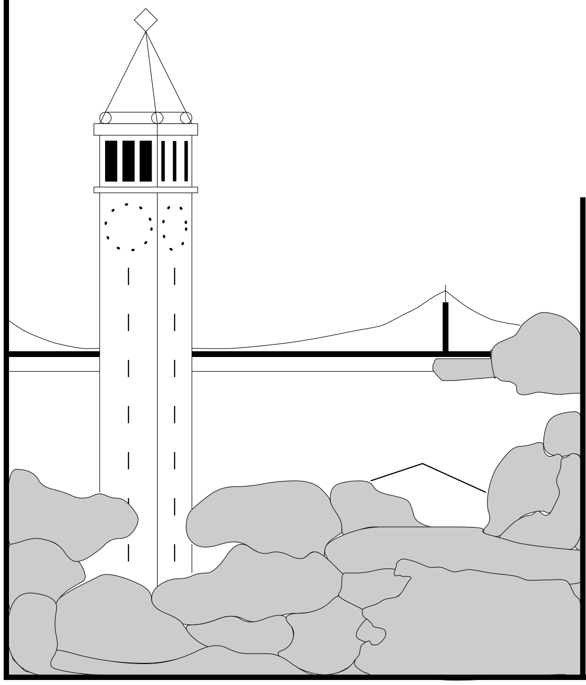
Non-Transparent Debugging of Optimized Code

Caroline Mae Tice



Report No. UCB//CSD-99-1077

November 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Non-Transparent Debugging of Optimized Code

by

Caroline Mae Tice

B.S. (The College of William & Mary, VA) 1987
M.S. (University of California, Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Susan L. Graham, Chair
Professor Alexander Aiken
Professor Alison Gopnik

Fall 1999

The dissertation of Caroline Mae Tice is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 1999

Non-Transparent Debugging of Optimized Code

by
Caroline Mae Tice

Abstract

Non-Transparent Debugging of Optimized Code

by

Caroline Mae Tice
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Susan L. Graham, Chair

Debugging optimized code is a problem for which a widely accepted solution has yet to be found. Over the years many approaches have been suggested, including limiting the compiler optimizations, restricting the debugger functionality, using recompilation or dynamic de-optimization to undo the optimizations, and having the debugger determine the effects of optimizations and mask them from the user. All of these approaches have a common thread: they place a barrier between the user and the optimizations, either altering, undoing, or hiding the effects of optimizations.

This work presents a completely different approach. By allowing users to see some of the effects of optimizations, many of the problems traditionally associated with debugging optimized code are either simplified or eliminated. The basic idea is to generate an optimized version of the source program, which accurately reflects the effects of the compiler optimizations. The user and the debugger then communicate via this optimized source program.

The theoretical issues involved in generating optimized source code, as well as the compiler support necessary for debugging optimized code are presented in this work. It also explores options for displaying the optimized source program to the user in the most useful, least confusing manner. It introduces the concept of key instructions and shows how they are critical for simplifying determining current variable locations within the executable, and mapping accurately between binary instructions and statements in the optimized source program. It also presents a pioneering approach for eviction recovery within the debugger.

These ideas have been fully implemented in Optview and Optdbx, which are de-

scribed in this dissertation. Optview is a prototype tool that generates optimized source code for C programs. It is embedded within the SGI MIPS-Pro 7.2 C compiler. Optdbx is a modified version of the SGI dbx debugger, which makes use of the optimized source code. It contains the only known implementation of eviction recovery, and it has a simple graphical user interface for displaying the optimized source program to the user. Experiments on these systems have proven that this approach provides a simple, practical, complete solution to the difficult problem of debugging optimized code.

This work is dedicated to my family: To my husband, Randall, who always believed in me and supported me throughout the most trying and difficult parts of this work, with love and patience; and to my daughter, Gwendolyn, and my son, Roland, who put up with not always getting to see Mommy as much as they would have liked, who have made the journey a little more challenging, but infinitely more enjoyable, and who have been critical in helping me maintain a balanced, clear perspective on what is truly important.

Contents

List of Figures

List of Tables

Acknowledgements

In the course of this work I received a lot of help and support from many different people. Although it would be impossible to acknowledge them all, I will do my best.

First of course is my advisor, Professor Susan Graham. She has truly helped me to develop as a researcher, giving me guidance in many areas. I have learned a lot from her. Her insights and suggestions have been invaluable. Professor Kathy Yelick also has been extremely helpful, and has given me much help and sound advice over the years.

I could not have performed my research without the help and cooperation of a huge number of people at Silicon Graphics, Inc. (SGI), particularly within the compilers group. They allowed me access to the source code for their compiler and debugger; gave me whatever equipment I needed; and answered all my questions. There are some people who especially stood out in the amount of time and help they gave to me. In particular Robert Kennedy and Dave Anderson were very helpful when I was trying to understand how certain pieces of the compiler and debugger worked, and they helped me track down some very difficult bugs. Woody Lichtenstein first granted me permission to use the SGI compiler and debugger in my work. Ross Towle and Ron Price have continued to support me in my work and research and allowed me to continue at SGI throughout the difficult times of the past few years. Jim Dehnert gave me lots of help and sound advice. Lastly I thank Jay Gischer, who first suggested that SGI would be an appropriate place for me to do my research, who helped me get started there, and who was always available to help me in any way.

In addition to those mentioned above, there are a few other people who deserve special acknowledgements. Michelle Ruscetta first got me interested in debugging optimized code while I was a summer intern at Hewlett-Packard. She helped me get started in my research, spending much time answering my questions and helping me find important papers. Mor Harchol-Balter has served both as a role-model and an inspiration, as well as a most sympathetic and helpful friend. Her help, encouragement, sympathy and advice I have found to be invaluable. Words cannot begin to express all that I owe to her. Last, but far from least, is Dr. Sheila Humphreys. She first welcomed me to U.C. Berkeley. Her door has always been open to me. She has supplied a shoulder to cry on when I most needed it; she always believed in me; she made sure I always had funding. Her kindness, sympathy and support have gone far above and beyond the call of duty.

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. F30602-95-C-0136 and Grant No. MDA972-92-J-1028, by the National Science Foundation under Infrastructure Grant Nos. EIA-9802069 and CDA-9401156, by an endowment for the Chancellor's Professorship Award from the University of California, Berkeley, and by Fellowship support from the GAANN. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Chapter 1

Introduction

The number and complexity of optimizations performed by today's compilers has far outstripped the technology for debugging the resulting programs. Over the past twenty years optimization technology has progressed by leaps and bounds. However over the same twenty years, there has been very little progress made in our ability to debug programs to which these optimizations have been applied. This lack of progress is due in a large part to two common misconceptions: that there is no real need to debug optimized code, and that debugging optimized code is too difficult a problem to solve. In this chapter we point out the fallacies of the first misconception, and introduce basic compilation and debugging paradigms. We also discuss the effects optimization has on debugging and some of the problems in presents. In later chapters we disprove the second misconception by describing the design and implementation of a solution to the problem of debugging optimized code.

1.1 Quick Overview of Compilers and Optimizations

Computers work by reading and executing sets of instructions, called *programs*. The programs that the computers directly read and execute must be written in the native language for the particular machine on which the program is executing. This language, referred to as the *machine language*, usually consists of ones and zeros. Therefore the programs written in such a language are often referred to as *binary programs*, or *binaries*.

When the first modern computers were built, back in the 1940's, people wrote their "programs" directly in machine language, usually by flipping switches on the front of the computer. As programming needs became more complex, programming directly in machine language quickly became impractical, and *assembly language* was developed. Assembly language is a low-level language that uses symbolic representations of the operations and the data locations contained in the instructions. Usually there is a one-to-one correspondence between assembly instructions and machine instructions. Along with assembly language came the *assembler* which translates assembly language programs into machine language programs. Writing large, complex programs in assembly language was still an arduous task, so in the mid-50's the first widely used high-level programming language, FORTRAN, was developed, along with a compiler to translate programs written in FORTRAN to assembly language or machine language. The FORTRAN compiler was an optimizing compiler. The developers of FORTRAN feared that unless the compiler produced assembly or machine programs nearly as efficient as those coded by hand, then no one would use FORTRAN.[6]

Optimizing compilers perform "optimizing" transformations on the program while translating it from the language in which it was written (the *source language*) to assembly or machine language (the *target language*). An optimizing transformation, or *optimization*, is a transformation that preserves the semantics and correctness of the code, and reduces either the amount of space the computation requires, or the amount of time it takes to run, or both. The term "optimization" is a misnomer, as the resulting program is rarely optimal. However optimized programs usually run faster and require less space than unoptimized programs.

Since their first introduction, optimizations have become steadily more common and more complex. Today most compilers come with at least some optimizations, and in some cases the optimizations have become such an integral part of the compiler that they cannot be turned off.

One of the reasons for the increased use of optimizations in compilers is the corresponding growth in the size and complexity of computer programs. As computer programs become more and more complicated and computer systems require more and more programmers to complete a single piece of software, the importance of writing code that is easy to read and easy to maintain increases as well. It is a well known fact, that throughout the entire life of a successful computer program, roughly 67% of the programmer time and effort will be spent in maintaining the program (debugging and modifying it).[25, 45] Furthermore it is quite common for the maintainers of the program to be different from its authors. Consequently when writing computer programs, a good programmer has three major goals in mind: Correctness of the program; readability of the program; and, ease of maintaining or modifying the program. The last two goals are especially important in light of the current tendencies towards ever larger and more complex computer programs. Unfortunately readability and ease of maintenance tend to be in direct conflict with the efficiency of the resulting code. Often the programs that are easy to read and maintain are seemingly inefficient in their use of the computer resources.

Enter the optimizing compiler. Because of the optimizations performed by these compilers, programmers are able to write programs that are relatively easy to read and maintain, and yet make reasonably efficient use of the computer resources. Thus we are allowed to "have our cake and eat it too".

1.2 General Overview of Debuggers

There is a price to pay, however, for having our cake and eating it too. The price is in the area of debugging. As Zellweger stated, "The basic presuppositions of optimization and source-level debugging conflict."[47, p. 3]. A good place to begin examining this problem is with a general overview of debuggers and how they fit into the software development process.

When creating a computer program, a software engineer usually starts with a description of the problem to be solved and possibly an algorithm with which to solve it. The engineer then sits down and creates a design of the program. Next she translates the program design into a high-level programming language, and writes the program. After writing the program, the engineer compiles it, often without any optimizations. Since most compilers, in addition to translating the source program, perform basic syntactic and semantic checks on the program, the engineer can be fairly certain that the program, once compiled, does not contain any of these basic errors.

However the program could easily contain logic errors, either because the algorithm the engineer selected for solving the problem is not a correct algorithm for the given problem, or because the engineer did not properly translate the algorithm into the programming language. Most errors of this kind can be detected only by running the program on a variety of inputs, and either examining the outputs to see if they are correct, or noting that the program "crashed" during execution (i.e. it halted execution early, with an error message, rather than terminating normally).

Once an error has been detected, the engineer needs to determine what portion of the source program caused the error. There are several alternative approaches the engineer can take:

- Visually examine the code in an attempt to spot the error.
- Simulate execution of the program by hand.
- Insert print statements into the program, to print out intermediate values during the computation. Recompile and rerun the program. Look at the intermediate values in an attempt to pinpoint where the error is occurring.
- Run the program in an interactive debugger to monitor the program as it executes.

The last choice is usually the easiest and least time-consuming option for the programmer. A debugger is a special tool that allows programmers to monitor other programs while they are executing. An *interactive* debugger is one that allows the user to enter commands and requests, controlling the execution of the target program being monitored. A *source-level* debugger is one that allows the user to enter the requests and commands in terms of the original *source program*, from which the target program was generated. A source-level debugger returns responses in terms of the source program as well.

Source-level debugging is very useful because the error exists in, and must be corrected in, the original source program. Therefore being able to relate the program execution directly to the original source program is a big help. Furthermore the programmer may have an imperfect understanding of the target language. In the remainder of this dissertation the term "debugger" always refers to an interactive source-level debugger.

In order to be useful, an interactive debugger must have at least three core pieces of functionality. First it should allow the user to manipulate the execution of the target program. In particular the user should be able to suspend and resume execution of the target program at any point during the computation upon request. Second, the debugger must be able to convey the control state of the suspended program to the user. The user must be able to determine, upon suspension of execution, which parts of the program have executed, and which parts have not. Finally the debugger needs to convey the data state of the target program to the user. The debugger needs to be able to report to the user the value of any program variable within scope at the point where execution was suspended. It also should display, on request, the current call stack, giving the user an idea of the current position in the overall computation, and allow the user to change context by moving up or down the call stack, to examine some of the values that brought the computation to its current state. A debugger that does not provide at least these basic pieces of functionality is of questionable usefulness.

Requests from the user to the debugger, and replies from the debugger to the user, are usually phrased in terms of the original source program. However the program being executed and monitored is the target program, which is written in machine language. Thus the debugger must communicate with the computer about the executing target program in terms of the machine language program. Therefore the debugger must act as two-way interpreter, translating user requests about the source program to equivalent requests about the target program before passing them on, and translating the computer responses about the target program into equivalent responses in terms of source program before presenting them to the user. This situation is illustrated in Figure 1.1.

In order to accomplish this two-way interpretation, the debugger needs a lot of information about the source program and how it relates to the target program. Since only the compiler knows fully how the source program and the target program relate to each other, the debugger must obtain this information from the compiler.

The standard method for passing this information from the compiler to the debugger is via the *symbol table* and the *line table*, extra sections of information inserted into the target program by the compiler specifically for use by a debugger. These sections are completely separate from the executable instructions and have no impact on program execution. The symbol table contains information about all the names that occur in the source program. This data includes file names, subroutine names, and the names of variables and constants, as well as user-defined type names. For the variable names, the symbol table also contains information about the type of value the variable is supposed to contain, the scope(s) in which the variable should be visible, and the memory location in which the computer will be storing the value of the variable. There are two properties of unoptimized programs that make it relatively simple for the symbol table to indicate a variable's location: Each variable is assigned a unique relative memory location that will be the variable's home location for the entire time the variable is within scope during program execution; and, every time the variable's value is updated during the computation, the new value is written to the variable's home location in memory. Thus by reporting the variable's single location in memory (and its scope), the compiler has told the debugger all it will ever need

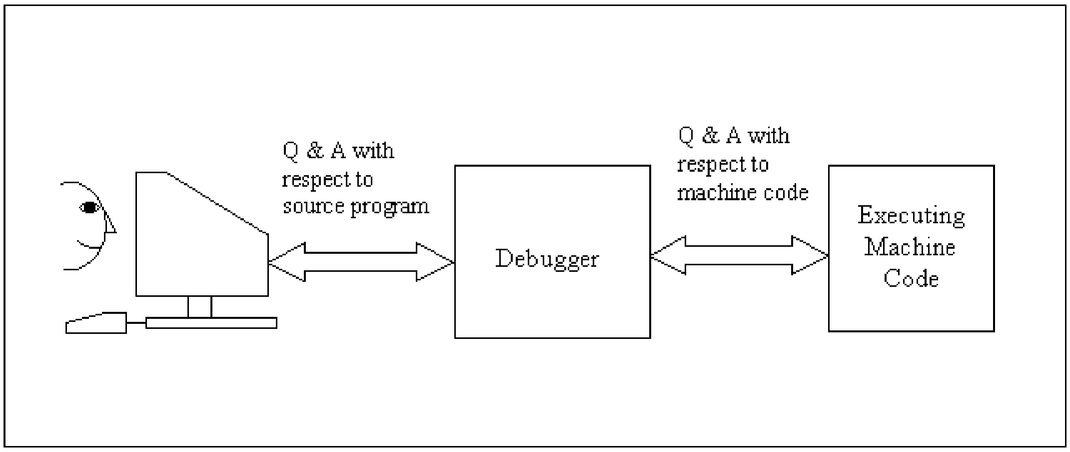


Figure 1.1: Debuggers act as interpreters.

to know in order to look up the variable's value.

In addition to the locations for the variables (the data locations), the symbol table must also give the debugger information about the locations for the pieces of code that are to be executed (the code locations). The debugger needs to know the names of all the subroutines in the program and where the code for each subroutine is located, and it also needs to know how to relate the subroutines in the target program (binary instructions) to the subroutines in the source program. The line table is used by the debugger to map code locations between the source program and the target program. The standard practice in debuggers for unoptimized code is to allow users to specify *control breakpoints* (positions at which to suspend program execution) at statement boundaries in the source program. The user sets these breakpoints by indicating the line number associated with a statement in the source program. The debugger then suspends execution of the target program immediately prior to executing the first machine instruction for the specified statement. Identifying source statement boundaries in the unoptimized source is relatively simple, because of some relationships and properties that exist between machine instructions in the unoptimized target program and the statements in the source program from which they were generated. In particular, all binary instructions that were generated from a single source statement occur contiguously in the target program[1], and the binary instructions in the target program occur in the same execution order as their corresponding statements in the source program. The line table only needs to record, for each source statement, the location of the first in the sequence of corresponding target instructions. Doing so is much simpler than mapping each individual instruction back to the source. When a user chooses to set a control breakpoint at a particular source line, the debugger looks in the line table for the first instruction for that source line, and sets a breakpoint at the corresponding instruction in the target program.

In addition to the user setting a control breakpoint, there are three other ways in which the execution of the target program may be suspended. The first is a *data breakpoint*. A data breakpoint is set ahead of time by the user. It differs from a control breakpoint in that the user requests that execution be suspended when the value of a variable is read or written, rather than when execution reaches a certain source program statement. The second potential cause of execution suspension is a *user interrupt*. In this case, the user

[1]There are a few minor exceptions, but they do not cause significant problems in the generation of line table information.

types some control sequence on the keyboard (or perhaps clicks on something with a mouse) that immediately suspends execution of the program. Users might do so if a piece of the program is running an unusually long time, and the user suspects the program has gone into an infinite loop. The final cause for suspension of execution of the target program is a *fault*. If the program asks the computer to do something illegal, such as dividing by zero, or if some system error occurs while the target program is executing, the computer can immediately suspend execution of the target program. When program execution is suspended, for whatever reason, the debugger must report to the user the corresponding position in the source program at which execution stopped. It determines this position by comparing the value of the current program counter against the line table, and reporting that execution halted at or within the appropriate source statement.

1.3 Effects of Optimizations on Debugging

Optimizations can transform a program in many different ways. Some of the more common possible effects of optimizations include moving or deleting entire source statements; changing the order in which variable values are updated; altering the control flow of the program; changing the relative order of execution events; eliminating or overlaying variables; and, assigning variables multiple home locations, including registers, during certain portions of the program.

Some combinations of these effects can have a devastating effect on the ability of a debugger to perform its tasks. Since all of the optimizations performed by the compiler are supposed to preserve the semantics and the correctness of the program, the optimized target program is guaranteed to return exactly the same results and output as the unoptimized target program, for all inputs (assuming for the moment there are no bugs in the source program or the optimizer). However the details of the computation in the optimized and the unoptimized target programs can be quite different. Figure 1.2 shows an extreme example of this situation.

The original C program shown in Figure 1.2(a) performs a relatively simple operation on a matrix: it progresses through the matrix in a linear manner, assigning to each element the sum of its four nearest neighbors. The optimizer performs three common high-level loop optimizations on the program, skewing, loop interchange, and strip mining. The resulting optimized program is shown in Figure 1.2(b). As can be seen, the optimized

```
Original Program

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



    matrix[i,j] =
             matrix[i-1,j] +
             matrix[i,j-1] +
             matrix[i+1,j] +
             matrix[i,j+1];
  }
}

(a)
```

```
After Loop Optimizations

for (j = 2; j < m + m; ++j) {
  for (i = max(i,j-m);
       i < (min(m-1,j-1) -
               max(1,j-m)/64-1)*64;
       i+=64) {
   matrix[i:i+63,j-1] =
             matrix[i-1:i+62,j-i] +
             matrix[i:i+63,j-i-1] +
             matrix[i+1:i+64,j-i] +
             matrix[i:i+63,j-i+1];
   }
   matrix[i:max(1,j-m),j-1] =
     matrix[i-1:max(1,j-m)-1,j-i] +
     matrix[i:max(1,j-m),j-i-1]   +
     matrix[i+1:max(1,j-m)+1,j-i] +
     matrix[i:max(1,j-m), j-i+1];
}

(b)
```

Figure 1.2: Example of source code before and after optimizations.

program is very different from the original, in spite of the fact that they achieve the same result.

As was mentioned earlier, there are several properties and relationships between an unoptimized target program and its source program that make generating the symbol table and the line table for debuggers for unoptimized code relatively easy and straightforward. For review, these properties are:

- All program variables are given storage locations in memory, which do not change throughout the entire execution of the program.

- All updates of variable values are written to their locations in memory.

- All binary instructions that were generated from a single source statement occur contiguously in the target program.
- The binary instructions in the target program occur in the same execution order as their corresponding statements in the source program.

Once the target program has been optimized however, none of these properties are necessarily true any more. A program variable may be assigned different memory storage locations at different points in the target program. For some portions of the target program, a variable may have no home location at all, or the home location may be a register instead of memory. Not all updates of variable values are written to the home memory location; sometimes only the register location is updated, and sometimes, in the case of a dead store, no location is updated. The binary instructions generated from a source statement may no longer be contiguous due to instruction scheduling. Instructions from multiple source statements may be interleaved. The same machine instruction may have been generated from multiple source statements. Some source statements have no corresponding instructions (dead code), and other source statements have their instructions duplicated. The order in which source statement instructions occur in the binary no longer matches the order in which the source statements themselves occur. Because none of these properties can be relied on once a target program has been optimized, the task of writing a debugger that translates requests and replies between the target and the source programs becomes much more difficult. Figures 1.3 and 1.4 illustrates this difference between optimized and unoptimized target programs.

Figure 1.3 shows a simple C source program and the unoptimized assembly code that was generated from it on an SGI Mips R4000 workstation. Figure 1.4 shows the same source program, with the optimized version of the assembly code. The numbers to the left of the source statements in Figures 1.3(a) and 1.4(a) are the line numbers for the statements. The numbers to the left of the instructions in Figures 1.3(b) and 1.4(b) indicate the source statements from which the instructions were generated. As can be seen, the instructions in Figure 1.3(b) all occur in contiguous blocks corresponding to their source statements, and they occur in the same order as the source statements. Also the variable values are all written to memory. None of that is true in Figure 1.4(b). The instructions for source statements are all interleaved and jumbled, and *none* of the variable values are written to memory.

```
Original Source                    Unoptimized Assembly

 0 void
 1 case_1a (int num, char *str){  /* num starts in $4, str starts in $5 */
 2   int i;
 3   int j;                        6 addiu $16,$4,3  /* i gets num+3    */
 4   char *s;                      6 sd $16,0($sp)   /* store i to mem  */
 5                                 7 lbu $17,0($5)   /* j gets mem[$5]  */
 6   i = num + 3;                  7 sd $17,4($sp)   /* store j to mem  */
 7   j = (int)str[0];              8 sd $5,8($sp)    /* copy str to s   */
 8   s = str;                     10 lw $25,%call16(foo)($gp)
 9                                10 ld $4,8($sp)    /* load s into $4  */
10   foo(s);                      10 jalr $25        /* jump to ``foo'' */
11   bar(i);                      10 nop
12   bar(j);                      11 lw $25,%call16(bar)($gp)
13 } /* case_1a */                11 ld $4,0($sp)    /* load i into $4  */
                                  11 jalr $25        /* jump to ``bar'' */
                                  11 nop
                                  12 lw $25,%call16(bar)($gp)
                                  12 ld $4,4($sp)    /* load j into $4  */
                                  12 jalr $25        /* jump to ``bar'' */
                                  12 nop
                                  13 jr $31      /* return from case_1a */
                                  13 addiu $sp,$sp,48 /* reset stack ptr*/

                                  Summary: 19 instructions; 3 nops;
                                           3 memory writes; 3 memory reads

        (a)                                            (b)
```

Figure 1.3: Source program and equivalent unoptimized assembly code

```
Original Source                    Optimized Assembly

 0 void
 1 case_1a (int num, char *str){  /* num starts in $4, str starts in $5 */
 2   int i;
 3   int j;                         1 or $30,$4,$0    /* copy num to $30 */
 4   char *s;                    8,10 or $4,$5,$0     /* copy str to $4  */
 5                                 10 lw $25,%call16(foo)($gp)
 6   i = num + 3;                   7 lbu $1,0($5)    /* $1<-deref of $5 */
 7   j = (int)str[0];              10 jalr $25        /* jump to ``foo'' */
 8   s = str;                       7 sd $1,24($sp)   /* spill $1        */
 9                                 11 lw $25,%call16(bar)($gp)
10   foo(s);                       11 jalr $25        /* jump to ``bar'' */
11   bar(i);                     6,11 addiu $4,$30,3  /* $4 gets num + 3 */
12   bar(j);                       12 lw $25,%call16(bar)($gp)
13 } /* case_1a */               7,12 ld $4,24($sp)   /* load spill value*/
                                   12 jalr $25        /* jump to ``bar'' */
                                   13 ld $30,8($sp)   /* spill code      */
                                   13 jr $31      /* return from case_1a */
                                   13 addiu $sp,$sp,48 /* reset stack ptr*/

                                 Summary: 15 instructions; 0 nops;
                                          1 memory writes; 1 memory reads

       (a)                                              (b)
```

Figure 1.4: Source program and equivalent optimized assembly code

1.3.1 Updating Variables in Optimized Code

A common and frequently used function in debuggers for unoptimized code allows the user, in addition to seeing the current value of a variable, to modify the value. The target program then uses the modified value throughout the rest of the computation. This functionality is very difficult to provide once a program has been optimized, and is usually disallowed.

Whenever the optimizer can determine that a variable will contain a particular value for a particular portion of the program, it takes advantage of this fact. Because of optimizations such as folding, propagation, and common subexpression elimination, any attempt to update a variable in the optimized program might require updating several other variable values, as well as possibly requiring updating constant values stored in the instructions themselves. Identifying all the locations that might need to be updated would require the debugger to have complete information about how the variable was used in all optimizations. If the debugger required this information for all program variables, the amount of information would be too large to be practical. Furthermore some optimizations, such as dead code elimination, may be based on the fact that the compiler knows a variable will contain a certain value at a certain position. Therefore the optimizer can determine that a predicate will return a certain value, and the "alternative" branch code will never be needed, so the compiler never generates any code for it in the target program. If the user were to update the value of a variable which the compiler happened to use in such an optimization, the resulting predicate value might change, the "alternative" branch code would be needed, and there would not *be* any alternative code. The program would apparently do the wrong thing, making the user believe there was an error in the program when there was no such error. For these reasons updating variable values in debuggers for optimized code is universally disallowed. Exploring potential acceptable solutions to this problem is an open research issue.

1.4 Why Debug *Optimized* Code at All

Given the fact that it is much easier to write debuggers for unoptimized target programs, one might wonder if there is really a need to create debuggers for optimized programs. It seems much simpler to debug the unoptimized target program, and to compile with optimizations only after all the errors have been found and corrected. This approach sounds nice in theory, but there are several reasons why it is not practical.

1.4.1 Latent Bugs in Unoptimized Programs

It sometimes happens that a program exhibits one behavior when unoptimized, and a different behavior once it has been optimized. A common misconception is that if there is a difference in behavior between the optimized and unoptimized versions of a program, the optimizer must have introduced an error into the program. This assumption is not necessarily true. There are two types of circumstances that can lead to differing behavior between the optimized and the unoptimized programs: loosely or poorly defined semantics in the programming language, or an incorrect program. Loose semantics are not uncommon in the specification of high-level programming languages. A common example of loose semantics in the language specification is that the order of evaluation of subexpressions (or arguments, in function calls) is undefined, and left up to the discretion of the compiler. It is possible that a program may exhibit different behaviors if the optimized program chooses one order of evaluation and the unoptimized program chooses another.

There are certain types of bugs that, while present in the unoptimized program, do not manifest themselves until the program is optimized. Usually these bugs take advantage of loose semantics in the programming language. Since these bugs only appear in the optimized code, they must be debugged there. Some program characteristics that optimizations can affect, and which might cause some of these latent bugs to appear, include the values of uninitialized variables, data layout within the program, and execution time of various parts of the program (i.e. race conditions). The most common errors of this type are dynamic memory errors. While there exist tools such as Purify [20] and the UNIX lint utility which can help find some of these bugs in the unoptimized code, they do not work in all cases. Some programs use system memory management options to allocate large chunks of memory at a time. These programs then allocate and deallocate dynamic memory within the large allocated chunks, only releasing these large blocks of memory at the end of execution. Purify and similar tools will not find any dynamic memory errors in these types of programs, yet the memory errors can and do exist, and can exhibit the latent behavior described above, hence requiring the ability to debug optimized code.

1.4.2 Debugging Optimized Core Dumps

Sometimes a program bug is manifested by a complete program crash, accompanied by a core dump. Furthermore there are times when it is necessary to debug from such a core dump rather than by starting the program over from the beginning. This kind of debugging is sometimes referred to as *post-mortem* debugging, because execution has already been halted, and cannot be resumed. All that can be done is an examination of the state the program was in when it halted. If the data on which the program was being executed was real-time data that cannot be recreated, post-mortem debugging may be the only option. Similarly if a customer reports that a vendor's program crashed with a core dump and sends the core to the vendor for debugging, post-mortem debugging must be used. In such situations if the core dump is from a program that was optimized, then the problem reduces to the problem of debugging optimized code.

