an assignment to the variable is deleted during optimization. Although the concept of using the values of other variables and temporaries to recover a deleted variable's expected value has been briefly mentioned in some of the previous work [8], [10], my scheme provides a more general and systematic approach which can handle almost all kinds of code deletion and recover the expected values of deleted variables whenever it is possible.

The range information in my framework is calculated based on the anchor point information and the variable definition information that is generated and maintained by the compiler. The variable definition information basically tells the range record calculation algorithm whether a source assignment of a variable still *practically* exists after optimization, and where to find the value defined by this assignment if it does exist. The anchor point information is used to determine where a source assignment should start to take effect (which in turn decides where the corresponding range record should

begin). In the following sections I will discuss how the variable definition information is maintained for various optimization techniques and how the range information is built based on the information collected by a data-flow algorithm.

5.1 Variable Definition Information

For each source variable, the compiler maintains a set of *definitions*, each of which corresponds to a movement of a source-level value of the variable to a storage location. For each definition, the compiler keeps track of the following information:

Definition type: The type of a definition can be *original*, *equivalent*, *deleted*, or *virtual*. A definition that corresponds to a source assignment and is not deleted is *original*. When a definition of a variable is deleted during optimization, if we can manage to find the value defined by this definition somewhere, the definition is practically existent and the type will become *equivalent*. Otherwise, the definition type becomes *deleted*. Definitions introduced by optimizations are *virtual* definitions. Unlike other types of definition, a virtual definition of a variable does not correspond to any source assignment of the variable. It simply denotes that the run-time location of the variable is changed at some point of the program.

Variable location expression: This information denotes where or how to obtain the value of a variable. It can be a constant, a register, a memory location, or an arithmetic expression consisting of various storage locations.

Actual definition point(s): For an original or a virtual definition of a variable, the instruction that moves a source-level value of the variable to a storage location is the *actual definition point* of the definition. For an equivalent definition, the instructions whose destinations constitute the variable location expression of the variable are the *actual definition points*. Each actual definition point is annotated with a definition-point attribute.

Source ordering of the definition: This information contains the the source ordering information of the corresponding source assignment statement. It is only meaningful for non-virtual definitions.

Before any optimization is performed, a *definition* D is associated with each source assignment statement S of variable V. The instruction I originating from S that moves a source-level value of V to a storage location L is the actual definition point of D. The type of D is *original* and the variable location expression is L. Note that without precise interprocedural analysis, an indirect assignment or a function call with a pointer (or the address of a variable) as an argument is considered a definition of every variable whose storage location might be in the memory during the execution.

I discuss how the variable definition information is maintained for various optimizations that may cause the data value problem in the following cases.

1. *Register Allocation.*

   During register allocation, when a virtual register is assigned a physical register number, the compiler will check the variable definition information maintained and

then update accordingly the variable location expressions with the virtual register as an operand.

2. *Optimizations that transfer a source-level value of a variable from one storage location to another.*

   Optimizations in this case include, among others, register promotion (including global variable migration) where a memory variable is promoted to a register over a region of the code, and register spilling where a register is spilled to a memory location. A *virtual* definition is created for each transfer of a source-level value from one storage location to another during this kind of optimization.

3. *Code deletion.*

   When an instruction I, which is an actual definition point of a non-virtual definition D of variable V, is deleted, assuming d is the destination of I that appears in the variable location expression of D,

   (a) If the value of d can be obtained from expression E whose operands are available at I, and I is the only actual definition point of D which defines d, then

      (i) d (in the variable location expression of D) is replaced by E,

      (ii) the instructions which define the operands of E jointly replace I to become the actual definition points of D, and

      (iii) the type of D becomes *equivalent* if it isn't already.

Sometimes some operands of E might not be available due to the optimizations performed later on, the compiler will need to check the availability of E's operands before building the range information.

(b) Otherwise, the type of D becomes *deleted* and there are no actual definition point and location expression for D.

Refer to the program shown in Figure 5.2. Figure 5.2(a) shows an unoptimized program where instruction $I5$ is an actual definition point of an *original* definition of variable a. When instruction $I5$ is deleted as illustrated in Figure 5.2(b), we can see that a's source value assigned by $I5$ can be recovered by the expression $r1 + r2$. Therefore $r1+r2$ becomes the new location expression of a for the deleted definition, instructions $I1$, $I2$ (both of which might define $r1$ used in $I5$) and $I4$ (which defines $r2$) become the actual definition points, and the definition type becomes *equivalent*. If instruction $I4$ is deleted later on during optimization as shown in Figure 5.2(c), since $r2$'s value can be recovered by $r6 + 4$ where $r6$ is defined by $I3$, the location expression of a becomes $r1+r6+4$ and $I3$ replaces $I4$ to become an actual definition point of the deleted definition of variable a. If instruction $I1$ is also deleted later on as shown in Figure 5.2(d), since $I1$ is not the only actual definition point which defines $r1$, we can see from Figure 5.2 that if the control reaches basic block C from basic block A, a's value can be recovered by the expression $r5 - 8 + r6 + 4$, whereas if the control reaches basic block C from basic block B, a's value should be recovered by the expression $r1 + r6 + 4$. Because we don't know the run-time

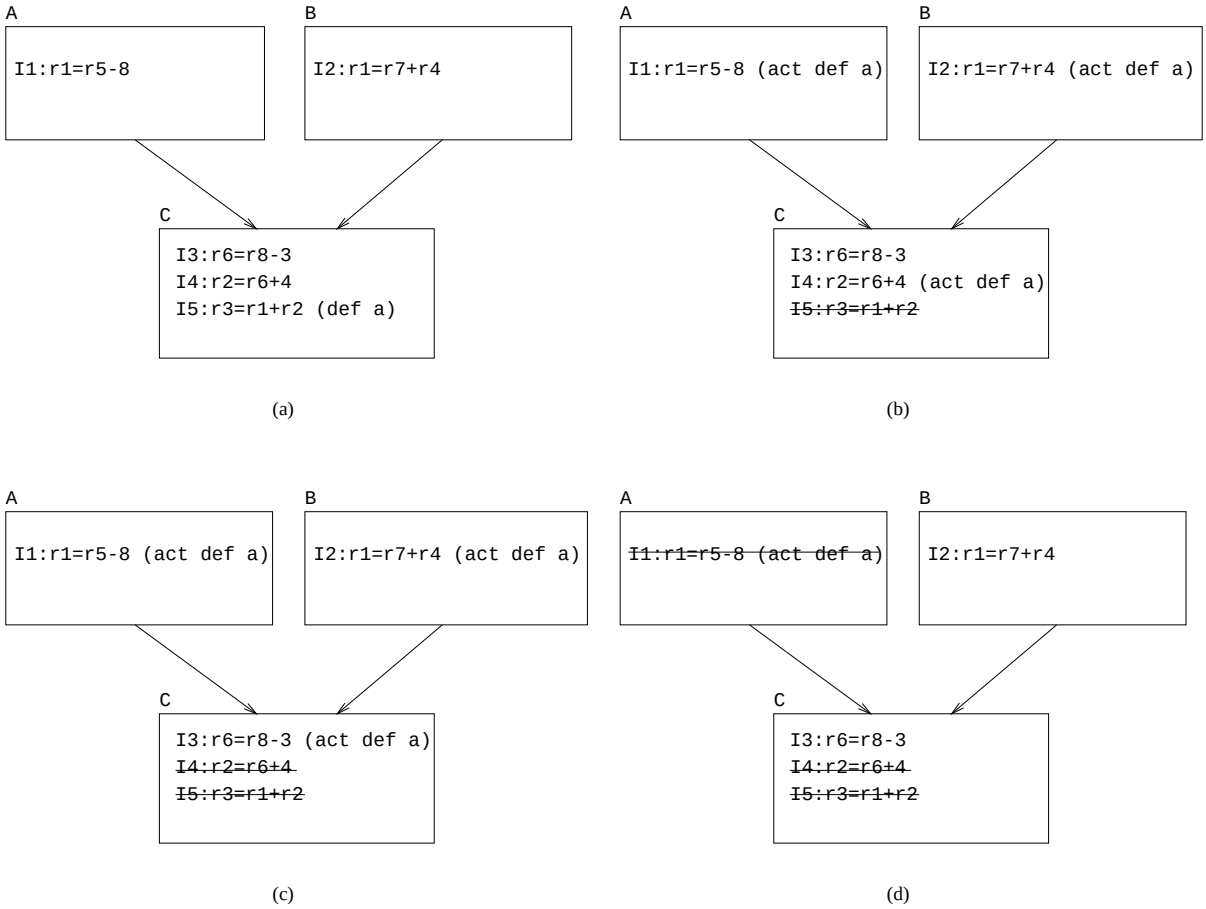


Figure 5.2 (a) Unoptimized code (b) Optimized code after $I5$ is deleted (c) Optimized code after $I4$ is deleted (d) Optimized code after $I1$ is deleted.

---

control flow at compile time and therefore cannot determine how to recover the value of a defined by $I5$, my current approach simply doesn't try to recover the value of a in this case and the type of this definition will be set to *deleted*.

