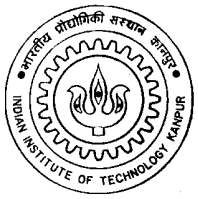
Debugging Optimized Code: Value Change Problem

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

Master of Technology

by

M Sarath Kumar



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*Debugging Optimized Code: Value Change Problem*", by *M Sarath Kumar*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

April, 2001

(Dr. Sanjeev Kumar Aggarwal)
Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Compiler optimizations are an integral part of all the modern compilers. Current compiler optimizations show a considerable improvement in space and time requirements of a program. Optimizations reorder the execution of the statements and affects the ability to debug optimized code.

Source-level debuggers allow the value of a variable to be changed at debug time. Optimizations affect the ability of the debugger to change the value of a variable at debug time. We studied how optimizations affect the ability of the debugger to change the value of a variable at debug time. We proposed and implemented a framework for overcoming the perturbations caused by optimizations, for changing the value of a variable at debug time.

Acknowledgements

I would like to express my sincere thanks to my guide Dr Sanjeev Kumar Aggarwal for his constant support and guidance throughout this work. I am very grateful for his interest and suggestions during the work.

I would like to thank all the faculty of Computer science department for all the knowledge and learning experiences that they have imparted to me. I would like to thank the self-styled compiler group involving Prasanth, Krishna Reddy, Kingshuk Karuri, Raghavendra Maloo and me for the rewarding discussions that we enjoyed. I thank Vamsi, Sreekanth, Reddy, Kiran, Prasanth, Praveen, Nilabh and Masa, for the long useful and useless discussions that we had. I thank all the Mtech99 batch for the good days that we had in IITK.

I thank my parents for their constant support and patience. I thank all the IITK community for making my stay memorable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Compiler optimizations are an integral part of all the modern compilers. Current trends in processor design are towards an increasing reliance on compiler transformations to achieve high performance. In modern processors many resources like large register files, instruction level parallelism and memory hierarchy are exposed to the compiler. Compiler uses all these resources optimally to realize a higher performance of the processor. Current compiler optimizations show considerable improvement in space and time requirements of a program, forcing the software product developers to release their product after performing optimizations.

However, optimizations reorder, eliminate and duplicate code. Therefore, source level debuggers working with optimized code cannot mimic behaviour of the original unoptimized program. The perturbations caused by the optimizations prevent most source-level debuggers from providing support for debugging optimized code. The traditional ways of designing production code is to first test and debug the unoptimized code and then optimize the code to get the performance enhancements and release the product.

1.1 Need for debugging optimized code

There are many situations where debugging optimized code is necessary and desirable.

1. A program may run correctly when compiled without optimizations but may fail when compiled enabling optimizations. This can occur even though the optimizer is correct. Compiler optimizations are correctness preserving transformations for a correct program. Correctness preserving transformations are not guaranteed to preserve the behavior of a program that behaves differently for different executions. For example, programs using values of uninitialized variables behave differently for different executions.

   Examples:

   (a) Reordering of the statements may cause underflow or overflow.

   (b) Exceptions may occur because the value of the uninitialized variable changes when we change the data layout of the program. An array out of bound exception might occur due to change in the data layout of the program caused by optimizations.

2. Final production code is generally an optimized version. One should be able to debug this optimized code by looking at the core file and a bug report.

3. Programs with timing and instability problems behave differently when compiled with optimizations and without optimizations. In such cases we should have the ability to debug optimized code.

4. There may be constraints on the space and time requirements of a source program. This situation mainly occurs in designing the embedded software systems. The optimized version of a program may fit into the requirements, when we need to debug such a program we should have the ability to debug optimized code.

5. The compiler may not be able to generate unoptimized code.

6. Optimizing compiler may have bugs.

1.2 Existing source-level debuggers and their support for debugging optimized code

Most compiler systems provide a source-level debugger. In this section we look at the existing source-level debuggers and their support for debugging optimized code.

1. **Gdb:** Gdb [15] is the source-level debugger that comes with the GNU compiler systems. Gdb has support for debugging programs written in C and C++. Gdb has partial support for debugging programs written in FORTRAN, Modula-2 and CHILL. It does not have special support for debugging optimized code. When gdb debugs an optimized program, it just shows what is there. It gives surprising results to the user. It does not map the breakpoints in the source code to the correct(user expected) places in the optimized object code. The user may see that the execution path that is followed is not the expected execution path. At a breakpoint the user may also get the values of the variables different from what one expects. This debugging of optimized code is not helpful to a novice user rather, it may lead to wrong conclusions about the program.

2. **Ladebug:** Ladebug [3] is a source-level debugger that comes with the Compaq compiler systems. Ladebug has support for debugging programs written in C, C++, FORTRAN-77 and FORTRAN-90 on Compaq's $TRU64^{TM}$ UNIX and Linux for Compaq $ALPHA^{TM}$ systems. This debugger has support for debugging optimized code. When the user wants to know the value of a variable at a breakpoint this debugger checks if the variable is eliminated due to optimizations and prints the appropriate message. If the variable has a runtime value then it displays the value of the variable and also prints the information saying whether it is the expected value or not. This debugger correctly maps the breakpoints in the optimized object code. This debugger does not have support for changing the value of a variable when debugging optimized code.

3. **Wdb:** HP Wdb 2.1 [8] is an HP supported implementation of gdb 5.0. It supports debugging of programs written in C, C++, FORTRAN-77 and FORTRAN-90 on HP-UX release 10.20 and later. Support for debugging optimized code in Wdb is same as that in Gdb mentioned earlier.

4. **SUN's** *Forte*^{TM}**:** Forte [17] is a set of tools that can be used to develop applications on the *Solaris*^{TM} operating environment. Forte C and Forte C++ are some of the components of Forte set of tools. Forte C and Forte C++ contain a debugger that has support for debugging optimized code.

5. **Microsoft's Visual C++:** Microsoft Visual C++ [10] contains an integrated source-level debugger for debugging programs written in C and C++. This debugger does not have any support for debugging optimized code at source level. It does not allow optimizations and source-level debugging simultaneously. If the user wants to debug optimized the code then he has to perform debugging at the assembly language level. Debugging program at assembly language level is very tedious and is less useful for novice programmers.

1.3 Previous work

Hennessey[7] addresses the problems in debugging of optimized code. He introduces the terms nonresident and noncurrent variables. At a breakpoint, a variable is said to be nonresident if there is no runtime location corresponding to the variable. At a breakpoint, a variable is said to be noncurrent if there is a runtime location corresponding to the variable, but the value of the variable is not the expected value of the variable. He addresses how to detect the variables that are nonresident and noncurrent at a breakpoint when local optimizations are performed. Hennessey does not address how to detect nonresident and noncurrent variables when global optimizations are performed and how to correctly map the breakpoints.

Zellweger [18] addresses the problem of correctly mapping the breakpoints set in the source code to the corresponding places in optimized object code. DOC [5] is a first source level debugger, which supports debugging optimized code. DOC shows

that debugging optimized code is viable. DOC performs three optimizations: register promotion and assignment, loop variable elimination and instruction scheduling.

Brooks, Hansen and Simmons[6] show that when debugging optimized code providing the same behavior as an unoptimized program is not feasible and is impractical.

Copperman [4] investigates the problem of detecting endangered variables caused by global optimizations. Ali Reza [13] [14] [12] describes how to find endangered and noncurrent variables and ways to recover the expected values of the variables. He also addresses the code location problem in detail. Le-Chen Wu, Rajiv Mirani etal[9] describe a new framework for debugging optimized code.

1.4 Scope of our work

The problem of how optimizations affect the ability of the debugger to change the value of a variable at debug time has not been addressed. All the available source-level debuggers that support debugging of optimized code do not allow the user to change the value of a variable at debug time when debugging optimized code. In this thesis we address how optimizations affect this functionality of the source-level debuggers. We also address how to overcome the perturbations caused by optimizations and change the value of variables at debug time.

1.5 Organization of the report

The rest of the thesis report is organized as follows.

In **Chapter 2** we explain in more detail debugging of optimized code. We present model of the debugger and the existing approaches for debugging optimized code.

In **Chapter 3** we present the data value problem, how we can solve it. We describe the additional information that is to be passed by the compiler to the debugger.

In **Chapter 4** we present the code location problem, how we can solve it. We

describe the additional information that is to be passed by the compiler to the debugger.

In **Chapter 5** we discuss the value change problem. We look at the advantage of providing the user with the ability to change the value of a variable at debug time. We discuss perturbations caused by optimizations to value change problem. We present methods to overcome the perturbations caused by optimizations and change the value of the variable at debug time.

In **Chapter 6** we present the experimental framework.

In **Chapter 7** we present conclusion and scope for future work.

Chapter 2

Debugging optimized code

In this chapter we present the model of the debugger. We explain with examples how optimizations affect source-level debugging of optimized code. We also present different approaches to debug optimized code.

2.1 Model of the debugger

The main aim of a source-level debugger is to provide a source-level view of the execution of a program. The source-level debuggers provide interactive debugging and post-mortem debugging. In interactive debugging the user specifies some events, the debugger returns control to the debugger when any such event occurs and the user can observe the state of the program. These events can be control breakpoints, conditional breakpoints, watchpoints or data breakpoints. In post-mortem debugging the debugger is invoked after a program is terminated. The user can find the source statement that caused the program to terminate and can observe the state of the program, however execution of the terminated program cannot be resumed.

Source-level debuggers provide much functionality. The debugger model that we consider through out this thesis work provides a core set of functionalities. Other auxiliary functionalities can be provided using this core set of functions. The core set of functionalities provided by our debugger are:

- User can set breakpoints at any source statement in the given program. The

user can also set breakpoints on functions, the breakpoints set on the functions are triggered when a call to that function occurs and the control is transferred to that function. After a breakpoint has occurred the user can resume the execution of the program. The user can set and reset breakpoints.

- At a breakpoint, the debugger can be used to display the value of a variable in the program.
- At a breakpoint, the debugger can change the value of a variable. This functionality is not provided by most of the debuggers that work with optimized code.
- After a breakpoint has occurred, the user can step through the statements of the programs iteratively.
- The debugger has support for post-mortem debugging.

Debugger has to modify the object code to set up control and data breakpoints. There are two types of modifications that a debugger performs on a program's address space: changes of data layout and changes of code segment. Changes in data layout are needed to set data breakpoints. Changes to data layout may mask some error or may cause some other error therefore; it is not advisable to change the data layout of a program that is being debugged. Changes in the code segment are needed to implement control breakpoints. Reza [12] classifies the debuggers into non-invasive debuggers and invasive debuggers, based on the property of whether the debugger changes the data layout and program segment of optimized code or not. A debugger is called non-invasive if it does not require modifications to the data layout of a program and the changes in code segment are needed only to implement control breakpoints. Otherwise, the debugger is said to be invasive. we consider a non-invasive debugger model.

2.2 Perturbations caused by optimizations

The debugger maintains source to object and object to source mappings to set breakpoints and report asynchronous breakpoints (runtime exceptions, user interrupts). In source to object mapping, for each source statement the debugger stores the object code location(s) where the breakpoint can be mapped in the object code. Debugger uses source to object mapping to set control breakpoints. In object to source mapping, the debugger stores the source statement(s) corresponding to each object instruction. Debugger uses object to source mapping to report asynchronous breakpoints.

When debugging unoptimized code, source to object mapping and object to source mapping are same. For each source statement we have a fixed object code location where the breakpoint set on that source statement can be mapped. For each group of object statements we have a corresponding unique source statement. The object to source and the source to object mappings are both one-to-one mappings. Optimizations reorder, eliminate and duplicate source statements therefore, the source to object mapping and the object to source mappings are different. Both the mappings are no longer one-to-one mappings. For each source statement we may have more than one object code locations, in such a case the debugger has to decide to which object code location a breakpoint has to be mapped. The problem of correctly setting the breakpoints in optimized code is called code location problem.

In unoptimized code object instructions are executed in the order specified in the source program so, the variables have expected values. Optimizations eliminate and reorder the execution of the statements. The order in which the object instructions execute is different from the source execution order and this causes the variables to have values different from expected values. In unoptimized code each variable has a runtime location corresponding to that variable all along the scope of the variable. Due to optimizations a variable may not be allocated a runtime location or a runtime location allocated to a variable is reused to store the value of another variable. When the debugger wants to display the value of a variable, there is no runtime location for the variable or more than one location might contain the value of the variable and the debugger cannot decide which location contains the expected

value of the variable. These problems are called data value problems.