1.4.3 Debugging Software Products

Software vendors must be concerned with the efficiency, as well as with the correctness, of the programs they develop and sell. Since optimized programs make far better use of computer resources than unoptimized programs, it is highly desirable, from the vendor's point of view, to ship optimized products. At the same time, the software market is extremely competitive. In order to survive, it is imperative to get products released to market as quickly as possible. Furthermore since no one who sells products that contain many bugs remains in business for very long, software vendors devote a large amount of their product development time to testing and debugging their products before they release them. These circumstances leaves the vendor with some very difficult choices to make. The options available are: 1). Attempt to do the entire test-debug cycle on optimized code (which requires a good tool for debugging optimized code); 2). Ship the product unoptimized; 3). Test and debug only the unoptimized version, and have faith that the optimizer will not introduce or uncover any new bugs; or, 4). Double the length of time spent testing and debugging, once unoptimized, then again optimized[2]. Option one would obviously be the best, but there do not currently exist any widely accepted tools for debugging optimized code. Some software vendors settle for option two, especially as this option also makes it

[2]The hope is that no new bugs are uncovered when the code is optimized; otherwise there would still be the need for a tool to debug the optimized code.

easy for them to debug any errors that customers might find in the software. Those who do ship optimized code probably settle for option four. While it may be exaggerating slightly to state that the test-debug cycle will be just as long the second time around (since the assumption is that most bugs will have been found in the unoptimized code), the second round of testing could well be a significant fraction of the first, since it takes time just to re-run all of the tests and examine all of the results, and since any changes caused by fixing a bug would require the two-phase process to repeat.

1.4.4 Debugging Programs with Large Resource Requirements

There are some programs which, either because of the time they take to run or the amount of memory and other resources they require or both, can only be run if the program is optimized. Thus only the optimized version of such programs can be debugged with an interactive debugger tool. It might be argued that as hardware continues to speed up at a phenomenal rate this type of problem will eventually become a non-issue. However it could be just as reasonably argued that as hardware becomes faster, bigger and bigger problems (problems that used to be too big to be reasonably computed) may be programmed, so that the one effect nullifies the other, leaving a need for debugging optimized code. History tends to bear this trend out.

1.4.5 Required Optimizations

It is the case for certain compilers that some optimizations are such an integral part of the compiler that the compiler cannot be run without them. Therefore if one wants to debug programs compiled on such a compiler, one needs the ability to debug optimized code.

1.5 Why Not Use "Standard" Debuggers

Debuggers designed for unoptimized programs make use of the assumptions and relationships described previously that exist between a source program and an unoptimized target program. Such a debugger, when presented with an optimized target program, will make invalid assumptions, and its behavior can become nondeterministic, or worse, it can mislead the programmer by giving completely false information about the program being

```
Original Program                           Corresponding Instructions

S0:  x = i * j + offset;                    I0: ld   $r4,$sp(0)    (S0)
S1:  /* code not referencing y */           I1: ld   $r5,$sp(8)    (S0)
S2:  r = foo(x);                            I2: mult $r6,$r5,$r4   (S0)
S3:  y = (r >offset) ? 1 : 0;               I3: add  $r4,$r6,$r7   (S0)
S4:  if (y) {                               I4: jalr ``foo''       (S2)
S5:     /* code not referencing x */        I5: slt  $r1,$r7,$r2   (S3)
S6:  }                                      I6: bgez $r1,label.1   (S3)
                                            I7: ldi  $r4,1         (S3)
                                            I8: b    label.end     (S3)
                                            I9: label.1            (S3)
                                           I10: ldi  $r4,0         (S3)
                                           I11: label.end          (S3)
                                           I12: bgnez $r4,...      (S4)

            (a)                                        (b)
```

Figure 1.5: Source code and instructions illustrating two variables with non-overlapping live ranges sharing a storage location.

executed. For example in an optimized target program, the compiler may have determined that two particular variables are never live at the same time. Therefore it assigns them both to the same register. Figure 1.5 illustrates this problem. Variable x in Figure 1.5(a) is live from statement S0 through statement S2, while variable y is live from statement S3 through statement S6. These two live ranges do not overlap at all. Therefore the compiler decides to store both variables in Register 4. Figure 1.5(b) shows the corresponding instructions. Variable x is in Register 4 from I3 through I6. Thereafter variable y is stored in Register 4, and the overwritten value of x is lost. Now suppose the user requests the debugger to halt execution of the program at statement S4. At this point in the program variable x is still within scope, so the user can ask to see the value of variable x. The debugger looks in x's location (Register 4), which at the current time happens to hold the value of y, and returns that value. Now, unless x and y are supposed to have the same value at that point, the user believes that variable x has the wrong value, and could spend a very long time tracking down this seeming error, which really is not at error at all. This type of misinformation can lead to significant amounts of time being wasted chasing chimerical bugs.

1.6 Basic Problems Debugging Optimized Code

When designing a debugger for optimized code, the very first questions that need to be answered are: What information about the target program is the debugger going to present to the user? In what form will it be presented, and how? These questions comprise what we call the *Debugger-User Communication Problem.* Up to this point, we have been talking about presenting the information to the user in terms of the source program. Indeed that is the common and most desirable form since, as previously mentioned, the bug will eventually have to be found and corrected in the source program. However that is not the only option. There are a large number of alternatives, some more reasonable or practical than others.

An important issue when considering the debugger-user communication problem, is transparency. *Transparency* is the illusion given the user that there is no optimization. A debugger exhibits *transparent behavior* if it returns exactly the same responses for all requests as it would if the target program were not optimized. A debugger that behaves differently for an optimized program exhibits *non-transparent behavior.* Transparent behavior is also referred to as *expected behavior*, because the debugger behaves the way a user expects it to, based on the original source program. The decision about how transparent the debugger's behavior should be has a large impact on the rest of the design of the debugger. There is really a continuum of possible behaviors, with varying degrees of transparency, as illustrated in Figure 1.6. At one end of the continuum is pure transparent behavior. In this case the debugger returns exactly the same responses, in all cases, for all inputs, as it would if the program had not been optimized. At the other end of the continuum is pure non-transparent behavior. Pure non-transparency means that the debugger makes all the effects of all optimizations apparent to the user. We do not believe it is possible to achieve either extreme. Therefore it is more a question of where along the continuum desired behavior lies. As is indicated in Figure 1.6, the closer one gets to pure transparent behavior, the easier it is to relate what one is seeing back to the original source program (the view being presented is very close to the original program), but the harder it is to understand why the program is behaving as it is, as the source program one is viewing may bear very little resemblance to the code that is actually executing. On the other hand, the closer one gets to pure non-transparent behavior, the easier it is to understand why the program is behaving the way it is and why it is returning those responses, as the source program being viewed corresponds more closely to the executing code, but the harder it is to relate those responses to the original source program. As there are tradeoffs in either direction, one must decide on what seems the most appropriate compromise between the two.

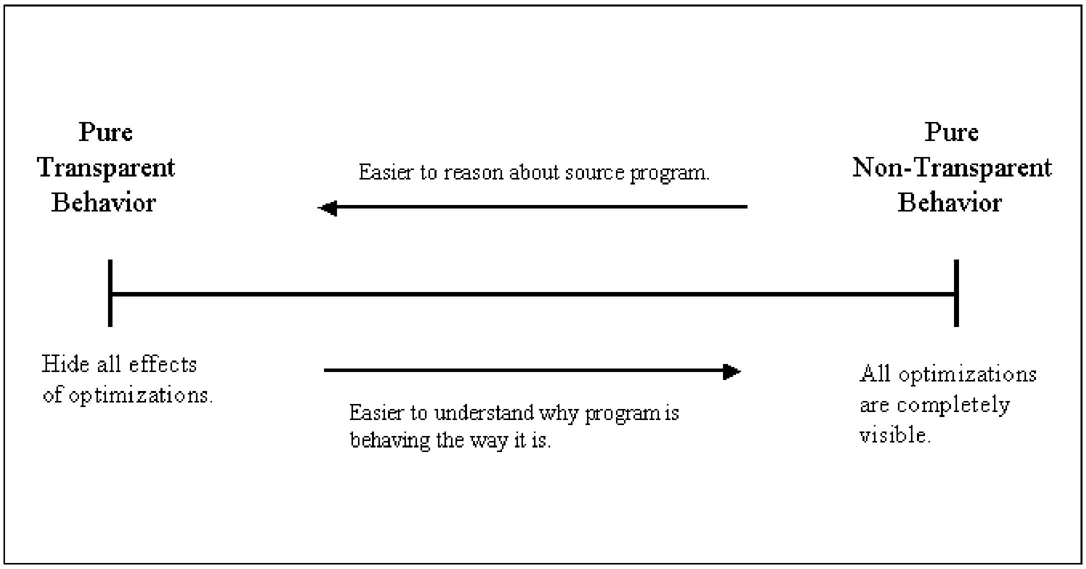


Figure 1.6: The Transparency Continuum.

Once the questions for the Debugger-User Communication Problem have been resolved, one must determine what information the debugger will need from the compiler in order to perform its tasks and present the data to the user, and how it will obtain this information from the compiler. These issues make up what we call the *Compiler-Debugger Communication Problems.*

The Compiler-Debugger Communication Problems fall into two broad classes. The first class has to do with mapping between the source program (or whatever other form is chosen for presenting data to the user) and the target program. These mapping problems are commonly known as the *location problems.* The other class of problems has to do with finding and returning the current value of a variable in response to a user's request. These problems are referred to as the *data value problems*.

1.6.1 The Location Problem

The location problem is a bidirectional mapping problem. The debugger needs accurate mapping in both directions: the source-to-target mapping is used when the user requests a breakpoint at a particular position in the source program; and the target-to-source mapping is used when execution is suspended either because of a data breakpoint, user interrupt or program fault, and the debugger needs to report where, in the source program, the execution stopped. These problems are compounded by the fact that once a target program has been optimized, there is no longer a nice one-to-one correspondence between instructions in the target program and statements in the source program. A single instruction, or set of instructions, may have been generated from multiple source statements; or a single source statement may not have generated any instructions (because the compiler determined it would never be executed). The rearranging, duplicating, and interleaving of instructions (and even of entire statements) that occurs in optimized code further complicates the matter.

The source-to-target mapping is the harder of the two problems. The real question to resolve is where to set breakpoints for source statements in the target program. Unlike in the unoptimized program, there are no clear statement boundaries in the optimized code. It is no longer reasonable to automatically use the first instruction for a statement as its breakpoint location, as this instruction may occur a long distance from the rest of the instructions for the statement. The difficulty is increased because a user can have several different reasons for setting a breakpoint. One reason for setting a breakpoint is to examine the results of a previous statement, which the user assumes will have finished executing by the time the breakpoint has been reached. Another possibility is that the user wishes to stop just before the execution of a statement, in order to examine values that the statement will use or modify before the modification takes place. Because the locations of the end of the preceding statement and the beginning of the next are the same location in the unoptimized target program, implementing a single type of control breakpoint works well for both cases in unoptimized programs. This property is not necessarily true for optimized programs. In her pioneering research in this field [47] Zellweger pointed out that there are two basic

```
Original Program                      After Optimization

    ...                                   ...
12: while (i < j) {                   13: j = 10;
13:   j = 10;                         12: while (i < j) {
14:   A[i] = B[i-j];}                 14:   A[i] = B[i-j];}
    ...                                   ...

      (a)                                   (b)
```

Figure 1.7: Example of optimizations changing the order of variable assignments

options when deciding where to place a statement's control breakpoint in optimized code, either use a *semantic breakpoint* or a *syntactic breakpoint*.

The idea behind syntactic breakpoints is that the order of source statements in the original program, relative to one another, is preserved. Thus when single stepping through a program the apparent execution order of the statements is exactly what one would expect from examining the original source program. For example looking at Figure 1.7, the syntactic breakpoint for line thirteen will occur after the breakpoint for line twelve and before the breakpoint for line fourteen, in spite of the fact that line thirteen does not occur between lines twelve and fourteen in the optimized program. Because line thirteen is a loop invariant statement the optimizer moved it out of the loop and placed it before line twelve. Using the syntactic breakpoint scheme, and setting a breakpoint on line thirteen would cause execution to suspend on every iteration of the loop, prior to any breakpoint that might be set on line fourteen, even though the actual code for statement thirteen is not there any more. Syntactic breakpoints can present difficulties to implementors, as they sometimes require a breakpoint for a source statement be placed in a location where there are not any instructions for the statement.

In contrast to syntactic breakpoints, semantic breakpoints are set on the actual code for the source statement, wherever the code happens to occur in the optimized program. As Zellweger points out, if one chooses a semantic breakpoint scheme one still has several options as to which instruction, *exactly*, gets selected as the breakpoint location: the very first occurring instruction for the statement (which may be quite far removed for any of the other instructions for the statement); *every* instruction for the statement, or at least every block of contiguous instructions (if there are any); the first instruction for the statement that causes any alteration in the user-visible execution state; the first instruction (if any) that alters any user-visible variable; or the first instruction that "corresponds to the core semantic effect of the statement". Most research since Zellweger has chosen the last option as the definition of a semantic breakpoint ([2, 15, 16, 36, 42, 43]).

1.6.2 Data Value Problems

Data value problems have received the vast majority of attention to date in research on debugging optimized code ([1, 2, 14, 15, 16, 21, 22, 29, 30, 31, 42, 43]). There are actually two or three data value problems, depending on whether the debugger is taking a transparent or non-transparent approach. The first problem is the *data location problem*. The data location problem is determining where the current value of the requested variable is stored, assuming for the moment that it is currently stored somewhere. This problem arises in optimized programs because, unlike in unoptimized programs, a variable's value may be in any of several different locations, either in memory or in registers, at any given point during the computation. Furthermore a common optimization is to entirely eliminate updating a variable's "home location" in memory, unless absolutely necessary (writes to memory are expensive operations), so knowing the variable's home location in memory is not much help.

The second data value problem is the *residency* problem. It is not uncommon for an optimizer to determine that a particular variable's value is not needed for some portion of the computation. If the value was stored in a register (a fairly common occurrence), the compiler just re-uses the register for some other purpose without bothering to save the value that was currently in it, as with variable x in Figure 1.5. Thus the variable's "current" value no longer exists in the computer. The computer has *evicted* the variable, and it has become *non-resident*. (Variables whose current values still exist somewhere in the computer are *resident*). This situation creates a major problem for debuggers because a non-resident variable is still within the current scope, and so the user might reasonably and legally request to see its current value. Adl-Tabatabai and Gross [1] ran some experiments to determine the frequency of non-resident variables, and found that for the three benchmark programs they checked, non-resident variables occurred at 30-60% of all possible breakpoint locations.

```
Original Program

   ...
i = 10;
j = 10;
if (cond) {
  for (i = 0; i < n; i++) {
    j = x;
    a[j+i] = a[i];
  }
}
```

(a)

```
After Optimization

   ...
i = 10;
j = 10;
if (cond) {
  j = x;
  i = 0;
  while (i < n) {
    i++;
    a[j+i-1] = a[i-1];
  }
}
```

(b)

Figure 1.8: Example of original source program and corresponding optimized source

The Currency Problem

If one is attempting to create a relatively transparent debugger of optimized code (i.e. hide the effects of optimizations and act as if the code were unoptimized), then there is also a third data value problem, the *currency problem*. The currency problem was first explored by Hennessy [21]. It arises because the debugger is attempting to meet the user's expectations for the target program's behavior, i.e. it is trying to exhibit *expected behavior*. Those expectations are based on the source program, which the target program may not resemble very much after optimization. The main thrust of Hennessy's work is determining what response the debugger should give in response to a request for a variable's value (assuming for the moment that all values are resident and the debugger knows where to find them). The problem arises from the fact that if two variable updates in the source program are independent of each other, then the optimizer is free to change the order in which the updates take place, which it frequently does. Thus at any given point during program execution, a variable's *actual value* (the current value in the target program) may or may not correspond to its *expected value*, based on what the user sees in the source program. Figure 1.8 shows an example of this problem.

Figure 1.8(a) shows a small fragment of code, and Figure 1.8(b) shows the equivalent code after optimizations have been performed. The optimizer moved the loop invariant assignment to j out of the `for` loop, and made the increment of i the first statement in the loop rather than the last. Suppose that a user, looking at the original source, sets a breakpoint at line 6. Execution would be suspended at line 8 in Figure 1.8(b). At this point if the user requests to see the value of variable i, the debugger must determine that i has been updated prematurely, and somehow recover its old value. If the user were to set a breakpoint at line 5 in Figure 1.8(a), and if the debugger is using a syntactic breakpoint scheme, the corresponding breakpoint would actually be set at line 7 in Figure 1.8(b). At this point variable j has been prematurely updated, but only on the first iteration through the loop. On every subsequent iteration the value of j would be current. This example illustrates just a small sample of some of the issues in the currency problem.

At any given breakpoint, for any particular variable, one of several situations may be the case. An assignment to the variable which occurs further on in the source program has already executed; an assignment to the variable which has already occurred in the source program, has been delayed and has not executed yet; the assignment statement (or even the entire variable) has been optimized away; or, the assignments to the variable in the source program happen to match the ones that have executed. In the last case, the variable's actual value matches its expected value, so its value is *current*. In all of the other cases the actual value differs from the expected value (or there is no actual value), so the variable is *non-current*. Sometimes the debugger cannot tell for certain whether a variable's value is current or not, as that may depend upon the execution path taken to the current point. In this case the variable's value is said to be *suspect*. A variable whose value is either suspect or non-current is *endangered*.

If a user requests to see a variable's value, and the debugger naively returns its actual value, and that value is different from the expected value, then the user, who has no idea of what optimizations have been performed, may well assume that the value being returned is incorrect and is the result of a program error. The user could then waste a large amount of time searching for a non-existent bug. Obviously this situation is not desirable. Therefore debuggers that attempt to exhibit expected behavior must somehow determine whether the variable is current or not, and if not, must either inform the user that the current value is unavailable (falling back on *truthful behavior*); display the actual value and try to explain where it came from, or at least indicate that it is not an erroneous value (an alternative form of truthful behavior); or somehow determine what the expected value is and present that value. The last option is the most preferred, given that the debugger is attempting expected behavior, but is also the most difficult, especially given that one must do this work for every program variable at every possible breakpoint in the program. There is a large body of research that explores various ways to determine currency and recover expected values ([1, 2, 14, 15, 16, 21, 22, 29, 30, 31, 42, 43]).

1.7 Characterizing A Solution for Debugging Optimized Code

Before proceeding, it seems reasonable to discuss the desirable characteristics of a good debugger for optimized code. In truth these are the same characteristics that are desirable in a good debugger for unoptimized code. These characteristics include:

- The debugger should allow the user to accurately monitor the flow of control through the program during program execution. In order to do so it needs to give the user an accurate picture of which statements are executing, and the order in which they are being executed.
- The debugger should allow the user to examine the current state of the target program, including the current value of any variable within the current scope of the target program. The current state also includes the call stack.
- The debugger should allow the user to monitor the execution of the target program that actually exhibited the erroneous behavior.
- The debugger should be *non-invasive*, not altering in any way the target program it is monitoring.
- The debugger should run reasonably efficiently.

The first two characteristics are part of the general definition of a debugger. The third characteristic in some ways seems like a tautology: If a particular target program is exhibiting erroneous behavior, why would someone attempt to fix the problem by debugging a different program? However as will be seen in the next chapter, some attempts to design a debugger for optimized code have taken this approach.

The fourth characteristic is in some ways an extension of the third: if the debugger itself alters the target program, then the program being debugged is no longer the same as the one that exhibited the erroneous behavior. This characteristic cannot be strictly adhered to, as a debugger must alter the code slightly in order to insert (and remove) user requested breakpoints. However this type of modification is the only acceptable one a debugger can make to the target program.

The final characteristic is added for practicality. Assuming that it were possible to build a completely accurate, perfect (except for efficiency) debugger for optimized code, if the debugger takes too long to run no one will want to use it.

1.8 Summary

In this chapter we have presented an overview of compilers and optimizations, and explained how debuggers work on unoptimized programs. We have also discussed how optimizations affect debugging. We have explained the need for the ability to debug optimized code, shown how standard debuggers are inadequate for the task, and described the basic problems involved in debugging optimized code. We have also briefly touched upon the characteristics that are desirable in a debugger for optimized code.

1.9 Scope and Goals of this Work

The goal of this dissertation is to present a new paradigm for debugging optimized code and to demonstrate the benefits and practicality of this paradigm. In the past most research in this area has started from the assumption that the expected behavior approach, masking optimization effects whenever possible, is the best approach. In this dissertation we show the fallacy of this base assumption and demonstrate that there is an acceptable alternative that is at least as effective in allowing users to debug optimized code.

In Chapter 2 we discuss previous and related work in this field. Chapter 3 presents the theory of our solution to the problem of debugging optimized code. In Chapter 4 we describe the implementation of our solution within an existing commercial compiler and debugger. Chapter 5 presents measurements and analysis data from our implementation and discusses these results, and Chapter 6 presents our conclusions.

Chapter 2

Related Work

In this chapter we discuss past and present research in the area of debugging optimized code, as well as some other research that relates to our approach. Research on debugging optimized code can be classified into five broad categories: non-interactive approaches; transparent, invasive approaches; transparent, non-invasive approaches; currency determination; and non-transparent approaches.

2.1 Non-interactive approaches

A major motivation for debugging optimized code is the fact that a program may contain a latent bug which only manifests itself after the program has been optimized. Since the program only exhibits its erroneous behavior when optimized, it must be debugged optimized. As explained in the previous chapter, this difference in behavior between the optimized and unoptimized versions of the program (assuming for the moment that the optimizer is correct) can be attributed to incompleteness of the source language specification. Anything in the source language specification that is left as "undefined" leaves an opening for this kind of behavior. Dynamic memory access errors, while not the only culprits, are a common source of these undefined behaviors, especially in C. Eliminating these errors could, in some cases, significantly reduce the occurrence of latent bugs that only appear after optimizations. Since dynamic memory access errors are very common and often quite difficult to pinpoint, several approaches have been suggested that are aimed specifically at finding or preventing these types of errors, before they manifest themselves [5, 18, 20].

One of these approaches was proposed by Austin, Breach and Sohi [5]. In their method special bookkeeping data structures are wrapped around all pointers within the program. As the program executes, information in these special data structures is updated appropriately. Before reading or writing through any pointer reference, the contents of the bookkeeping data are checked and any attempts to incorrectly access memory are caught and reported. This approach finds all pointer and array access errors within those portions of the code that execute. Implementing it requires a complete revision of the compiler, as well as modifying `malloc`, `free`, `realloc`, and any dereferencing operators.

Evans [18] proposed a static approach for detecting dynamic memory errors. It involves the programmer adding annotations to the code at variable declarations and interface points. These annotations can then be used by his tool, LCLint, to help catch dynamic memory errors. This approach works very well and succeeds in catching many, though not all, such errors, as some can only be caught at runtime.

Purify [20] is a well-known widely available tool that also help programmers find dynamic memory errors in their programs. Purify works by inserting instructions directly into the object code generated by the compiler. These extra instructions check every memory access during program execution, allowing the tool to detect access errors. This approach is very effective much of the time.

All of the approaches described above help reduce dynamic memory errors, which can cause latent errors that require debugging of optimized code. Thus they would complement a debugger for optimized code, though none of the work is particularly directed towards that goal. In contrast Jaramillo, Gupta, and Soffa [24] attempt to provide a complete alternative to interactive debugging of optimized code. They have designed and built a system that takes both the optimized and unoptimized executables for a program, and compares the runtime behavior of the two versions of the program. Their tool, given an input, executes both versions of the program in parallel on the input, monitoring their executions. If at any time the semantic behavior of the optimized program diverges from that of the unoptimized program, their tool flags the location at which the divergence occurs, and determines which optimizations were responsible for the differing behavior. The user can then either turn off the optimization responsible for the differing behavior, or examine the source near the identified location in order to determine which piece of code caused the optimizer to generate incorrect code.

2.2 Transparent approaches

Research on source-level interactive debugging of optimized code really started in the early 1980's, with work by Hennessy [21] and by Zellweger [46, 47]. In his paper Hennessy first identified and described the *currency problem*. Recall that the currency problem arises when the user requests to see the value of a variable, and the variable's actual value may be different from the value the user expects, due to optimizations that may reorder or delay updates of variable values. In his paper Hennessy defines the terms current, noncurrent, and endangered with respect to variable values, as well as presenting the first scheme for attempting to recover the expected value for noncurrent variables, using roll-forward and roll-backward techniques. His ideas serve as the foundation for much of the research done in this area. Wall, Srivastava, and Templin [39] and Copperman and McDowell [14] later revised Hennessy's original algorithms to correct for changes in compiler technology that invalidated some of Hennessy's original assumptions.

Zellweger [47] was the first researcher to thoroughly investigate the topic of debugging optimized code. The first half of her dissertation is a complete analysis of all the problems optimizations can cause for debuggers, as well as descriptions of various possible approaches that might be taken to solve these problems. She does an excellent job of identifying and defining all of the major problems in this area. She also does a very thorough, complete job of considering all possible approaches for solving the problems, outlining the advantages and disadvantages of each approach. She defines many terms that have since become standard in debugging optimized code literature. Nearly all later work on debugging optimized code (including this) has built heavily on her work. In the second half of her dissertation, Zellweger describes her solution to debugging optimized code in the presence of two control-flow optimizations, cross-jumping and function call inlining. In her approach, the debugger inserts code into the optimized target program to collect information about execution paths. It then uses this information to help mask the effects of these two optimizations from the user and allow the user to perform "normal" debugging on the optimized program. This approach worked reasonably well on the two optimizations it addressed.

2.2.1 Invasive Debugging

In this section we will briefly review transparent approaches to debugging optimized code that are *invasive*, i.e. they make alterations to the binary, in order to simplify or allow debugging. A concern with this type of approach is that altering the binary might, in some cases, mask the error. The likelihood of an error being masked depends heavily on characteristics of the source language of the program.

Hölzle, Chambers, and Ungar [22] take the approach of dynamically deoptimizing the function in which the user wishes to perform debugging. They also chose not to have their compiler perform certain optimizations, in order to allow the user full debugging capabilities. Their approach provides full expected behavior without requiring any changes to the debugger. The language for which they did this work is SELF, which is a pure object-oriented language with garbage collection. Therefore, while their approach does not debug the *optimized* code, the probability of the error being masked is small.

Pollock, Bivens, and Soffa [29] take a similar approach. They use incremental analysis and recompilation to "tailor" programs. If the user enters a request that cannot be properly serviced by the debugger because of optimizations, they incrementally re-compile the program, eliminating those particular optimizations that prevent the debugging request from being serviced in the expected manner.

Shu [31] makes heavy use of hidden breakpoints to capture data about which execution path was taken in order to provide expected behavior. On reaching a user-specified breakpoint, he uses information obtained from the optimizer about the optimizations that were performed to "undo" the optimization effects for the user. His solutions appears preliminary; he does not explain how he would deal with several important aspects of debugging optimized code.

2.2.2 Non-Invasive Debugging

The first complete, non-invasive solution for debugging optimized code was presented by Coutant, Meloy, and Ruscetta [16] in 1988. Their approach made no changes to the optimized code whether it was compiled for debugging or not. They were the first to track variable locations through registers as well as memory. They introduced the concept of a *variable range table* to keep track of all the locations where variables are kept throughout execution. Variable range tables have since become standard for debuggers of optimized code. Coutant, Meloy, and Ruscetta concentrated more heavily on the data value problems, than on code location and mapping issues. They detect noncurrency, but unlike most previous approaches, do not attempt any recovery. Instead they take a conservative, truthful approach, indicating when variables values are unavailable, or how they might be different from what the user expects. The debugger they describe, the Hewlett-Packard DOC debugger, is one of only two commercial debuggers for optimized code of which we are aware.

Adl-Tabatabai was the first to identify and study the variable residency/eviction problem [1, 2, 3]. He uses dataflow analysis techniques to identify evicted variables and also to detect current, noncurrent, or endangered variables. His approach performs limited recovery of noncurrent variables, but no recovery for evicted variables. As with Coutant, Meloy, and Ruscetta, Adl-Tabatabai concentrates heavily on data value problems. His approach is a truthful one, telling the user when variable values cannot be accurately reported due to optimizations, and attempting minor recovery of the expected values.

The most recent transparent approach has been taken by Wu et. al. [43, 44]. Wu focuses heavily on breakpoints and mappings between source statements and machine instructions, and in particular his focus is on providing expected behavior at user defined control breakpoints. In order to do this, the debugger (with the aid of special information from the compiler) identifies an *intercept* instruction in the binary which precedes the instruction for the actual breakpoint. When execution reaches the intercept instruction, the debugger takes over control and emulates execution of the program up to the breakpoint instruction. During the emulation, the debugger is careful to execute exactly those instructions (and only those instructions) necessary to provide the user with expected behavior at the breakpoint. Once the breakpoint has been serviced, the debugger performs some clean up and eventually returns execution control to the kernel. This approach works pretty well for those cases it is designed to handle. However because it is emulating expected behavior rather than truly executing all the binary instructions, occasionally it will return a value that is different from the value the binary would really have calculated, which might in effect *mask* a bug that the user is trying to find. It also has trouble if execution halts for any reason other than a user set control breakpoint.

2.2.3 Currency Determination

Detecting and recovering from noncurrent or endangered variables is one of the most difficult aspects of transparent debugging of optimized code. Consequently several researchers have concentrated exclusively on finding good solutions to the currency problem.