If an instruction I which is an actual definition point of a virtual definition gets deleted, the compiler simply deletes the virtual definition, as it does not correspond to any source assignment.

4. *Code movement*

When an instruction I, which is an actual definition point of a definition D of variable V, is moved to a place *control-equivalent* to its original location,[2] nothing really needs to be done as far as variable definition information is concerned, as the definition is not deleted and the variable location is not changed. Instruction I remains to be an actual definition point of D. However, when instruction I is moved to a place which is not *control-equivalent* to its original location, the assignment to variable V might actually take place when it shouldn't have based on the source program semantics, or vice versa. To be conservative, non-control-equivalent code movement is treated as code deletion. I use the examples shown in Figure 5.3 to illustrate how non-control-equivalent code movement is handled. Figure 5.3(a) shows a program where instruction I, which is an actual definition point of one of variable V's definitions, is speculatively moved up to basic block A. This case is treated as if instruction I is deleted. Since the value of V defined by I can still be found in register $r4$ whose value is available at the original place of I, the location expression of this definition is $r4$, the type of the definition becomes *equivalent*, and I' becomes the actual definition point of the definition. Figure 5.3(b) shows another example where instruction I is moved down to basic block C due to partial dead code elimination. This case is also treated as if instruction I is deleted. If the values of both $r1$ and $r2$ are available at the original place of I, the variable location expression of the definition is $r1 + r2$, the type of the definition becomes

[2]Two machine locations are *control-equivalent* if one dominates and is post-dominated by the other.

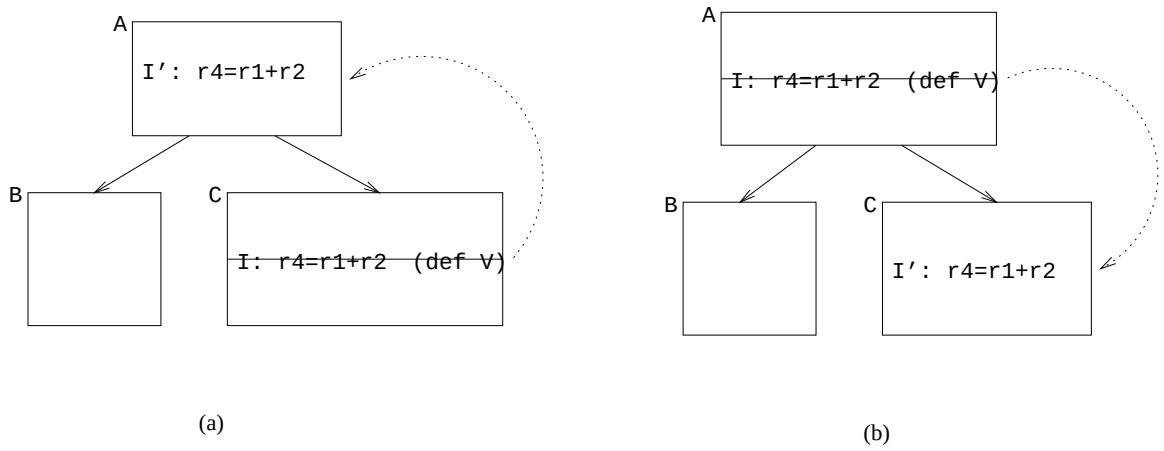


Figure 5.3 Code moved to a non-control-equivalent place (a) Speculative code motion (b) Partial dead code elimination.

equivalent, and the definitions of register $r1$ and $r2$ that can reach I's original place become the actual definition points.

Another issue regarding code movement is caused by the limitation of the proposed breakpoint implementation scheme. In my breakpoint scheme, for an actual definition point I of a definition D, if another instruction J with the same destination as I gets moved below I,[3] the value defined by I will never be seen at a source breakpoint in my recovery scheme because J will be a pre-breakpoint instruction whenever I is a pre-breakpoint instruction with regard to a source breakpoint and J will always be executed later than I. Definition D is practically deleted in this case, so D's type will be changed to *deleted*.

[3]This can occur when I and J originally assign to different virtual registers. The code movement occurs before register allocation. And then both registers are assigned to the same physical register.

5.2 Available Expected Variable-Location Pair Analysis

Range information is based on the variable definition information. For a source variable, a range record is created for each of its definitions which are not deleted. Intuitively, a range record corresponding to a variable definition should start from the actual definition point and ends at the place where the variable is redefined or the variable location is killed. However, for a source assignment of a variable, the actual definition point(s) might get moved around or even deleted during optimization, hence the original source ordering between the assignment and a source breakpoint may not hold in the optimized code, which, in turn, causes the problem illustrated in Figure 5.1. Therefore, instead of starting from the actual definition points as previous approaches did [10, 13], range records in my scheme start from the *effective definition points* of variable definitions. For a non-virtual definition of variable V, an effective definition point of the definition is the object code location from which the corresponding source assignment should take effect based on the semantics of the original program. Since the anchor point information (see Section 3.1) preserves the reaching conditions and execution order of the original source statements, for a definition D, which corresponds to source assignment statement S, of variable V, I use the anchor point(s) of source assignment S as the effective definition point(s) of D. At debug time, when user requests for V's value at a source breakpoint, the debugger in my scheme will use the address of the anchor point of the source breakpoint to compare with V's range records (derived from effective definition points) to find

out the expected variable value. As for a virtual definition, since it does not correspond to any source assignment, its actual definition point is also its effective definition point.

To calculate the range information, the *available expected variable-location pair analysis*, which is basically extended from the variable residency analysis proposed by Adl-Tabatabai [3], needs to be first performed. The analysis is performed through a data-flow algorithm operated on *expected variable-location pairs*. An expected variable-location pair, $< V, L >$, contains a source variable name, V, and a variable location expression, L. We say an expected variable-location pair $< V, L >$ is *available* at an instruction I if, under my breakpoint implementation scheme, the expected value of variable V can be obtained (or recovered) from location expression L at the source breakpoints with I as an anchor point.

Before the data-flow algorithm is applied, for each definition D of variable V, the effective definition point is identified and annotated. If the type of D is *equivalent*, it is also necessary to check if all the operands of D's variable location expression are available at the effective definition point. If not, the type of D becomes *deleted*.

For each variable V, let $vl\_pair\_gen[I]$ be the set of expected variable-location pairs made available by instruction I and $vl\_pair\_kill[I]$ be the set of expected variable-location pairs killed by I. I also define $All\_Possible\_Loc\_Expr[V]$ to be the set of all the variable location expressions which have ever been assigned to V in the program.

Given an instruction I, if I is annotated as an effective definition point of the definition D of variable V, where D's variable location expression is L, then immediately after I, the expected value of V should be obtainable from L (if D's type is not *deleted*) under my

breakpoint implementation scheme. Hence, instruction I generates $< V, L >$. Meanwhile, all the source-level values defined by other definitions of V are no longer available after I. Also, for every variable V', the source-level values defined by the definitions with L as their variable location expression are no longer available after I because, effectively, L will be holding a source-level value of V after I.

If location L is a destination of instruction I, for every variable V, the source-level value obtained from every location expression E in $All\_Possible\_Loc\_Expr[V]$ that has L as an operand is no longer available after I because L is redefined by I. However, if I is an actual definition point of an *original* definition D of a variable, no source-level values are killed by I as they will be killed by D's effective definition point. Refer to the example shown in Figure 5.4. Figure 5.4(a) shows an unoptimized code and Figure 5.4(b) shows an optimized code where instruction $I7$ is moved up after optimization. Assume the effective definition point of the second definition of variable b is $I8$. The expected variable-location pair $< a, r1 >$ generated by $I1$ will be killed at $I3$ because $r1$ is re-defined by $I3$. However, the expected variable-location pair $< b, r2 >$ generated by $I4$ will not be killed by $I7'$ because $I7'$ is the actual definition point of an original definition of b. Instead, this variable-location pair will be killed at the effective definition point of b's second definition, $I8$. We can see that if the expected variable-location pair $< b, r2 >$ generated by $I4$ was killed by $I7'$, there would be no expected variable-location pair of b available at instruction $I6$. However, it is desirable for the variable-location pair $< b, r2 >$ to still be available at $I6$ as the forward recovery scheme in my debugging

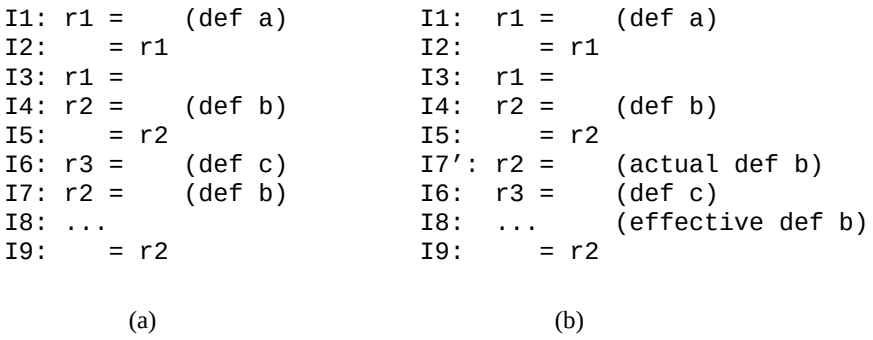


Figure 5.4 (a) Unoptimized code (b) Optimized code.

framework would provide a correct value of $r2$ even though $r2$ is modified prematurely by $I7'$.