2.2.1 Example : Dead code elimination and loop invariant code motion

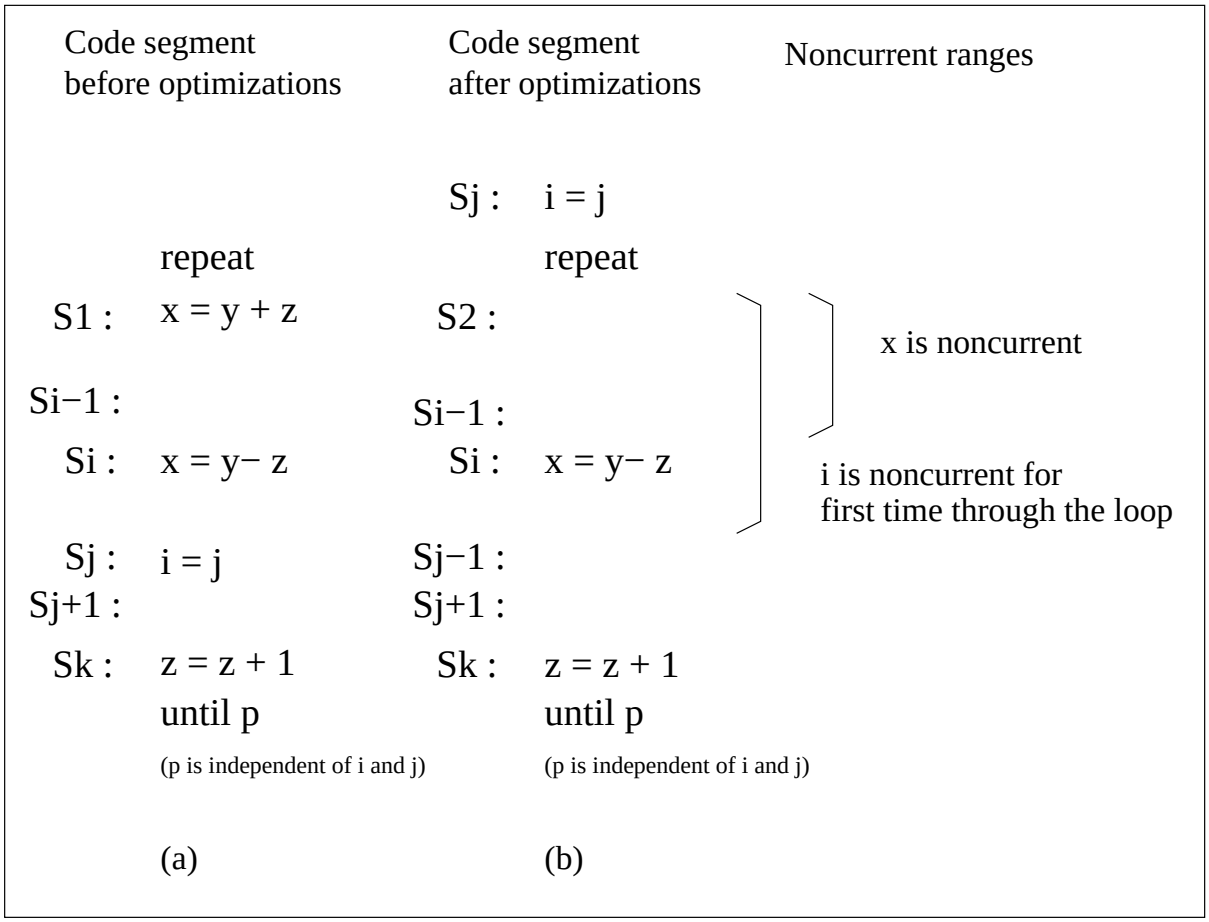


Figure 2.1: Sample code to demonstrate the perturbations caused by optimizations

In figure 2.1 we present a simple example to depict how optimizations affect debugging of optimized code. figure 2.1(b) is the resultant code after performing

Breakpoint in fig 2.1(a)	Breakpoint in fig 2.1(b)	Number of times breakpoint is triggered	Endangered Variables at the breakpoint
S_j	S_j	once	variables assigned in S_1 to S_{j-1}
S_j	S_{j-1}	n times	variable assigned in S_j
S_j	S_{j+1}	n times	None

Table 2.1: Table showing effect of breakpoint mapping on data value problems

dead code elimination and loop invariant code motion on the code in figure 2.1(a). The optimizations performed cause both code location problems and data value problems.

Code location problems

Statement S_j is moved out of the loop due to loop invariant code motion. When the user sets a breakpoint at S_j, the debugger should find the appropriate location in the object code to map this breakpoint. If the debugger sets the breakpoint at statement S_j in figure 2.1(b) then the breakpoint is triggered only once and all the variables assigned in statements S_1 to S_{j-1} are noncurrent. If the debugger sets the breakpoint at statement S_{j-1} in figure 2.1(b) then the breakpoint is triggered expected number of times but, the variable i is noncurrent at the breakpoint. If the breakpoint is set at statement S_{j+1} in figure 2.1(b) then the breakpoint is triggered the expected number of times and the variable i is current but, if the statement S_j is the last statement of the basic block then there is no statement after S_j. The effect is summarized in Table 2.1.

Data value problems

There are no uses of x in statements S_2 to S_{i-1} so the assignment to x in S_1 is dead and that statement is eliminated. As the statement S_1 is deleted, at all breakpoints set at statement S_2 to S_{i-1} the value of x is noncurrent. Statement S_j is moved out of the loop, this causes the value of i to be noncurrent from statements S_2 to S_{j-1} for the first time through the loop.

2.3 Alternative approaches for debugging optimized code

There are four main approaches for debugging optimized code. The first approach ignores the fact that optimizations have been performed resulting in an anomalous behaviour. The second approach tries to hide all optimizations performed and and provides the expected behaviour. The third approach exposes all the optimizations that are performed and it is left to the user to figure out how optimizations affect the program. The fourth approach provides truthful behaviour.

2.3.1 Ignoring optimizations

In this approach the debugger does not consider whether optimizations are performed or not. The debugger just shows the current execution state of the program. When the value of a variable is queried the debugger just displays the value present in the runtime location corresponding to the variable and this value may be different from the expected value of the variable. The execution path followed may be different from the expected execution path. These types of debuggers are easy to design and are less helpful when debugging optimized code.

There are some debuggers that allow debugging without considering whether optimizations are performed or not. Gdb follows this approach. This kind of debuggers show values different from the expected values of the variables and execution paths different from the expected paths are taken. This type of the debuggers are easy to design, but not of much use to a novice user when debugging optimized code.

2.3.2 Providing expected behaviour

This is the ideal approach for debugging optimized code. In this approach the debugger manages the effects of the optimizations and provides the illusion that one source statement is executed at a time and gives the expected values of the variables. The debugger executes the object statements in the expected source order to give expected values of the variables. As optimizations reorder, duplicate and eliminate

source statements it is not possible to execute the statements in the expected source order without making changes to the data layout and code segment of the program.

The debuggers built using this approach use several techniques to provide the expected results. They constrain the number of locations where the debugger can be invoked, they constrain some of the optimizations, and they add instrumentation code and disturb the data layout and code segment of the object code. These debuggers are invasive as they disturb the data layout and code segment of a program.

Le-Chen Wu and Rajiv Mirani etal [9] propose a new framework for debugging optimized code. In this scheme they present that with perturbations to data layout and code segment one can provide expected behaviour for all optimizations including the global optimizations. This scheme does not allow the value of the variable to be changed at debug time. when the user sets a breakpoint at a source statement, the debugger takes control of the program early before executing any of the statements that have to be executed after this statement as per the source execution order. After this it emulates only those instructions that have to be executed before the statement as per the source execution order and reports that breakpoint has occurred. As all the statements that have to be executed before the breakpoint have executed and none of the statements that have to be executed after it have executed, the state of the program is the expected state and thus provides the expected behaviour.

2.3.3 Exposing optimizations to the user

In this approach all the optimizations performed are exposed to the user in the source level terms. The debugger shows source-level view of the optimized program. This approach is especially useful when the debugger performs loop optimizations like loop peeling, loop interchanging and loop tiling. There are many disadvantages in this approach. The debugger user should have full knowledge of optimizations, which generally is not the case. There are some optimizations like instruction scheduling and register allocation whose effect is difficult to express in the source level terms.

2.3.4 Providing truthful behaviour

In this approach the debugger detects when optimizations do not affect debugging and hides the optimizations. When the debugger is unable to hide the optimizations it exposes the optimizations to the user. When optimizations are to be exposed to the user, this approach manages the effects of the optimizations by relating runtime values in the optimized code to source-level values. The burden of the analysis falls on the debugger and less knowledge of the optimizations is required on part of the debugger user. Hennessey [7], Zellweger[18], Coutant [5], Copperman[4], and Ali reza [12] use this approach.

Chapter 3

Data value problems

Data value problem refers to the problems arising in reporting the value of a source variables at a breakpoint. Optimizations change the order of execution of the statements. Changing the order of the execution of the source assignment statements causes the variables to have values different from the expected values.

When user asks the debugger to display the value of a variable the debugger reports the state of the variable as nonresident, noncurrent, current or suspect.

- At a breakpoint, a variable is said to be nonresident if there is no runtime location containing the value of the variable hence, no runtime value for the variable. For example in the figure 3.1, there are no uses of X after *E4* so same register is shared by both X and Y. The next definition of Y after *E4* assigns a new value of Y to the register shared by both X and Y hence, there is no runtime value for X and is nonresident at a breakpoint *Bkpt4* set after *E5*.

- At a breakpoint, a variable is said to be noncurrent if it is resident and the value of the variable is not the expected value. For example in figure 3.1 *E0* is dead code and is eliminated. The value of X present at *Bkpt1* is not the expected value and hence X is noncurrent.

- A variable is said to be current if the variable is resident and the value is same as the expected value. At breakpoint *Bkpt3* the value of X is the expected

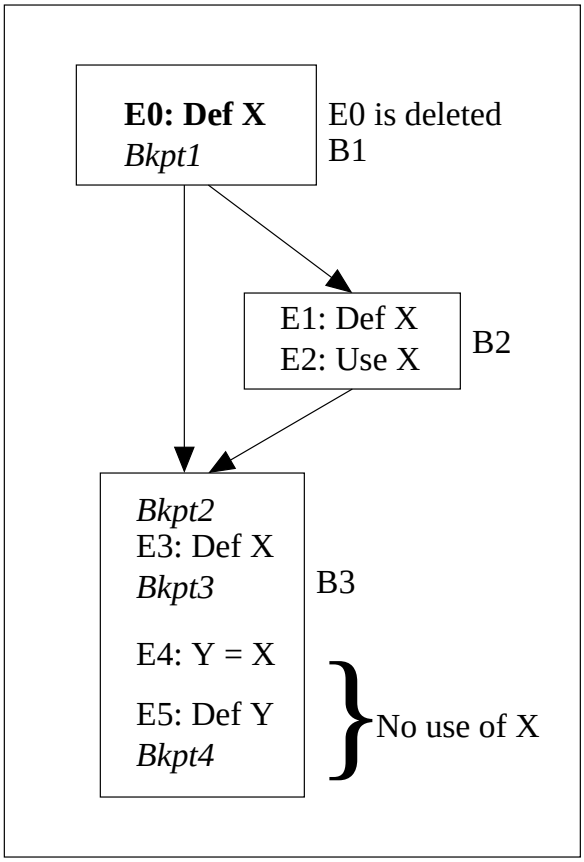


Figure 3.1: Sample code to demonstrate possible states for a variable

value hence, X is current.

- A variable is said to be suspect, if it is resident and the debugger cannot determine if it is current or noncurrent. In figure 3.1 the variable X is current at *Bkpt2* if the path taken is *B1*, *B2* and *B3* and X is noncurrent if path followed is *B1* and *B3*. Variables that are suspect and noncurrent are called endangered variables.

Data value problem is subdivided into the problem of finding nonresident variables and the problem of finding endangered variables at a breakpoint.

3.1 Detecting nonresident variables

A variable is said to be nonresident at a breakpoint if there is no runtime location corresponding to a variable at that breakpoint or, there are multiple runtime locations containing the value of the variable and the debugger is unable to decide which location contains the expected value of the variable. We use Ali Reza's approach [12] to detect nonresident variables. When no optimizations are performed all variables are assigned a unique memory location except the register variables that are stored in registers. The mapping from variables to runtime locations is one-to-one. Optimizations like register allocation and register coalescing disturb this one-to-one mapping and cause nonresident variables.