The first of these was Copperman [13, 15]. He constructs a unified flow graph that contains information about both the unoptimized and the optimized programs, and how they relate. He performs a reaching definitions analysis on this graph to determine noncurrency of variables. His approach captures the effects of global optimizations, but not of translation to machine code, for example he does not deal with register allocation. He does not discuss recovery of noncurrent values. He does mention in passing the possibility of using a "representative instruction" for setting breakpoints. This is very similar to our idea of key instructions.

Wismüller also uses flow graphs to determine currency of variables [42]. Unlike Copperman, who combined all his information into a single graph, Wismüller uses separate source and object code flow graphs, and maintains information for mappings between them. This approach allows him to cope with optimizations that may make major changes to the control flow of the program. Copperman's method could not deal with such optimizations. Like Copperman, Wismüller uses a reaching definitions dataflow analysis on the graphs to help determine the currency of variables at breakpoints. In his approach, the compiler generates the graphs and debug information, and a separate program, which can be embedded in the debugger, does the currency determination. This approach makes no distinction between noncurrent, nonresident, or endangered variables. It also makes no attempt at recovery.

More recently Dhamdhere and Sankaranarayanan have proposed a dynamic approach to currency determination [17]. They create a minimal unrolled graph of the program and use timestamps collected dynamically during program execution to determine execution paths. They then perform dataflow analysis on the minimal unrolled graph for the execution path taken to determine the currency of variables at a breakpoint. They also attempt recovery of noncurrent scalar variables in certain situations. It is unclear how invasive their approach is.

2.3 Non-transparent approaches

The Convex debugger, CXdb [9, 35], is the only other commercially available debugger for optimized code of which we are aware (the DOC debugger described by Coutant, Meloy, and Ruscetta is the first). The designers of CXdb take a completely different approach to debugging optimized code. They abandon the idea of transparent behavior in favor of accurate, truthful behavior. CXdb displays the original source code for the user. However it allows the user to "see" the optimizations that have been done to the code by the use of some complex code highlighting and animation. It uses visual feedback to present the effects of the optimizations on a program's behavior. CXdb also tackles debugging optimized code at a finer granularity than is usually attempted, analyzing the code at the expression level rather than the statement level. While we believe this approach is a large step in the right direction, there are several drawbacks to the implementation. CXdb displays the original source and then attempts to explain what is actually happening in the binary by highlighting various expressions within the source statements as the corresponding instructions are executed. There are no accompanying explanations given as to why the cursor is hopping and leaping all over the code (sometimes in a rather repetitive manner). Unless the user is pretty well acquainted with the types of optimizations that might be performed on the code, she may have a hard time understanding what is going on. Furthermore this approach seems to depend heavily on the user single-stepping through the code. If the user sets a breakpoint and executes up to that point, or if the user is attempting to debug from a core dump, there is no moving cursor to aid the user in understanding how optimizations affected the code.

Cool [12] took an approach more similar to ours. He explored displaying the effects of instruction scheduling on a VLIW machine by showing the user what the transformed source would look like. His approach generates an optimized version of the source for a single optimization (instruction scheduling), and displays the optimized source side by side with the original source. He does not discuss data value or code location problems in depth. His design has not been implemented.

Another piece of work along similar lines was done by Faith [19]. In his dissertation he investigates the tracking of information necessary to provide debugging capabilities for optimizations that are performed as a set of program transformations on a tree-based representation of the program. Given such an optimizer, he defines an automatic system for tracking the necessary information. This approach is non-invasive and requires minimal assistance from the optimizer writer. As optimizations are performed, his system builds a log of the actions applied to various parts of the tree. This log can be "replayed" to transform the original program into the optimized program. Thus any particular stage of the optimization can be reconstructed if desired. This information allows an accurate mapping between the original source and the optimized target program. Any intermediate phase of the AST can be rebuilt and then translated back into a program in the source (or target) language. His system is not designed to handle currency determination, but can be extended to allow implementation of Adl-Tabatabai's currency determination dataflow analyses. His system works well for providing truthful explanations of variable values. However it does require that all optimizations be implemented as tree transformations.

2.4 Other related research

The work described in this section is not directly related to debugging optimized code, but is related in various ways to our work. Both Arsac [4] and Loveman [26] have investigated performing direct transformations on a source program to optimize the source (before it goes through compilation). Arsac selects the appropriate source transformation from a preexisting catalogue of transformations. Loveman discusses having the programmer perform (with some slight computer aid) many optimizing source transformations that require having a certain knowledge about the program which it would be difficult or impossible for the compiler to deduce without human intervention. These papers demonstrate that many interesting optimizations can be represented at the source level.

Boyd and Whalley [7] designed a debugger for their optimizer itself, rather than for the optimized program. Their optimizer, *vpo*, is designed to be very portable, and has been incorporated into C, Pascal, and Ada compilers. Their approach requires recompiling the program in order to find the errors. It finds the first transformation (optimization) that produces incorrect output. They make the assumption that if there is any observable difference in the behaviors of the optimized and unoptimized versions of the program, the optimizer must have introduced an error.

Ottenstein and Ottenstein [28] point out that in the process of performing optimizations, certain types of program bugs may become obvious which would otherwise be difficult to identify and locate. Thus they recommend performing correctness analyses while optimizing the program, to eliminate some of these bugs.

Some of the work we perform when generating the optimized source code is reminiscent of term rewriting systems and could possibly be done using such a framework [38]. In particular, rewriting multifunctional language constructs (single language constructs that perform multiple tasks, such as the `for` statement in C) in simpler terms, as well as displaying the effects of certain global optimizations could all be done using such a system.

The tracking of key instructions throughout the compiler also has a lot in common with this work, but unless the compiler were written with such a system in mind, modifying it to propagate key instructions in this manner would require too much work.

Finally there is a large body of work that has been done in the area of reverse engineering [8, 10, 11, 23, 33]. We do not use reverse engineering techniques for generating the optimized source, other than to help determine the correct order for the optimized source statements based on the optimized binary.

Chapter 3

Non-Transparent Solutions for Debugging Optimized Code

This chapter presents a non-transparent approach to debugging optimized code. It explains what an optimized source program is, how it can be generated, and the use of optimized source programs for debugging optimized code. It also introduces the notion of key instructions, and explains their importance and use. It presents solutions to the classical problems associated with debugging optimized code, as well as presenting a practical scheme for eviction recovery. In addition it discusses the role of a graphical user interface in this approach.

3.1 Solution Overview

Most of the problems that occur when attempting to debug optimized code stem from the fact that the optimized target program does not correspond very closely to the source program, and therefore none of the nice relationships used by standard debuggers exist between these two versions of the program. Most previous approaches to debugging optimized code attempt to hide this fact. In these approaches the compiler and debugger are forced to go through elaborate contortions in order to match the user's expectations about the program. Even so, these approaches do not always succeed. In some cases optimizations have changed the program so much that the user's expectations cannot be met. In those situations these approaches fall back on truthful behavior, usually telling the user that the requested task cannot be performed or the requested data cannot be retrieved. These unexpected failures can be confusing for the user.

Instead of masking the effects of optimizations, we reveal them to the user, thus giving the user a better understanding of the target program and simplifying many of the problems associated with debugging optimized code. The cornerstone of our solution is the idea that rather than the debugger attempting to meet the user's expectations of what the program state *ought* to be, the debugger should be enlightening the user as to what the program state *is*. This goal can be achieved by generating and using an optimized version of the source program. This optimized source allows the user to have a more accurate picture of the executing target program. The optimized source program, while similar enough to the original program to be recognized and understood by anyone familiar with it, will correspond much more closely to the optimized target program. Ideally this closer correspondence will allow many of the nice relationships that existed between the original source and the unoptimized target program to also exist between the optimized source and the optimized target program. The debugger can then communicate with the user about the executing target program via the optimized source. This use of the optimized source will enable the debugger to perform its tasks reasonably well with very little modification.

By far the most difficult of the problems associated with debugging optimized code is the currency problem. The root of this problem is two-fold: The traditional debugger is attempting to hide all the effects of optimizations, pretending that the original source program gives an accurate view of the executing target program; however, the source program does *not* accurately reflect the order of execution events in the target program. These circumstances lead, among other things, to variable assignments occurring out of order, with respect to the original source program. Detecting these out-of-order assignments and recovering the expected variable values constitutes the currency problem. By generating an optimized version of the source program it is possible to show the user the order in which variable assignments are actually executed. The user then expects assignments to occur in the same order as shown in the optimized source, which they do. Since there are no out-of-order assignments in this case, the entire currency problem, with all of its attendant difficulties, disappears.

Some of the advantages of using optimized source to reveal the effects of optimizations include:

- This approach allows for a simple solution to the location problem, since statements in the optimized source occur in the order in which they are executed in the target program.

- This approach eliminates the currency problem entirely, as there are no out-of-order variable updates in the optimized source.
- Unlike some approaches, this one does not require a tight coupling of the compiler and the debugger. Also it can be retrofitted to existing compilers and debuggers.
- By revealing some of the effects of optimization, such as a piece of code that is unexpectedly dead, this approach may allow users to find certain errors more easily than they could using a transparent approach.
- By displaying some of the effects of optimization, it becomes easier to identify certain types of bugs within the optimizer itself, and to determine whether the error existed in the source program or was introduced by the optimizer.
- Allowing programmers to see optimization effects helps them learn to write more efficient programs. Seeing how the compiler optimized their programs may point out issues they had not previously considered.

Revealing some of the effects of optimizations is not a completely novel concept. In her dissertation, Zellweger states, "A major stumbling block when debugging optimized code is that the programmer does not know what the transformed program looks like."[47, p. 85] Zellweger then goes on to consider the possibility of "an automatic source-translator that could give the programmer a new version of the source program that showed the effects of optimizations". She discusses the major problems she sees with such a proposal, and ends up rejecting the idea. At the time she wrote her dissertation, this conclusion was reasonable. However the state of computer science, particularly with respect to user interfaces, has changed dramatically since then, and what was infeasible in 1984 is now practical and achievable.

Zellweger's main objection to this approach is that the user will be confronted with a new program which she did not write, and she will have to expend large amounts of time and energy understanding this new program and how it relates to the original source. No useful debugger activity can take place until the user has grasped how the new program works and its relation to the original. This problem would indeed have been insurmountable in the early 1980's. With the advent and proliferation of graphical user interfaces, windows, mice, screen pop-ups, etc. it is now possible for the computer, via a good interface, to assume a large portion of this task, displaying relationships between the optimized source and the original source in a variety of ways. The user's burden of understanding the optimized source can be minimized, and this approach becomes viable.

3.2 What is Optimized Source?

Up to this point we have been talking about an optimized source program without giving a really good definition of what we mean. Although the phrase "optimized source program" could be applied to a source program that has undergone any number of modifications and optimizations, our concern is primarily with user-visible structural changes. The relevant modifications are obtained by applying a series of transformations to the source program to reflect the optimizations performed by the compiler. These transformations can include reordering the code, rewriting expressions, replacing language constructs with different but equivalent constructs, duplicating code, or deleting code. As the name implies, the optimized source program is a partially optimized version of the original source program.

To make some optimization effects visible, the original source language may need to be extended by pseudocode, and additional comments may be inserted into the optimized source code for explanatory purposes. The need for pseudocode will depend on the source language and the optimizations to be shown. For example, if the source language is FORTRAN, and one were trying to show a strength reduction optimization explicitly in the source, one might need to introduce pointers into the optimized source. Since standard FORTRAN does not contain pointers, this case is one where the introduction of pseudocode would be appropriate. Obviously if pseudocode is used in the optimized source, that code will not compile. Since its purpose is to convey information, serving as an intermediary between the original source code and the target code, non-compilability is not a problem. For some languages, such as C, there may be no need to introduce pseudocode.

Since one knows at the time one is designing the optimized source generator what the source language and the selected optimizations are, one ought to be able to determine ahead of time what types of pseudocode will be required, and which situations will require it. This advance knowledge allows for careful design of the pseudocode, keeping the use of such constructs to a minimum. We do not advocate an ad hoc use of pseudocode, as introducing too many foreign constructs to a user may confuse her too much, rendering the optimized source useless.

Given that we are trying to generate a source program that more closely resembles the target program, and given that reverse engineering, a moderately well understood discipline, generates source programs from target programs, a natural question is why not use reverse engineering techniques to obtain an optimized source program directly from the optimized target code [8, 10, 23, 33]. The reason for not doing so is that the new program must be recognizable and understandable by someone familiar with the original program. Due to the loss of some types of control flow data, as well as the nature of certain optimizations such as software pipelining and loop optimizations, code generated by reverse engineering is likely to be unfamiliar and hard to comprehend. To reiterate a very important point, if the user cannot understand how the optimized code relates to the original program, then the user will have difficulty in determining what meaning and significance to attach to it or how to use it. By starting from the original program and gradually transforming it we retain all the parts of the original program that were not involved in the compiler optimization process, such as comments, declarations, or pre-compiler directives, as well as high level constructs such as structured control flow statements, which get lost in the compilation process. By keeping these features of the original program, the modified source code is more easily recognizable. Also, by starting from the original program, irrelevant information about the optimizations can be suppressed. Using reverse engineering, one has no such option.

Not all optimizations can be expressed at the source level, nor is it desirable to do so. Fine-grained instruction scheduling is an example of such an optimization. Another example is register allocation and spilling. If the modified source code were required to express all the effects of all optimizations, assuming that were possible, the final result would be very similar to the assembly code, thus defeating the purpose of using the source language. This situation raises two important questions: which optimizations should be considered when deriving the optimized source, and how can the effects of these optimizations be reflected in the source language. First we will address the choice of optimizations. Showing the effects of optimizations is discussed in Section 3.5.

When deciding which optimizations to reflect in the optimized source, one should keep in mind its purpose. The optimized source program shows the user what is happening in the target program, but *at the source level*. The user wants to know which source statements are executing and when they are executing. The user needs an accurate idea of which variables exist at any given time and what values they contain. Users need not see low level machine specific optimizations. Therefore the optimizations of interest are those that visibly affect source code constructs, namely those that eliminate source code, move source statements, or change the form of source statements (e.g. altering expressions to use different constants, variables, or operators). One needs to be careful in considering optimizations that change the form of source statements. Some changes, such as replacing multiplication with register shifts, are probably not relevant, while others, such as using a different constant or variable in an expression, are. The optimizations that we have focused on in our implementation are code motion, coarse-grained instruction scheduling, common subexpression elimination, partial redundancy elimination, copy propagation, and dead code elimination. Other important optimizations that visibly affect source constructs include function call inlining and high-level loop optimizations.

There are two basic requirements the optimized source code must meet in order to be useful:

- It must be recognizable and understandable by someone familiar with the original program.
- It must correspond closely enough to the optimized target program to allow for a consistent and meaningful mapping between the two.

These two requirements are in direct conflict with each other. One of the difficult and interesting research issues has been to find an appropriate balance between them. The more one modifies the original program to correspond to the target program, the less recognizable and understandable it is likely to become. Referring to the transparency continuum shown in Figure 1.6 and reproduced here as Figure 3.1, one could visualize that at one end of the continuum is the original source code, and at the other end is a program obtained by reverse engineering the optimized target program. Our concept of optimized source code falls somewhere between the two – the exact point on the continuum will vary depending on the intended use and audience.

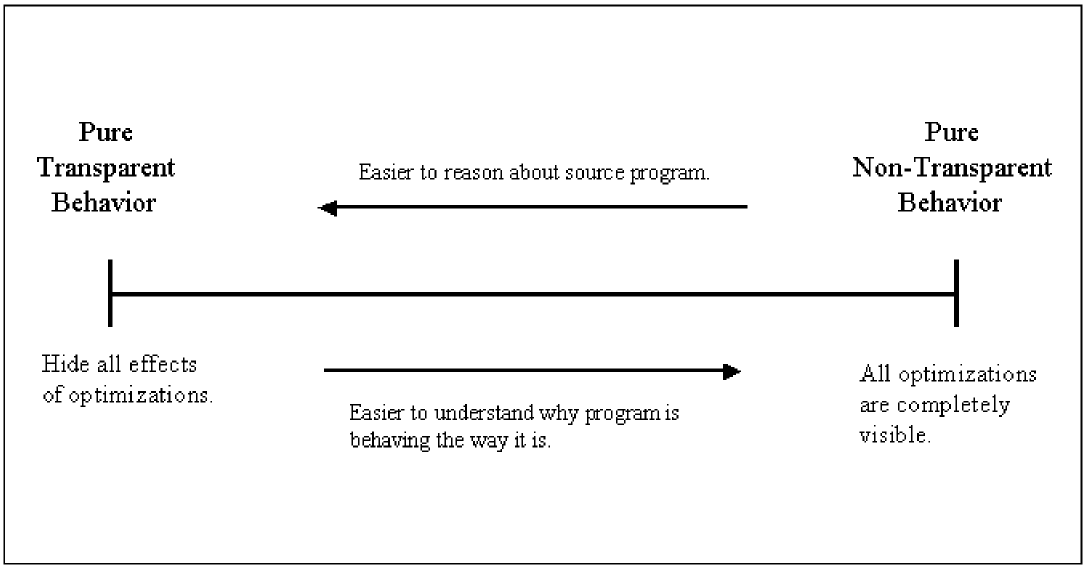


Figure 3.1: The Transparency Continuum

(a)	(b)

`j = num++ * 3 + (int *)str[0];`	`I0: lbu $21,0($5)`
	`I1: mult $1,$4,3`
	`I2: addu $18,$21,$1`
	`I3: addiu $16,$4,1`

Figure 3.2: Selecting key instruction for assignment statement

3.3 Key Instructions

A critical characteristic of the optimized source is that the order of statements in the source must reflect their order of execution. Determining the order in which the source statements will be executed becomes complicated after optimization, as there is no longer a clear concept of individual source statements in the target code. The instructions for statements have been broken up, duplicated, recombined, and interleaved, making it difficult to state where one statement ends and another begins. In order to deal with this problem we use *key instructions*. For any source statement, its key instruction is the single low level instruction that most closely embodies the semantics of that statement. For example, the key instruction for an assignment statement is the one that stores the assignment value either to the variable's location in memory, or to a register (if the write to memory has been eliminated). The key instruction for a function call is the jump to the code for that function. The concepts underlying key instructions are not entirely new. Both Zellweger [47, p. 59] and Copperman [15, p. 55] mention something similar when discussing possible implementations of semantic breakpoints. Zellweger suggested using instructions that show the *key statement effects* as one reasonable method for implementing semantic breakpoints. Copperman mentioned the possibility of a *representative instruction*, the "instruction generated from a statement S that best corresponds to the statement boundary". However neither of them pursued the concept or fleshed it out.

Identifying key instructions for most types of statements is straightforward. Given the statement type and the set of instructions generated from the statement it is easy to pick out the key instruction. For control flow statements it is the first branching instruction; for function calls it is the jump to the code for the function. The key instruction for an assignment statement, on the other hand, is more difficult to identify. Figure 3.2 illustrates this point. Figure 3.2(a) shows an assignment statement in C. Figure 3.2(b) shows the optimized instructions generated from that statement. The variable *num* begins in register 4, and the variable *str* begins in register 5. The first instruction, I0, gets the value corresponding to "str[0]" and stores it in register 21. The next instruction multiplies num by 3 and saves the result in register 1. The third instruction adds the results of the first two, leaving this sum in register 18. The final instruction then adds one to the value of *num* and puts the result in register 16 (which now becomes the new location for *num*). By doing this careful semantic analysis of the instructions, one can quickly conclude that the key instruction for the assignment statement in Figure 3.2(a) is I2, which calculates the entire right-hand side of the assignment and leaves the result in register 18 (the new location for *j*). However the *only* way to select the key instruction for this statement is through such a semantic analysis of the source statement and the instructions generated from it. There are three circumstances that contribute to this difficulty in identifying key instructions for assignment statements. First, optimizations often eliminate the instruction that writes the value to the variable's location in memory, as is the case in this example. Thus the key instruction will be the instruction that calculates the final value for the right-hand side of the assignment statement. It could be any one of a number of operator instructions, any of which could occur multiple times in the calculation (in the example shown, the key instruction is an addition instruction). The upshot is that one cannot rely on the type of the instruction for identifying key instructions for assignment statements. Neither can one use variable names to sort out the key instruction, because there are no variable names in the instructions. This lack of variable names is the second contributing factor. Finally, one cannot rely on the order of the instructions to identify the key instruction for an assignment statement. While the key instruction is likely to be among the last few instructions for the statement, it is not necessarily the final one. The example in Figure 3.2 shows a circumstance where the key instruction is *not* the last instruction for the source statement. The combination of these three circumstances makes it impossible to select the key instruction for assignment statements purely by examining the instructions. As stated above the only way to select the key instruction for an assignment statement is to carefully examine the semantics of the source assignment statement and of each of the instructions. Doing so would require a complicated analysis. However the compiler itself *does* perform the required analyses earlier in the compilation process, when it first parses the source and generates the internal representation of the program. Therefore it makes sense to take advantage of the compiler. In our implementation of these ideas we have the front end of the compiler tag the pieces of the internal representation that will eventually become these key instructions. It tags them when the internal representation is first generated. These tags are then propagated through the compilation as the optimizations are being performed, and when the final instructions are generated, the key instructions for assignment statements are already tagged.

Key instructions turn out to have a variety of uses. They represent the locations in the target program where their corresponding source statements execute. Therefore they are the obvious choice for setting semantic breakpoints, thus solving the most difficult part of the location problem. Also the execution order of the key instructions indicates the order in which the corresponding statements should occur in the optimized source. Lastly by indicating which instructions correspond to assignments to source variables, key instructions allow a much simpler, more robust scheme for collecting variable location information than was previously possible (see Section 3.6 for details).

3.4 Generating Optimized Source

During the optimization process, the source program is first translated to an internal representation, and then portions of the internal representation are moved, duplicated, separated, eliminated, or altered. In order that the optimized source program be recognizable to someone familiar with the original program, fragments of the original program should be used as much as possible to construct the optimized source code. The optimized source may contain original source statements that have been reordered, modified slightly, or split apart. The optimized source may also contain new statements inserted to make particular optimization effects explicit. As mentioned earlier it might contain fragments of pseudocode designed to explain important optimization effects which cannot be adequately described using constructs in the source language. In addition, the optimized source program must somehow indicate original source statements that have been eliminated.

Before the effects of optimizations performed by the compiler can be reflected in the optimized source, the tool generating the optimized source must know *precisely* what optimizations (at least of those on the selected list) the compiler performed where, and precisely what it did (e.g. it substituted x for y at line 73). Therefore the compiler must collect detailed information about the optimizations of interest as it performs them and pass this information on to the tool that generates the optimized source. Decoupling the performance of the individual optimizations from the generation of the source that shows the optimization effects, has several benefits. This approach allows the generation of the optimized source to be completely independent of the implementation details of the optimizations. It is a more robust approach, as it is less likely to break every time the compiler writer wishes to improve the optimizer. Furthermore it allows the generation of the optimized source to be postponed until *all* optimizations have been performed, thus making sure that the optimized source is as accurate as possible, and allowing optimizations whose effects were overwritten by later optimizations to be “skipped” when generating the optimized source (one can go directly to presenting the effects of the later optimizations).

3.4.1 Reordering source statements

An essential requirement for the optimized source is that the order in which statements occur in it accurately reflect the order in which the instructions for those statements will be executed. In order to meet this requirement, one must have knowledge about how the compiler rearranged the statements and some mechanism for identifying source statements in the optimized internal representation. Key instructions have been introduced precisely for this purpose. The compiler can tag the key instruction(s) in the generated target program for each source statement that was not eliminated. The execution order in which these key instructions occur in the target program then dictates the order in which their corresponding source statements should occur in the optimized source code.

3.4.2 Modifying source statements

To show the effects of certain optimizations, or to preserve the semantics of the program, some original source statements may need to be modified. Figure 3.3a shows a few lines of code from a source program. After these lines have completed executing, the variable i should contain the value of “3 * y - 17” and the variable *num* should contain the value of “y + 1”. During the optimization process, the order of these calculations was changed, as shown in Figure 3.3b. To preserve the semantics of the original program it is necessary to change the constant from 17 to 20, to account for the fact that *num* is now incremented *before* the value of i is calculated. Figure 3.3c shows the correctly updated

```
(a) Original Code

  num = y;           /* num == y          */
  i = 3 * num - 17;  /* i   == 3 * y - 17 */
  num++;             /* num == y + 1      */

(b) After Reordering (Wrong)

  num = y;           /* num == y                */
  num++;             /* num == y + 1            */
  i = 3 * num - 17;  /* i   == 3 * (y + 1) - 17 */
                     /*     == 3 * y - 14       */

(c) Reordered & Updated (Correct)

  num = y;           /* num == y          */
  num++;             /* num == y + 1      */
  i = 3 * num - 20;  /* i   == 3 * y - 17 */
```

Figure 3.3: Reordering & Updating Code

code.

3.4.3 Inserting new code

There are transformations such as common subexpression elimination for which new statements must be added to the source code to illustrate the optimization. For example, an assignment statement might be inserted at the point where the subexpression is evaluated, assigning the subexpression to a newly generated variable. This new variable then can be substituted throughout the modified source wherever the original subexpression evaluation is eliminated by the compiler.

3.4.4 Eliminating source code

A standard feature in most optimizing compilers is elimination of dead code. The optimized source code must make clear which source statements were eliminated by the compiler. There are two alternative methods for presenting source code elimination to the user. The first is to eliminate the dead source statements from the optimized source program entirely. This approach does not seem like a very good solution, as the user may not realize immediately that a particular source statement has been eliminated entirely from the program; the user may assume instead that the statement has been moved to some other portion of the program. In such a case she may examine the entire program before realizing that a particular statement is not in the optimized source. Furthermore upon realizing that the statement is missing, the user still may not attach the desired significance to the discovery. She might assume instead, for example, that the compiler or the optimized source generator made some kind of mistake. The alternative method is to surround the dead statement with comments, indicating that the statement is dead. The comments imply the lack of corresponding instructions in the target program, and can include an annotation explicitly stating that the statement is dead. This approach seems better than the first, as it is more clear and does not force the user to guess or deduce what has happened to the dead code.

3.4.5 Splitting apart statements

High-level, powerful source languages often contain single constructs that embody multiple pieces of functionality. The optimizer may scatter the functionally separate pieces of such a construct widely throughout the optimized target program. Since the task at hand requires that the optimized source program accurately reflect the location and order in which these functional events occur in the target program, it becomes necessary for the optimized source to split apart these multi-functional constructs. A simple example should clarify this point. Figure 3.4a shows a common C construct, the `for` statement. The `for` statement header contains three parts: the initialization statements, the loop test, and the increment statements. In order to allow the modified code enough flexibility to mirror the order of events in the target program, such a construct needs to be rewritten as simpler constructs, each embodying a single piece of functionality. Figure 3.4b shows the same `for` loop, rewritten as multiple C statements. The modification allows these various statements to be moved and reordered as necessary in the optimized source code. When designing a tool to generate optimized source programs for a particular language, one needs to carefully consider which language constructs will need to be broken down and split apart in this manner. Ideally whenever such a multifunctional construct needs to be rewritten, the source language will contain simpler constructs which can be used for this purpose. If that is not the case, pseudocode can be used to represent the pieces of functionality that need to be separated. Figure 3.4c shows the same `for` statement split apart and represented with pseudocode.

```
for (current = list;
  current;
  current = current->next) {
   ...
}
(a) Original Statement
```

```
current = list;
while (current) {
 ...
 current = current->next;
}
(b) Using C Constructs
```

```
current <--  list;
loopwhile (current) do
 ...
 current <-- current->next;
od
(c) Using Pseudocode
```

Figure 3.4: Rewriting `for` statements.

3.5 Solving the Wandering Data Problem

A critical feature of any debugger is its ability to return the value of a variable in response to a user's request. However, as explained in Chapter 1, once a program has been optimized this task becomes complicated by the fact that a single variable's value may reside in one of several different locations, including registers, depending on which portion of the target program is currently executing. In light of this situation, the type of information generated in symbol tables for unoptimized programs is completely inadequate for optimized programs. Rather than a single location for each variable, the debugger needs to know the whole list of possible locations, as well as which location is valid for that variable for which portions of the target program. What is really required is a *variable range table*, such as the one described by Coutant, Meloy and Ruscetta [16]. The basic idea is to create a table that indicates for every variable in the program, for every location in which the variable resides, the range of target code instruction addresses for which the variable resides in that location. In the Coutant, Meloy and Ruscetta paper, a location could be one of three basic types: memory address, register, or constant. We have modified this scheme slightly, adding a fourth type: evicted. A flag in the table for each entry indicates which type of location the entry represents. The type determines how the data field is to be interpreted. This table must be created and filled in by the compiler, as only the compiler has access to all the data necessary. The table is then passed to the debugger, which uses it to find the variables.