If instruction I is a function call, since function calls might reuse caller-saved registers, for every variable V, every location expression L in $All\_Possible\_Loc\_Expr[V]$ that has a caller-saved register as an operand should be assumed to no longer hold the expected source-level value of V after the function call. That is, $< V, L >$ should be killed by I.

Therefore, based on the above intuition, given an instruction I, $vl\_pair\_gen[I]$ and $vl\_pair\_kill[I]$ are constructed using the following rules:

Rule 1 : If I is annotated as an effective definition point of the definition D of variable V, where D's variable location expression is L,

1.1 $< V, L > \in vl\_pair\_gen[I]$ if D's type is not *deleted*

1.2 $\forall L' \in All\_Possible\_Loc\_Expr[V], < V, L' > \in vl\_pair\_kill[I]$

1.3 $\forall V' where L \in All\_Possible\_Loc\_Expr[V'], < V', L > \in vl\_pair\_kill[I]$

Rule 2 : If I defines location L and I is not an actual definition point of an *original* definition of any variable, for every expected variable-location pair $< V, E >$ where E is in $All\_Possible\_Loc\_Expr[V]$ and has L as an operand,

- $< V, E > \in vl\_pair\_kill[I]$

Rule 3 : If I is a function call, for every expected variable-location pair $< V, L >$ where L is in $All\_Possible\_Loc\_Expr[V]$ and has a caller-saved register as an operand,

- $< V, L > \in vl\_pair\_kill[I]$

An expected variable-location pair $< V, L >$ is available at I only when it is available at the points immediately after all of I's predecessor, therefore the *in* set for instruction I is

$$vl\_pair\_in[I] = \bigcap_{J\ is\ a\ predecessor\ of\ I} vl\_pair\_out[J]$$

And the *out* set is

$$vl\_pair\_out[I] = vl\_pair\_gen[I] \cup (vl\_pair\_in[I] - vl\_pair\_kill[I])$$

The standard iterative algorithms for solving data-flow equations [22] can be used to find out the vl_pair_in set and vl_pair_out set for each instruction.

To illustrate how the proposed data-flow analysis works, consider the example shown in Figure 5.5 (this example will be used throughout the rest of the chapter). Figure 5.5(a) shows the unoptimized code where instruction $I1$ from statement $S1$ and instruction $I16$ from statement $S7$ are source definitions of variable a, instruction $I2$ from statement

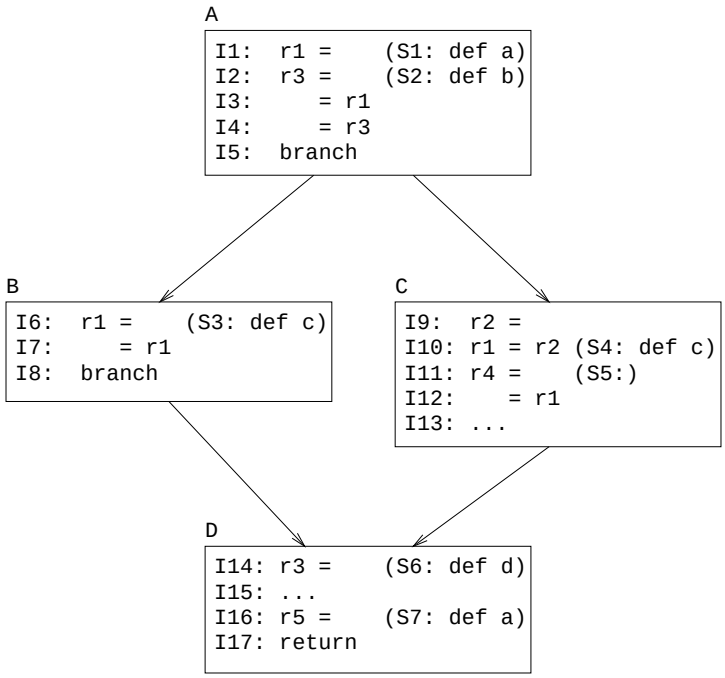
$S2$ is a source definition of variable b, instruction $I6$ from statement $S3$ and instruction $I10$ from statement $S4$ are both source definitions of variable c, and instruction $I14$ from statement $S6$ is a source definition of variable d. I also assume the anchor points for statement $S1$, $S2$, $S3$, $S4$, $S5$, $S6$, and $S7$ are instruction $I1$, $I2$, $I6$, $I10$, $I11$, $I14$, and $I16$, respectively. Figure 5.5(b) shows the optimized code after the following transformations:

1. Instruction $I6$ which is a definition of source variable c is moved up to basic block A as a result of speculative code motion and becomes $I6'$. Since this is a non-control-equivalent code motion, the definition type becomes *equivalent*. However, the location expression of the definition is still $r1$. (See Section 5.1.) Instruction $I7$ becomes the new anchor point for statement $S3$.

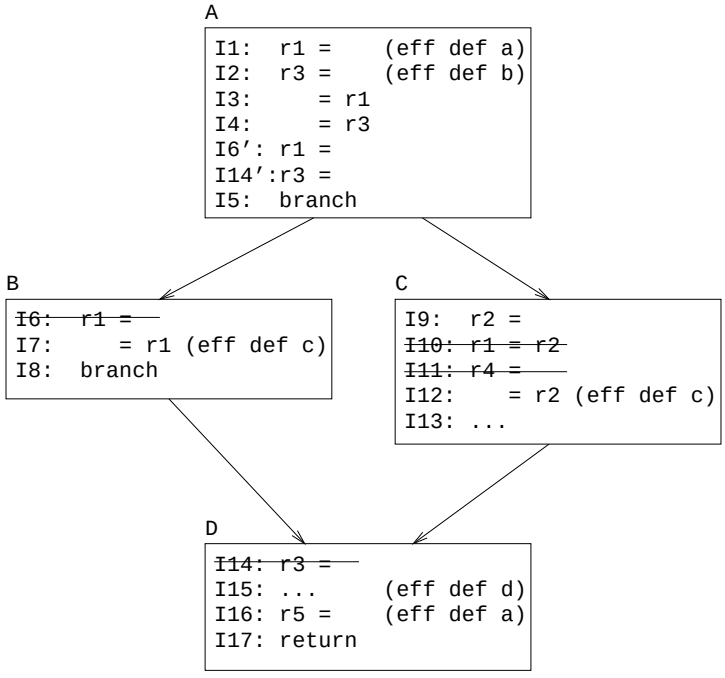
2. Instruction $I10$, another definition of variable c, is deleted after copy propagation (and the use of $r1$ in instruction $I12$ is changed to $r2$). The definition type becomes *equivalent* and $r2$ becomes the new location expression. Also instruction $I11$ is deleted (assuming it is dead). Instruction $I12$ becomes the anchor point for both statement $S4$ and statement $S5$.

3. Instruction $I14$, a definition of variable d, is moved up to basic block A which is control equivalent to basic block D and becomes $I14'$. Instruction $I15$ becomes the anchor point for statement $S6$.

The variable definition information for each variable after optimization is summarized in Table 5.1.



(a)



(b)

Figure 5.5 (a) Unoptimized code (b) Optimized code.

Table 5.1 Variable definition information for variables in the code example shown in Figure 5.5.

Variable	Definition Type	Location Expression	Actual Definition Point(s)
a	Original	r1	I1
	Original	r5	I16
b	Original	r3	I2
c	Equivalent	r1	I6'
	Equivalent	r2	I9
d	Original	r3	I14'

The effective definition point for each source variable definition in the example program is identified based on the anchor point information and annotated as shown in Figure 5.5(b). The *All_Possible_Loc_Expr* sets for the source variables are as follows:

$$All\_Possible\_Loc\_Expr[a] = \{r1, r5\}$$

$$All\_Possible\_Loc\_Expr[b] = \{r3\}$$

$$All\_Possible\_Loc\_Expr[c] = \{r1, r2\}$$

$$All\_Possible\_Loc\_Expr[d] = \{r3\}$$

Here I assume the non-branch instructions without destinations denoted in the example (namely, instruction $I3$, $I4$, $I7$, $I12$, $I13$, and $I15$) do not affect variables a, b, c, and d. Thus, the destinations of these instructions do not appear in the *All_Possible_Loc_Expr* set of any variable. I also assume the branch instructions $I5$ and $I8$ as well as the return instruction $I17$ do not write to any general registers or memory locations.