Register allocation tries to use registers to store the values of frequently accessed variables, constants and temporaries to reduce their access time. We consider only source variables, as only these variables are visible at the source level. The register allocator takes the live ranges of the variables and assigns registers to these live ranges using some heuristics. Some register allocation methods split the live range of the variables and assign each split range a register or a memory location independent of other live ranges of the same variable. Additional code is added to transfer runtime value from runtime location of one split range to the runtime location of another split range when control moves from one split range to another split range.

The variables that are not allocated a register are allocated a memory location on the activation stack. Sharing of the memory locations by non-overlapping live ranges reduces the size of the activation stack. Splitting the live ranges of a variable makes variable to runtime location mapping one-to-many. Assigning same memory location and same register to different non-overlapping live ranges makes variable to runtime location mapping many-to-one.

Register coalescing[11] eliminates move instructions from one variable to another by allocating same register to both these variables if the live ranges of these variables are non-overlapping. Register coalescing makes variable to runtime location mapping many-to-one and causes nonresident variables.

3.1.1 Detecting nonresident variables caused by register allocation and register coalescing

Using a simple and a conservative approach we say a variable is nonresident outside the live range of that variable. Looking at the live range information in the symbol table we can do this. This approach is simple but misses some opportunities. A storage location allocated to a variable holds the value of the variable all along the live range of the variable. Even after the live range of a variable a storage location allocated to a variable holds the value of the variable until a new variable is assigned to this location. A definition (assignment to a variable) is called an evicting definition if it assigns the value of the variable to a location that previously contained value of another variable.

S1 :	x = 2	s1 = 2	r1 = 2
S2 :	y = 4	s2 = 4	r2 = 4
S3 :	w = x * y	s3 = s1 * s2	r3 = r1 * r2
S4 :	z = x + 1	s4 = s1 + 1	r3 = r1 + 1
S5 :	u = x * 2	s5 = s1 * 2	r1 = r1 * 2
S6 :	x = z * 4	s1 = s2 * 4	r2 =r3 * 4
	(a)	(b)	(c)

(a) A simple example of register allocation; (b) symbolic register assignment for it; (c) an allocation for it with three registers assuming y and w are dead on exit from this code

Figure 3.2: Sample code to show evicting definitions

A location may contain the value of a variable even after the live range of that variable. Improved response can be given to the user if all evicting definitions are found. A variable V whose value is found in location L is resident even after the live range of V until an evicting definition assigns to location V. In the figure 3.2(c) register *r2* is used to store the value of y. The live range of y is from *S2* to *S4*.

The value of *r2* is changed by statement *S6*. The value of y is present in *r2* till *S6* which is outside the live range of y. The evicting definition is *S6*.

We say that a variable V is resident in location L at a breakpoint B if there exists an assignment to V along all paths from start of the program to B and the value of the variable is stored in L and there is no evicting definition for location L after last occurrence of assignment to V. We perform simple dataflow analysis to get the list of the variables and the corresponding runtime locations that reach a particular point in the program.

Dataflow Analysis

A residence which is a tuple $<V,L>$ indicates that a variable V is stored at location L. A basic block B generates a residence $<V,L>$ if there exists an assignment to V in the block B and there are no evicting definitions of L after last such occurrence of assignment to V. A basic block kills a residence $<V,L>$ if there is an evicting definition of location L.

Definition 3.1 (AvailResGen) *AvailResGen indicates the set of residences generated by a basic block.*

Definition 3.2 (AvailResKill) *AvailResKill indicates the set of residences killed by a basic block.*

Definition 3.3 (AvailResIn) *AvailResIn indicates the set of residences that reach a basic block.*

Definition 3.4 (AvailResOut) *AvailResOut indicates the set of residences are available after a basic block.*

Initially find the *AvailResGen* and *AvailResKill* sets for all the basic blocks. The sets *AvailResIn* and *AvailResOut* of all basic blocks is found by solving the dataflow equations 1 and 2.

$$AvailResIn(I) = \bigcap_{J \in Pred(I)} AvailResOut(J) \tag{1}$$

$$AvailResOut(I) = (AvailResIn(I) - AvailResKill(I)) \bigcup AvailResGen(I) \quad (2)$$

We use an iterative approach to solve the dataflow equations. The algorithm for finding the *AvailResIn* and *AvailResOut* sets of a basic block is given below: The algorithm takes *AvailResKill* and *AvailResGen* sets of each basic block as input and generates *AvailResIn* and *AvailResOut* sets of each basic block.

for each basic block B **do** *AvailResOut[B] = AvailResGen[B]*
change = **true**
while change **do begin**
    change = **false**
    for each basic block B do **begin**
        AvailResIn(B) = $\bigcap_{J \in Pred(B)}$ AvailResOut(J)
        oldout = *AvailResOut[B]*
        AvailResOut(B) = (AvailResIn(B) - AvailResKill(B))$\bigcup$AvailResGen(B)
        if *AvailResOut[B]* $\neq$ oldout **then** change = **true**
    end
end

3.2 Detecting endangered variables

Optimizations that affect assignments to source variables cause endangered variables. Optimizations delete and reorder assignment statements causing endangered variables. We use Ali Reza's approach[12] to detect endangered variables.

3.2.1 Optimizations that cause endangered variables

Optimizations that eliminate and move source assignment statements cause endangered variables.

Code elimination:

Optimizations such as dead code elimination delete assignments to source variables. Eliminating assignment statements causes endangered variables. The user

expects the variable to have the value that would have been assigned by the deleted assignment statement, whereas the actual value is different. Programmer does not generally add dead code, it results from performing some optimizations such as constant propagation, copy propagation and induction variable strength reduction etc.

Code motion:

Optimizations such as loop invariant code motion, code hoisting and instruction scheduling move statements from one location to another location. Moving the statements causes the actual execution order to be different from the expected execution order. Code motion optimizations move code either in the direction of control flow or in the direction opposite to control flow creating endangered variables.

Concentrating on two key transformations code hoisting and dead code elimination captures data value problems caused by all the global transformations.

3.2.2 Optimizations that do not cause endangered variables

There are some optimizations that do not cause endangered variables. Optimizations such as constant folding, induction variable strength reduction and copy propagation do not affect assignments to source variables. These optimizations create new opportunities for dead code elimination indirectly contributing to creation of endangered variables. Optimizations that affect assignments to compiler temporaries do not cause any endangered variables, as we are concerned with only source variables.

3.2.3 Endangered variables caused by instruction scheduling

Instruction scheduling reorders and interleaves the execution of source statements. Semantics of the source programming language defines an ordering on the execution of the statements, which is same as the expected execution order. Instruction scheduling moves assignment statements and causes endangered variables. Endangered variables occur due to either early execution or delayed execution of source assignment statements. At a breakpoint B if assignment to a variable V has been

executed early then the variable is said to be a roll back variable indicated by BackVar(V,B). At a breakpoint B if assignment to a source variable V is delayed to be executed after the breakpoint then V is called a roll forward variable indicated by ForwardVar(V,B).

Finding all source assignment statements that have executed out of order we detect endangered variables. The debugger maintains source execution order and object execution order to find the assignment statements that have executed out of order. Source execution order refers to the order in which the source assignment statements have to be executed as per the semantics of the source programming language. Each source assignment statement S is assigned a sequence number seq(S) to indicate the source execution order. For all source assignment statements that immediately follow a source assignment statement S_i the sequence number seq(S_i)+1 is assigned. For two statements S_1 and S_2, if seq(S_1) $<$ seq(S_2) then S_1 has to execute before S_2 and if seq(S_1) $>$ seq(S_2) then S_1 has to execute after S_2. If seq(S_1) $=$ seq(S_2) then there is no fixed ordering between S_1 and S_2.

Definition 3.5 (Source assignment descriptor) *Source assignment descriptor for an object instruction O is a tuple <I,N,V> where I is the object instruction, N is the sequence number of the source instruction for which O is generated and V is the variable that is assigned in O.*

Definition 3.6 (INST()) *For a source assignment descriptor D, INST(D) returns the object instruction in the descriptor.*

Definition 3.7 (seq()) *For a source assignment descriptor D, seq(D) returns the sequence number of the source instruction for which the object instruction of D is generated.*

Definition 3.8 (VAR()) *For a source assignment descriptor D, VAR(D) returns the variable that is assigned by the object instruction of D.*

Object execution order refers to the order in which the instructions generated for the source assignment statements execute. For each object instruction O that corresponds to a source assignment statement we add a source assignment descriptor.

A breakpoint B is represented by $<S,O>$ where S is the source statement where the breakpoint is set in the source and O is the object instruction where the breakpoint is mapped in the object code. We find all roll back and roll forward variables at a breakpoint and using this information we find if a variable is endangered at the breakpoint.

A variable is said to be a roll forward variable at a breakpoint B<S,O> if an assignment with assignment descriptor D, that has to be executed before the breakpoint as per the source execution order(seq(D) < seq(S)) is slated to be executed after the breakpoint(INST(D) >= O). Similarly, a variable is said to be a roll back variable or not. Equations 3 and 4 can be used to find the roll back and roll forward variables.

$$((seq(D) < seq(S))\&(INST(D) >= O)) => ForwawdVar(VAR(D), B) \quad (3)$$

$$((seq(D) > seq(S))\&(INST(D) < O)) => BackVar(VAR(D), B) \quad (4)$$

3.2.4 Endangered variables caused by local optimizations

Local optimizations cause endangered variables by eliminating assignment statements. Assignment statements can be eliminated if an assignment is locally available or an assignment is a dead assignment. An assignment to a variable V is called an available assignment statement if on all paths from the starting point of the program to this point, there is an assignment statement same as this and there are no further assignments to V and all other variables present in the assignment statement after last such occurrence of the assignment to V. Elimination of available assignment statements does not cause any endangered variables, as the deleted assignment statement assigns a value that is already available.

Elimination of dead assignments

An assignment statement S that assigns to a variable V is a dead assignment statement if there is no use of V until the next definition of V or the end of the program. Elimination of dead assignment statements causes endangered variables.

The variable is expected to have the value that would have been assigned by the deleted assignment statement. At all breakpoints set between the deleted statement and the next assignment to the variable V the value of V is different from the expected value.

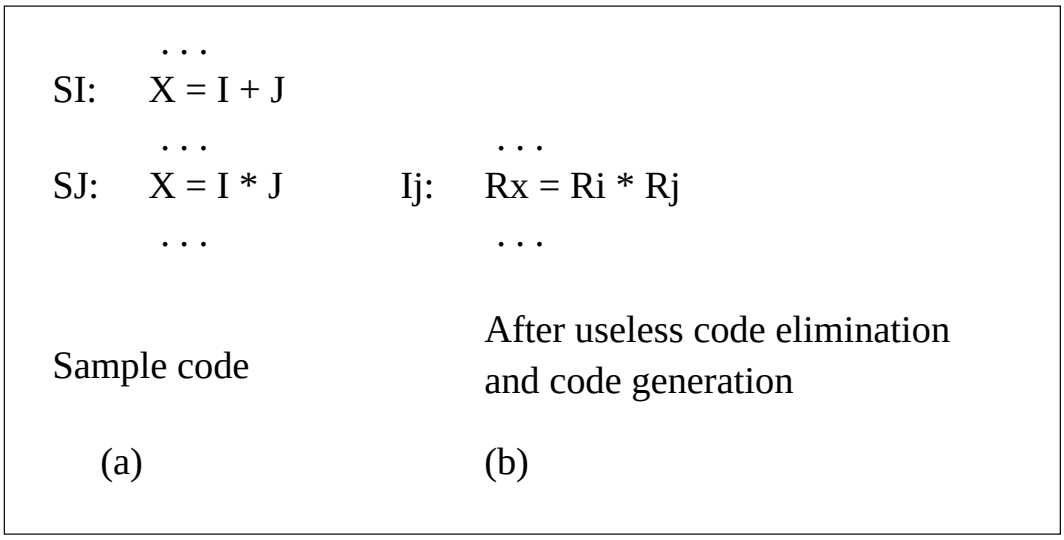


Figure 3.3: Sample code to demonstrate effects of dead code elimination

In the example given in figure 3.3 the assignment statement S_I is dead as there are no uses of x in statements S_{I+1} to S_{J-1}. Eliminating S_I makes x endangered at breakpoints set at statements S_{I+1} to S_{J-1}. For every dead assignment statement A that is deleted we add a dead assignment descriptor $<I,N,V>$ where I is the object instruction corresponding to the instruction A', which is the next assignment statement that assigns to V after A, N is the sequence number of the source instruction A and V is the variable assigned in A. At a breakpoint B $<S,O>$ a dead assignment descriptor D creates an endangered variable if these three conditions hold:

1. The source statement where the breakpoint is set comes after the dead assignment statement as per the source execution order.