3.6 Eviction Recovery

Variable eviction is a widespread phenomena in optimized programs, occurring at 30-60% of all possible breakpoint locations[1]. Since returning a variable's value in response to a user request is one of the most basic required core pieces of functionality for a debugger, a useful debugger of optimized code should have some method for dealing with the variable eviction problem. The simplest method would be to merely keep track of which variables are evicted at each location in the target program. In response to a request to see a variable's value, if the variable is evicted, the debugger can inform the user that the variable's value is not currently available. While relatively easy to implement, this approach is not likely to be satisfactory. Given the frequency with which eviction occurs, it is probable that a large number of user requests would receive such a response. Another option is to keep track of the pieces used to obtain the variable's value before it got overwritten, and to attempt to recreate its value from the pieces. There are several problems with this approach. To begin with, keeping track of all the components of each variable update in the program would require a huge amount of additional data. Second, even if that data were collected and passed to the debugger, it is quite possible that one or more of the "pieces" will have been updated since the evicted variable received its value. In fact it is likely that at least some of the pieces themselves have been evicted. The sheer volume of information required by this approach makes it impractical. A third approach is to make a copy the variable's value before it gets overwritten. This copy could then be returned in response to user requests.

This last approach is the one we take. However even this approach has some problems.

In order to make a copy of the variable's value before it is overwritten, execution of the target program must be suspended prior to the eviction and control given to the debugger. This requirement in turn means that the debugger must set breakpoints on all instructions that evict variables. Furthermore these are not user-specified breakpoints; since the debugger does not pass control to the user at these positions (unless such a breakpoint happens to coincide with a breakpoint set by the user), they are *hidden breakpoints*. The user is not supposed to know that they exist. There is a problem with introducing hidden breakpoints. Recall that one of the desirable characteristics of a solution for debugging optimized code is that the debugger be non-invasive, not altering the target program (except to implement user-requested breakpoints). Every time a hidden breakpoint is reached, the computer must switch contexts and give control to the debugger, which will collect the appropriate value and return control to the computer, which then resumes executing the target program. Given the high frequency of variable eviction, using hidden breakpoints to save the value of every variable that is about to become evicted will have a highly visible, negative impact on the running time of the program. Thus there is a conflict between needing the ability to present evicted variables' values to the user, and keeping the impact on the execution speed of the target program to a minimum. We have not been able to find an optimal resolution for this conflict, so we have settled for what seems to be a reasonable compromise: The debugger will use hidden breakpoints to copy the values of evicted variables, but only in those subroutines where the user has set a breakpoint. Thus, while the running time for those subroutines will be adversely affected by the eviction recovery scheme, the rest of the program will not. Since, by choosing to set a breakpoint in the subroutine, the user has already indicated a willingness to slow down (stop) execution time within the subroutine, this tradeoff seems fair.

The benefits of this approach are that, whenever execution of the target program stops at a user-specified control breakpoint, the values of all variables within that subroutine's scope will be available for the user to examine. The values of any arguments to any functions on the call stack will also be available for examination, as they must be live (they were used in the function call), and live variables cannot be evicted. By limiting eviction recovery to subroutines where the user has set breakpoints, the impact of eviction recovery on program run time is minimized.

Obviously this approach also has some limitations. Target program execution can be suspended for reasons other than a user-specified control breakpoint (for example, a data breakpoint, a user interrupt, or a fault). In cases where execution is halted for some other reason, eviction recovery may not have been performed, so variables within the current subroutine may or may not be available (depending on whether there happens to be a user-specified control breakpoint somewhere within the subroutine or not). Another problem is that, when moving up or down the call stack, variables that are *not* arguments to the function calls may or may not be available. Similar problems also exist for debugging core dumps (post-mortem debugging). Furthermore execution time will be affected in those subroutines where eviction recovery is being performed.

3.7 User Interface Options and Issues

Given the fact the user is being presented with a fair amount of new data, and given the underlying necessity of not confusing the user or placing an undue burden of optimization understanding on her, the user interface is critical.

To begin with it is vital that the user be able to relate any position or statement in the optimized source back to the corresponding statement in the original source. A graphical interface can greatly facilitate this task by displaying both versions of the source, and allowing the user to select any statement in either version. Once a statement has been selected the interface can display the corresponding statement(s) in the other version. Thus if the user, looking at the optimized code, cannot find a particular statement that is known to be in the original source, the user can go directly to the original source, select the statement in question, and have the corresponding statement(s) in the optimized source displayed for her. Similarly if the user is looking at a particular statement in the optimized source and cannot figure out where it came from, she can select it, and the corresponding original source statement(s) will be displayed.

This scheme works well if one is only dealing with optimizations that do not drastically alter the appearance of the source program. Optimizations such as function call inlining and high-level loop nest optimizations, however, *would* require making major changes to the appearance of the source program in order to make their effects explicit. Thus a graphical user interface will need to do more work to keep the user from being overwhelmed by the differences between the optimized and the original source programs, once such optimizations have been applied.

One preliminary design for helping with the effects of function call inlining involves generating *two* pieces of optimized source for the call site. One piece, the default, would be similar to the original source program (modulo other optimization effects), i.e. it would show the function call as if it had not been inlined. Beside the function call there would be a flag or icon indicating that this default optimized source is not completely accurate. The user would have the option of requesting to see the fully optimized source. Selecting this option could cause another window to pop up displaying the full effects of the function call inlining. The user could make this pop up window go away again, if desired. Within this new window the user could perform standard debugging activities with respect to the fully optimized source, such as single-stepping or setting breakpoints. When looking at the default optimized source, attempting to single-step into the inlined function call would cause the fully optimized source to appear. The inlined function call would appear on the call stack, but would be annotated as "inlined".

The advantage of an approach like this one is that the default view of the optimized source remains very similar to the original source, which in turn minimizes the difficulty users will have in recognizing and understanding it. At the same time, in those instances where the user needs a truly accurate picture of the effects of function call inlining, they are available to be seen. Even in the default view, though, some effects of function call inlining may need to be represented. For example the inlining may enable certain optimizations such as constant folding or common subexpression elimination to cross the boundaries between the call site and the inlined function. The effects of such cross-boundary optimizations must be faithfully represented in the default code surrounding the call site, as it is critical that the user have an accurate picture at all times of what values get assigned to which variables.

Using dual versions of the optimized source as advocated in the previous paragraph addresses the most critical problem for displaying the effects of function call inlining, namely allowing the user to have accurate information about the optimizations performed while not overwhelming her by drastic changes to the source program. However this approach also raises some difficult technical questions. Some of these questions include how to store and access two separate versions of the optimized source; how to assign source line numbers to both versions of the source and to report those line numbers to the symbol table for the debugger; how to assign the corresponding source positions to the instructions in the target program; and how to divide the work of using dual versions of the optimized source between the graphical user interface and the debugger.

With regards to storing and accessing the two separate versions of the optimized source, it might be possible to store them both sequentially in the same source file. There could be special delimiters added to the file to separate the two versions. The user interface would then use these delimiters when reading in the source file and display the appropriate pieces of code in the appropriate windows. Assigning line numbers to the two versions of the optimized source is somewhat tricky, as the line numbers for unique lines in each version should be distinct and yet should somehow all fit logically within the line numbers assigned to code surrounding the call site. Statements that occur in both versions of the optimized source should have the same line number in both. Perhaps in the default view the function call itself could be given a *range* of line numbers, representing the line numbers in the fully inlined version. The source positions assigned to the instructions in the target program should correspond to those in the fully optimized version of the source. The user interface will have to keep track of which version the user wants to display, and will have to parse the optimized source file to find and separate the dual versions of the optimized source. The debugger will also have to keep track of which view the user has of the source, for performing tasks such as displaying stack traces. The debugger will further need to implement some scheme for stepping over the inlined function call in the default view.

High-level loop nest optimizations cause the same types of problems as function call inlining. Thus a similar solution, namely dual versions of the optimized source might be an adequate approach. If users really want to see all the details of the optimizations, they can step through the fully optimized view of the loop. If the user is uninterested in the optimizations performed on the loop, the user can use the default view of the loop (which will look very similar to the original source for the loop) and skip over the loop. The worst difficulties arise if the user attempts to single step through the default view of the loop. At this point there are several possibilities. The debugger could insist that if the user wants to single step through the loop the user must use the fully optimized source. Alternatively the user would have the option of stepping over the entire loop, treating the whole thing as one giant statement. If the debugger allows the user to step through the default view of the loop, the user could be warned that the information given when within the loop will be inaccurate, and the debugger could attempt to give the user clues such as "updating induction variable", "executing first, third, and fifth iterations of the loop", "modifying array elements 3, 8, and 11", etc. This is all very preliminary.

Once one starts considering using multiple versions of the optimized source to help users understand and deal with optimizations, an attractive idea is that of allowing the user interface itself to transform the original source to the optimized source in a series of layers or steps. The user might be given control over applying or removing any particular optimization. Thus by "stepping" through the various stages or layers of the transformation, the user could gain a better understanding of the optimized source and how it relates to the original. The greatest flaw with this idea is that, because the user is in control of which optimizations do or do not get applied, the resulting optimized source may not present a very accurate picture of the optimizations performed by the compiler, and it may correspond very poorly to both the target program and to the information in the symbol table. Therefore this idea does not seem practical for a debugger. However it might be useful as a secondary tool designed to teach people about optimizations or to help them understand the optimized source that is generated by Optview. For such a tool one would not need to be concerned about maintaining correspondences with the symbol table or with the target program.

3.8 Summary

In this chapter we have presented a theoretical overview of our solution to the problem of debugging optimized code. We have explained what we mean by "optimized source" and have discussed in detail the theoretical issues involved in generating optimized source. The concept of key instructions has been introduced and we have shown how they can be used to solve multiple problems. We have explained the use of variable range tables to solve the wandering data problem, and we have presented a practical scheme for performing eviction recovery in the debugger. Finally we have discussed the various options available for a user interface for our solution and shown how they can minimize user confusion with this approach.

Chapter 4

Optview and Optdbx

The theoretical solutions outlined in the previous chapter have great potential for easing the task of debugging optimized code and for allowing simple, practical solutions to the problems in this area. A critical component of this solution, however, is the ability of users to recognize, understand, and use the optimized source. The only way to determine conclusively whether users can or cannot understand the optimized source is to generate such source and show it to them. Therefore we have implemented the solutions described in Chapter 3, and have distributed our implementation to a set of users. In this chapter we describe the details of our implementation, as well as problems encountered during the implementation process.

4.1 Overview

There are three basic components to our implementation: the compiler, Optview, and Optdbx. Optview is the prototype tool we have written to generate optimized source code for C programs. In our implementation we chose to embed Optview within the compiler, although it is not really necessary to do so. Optview could have been written as an entirely separate tool, at the cost of the compiler having to write out information that it currently passes directly to Optview. The compiler that we modified is the MIPS Pro 7.2 C compiler, an aggressively optimizing commercial compiler developed by Silicon Graphics, Incorporated (SGI). In addition to collecting data for and invoking Optview, the compiler was modified to gather and generate the extra symbol table information necessary for accurate debugging of optimized programs. Optdbx is a debugger that uses optimized source for debugging. It consists of a graphical user interface, written in Python 1.5.1, which uses Tk/Tcl 8.0.4, and a modified version of the SGI 7.2 Dbx debugger. Optview and Optdbx were developed on an SGI MIPS R4000 processor, running IRIX 6.2 (a version of Unix).

4.2 Optview Overview

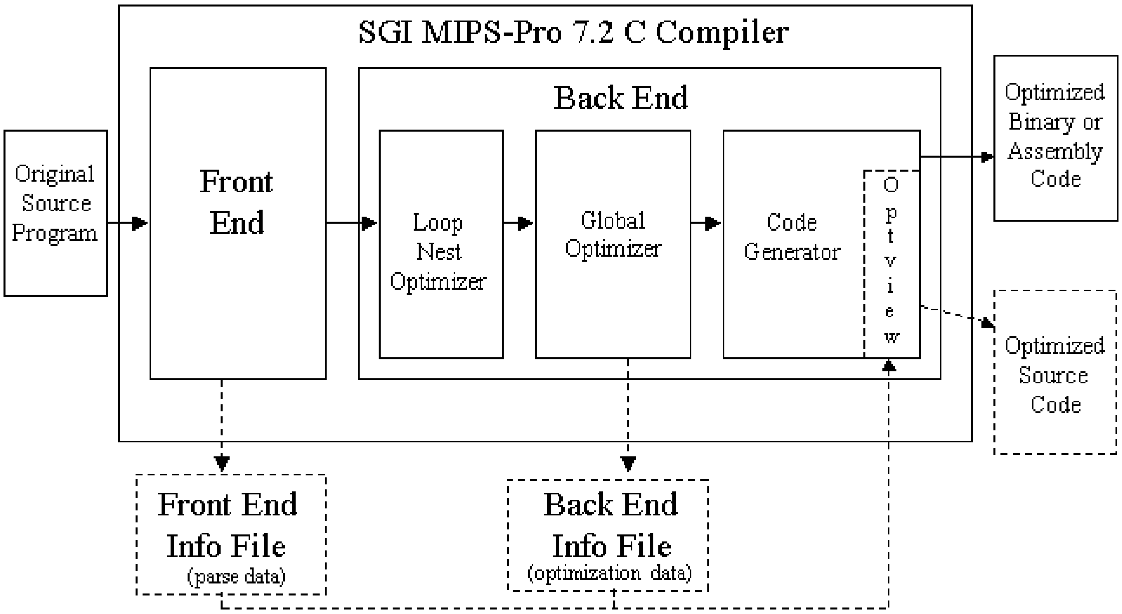


Figure 4.1: How Optview relates to the rest of the compiler

Figure 4.1 gives a high-level overview of how Optview fits into the compiler. The solid black lines indicate pieces of the original compiler, and the dotted black lines indicate pieces added for Optview. A source program is passed into the front end of the compiler, which, in addition to its standard syntactic and semantic analyses, collects parsing information about C language constructs that may later need to be rewritten or split apart. In particular it collects information about the location of increment and decrement operators (`++`/`--`), loops (`for`, `while`, and `do-while`), conditional expressions, function start and end positions, and declarations. This collected information is written to a temporary file, the Front End Information File. The front end of the compiler then passes its internal representation of the program to the back end of the compiler. High-level loop optimizations have not yet been implemented in Optview, so the back end currently skips over the loop nest optimization phase and proceeds to the global optimization phase. This stage is where a large number of the optimizations important to Optview are performed. The global optimizer, in addition to performing its usual optimizations, collects data Optview needs about the optimizations as it is performing them. This information is written to another temporary file, the Back End Information File, as a chronological log of the actions performed by the compiler. The optimized internal representation of the program is then passed to the code generator, which proceeds with lowering the internal representation and performing such optimizations as register allocation and assignment, and instruction scheduling. Immediately prior to emitting the optimized assembly or target program, the code generator invokes Optview. Optview takes a copy of the original source program, the front end information file, the back end information file, and the final internal representation of the program, and from these pieces Optview generates the optimized source program. It does so by starting with a copy of the original source program and performing a series of rewrites or forward transformations on the source. All of these transformations take place at the source level. Optview relies on the back end information file to determine exactly how to transform the source. In the process of generating the optimized source program, Optview identifies all the pieces of the internal representation that represent key instructions. Doing so is necessary, as Optview uses the order of the key instructions to determine the correct order for source statements in the optimized source. After Optview finishes generating the optimized source, it is written out to a file. The code generator then proceeds to emit the optimized binary or assembly program.

Given that the goal of Optview is to produce a source program that accurately reflects what is occurring in the binary, a reasonable alternative approach might have been to use reverse engineering techniques to obtain the optimized source directly from the optimized binary. Optview does not take this approach for several reasons. The optimized source must meet two critical conflicting goals. It must reflect the optimized binary program accurately enough to allow a relatively simple mapping between itself and the binary, yet at the same time it must remain recognizable and understandable to anyone familiar with the original source program. If the users cannot recognize and understand the optimized source, then the system is useless. Although reverse engineering would meet the first condition admirably, accurately reflecting the binary, it would fail to meet the second. In reverse engineering one starts with a very low-level representation of the program. Thus, for example, all higher level control flow constructs, such as conditionals and looping statements have been converted to goto's. Also, non-executable portions of the original source program such as white space and comments are not present in the program representation. They have been lost. The loss of high level control flow constructs and of comments and white space makes the resulting program much harder for the user to recognize. To further complicate the matter, a reverse engineered program must reflect *all* optimizations performed by the compiler, as the binary from which it starts contains all the optimizations. By starting with the original source program and performing a series of transformations on that program, Optview has the option of picking and choosing which optimizations to reflect in the optimized source. It also is able to retain and use many pieces of the original program, including high-level control flow constructs and comments. The net result is that programs generated by Optview bear a much closer resemblance to the original source program than programs reverse engineered from binaries.

4.2.1 Selecting Optimizations to be Made Explicit

In order to prevent the optimized source from becoming completely unrecognizable, one must carefully select the optimizations whose effects are to be reflected in the optimized source. As mentioned previously this selection is *not* in any way limiting the optimizations the compiler will *perform*; it will continue to execute its full range of optimizations on the program. We are only selecting which optimizations need to have their effects reflected in the optimized source. The optimizations we chose to reflect in Optview are copy propagation, constant folding, common subexpression elimination, partial redundancy elimination, dead code elimination, code motion (hoisting and sinking), and instruction scheduling (at a coarse granularity)[1]. These optimizations were selected as having the most impact on the

[1]When we say Optview reflects instruction scheduling at a coarse granularity, we mean it reflects the instruction scheduling performed on key instructions, ignoring scheduling performed on the other instructions. The reason for doing so is that each source statement can have multiple corresponding instructions in the target program. After instruction scheduling the instructions for any given source statement may be interleaved with instructions from several other source statements. Also a single instruction may be

ability of the debugger to map statement locations and data between the source and target programs. Our original list also included function call inlining and loop nest optimizations, as these also have a large impact on debugging. We intended to incorporate inlining and loop nest optimizations into Optview in a second phase of implementations, as these two optimizations would cause more drastic changes in the appearance of the optimized source, and would therefore require a more complicated user interface for displaying their effects to the user. At this time we have not had the opportunity to add these two optimizations to Optview.

4.2.2 Optview Details

In this section we first give a brief overview of the stages Optview goes through when transforming a source program into optimized source. We then proceed to describe each stage more fully.

Optview transforms the original source program into optimized source in a series of eight distinct steps. Figure 4.2 illustrates these steps.

Step One. In the first step, Optview reads in the original source file and stores the text and some additional information for each source line in an array, one entry per source line. Each source line is labelled as being white space, a comment, a declaration, a pre-processor directive, or executable code.

Step Two. Once the original program has been read, Optview rewrites all of the "multifunctional" C constructs used in the program (`for` loop headers, conditional expressions, increment or decrement operators that are embedded within other statements, and initialized declarations) in terms of simpler constructs that can then be separated and moved around more freely in the optimized source. Optview uses the data file that was created earlier by the front end of the compiler to locate the appropriate constructs in the source (recall that Optview is not invoked until the very end of the compilation process, immediately prior to the target program being emitted).

Step Three. The next step is to determine the new order in which statements should appear in the optimized source. The new order is determined from the order of instructions in the target code as follows. For each source statement that has corresponding instructions in the target program, a key instruction is identified. Once every source state-

attributable to multiple source statements. It is therefore impossible to accurately reflect the total ordering of instructions at the source level.

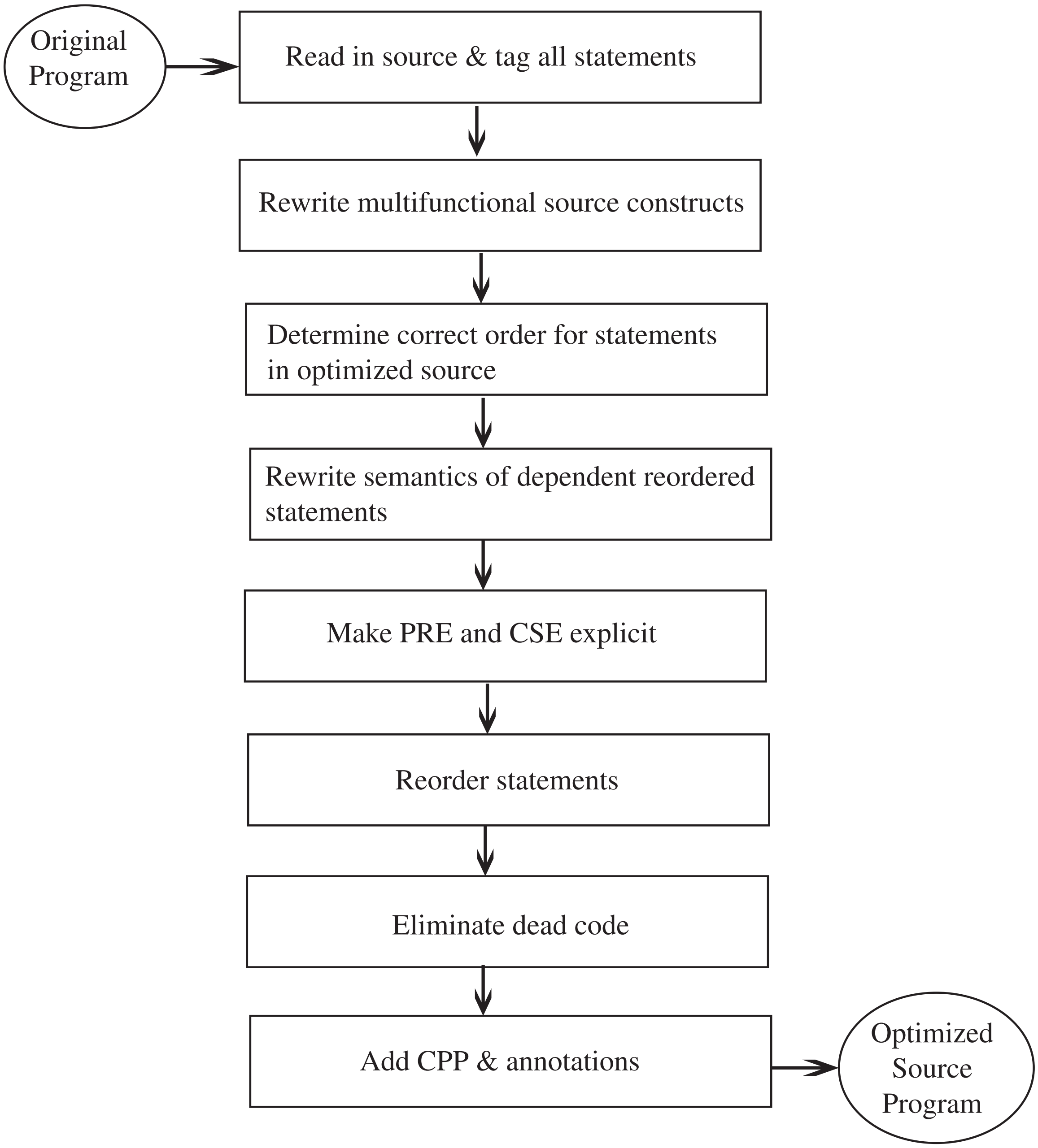


Figure 4.2: The stages of Optview

ment has a key instruction associated with it, the execution order of these key instructions in the target code becomes the new order for the source statements in the optimized source code.

Step Four. After the new order for the source statements has been determined, Optview goes through the source and updates those statements whose semantics need to be updated due to the reordering.[2] Optview makes use of an existing tool within the SGI compiler to perform the semantic rewrites.

Step Five. Once these source statements have been rewritten, Optview uses the data file written by the back end of the compiler to make partial redundancy elimination and common subexpression elimination transformations explicit in the source code.

Step Six. The next step is to use the statement order obtained in step three and rearrange the optimized source statements to match this order.

Step Seven. This step adds dead code elimination information to the optimized source. Any source line in the original program which contains executable code, and which does not have any associated instructions in the target code is determined to be dead code that was eliminated, and is indicated as such by being commented out.

Step Eight. After dead code elimination, Optview makes a final pass over the optimized source code and annotates statements at which copy propagation, common subexpression elimination, or constant folding occurred. Once the program has been annotated, the optimized source is complete.

Rewriting Multi-functional Constructs

An important aspect of the optimized source is that it accurately reflects the order in which events occur in the binary, particularly with respect to the order in which variables are updated.[3] It is critical that the optimized source be completely accurate with respect to variable updates, as the removal of out-of-order variable assignments (the currency problem) is one of the major benefits of this approach to debugging optimized code.

[2] Sometimes the optimizer reorders assignment statements that are dependent on each other. Such reordering occurs most often when one of the statements increments or decrements a variable by a constant amount. In such cases, the optimizer determines that semantic equivalence can be preserved by adding or subtracting a constant value to one of the reordered statements. As these two statements will also be reordered in the optimized source, the appropriate source statement needs to be updated to add or subtract the appropriate constant as well.

[3] "Events" that can occur in the binary include updating variable values, testing predicate values, taking control flow branches, and calling subroutines.

C, in common with many high-level programming languages, contains multifunctional language constructs that allow multiple variable assignments to be represented within a single statement. The optimizer may choose to separate and scatter the corresponding assignments in the target program. Therefore the optimized source needs the ability to represent each variable assignment separately. Therefore Optview replaces constructs allowing multiple variable assignments with simpler constructs, resulting in at most a single variable assignment per statement in the optimized source.

We identified three source language constructs in C which inherently allow multiple assignments per statement and that need to be split apart or rewritten: `for` statements, conditional expressions, and increment/decrement operators (`++`/`--`) embedded within other statements. We also discovered a need to rewrite initialized declarations, in order to correctly place the initialization assignment within the source.

Recall that the front end of the compiler writes a data file used by Optview to identify the source locations of the constructs that may need to be rewritten. As the optimized source is generated, those lines are parsed to find and update the text that needs to be rewritten or moved. We take advantage of the fact that the input *must* be syntactically correct at this stage. This fact allows us to use a relatively simple parsing strategy. Optview rewrites those constructs that need rewriting, using simpler C constructs, as shown in Figure 4.3. `For` statements are rewritten as `while` loops, with the loop initializations before the loop, and the loop increments inserted at the end of the loop body. Assignment statements that have conditional expressions on the right hand side are rewritten as `if-then-else` statements. Increment and decrement operators that are embedded within other statements are pulled out of the statements in question and written as separate statements, either before or after the containing statements, as appropriate. Finally, initialization assignments are removed from variable declarations and inserted just prior to the next executable statement. These changes permit much greater flexibility for rearranging and modifying these assignments as needed to reflect the optimizations.

In writing Optview we made the simplifying assumption that the original source program has at most one statement per source line. Source programs that do not meet this requirement can be run through a simple pretty-printing processor for Optview. This assumption was made initially to allow the use of a relatively simple parsing scheme within Optview. Optview does not contain a full-fledged parser or lexical analyzer. At first it did not seem necessary to incorporate one into Optview. In retrospect we realized that

```
Original Source Code                      Modified Source Code

(a) for statements

                                           i = 0;
 for (i = 0; i < max; i++) {               while (i < max) {
   ...                                       ...
 }                                           i++;
                                           }


(b) conditional expressions

                                           if (i > j)
 max = (i > j) ? i : j;                      max = i;
                                           else
                                             max = j;

(c) increment/decrement operators

                                           --j;
 A[i++] = B[--j];                          A[i] = B[j];
                                           i++;

(d) initialized declarations

  int found = 0;                           int found;
  int result = 1;                          int result;

  ...                                      found = 0;
                                           result = 1;
                                           ...
```

Figure 4.3: Rewriting Multi-Functional C Constructs

Optview probably should have had one, but by that point adding one would have meant making massive changes to the system. Based on our experiences we recommend that any other tools attempting to generate optimized source start with a correct, complete parser and lexical analyzer for the source language.

The one language construct that Optview probably should rewrite and which it currently does not is the statement of the form “a = b = c”. This type of statement was accidentally overlooked in our initial implementation. Making Optview find and rewrite statements of this type should be relatively simple and is on the current list of fixes and enhancements planned for Optview.

Rewriting Reordered Code

During the optimization process the compiler changes the order in which source statements may execute pretty freely. Changing the order of source statements is not a problem if the statements that are reordered are independent of one another. However sometimes the compiler chooses to reorder dependent statements. When it does so the compiler must also update the calculations performed by these statements in order to preserve the original semantics of the program. Recall the example of this problem shown in Figure 3.3, reproduced here as Figure 4.4.

Figure 4.4a shows a small fragment of code. The second statement in this fragment assigns a value to *i*. This value depends on the variable *num*, which is updated in the third statement. During the optimization process, the compiler chose to reorder the second and third statements. Now *num* gets incremented *before* it is used in the assignment to *i*. As indicated in the comments in Figure 4.4b, reordering the statements without making any other changes results in the variable *i* receiving a different value than in the original program. To correct for this difference, the right-hand side of the assignment to *i* needs to be updated to take into account the new statement order. In this case, the constant value needs to be changed from seventeen to twenty, as shown in Figure 4.4c.

The compiler of course performs the appropriate transformations internally when it rearranges the statements. However Optview *also* needs to reflect these changes in the optimized source. Merely rearranging the source statements will cause the resulting program to appear to be calculating the wrong values, and will moreover completely mislead the programmer, as it will *not* accurately reflect the values being calculated by the executing

```
(a) Original Code

  num = y;               /* num == y          */
  i = 3 * num - 17;      /* i   == 3 * y - 17 */
  num++;                 /* num == y + 1      */

(b) After Reordering (Wrong)

  num = y;               /* num == y                */
  num++;                 /* num == y + 1            */
  i = 3 * num - 17;      /* i   == 3 * (y + 1) - 17 */
                         /*     == 3 * y - 14       */

(c) Reordered & Updated (Correct)

  num = y;               /* num == y          */
  num++;                 /* num == y + 1      */
  i = 3 * num - 20;      /* i   == 3 * y - 17 */
```

Figure 4.4: Reordering & Updating Code

target program.