vl_pair_gen and *vl_pair_kill* sets for each instruction are then calculated based on the rules described above. We can see in Figure 5.5(b) that instruction $I1$ is annotated

as an effective definition point of a definition of variable a with location expression $r1$. Therefore, from Rule 1.1, $< a, r1 >$ should be in $vl\_pair\_gen[I1]$. From Rule 1.2, both $< a, r1 >$ and $< a, r5 >$ should be in $vl\_pair\_kill[I1]$. Also from Rule 1.3, $< c, r1 >$ should be in $vl\_pair\_kill[I1]$. That is,

$$vl\_pair\_gen[I1] = \{< a, r1 >\}$$

$$vl\_pair\_kill[I1] = \{< a, r1 >, < a, r5 >, < c, r1 >\}$$

Note that since $I1$ is an actual definition point of an *original* definition and is not a function call, Rule 2 and Rule 3 don't apply here. Similarly, the vl_pair_gen and vl_pair_kill sets for instruction $I2$, $I7$, $I12$, $I15$, and $I16$, all of which are annotated as an effective definition point of some variable definition, can be calculated in the same fashion.[4]

Instruction $I6'$ and $I9$ are not denoted as an effective definition point of any source definition. Since neither of them is an actual definition point of an *original* definition (note that they are actual definition points of *equivalent* definitions), from Rule 2, we know $vl\_pair\_kill[I6']$ should contain $< a, r1 >$ and $< c, r1 >$, while $vl\_pair\_kill[I9]$ should contain $< c, r2 >$. Neither of them generates any variable-location pair. Hence,

$$vl\_pair\_gen[I6'] = \phi$$

$$vl\_pair\_kill[I6'] = \{< a, r1 >, < c, r1 >\}$$

$$vl\_pair\_gen[9] = \phi$$

[4]Some of these effective definition points (such as $I7$, $I12$, and $I15$) are not actual definition points and therefore we will need to apply Rule 2. However, since it is assumed that their destinations are not in the *All_Possible_Loc_Expr* set of any variable, no variable-location pair will be added to the *vl_pair_kill* set based on Rule 2.

$$vl\_pair\_kill[I9] = \{< c, r2 >\}$$

Instruction $I14'$ is not an effective definition point of a source definition either. Since it is an actual definition point of an *original* definition of variable d, Rule 2 does not apply (as discussed and illustrated in Figure 5.4). Therefore both the *vl_pair_gen* set and *vl_pair_kill* set of $I14'$ are empty.

Instruction $I3$, $I4$, $I5$, $I8$, $I13$, and $I17$ are not effective definition points and they don't define any registers and memory locations which constitute the location expressions of any user variable, therefore the *vl_pair_gen* sets and *vl_pair_kill* sets for these instructions are all empty.

With these *vl_pair_gen* sets of *vl_pair_kill* sets, the *vl_pair_in* set and *vl_pair_out* set for each instruction are calculated using the standard iterative algorithm for solving data-flow equations. The *vl_pair_gen*, *vl_pair_kill*, *vl_pair_in*, and *vl_pair_out* sets for each instruction in the example are shown in Table 5.2.

Note that the representation of the expected variable-location pair in the data-flow analysis is the same as that of the *residence* described in Adl-Tabatabai's variable residency analysis [3], but they have totally different meanings. As mentioned earlier, an expected variable-location pair $< V, L >$ is *available* at an instruction I if, under my breakpoint implementation scheme, the expected value of variable V can be recovered from location expression L at the source breakpoints with I as an anchor point, while a residence, $< V, L >$, being available at a given point in the object code only means a source-level value of V (not necessarily expected) can be found in L at the object loca-

Table 5.2 *vl_pair_gen*, *vl_pair_kill*, *vl_pair_in*, and *vl_pair_out* sets for each instruction in the code example shown in Figure 5.5.

Instruction	*vl_pair_gen*	*vl_pair_kill*	*vl_pair_in*	*vl_pair_out*
I1	$< a, r1 >$	$< a, r1 >, < a, r5 >, < c, r1 >$	ϕ	$< a, r1 >$
I2	$< b, r3 >$	$< b, r3 >, < d, r3 >$	$< a, r1 >$	$< a, r1 >, < b, r3 >$
I3	ϕ	ϕ	$< a, r1 >, < b, r3 >$	$< a, r1 >, < b, r3 >$
I4	ϕ	ϕ	$< a, r1 >, < b, r3 >$	$< a, r1 >, < b, r3 >$
I6'	ϕ	$< a, r1 >, < c, r1 >$	$< a, r1 >, < b, r3 >$	$< b, r3 >$
I14'	ϕ	ϕ	$< b, r3 >$	$< b, r3 >$
I5	ϕ	ϕ	$< b, r3 >$	$< b, r3 >$
I7	$< c, r1 >$	$< c, r1 >, < c, r2 >, < a, r1 >$	$< b, r3 >$	$< b, r3 >, < c, r1 >$
I8	ϕ	ϕ	$< b, r3 >, < c, r1 >$	$< b, r3 >, < c, r1 >$
I9	ϕ	$< c, r2 >$	$< b, r3 >$	$< b, r3 >$
I12	$< c, r2 >$	$< c, r1 >, < c, r2 >$	$< b, r3 >$	$< b, r3 >, < c, r2 >$
I13	ϕ	ϕ	$< b, r3 >, < c, r2 >$	$< b, r3 >, < c, r2 >$
I15	$< d, r3 >$	$< d, r3 >, < b, r3 >$	$< b, r3 >$	$< d, r3 >$
I16	$< a, r5 >$	$< a, r1 >, < a, r5 >$	$< d, r3 >$	$< d, r3 >, < a, r5 >$
I17	ϕ	ϕ	$< d, r3 >, < a, r5 >$	$< d, r3 >, < a, r5 >$

tion. Also, the storage location of a residence can only be a register or a stack frame slot.

5.3 Range Calculation

A range record of a variable basically consists of a start address, an end address, and a variable location expression. Once the available expected variable-location pair analysis is done, the range information can be easily built based on the availability of expected variable-location pairs. Basically, a new range record for variable V with variable location expression L starts from instruction I if $< V, L >$ is first seen in *vl_pair_out*$[I]$, and ends at instruction J if $< V, L >$ is no longer in *vl_pair_out*$[J]$ or if J is a program exit point

or a function return. The detailed algorithm using the data-flow information to calculate the range information is shown in Figure 5.6.

This algorithm steps through each instruction on the final code layout to determine the start and end addresses of the range records of all variables. For each variable, two book-keeping variables, $CurrentRangeRecord$ which tracks the currently identified range record, and $CurrentVarLocPair$ which tracks the variable location pair corresponding the currently identified range record, are kept during the range record calculation. They are initialized to *nil* before the calculation. To explain how the algorithm works, I will again use the code example shown in Figure 5.5. The code layout for the example program is shown in Figure 5.7(a). Although this algorithm calculates the range records of all the variables in one single pass through the code, for simplicity, only variable a will be discussed here. We start from instruction $I1$. $CurrentVarLocPair[a]$, which is initialized to *nil*, is obviously not in either $vl\_pair\_in[I1]$ or $vl\_pair\_out[I1]$. However, from Table 5.2 we can see that $< a, r1 >$ is in $vl\_pair\_out[I1]$. Therefore, a new range record (say $RR1$) is identified. Its starting address is `2000` and its location is $r1$. $CurrentRangeRecord(a)$ is set to $RR1$ and $CurrentVarLocPair(a)$ is set to $< a, r1 >$. We then continue to process the next instruction, $I2$. Since $CurrentVarLocPair(a)$ (i.e. $< a, r1 >$) appears in both the $vl\_pair\_in[I2]$ and $vl\_pair\_out[I2]$ (see Table 5.2), nothing needs to be done. The same goes for instruction $I3$ and $I4$. When we move on to instruction $I6'$, we find out $CurrentVarLocPair(a)$ is in $vl\_pair\_in[I6']$, but not in $vl\_pair\_out[I6']$. Therefore $CurrentRangeRecord(a)$ ($RR1$) ends at instruction $I6'$. That is, $RR1$'s end address is `2016`. Both $CurrentRangeRecord(a)$ and $CurrentVarLocPair(a)$ are reset to *nil*. Since

```
/* initialization */
for each variable V {
    CurrentVarLocPair(V) = nil
    CurrentRangeRecord(V) = nil
}

Starting from the first instruction of the function,
for each instruction I {
    for each variable V {
        if (CurrentVarLocPair(V) is not in vl_pair_in[I]) {
          if (CurrentVarLocPair(V) ≠ nil) {
            CurrentRangeRecord(V).EndAddr = AddrOf(PreviousInstr(I))
            CurrentVarLocPair(V) = nil
            CurrentRangeRecord(V) = nil
          }
          if (a variable-location pair < V, L > of variable V is found in vl_pair_in[I]) {
            CurrentVarLocPair(V) = < V, L >
            CurrentRangeRecord(V) = NewRangeRecord()
            CurrentRangeRecord(V).StartAddr = AddrOf(I)
            CurrentRangeRecord(V).Location = L
          }
        }
        if (CurrentVarLocPair(V) is not in vl_pair_out[I]) {
          if (CurrentVarLocPair(V) ≠ nil) {
            CurrentRangeRecord(V).EndAddr = AddrOf(I)
            CurrentVarLocPair(V) = nil
            CurrentRangeRecord(V) = nil
          }
          if (a variable-location pair < V, L > of variable V is found in vl_pair_out[I]) {
            CurrentVarLocPair(V) = < V, L >
            CurrentRangeRecord(V) = NewRangeRecord()
            CurrentRangeRecord(V).StartAddr = AddrOf(I)
            CurrentRangeRecord(V).Location = L
          }
        }
        if (I is an exit point or a function return) {
          if (CurrentVarLocPair(V) ≠ nil) {
            CurrentRangeRecord(V).EndAddr = AddrOf(I)
            CurrentVarLocPair(V) = nil
            CurrentRangeRecord(V) = nil
          }
        }
    }
}
```