2. The object instruction generated for the next source assignment that assigns to the same variable as dead assignment, comes after the object instruction where the breakpoint is mapped, and

3. The variable is not a roll forward variable at the breakpoint.

All the three conditions are checked in the following equation.

$((seq(D) < seq(S))\&(INST(D) \geq O)\& \sim ForwardVar(Var(D), B)) \Rightarrow EndangeredDead(Var(D), B)$

We find all variables endangered due to dead code elimination at a breakpoint and use that information to find if a variable is endangered due to local dead code elimination at this breakpoint.

3.2.5 Detecting endangered variables caused by global optimizations

Global optimizations like code hoisting, code sinking, loop invariant code motion, global dead code elimination and global instruction scheduling move and delete source assignment statements causing data value problems. Concentrating on two key transformations code hoisting and global dead code elimination captures data value problems caused by all the global optimizations.

Detecting endangered variables caused by code hoisting

Code hoisting is one form of partial redundancy elimination. Adding expressions at some places to make other partially redundant expressions fully redundant is called code hoisting[2]. Code hoisting moves expressions in the direction opposite to the control flow. Some of the assignment statements are executed earlier than expected causing endangered variables. Expressions that are added by code hoisting are called hoisted expressions. Expressions that become redundant after code hoisting and are deleted are called redundant expressions.

In the figure 3.4 we show with an example how code hoisting affects debugging. In the sample code shown in figure 3.4 expression $X = U$ - V is executed twice if the path followed is *B0*, *B1* and *B3* and the same expression is evaluated once when the path is *B0*, *B2* and *B3*. Expression *E2* is partially redundant. As part of code hoisting the expression $X = U$ - V is moved ahead into block *B2* making partially

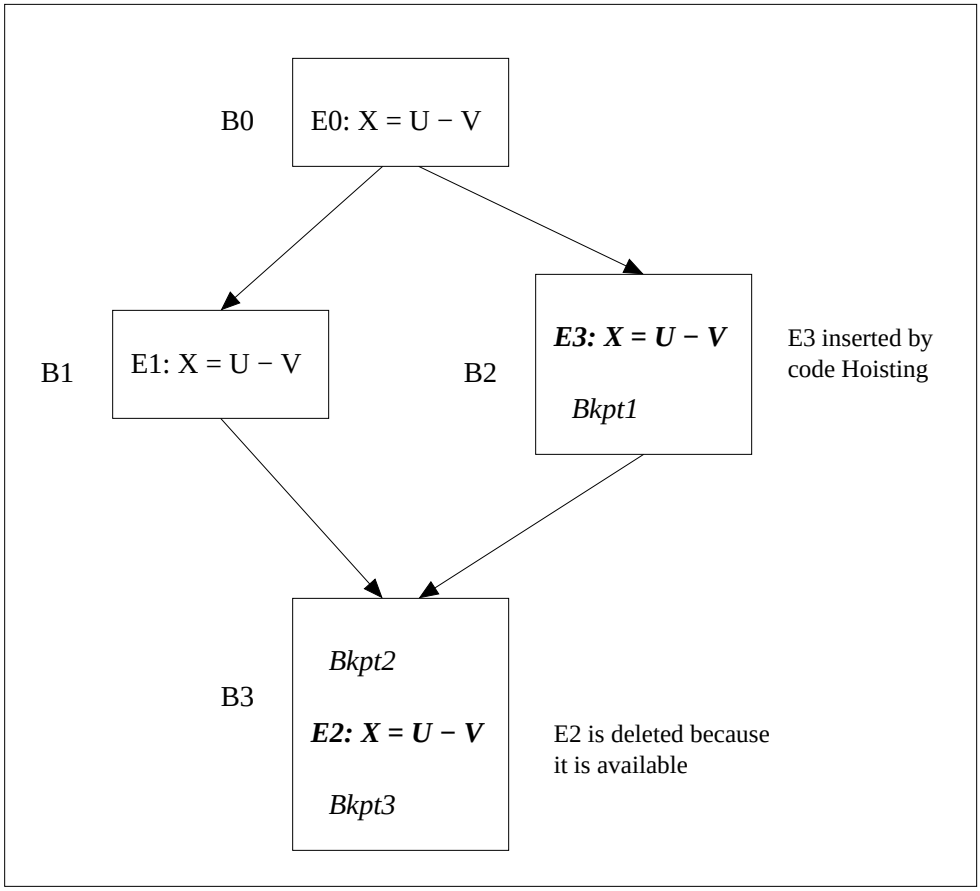


Figure 3.4: Sample code to demonstrate effects of code hoisting

redundant expression *E2* fully redundant and a candidate for further optimizations. Expression *E2* becomes available expression after code hoisting and is deleted.

If we set a breakpoint after *E3* in *B2* then the value of X is noncurrent as the value that is present is the value assigned by *E3* whereas the expected value is the value assigned by *E0*. If we set a breakpoint before *E2* in *B3* then the variable X is current if the path followed is *B0*, *B1* and *B3* and the variable X is noncurrent if the path followed is *B0*, *B2* and *B3* forcing the X to be suspect. This information is summarized in the table 3.1.

Code hoisting causes both suspected and noncurrent variables. When we set a breakpoint after *E3* in *B2* the variable x is noncurrent. The value present in x

Breakpoints set at	State of X
Bkpt1	X is noncurrent
Bkpt2	X is suspect
Bkpt3	X is current

Table 3.1: State of a variable due to code hoisting

is the value that is assigned by *E3* whereas the expected value is the value that is assigned by *E0*. For every hoisted expressions E_h we add a hoist descriptor containing the information of the variable that is assigned by E_h, the corresponding redundant expression E_r and the line number of the source instruction for which E_r is generated. For all redundant expressions E_r that are deleted by code hoisting we add redundant descriptor containing the information of line number of the source instruction to which it corresponds and the variable that is assigned.

We find all hoist descriptors that reach a particular breakpoint to find if a variable is endangered or not. All hoist expressions that reach a particular instruction is found by using global dataflow analysis similar to reaching definition analysis. After finding the data flow information, we say a variable V is noncurrent at a breakpoint if on all paths from the starting point of the program to this breakpoint there is a hoisted expression that assigns to V and the corresponding redundant expression is not executed. If the above case holds for some paths then the variable V is said to be suspect. We find the set of hoisted expressions that reach a breakpoint along all paths and the set of hoisted expressions that reach along any path.

Dataflow Analysis

Definition 3.9 (HoistGen) *HoistGen indicates the hoisted expressions generated by a block.*

Definition 3.10 (HoistKill) *HoistKill indicates the hoisted expressions killed by a block.*

Definition 3.11 (AllHoistReachIn) *AllHoistReachIn indicates the hoisted expressions that reach a basic block along all paths from the starting of the program to the basic block.*

Definition 3.12 (AllHoistReachOut) *AllHoistReachOut indicates the hoisted expressions that are present after a basic block.*

Definition 3.13 (AnyHoistReachIn) *AnyHoistReachIn indicates the hoisted expressions that reach a basic block along any paths from the starting of the program to the basic block.*

Definition 3.14 (AnyHoistReachOut) *AnyHoistReachOut indicates the hoisted expressions that are present after a basic block.*

A basic block generates a hoisted expression E_h, assigning to V if a hoisted expression exists in the basic block and no other assignment statement or another hoisted expression or a redundant expression assigns to V. HoistGen(B) indicates the hoisted expressions generated by block B. A basic block kills a hoisted assignment statement if there is a statement or a redundant expression that assigns to the same variable as that of the hoisted expression. HoistKill(B) indicates the hoist expressions that are killed by block B. We use the iterative method similar to the one presented in section 3.1.1 to solve the following dataflow equations.

$$AllHoistReachOut(I) = (AllHoistReachIn(I) - HoistKill(I)) \bigcup HoistGen(I) \tag{5}$$

$$AnyHoistReachOut(I) = (AnyHoistReachIn(I) - HoistKill(I)) \bigcup HoistGen(I) \tag{6}$$

$$AllHoistReachIn(I) = \bigcap_{J \in Pred(I)} AllHoistReachOut(J) \tag{7}$$

$$AnyHoistReachIn(I) = \bigcup_{J \in Pred(I)} AnyHoistReachOut(J) \tag{8}$$

- *Detecting endangered variables caused by global dead code elimination*

Dead code elimination deletes all assignment statements that assign to a variable if the value that is assigned in this assignment is not used later. If an assignment

statement that assigns to V is deleted then the variable V is noncurrent until the next definition of V. We present an example to show how dead code elimination affects debugging.

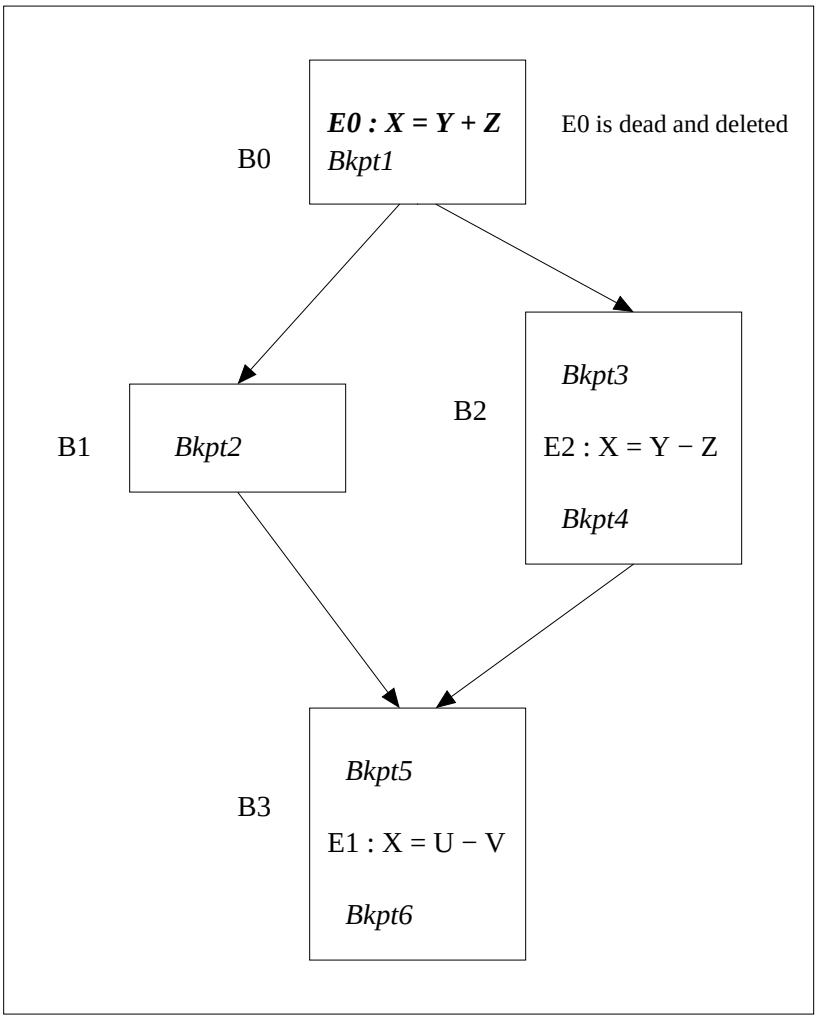


Figure 3.5: Sample Code to demonstrate effects of dead code elimination

In the sample piece of code given in figure 3.5 expression *E0* is dead as there are no uses of x until the next definition of x. At a breakpoint set after *E0* in *B0* variable x is noncurrent. At a breakpoint set in block *B1* variable x is noncurrent. At a breakpoint set before *E2* in *B2* x is noncurrent. At a breakpoint set after *E2* in *B2* x is current. At a breakpoint set before *E1* in *B3*, x is current if the path

Breakpoints set at	State of X
Bkpt1	X is noncurrent
Bkpt2	X is noncurrent
Bkpt3	X is noncurrent
Bkpt4	X is current
Bkpt5	X is suspect
Bkpt6	X is current

Table 3.2: State of a variable due to dead code elimination

followed is *B0*, *B2* and *B3* and x is noncurrent if the path followed is *B0*, *B1* and *B3* making x suspect. If a breakpoint is set after *E1* in *B3* then x is current as the value of x is the value assigned by *E1*. The effect is summarized in Table 3.2.