In order to modify the source program appropriately after reordering the statements, Optview uses an existing tool within the SGI compiler. This tool takes fragments of internal representation and translates them into C code. Therefore for each statement that will need to have its right-hand side rewritten in the manner described above, Optview finds the piece of optimized internal representation generated by the compiler that corresponds to the right-hand side of the statement. Optview then passes this piece of internal representation through the compiler tool, which returns the corresponding fragment of C code. Optview then inserts the new C code into the right hand side of the statement in the optimized source. Since the intermediate representation from which this fragment was generated has already been optimized by the compiler, which correctly transformed the program when it rearranged the statements, the result is correctly updated source.

One of the side effects of rewriting these statements in this manner is that the results of constant folding are made apparent in the optimized source without any special effort on the part of Optview to identify where the compiler performed this optimization.

Original Source Code	Modified Source Code
	`cse_var_1 = 2 * y;`
	`cse_var_2 = cse_var_1 + 3;`
`c = 2 * y + 3;`	`c = cse_var_2;`
`a = 5 + 2 * y;`	`a = 5 + cse_var_1;`
`b = (4 + 2 * y - 1) / z;`	`b = cse_var_2 / z;`

Figure 4.5: CSE in Optview

Collecting Optimization Information

To accurately reflect the effects of optimizations such as common subexpression elimination (CSE), partial redundancy elimination (PRE), and copy propagation, Optview needs to know precisely what the compiler did and precisely where it did it. We modified the compiler to collect information about these optimizations as it is performing them and to write this information to a file which Optview later uses to reflect these optimizations in the modified source code. The information Optview requires includes the location where assignment statements are inserted; the left- and right-hand sides of these assignment statements; the location of source statements requiring substitutions; which assignment statement is relevant for each substitution; and which expressions were propagated to which statements. Optview must also determine whether the expressions involved in these optimizations came from the original source program (as opposed to being introduced during the compilation process), since only optimized source expressions are shown.

Optview reads the information file generated by the optimizer to obtain all the necessary information about the optimizations performed. It then modifies the source code to make the effects of CSE and PRE explicit, as shown in Figures 4.5 and 4.6. First it inserts a new assignment statement into the source, assigning the “common” expression to a new temporary variable. Next it replaces the relevant expression or subexpression in the appropriate statements with the new temporary variable. Although these new variables are not part of the original program, our expectation is that, if properly annotated and documented, they will not be too confusing to the user.

In addition to collecting information about where common subexpression elimination and partial redundancy elimination occur in the compiler, we modified the compiler to assign new unique "source positions" to the pieces of internal representation that calculate the subexpressions and assign them to compiler temporary variables. These intermediate representations correspond to the new assignment statements that will be inserted into the optimized source. The new unique "source positions" given to these pieces in the global optimizer are then propagated through the rest of the compiler. Thus in the final internal representation of the program, the instructions that correspond to these new statements are clearly indicated, which allows us to know exactly where to insert the new assignment statements in the optimized source.

One difficulty we encountered when implementing this part of Optview is that the common subexpressions generated by the compiler are in a canonicalized form, whereas the subexpression in the source statement where the substitution is to be performed may not be. Since we use syntactic comparisons to perform the substitutions, it is hard sometimes to find the subexpression to be replaced. The solution we used is to pass the intermediate representation for the original source statement through a tool that outputs the text in the same canonical form as that used by the compiler.[4] Once both the source text and the compiler optimization data are in canonical form, the substitution becomes relatively simple.

Another small problem we have found is that the compiler is very free in its use of the intermediate compiler temporary variables, and sometimes assigns values to them where it is unnecessary. For example the compiler sometimes performs assignments of the form

```
cse_var_3 = foo;
```

where *foo* is a simple variable. The compiler probably performs this type of assignment on the assumption that doing so will enable later potential optimizations. However sometimes nothing further is done with the compiler temporary, other than to replace all occurrences of *foo* with it. Optview currently treats such instances like any other occurrences of common subexpression elimination, inserting the assignment into the optimized source and performing all the corresponding substitutions. In the future we hope to make Optview check for these non-helpful types of substitution, and to eliminate them from the optimized source.

[4] This tool is the one mentioned earlier that translates pieces of internal representation to C.

```
Original Source Code            Modified Source Code

if (...) {                      if (...) {
 a = x;                           a = x;
 y = a + b;                       cse_var_1 = a + b;
                                  y = cse_var_1;
} else {                        } else {
  a = y;                          a = y;
                                  cse_var_1 = a + b;
}                               }
z = a + b;                      z = cse_var_1;
```

Figure 4.6: PRE in Optview

Reordering the Optimized Source

A critical aspect of the optimized source is that it accurately reflects the order in which events occur in the binary. Thus the order of statements in the optimized source is driven by the final internal representation of the program (which has a one-to-one correspondence with the instructions generated for the target program). Optview determines the order of "statements" in this final program representation in several stages. Initially Optview identifies all the key instructions that occur in the program representation, and consolidates basic blocks into superblocks wherever possible.

Optview constructs its own representation of the program by traversing the compiler's internal representation in execution order. It begins by constructing "path segments" which correspond to superblocks, i.e. sequences of basic blocks that have only a single predecessor or a single successor. Back-edges (looping edges) are not added to the representation, although Optview does annotate the path segments corresponding to the top and bottom of each loop. Enough data is maintained to accurately represent the control flow of the program, even if the control flow is not properly nested.

While traversing the compiler's representation of the program, Optview takes note of the source positions attached to each instruction. Optview makes a list of all such source locations found in the internal representation. These are the statements that will be "live" in the optimized source, and for which Optview must identify the key instructions, in order to correctly reorder the source statements. Initially Optview records the source positions in a list, along with whether or not a key instruction for the corresponding statement has been found. Assignment statements already have their key instructions marked by this point, as the tag for key instructions for assignment statements has been passed through the compiler from earlier stages. Source positions in the list for which key instructions have not been found will be marked with information about the path segments in which the corresponding instructions occurred. The first branching instruction (if any) for a source statement is marked as its key instruction as soon as Optview encounters it. Similarly instructions that jump to other functions (function calls) are tagged as key as soon as they are encountered.

Once Optview has finished constructing the path segments, and has recorded all the source statements for which there are instructions, Optview goes back through the list of statements to see if any of them still do not have key instructions. If there are any such statements, Optview goes to each superblock that contains instructions for the statement and marks the last instruction for that statement in the superblock as the key instruction for the statement.

Now that the key instructions for all the source statements have been marked, Optview records, for each path segment, which source statements have key instructions in the segment, and what the order of the source statements (key instructions) is for each path segment.

Next Optview constructs a DAG (Directed Acyclic Graph) that corresponds to the internal representation of the program. The nodes of the graph are the path segments that Optview previously constructed. The edges of the dag are the forward branching edges of the internal representation. Back-edges (looping edges) are not represented as edges in Optview's graph, as the graph is supposed to be acyclic. Instead looping information is recorded as follows. Given a back edge E from node n to node m, node n is annotated with information indicating it is the bottom of a loop whose top is node m. Node m is annotated as the top of a loop. A counter in node m is also incremented to keep track of the number of loops for which it is the top (a single node may be the top of multiple loops).

At this stage the basic DAG Optview has constructed accurately mirrors the control flow of the optimized target program. Each node in the DAG contains a list, in order, of the source statements whose key instructions occur in the corresponding portion of the target program. Therefore Optview could use this DAG directly to determine the order in which statements should occur in the optimized source. However certain optimizations such as loop unrolling duplicate source statements or sequences of source statements. Using the DAG at this point would cause those duplications to all be reflected in the optimized source, making it more difficult for users to recognize and understand. In some cases where the duplicated ordering of key instructions is identical, it may be possible to hide the duplication from the user by making all the duplicate sequences of key instructions in the target program correspond to a single sequence of statements in the optimized source. Thus it is desirable to undo some of these duplications in the DAG, wherever possible, before generating the optimized source. For these purposes Optview performs a series of optimizations on the DAG to reduce and consolidate it as much as possible. There are five optimizations that Optview performs on the DAG. The simplest optimization is to go through the DAG and remove any nodes that have no source statements associated with them (empty nodes). Optview actually does this optimization at three different times, as some of the other optimizations cause new empty nodes to appear in the graph. The next simplest optimization is to find nodes that contain only a single statement, and all of whose predecessor nodes end with the same statement. In that case the single-statement node is merged with each of its predecessors. The other three optimizations are slightly more complicated.

The first of these more complicated optimizations looks for two nodes, one a direct successor of the other, that contain exactly the same statements in exactly the same order. If these two nodes also have exactly the same predecessors (other than themselves), they are merged into a single node. This particular optimization was introduced to "undo" loop unrolling, which usually leaves two versions of the loop, one that represents the loop unrolled some number of times, and the other to "clean up" any left over iterations. As these two versions of the loop usually execute exactly the same statements in exactly the same order, they can be represented as a single loop in the optimized source, thus eliminating one possible source of user confusion.

The last two DAG optimizations involve merging loop nodes where a single back-edge (looping edge) has been split into two separate edges. It is necessary to find and merge such split back-edges in order to avoid introducing new `goto` statements into the optimized source, which would make it much harder to understand. Examples of these two types of optimizations are shown in Figure 4.7 and 4.8. The graph in Figure 4.7a shows two nodes, b and c, on different paths of a conditional both looping back to the same position. Since the branch statement in each of these nodes is by definition always the last statement in the node, this situation is easily remedied by inserting a new node, e, as a successor to both paths of the conditional (and a predecessor to all their original successors). The loop statement is moved out of b and c into the new node, e, thus consolidating the looping statement into a single node.

Figure 4.8 shows a slight variation on the same theme. There are two nodes on the same path, one a successor of the other, both looping back to the same position. From either node the only place to go is either to the top of the loop or to the other node. Again the solution is to introduce a new node into the graph, to which both looping nodes point, and to move the looping statement out of each of the old nodes and into the new node.

As mentioned previously, the purpose of performing all of the optimizations and graph reductions is to reduce as much as possible any redundancy of source statements in the graph, and to reduce or eliminate the need for introducing `goto` statements into the source. Ideally each original source statement will occur exactly once in the final graph, as this situation will have the most direct, easiest to understand mapping back to the original source. In those cases where a source statement does occur multiple times in the final Optview DAG, the source statement is duplicated in the optimized source.

After the graph is complete Optview traverses the graph and finds the post-dominator node (if one exists) for every node that has multiple successors. At this point Optview has a completely accurate picture of which source statements should occur in which order. The next step is to express this order in a linear manner (rather than as a graph), since C programs are written linearly. Optview traverses its DAG representation of the optimized program and constructs a linear final ordering of the source statements for the optimized source. It then uses this linear ordering to rearrange statements from the original program in order to obtain the optimized source.

Copy Propagation and Dead Code Elimination

Optview handles copy propagation (CPP) differently from the other optimizations. Copy propagation is an optimization that enables CSE and other optimizations[41]. Thus the compiler applies copy propagation before applying many other optimizations. However the effects of copy propagation are often transitory, as a later optimization that was enabled by copy propagation may completely overwrite or eliminate the propagated expressions. Our solution to this problem has two parts. First Optview keeps track (via the data gathered

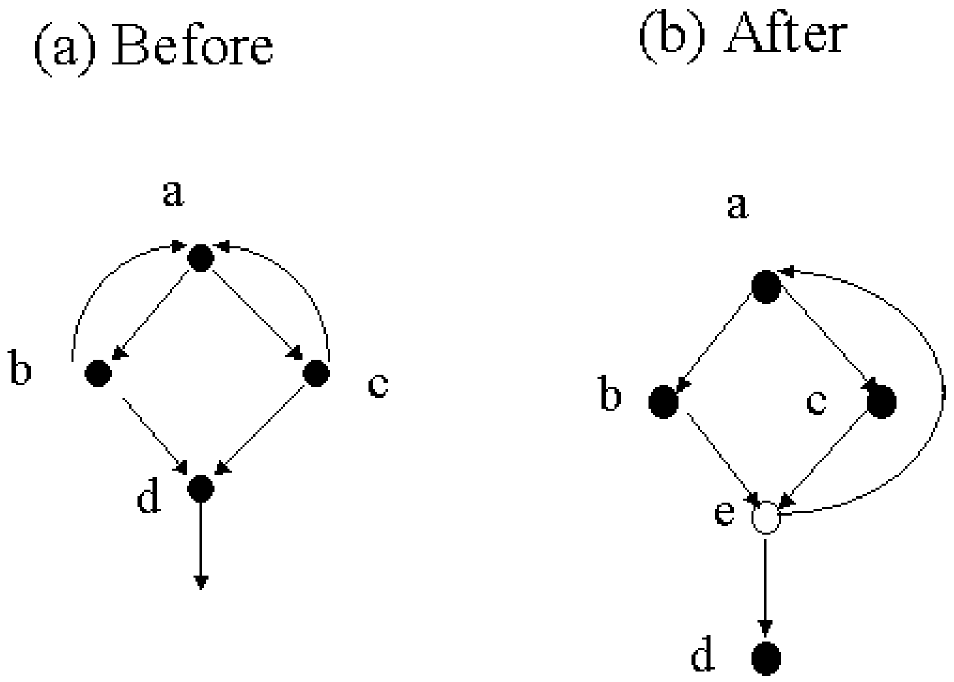


Figure 4.7: Merging split loop nodes

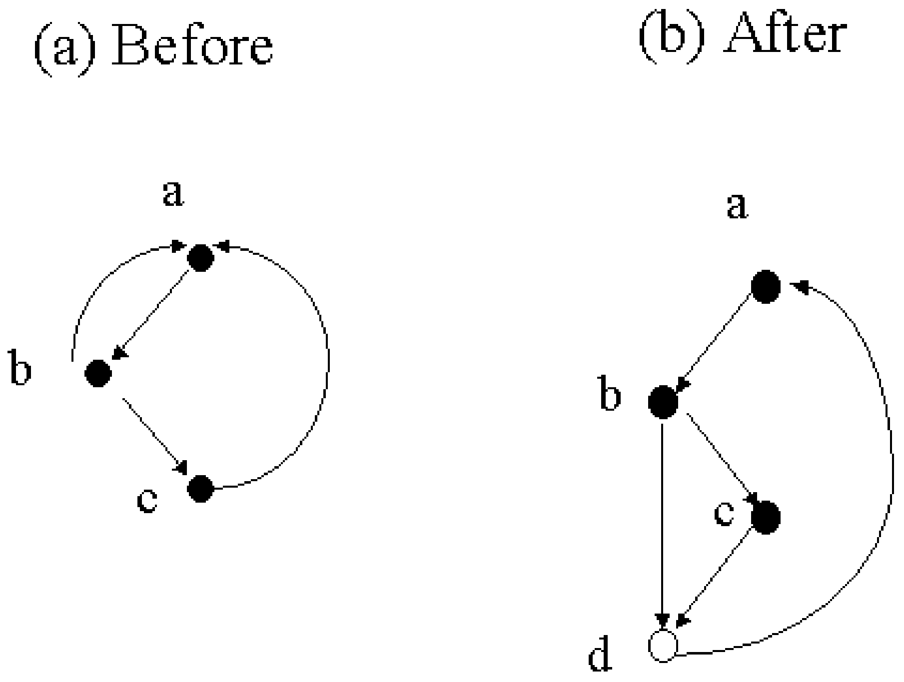


Figure 4.8: More merging split loop nodes

by the compiler) of every source statement to which an expression was propagated, as well as the exact propagated expression. After all of the other optimization effects have been reflected in the modified source code, Optview goes through and checks each resulting source line to which an expression was propagated. A comment is added to the end of each line stating that copy propagation occurred there. If the original variable still exists in the source statement, Optview replaces it with the propagated expression.

The final stage that Optview goes through is finding and marking dead code in the optimized source. Optview starts with a list of statements that the global optimizer thinks it may have eliminated. Optview removes from this list any statement for which instructions were generated in the final representation of the program, since any statement for which there are instructions cannot be dead. One might expect that such a step would be redundant, as it seems logical that if the optimizer eliminated a statement, there could not be any instructions generated from that statement. The key to understanding this apparent contradiction lies in the following two facts. 1). Every time the global optimizer determines that a particular piece of the internal representation is dead, it eliminates that piece of code, and records the source position for the statement corresponding to the internal representation that is eliminated. This generates the list from which Optview starts. 2). During the course of the optimization process, a source statement or pieces of the statement, may be duplicated, perhaps several times. It is possible that some of the duplicated pieces are later removed during dead code elimination, but others are not. Because of these two circumstances it is possible for a source statement both to be on the list of statements the global optimizer eliminated and to generate instructions that appear in the final internal representation of the program.

Once Optview has removed from the "dead list" any statement for which instructions were generated in the final representation of the program, it goes through the remaining list. For each statement on the list Optview finds the corresponding statement in the optimized source, surrounds the statement with comments, and adds a note indicating that the statement is dead. We decided to take this approach rather than actually removing the statements from the optimized source, as we thought our approach would lead to less confusion. If the dead statements were actually removed from the source the user may be left wondering what happened to the code, or think that it was moved to some other part of the program. By commenting out the code it becomes obvious that the code will not execute, and that it has not been moved or accidentally left out.

4.3 Other Compiler Modifications

In addition to modifying the compiler to collect data for Optview and to invoke Optview at the appropriate time, we needed to make a few other modifications to the compiler. In particular we modified it to create and propagate key instruction tags for assignment statements; to collect data about the locations of variables during program execution; and to output this information to the symbol table.

4.3.1 Key Instructions

As described in the last section, Optview uses key instructions to determine the correct order for statements in the optimized source. For most statement types, the key instruction for the statement can be identified by knowing the statement type and examining the instructions generated for that statement. This property is not true for assignment statements. Instructions make no mention of variable names, and the "store" instructions may be optimized away altogether, leaving operator instructions whose destination register becomes the store location. With no mention of variable names and no knowledge of what the opcode for the store instruction may be, one needs further clues to pick out the key instruction for assignment statements. In our implementation we allow the compiler to help us with this task. We modified the front end of the compiler to tag, for each assignment statement in the source program, the intermediate representation statement that stores the value. As the intermediate representation goes through various optimizations, transformations, and lowering stages, this tag is tracked and updated appropriately, and passed through the compiler. Therefore when the final internal representation of the program is constructed, the key instructions for assignment statements are already tagged as such.

While implementing the stepping and breaking functions in the debugger, we came across an interesting problem related to key instructions in delay slots. In optimized code it sometimes happens that a key instruction for one source statement will be scheduled in the delay slot for another source statement. Such occurrences cause problems when attempting to set a breakpoint on the statement whose key instruction is in the delay slot. In the MIPS processor, delay slot instructions can never be executed separately from their branch instruction, thus execution can never halt on a delay slot instruction. Since key instructions indicate where breakpoints for source statements are supposed to be set, the result in our initial implementation was that breakpoints for statements with delay slot key instructions were never reached. They appeared to be skipped over.

Our solution to this problem is to make the branch instruction itself the key instruction for the delay slot statement. In those situations where the branch instruction is also a key instruction for a different source statement than the delay slot, the instruction is annotated with both optimized source locations. Whenever execution halts on the branch statement, the current source position is indicated as being *both* source statements.

A similar type of problem would exist if one were attempting to use key instructions for setting breakpoints on a VLIW machine: Some number of instructions get executed together atomically, and any subset of those instructions may be key instructions. Thus, using our scheme on a VLIW machine, at any given breakpoint one could simultaneously be at two or more source statements. Of course there are many other problems particular to debugging on VLIW machines. Discussing them all is beyond the scope of this work.

4.3.2 Collecting Range Table Information

As mentioned in Chapter 3, in order to determine where to look for a variable, after the program has been optimized, the debugger needs a list of potential locations, and, for each location, the range of instruction addresses (program counter values) for which it is valid. This information, usually referred to as the variable range table, must come from the compiler.

In previous approaches to debugging optimized code this information was collected by carefully tracking the location of each variable through all of the optimizations. Such an approach is arduous and rather brittle: any time an optimization is added or changed, the code for collecting the range table information needs to be changed too. Thanks to key instructions, we were able to take a simpler, more robust approach.

We collect the data for the variable range table in two passes. The first pass is performed at the end of the global optimization phase of the compilation. In this pass, the internal representation of the program is traversed to find every node that is annotated as "key" (i.e. will eventually become a key instruction). Remember that at this point these key annotations represent all source level assignments to source level variables, and *only* such assignments. Whenever such a "key" node is found, an entry is made in a table recording the name of the variable receiving the new value, and the original source location of the assignment.

The second pass is performed after Optview finishes generating the optimized source. This pass performs a variation of dataflow analysis on the compiler's final control flow graph (instructions) representing the program. The values being traced in this analysis are *location records*, where a *location record* is a tuple containing an address or register number, a variable name, the starting address for the record and the ending address for the record. The starting address and the ending address indicate the range of addresses in the binary for which the location record is valid. The starting address gets filled in when the record is generated. The ending address gets filled in when the record is killed. Killed records are retained, though not passed on during the dataflow analysis. At the end of the dataflow phase, all information in the killed records is used to construct the range table.

The initial set of location records indicates the parameters for the current function, and where these parameters are initially located. As the analysis is performed, each instruction is examined. If it overwrites a location in the current set of live location records, the corresponding record is killed at that point. Every time a source assignment is encountered (annotated by a key instruction), a new location record is generated. If the actual location specified by the new location record also occurs in another record in the current set, the old record is killed at that point. Similarly if a record already existed for the variable receiving the assignment, the old record is killed. Since variable names are not available in the instructions at this point, the source location associated with the instruction is used to look up the variable name in the table created during the first pass. The *out* set for each basic block is the standard definition: the *in* set unioned with the set of generated records, minus the set of killed records. The *in* set for each basic block (except the first one) is the union of the *out* sets for all predecessor blocks.

Once the second pass is finished, the data gathered is combined and written into a variable range table, which will be stored as part of the debug symbol table information in the object code. One very nice characteristic of this approach is that the single dataflow pass not only collects actual location information; it also, as a side effect, collects information about variable eviction and uninitialized variables. Recall that an evicted variable is one that is in the current scope, but which is currently dead, and whose value has been overwritten (i.e. it is not stored anywhere). In the process of doing dataflow analysis to determine the variable range table information, it is determined exactly where variables become evicted. Entries are then made in the variable range table indicating ranges where variables are evicted.

We encountered one interesting problem when performing the dataflow analysis. On reaching a basic block with multiple predecessors, it is possible that a variable which is not live (and not evicted) at the entrance to the basic block may have its value in a different place in each preceding basic block. Thus the correct location for the variable at this point will depend on which execution path was taken. There are several possible solutions for this problem, including putting all locations into the table and having the debugger insert *path determiners* ([46]) in the binary during the debug session to indicate how execution reached the current position; putting all locations into the table and returning values from all locations to the user, if the user requests the value at that point in the execution; or, ending all ranges for the variable at the end of the preceding basic blocks and making the value unavailable. The last option in effect turns the variable into an *evicted* variable at that point in the program.

We chose the last solution. We mark the variable as evicted at the end of each of the preceding basic blocks, and use our eviction recovery scheme in the debugger to produce the value of the variable in question at the user's request.

4.3.3 Modifying the Symbol Table

The SGI compiler and debugger we modified use the DWARF Standard Debugging Information Format ([32]) for representing and storing the symbol table information. This standard, while not designed particularly for debugging optimized code, contains all the pieces we needed, so we were able to add all the extra information necessary to the symbol tables without changing the format used or violating the standard. There were four changes we had to make to the symbol table, besides making it use the optimized source file rather than the original one.

The first change was to associate with each variable a list of locations that are valid for that variable, as well as the range of program counter values that are valid for each location. DWARF allows either a single location or a location list to be associated with variables. The original version of the compiler only stored single locations with each variable. We took the information from our variable range table, built the appropriate location list for each variable, and stored it in the symbol table.

In addition to indicating valid locations for looking up variable values, we also wanted the symbol table to indicate those portions of the program for which a particular variable was uninitialized or evicted. At the same time we were trying to work within the DWARF standard, which has no way of specifying non-location values in the location lists. We resolved this problem by using two special absolute addresses that would never be valid variable addresses in the computer to represent these two special conditions. We chose the absolute address 0x01 to represent an evicted variable, and the address 0x04 to represent an uninitialized variable. We then modified the debugger to interpret these two absolute addresses accordingly.

The second change we made to the symbol table was in the line table. The line table is the portion of the symbol table that indicates the mapping between source statement positions and binary instructions. Rather than record the source statement position for each binary instruction in the target program, the compiler only records the positions for instructions whose source position is different from that of the preceding instruction (in a linear traversal of the target program). This approach allows the compiler to skip recording a lot of duplicate information, thus saving space in the line table. The compiler also assumed that, since it was only making entries when the source position changed, every entry in the line table represented the start of a new source line and should be treated as a statement boundary. This assumption is reasonable to make for debugging unoptimized code, as in that case the instructions for each source statement occur in contiguous blocks. However once the program has been optimized this assumption is no longer correct. We modified the compiler to only mark entries corresponding to key instructions as statement boundaries in the line table.

The trickiest change we had to make to the symbol table was to add entries for the "compiler temporary variables" that we introduced in the optimized source. If we show those variables to the user and make them an integral part of the optimized source, then we must allow the user to query their values, just like any other variables in the program. Therefore they must have entries in the symbol table. When collecting the variable range table information, via the dataflow analysis, we collect information about these variables as well. Thus our variable range table contains the necessary information for these variables. We use the range table to create the entries for these variables, after Optview has been run and before the target program is generated. (The symbol table entries for "normal" variables are created by the front end of the compiler, much earlier in the compilation process.) One problem we encountered when creating these new entries was assigning them a correct data type. The debugger refuses to attempt to display a variable's value unless it knows what type of data the variable is supposed to represent. However these compiler temporaries could hold any type of data. We finally settled for declaring them to be long integers. Doing so at least allows the debugger to show the value to the user. Since the language for which the compiler and debugger were implemented is C, the user has the option of coercing the debugger into displaying the variable as a different type, via casting, if desired. If this implementation were for a more type-safe language, getting the correct type for these compiler temporaries would be a more serious problem.

The last change we had to make to the symbol table was to add a new "vendor-specific" section, as allowed by the DWARF standard. This new section is the eviction table section. The eviction table allows the debugger to quickly identify, for each function, every instruction within the function that evicts a variable. The eviction table contains one entry for each eviction that occurs in the target program. Each entry indicates the program counter address of the evicting instruction, the name of the variable that the instruction evicts, and the location from which the variable is evicted. The debugger then makes use of this information to perform its eviction recovery.

4.4 Optdbx

Optdbx is the debugger we implemented for debugging optimized code. It has two main components: a modified version of the SGI Dbx command line debugger, which handles all the real debugging functionality; and a graphical user interface, which is responsible for presenting the optimized source to the user in a relatively non-confusing manner and for helping the user to find the correspondences between the original and optimized source programs.

4.4.1 Eviction Recovery

By far the largest change we had to make to the SGI Dbx debugger was implementing eviction recovery within in. Recall that an evicted variable is one that is in the current scope, but which is currently dead (its value is not needed any more), and whose current value has been overwritten (i.e. it is not stored anywhere). In the process of doing dataflow analysis to determine the variable range table, Optview determines exactly where variables become evicted, and this information is passed to the debugger via the symbol table. The debugger then recovers the values of evicted variables as follows. Whenever the user sets a location breakpoint within a function (at function entry, at a particular line, etc.), if it is the first breakpoint set in that function, the debugger will set hidden traps immediately prior to all instructions that evict variables in that function. On reaching such a trap, the debugger first checks the function information to determine what variable is being evicted and where the value for the variable is. The debugger then saves the value of the variable in its stack frame information[5] and resumes execution. If the user requests to see a variable which has been evicted, the range table will indicate the variable is evicted (i.e. the variable will be indicated as having an absolute address of 0x01). The debugger will then look for the variable's value in the debugger's frame information. If the debugger has saved the value for the variable, it returns the value to the user. If the debugger does not have the value, e.g. perhaps execution halted for some reason other than a user-specified control breakpoint, then the debugger reports that the variable's value is currently unavailable. When the last user-specified control breakpoint is removed from a function, the debugger also removes all evicted variable traps from the function.

Although the number of such traps in a given function may be significant, the expectation is that the increase in execution time created by this caching of evicted values is overshadowed by the time spent in the interactive breakpoint(s). By limiting this behavior to only those functions where a breakpoint has already been set, most execution of the program should be unaffected.

The advantages of our recovery scheme are that whenever a user-specified breakpoint is reached, the values of all local variables are guaranteed to be available to the user, as well as all global variables (which are always live, unless the compiler has an unusually sophisticated analysis scheme), and the values of any variables that are arguments to functions up or down the call stack (which must also be live). All of this is gained without affecting the execution of the program other than in those functions where the user has already indicated a willingness to change the execution.

The disadvantage of this approach is that it is only a partial recovery scheme for the eviction problem. It is not a complete solution because the debugger can suspend execution for reasons other than a user-specified breakpoint, e.g. an execution error, a user interrupt, or a watchpoint. In those cases the debugger will not have saved the values of evicted variables, unless such a suspension happens to occur in a function where a breakpoint was

[5] This information is the *debugger's* information about the executing frame. The debugger does not touch the actual stack frames of the executing target program.

set. Also, if the user changes context by moving up or down the call stack, variables that are not arguments to functions in the call stack may or may not be available.