Figure 5.6 Range calculation algorithm.

no variable-location pair of a is found in $vl\_pair\_out[I6']$, $CurrentRangeRecord(a)$ and $CurrentVarLocPair(a)$ remain to be *nil* when we come to process the next instruction. Because no variable-location pair of a is found in the *vl_pair_in* and *vl_pair_out* sets of instruction $I14'$, 5, $I7$, $I8$, $I9$, $I12$, 13 and 15, we therefore don't have to do anything (as far as the range information of variable a is concerned) until instruction $I16$ is being processed. We can see that $< a, r5 >$ first appears in $vl\_pair\_out[I16]$, therefore another range record (say $RR2$) is identified. Its starting address is `2052` and its location is $r5$. $CurrentRangeRecord(a)$ is set to $RR2$ and $CurrentVarLocPair(a)$ is set to $< a, r5 >$. We then process instruction $I17$. Since it is a function return instruction, $CurrentRangeRecord(a)$ ($RR2$) ends here. That is, $RR2$'s end address is `2056`. The range information for all the variables is shown in Figure 5.7(b).

Note that in the algorithm, the range record RR built for a definition D (which corresponds to a source assignment S) of variable V starts from the effective definition point I of D, instead of the instruction immediately following I. The reason why instruction I is included in range record RR is because I, which is an anchor point of S, might also serve as an anchor point of other source statements. Some of these statements might be expected to happen before assignment S in the original source code and some of them might be expected to happen later than S. Refer back to the example shown in Figure 5.5. After both instruction $I10$ and $I11$ are deleted, instruction $I12$ becomes the anchor point for both statement $S4$ (a source assignment of variable c) and statement $S5$. Statement $S5$ is expected to happen after statement $S4$. For a source breakpoint set at $S4$, when the debugger compares the address of $I12$ (an anchor point of $S4$) with

```
  address
A: 2000   I1:  r1 =        (start of RR1)
   2004   I2:  r3 =
   2008   I3:     = r1
   2012   I4:     = r3
   2016   I6': r1 =        (end of RR1)
   2020   I14':r3 =
   2024   I5:  branch B, C
B: 2028   I7:     = r1
   2032   I8:  branch D
C: 2036   I9:  r2 =
   2040   I12:    = r2
   2044   I13: ...
D: 2048   I15: ...
   2052   I16: r5 =        (start of RR2)
   2056   I17: return      (end of RR2)
```

(a)

Variable	Start Addr	End Addr	Loc	Type
a	2000	2016	r1	register
	2052	2056	r5	register
b	2004	2048	r3	register
c	2028	2032	r1	register
	2040	2044	r2	register
d	2048	2056	r3	register

(b)

Figure 5.7 (a) Code layout (b) Range information.

---

the range records of variable c, based on the original source program, the range record corresponding to source assignment $S4$ should not cover $I12$ as the value defined by $S4$ should not be seen at the breakpoint set at $S4$. However, for a source breakpoint set at $S5$, we would like the range record to cover $I12$ as the value defined by $S4$ should be seen at $S5$. Without the knowledge of where the breakpoints will be set in advance, I always include $I12$ in the range record (which, in turn, might cause this range record to overlap with the previous range). I also extend range information to include the source ordering

information of the corresponding source assignments.[5] At debug-time, the debugger will use the source ordering information of the range records to determine whether a range record actually covers a source breakpoint if the anchor point of the breakpoint falls within the range record, or which range record covers the breakpoint if the anchor point falls within two overlapping ranges. In our example, when the user sets a breakpoint at $S4$, although the anchor point of $S4$ ($I12$) falls within the second range record of variable c (see Figure 5.7(b)), after comparing the source ordering of the range record with that of the breakpoint, the debugger knows this range record should not cover the breakpoint and will report to the user that no expected value of variable c can be provided. This is consistent with what the user would expect from the unoptimized code. On the other hand, when the user sets a breakpoint at $S5$, by comparing the source ordering information of the breakpoint and the range record, the debugger knows that the second range record of variable c indeed covers the breakpoint. It will then retrieve the value in $r2$ and report it to the user.

[5]For a range record built for a virtual definition, there will be no source ordering information.

CHAPTER 6

EMPIRICAL EVALUATIONS

The compiler support for the proposed debugging framework is implemented within the framework of the IMPACT compiler [26]. A prototype source-level debugger, *idb*, which incorporates the new breakpoint implementation scheme is also designed and implemented. The evaluations presented in this chapter were conducted based on the fully-implemented compiler support and prototype debugger. In this chapter, I first describe the experimental framework used for the empirical evaluations. The overhead in compile time and executable file size due to the proposed debugging framework is then presented. The cost incurred in setting and reporting source-level breakpoints under the new debugging framework as well as the effectiveness of the approach are also quantitatively evaluated.

6.1 Experimental Framework

All the evaluations presented in this chapter were conducted on HP's PA-RISC based workstations running HP-UX 10.20. Six integer C programs from the SPEC95 benchmark suite were used for the experiments. These benchmark programs are summarized in Table 6.1. In the evaluations regarding the cost of the new breakpoint implementation scheme and the effectiveness of the proposed framework, experimental data were collected

Table 6.1 Benchmark descriptions.

Benchmark	Description	Total possible source breakpoints
124.m88ksim	Simulator for the 88100 microprocessor	6082
129.compress	Performs adaptive Lempel-Ziv coding	358
130.li	Lisp interpreter	2529
132.ijpeg	Image compression and decompression	7871
134.perl	Perl interpreter	12434
147.vortex	Single-user object-oriented database transaction benchmark	21687

for every possible source breakpoint (that is, every source line which contains a source statement) in the benchmark programs. The last column of Table 6.1 shows the total number of all possible source breakpoints in each benchmark program.

6.1.1 Compilation environment

The benchmark programs were first compiled and optimized by the IMPACT compiler. The generated assembly files were then assembled and linked by *gcc* to generate the executables. The IMPACT compiler is a retargetable, optimizing C compiler being developed at the University of Illinois. Various compilation paths within the IMPACT compiler can be chosen to study different compilation techniques, architecture features, and compiler/architecture tradeoffs for ILP processors. In my dissertation work, an optimizing compilation path in the IMPACT compiler was selected to prototype the compiler support for the debugging framework.

On this prototyped compilation path, C source code is first translated to the highest level IR, *Pcode* [27], which is a hierarchical representation of the C source with source-level constructs such as loops and if-statements intact. Basically all the source code informa-

tion required for the debugging support is well preserved at the Pcode Level. Pcode files are translated to the middle level IR, *Hcode*, which is a flattened C representation with simple if-then-else and go-to control flow constructs. Sequence numbers are first assigned to basic blocks at the Hcode level. Hcode files are then converted to the low-level IR called *Lcode*. Lcode is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. At the Lcode level, machine-independent optimizations are first applied. These include the classic local, global, loop, and jump (branch) optimizations described as follows [28].

Local optimizations : constant propagation, forward/backward copy propagation, memory copy propagation, common subexpression elimination, redundant load/store elimination, constant combining/folding, strength reduction, logic reduction, operation folding, operation cancellation, code reordering, dead code removal, and register renaming.

Global optimizations : constant/copy propagation, memory copy propagation, common subexpression elimination, redundant load/store elimination, dead code removal, and dead if-then-else removal.

Loop optimizations : loop invariant code removal, loop back branch simplification, global variable migration (global register promotion), induction variable strength reduction, induction variable elimination, and dead loop removal.

Jump optimizations : dead block removal, branch-to-next-block elimination, combining branches to the same target, branch-to-unconditional-branch combining, merg-

ing always successive blocks, label combining, branch target expansion, and branch swapping.

After machine-independent optimization, machine-dependent optimization and code generation for HP PA-RISC architecture are performed. The machine-dependent optimizations include acyclic scheduling, register allocation, and peephole optimization. Acyclic scheduling [29], [30] is applied both before register allocation (prepass scheduling) and after (postpass scheduling) to generate an efficient schedule. The IMPACT global register allocator [31] is based on the graph-coloring algorithm described in [32]. When possible, the register allocator tries to minimize the number of registers used so that the number of registers that need to be saved and restored at procedure call boundaries can be reduced. At various points in the code generation process, a set of specially tailored peephole optimizations are performed. These peephole optimizations are designed to remove inefficiencies introduced during the early phase of code generation, to take advantage of specialized opcodes available in the architecture, and to remove inefficient code inserted by the register allocator. While instruction source ordering information, anchor point information, and variable definition information are generated and continuously maintained throughout every code optimization stage, the range information is calculated after all the optimizations (including register allocation) have been performed.

6.1.2 Prototype debugger

The prototype debugger, *idb*, is a command-line based, source-level interactive debugger which is implemented on HP's PA-RISC based workstations running HP-UX 10.20.