To find if a variable V is endangered due to dead code elimination at a breakpoint we find if a dead assignment statement existed along the path and no other definitions of V has occurred after last such dead assignment statement. For each assignment statement that is eliminated due to dead code elimination we add a dead assignment descriptor containing the information of the variable assigned, and the sequence number of the source assignment statement corresponding to the deleted assignment statement. We find all dead assignment statements that reach a particular point along all paths to say if a variable is noncurrent due to dead code elimination. We find all dead assignment statements that reach a particular breakpoint along any paths to say if a variable is suspect due to dead code elimination.

Global data flow analysis is used to find the required sets. The dataflow analysis similar to that for finding endangered variables due to code hoisting is used. Having found the required sets we can easily say if a variable is endangered due to dead code elimination at a breakpoint.

Chapter 4

Code location problem

Optimizations such as loop invariant code motion, dead code elimination and instruction scheduling insert, delete and move code and make mapping of breakpoints difficult. The debugger has to correctly map the breakpoints and check that the breakpoints are triggered the expected number of times and at expected source locations. Optimizations that involve code elimination, code duplication and code motion affect mapping of the breakpoints. The object code locations where the source breakpoints are mapped influence the number of the variables that are nonresident and endangered at a breakpoint.

The debugger uses source to object mapping and object to source mapping to map the source breakpoints and report asynchronous breakpoints. Compilers generally use different intermediate representations to perform different optimizations. Initially when the source program is compiled and converted into some intermediate representation, the compiler has to construct the source to object and object source mappings. The compiler updates the mappings when optimizations are performed. When the intermediate representations are lowered the compiler propagates the mappings to the lower level until the final object code is obtained.

4.1 Code elimination

Optimizations like dead code elimination eliminate a source statement completely. There is no object code corresponding to the deleted source statement. When the user sets a source-level breakpoint at a statement that is completely eliminated the debugger has to determine the appropriate location in the object code where the breakpoint should be mapped. The following cases arise:

1. In the simple case there is at least one source statement after the deleted source statement in the same basic block. In this case the debugger sets the breakpoint at the object code generated for the next source statement.

2. The statement that is deleted is the last statement in the basic block. In this case the debugger maps the breakpoint to the starting instruction of all the successor basic blocks of the basic block containing the deleted statement. The breakpoint is reported more than the expected number of times.

3. The statement that is deleted is the only statement in the basic block and the whole basic block is deleted. In this case the breakpoint is set at the starting instructions of all the successor blocks.

The example given in figure 4.1 illustrates the first two cases. figure 4.1(b) shows the code obtained after performing dead code elimination on code given in figure 4.1(a). S3 is the only statement present in block B2. After dead code elimination block B2 is completely deleted. If the user sets a breakpoint at S3, and the debugger maps the breakpoint to the starting instruction of block B3 then the breakpoint is triggered more than the expected number of times. The breakpoint is triggered whenever control reaches block B3 but as per the source semantics the breakpoint should be triggered only when the control reaches S3. We can set a conditional breakpoint based on the condition in S1 such that the breakpoint is triggered expected number of times. When code elimination is performed the compiler updates the source to object and the object to source mappings to reflect the changes introduced by code elimination.

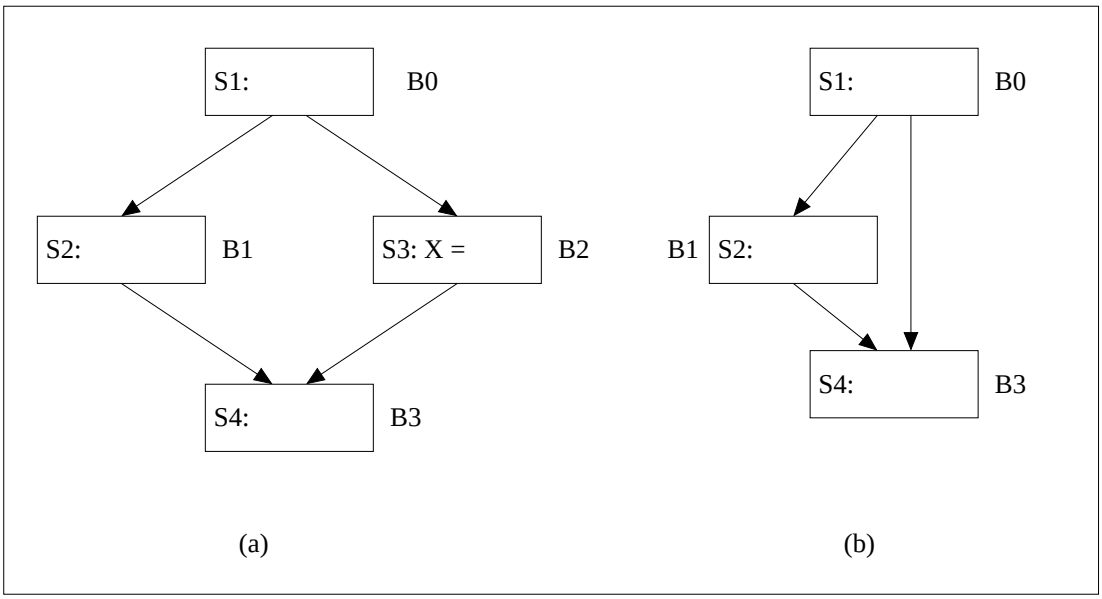


Figure 4.1: Example for code location problems caused by code elimination

4.2 Code insertion

Optimizations such as code hoisting and common subexpression elimination insert new code. The code that is added should be corresponded to a source statement so that if an asynchronous break occurs at the newly added code then the break can be reported at the appropriate source statement. Runtime exceptions and user signals are called asynchronous breaks. The compiler updates the source to object and object to source mapping by corresponding the newly added code to some source statement.

In the example shown in figure 4.2 common subexpression elimination is performed on the code shown in fig 4.2(a) resulting code in 4.2(b). The statement $Temp1 = y + z$ is the newly added statement. When an asynchronous break occurs while evaluating $y + z$ in $Temp1 = y + z$ then the debugger reports that asynchronous break has occurred at statement S1 in the source.

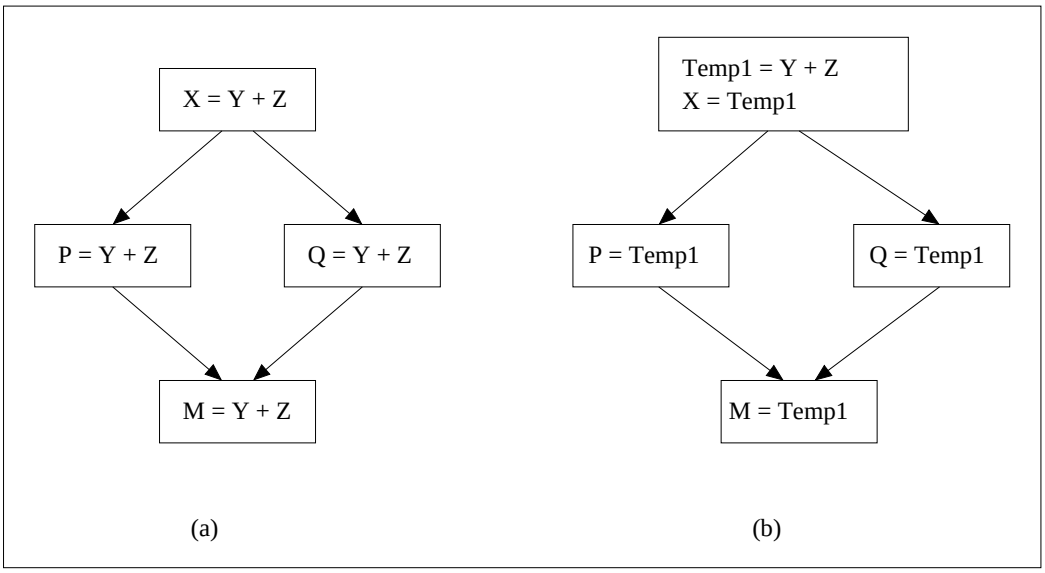


Figure 4.2: Example for Code location problems caused by Code insertion

4.3 Code motion

Many optimizations such as loop invariant code motion and partial redundancy elimination move code from one location to another location. When code motion is performed the compiler updates the source to object and object to source mappings. When a breakpoint is set at a statement S that is moved by code motion then the debugger can map the breakpoint at:

1. The object code corresponding to the moved statement.
2. The object code generated for the statement after the moved statement.
3. The object code generated for the statement immediately preceding the moved statement.

The choice of where the breakpoint is mapped influences the number of the locations that are endangered and noncurrent.

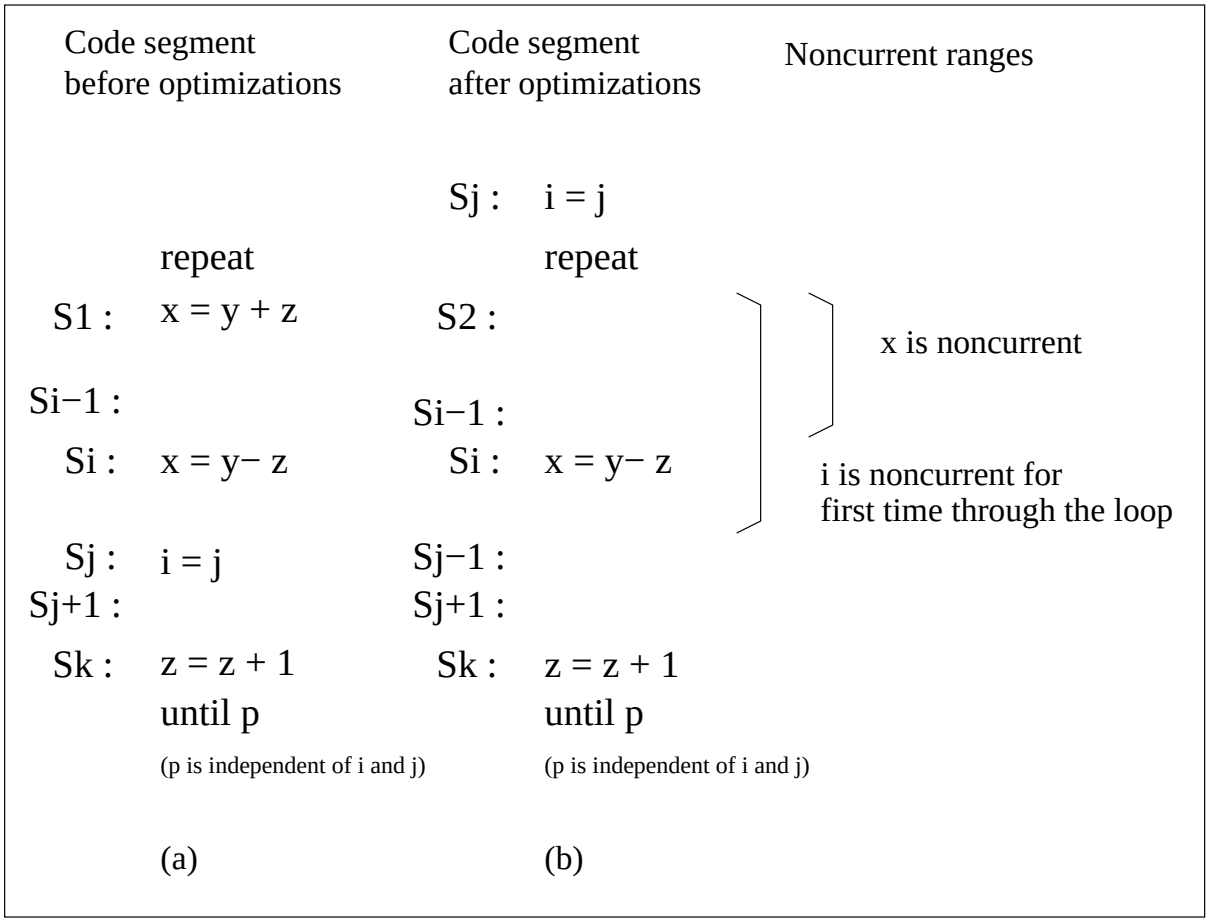


Figure 4.3: Example to show how breakpoint mapping affects data value problems

When the debugger sets the breakpoint at statement S_j in figure 4.3(b) the breakpoint occurs only once and all variables assigned in statements S_1 to S_{j-1} are noncurrent. When the debugger sets the breakpoint at statement S_{j-1} in figure 4.3(b) the breakpoint is triggered expected number of times but variable i is noncurrent at the breakpoint. If the breakpoint is set at statement S_{j+1} in figure 4.3(b) then the breakpoint is triggered the expected number of times and the variable i is current but, if the statement S_j is the last statement of the basic block then there is no statement after S_j. The effect is summarized in Table 4.1.