4.4.2 Other Changes to Dbx

Other than eviction recovery we had to make very few changes to Dbx in order for it to work with the optimized source. There were a few minor changes however. The original Dbx was written with the assumption that a variable would always come with a single location, rather than a location list (in spite of the fact that the symbol table format, DWARF, allows for variables to have location lists). Therefore we had to modify Dbx slightly to look up a variable's value in the location list, if a list is provided. Two tiny but important associated changes were: If the variable is shown as having an absolute address of 0x04, the debugger reports to the user that the variable is currently uninitialized; and, if the variable has *no* location entry at all the debugger reports to the user that the variable was optimized away by the compiler. These two changes required almost no work in the debugger but they convey *very* important, useful information to the user about the program.

The other change we had to make involved slightly more work, but still not a lot. Since key instructions are being used to indicate where source statements occur in the target program, we modified the debugger so that when the user sets a breakpoint on a source statment, or chooses to single-step, next-step, or return from a function, the instruction at which execution is actually suspended is always a key instruction.

4.4.3 The Graphical User Interface

As mentioned previously, using optimized source to allow the user to see some of the effects of optimization greatly simplifies, or even eliminates, many of the classical problems in debugging optimized code. However it does introduce a new problem: The user is now confronted with a program she did not write, and which she must understand reasonably well before any significant debugging activity can take place. By judiciously selecting the optimizations whose effects are to be shown to the user, and carefully planning how to present those effects, the impact of this problem can be reduced somewhat. However the greatest, most important tool for minimizing this burden of understanding on the user is the graphical user interface for the debugger.

The single most important aspect of understanding the optimized source program, for the user, is determining how it maps and relates to the original source program, as the error will eventually have to be found and fixed in the original source, not the optimized source. In order to facilitate this understanding we have created a graphical user interface that displays the original source program and the optimized source program side by side. By default the current execution position is highlighted in both sources. However if the user wishes to figure out where a particular piece of code in the optimized source came from, she can click on that code, and the corresponding code will be highlighted in the original source. Similarly if the user is trying to figure out what happened to a piece of code from the original program, she can click on the original code, and the corresponding piece(s) of code in the optimized source will be highlighted. Figure 4.9 is a screen dump showing what the interface looks like.

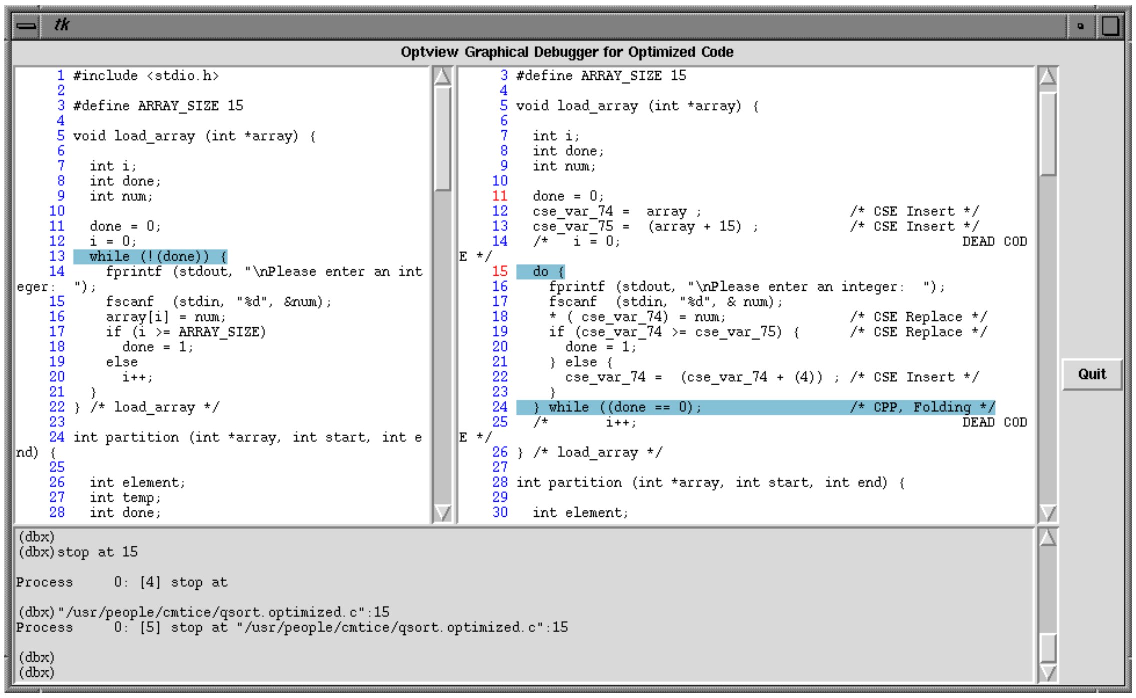


Figure 4.9: Screen dump of Optdbx

The scrolling text field on the left-hand side shows the original source program. The optimized source is in the scrolling text field on the right. At the bottom is a third scrolling text field, for entering standard dbx commands.

This graphical user interface was written using the Python programming language [27], which is layered on top of Tcl/Tk [40]. On startup the graphical user interface creates two Unix pipes, then forks a process running our modified Dbx, with a pseudoterminal between the pipes and Dbx. The user types standard Dbx commands in the scrolling text field at the bottom of the screen, and the interface pushes these commands down the pipe to Dbx, which executes the commands. Responses from Dbx return up the pipe and are displayed in the Dbx command field. In addition to sending user commands to Dbx, the graphical user interface also occasionally sends "hidden" messages to Dbx asking about the current location in the source file, to make sure the source displays are updated appropriately. This simple interface works well for the optimizations we have chosen to display in Optview. For other optimizations, which may more seriously alter the appearance of the source program, such as function call inlining or high-level loop optimizations, other approaches are possible.

Chapter 5

Measurements and Results

In the last two chapters we described a theoretical solution to the problem of debugging optimized code, and an implementation of that solution. In this chapter we will examine the practicality and effectiveness of this approach. There are three main questions which this chapter attempts to answer: How does this approach affect the efficiency and correctness of the compiler? How does it impact the efficiency and correctness of the debugger? How difficult is it for people to use and understand the optimized source code for debugging?

5.1 Compiler Data

5.1.1 Reliability

A critical requirement of any compiler is that the machine code it produces be a correct translation of the original program. Therefore we will start by addressing the correctness of the code produced by our modified compiler. The compiler from which we started, the SGI Mips Pro 7.2 C compiler, is a commercial compiler, and therefore produces correct code (most of the time). Thus if we can show that none of our modifications altered or affected in any way the machine instructions generated by the compiler, then the machine code output by our version of the compiler must also be correct.

Figure 5.1 shows a complete list of the modifications we made to the compiler, with similar types of modifications grouped together.

The first group of modifications are those where we added fields or structures

- Added structures to representation.
  - Added structures for tracking key instruction tags.
  - Added structures to track source position information at the expression level.
- Propagated key instruction tags throughout compiler.
- Collected information during compilation.
  - Collected important pieces of parsing information and wrote them to a file.
  - Collected information about performance of certain optimizations and wrote it to a file.
  - Collected names of source level variables receiving assignments, and the source positions of the assignment statements.
  - Collected eviction table and range table information (via dataflow analysis).
- Updated source position information.
  - Propagated source position throughout the compiler at the expression level.
  - Corrected propagation of source positions throughout optimizations.
  - Modified the source positions in the final internal representation to refer to the optimized source.
- Added call to Optview prior to emitting machine code.
- Added or updated debug information.
  - Modified debug information to point to optimized source rather than original.
  - Added range table and eviction table information to debug information.
  - Added compiler temporary variables that appear in the optimized source to debug information
  - Modified the source positions in the debug information to refer to the optimized source
- Wrote out mapping between original source and optimized source.

Figure 5.1: Modifications made to the SGI Mips Pro 7.2 C Compiler

to the existing data representations within the compiler. By adding to the existing data representation we have not changed the way in which the compiler stores or uses any of its data. Therefore these changes have not affected the correctness of the code generated by the compiler.

Propagating the key instruction tags through the compiler entailed identifying every location where a piece of internal representation is created, copied, or updated, checking the tags on the code before the create, copy or update operation is performed, and then modifying the tag fields appropriately after the operation. At no point does this alter or interfere with operations the compiler was performing or the data on which the operations were based. Thus propagating key instruction tags does not affect the correctness of the code generated by the compiler.

To collect the parse data and the optimization information data, we added code to the compiler at the places where it parses the structures in which we are interested, or where it performs the optimizations to be reflected in the optimized source. The code that we added to the compiler records the type of operation being performed by the compiler, the original source position of the code fragments on which the operation is performed, and other information, such as the common subexpression, which we may need to recreate the effect of the optimizations in the optimized source. This information is appended to data files that are later used by Optview. None of the code that collects this information interferes with or alters in any way the processing performed by the compiler.

The names and source positions of variables receiving assignments are collected by performing a traversal of the internal representation of the program, and writing the information out to a file. Similarly the eviction data and range table information are collected by performing a dataflow analysis on the final internal representation of the program. As none of this data collection work interferes with or alters either the operations the compiler is performing or the data on which the operations are based, it does not affect the correctness of the code generated by the compiler.

Since the original compiler did not allow for recording source position information at the expression level, we had to add data structures to contain this additional information (as mentioned above). Thus propagating the source position information at the expression level consisted of updating the values in these new, additional data structures. Doing so in no way affected either the operations performed by the compiler or the data on which those operations are based. On the other hand correcting the propagation of source positions in the existing structures throughout the optimizations, and updating the source positions in the final internal representation to refer to the optimized source both involved changing data in existing structures. This has the potential of changing the behavior of the compiler. However the compiler itself never examines the data in these fields, and therefore never bases any of its operations on the values in these fields. The sole purpose for these fields existing at all in the compiler, is to allow the source positions to be written into the debug information, if debug information is being generated. The contents of these source position fields have no influence at all on the machine instructions which the compiler generates. Therefore our changing and correcting the values in these fields does not affect the correctness of the code generated by the compiler.

Adding the call to Optview could only change the correctness of the code generated by the compiler if either the data on which the compiler is working, or the control flow path through the rest of the compiler were altered as a result of calling Optview. Optview generates the optimized source file, and in the process it reads the data files collected earlier by the compiler, as well as traversing the final internal representation of the program. However it does not alter the representation or any of the compiler's data, nor does it have any side effects that would cause the compiler to change its execution control flow on returning from Optview. Therefore the call to Optview does not affect the correctness of the code generated by the compiler.

The debug information generated by the compiler is stored in data structures that are completely separate from the code being generated. The compiler writes information into these structures, but never refers to or uses the data in any way to generate the machine code. As with the source positions, the only use the compiler has for the debug information is to write it out to the appropriate sections of the object file (completely distinct from where the machine instructions are written) to be used later by the debugger. Because the compiler does not read or use this data in any way, our modifying it and adding new pieces to it does not change either the operations performed by the compiler (to generate the machine code), nor the data on which those operations are performed or based. Thus it does not affect the correctness of the code generated by the compiler.

The information about mapping between the original and the optimized source files is contained within data structures created by Optview. These structures did not exist within the compiler previously, and are not used by the compiler at all. Therefore maintaining the data in these structures and writing it out to a file does not alter the operations performed by the compiler or the data on which those operations are based, and therefore does not affect the correctness of the code generated by the compiler.

As we have just shown in detail, none of the modifications we made to the SGI compiler have any effect on the correctness of the code being generated. Thus our modified compiler is at least as correct and reliable as the SGI Mips Pro 7.2 C compiler.

5.1.2 Efficiency

Having established the reliability of our compiler, the next most common concern is how efficient it is. There are several different ways in which efficiency can be measured, including the speed of the compiler, the speed of the compiled code, and the size of the file(s) generated. As was pointed out above in detail, the compiled code is the same as that generated by the standard SGI Mips Pro 7.2 C compiler. Thus the speed of the compiled code has already been extensively measured and documented (see Appendix B). Therefore we will concentrate here on the speed of the compiler itself and the size of the resulting files.

Compilation Time

We measured the compilation time of our version of the compiler and compared it against the compilation time of the original unmodified compiler. The programs on which we measured the compiler included the eight C programs from the SPEC95 benchmark suite (`go, m88ksim, gcc, compress, li, ijpeg, perl, vortex`). As we were concerned about the possibility that the SPEC benchmark programs might have characteristics that are not really representative of non-benchmark C programs, we also measured five other C programs, `test suite, philspel, eval-emon, measure1, and sm`. These programs were written by graduate students at the University of California, Berkeley. `Test suite` consists of twenty small programs, ranging in size from 18 lines to 400 lines, which were written to test the compiler as we were modifying it. `Eval-emon` is a 5,000 line program that takes as input files containing data from hardware counters, and it performs a series of calculations on this data, determining things such as cache miss rates. `Philspel` is a small program that takes a dictionary and a document and performs spell-checking on the document. `Sm` is a mid-sized (14,000 lines) graphics program. `Measure1` is a group of simulation scripts for testing a network resource clustering algorithm.

The compilation times were obtained by using the Unix "time" command. For those programs that consist of multiple source files, we timed the compilation of each separate source file, and added the times to get the overall time for each program. Each measurement was taken multiple times to verify its correctness. Before going into the details, we should mention up front that no general conclusions about this approach should be drawn from these numbers. They are specific to our particular implementation which, being a prototype, was written with an emphasis on correctness rather than efficiency. The results of our measurements are shown in Table 5.1. This table shows three compilation times for each program. The first is the time for the original compiler, performing no optimizations but generating debug information. The second is the time for the original compiler performing optimizations and generating some debug information. The third is the time for the Optview compiler performing optimizations and generating debug information and optimized source. The last column in Table 5.1 shows the percentage increase in compilation time between the original compiler performing optimizations and the Optview compiler. The numbers in this table reflect the total elapsed time, in seconds.

As can be seen, our compiler on average takes 8 times longer to run than the unmodified Mips Pro 7.2 C compiler. The SPEC benchmarks took anywhere from 3.4 times longer to 12.1 times longer, averaging a 7.1 increase in compilation time. The increase for the non-SPEC benchmarks was 3.0 to 26.4, with the average increase being 9.3. From these numbers it seems fair to say that the SPEC benchmarks seem to be fairly representative of other programs, at least for compilation times.

The worst case slowdown we measured, compiling `eval-emon`, was a 26.4 increase in compilation time. As this is more than double the increase measured for any other benchmark we examined the compilation of `eval-emon` more closely. Nearly two-thirds of the compilation time was being spent compiling two functions (the program contains twenty-four functions). Half of that time was being spent recording the optimization information for those two functions in the back end information files, which were unusually large for the two functions in question. On further investigation we discovered that the two functions are both very long (roughly 800 lines of code each). They both contain very large `switch` statements in which many calculations are duplicated, allowing for large amounts of common subexpression elimination. Thus most of the slow down for this particular benchmark can be attributed to reading and writing the back end information file for the two functions mentioned above.

	Original Compiler (no opt)	Original Compiler (opt)	Optview Compiler (opt)	Percent Increase
go	25.6	115.1	1,408.2	1123.5 %
m88ksim	45.1	85.1	402.8	373.3 %
gcc	158.9	645.2	8,472.1	1213.1 %
compress	2.3	4.3	19.6	355.8 %
li	14.6	33.6	149.3	344.3 %
ijpeg	39.4	101.8	521.9	412.7 %
perl	34.0	105.1	1,344.4	1179.2 %
vortex	79.2	251.4	2,024.5	705.3 %
test suite	6.2	10.7	35.9	235.5 %
philspel	0.7	1.2	4.8	300.0 %
eval-emon	4.1	17.3	474.6	2643.4 %
measure1	7.3	17.2	104.3	506.4 %
sm	30.7	88.3	967.2	995.4 %
Avg. (SPEC)	–	–	–	713.4 %
Avg. (non-SPEC)	–	–	–	936.1 %
Avg. (All)	–	–	–	799.1 %

Table 5.1: Compilation times of original and modified compilers (in seconds)

Even discounting the `eval-emon` benchmark, the numbers shown in Table 5.1 represent a significant increase in compilation time. However these numbers should really be regarded as a worst-case measurement of the slowdown to be expected. As we stated previously, our modifications were not written with efficiency in mind. As this was a prototype system, we were much more concerned with the correctness of what we were writing than with making it efficient. As a result of this emphasis on correctness rather than efficiency, there are many parts of our implementation that leave much room for speed improvements. For example while generating the optimized source Optview parses pieces of the original source program. The code that does this is not very efficient and may traverse a segment of code multiple times searching back and forth for particular language constructs. Replacing this inefficient parsing code might significantly improve the compilation time for Optview. Another reason for the large slowdown can be attributed to the fact that we had to work within an existing compiler, adding things to a system that was never originally designed for them. For example we modified both the front end and the back end of the compiler to collect data that we would need later for constructing the optimized source. As the compiler itself did not allow for any method of passing this data around internally, our implementation writes the data to temporary files and later reads it back from those files. By designing a compiler which could keep such information in internal structures, thus avoiding reading and writing these files, the compilation time could be significantly improved. In addition there were certain data structures that we were not allowed to modify directly. Therefore we had to resort to roundabout, inefficient methods in some cases for adding data about key instruction tags and source positions to the compiler. In summary, while the measured speed of our compiler does not compare well against the unmodified compiler, it should not be taken as a very accurate predictor of the impact of our approach on a compiler better designed to support debugging of optimized code.

File Size

In addition to the compilation time, we also measured the size of the optimized source files, and the size of the debug information generated by our compiler. When generating the optimized source, information and statements are added to and inserted in the original source, but nothing is removed. Therefore one should expect the optimized source to be at least as large as the original source, usually larger. The real question is, how

	Original Source	Optimized Source	Percent Increase
go	18,671	42,224	126.15 %
m88ksim	16,826	26,432	57.09 %
gcc	180,019	267,732	48.72 %
compress	1,427	1,942	36.01 %
li	6,731	9,161	36.10 %
ijpeg	24,815	35,820	44.35 %
perl	23,657	38,592	63.13 %
vortex	52,633	74,641	41.81 %
test suite	2,519	2,897	15.01 %
philspel	141	249	76.60 %
eval-emon	5,024	7,705	53.36 %
measure1	2,866	4,852	69.30 %
sm	14,102	21,989	77.20 %
Avg. (All)	–	–	57.30 %

Table 5.2: Comparison of size (number of lines) of optimized source with original source

much larger? Using the same programs we used for measuring the compilation speed, we examined the size of the optimized source programs. Table 5.2 shows a size comparison of the optimized source versus the original source. Again, in those cases where the program listed actually consists of multiple source files, the numbers shown are the sum from the source files involved. They indicate the total number of lines in the source files, including comments, white space, preprocessor directives, etc. To make the comparison more accurate we only counted source files for which the compiler generated optimized source, i.e. only files that contain executable code. Files such as header files that contained only declarations or data definitions were not included in the count. As can be seen, in most cases the optimized code is not much more that 50% larger than the unoptimized code. The one exception is the `go` benchmark. This difference is due mainly to the heavy use of arrays in `go`. The Mips Pro 7.2 C compiler aggressively optimizes array index calculations. Optview currently makes all of these optimizations and calculations explicit in the optimized source, introducing many compiler temporaries into the source in the process. Because arrays are used so frequently in `go`, and because of the nature of arrays in C, the resulting optimized source is quite a bit larger than the original.

Since the debug information gets written into the object files that the compiler generates, in order to determine the increase in the size of the debug information we measured the size of the object files generated by our compiler. Again the size of the object file for each source file was measured and then these measurements were summed to get the total for each program. Table 5.3 compares the size of the object files, in bytes, generated both by the original compiler and by our modified compiler. For the original compiler we measured the object file size both with and without optimizations. Most of the time the optimizations caused the object file to be smaller, although in a few cases they actually made the object file larger. As described earlier in this chapter, the machine instructions generated by both versions of the compiler are identical. Therefore any difference in file sizes between files generated by the original compiler performing optimizations and those generated by the Optview compiler must be due entirely to additions to the debug information. As can be seen, adding all the necessary information for accurate debugging of optimized code significantly increases the size of the object files (on average the overall size increases by 1.7 times).

This increase in size can be attributed to several different causes. To begin with, one must remember that the debug information generated by the original compiler is both inaccurate and incomplete, since the compiler performed optimizations but did not account for that when generating the debug information. Thus the debug information generated by the original compiler contains only a single location for each local variable, whereas mine contains complete range location lists. The original compiler's debug information does not contain any information about compiler temporary variables, whereas mine does. It does not contain any variable eviction information; mine does. Finally its line mapping information, for mapping between locations in the optimized machine code and the source program, is completely inadequate and incorrect. In other words, to summarize, the debug information generated by the original compiler is smaller, but it is also useless for reasonable debugging of optimized code.

On close examination of the sizes of the object files one can see that once again, the size for the `go` benchmark is significantly larger than any of the others. This is not surprising, since the optimized source for `go` has so many more compiler temporary variables in it. The debug information generated for `go` has to have name, scope, and type entries for all these extra variables, in addition to needing the range table location for each of them. This would make the size of the debug information for this benchmark very large as well.

	Original .o size (no opt)	Original .o size (opt)	Optdbx .o size (opt)	Percent Increase
go	909,672	830,792	3,401,336	309.40 %
m88ksim	973,744	974,044	1,911,408	96.23 %
gcc	5,415,464	4,856,320	16,122,120	231.98 %
compress	99,436	99,132	132,888	34.05 %
li	351,076	359,032	731,284	103.68 %
ijpeg	1,062,092	1,021,080	2,471,196	142.02 %
perl	1,076,740	1,006,968	2,986,192	196.55 %
vortex	2,462,200	2,555,924	6,678,084	161.28 %
test suite	134,828	130,712	204,224	56.24 %
philspel	15,208	14,724	30,520	107.28 %
eval-emon	164,076	119,224	376,024	215.39 %
measure1	173,640	172,064	480,188	179.08 %
sm	766,040	528,072	2,035,164	285.40 %
Avg. (SPEC)	–	–	–	159.40 %
Avg. (Non-SPEC)	–	–	–	168.68 %
Avg. (All)	–	–	–	162.97 %

Table 5.3: Size of generated object files, in bytes

5.2 Debugger Data

In the implementation of our ideas we chose to perform limited eviction recovery within the debugger, only capturing the values of evicted variables within those functions where the user has already specified a control breakpoint. The reason given for choosing this solution, rather than performing eviction recovery throughout the entire program, was the claim that attempting the latter would have too much of a negative impact on the execution time of the program, while the solution we chose would not.

To verify this claim we took several different timing measurements within the debugger. These measurements were taken by running our benchmark programs from within the debugger, using various levels of eviction recovery. We modified the debugger to perform eviction recovery in the specified functions without there being any control breakpoints, as such breakpoints would interfere with our timing measurements. The time it took each program to execute was measured. As the most important question to answer is not how long it takes the CPU to process, but how long the user has to wait for the program to complete, the time measured was the total elapsed time, from the user's perspective. This time was obtained by modifying the debugger to print the current system time both when it initially starts executing the program to be monitored, and when the execution of the program terminates. We then take the difference between these two times.

Finding appropriate functions in which to perform eviction recovery for our measurements was more difficult than we anticipated. The SPEC benchmarks, because they are used to measure compiler and CPU efficiency, are very CPU intensive. We came across one case where a single function was called over 100 million times during execution. Although for each call only 30 eviction recoveries were performed, the overall result was a total of *3 billion* eviction recoveries being performed. We were unable to measure the total execution for eviction recovery in that particular function. In another instance a function was called over 500,000 times, resulting in over 7 million eviction recoveries.

Because of situations like these we were unable to find suitable functions for measuring the effects of eviction recovery for three of the SPEC benchmarks, `gcc`, `m88ksim`, and `vortex`. `Test suite` and `sm` both require much user interaction when executing, and so no meaningful timing measurements could be taken for them either. The results of our eviction recovery measurements on the other eight benchmarks are shown in Table 5.4. All the times shown are in seconds.

	Time w/o Recovery	Time w/ Recovery	Slowdown	No. Static Traps	No. Dynamic Traps	Avg. Time/ Recovery
go	666	706 (4 funcs)	40	37	1,614	0.025
compress	784	788 (1 funcs)	4	19	552	0.007
li	566	799 (1 funcs)	233	12	21,600	0.011
ijpeg	270	377 (6 funcs)	107	64	8,136	0.013
perl	309	323 (7 funcs)	14	74	590	0.024
philspel	1	122 (3 funcs)	121	16	9,940	0.012
eval-emon	1	97 (4 funcs)	96	570	612	0.168
measure1	9	1,985 (1 funcs)	1,976	149	78,204	0.025
Average	–	–	–	–	–	0.036

Table 5.4: Eviction recovery timing measurements (in seconds)

The first column shows the time it took to execute the program without any eviction recovery. The second column shows the execution time with eviction recovery being performed in one or more functions. The number of functions with eviction recovery is shown in parentheses. The third column shows the difference between the first two columns, indicating the time spent on eviction recovery. The next two columns show the number of eviction traps set (static) and the number of eviction recoveries performed during execution (dynamic). The final column shows the average time spent on each eviction recovery. As can be seen this varies anywhere from 0.007 seconds to 0.168 seconds, with the overall average being 0.036 seconds.

At first glance this slowdown may seem a heavy penalty to pay for eviction recovery. However the impact really depends on the dynamic number of eviction recoveries performed. In all the benchmarks shown except one, the overall execution times were extended by four seconds to at most two minutes. This is not bad. For `measure1` it was extended by thirty-three minutes, but 78,204 eviction recoveries were performed.[1] Based on our data and observations it is correct to state that performing eviction recovery everywhere would cost too much in terms of execution time.

Given situations such as the single SPEC benchmark function that resulted in over 3 billion eviction recoveries, and which, at an average of 0.036 seconds per recovery would

[1] The function contained several loops that executed thousands of times.

take roughly 3.4 *years* to execute, one might be misled into assuming eviction recovery is too costly even within a single function. However one should recall that eviction recovery will only be performed in a function while the user has a control breakpoint set in the function. A user is highly unlikely to want to stop in each of the 100 million calls to the function mentioned above. Even if the user did so, the time spent by the debugger in processing the user breakpoint would overshadow the time spent performing eviction recovery (less than one second per function call). As soon as the user removes the breakpoint from the function, the debugger will stop performing eviction recovery, and execution should return to its former speed.

Another possibility to consider is speeding up the execution time per eviction recovery by using code patches rather than full-blown breakpoints. In our current implementation each eviction recovery requires two full context switches, which is expensive. If one could discover a way to perform eviction recovery without these context switches it would significantly lower the time cost.

5.3 Usability of the Optimized Source

The third question which we wanted to answer, how difficult it is for people to use and understand the optimized source code for debugging, is the most important one. If people cannot use and understand the optimized source code, then this entire approach to debugging optimized code becomes invalid. At the same time, this is the most difficult question to answer. Unlike such information as average increase in the size of debug information, or the average slowdown of the compilation, the usability of the optimized source code is not something that can be easily and objectively measured. It is subjective for each user, and the same piece of optimized code might get two very different usability ratings from two different users. Some factors that may affect whether or not a user finds the optimized code usable and understandable include how much experience the user has attempting to maintain code that other people wrote; how well the user understands the types of transformations an optimizer might perform on code; how familiar the user is with the programming language being used; how much experience the user has writing programs; and how well the programmer knows the particular program that was transformed. In addition, the manner in which the optimized source is presented to the user, and the types of help and explanations (if any) provided by the debugger, will have a huge impact on the user's impression of how easy or hard it is to understand the optimized source.

In order to collect some information on the understandability of the optimized source, we made our implementation available to people at SGI and to other computer science graduate students at the University of California, Berkeley. We asked people to try out our tools for debugging optimized code, and to give us some feedback on how usable they found the system.

Most of the people who tested our system did not really have a bug they were trying to find. They monitored correctly working programs, stepping through execution and examining variable values, in order to get a feeling for how easy or difficult it was to use and understand the optimized source. A significant portion of those who were interested in trying out our tool did not have any C programs of their own to use. We supplied them with three programs they could try: a small (100 lines) quicksort program, a Lisp interpreter program, and a file compression program. The last two were from the SPEC95 C benchmark suite of programs. About half of the people who examined the optimized source using our tool were graduate students, specializing in many different areas of computer science. The other half were programmers from SGI, all of whom work in the compilers group.

5.3.1 Summary of User Feedback

In the end sixteen people tried out our tool and gave us feedback on it. Out of those sixteen, two people had a lot of trouble understanding the optimized source. One of those two did not know anything about optimizations, and felt that this fact, combined with the facts that there was no assigned task to be performed and that the code being examined had been written by someone else, made it very difficult to get a real feeling for how easy or hard it would be to use optimized source for debugging. The second person felt that the main obstacles to using and understanding the optimized source were the meaningless names given to the compiler temporary variables introduced into the optimized source, and the bugs in our source-to-source correspondence mapping. The second person felt that if these problems were fixed the optimized source would be usable.

In addition to the two who had a lot of trouble understanding the optimized source, it was unclear from the feedback we received from a third person whether or not that person could use and understand the optimized source.

The remaining thirteen users were all able to use and understand the optimized source, with varying levels of comfort. One commented that it was a bit confusing at first, but one could get used to it after a few tries. Another person expressed the opinion that people who understand optimization techniques would have no trouble understanding the optimized source, but those who were unfamiliar with optimization techniques might find the optimized source hard to use. A comment we received from yet another person was that the tool would probably be much more useful to compiler developers and testers than to normal compiler users.