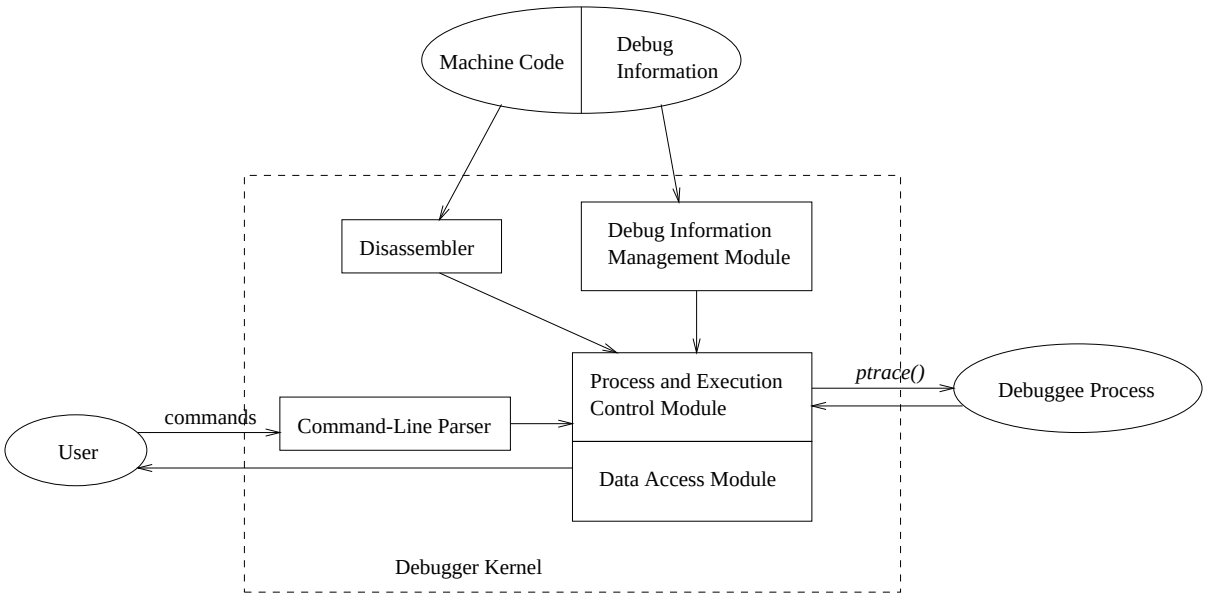


Figure 6.1 The architecture of the prototype debugger.

It supports functionalities commonly seen in most of the current commercial symbolic debuggers such as running and continuing a program, setting and deleting source breakpoints, inquiry of variable values at breakpoints, etc. The architecture of the prototype debugger is shown in Figure 6.1.

When debugging a program, the *disassembler* of the debugger is first invoked to disassemble the object code. The disassembled program is represented in the *Lcode* format (see previous subsection) so that the routines written for the debugger (such as the calculation of interception points, finish points, and escape points) can also be used by the compiler, which greatly reduces the complexity of my static empirical evaluations. Also the time to develop the debugger was reduced by exploiting some of the control-flow and data-flow routines already developed and tested on the compiler side.

The *debug information management module* reads in the debug information propagated from the compiler and builds internal data structures such as variable and function symbol tables, source statement tables (including anchor point information), variable type and range information, etc. The debug information management module also annotates the instructions of the disassembled program with the source ordering information.

In response to the user's debugging commands (passed from the *command-line parser*), the *process and execution control module* controls the creation and termination of the debuggee process, running and continuing the program, as well as setting and deletion source breakpoints. At the core of the process and execution control module is the proposed breakpoint implementation scheme. For a source breakpoint, the module calculates the corresponding interception points, finish points, and escape points using the anchor point and the source ordering information. It takes over control at the interception point and starts forward recovery as described in Chapter 4.

When the user requests the value of a variable at a source breakpoint, the *data access module* uses the range information to find out whether the variable value exists or not and, if it does, where to obtain it. It then accesses the storage locations (register file or memory) from debuggee's address space and presents the value to the user with the help of the variable type information. The debugger controls the debuggee process and accesses debuggee's address space all through UNIX *ptrace* system call.

6.2 Overhead in Compile Time and Executable File Size

Besides the traditional debug information, the extra information that the compiler needs to emit for the proposed debugging framework includes the source ordering of instructions, the anchor point information (which actually replaces the source line table in the traditional debuggers), and the range information which records the run-time locations of variables. Table 6.2 summaries the experimental results for the increase in executable file size due to the extra debug information.

Row 1 of Table 6.2 shows the size of the stripped executable file for each of the optimized SPEC95 programs without any debug information. Row 2 shows the increase in executable file size (in byte and in percentage) for source ordering information. Row 3 and Row 4 show the same evaluation for anchor point information and range information. The average increases in executable file size for the source ordering information, anchor point information, and range information are 38%, 62%, and 282%, respectively. On average, the file size increase for all the debug information (including traditional and extra information) is about 406% as shown in Row 5. In comparison, the average executable file size increase due to the debug information when the benchmark programs were compiled by *gcc* (without optimization) is around 319%. While the size of debug information may vary with different sets of optimizations performed, the results presented in Table 6.2 can nonetheless demonstrate that the overhead in executable file size introduced by my approach is acceptable.

Table 6.2 The size of the debug information for six optimized SPEC95 programs.

Program		*m88ksim*	*compress95*	*li*	*ijpeg*	*perl*	*vortex*
Executable file size in byte (stripped)		294944	98328	147485	323616	536618	938032
Source ordering information	Size (byte)	127824	7452	56868	166332	261684	382044
	Inc. percentage	43%	8%	39%	51%	49%	41%
Anchor point information	Size (byte)	200640	11888	90552	260224	405320	721696
	Inc. percentage	68%	12%	61%	80%	76%	77%
Range information	Size (byte)	817200	27800	269272	851760	2722828	4042368
	Inc. percentage	277%	28%	183%	263%	507%	431%
All debug information	Size (byte)	1215840	52424	468144	1386440	3475000	5391088
	Inc. percentage	412%	53%	317%	428%	648%	575%

I also noticed that compared to other programs, *compress95* has a relatively small percentage of the executable file size increase for the debug information. After examining the executable files, I found out that for *compress95* the code sections occupy only a small portion of the executable file (which contains a very big data section), while for all of the other programs, the code sections occupy about half the size of the executable files. It is easily understood that the percentage of the file size increase due to debug information will be small when the code sections occupy only a small portion of the executable file.

Table 6.3 summaries the compile time increased due to the compiler support for the proposed framework. Since the generation and maintenance of the necessary debug information are primarily taking place at the code optimization and code generation phases on the prototyped compilation path, I only measured the compile time increases for these two phases. On average, the compile time increases by about 38% at the optimization phase and by about 142% at the code generation phase.

Table 6.3 Compile time increase due to the debugging framework for six optimized SPEC95 programs.

Program	*m88ksim*	*compress95*	*li*	*ijpeg*	*perl*	*vortex*
Code optimization phase	41%	10%	41%	19%	87%	29%
Code generation phase	118%	36%	206%	123%	115%	251%

6.3 Overhead in Debug-Time Strategy

It is interesting to know how much cost is incurred in reporting source-level breakpoints under the new debugging framework. To answer this questions, my experiments were conducted statically by analyzing the optimized code and collecting data for every possible source breakpoint in the benchmark programs. First 4 rows of Table 6.4 show the average numbers of anchor points, interception points, finish points, and escape points for a source breakpoint.

In my framework, hitting of an interception point at run time starts forward recovery. During forward recovery, instructions are executed one by one (single-stepped) until a finish point or an escape point is reached. Row 5 of Table 6.4 shows the total number of instructions executed during forward recovery. These numbers were obtained statically by looking at all the possible paths from every interception point of a source breakpoint to either a finish point or an escape point. Although in some worst cases, hundreds of instructions might need to be executed during forward recovery, on average only about 2 to 25 instructions will need to be executed before the debugger can determine if a source breakpoint should be reported to the user or not. Note that if there is no code

Table 6.4 Results from static analysis on six optimized SPEC95 programs.

Program		*m88ksim*	*compress95*	*li*	*ijpeg*	*perl*	*vortex*
Average no. of anchor points per source breakpoint		1.05	1.06	1.05	1.07	1.09	1.03
Average no. of interception points per source breakpoint		1.17	1.22	1.07	1.22	1.19	1.08
Average no. of finish points per source breakpoint		1.12	1.20	1.09	1.21	1.27	1.06
Average no. of escape points per source breakpoint		0.28	0.21	0.03	0.33	0.44	0.14
Instructions executed per forward recovery	Average	10.76	15.00	2.13	24.91	8.07	13.35
	max	206	167	76	581	294	714
	min	0	0	0	0	0	0

being moved across a source breakpoint, the interception point and the finish point both coincide with the anchor point and no instruction is executed during forward recovery.