Breakpoint in fig 4.3(a)	Breakpoint in fig 4.3(b)	Number of times breakpoint is triggered	Endangered Variables at the breakpoint
S_j	S_j	once	variables assigned in S_1 to S_{j-1}
S_j	S_{j-1}	n times	variable assigned in S_j
S_j	S_{j+1}	n times	None

Table 4.1: Table showing effect of breakpoint mapping on data Value problems

4.4 Instruction scheduling

Instruction scheduling interleaves the object code generated for different source statements. Due to this, the object instructions generated for a source statement are not consecutive. The object instructions generated for a source instruction are split into different ranges and object instructions corresponding to any source statement can come in between these ranges. If the user sets a breakpoint on an instruction whose object instructions are split into different ranges the debugger faces of the problem of where to map the breakpoint.

The debugger can set the breakpoint at the first object instruction generated for the source statement or set the breakpoint at the first side-effecting instruction generated for an instruction. Generally assignments, function calls and branches are treated as side-effecting instructions. Because instruction scheduling interleaves the object instructions corresponding to source statements, if we set a breakpoint then we cannot perfectly say whether all the instructions that have to execute before the breakpoint have executed and that no instructions that have to execute after the breakpoint has not executed.

Chapter 5

Value change problem

Source-level debuggers while debugging unoptimized code, allow the value of a variable to be changed at debug time. When optimizations are performed it is unsafe to change the value of a variable at some breakpoints in the program. Changing the value of a variable at some locations may lead to unexpected and erroneous results. Source level debuggers like Ladebug [3] that support debugging optimized code do not allow the value of a variable to be changed when debugging optimized code.

When the compiler performs optimizations based on the value of a variable and relation between the values of variables, the debugger should not allow the user to change the value of the variable such that it violates the properties based on which the optimizations are performed. The debugger has to reevaluate some of the expressions when the value of a variable has to be changed.

Value of variables that are nonresident at a breakpoint cannot be changed at debug time. Value of some of the endangered variables can be changed at debug time. Value change problem refers to the problem of finding whether it is safe to change the value of a variable at a particular breakpoint and changing the value of the variable without affecting the semantics of the program. Allowing the user to change the value of a variable requires the compiler to pass additional information to the debugger. The debugger has to perform additional analysis to change the value of a variable. Allowing the debugger to change the value of a variable is useful in many situations. We enumerate the situations where it is beneficial to allow value

change at debug time. Some optimizations like constant folding that do not affect data value problem and code location problem but affect value change problem.

5.1 Advantages of allowing value change at debug time

1. At a particular breakpoint if we know that the value of a variable is causing an error we can change the value of the variable and then proceed to check if the rest of the program is working correctly without going through the *edit-compile-debug* cycle again. Thus, we can reduce the effective debugging time.

2. Allowing the debugger to change the value of a variable allows us to traverse all paths of the program and check that each path is working correctly. This is very useful in cases where it is difficult or impossible to change the inputs so that all the paths are traversed or a particular path is traversed.

5.2 Constant propagation and constant folding

In constant propagation, for all statements of the form $x = c$(c is a constant) the compiler replaces the later uses of x with the value of c. The statement $x = c$ is a candidate for dead code elimination. In constant folding all expressions whose operands are all constants are evaluated at compile time and the expression is replaced with the result of the evaluation. Constant propagation along with dead code elimination causes endangered variables.

Constant folding when applied individually does not affect debugging. When constant folding is applied on a constant expression where one of the operand is a constant propagated variable we cannot change the value of that variable. If we change the value of a variable that is constant propagated the changed value of the variable is not reflected in the expression where the variable is constant propagated. When the user wants to change the value of variable that is constant propagated and one of the instances is constant folded the debugger has to reevaluate the expression

that is folded with new value of the variable and replace the uses of the constant folded expression with the changed value.

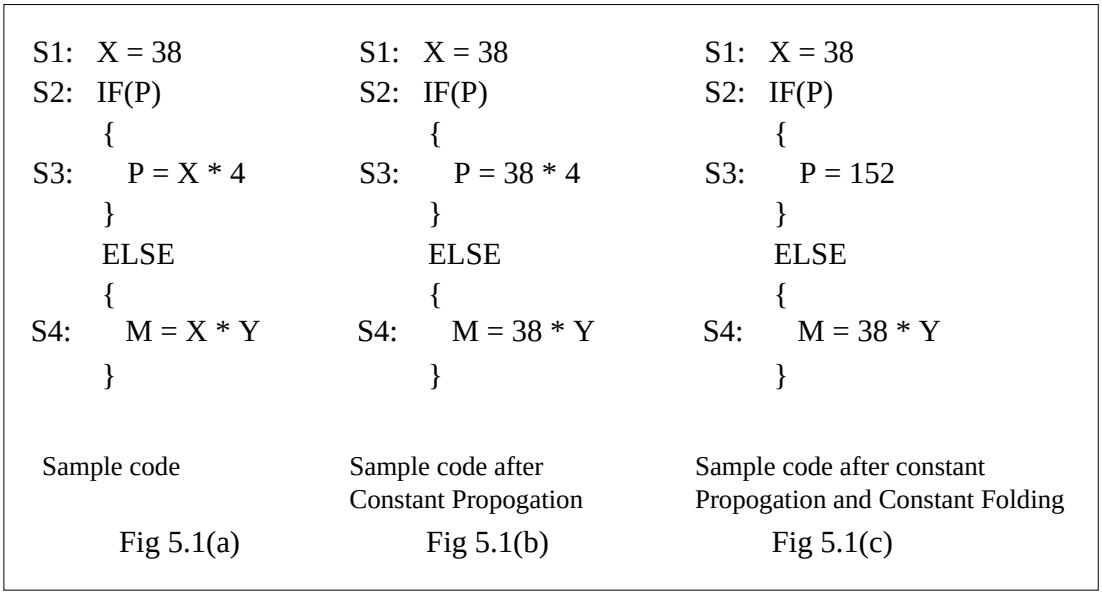


Figure 5.1: Effects of constant folding and propagation on value change problem

In the figure 5.1, sample code in fig 5.1(b) is the result of applying constant propagation on code given in fig 5.1(a) and fig 5.1(c) is the result of applying constant propagation and constant folding on code given in fig 5.1(a). If the user sets a breakpoint before statement S2 and wants to change the value of x at that breakpoint then the debugger has to reevaluate the expression in S3 with the changed value of x. The compiler must pass all the expressions that are constant folded to the debugger.

Definition 5.1 (Constant propagation definition descriptor) *Constant propagation definition descriptor is a tuple* <N,V,Value,const-prop-id> *where* N *is the line number of the source instruction,* V *is the variable that is assigned,* Value *is the constant value that is assigned and* const-prop-id *is the unique identifier assigned for each constant propagation definition descriptor.*

Definition 5.2 (Constant propagation descriptor) *Constant propagation descriptor is a tuple* <N,const-prop-id> *where* N *is the line number of the source*

instruction for which this is generated and const-prop-id *indicates the identifier of the corresponding constant propagation definition descriptor.*

For every assignment expression that is constant propagated, we store a constant propagation definition descriptor. For each expressions where constant propagation is performed we add a constant propagation descriptor.

For all constant folded expressions that contain a constant that is result of constant propagation, we store the expression and pass on that expression to the debugger. If the user wants to change the value of a variable at a breakpoint then the debugger checks if any constant propagation definition descriptors involving this variable reaches the breakpoint. If such constant definitions exist then find if any of the instances where this is constant propagated is involved in constant folding, if so then the debugger has to reevaluate the expression and replace the old value with the new value. We use simple global dataflow analysis similar to reaching definitions analysis to find all constant propagation definition descriptors that reach an instruction.

If the compiler performs constant propagation and constant folding recursively then the amount of the information (the number of the expressions that are) needed to be stored grows exponentially. We give an example to show the above case.

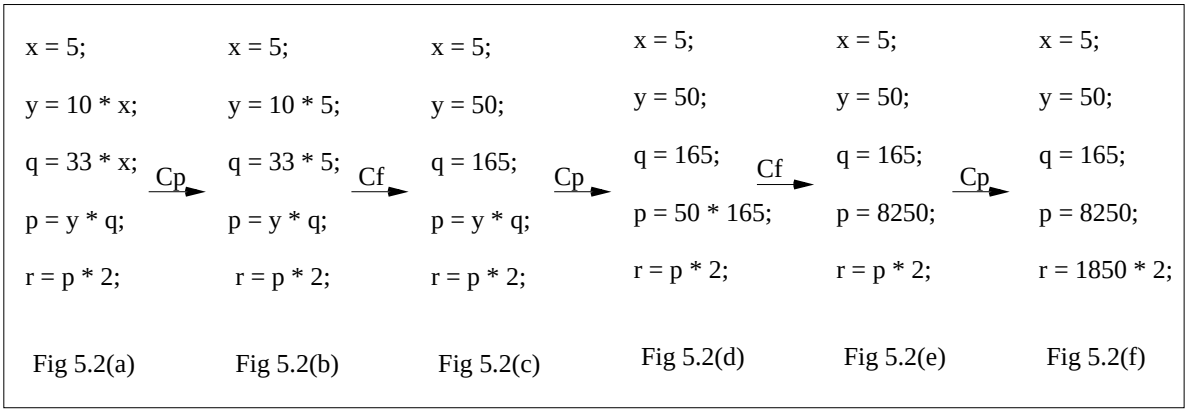


Figure 5.2: Effects of constant folding and propagation on value change problem

Constant propagation of a constant assignment makes some of the expressions

constant expressions and those constant expressions can be folded. Some of the constant expressions thus folded may lead to constant assignment statement and that constant assignment can be propagated and this chain can continue. For example, in fig 5.2 we show sample code on which constant propagation and constant folding are repetitively performed. In such a case at each cycle we have to store all the expressions that are folded, the places where the constant definition is propagated. As the number of times this cycle is repeated increases, the amount of information to be stored also increases. When the user wants to change the value of a variable that is involved in such a cycle the debugger has to reevaluate all the expressions that are folded at each stage of the cycle.

When the user wants to change the value of x before the statement *y=50* in fig 5.2(e) the debugger has to reevaluate the expressions $y = 10 * x$, $q = 33 * x$, and $p = y * q$ with the changed value of x and replace the expressions with the corresponding newly evaluated value. For this the debugger has to store all the constant expressions that are folded.

5.3 Copy propagation

For every assignment statement of the form $x = y$ we replace later uses of x with y until the intervening instructions do not change the value of either x or y. Copy propagation performs optimizations based on the relation between the value of the variables so the debugger should not change the value of the variable such that the relation is violated. Copy propagation affects value change problem. The debugger cannot directly change the runtime value of a variables that is involved in copy propagation. If the debugger just changes the runtime value of a variable that is copy propagated then the changed value is not reflected in the statements where it is copy propagated as the uses of the variable in these statements are replaced with uses of another variable.