The one person who supplied her own code, and who had a bug to find, did succeed in finding her bug using our tool. Several other people commented that if they had had this tool a few months ago, it would have helped them find bugs they had been working on.

A frequent comment in the feedback was that showing both the original and the optimized source, and highlighting the correspondences between them was crucial to understanding the optimized source. A common complaint, echoing one of the two users who had trouble with the optimized source, was that the compiler temporary variables introduced into the optimized source have meaningless names, and this made it hard to relate them back to the original source. We received several suggestions for dealing with this problem, including using more meaningful names, removing these variables from the optimized source wherever possible, and using special colors, highlighting, or tool-tip texts to indicate how they relate to variables in the original source.

As this is a prototype system, not all the bugs have been worked out of it. The users found several which they reported. Fixing these bugs should make understanding the optimized source much easier. They also sent us quite a few suggestions for improvements and features they would like to see. Figure 5.2 lists a summary of the bugs reported, and Figure 5.3 lists a summary of the suggested improvements. Appendix A contains the complete texts of the feedback we received, with any personal or identifying information removed.

5.4 Summary

In this chapter we have shown that the machine code generated by our compiler is reliable and correct. We have presented measurements of the compilation speed of our compiler, the size of the optimized source files generated, and the size of the debug information. We have also shown the impact of eviction recovery on the execution times within

- Dereferencing compiler temporary variables (pointer values) does not work properly
- Could not always print values of all variables shown in optimized source
  - Some “dead” variables still show up in source
  - Not all temporary variables used by compiler are shown in source
- GUI error caused some attempts to disable breakpoints to fail
- Source-to-source mapping in GUI not always correct
- Variable range table information, in symbol table, occasionally off by one instruction address
- GUI is slow to respond
- Standard dbx command “set $hexints=1” causes GUI to crash
- Dbx loses context after program being monitored terminates (reaches end of normal execution)
- Optimized source is not always correct, especially for more complicated programs

Figure 5.2: Summary of bugs reported in our tool.

the debugger, and we have presented feedback from users who tested our tool and examined the optimized source.

Using our compiler takes significantly longer than using the original compiler. This fact is not particularly disturbing, however, as our implementation focussed on correctness rather than efficiency. There are many places in our implementation where the compilation speed could be improved, particularly if one had free reign to modify or redesign the compiler.

Not surprisingly, the optimized source is larger than the original source. However it is usually less than twice the size of the original. Adding all the debug information necessary for accurate debugging of optimized code significantly increased the size of the object files (on average making them 1.8 times larger). Unfortunately optimizations greatly complicate the debugging information one needs to maintain. Without this extra information one cannot hope to have a debugger that behaves correctly in the presence of optimizations.

The vast majority of the user feedback we received was positive. People did encounter some bugs in our prototype, and there were many suggestions for improvements. Overall people seemed to feel that the optimized source is both usable and understandable,

- Use more meaningful names for compiler temporary variables
- Support some more sophisticated optimizations
- Extend to C++
- Add more explanations of how & why optimizations are performed
- Use tool-tip text to explain some optimizations in optimized source
- Add declarations for compiler temporary variables
- Remove uses of compiler temporary variables wherever possible
- Add comments indicating which macros were expanded
- Add syntax highlighting
- Make the GUI windows movable and resizable
- Add a menu for common commands
- Add a window for watching variables
- Add program slicing to help understand compiler temporary variables
- Add more buttons & features that are standard to window-based tools
- Make the debugger explanations more helpful, less vague

Figure 5.3: Summary of suggestions for improving our debugger tool.

and that our approach to debugging optimized code is a good, viable, usable approach.

Chapter 6

Conclusion

This dissertation has examined both the theoretical and the practical issues involved in non-transparent debugging of optimized code. In particular it has explored the difficulties and benefits resulting from using an optimized version of the source program, which reflects the effects of some of the optimizations performed by the compiler, for all user debugging activities. This work culminated in the implementation of these ideas within an existing commercial compiler and debugger, and the collection of measurements and user feedback on the system. These results indicate that this is indeed a viable approach to debugging optimized code, one that greatly simplifies many of the problems and significantly advances the technology in this area.

6.1 Summary of results

The problem of designing a source-level interactive debugger for optimized code has received a fair amount of attention over the last fifteen years. While in many situations compiling a program without optimizations and using a standard debugger on the unoptimized code may be sufficient to find and correct all the errors in a program, there are situations wherein this approach may be insufficient. Some of these cases include when the program's behavior differs depending on whether or not optimizations are applied, when debugging must be performed on an optimized core dump, or when either because of resource requirements of the program itself, or peculiarities of the particular compiler used, the code cannot be compiled and executed without optimizations.

In such situations debugging must be performed on the optimized program. Since "standard" source-level interactive debuggers cannot correctly handle optimized code, this leaves the user with the option of either attempting to debug the program at the assembly code level (which requires intimate knowledge of the assembly language for the machine in question), or else using a tool specifically designed for debugging optimized code.

In the past designers of debuggers for optimized code have advocated taking a transparent approach, presenting the user with the original source program, and attempting to respond to all user requests exactly as if the program were unoptimized, in effect making the optimizations *transparent* to the user. These approaches all have serious flaws. To begin with, there are certain situations in which it is impossible for the debugger to hide the effects of all optimizations from the user. In such cases, these debuggers fall back on *truthful* behavior, reporting to the user that optimizations either caused variables to contain unexpected values, or that because of optimizations a variable's value may be unavailable, or explaining other inconsistencies between the user's view and the underlying, executing code, which have been caused by optimizations.

In addition to full transparency not being realizable, by attempting to map correspondences between the optimized target program and the original source program, which may be quite far removed from one another due to optimizations, these transparent approaches greatly add to the difficulty and complexity both of implementing the compiler to build and maintain these complex correspondences, and of implementing the debugger to decipher and use them. Finally, by forcing the user to look at a source-level representation that bears very little resemblance to the underlying executing code, these approaches may actually *mask* bugs from the user, making them harder to detect.

By taking a non-transparent approach to debugging optimized code, allowing users to see how the optimizer altered their programs, we have greatly simplified many of the problems associated with debugging optimized code.

6.1.1 Revealing effects of optimizations

In this dissertation we have presented an alternative approach to debugging optimized code. By creating a source-level representation of the program that reveals many of the effects of optimizations performed by the compiler, our solution allows users to obtain a much clearer, more accurate picture of what is occurring in their programs. Furthermore by removing the artificial constraints involved in hiding all the optimization effects from the end user, this approach greatly simplifies the implementation of debuggers for optimized code, both in the compiler and in the debugger. In fact the *currency problem*, which is the single most difficult and complex problem facing transparent approaches to debugging optimized code, vanishes entirely, merely by allowing users to see the actual order in which source assignments take place in the optimized program. (Recall that the currency problem occurs because the actual value a variable contains may be different from that which the user expects, due to variable updates occurring in different orders and locations than that indicated in the original source program.)

We have described a particular non-transparent approach to debugging optimized code, namely generating an optimized version of the original source program, based on the actual optimizations performed by the compiler. This optimized source program reflects those optimization effects which have the greatest impact on the ability of a source-level debugger to give the user accurate information about the executing target program, namely those optimizations that reorder, rearrange, insert, or remove code from the program, and those which affect when and where source-level variables are updated. We have carefully described both the theoretical and practical considerations necessary for designing and implementing a tool to generate such optimized source.

We have demonstrated the practicality of these ideas by implementing them within a tool Optview, which we have embedded with a commercial, highly aggressively optimizing compiler, the SGI Mips-Pro 7.2 C compiler. Optview generates optimized source for C programs. We have also modified the SGI dbx debugger to work correctly using the optimized source, and we have implemented a graphical user interface to help users understand the optimized source and relate it back to the original.

6.1.2 Key Instructions

In addition to describing and demonstrating a non-transparent approach for debugging optimized code, we have introduced the concept of *key instructions*. Although the idea of identifying, for each source statement, the single machine instruction that most closely embodies the semantics of the statement was mentioned in passing both by Zellweger[47] and by Copperman[15], neither of them pursued the concept. We initially worked on the idea of key instructions only for their use in aiding Optview to generate the optimized source programs. However we have found key instructions to be extremely versatile and useful. In fact much of their usefulness is entirely independent of whether one is taking a transparent or non-transparent approach to debugging optimized code. In addition to helping determine the final ordering of source statements in the target program (to reflect the same order in the optimized source), key instructions provide an easy and natural solution for the code location mapping problems. Key instructions are good indications of where to set control breakpoints that correspond to source statements, and they are also useful for reporting the location of execution errors.

Finally using key instructions allows data for the variable range table to be collected by performing a single dataflow analysis pass on the compiler's final internal representation of the program. This is much simpler and more robust than the old approach of tracking variables' locations through each step of the optimizer. By waiting until the very end of the code generation phase of the compiler, the representation on which the dataflow analysis is performed explicitly shows all reads and writes to memory, all register allocations, all register spill and recovery code, etc. This approach is completely independent of which optimizations the compiler performs and how those optimizations are implemented. The analysis only needs to be performed once, as the data at this stage will not change, and the data obtained from the analysis is completely accurate. However this technique could not be used if one could not determine where, in the representation, assignments to source level variables were occurring and which variables were receiving the assignments. Prior to our introduction of key instructions, this information was unavailable at this stage in the compilation. Thus previous approaches to creating variable range tables had to pursue more difficult, less robust options.

6.1.3 Eviction recovery

By implementing our approach not only within the compiler, but also within the SGI dbx debugger, we have been able to describe, implement and test a partial solution to the eviction recovery problem. Recall that the eviction problem arises because a variable which may be within the current scope can also be "dead", i.e. its value is no longer needed for the computation. In such cases, the optimizer allows the variable's storage location to be re-used for some other value without saving the value for the dead variable anywhere. From the compiler's point of view this is not a problem, as the dead value is not needed any more. From the debugger's point of view this is a nightmare, because although the value may not be needed any further in the computation, the user may still wish to examine it. However as the compiler has allowed the value to be overwritten without saving it, the requested value is lost and unavailable. We have designed and implemented a scheme whereby the debugger captures and saves these dead values before they are overwritten and lost, thus allowing the debugger to answer user queries about these values. Our solution is a partial solution, because the debugger only performs this eviction recovery in those functions where the user has set a control breakpoint.

6.1.4 Using and understanding optimized source

As we mentioned previously, taking a non-transparent approach to debugging optimized code greatly reduces or eliminates many of the problems involved in this topic. However using optimized source to represent some of the effects of optimizations also introduces a new concern: users are now confronted with source code that is different from what they wrote, and they are expected to understand it and use it to debug the program. It is *critical* that users be able to understand and use the optimized source in this approach. This requirement has been known from the beginning and has shaped many of the design decisions in Optview. Some of the decisions made to minimize the relative strangeness of the optimized source include not attempting to represent all optimizations in the optimized source; performing all transformations at the source level, starting with the original source, rather than attempting to reverse engineer the optimized source from the optimized binary, and re-using fragments of the original source program wherever possible in the optimized source, especially white spaces, comments, declarations, and high-level control-flow constructs.

In addition to directly minimizing the differences between the optimized source and the original source, we further reduced the burden of understanding the optimized source by introducing a graphical user interface (GUI) front end to the debugger which shows the user both the optimized source and the original source simultaneously, and highlights their correspondences.

6.2 Practicality of this approach

In addition to designing and implementing the solutions described above, we collected data on the speed of our compiler, the size of the files it generates, the impact of eviction recovery on the execution times of programs, and the understandability and usability of the optimized source programs generated by Optview, in order to determine whether or not this approach is really practical.

Our modified version of the compiler took significantly longer to compile the test programs than the unmodified version of the compiler (on average it took 6.6 times longer). However, as explained in great detail in the previous chapter, this is not really a meaningful measurement, as our modifications were never made with speed or efficiency in mind, and we were forced to work within the very narrow constraints of an existing compiler, certain portions of which we were not free to modify.

The optimized source files were, on average, roughly 50% larger than the original source files. Some increase in file size is to be expected as our implementation occasionally adds to the original source when creating the optimized source, but it never removes anything from the original source. This size does not seem unreasonable.

The object files generated by our modified compiler were also larger than those generated by the unmodified compiler, averaging 1.8 times bigger. This increase in size is due to all the additional debug information that needs to be recorded and stored for accurate debugging of optimized code. It includes more accurate source position mappings between the target program and the optimized source; eviction recovery information for the debugger; the variable range table information, for looking up variable values; and, information about any compiler temporary variables introduced into the optimized source. With the exception of the information about compiler temporary variables, all the information listed above is necessary for accurate debugging of optimized code, independent of whether one is taking a transparent or non-transparent approach. Without it one cannot hope to obtain correct information about the state of the executing program. Also one should remember that the debug information generated by the unmodified compiler, while smaller in size, is also incomplete and inaccurate in the presence of optimizations.

The eviction recovery measurements indicate that when eviction recovery is performed in a single function, the impact on run time is minimal (0-9 seconds). However increasing the number of functions in which eviction recovery is performed only slightly causes a much more visible degradation in the execution time of the program (14 seconds up to 13 minutes). From these numbers, and from the fact that we were unable to get any program to complete in a reasonable amount of time when performing full eviction recovery, we believe it is fair to conclude that until a faster, more efficient eviction recovery scheme is discovered, full eviction recovery is impractical.

The user feedback we received, indicating the usability and understandability of the optimized source generated by Optview, was mostly positive. Those who did have some trouble with the optimized source had suggestions that they believed would have made the optimized source usable for them. The vast majority of the users found the optimized source usable, in spite of some bugs they found in our system. The overall conclusion we have reached from reviewing the user feedback is that, as it stands, our optimized source is a little difficult to understand, but is usable. There are a few modifications we could make to the system and errors we could fix that would make it much easier to use and understand. Most of the problems that users had seem to be attributable to bugs or peculiarities of our implementation, rather than to any flaw in the basic ideas and concepts. Thus, while our system still needs work we have demonstrated that, overall, this is a viable, reasonable and practical approach for debugging optimized code.

6.3 Future Directions

While the system we have designed and implemented is sufficient to demonstrate the viability of this approach to debugging optimized code, there is much room for future work and improvements. For example the optimizations that Optview currently incorporates into the optimized source do not radically alter the original source program. The main changes Optview makes to the source are to reorder and move statements around somewhat, insert assignment statements, and substitute compiler temporary variables into expressions. One avenue it would be interesting to pursue would be to investigate extending the optimized source to include some optimizations that more drastically change the appearance of the source. Good examples of this type of optimization include function call inlining, and high-level loop nest optimizations. Both of these types of optimizations are representable at the source level, but both would require significantly changing the appearance of the source program. It would be interesting to further pursue exactly how to represent these optimizations in the source so as to minimize the user's difficulty in recognizing and understanding the source. In parallel to designing the optimized source itself, one would also need to investigate various possible methods for presenting the drastically altered optimized source to the user. One possibility is to use multiple layers of representation. The tool could actually generate two or three representations of the optimizations, with various levels of accuracy or resemblance to the original source. The user would then have the option of navigating between these various representations moving either towards greater accuracy in the optimization representation or towards greater resemblance to the original source (at the expense of optimization accuracy). The GUI interface would be absolutely vital in helping the user to understand and navigate through these more complex optimizations. The GUI would also require a more complete help system, perhaps even a simple tutorial on optimization techniques, for those who would be interested. All of this would be a very interesting, valuable area to pursue.

Another area of research interest would be the use of optimized source in allowing debuggers of optimized code to update variable values. To date all debuggers for optimized code (including ours) automatically disable allowing users to directly update variable values. However this is a very common feature in standard debuggers for unoptimized code, and one that many programmers would desire. The reason that debuggers for optimized code disable this functionality is as follows. Optimizers always attempt to determine the actual values of the variables they are manipulating, if possible, as that often enables them to perform more optimizations. Some of these value-specific optimizations include directly using the value of the variable in the target program instead of referring to the variable by name, or even eliminating entire pieces of the computation because the optimizer can determine, if the variable has a particular value, that a particular piece of code will never be executed. Thus the conventional wisdom has been that if users are allowed to update variable values after optimizations have taken place, they may invalidate some of the assumptions the compiler made, and the result could cause the program to behave incorrectly. However if all of the assumptions made by the optimizer can be explicitly shown in the optimized source (i.e. users can see where constants are used instead of variable values and which pieces of code have been optimized away) then perhaps users can update variable values after all with no bad effects.

It would also be useful to pursue the possibility of creating and using code patches to perform eviction recovery. In the scheme we implemented, the debugger has to perform a complete context switch (two, actually) every time it needs to capture an evicted value. This is very expensive. If it were possible somehow to create code patches for capturing these values instead, eviction recovery could run much faster, and it might even become practical to perform full eviction recovery.

Finally in our work we have concentrated heavily on the C language, as that was the compiler we had to work with. It would be both interesting and beneficial to further investigate the practical side of what it would take to implement these ideas for other languages, especially C++ and Java, as those are two widely used languages, each with interesting features. Extending the work to C++ would mean incorporating function call inlining into the optimized source, as well as type inferences and dynamic dispatch (both of which would be relatively simple). Covering Java would mean dealing with garbage collections, which may add another layer of complexity, especially to the problem of tracking variable locations. In addition, there are languages such as ML which have very different kinds of optimizations, and which also have much more complete language descriptions. It is less clear whether there would be any benefit in applying this kind of approach to those languages.

6.4 Advantages and contributions of this approach

6.4.1 Advantages of this approach

In this dissertation we have presented a complete solution for debugging optimized code, outlining and implementing the work to be done in the compiler, in the debugger, and in the tool to generate the optimized source. This solution does not in any way limit or alter the optimizations performed by the compiler, nor the functionality of the debugger[1]. It does not require special recompilations of the source, it puts no limits on the users' actions, and it does not alter in any way the code generated by the compiler. The code that is debugged is exactly the same as that which first exhibited the error.

Furthermore, this approach required only minimal changes to the symbol table and to the debugger. The SGI Mips Pro 7.2 C compiler uses the DWARF 2.0 Standard[32] symbol table format, which already contains all the features necessary for debugging optimized code. Implementing all the changes necessary for the symbol table and the debugger combined, including the time required to learn them, took approximately four months.

The optimized source code generated by Optview is very versatile. In addition to being useful for debugging of optimized code, such optimized source would also be very helpful to those who work on performance analysis and tuning, as those people are confronted with many of the same problems facing people debugging optimized code: they

[1] With the exception that is does disable users updating variable values from within the debugger.

must find some mapping between the optimized target program and the source program in order to attach meaning to the numbers and measurements they obtain. Optimized source would also be useful as a teaching tool for classes or individuals who wish to learn more about optimization techniques and who do not wish to examine the assembly code.

Yet another benefit of this approach to debugging optimized code is that, for the very first time, it allows accurate blame assignment. In those cases where turning on optimizations causes program behavior to differ, the very first question that needs to be addressed is: did the optimizer do something wrong, or was there a bug in the original program, or are the language semantics underspecified? Other approaches to debugging optimized code have always started with the assumption that the optimizer is correct. Unfortunately this assumption is not always true. By showing the effects of optimizations in the optimized source, this approach allows the user to see what the optimizer actually did, for those optimization whose effects are revealed in the optimized source. Thus the programmer can look at the optimized source and decide if the optimizer did indeed do something incorrect, or if the bug was in the original program, or if the program is, in fact, correct according to the language definition.

The final advantage of this is approach is, as we have demonstrated, that it can be retrofitted to existing compilers and debuggers.

6.4.2 Contributions of this work

By taking a non-transparent approach to debugging optimized code, we allow users to see which variables get updated and what actual values these variable receive. This in turn means there are no out-of-order variable updates. Thus the entire currency problem has been eliminated.

This approach also introduces the concept of key instructions, and explains their many uses not only for generating optimized source, but also for providing simple solutions to the code location mapping problem and for collecting the variable range table information, independent of the transparency of the solution being implemented.

We have pioneered a scheme for eviction recovery within the debugger. Previous research stopped at discovering evicted variables, proposing only to tell users that the desired values were unavailable.

We have implemented a simple graphical user interface and have shown how such a device can drastically reduce the burden on the user of understanding the optimized source. The user feedback we received confirmed that using the GUI allowed them to use and understand the optimized source; without it they would have been lost.

Overall we have demonstrated the practicality of revealing optimization effects to users for debugging optimized code. We have provided a general theoretical framework for our solutions, of which Optview is one implementation.

Appendix A

User Feedback Data

Below are the complete texts of the user feedback we received on our debugger tool. The lines mark divisions between feedback from different users.

```
1. It was very confusing to me in the beginning, to figure out where
   and what the optimized code was. I had some trouble mapping the
   optimized code to the original code I wrote, because it looked so
   different. But that's because your highlighting wasn't working, and
   I think it would have helped a lot.

2. I think the split screen approach works, because it gives a
   reference point of the original code. However, I wonder if the
   person using the debugger was not the author of the code, then
   maybe it would be actually easier because he/she won't be
   constantly trying to map from optimized to original code.

3. The introduction of new variable names which meant nothing threw me
   off a bit. I didn't know what cse_var_x corresponds to. Maybe
   assigning names that are similar to the original variables can help
   a bit.

4. Although confusing a first, I still found my bug. So I think after
   a few tries, people can use this debugger to looked at optimized
   code, instead of "make clean; make unoptimized" to debugged, and
   then after the bugs are found, "make optimized" again. This is what
   I usually do, because I cannot step through the original code when
   the executable is actually optimized.


--------------------------------------------------------------------------
```

Overall I really like it and think that it can be very useful for development, however it may take major effort to make it from a prototype to a real product.

Features I like:
1) highlighting of optimized code and its corresponding source
2) no problem printing out values of variables and parameters in the optimized source
3) low-level dbx debugging features continue to be supported
4) generated names reflect what kind of optimizations has been done

Things that are of concern and may improve on:
1) optimizations supported can be extended to more sophisticated ones, like loop-nest or even mp-related
2) users need to understand the optimizations performed to get a clear picture of what's going on and how to debug
3) extending the language support to C++ will be more useful for the compiler group people

----

Overall, I think it is a reasonable tool to debug optimized code. The interface was pleasant, it was VERY useful to have both sources visible with the debugger pointing out the line it is executing in BOTH sources.

There are, however, some issues with the examination of data. It would be nice to be able to dereference cse_ variables. Some variables are apparently tossed out in the optimized code, it would be useful if we could tell which ones. Some of the assignments to cse_ variables were not in the optimized source, which made the execution a bit puzzling. It also seemed difficult to disable breakpoints.

But the breakpoint system worked well, even when including conditions on breakpoints.

(In somewhat random order). I think seeing the correspondence between the optimized and unoptimized code is very useful for debugging under conditions where it is an issue.

However, (on the list of things which would be nice but aren't implemented) I think the mechanisms by which the compiler distorts the code are necessary to understand what is going on. I spent a fair amount of my time with the system trying to understand what the mechanism and rational for the various transformations was. I know that some of it was labeled through the comments, but a production system should document "why" and "how" of the optimization, not just "what".

Your debugger does the hard part (isolating the "what"), with the why and how being more a matter of interactive documentation and annotations. Similarly, some notion of position or completeness when a single statement occupies multiple steps is desirable.

In all, it seems like a rough prototype of a very useful tool, akin in some ways to Purify, a powerful tool used somewhat as a last resort, when nothing else can do the job.

I found that Python (is that its name?) was a very reasonable approach to trying to debug problems that surface only in optimized code. I think that a more aggressive effort on the user-interface aspect of the program could increase its user-friendliness greatly. As it stands, it is not a viable substitute for standard debugging; but with the enhancements below, I think it could be made to be as usable as a standard debugger.

The main difficulty I encountered was matching up the semantic position of the optimized and unoptimized sources. It would have been much easier if the relationships between the optimized source variables and unoptimized source variables were made explicit. For example, if the variable names were color-coded to reflect their relationships, then a matching could have been easier. Also, naming the variables introduced in the optimization process could have made the link easier; for example, instead of cse_var_77, a name like offset_index_77 would make it clear that a certain variable is being used to calculate an array offset.

Another difficulty I found was in accessing variable values by name; in the optimized source-building process, it seems that some variable names are lost, making access to them during debugging impossible.

This seems to be a critical flaw, but hopefully one which can be fixed easily.

One suggestion which may or may not be possible to implement is the following: it would be very helpful if the optimizations could be explained, with the use of tool-tip texts, for example. For instance, if I move my mouse over a statement which has been repositioned, it would be nice for a small blurb to come up and explain briefly why it was moved. Even if a full explanation cannot be given, it would be nice for it to acknowledge the repositioning, so the user knows that it is not an error. Presumably, since the compiler has a finite number of transforms, this is (in principle) possible.

It seems like a really neat tool, which allows you not only to see the source correspondence between statements in the optimized code and the corresponding statements in the original code, but also to debug and make references to compiler-generated temporaries and set breakpoints at lines in the optimized version of the code. It is also worth noting the neat feature indicating uninitialized values when doing a "print" command before the first def of a variable.

This provides a much finer grain of debugging than if all debugger commands only could be issued with respect to line-numbers and variables as they occur in the original source. Debugging from the perspective of the transformed source is definitely the right way to go.

This tool provides a debugger (for transformed source), a view of of transformed source in a high-level language (same as input language), and a mapping from statements in the transformed source to statements in the original source. These utilities could be immensely useful for many purposes, among which the following comes to mind:

1) To look at values held in compiler-generated temporaries, and general debugging with respect to the transformed code.

2) To investigate transformations the compiler has done (similar to the way we use whirl2c/whirl2f now) to tune for optimal performance. This tool currently operates at a lower level than whirl2c/whirl2f (which does not work well after wopt), and can be used to tune for low-level optimizations.

3) Similar to 2, but as an aid to compiler writers to find bugs and places where transformations could have been done better (beats looking at assembler or compiler-intermediate code).

Overall the current version of the tool demonstrates well how the ideas implemented provides a very useful and powerful mechanism for debugging optimized programs. It worked well for almost all compiler commands I tried, and the transformed source, variable values, and the mapping to original source were correct most of the time. As proof of concept, this is excellent. As a productivity tool, it needs to be made more robust:

- After doing a "next" from a "def", I sometimes had to do a "step" to see the right value in compiler generated temps.

- Sometimes compiler-generated temps "lose" the value before going out of scope (should have PU scope?).

- It would be useful to see the live-range and type of compiler-generated temps, either by declarations in the transformed source, or by other means. The type can be seen by doing "whatis tmp-name", but is not currently an accurate type.

- The user-interface was a little shaky, and died a little too often when certain dbx commands where issued.

These are all really small problems that could be relatively easily fixed, and they did not diminish the overall impression of the tool as a very user-friendly debugger of optimized code, which is way beyond anything else I have seen w.r.t. debugging optimized code.

Feedback concerning Tool for Debugging Optimized Code

How useful is it? I split the user community for this tool into two categories: compiler developers/testers and compiler end-users. I believe this tool would have differing levels of usefulness across these two user communities.

For compiler developers and compiler testers, this tool would be pervasively useful primarily in tracking down bugs created or uncovered by compiler optimization. It is sometimes the case that an optimizing compiler will perform an optimization incorrectly such that an application will then produce incorrect results. These types of bugs can be very difficult to track down with ordinary debuggers. As a compiler tester I can think of number of occasions where I would have been able to solve a problem many times faster using a tool like this rather than a standard debugger. I prefer the methodology used in this tool, to those found in other optimized debuggers in that I'd much rather look at the optimized source code to find the problem rather than having a problem masked by only being able to look at the original code.

For end-users, this tools would be less generally useful because many user communities compile with no optimization during development and only use compiler optimization near the end of product development. They do this for two reasons, their product builds faster with no optimization, and is easier to debug. While this tool would make it easier to debug optimized code, I don't believe that the tool by itself is compelling enough to get users to only do optimized builds throughout the development cycle. So that leaves many end-users using this tool only at the end of the development cycle. I would also say that since it is difficult for most software developers to read optimized code, that whenever possible they would prefer to use a standard debugger to debug unoptimized code. However in situations where they had to debug optimized code, they would find this tool useful. It should be noted that my projection of the lack of general usefulness of this tool in the broader user community is based more on the lack of use of compiler optimization throughout the development cycle rather than any conceptual or implementation problems with Caroline's debugger.

So, in a gist, it's not quite ready-for-prime-time.

1. dbx is lame. It'd be cool to force SGI to use gdb.

2. I'd like an emacs interface, so I can be in my environment where I'm most comfortable coding. Plus, you get syntax highlighting for free.

3. I don't quite understand the cse_var variables. I have a lack of confidence that I know which variables they came from. Perhaps

renaming them to something more like the original source variable. Perhaps a tooltip-like approach where you pop up a little display of what the variable belongs to when the user lingers their mouse on top of it.

4. You have to perform the copy propagation and removal of cse_vars whenever possible. It'll make things MUCH easier to understand in the optimized source.

5. Macro substitution is a double substitution. The optimized source only shows the final target. It'd be nice if the comments in the opt source could indicate which Macro this expression came from and what its non-optimized representation looked like.

6. You have to work on getting the correspondence between the two sources working better. Perhaps making your parser understand // C++ comments and if statements where the consequent is on the same line as the predicate.

7. If you don't switch the environment to emacs, here's a list of UI desires:

   a. syntax highlighting
   b. movable pane between the opt source and regular source.
   c. vertical layout of windows instead of horizontal.
   d. multiple windows, one for opt source, one for reg source so I can position them where I want.
   e. ability to make the dbx window bigger.
   f. Can you click on a line number to make a breakpoint? I couldn't tell.
   g. Add a menu for common file commands and the more obscure debugging commands.