From the experience with the prototype debugger, the time to set a breakpoint under my proposed scheme was only slightly longer than that required by GDB's regular breakpoint, and the extra time is hardly noticeable by a human user. Furthermore, as the average number of instructions executed during forward recovery is only around 12 (based on the measurements shown in Table 6.4), even though forward recovery is time consuming, most of the time it did not add an overhead that is noticeable in an interactive debugging situation.

6.4 Effectiveness of The Framework

To show the effectiveness of my approach, I first measured the effects of global optimization on the data value problem. The stacked columns labeled *base* in Figure 6.2

show the average numbers of local variables that are *uninitialized*, *current*, and *non-current* at a source breakpoint for the optimized benchmark program without applying any recovery scheme. As discussed in Chapter 2, a variable is *current* at a breakpoint if its value is consistent with what the user expects from the original source program at this breakpoint [8]. Otherwise, it is *non-current*.[1] A variable is considered *uninitialized* at a source breakpoint if the variable is not defined on any path leading to the source breakpoint from the function entry point. Again, the results were obtained by collecting data for every possible source breakpoint. Since most of the global variables were found to be current at source breakpoints based on my experiments,[2] I only present the results for local variables. We can see from Figure 6.2 that, on average, about 10-33% of the local variables in scope are non-current at a source breakpoint without recovery.

In contrast, the stacked columns labeled *proposed scheme* in Figure 6.2 show the results under the proposed debugging framework. On average only about 7% of the local variables in scope are non-current.

Table 6.5 further illustrates how often the expected values of non-current variables can be recovered by my debugging framework. Row 1 shows the average number of local variables that are non-current at each source breakpoint. These numbers correspond to the top portion of the stacked columns labeled *base* shown in Figure 6.2. As mentioned before, a variable can be made non-current at a source breakpoint because of the following optimizations:

[1] *Non-current* variables in my classification actually includes both *endangered* and *nonresident* variables defined in some previous work [8, 15]

[2] The fact was also observed and noted by other research work [3].

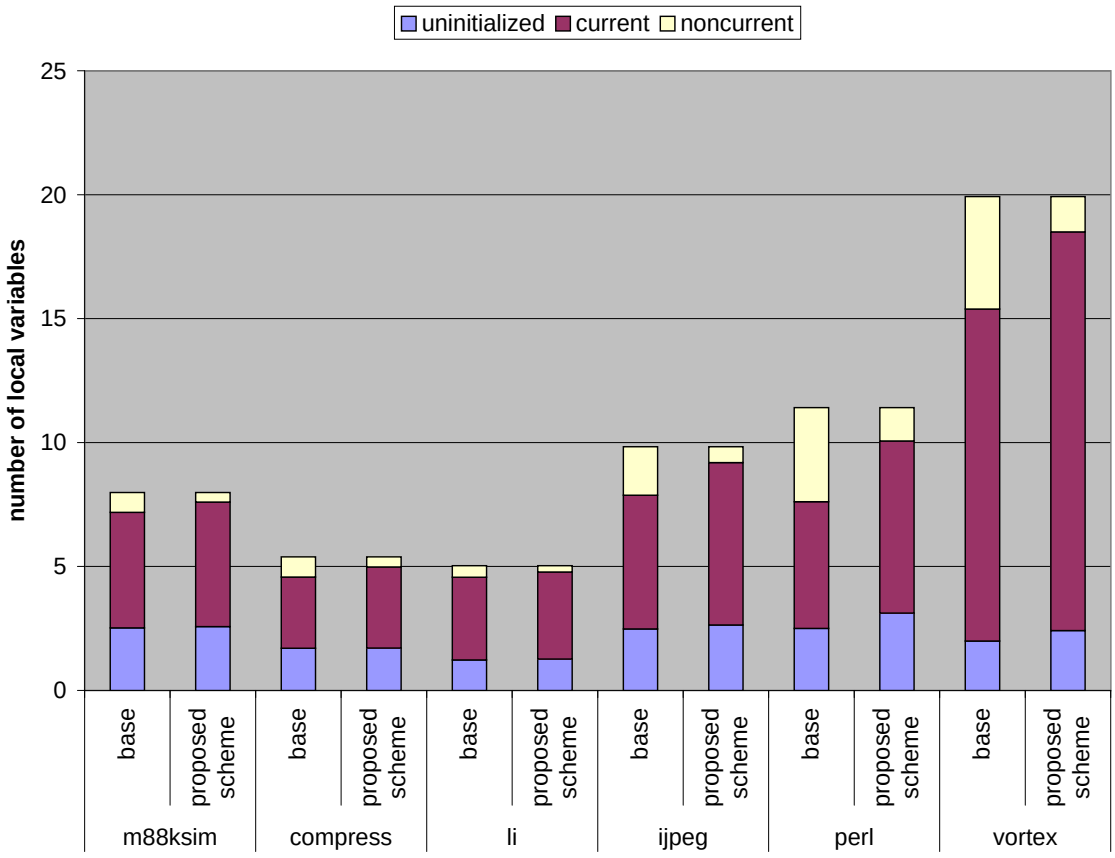


Figure 6.2 Average number of local variables in scope at each source breakpoint.

1. An assignment to the variable is moved across the breakpoint.

2. An assignment to the variable which can reach the breakpoint is deleted.

3. The storage location of the variable is reused due to register allocation, which in turn makes the source-level value of the variable non-existent at the breakpoint.

Row 2 of Table 6.5 shows the average number of the non-current variables that are caused by code reordering and code deletion. Row 3 shows the average number of non-current variables whose expected values can be recovered by my debugging framework. On average, my approach can recover around 58% of all non-current variables, as shown

Table 6.5 Effectiveness of the proposed debugging framework in the recovery of the expected values for non-current local variables.

Program	*m88ksim*	*compress95*	*li*	*ijpeg*	*perl*	*vortex*
non-current variables	0.796777	0.812849	0.467774	1.958836	3.799662	4.537926
non-current variables caused by code reordering and deletion	0.486518	0.452514	0.231712	1.6961	3.442255	3.398626
non-current variables recoverable	0.414173	0.405028	0.213523	1.318892	2.451585	3.112879
recovery % with respect to Row1	52%	50%	46%	67%	65%	69%
recovery % with respect to Row2	85%	90%	92%	78%	71%	92%

in Row 4. As my debugging framework is designed to recover the non-current variables caused by code reordering and deletion, it makes more sense for me to measure the recovery rate based on these variables. We can see from Row 5 of Table 6.5 that around 85% of the non-current variables caused by code reordering and deletion can be recovered by my approach.

To the best of my knowledge, Hennessy's work [8] is probably the only related previous work that has shown the recovery rate of non-current variables. Without register allocation, his approach can only recover 58% of the non-current variables caused by a limited set of local optimizations involving code reordering and deletion.[3] Compared with his experiment results, my measurements have shown that the approach I proposed is more effective in the recovery of expected variable values.

[3]The optimization techniques he considered only include local common subexpression elimination, local redundant store elimination, local reordering of computations, code motion from loops, and induction-variable elimination.

CHAPTER 7

CONCLUSIONS

7.1 Summary

Compiler optimizations cause debugging difficulties in two ways: (1) the mapping between source code and object code becomes very complicated, and (2) reporting values of source variables is either inconsistent with what the user expects or simply impossible. In this dissertation, I have presented a novel framework for debugging globally optimized code which addresses the abovementioned problems. While most of the previous approaches in this area have focused on providing truthful program behavior, the proposed framework was designed to recovered the expected program behavior, whenever possible.

Under the new breakpoint implementation scheme of the proposed framework, the debugger takes over the control of execution early to make sure the information required for recovery will not be destroyed permanently. It then moves forward executing instructions under a forward recovery model which maintains some data structures to keep track of the program states changed during the forward recovery. This enables the debugger to recover the expected behavior of a program even in the presence of optimization. In this breakpoint implementation scheme, the source breakpoints are reported to the user in

the order specified by the original source program and the behavior of exceptions meets what the user expects.

A new code location mapping scheme has been described in this dissertation. The new mapping scheme helps the debugger to determine where to suspend and resume the normal execution and decide if a source breakpoint should be reported. The algorithms and the theoretical foundations for constructing and calculating different mappings have been presented. A new instruction execution order tracking method at compile time has been described.

I have also presented a new data location tracking scheme for the recovery of expected variable values in this dissertation. In this scheme, variable definition information is generated and maintained during optimization and register allocation. Based on this variable definition information and the anchor point information, a data-flow analysis is performed to collect variable location data that is then used to generate the range information fitting to my breakpoint implementation scheme. With this range information, the debugger can unambiguously determine if the expected value of a variable is available at a source breakpoint under the breakpoint implementation scheme and how to recover it.

The framework has been prototyped in the IMPACT compiler and an experimental debugger to validate the concepts presented in this dissertation. Experiments conducted on several SPEC95 integer programs have yielded encouraging results. The overhead in executable file size and compile time incurred by the new framework is reasonable. The extra time needed for the debugger to calculate the limits of the forward recovery

region and to single-step instructions during forward recovery is hardly noticeable. On average, the expected values of around 85% of the non-current variables caused by code reordering and deletion can be recovered by my debugging framework. Compared with previous work, my approach is more general and effective in the recovery of the expected variable values. The experience with the prototype has shown that the proposed scheme provides a low-cost, practical approach to debugging optimized code.