In the sample code shown in fig 5.3 if the user sets breakpoint before S2 and wants to change the value of variable x, just changing the runtime value of the variable will not suffice as the changed value is not reflected in statements S2 to S5

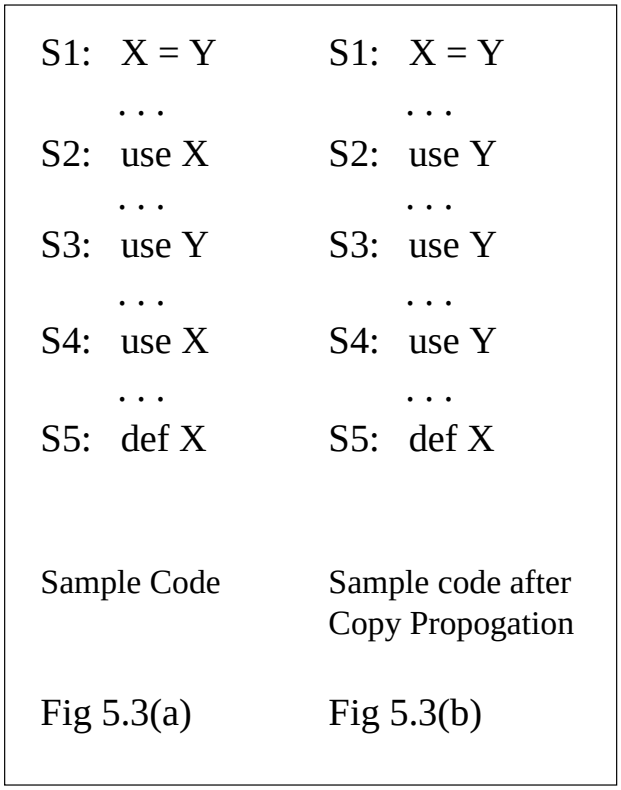


Figure 5.3: Effects of copy propagation on value change problem

as the uses of x in these statements are replaced by uses of y. If the user wants to change the value of y at breakpoint set after S2, then the debugger should not directly change the runtime value of y as the uses of x in statements S2 and S4 are replaced with y.

Compiler gathers more information about the optimizations and propagates that information to the debugger. The additional information is used by the debugger to change the value of a variable. For all the copy statements that are cause of copy propagation, we store the copy propagation definition descriptor $<N, V1, V2>$ where N is the line number of the source instruction, *V1* is the variable that is assigned and *V2* is the variable whose value is assigned. For all statements where copy propagation is done we store copy propagation descriptor $<N,$ *Copy-def-id*$>$ where N is the line number of the source instruction for which this instruction is

generated and *Copy-def-id* is the unique identifier for the copy instruction that is the cause of this copy propagation.

At a breakpoint, if user wants to change the value of a variable then check if the variable is involved in copy propagation. Two cases arise first, the variable that has to be changed is the value that is assigned and in the second case the variable that has to be changed is the variable whose value is assigned. In the first case for all the statements where this variable is replaced with another variable, change the statement so that it uses the new and changed value of the variable and change the runtime value of the variable. The debugger uses code patching to do this. In the second case where the variable to be changed is on the right hand side of the copy statement replace the uses of the variable that are copy propagated instances of another variable with uses of the old value of the variable and change the runtime value of the variable.

5.4 Common subexpression elimination

In common subexpression elimination if a subexpression is evaluated many times in the program then the compiler evaluates the subexpression once, stores the result in a temporary variable and replace the later evaluations of the subexpression with the temporary variable. Common subexpression elimination affects value change problem. When user wants to change the value of a variable that is involved in a common subexpression the debugger has to reevaluate the subexpression and change the runtime value of the temporary as well as the runtime value of the variable to be changed.

Definition 5.3 (cse assignment descriptor) *Cse assignment descriptor is a tuple* <N, Id, Temp, V_1, ..., V_n> *where* N *is the source line number to which the subexpression corresponds,* Id *is the unique identifier value that is assigned to each subexpression,* Temp *is the variable that is used to store the result of the subexpression and* V_1 to V_n *are the variables involved in the subexpression.*

Definition 5.4 (cse descriptor) *Cse descriptor is a tuple* <N, Cse-id> *where* N

is the source line number of the source statement corresponding to this instruction and Cse-id *is the unique identifier of the corresponding cse assignment descriptor.*

For common subexpression elimination as in copy propagation we capture the optimization information in the descriptors and use that information for changing the value of a variable at debug time. When subexpression elimination is performed at all instructions where the evaluation is done and the result is stored into a temporary, we add a cse assignment descriptor. At all places where the subexpression is replaced with a temporary variable we add the cse descriptor.

At a breakpoint if the user wants to change the value of a variable that is involved in a subexpression the debugger finds the cse assignment descriptor corresponding to the subexpression and reevaluate the subexpression with the changed value of the variable and update the runtime value of the temporary variable. We use a simple dataflow analysis similar to reaching definition analysis to find all cse assignment descriptors that reach a point and check if the variable that is to be changed is present in any one of these descriptors. If the variable is involved in any such subexpression then we reevaluate the subexpression with changed value of x and update the runtime value of the temporary. This requires the debugger to reevaluate the subexpression at debug time.

5.5 Code motion transformations

Optimizations like loop invariant code motion, global instruction scheduling and code hoisting change the order of execution of the source statements and source assignment statements. Changing the order of execution of the source assignment statements causes endangered variables. Changing the order of the execution of the statements affects value change problem. Code motion moves some statements to execute before the breakpoint. In this case when the debugger changes the value of a variable the changed value of the variable is not reflected in the statements moved earlier.

In figure 5.4 statements S1 and S3 are loop invariants and hence they are moved out of the loop. If user wants to change the value of a variable X at breakpoint

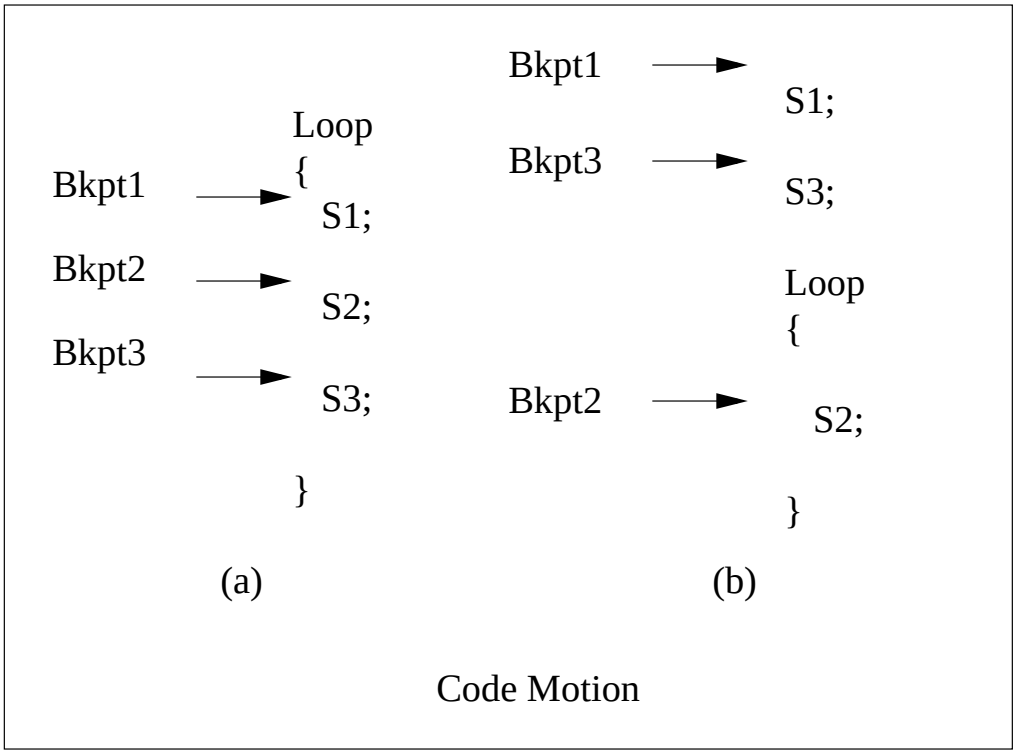


Figure 5.4: Effects of code motion on value change problem

Bkpt2 and the variable X is used in S1 and S3 then we cannot change the value of the variable as the statements S1 and S3 that have to use the changed value of X have already executed before the loop. When user wants to change value of a variable then the debugger has to check if until the next definition of X any statements involving the use of X have executed out of order if so then the debugger should not allow the value of the variable to be changed at that breakpoint. We find any statement has executed out of order using the source execution order and object execution order as described in section 3.2.3.

Chapter 6

Testing the framework, conclusions and future work

We developed a simple optimizing compiler and a simple source-level debugger for programs written in C to show that allowing value change at debug time while debugging optimized code is viable. The source-level debugger has support for debugging optimized code. The debugger has support to change the value of a variable while debugging optimized code.

Instead of designing a compiler that generates object code for some machine and designing a source-level debugger for that machine, we developed an interpreter for an intermediate format and built a source-level debugger on that interpreter. The optimizing compiler compiles and optimizes the program and outputs the intermediate format for the optimized program.

6.1 Development environment

We developed the debugger and the optimizer in the following environment.

- **Machine :** Pentium II machine
- **Operating system :** Red Hat Linux with kernel 2.2.5-15
- **Languages Used :** C++

- **Tools Used :** SUIF compiler system version 1 release 1.3.0.1, Machine SUIF version 1.

6.2 Optimizing compiler

We use SUIF compiler system for developing the optimizing compiler. The SUIF compiler system is a flexible framework for advanced compiler research [1]. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format called SUIF IR.

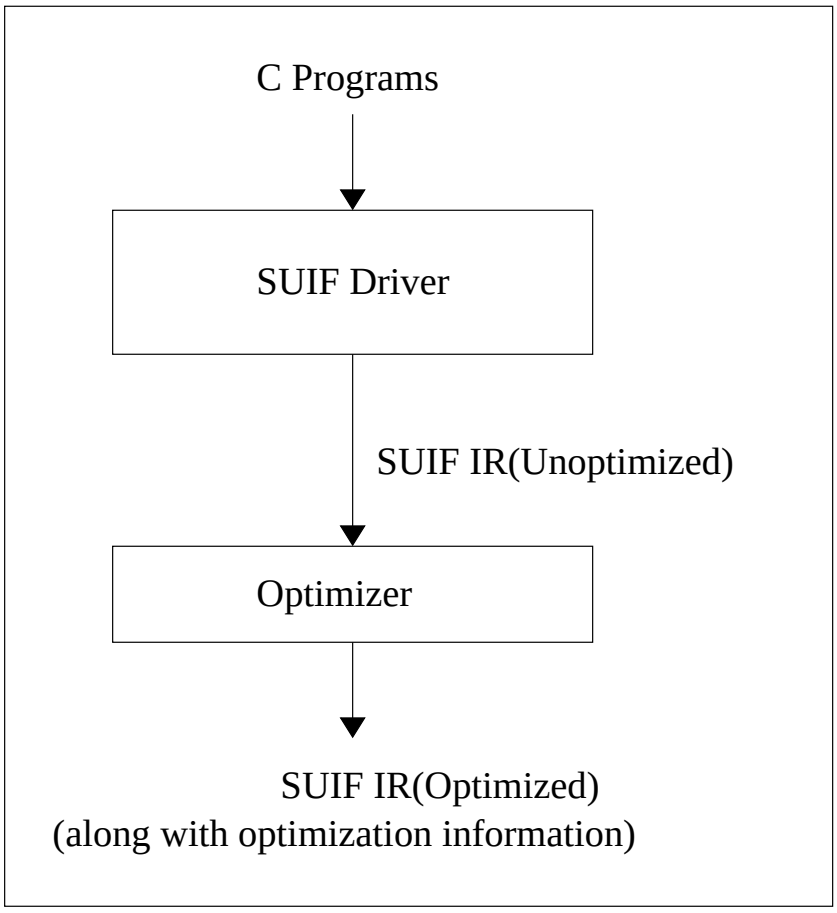


Figure 6.1: Structure of Optimizing Compiler

Our optimizing compiler shown in figure 6.1 first uses SUIF compiler and converts

the input C program into its equivalent SUIF IR. The optimization pass performs optimizations on SUIF IR and appends the optimization information that has to be passed to the debugger.

The SUIF IR is a flexible Intermediate format. There are various levels of SUIF IR, at the higher level it is present as an abstract syntax tree and at the lower level it is present as straight-line code. The instructions present in SUIF IR at the lower level are similar to a simple RISC instructions. We can add annotations to various objects like instructions, variables etc. The optimizations information that is to be passed to the debugger is added as annotations to the SUIF instructions. The optimizing compiler finally outputs the IR of the optimized program along with the information that has to be passed to the debugger.

The compiler performs the following scalar optimizations:

1. Constant folding
2. Constant propagation
3. Copy propagation
4. Dead code elimination
5. Common subexpression elimination
6. Loop invariant code motion
7. Unreachable code elimination

Each optimization is performed as a separate pass on the IR. All the scalar optimizations need some dataflow information. We use cfg library and dfa library provided by Machsuif package to do the necessary dataflow analysis. Machsuif [16] is a package that can be used with SUIF to perform machine specific optimizations.