8. Parser should be faster when switching source files. It's dog slow now. Trying recoding it in C?

I think you're on an exponential for understanding. If you were to make a few of these changes (3, 4 and 6 especially) it would make a world of difference.

Generally I was pleased with the design of the optimized code debugger. It appears that one can reliably display the values of variables in optimized code, both variables in the original source and in the optimized source. I think the idea of showing optimized source is an excellent one. Besides not being able to display variables, running the ordinary debugger on optimized code is usually very confusing since the line numbers jump all over the place. Seeing line numbers change in an orderly fashion is quite soothing, and the optimized source is interesting in itself to show what the optimizer is doing.

I would be interested in seeing this extended to C++ code.

I encountered a few bugs while experimenting with the optimized code debugger.

My program called a function "fill" to initialize an array of 100 integers and then a function "sum" to add them up. If I set a breakpoint at fill, run to the breakpoint and return, I cannot at that point display the elements of the array, although I can display them correctly at the entry to sum.

I could not always correctly display the induction variables, which spend most of their life in registers. I seemed to be getting stale values, since they were never updated.

Single-stepping through the sum function, occasionally I would not get the correct value for the partial result.

Attempting to do set $hexints=1 caused the debugger to crash. This is somewhat annoying, since I prefer to display addresses in hex.

Deleting breakpoints also crashed the debugger.

I used the debugger for a while. My comments:

a) it's not production quality (fixing those problems would be straightforward, I estimate). During initial use it hung up a couple times. With help I learned how to avoid the problems.

b) It is usable and useful, as is.

c) I found the parallel source windows (original and optimizer-output-source) very useful. It was fascinating to see the things the optimizer chose to do represented in source. While I have occasionally traced through assembly code trying to understand why something failed after being optimized, I never pieced together a coherent view over more than a very narrow region (doing things by hand). The consistent transformation of while(){} and for(){} into do{}while() because the first iteration was usually known to pass the test (from dataflow-analysis) was both interesting and reassuring (as to code quality).

d) Another benefit of the optimized source was that macros were shown after expansion. This was particularly helpful in understanding what was going on in a complicated piece of code I did not know much about. Both with simple constants and with things that looked like ordinary function calls really being something complicated that called some function with a completely different name.

If only I'd had it available a few months ago, when a library bug got assigned to me and it was really an optimizer problem! There is good reason to think a tool like this would have helped.

The concept of showing the optimized code for debugging purposes is a good one. Ideally, it would show developers exactly how their code was being executed. However, the optimized code can be hard to read. Showing how the optimized code is related to the developer's code is crucial for understanding.

The implementation is a good start. Highlighting a line in the original and the corresponding in the optimized code is a good idea. However, the names of the compiler-generated variables are hard to understand. It may be good to make the names be similar to those of the original variables that they replace, whenever possible. Of course, the parser bugs make it hard to understand as well, but those will go away as the parser is debugged. A window for watching variables would also be nice, and would reinforce how compiler variables are related to the original variables.

Programmers who are not aware of optimization techniques may be confused by the debugger, but experienced programmers who understand these techniques should not have a problem. The debugger could also be used for teaching purposes, concretely showing students what optimization techniques really do, without needing to go down to assembly code.

First, my impression is that a language-level representation of optimized code is a powerful tool. An optimizing compiler usually presents itself as black box: I say "-O2", and the compiler does something that's described vaguely (at best) in a manual page. But I get no information about the transformations that it performed. All I can do is run the executable (or maybe profile it) to see how its performance and output change.

There are two real benefits of your tool that I can easily see. First, if my program has a bug that shows itself only when I compile with optimization, by far the most direct way for me to uncover the source of that bug is to work with a program representation that reflects the transformations performed by the optimizer. Second, if I'm doing performance tuning, I again want to know what the optimizer has done. Assuming that I've got some understanding of the processor resources that are available (e.g., number of registers), I might be able to evaluate how well those resources are being used.

I found the ability to compare the original (unoptimized) code and the optimized code to be invaluable. Compared to the optimized code, the original code was much easier to navigate (especially since I wasn't familiar with it). I certainly appreciated the opportunity to avoid the complexity induced by optimization when I was looking at initialization code and similar, "less interesting" program parts. The use of highlighting to show correspondence between statements in the two version strikes me as essential, making the optimized code much easier to understand. The two representations aren't quite equals, in that breakpoints an only be set on the optimized code, and changing that would be useful.

As I mentioned, adding program slicing to aid in the understanding of the common-subexpression variables would be a useful feature. With a little practice, though, I wonder if I couldn't learn to use the corresponding-statements highlighting to get the same information.

Finally, the optimizer's loop conversions can be pretty confusing. Absolutely everything becomes a do-while loop, and that made it difficult for me to see which loop construct in the original code produced a given loop in the optimized code. Some kind of loop-construct correspondence highlighting (akin to the single statement correspondence highlighting) would help with this.

All in all, I think this is interesting stuff, and quite well done. It certainly seems useful for debugging, performance tuning, and any situation in which the programmer wants to understand the optimizer's behavior.

The following are some comments/feedback on the compiler and debugger (OPTVIEW) that you developed based on my 25-minute experience playing with it on June 4 (Friday morning):

The strengths of the tool (OPTVIEW):

1. By having the original code and the optimized code side by side, the users are able to understand the effects of compiler optimizations better. When the users select a line from the original code, the tool highlights the new optimized code. This mapping makes it very easy for users to follow the optimized code by working with the original code that they wrote. This strong "binding" between the original code that the users are "familiar" with and the optimized code that are "new" to users is very important factor in the human interface design.

2. The code that has been optimized by a compiler is often transformed in various ways: e.g. some of the original variables are not used, new variables are generated, rearranging of codes/instructions etc. Since the debugger is linked to the optimized code, the users are able to understand the transformed codes better, and access variables generated in the new optimized code that would otherwise be "obscure" to the users.

3. The human interface is fairly straightforward and easy to understand. So the learning curve is minimum.

4. It provides a good high-level view of source optimization.

Possible improvements:

1. Personally, I am used to the window based debugger tool (xdbx) where there are a fancy panel of buttons to click on to do the various tasks such as: printing variable values, creating check points etc. Incorporating these features in the tool will be great.

2. There are several bugs that can be corrected and improved on: (Carolyn, you have already identified these)
   - debugger that works correctly without being confused about which variable to print out.
   - dbx problem, once the program terminates, can't create checkpoints and run it again.

My overall comment is that the debugger looks basically usable. The most important thing to me, at first glance, was to be able to find the correspondence between the original code and the optimized code. At a superficial level, I felt I was able to do that. The preservation of comments, procedure names and levels of indentation helped with that. The code looked similar, and if one has some idea of how an optimizer works, then the optimized code is not completely mysterious.

When you saw me setting breakpoints, it was because I was trying to see if I could set breakpoints at particular points in a branch and look at variables at that point. I think that while I was trying to set one of these breakpoints, your debugger said that the line I wanted to break on was somehow ambiguous. If I were using the debugger, I would want that ambiguity cleared up for me either through and explanation or a suggestion of what to do.

I'm not sure I can evaluate your debugger in any depth. There were a couple things that I lacked going into this evaluation. The first is that I don't in a deep sense know what an optimizer does, or why I would want to debug optimized code. In the past I have run into bugs that caused my code to crash when it was optimized though it ran fine when unoptimized. I assume that that would be one of the reasons for using it. But in those situations, it helped to know what kinds of things the optimizer might be doing. I usually rely on some expert to tell me where the code is breaking and how I could change it to make the optimizer do something different. I'm not sure why I would otherwise be looking at optimized code.Also, you pointed out that in the optimized code window there were comments about the optimizer's actions. They meant basically nothing to me and I can only imagine they would be very helpful for someone who understood them.

The second thing I think I sorely needed was my own code. With my own code, I could have had a better notion if the optimized code looked meaningful to me. Further, I would have had and actual task in mind that I would want to try out on your debugger.

Evaluation of Optview debugger

I ran the debugger on one of my own programs (a C/OpenGL-based mesh generator), as well as on a simpler example which implements qsort.

Comments at the Conceptual Level: I feel like there is definitely a need for a debugger for optimized code, and while I have not experienced alternative approaches, I felt the optimized source code paradigm worked well for me. It provided an intuitive translation from my original source code to the optimized code shown in the opposite view. It was pretty straight forward to map code fragments in one section to another, both by observing which entities were highlighted, and by actually stepping through and watching the flow. This mapping was facilitated by the annotations inserted as comments by the debugger.

Functionality: By stepping through the code and looked at the resulting source code for several breakpoints, I was able to get a feel for using the debugger. As I did not actually have a problem that I was tracking down, I was not able to completely test the functionality, but I found nothing in principle that would prevent it from being very useful for debugging.

For my more complicated program, the code translation was incorrect in some cases. This was in some part due to coding style (e.g. my code violated the 'one statement per line' rule), and in part to apparent bugs in the optimized source generation. Assuming that the incorrect source code mapping was just an artifact of a prototype system, and not a problem inherent in the approach, I would find such a system very useful.

Suggestions for extensions:
I personally often use the ‘assign’ functionality of dbx while I am debugging, so I think it would be useful to include this feature in the debugger for optimized code.

Source code is only generated from a subset of the optimizations. I agree that it probably would be too difficult to provide a clear mapping from the original source to the completely optimized version, without going all the way down to the assembly level instructions. But I feel it might end up being misleading because you are still not really seeing the resulting code that is being executed. I don’t know if it is feasible to implement, but a multi-level approach might help when one is faced with a truly nasty bug. The first level is the source, the second the optimized source, and then varying number of levels below that that implement the other optimizations - all the way down to the assembly level. These levels would be hidden unless explicitly expanded by the user. I think this was mentioned already as a future direction for this work.

Evaluation of Caroline Tice’s Optimized Code Debugger

I volunteered to try out and evaluate the Optimized Code Debugger tool designed and developed by Caroline Tice. I spend almost four hours trying out the tool on five different C programs - some small, some large. I spend an additional hour with Caroline asking questions about the tool, her approach to the problem and her strategies for dealing with other issues.

The Concept

After discussions with Caroline, working with the tool and thinking about the issues, I believe that her concept of how to approach the problem of debugging optimized code has great merit. I have used other systems which tried to support such debugging but all fell short of the mark by a long way for various reasons.

Caroline's strategy fully supports several areas which I feel are absolutely critical:

1. Debug ''real'' optimized code. Other approaches which either instrument the optimized code or turn off some of the optimizations defeat the major reason for debugging optimized code - you want to save on the development time by only have to test & debug the final production code. Caroline's tool is directly debugging the fully optimized code with nothing addition added to the code path. It would be the final production code which ships. The support for her optimized code debugging is in instrumentation she added within the compiler, in the extended object symbol table, in the rewritten dwarf line number table and in the debugger tool itself.

2. The ''environment'' of debugging needs to be as familiar and comfortable to the user as debugging unoptimized code. If at all possible the tool needs to be the same one the user already uses without requiring the use of new commands and procedures. Caroline uses the standard tool - dbx - with a graphical front-end for synchronizing the source view with the debugging session.

3. The view of the source code during debugging need to be as close to the original code the user wrote as possible. Most other optimized code debuggers fail here because they reconstitute a source representation of the program which looks nothing like the user's original code (white space, comments, variable naming, etc). This is what I feel is the key contribution of Caroline's work. Her approach is to instrument the compiler to track transformations (optimizations) which have a reasonable source representation and then to apply those transformations independently to the original, textual source code. This completely preserves the ''look and feel'' of the original source. While not exactly the code the code the user wrote, they have all of their familiar ''landmarks'' in the code so they can immediately recognize and feel comfortable with it. Caroline purposefully does not try to represent every optimization (local scheduling for example) but those which affect the structural arrangement of statements.

4. The approach used must scale well to large applications. I've used other optimized code debuggers which work well for small programs but fail with anything real. Caroline's approach does

nothing extra to support debugging large programs but doesn't do anything to restrict it either. However well dbx does with large programs, this optimized code debugger will do. I see nothing in her approach that won't scale.

5. It must be simple to generate the debuggable, optimized program. Caroline's tool only requires that you use her instrumented compiler and assert the standard '-g3' option in addition to '-O2'. You also invoke the debugger with a different name. Other than that, nothing else is required.

Caroline's prototype tool only supports the C language and I explored with her what her strategy would be for other languages. In particular, Fortran 90 and C++ have language constructs which get broken down in compilation and distorted by optimizations in such a way that the final result can't be represented by legal source code in the original language. Her plan in this area would be to represent the transformed source in a pseudo-source form where needed. Since her whole scheme does not ever try to compile the transformed source, this will work technically. A question would be how understandable an applications programmer would find the pseudo-code. But I think that this is a reasonable approach to the problem.

The Users and the Tool

I gave some thought to who the user of such a tool would be and what would their expectations and experiences would be with debugging optimized code. [personal data omitted]

For the most part, I don't think that the potential user of this tool will be a compiler developer. Aside from coding mistakes, most problems in compilers are algorithmic based and are usually ''debugged'' with other tools and traces. There is a need to ''debug'' mis-compiled application program but the problems here are often at a finer granularity of optimization (source transformation) than this tool can show (by design).

I think that the real user body for this tool are application program developers. Here the ability to ''painlessly'' debug the final, fully optimized application could save a lot of time (by eliminating one full test/debug cycle). Assuming that one isn't dealing with a compiler-induced bug, the developer only needs to be concerned with the level of source transformations that this tool does support.

Again, I feel that any developer would feel ‘‘right at home’’ with the transformed source code presented during debugging.

The debugger front-end that Caroline has prototyped presents three panels to the user. The original source code, the transformed source code and the debugger session. As debugging proceeds, highlighting in both source windows tracks the execution. I found the feature of clicking on an original source statement to see what it was transformed into (and the opposite - querying where a transformed line came from) invaluable to navigating the program being debugged. Again, I'm used to and comfortable seeing optimization transformations. However; I feel that even someone who doesn't know what forward constant propagation or common sub-expression elimination is can easily follow, understand and feel comfortable with debugging their program.

My Use of the Tool

I spent most of my evaluation time exercising the tool itself. In general I was very impressed but hit little problems in almost everything I tried. Caroline has put a lot of thought, time and effort into this project and it shows but the prototype tool is just that - a prototype.

The feature I found most useful, was the dual-window presentation of original source and transformed source. Without it, I think a user would be lost. I liked the ability to click on a source line and (usually) see what line(s) it was transformed into. One suggestion for improvement is to vertically center the selected text and to try and keep the two representations (horizontally) in sync with each other. At times I found the comments classifying the optimization performed on the transformed source distracting. An option to turn them off would be helpful. The ability to resize the overall window and sub-windows is also needed.

Occasionally I had the problem that Caroline warned me about where the debugger would fail to come up. Trying the command again would work. In addition to the warning about multiple statements on a line not being supported (an implementation detail, not a problem with the concept or approach) I sometimes had problems with a single statement spanning multiple lines confusing the line tracker.

In general, the dbx dialogue window worked well but was often sluggish. A few times the tool, dbx or the application itself hung up and I had to abort the whole thing to regain control.

The instrumentation added within the compiler to track transformations did a good job with assigning names to compiler-generated temporaries and adding those names to the dwarf symbol table (allowing me to 'print' their value). It would be more useful if the generated name could be derived from a user given variable name when possible. For CSE'd expressions, this wouldn't work but when a temporary variable is assigned a simple variable, it's address or variable +/- constant (relatively common ``optimizations'') it could be given a name derived form the variable involved.

Most of the code I tried operated correctly so I just explored running it in the debugger, setting breakpoints (sometimes clicking on a transformed statement - one of many generated from a single statement- I'd get multiple breakpoints unexpectedly), printing variables, etc. For the most part this all worked ok and I had no problem navigating the optimized source code.

At one point I modified a source field, recompiled, re-entered the debugger and was presented with the new (original) source code and the old transformed source code. Somehow the new transformed code didn't appear. Deleting all generated files and recompiling cleared up the problem. The tool definitely needs to check date/time stamps on files and at least warn about out-of-date files.

I purposely introduced an error into one application (the weblint tool) and then did a real debugging session to find the bug. I was able to do it as easily (modulo a few tool hiccups) with the optimized code as I could have with the unoptimized code.

The tool isn't ready for ``prime time'' but it met the objective of allowing me to debug fully optimized programs as easily as I could the unoptimized versions.

Summary

I think that Caroline has done a great job with this project. She has pioneered a new approach to debugging optimized code (mimicked source transformation) which meets all the important criteria and is truly useful. It doesn't suffer the inherent problems of previous approaches. I also think it is an approach which could be adapted to any compiler since most of it is done externally and at the source level - the intrusion in the compiler is rather small and isolated. Caroline's ideas for solving the ''optimization transformation can't be represented in source'' problem of higher-level languages is sound and will probably work.

I think that the only weak part of the project is the implementation of the prototype tool itself. It is an excellent ''proof of concept'' which works well enough to explore the concept and see it's viability. However; I found that it wasn't robust in several areas. There are also a few usability issues which could be done better. I don't believe there are any major flaws in concept or design, but clean up and polish are needed.

Overall an excellent project and one which (successfully) advances the ability of debug optimized code.

Appendix B

SPECint95 Data for SGI Mips Pro 7.2 C Compiler

The following information was obtained directly from the official SPEC web site: http://www.spec.org/osg/cpu95/results/res98q3/cpu95-980701-02839.asc

SPEC Benchmark CINT95 Summary

Benchmarks	Base Ref Time	Base Run Time	Base Ratio	Peak Ref Time	Peak Run Time	Peak Ratio
099.go	4600	359	12.8	4600	331	13.9
124.m88ksim	1900	168	11.3	1900	131	14.5
126.gcc	1700	165	10.3	1700	159	10.7
129.compress	1800	149	12.1	1800	151	12.0
130.li	1900	161	11.8	1900	160	11.9
132.ijpeg	2400	215	11.2	2400	208	11.5
134.perl	1900	121	15.7	1900	121	15.7
147.vortex	2700	276	9.80	2700	277	9.74
SPECint_base95 (Geom. Mean)			11.8			
SPECint95 (Geom. Mean)						12.4

TESTER INFORMATION

SPEC License #: 04

```
          Tested By: SGI
          Test Date: Jun-98
     Hardware Avail: Jun-98
     Software Avail: Jun-98


                              HARDWARE
                              --------
         Model Name: O2: 250MHz R10k
                CPU: 250MHz MIPS R10000 Processor Chip Revision: 3.4
                FPU: MIPS R10010 Floating Point Chip Revision: 0.0
   Number of CPU(s): 1
      Primary Cache: 32KBI + 32KBD on chip
    Secondary Cache: 1MB (I+D)
        Other Cache: None
             Memory: 128MB
     Disk Subsystem: 1 SCSI 4.3GB
     Other Hardware: none


                              SOFTWARE
                              --------
   Operating System: IRIX 6.5
           Compiler: MIPSpro C Compiler 7.2.1
                     IRIX patches 2641 2909 2991 2992 3022 3048
                     IRIX patches 3065 3077 3131 3139 3140
        File System: XFS
       System State: Single-User


                               NOTES
                              -------
Portability Flags: all: -DUSG  gcc: -Dalloca=__builtin_alloca
FEEDBACK:
.   PASS1 = -fb_create /tmp/FBDIR/$(EXENAME)
.   PASS2 = -fb_opt /tmp/FBDIR/$(EXENAME)
Base Flags: -Ofast=ip32_10k -IPA:use_intrinsic, FEEDBACK
Peak Flags:
go:       -O2 -n32 -mips4 -IPA:plimit=1000:small_pu=60:use_intrinsic \
.         -TARG:platform=ip32_10k -OPT:Olimit=0 -OPT:goto=off, FEEDBACK
m88ksim:  -O2 -n32 -mips4 -TARG:platform=ip32_10k \
          -IPA:plimit=4000:space=100:small_pu=90:use_intrinsic \
.         -OPT:Olimit=0:fast_bit_intrinsics=on:ro=3:goto=off, FEEDBACK
gcc:   -Ofast=ip32_10k \
```

```
       -IPA:use_intrinsic:callee_limit=1500:plimit=750:space=120:clone=on\
       -OPT:fast_bit_intrinsics=on \
.      -LNO:fission=1:pwr2=off:prefetch=0 -OPT:ieee_arith=3, FEEDBACK
compress: -Ofast=ip32_10k -IPA:use_intrinsic, FEEDBACK
li:       -Ofast=ip32_10k \
          -IPA:use_intrinsic:min_hot=12:callee_limit=750:plimit=12000 \
.         -IPA:space=200:clone=on:aggr_cprop=on -OPT:goto=off, FEEDBACK
ijpeg:  -Ofast=ip32_10k -LNO:fission=0:interchange=off:pf2=0:ou_max=2 \
        -OPT:unroll_analysis=off:unroll_size=480:ieee_arith=3 \
        -IPA:use_intrinsic:callee_limit=950:plimit=730:space=130:clone=on\
.       -OPT:goto=off, FEEDBACK
perl:     -Ofast=ip32_10k -IPA:use_intrinsic, FEEDBACK
vortex:   -Ofast=ip32_10k -IPA:use_intrinsic, FEEDBACK
```

Bibliography

[1] Adl-Tabatabai, A. and Gross, T., "Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging", *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 371-383

[2] Adl-Tabatabai, A., "Source-Level Debugging of Globally Optimized Code", Ph.D. Dissertation, Carnegie-Mellon University, CMU-CS-96-133, May 1996

[3] Adl-Tabatabai, A. and Gross, T., "Source-Level Debugging of Scalar Optimized Code", *Proceedings of the ACM SIGPLAN '96 Conference of Programming Language Design and Implementation*, ACM SIGPLAN Notices 31(5), May 1996, pp.33-43

[4] Arsac, J., "Syntactic Source to Source Transforms and Program Manipulation", *Communications of the ACM*, Vol. 22, No. 1, January 1979, pp. 43-54

[5] Austin, T., Breach, S., and Sohi, G., "Efficient Detection of All Pointer and Array Access Errors", *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 29(6), June 1994, pp. 290-301

[6] Backus, J., "The History of FORTRAN I, II, and III" *ACM SIGPLAN History of Programming Languages Conference*, 1978 ADM SIGPLAN Notices 13(8), August 1978

[7] Boyd, M. and D. B. Whalley, D., "Isolation and analysis of optimization errors", *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 28(6), June 1993, pp. 26-35

[8] P. Breuer and J. Bowen, "Decompilation: The Enumeration of Types and Grammars", in *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, Sept. 1994.

[9] Brooks, G., Hansen, G.and Simmons, S., "A New Approach to Debugging Optimized Code", *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA USA, ACM SIGPLAN Notices 27(7), July 1992, pp. 1-11

[10] C. Cifuentes and K. Gough, "Decompilation of Binary Programs", *Technical Report FIT-TR-94-03*, Faculty of Information Technology, Queensland University of Technology, Australia, April 1994.

[11] Cifuentes, C., and Gough, K., "A Methodology for Decompilation", in *Proceedings for the XIX Conferencia Latinoamericana de Informatica*, Buenos Aires, August 1993, pp. 257-266

[12] Cool, L. "Debugging VLIW Code After Instruction Scheduling", M.S. Thesis, Oregon Graduate Institute of Science and Technology, July 1992

[13] Copperman, M. "Debugging Optimized Code Without Being Misled", *ACM Transactionson Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 387-427

[14] Copperman, M. and McDowell, C., "Technical Correspondence: A Further Note on Hennessy's "Symbolic Debugging of Optimized Code" ", *Transactions on Programming Languages and Systems* 15(2), April 1993, pp. 357-365

[15] Copperman, M., "Debugging Optimized Code Without Being Misled", Ph.D. Dissertation, UCSC-CRL-93-21, University of California, Santa Cruz, June 1993

[16] Coutant, D., Meloy, S., and Ruscetta, M., "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*, Atlanta, GA USA, ACM SIGPLAN Notices 23(7), July 1988, pp. 125-134

[17] Dhamdere, D., and Sankaranarayanan, K., "Dynamic Currency Determination in Optimized Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 6, November 1988, pp. 1111-1130

[18] Evans, D., "Static Detection of Dynamic Memory Errors", *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 31(5), May 1996, pp. 44-53

[19] Faith, R., "Debugging Programs After Structure-Changing Transformation", Ph.D. Dissertation, University of North Carolina, Chapel Hill, 1997

[20] Hastings, R., and Joyce, B., "Purify: Fast Detection of Memory Leaks and Access Errors", *Proceedings of USENIX Winter 1992 Technical Conference*, January 1992, pp. 125-136

[21] Hennessy, J., "Symbolic Debugging of Optimized Code", *Transactions on Programming Languages and Systems* 4(3), July 1982, pp. 323-344

[22] Hölzle, U., Chambers, C. and Ungar, D., "Debugging Optimized Code with Dynamic Deoptimization", *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA USA, ACM SIGPLAN Notices 27(7), July 1992, pp. 32-43

[23] D. Jackson and E. Rollins, "A New Model of Program Dependences for Reverse Engineering", in *Proceedings of the 1994 ACM SIGSOFT Conference*, December 1994.

[24] Jaramillo, C., Gupta, R., and Soffa, M.L., "Debugging Optimized Code Through Comparison Checking", *Technical Report*, TR-97-26, University of Pittsburgh, 1997

[25] Lientz, B.P., Swanson, E.B., and Tompkins, G.E., "Characteristics of Application Software Maintenance", *Communications of the ACM*, Vol. 21, No. 6, June 1978, pp. 466-471

[26] Loveman, D., "Program Improvement by Source-to-Source Transformation", *Journal of the Association for Computing Machinery*, Vol. 24, No. 1, January 1977, pp.121-145

[27] Lutz, Mark, "Programming Python", O'Reilly & Associates, Inc., Sebastopol, CA 1996

[28] Ottenstein, K. and Ottenstein, L., "High-level Debugging Assistance via Optimizing Compiler Technology", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, ACM SIGPLAN Notices 18(8), August 1983, pp. 152-154

[29] Pollock, L., Bivens, M., and Soffa, M., "Debugging Optimized Code via Tailoring", *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994

[30] Pollock, L. and Soffa, M., "Incremental Global Reoptimization of Programs", *Transactions on Programming Languages and Systems*, 14(2), April 1992, pp. 173-200

[31] Shu, W., "Adapting a debugger for optimised programs", *ACM SIGPLAN Notices* 28(4), April 1993, pp. 39-44

[32] Silverstein, J., ed., "DWARF Debugging Information Format", Proposed Standard, UNIX International Programming Languages Special Interest Group, July 1993

[33] Sites, R., Chernoff, A., Kirk, M., Marks, M., and Robinson, S., "Binary Translation", *Communications of the ACM*, Vol. 36, No. 2, February 1993

[34] Stallman, R., and Pesch, R., "Debugging with GDB" Edition 4.09 for GDB Version 4.9, Free Software Foundation, August 1993

[35] Streepy, L., Brooks, G., Buyse, R., Chiarelli, M., Garzone, M., Hansen, G., Lingle, D., Simmons, S., and Woods, J., "CXdb A New View on Optimization", *Proceedings of Supercomputer Debugging Workshop*, November 1991, pp. 1-22

[36] Tice, C., and Graham, S., "OPTVIEW: A New Approach to Examining Optimized Code", *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, ACM SIGPLAN Notices 33(7), July 1998, pp. 19-26

[37] Tolmach, A. and Appel, A, "Debugging Standard ML Without Reverse Engineering", *1990 ACM Conference on Lisp and Functional Programming*, June 1990, pp. 1 - 12

[38] A. van Deursen, P. Klint, F. Tip, "Origin Tracking", *Journal of Symbolic Computation 15*, 1993

[39] Wall, D., Srivastava, A., and Templin, F., "Technical Correspondence: A Note on Hennessy's "Symbolic Debugging of Optimized Code" ", *Transactions on Programming Languages and Systems* 7(1), January 1985, pp. 176-181

[40] Welch, B., "Practical Programming in Tcl and Tk", Second Edition, Prentice Hall PTR, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997

[41] D. Whitfield, M. L. Soffa, "An Approach for Exploring Code-Improving Transformations", *ACM Transactions on Programming Languages and Systems*, Volume 19, No. 6, November 1997

[42] Wismüller, R., "Debugging of Globally Optimized Programs Using Data Flow Analysis", *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 29(6), June 1994, pp. 278-289

[43] Wu, L. and Hwu, W., "A Novel Breakpoint Implementation Scheme for Debugging Optimized Code", *IMPACT Technical Report*, IMPACT-98-01, University of Illinois, Urbana-Champaign, 1998

[44] Wu, L., Mirani, R., Patil, H., Olsen, B., and Hwu, W., "A New Framework for Debugging Globally Optimized Code", in *Proceedings of the ACM SIPLAN '99 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 34(5), May 1999, pp. 181-191

[45] Zelkowitz, M.V., Shaw, A.C., and Gannon, J.D., "Principles of Software Engineering and Design", Prentice-Hall, Englewood Cliffs, NJ, 1979

[46] Zellweger, P., "An Interactive High-Level Debugger for Control Flow Optimized Programs", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, ACM SIGPLAN Notices 18(8), August 1983, pp. 159–171

[47] Zellweger, P., "Interactive Source-Level Debugging of Optimized Programs", Ph.D. Dissertation, University of California, Berkeley, Xerox PARC TR CSL-84-5, May 1984