7.2 Future Work

The framework described in this dissertation currently has a number of limitations which needs to be addressed in the future work. There are also several interesting lines of future research stemming from this dissertation.

As outlined in Section 3.2.2, the proposed scheme is incapable of handling non-monotonic loops such as modulo scheduled loops where instructions from different iterations in the original loop are mixed together in the same iteration of the new loop. To handle this, the current approach needs to be generalized so that it deals with the *dynamic instances* of instructions rather than the static instances considered here.

My debugging framework is not believed to be suitable for optimizations that reorder loop iteration spaces such as loop interchange, loop fusion, loop reversal, etc. For example, consider a loop that is reversed by the compiler so that the first source iteration is the last binary iteration and vice versa. To handle this, the debugger would have to set the interception point at the start of the loop and, during the forward recovery, save

the values modified by all instructions except those that belong to the last iteration (of the optimized loop). As I discussed in Chapter 2, this kind of optimization is probably better handled by presenting a transformed source to the user [16].

One engineering issue which can be further studied is how to pass the debug information to the debugger. Currently the additional debug information required for the new debugging framework is emitted by the compiler and propagated to the debugger in my own debug format. Existing debug information formats such as stabs [33] and DWARF [34] were not designed to support the extra information required by my approach, but some of them nonetheless do provide some extension capability. It is interesting to explore whether and how the debug information can be encoded into one of the existing debug formats.

Another area for further work is debug-time live range extension. In my current approach, if a variable becomes dead before the debugger reaches the interception point, then there is no way for the debugger to recover its last known value. The results presented in Section 6.4 have shown that most of unrecoverable variables in my framework are due to this problem. One way to overcome this problem would be to place a hidden breakpoint at a location just before the variable goes dead and have the debugger save its value. Although doing this for every variable that becomes dead can be very expensive, it may not be too costly if we do this only for the functions where breakpoints are set.

The approach described in this dissertation has only focused on user-set source breakpoints. That is to say, the approach depends on the fact that it can anticipate breakpoints and take over control of the debuggee early enough to perform debug time forward re-

covery. However, many debugger functions (such as signals, watchpoints, etc.) cannot in general be anticipated. Dealing with unanticipated stops is a major area for future work.

As mentioned in Section 4.2.2, when the forward recovery region contains a loop, my current implementation adopts a less aggressive approach which trades the availability of the values of some variables for the feasibility of the proposed scheme by moving the interception points or finish points to new places. An alternative strategy which is a potential extension for future implementation is to still use the normal forward recovery scheme on the code as is and fall back on moving the interception points or finish points when the number of instructions executed in the forward recovery mode grows too large.

Finally, my prototype debugger does not currently allow the user to change the values of variables at debug time. In general, this is not always possible: whenever the optimizer transforms code based on inferences about the values of variables (e.g, constant propagation, loop invariant code motion) or about the relations between the values of variables (e.g., many induction variable optimizations), the user must be prevented from modifying variables in such a way that these inferences are violated. Determining when it is safe to modify the value of a variable (and how to update all compiler generated temporaries that depend on the new value) so that the debugger can provide advice to the user whether the value of a variable can be changed at a given breakpoint is an interesting and important area for future research.

APPENDIX A

A DATA-FLOW ALGORITHM FOR FINDING FINISH POINTS

In the control flow graph G of the function, suppose an anchor point I of statement S is in basic block D and the function exit block is E (I assume there is a unique exit block for each function). To find out the finish points of S with regard to I, D needs to be first split into two basic blocks $D1$ and $D2$ in the same manner as described in Section 3.2.2.

Let V be the set of basic blocks which are on the paths from $D2$ to E (including $D2$ and E).[1] For each basic block B in graph G, $gen[B]$ and $kill[B]$ are defined as follows:

- If B is in V,

  $gen[B]$ = A one-element set containing the instruction which is either the instruction immediately preceding the earliest post-breakpoint function call or the last pre-breakpoint instruction (depending on which one is encountered first) in basic block B, if there is any. An empty set, otherwise.

  $$kill[B] = \begin{cases} in[B] & \text{if } gen[B] \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

- If B is not in V,

[1] V can be obtained through a simple depth-first search from $D2$.

$$gen[B] = kill[B] = \phi$$

The data-flow equations for *in* and *out* sets of B are:

$$\begin{aligned} in[B] &= \bigcup_{P \, is \, a \, predecessor \, of \, B} out[P] \\ out[B] &= gen[B] \cup (in[B] - kill[B]) \end{aligned}$$

The standard iterative algorithms for solving data-flow equations [22] can be used to derive the $out[B]$ for each basic block B in V. $out[E]$ is the set of all the finish points of S with regard to I.

REFERENCES

[1] L. Gwennap, "Intel, HP make EPIC disclosure," *Microprocessor Report*, vol. 11, pp. 1–9, October 1997.

[2] P. T. Zellweger, "Interactive source-level debugging of optimized programs," Ph.D. dissertation, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, 1984.

[3] A. Adl-Tabatabai, "Source-level debugging of globally optimized code," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.

[4] J. B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*. New York, NY: John Wiley and Sons, 1996.

[5] M. Copperman, "Debugging optimized code without being misled," Ph.D. dissertation, Computer and Information Sciences, University of California, Santa Cruz, CA 95064, 1993.

[6] L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W. W. Hwu, "A new framework for debugging globally optimized code," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999, pp. 181–191.

[7] L.-C. Wu and W. W. Hwu, "A new data-location tracking scheme for the recovery of expected variable values," IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-98-07 (ftp://ftp.crhc.uiuc.edu/pub/IMPACT/report/impact-98-07.dataloc.ps), September 1998.

[8] J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 323–344, July 1982.

[9] D. Wall, A. Srivastava, and F. Templin, "A note on Hennessy's "Symbolic debugging of optimized code"," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 176–181, January 1985.

[10] D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A practical approach to source-level debugging of globally optimized code," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 125–134.

[11] M. Copperman, "Debugging optimized code without being misled," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 387–427, May 1994.

[12] R. Wismuller, "Debugging of globally optimized programs using data flow analysis," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 278–289.

[13] A. Adl-Tabatabai and T. Gross, "Evicted variables and the interaction of global register allocation and symbolic debugging," in *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 371–383.

[14] A. Adl-Tabatabai and T. Gross, "Detection and recovery of endangered variables caused by instruction scheduling," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993, pp. 13–25.

[15] A. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code," in *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996, pp. 33–43.

[16] C. Tice and S. L. Graham, "OPTVIEW: A new approach for examining optimized code," in *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering*, June 1998.

[17] P. T. Zellweger, "An interactive high-level debugger for control-flow optimized programs," *SIGPLAN Notices*, vol. 18, pp. 159–171, August 1983.

[18] L. L. Pollock and M. L. Soffa, "High-level debugging with the aid of an incremental optimizer," in *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, January 1988, pp. 524–532.

[19] R. Gupta, "Debugging code reorganized by a trace scheduling compiler," *Structured Programming*, vol. 11, pp. 141–150, July 1990.

[20] U. Holzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 32–43.

[21] D. Ungar and R. B. Smith, "SELF: The power of simplicity," in *OOPSLA '87 Conference Proceedings*, October 1987, pp. 227–241.

[22] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[23] D. M. Lavery and W. W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, November 1995, pp. 327–337.

[24] D. M. Lavery and W. W. Hwu, "Modulo scheduling of loops in control-intensive non-numeric programs," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 126–141.

[25] R. Mirani, B. Olsen, and H. Patil, "Debugging of optimized code using debug time instruction reordering," *Unpublished Draft, Hewlett-Packard Company*, 1997.

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

[27] B. Cheng, "A profile-driven automatic inliner for the impact compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

[28] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[29] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[30] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

[31] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[32] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symp. on Compiler Construction*, June 1982, pp. 98–105.

[33] J. Menapace, J. Kingdom, and D. MacKenzie, *The stabs debug format.* Free Software Foundation, Inc., Contributed by Cygnus Support, 1993.

[34] Industry Review Draft, UNIX International Programming Language Special Interest Group, *DWARF Debugging Information Format*, 1993.

VITA

Le-Chun Wu was born in Hsinchu, Taiwan, in 1966. He grew up in Taipei, Taiwan, and attended Taipei Municipal Chien-Kuo Senior High School. He received his B.S. and M.S. degrees in Computer Science and Information Engineering from National Taiwan University in 1989 and 1991, respectively. In 1993, he came to the U.S. to pursue the Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign. He joined the IMPACT research group directed by Professor Wen-mei Hwu in the summer of 1995. His doctoral research focuses on compiler and debugger support for debugging optimized code. He spent the summer of 1996 at Rockwell Semiconductor Systems (now Conexant Systems, Inc.) in Newport Beach, California, and the summer of 1997 at Hewlett-Packard Laboratory in Cambridge, Massachusetts, working on various debugger and compiler design issues. He is a member of Phi Kappa Phi honor society. After completing his Ph.D. work in 1999, he joined Hewlett-Packard Company in Cupertino, California, as a software design engineer.