6.3 Debugger

We developed an interpreter for SUIF IR. We convert a C program into SUIF IR and the IR is interpreted using the interpreter. The debugger is built on top of the

interpreter. The debugger takes SUIF IR and debugs the program. The debugger provides basic functionalities of a source-level debugger. The functionalities provided by the debugger are:

1. Setting and removing breakpoints
2. Displaying the source-level value of the variable
3. Changing the value of a variable at debug time
4. Single stepping
5. Displaying call stack

The debugger does not have full support for changing the value of a variable. The debugger identifies whether it is safe or unsafe to change the value of a variable at debug time. In some cases when a variable is unsafe for value change, the debugger can perform some evaluations and allow the value of the variable to be changed. This case occurs when some optimizations like copy propagation and common subexpression elimination are performed. The debugger is not able to handle asynchronous breaks.

6.4 Sample Execution of the debugger

In this section we give two examples showing how the debugger debugs optimized code. In the first example we show data-value problems and how the debugger responds to them. In the second example we concentrate on code location problem.

Example 1

In this example we show data value problems caused when copy propagation and loop invariant code motion are performed. Copy propagation and loop invariant code motion are performed on code in file *Unoptimized_ ex1.c* to give code in file *Optimized_ ex1.c*. The statement p = q at line number 12 is copy propagated and the later uses of p are replaced with q. Use of p in line number 14 is replaced with q. The

statement x = p + 234 is replaced by x = q + 234 and is moved out of the loop by loop invariant code motion. Optimizations are applied on the file *Optimized_ ex1.c* to produce the SUIF IR file *Optimized_ ex1.spx*. The file *Unoptimized_ ex1.spx* is SUIF IR representation file generated for *Unoptimized_ ex1.c*.

Unoptimized_ex1.c

```
1 int main()
2 {
3     int i=0;
4     int j=34;
5     int p=23;
6     int q=1;
7     int x=12;
8     int m;
9
10     for(i=0;i<10;i++)
11     {
12         p = q;
13         j++;
14         x = p + 234;
15         p = j + 769 * 45;
16         m++;
17         printf("\nValue of I is :%d\tValue of m is :%d",i,m);
18     }
19 }
```

Optimized_ex1.c

```
1 int main()
2 {
3     int i=0;
4     int j=34;
5     int p=23;
```

```
6      int q=1;
7      int x=12;
8      int m;
9      x = q + 234;
10      for(i=0;i<10;i++)
11      {
12          p = q;
13          j++;
14          p = j + 769 * 45;
15          m++;
16          printf("\nValue of I is :%d\tValue of m is :%d",i,m);
17      }
18 }
```

When we set a breakpoint at line number 13 and ask the debugger to display the value of x then it displays the value of x and says x is noncurrent as the expected value is different from the actual value. At the current breakpoint we cannot change the value of x as the statement x = p + 234 is executed before the breakpoint. Similarly, we cannot change the value of p as it is copy propagated and the later uses of p are replaced by use of q. When we set a breakpoint at line number 16 then the debugger shows that the variables x and p are current and the change of value of p is allowed.

Debugging of Optimized Code

```
$ debugger Optimized_ex1.spx
debugger> breakline 13
The Breakpoint Number is :  1
The Line number where the bkpt is set is :  13
debugger> continue
The breakpoint number : 1 at line number :13 is triggered
debugger> display x
The value of the variable x is :235
The variable is non-current due to loop invariant code motion
```

```
debugger> display p
The value of the variable p is :1
The variable is current
debugger> change p 34
Request to change value of variable p with value :34
changing value of the variable is unsafe due to copy propagation
debugger> change x 23
changing value of the variable is unsafe due to loop invariant code motion
debugger> breakline 16
The Breakpoint Number is :  2
The Line number where the bkpt is set is :  16
debugger> continue
The breakpoint number : 2 at line number :16 is triggered
The value of the variable p is :34640
The variable is current
The value of the variable x is :235
The variable is current
debugger> change p 0
Request to change value of variable p with value :0
The variable is current
The value of the variable is changed
debugger> clear_line_breakpoint 13
The break point at line :13 is cleared
debugger> clear_line_breakpoint 16
The break point at line :16 is cleared
debugger> continue
Value of I is :0  Value of m is :1
Value of I is :1  Value of m is :2
Value of I is :2  Value of m is :3
Value of I is :3  Value of m is :4
Value of I is :4  Value of m is :5
```

```
Value of I is :5  Value of m is :6
Value of I is :6  Value of m is :7
Value of I is :7  Value of m is :8
Value of I is :8  Value of m is :9
Value of I is :9  Value of m is :10
Program Execution is complete
```

Debugging Unoptimized Code

```
$debugger Unoptimized_ex1.spx
debugger>breakline 13
The Breakpoint Number is :      1
The Line number where the bkpt is set is :      13
debugger>continue
The breakpoint number : 1 at line number :13 is triggered
debugger>display x
The value of the variable x is :12
The variable is current
debugger>display p
The value of the variable p is :1
The variable is current
debugger>change p 34
Request to change value of variable p with value :12
The variable is current
The value of the variable is changed
debugger>change x 23
Request to change value of variable x with value :23
The variable is current
The value of the variable is changed
debugger>breakline 16
The Breakpoint Number is :      2
The Line number where the bkpt is set is :      16
debugger>continue
```

```
The breakpoint number : 2 at line number :16 is triggered
The value of the variable p is :34640
The variable is current
The value of the variable x is :246
The variable is current
debugger>change p 0
The variable is current
The value of the variable is changed
debugger>clear_line_breakpoint 13
The break point at line :13 is cleared
debugger>clear_line_breakpoint 16
The break point at line :16 is cleared
debugger>continue
Value of I is :0        Value of m is :1
Value of I is :1        Value of m is :2
Value of I is :2        Value of m is :3
Value of I is :3        Value of m is :4
Value of I is :4        Value of m is :5
Value of I is :5        Value of m is :6
Value of I is :6        Value of m is :7
Value of I is :7        Value of m is :8
Value of I is :8        Value of m is :9
Value of I is :9        Value of m is :10
Program Execution is complete
```

Example 2

In this example we consider how breakpoint mapping is done while debugging optimized code. Constant propagation and loop invariant code motion are applied on code in *Unoptimized_ ex2.c* and it is transformed to code in *optimized_ ex2.c*. The statement x = 5 at line number 14 is constant propagated and the use of x in line number 16 is replaced with 5. The statement p = x * y which is replaced by p =

5 * y, is moved out of the loop by loop invariant code motion. We give the execution of the debugger for the optimized and the unoptimized versions of the program.

Unoptimized_ex2.c

```
1 int main()
2 {
3     int i=0;
4     int x=0;
5     int y;
6     int z=34;
7     int p=-1;
8     int q=34;
9     int m=43;
10    int j=1;
11
12     while(i<10)
13     {
14         x = 5;
15         i+=2;
16         p = x * y;
17
18         x = m;
19         q = x + z;
20         m = q + i;
21         j *= 2;
22     }
23     printf("\nThe value of i is :%d\t value of j is :%d\n",i,j);
24 }
```

Optimized_ex2.c

```
1 int main()
2 {
```

```
3     int i=0;
4     int x=0;
5     int y;
6     int z=34;
7     int p=-1;
8     int q=34;
9     int m=43;
10    int j=1;
11     p = 5 * y;
12     while(i<10)
13     {
14         x = 5;
15         i+=2;
17
18         x = m;
19         q = x + 34;
20         m = q + i;
21         j *= 2;
22     }
23     printf("\nThe value of i is :%d\t value of j is :%d\n",i,j);
24 }
```

When a breakpoint is set at line number 17 in the source program the breakpoint is executed once before entering the loop and once for each iteration the loop body is executed. When the breakpoint is triggered before entering the loop, the variables i and x that are assigned in the loop before line number 17 are reported as non-current as they do not have the expected values. When the breakpoint is triggered inside the loop the variables x and i have the expected values and are reported as current. As the loop is executed 5 times the breakpoint is reported 6 times. When the unoptimized code is executed the breakpoint occurred the expected number of time. The file *Optimized_ ex2.spx1* is the SUIF IR file for the optimized program *Optimized_ ex2.c* and *Unoptimized_ ex2.spx1* is the SUIF IR for the unoptimized

program *Unoptimized_ ex2.c*

Debugging Optimized code

```
$debugger Optimized_ex2.spx1
debugger> breakline 17
The Breakpoint Number is :  1
The Line number where the bkpt is set is :  17
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
debugger> display x
The value of the variable x is :0
The variable is non-current
debugger> display i
The value of the variable i is :0
The variable is non-current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :2
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :4
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :6
The variable is current
The value of the variable x is :5
```

```
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :8
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :10
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The value of i is :10    value of j is :32
Program Execution is complete
```

Debugging Unoptimized code

```
$debugger Unoptimized_ex2.spx1
debugger> breakline 17
The Breakpoint Number is :  1
The Line number where the bkpt is set is :  17
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
debugger> display x
The value of the variable x is :5
The variable is current
debugger> display i
The value of the variable i is :2
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
```

```
The value of the variable i is :4
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :6
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :8
The variable is current
The value of the variable x is :5
The variable is current
debugger> contiue
The breakpoint number : 1 at line number :17 is triggered
The value of the variable i is :10
The variable is current
The value of the variable x is :5
The variable is current
debugger> continue
The value of i is :10    value of j is :32
Program Execution is complete
```

6.5 Conclusions

We conclude that it is viable to allow the value of a variable to be changed at debug time. The compiler has to store some additional information about the optimizations and pass that information to the debugger. The debugger has to perform

some additional computation at debug time. The time taken for additional computations is negligible when performing interactive debugging. With a small degree of invasiveness we can allow value change at debug time.

6.6 Future work

All the variables that are reported as suspect can be classified as either current or noncurrent if we have the information of the execution path that is followed by the program. This improves the response of the debugger. We can extend the approach so that the debugger gathers dynamic information of the path that is followed by the program and performs some additional analysis at debug time.

The scope of the optimizations that we consider is less. One can extend this approach for many more optimizations like software pipelining, loop transformations etc.

When we are unable to change the value of a variable we can give improved response to the user indicating why we are not able to change the value of a variable. This requires little more analysis on part of the debugger.

If the debugger allows a degree of invasiveness then improved response can be given to the user and the value of a variable can be changed at many more locations. We can extend this approach allowing a degree of invasiveness.

Bibliography

[1] Suif documentation.
Available at http://suif.stanford.edu/suif/suif1/.

[2] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers Principles Techniques and Tools*. Addison-Wesley, 1999.

[3] Compaq computer Corporation. *Ladebug Debugger Manual*, March 2001.
Available at: ftp://ftp.compaq.com/pub/products/software/developer/.

[4] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems Vol 16, No 3*, May 1994.

[5] Michelle Ruscetta Deborah S Coutant, Sue Meloy. Doc:a practical approach to source-level debugging of globally optimized code. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June, 1988.

[6] Steve Simmons Gary Brooks, Gilbert J Hansen. A new approach to debugging optimized code. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, May, 1992.

[7] John Hennessey. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems. Vol. 4, No 3*, July 1982.

[8] Hewlett Packard. *HP wdb documentation*, ninth edition edition, June 2000.

[9] Harish Patil Bruce Olsen Wen-mei W. Hwu Le-Chen Wu, Rajiv Mirani. A new framework for debugging globally optimized code. *Proceedings of ACM*

SIGPLAN conference on Programming Language Design and Implementation, May 1999.

[10] Microsoft Corporation. *Visual C user manual.*
Available in MicroSoft's Visual Studio Documentation.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan kaufmann Publishers.

[12] Ali Reza. *Source-Level Debugging of Globally Optimized Code.* PhD thesis, School of computer science, Carnegie Mellon University, Pitsbugrh, PA 15213, 1996. Available at: http://cs.cmu.edu/ ali/thesis.ps.

[13] Ali Reza and Thomas Gross. Detection and recovery of endangered variables caused by instruction scheduling. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June, 1993.

[14] Ali Reza and Thomas Gross. Source level debugging of scalar optimized code. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, May, 1996.

[15] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB.* GNU, March 2000.
Available at: http://sunsite.ualberta.ca/Documentation/Gnu/gdb-5.0/.

[16] Michael D. Smith. Extending suif for machine-specific optimizations, July 1997.
Available from http://www.eecs.harvard.edu/hube/software/v130/overview.html.

[17] Sun Microsystems. *Forte C Datasheet*, October 2000.
Available at: http://www.sun.com/forte/c/datasheet.html.

[18] Polle Zellweger. An interactive high-level debugger for control-flow optimized programs. *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, 1983.