Source-Level Debugging of Globally Optimized Code

Ali-Reza Adl-Tabatabai

June 20, 1996

CMU-CS-96-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Thomas Gross, Chair
Bernd Bruegge
Peter Lee
Susan Graham, UC Berkeley

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152.

Keywords: Source-level debugging, compiler optimization, code generation, nonresident variables, endangered variables, suspect variables, noncurrent variables

Abstract

Compiler optimizations play a crucial role in the performance of modern computer systems. Debugger technology, however, has lagged behind in its support of optimizations. Debugging the unoptimized translation is often impractical, so handling of optimized code by the debugger is necessary. But compiler optimizations make it difficult to provide source-level debugger functionality: When a program is compiled with optimizations, mappings between breakpoints in the source and object code become complicated, and values of variables may be either inaccessible in the runtime state or inconsistent with what the user expects at a breakpoint. Although researchers and implementors have long acknowledged the need for source-level debugging of optimized code, compiler developers have generally avoided implementing this support. Designing a source-level debugger for globally optimized code remains an open problem.

This dissertation presents techniques that enable accurate source-level debugging of fully optimized code. These techniques are demonstrated in the context of a compiler that performs a complete set of global scalar optimizations for commonly used source languages such as C. The techniques presented in this dissertation are non-invasive — that is, they do not constrain optimizations in any way and do not require instrumentation of the debugged program. All interactions between the debugger and the user are in terms of the original source text.

In general, it is impossible to hide the effects of optimizations from the user and to provide the same interactions as in unoptimized code. The approach presented in this dissertation makes the effects of optimizations transparent whenever possible. Only when optimizations cannot be hidden from the user does the debugger inform the user of the effects of optimizations on the expected execution behavior of the program.

This dissertation analyzes in detail the problems caused by scalar optimizations such as partial redundancy elimination, register allocation, instruction scheduling, and many others. This dissertation also presents the low-level algorithms necessary in the compiler and debugger to track the effects of optimizations, and discusses the limitations of the techniques. Finally, this dissertation presents measurements of how often a user is likely to be affected by optimizations during debugging. These measurements are based on an implementation of these techniques in the `cmcc` compiler, an optimizing C compiler developed as part of this dissertation.

Acknowledgments

I am indebted to Thomas Gross, my advisor, whose support, encouragment, and technical feedback have made this dissertation possible. I am also grateful to Bernd Bruegge, Peter Lee, and Susan Graham, who took the time to be on my committee. Their many helpful comments and suggestions have greatly improved the quality of this dissertation.

I had the good fortune of collaborating with Guei-Yuan Lueh in building the `cmcc` compiler. Guei-Yuan is an expert in many aspects of compiler optimization; without his experienced help the `cmcc` project would not have been possible.

I would like to acknowledge the members of the iWarp/Nectar lunch seminar, who provided critical feedback on my research and helped me sharpen my presentation skills.

Special thanks to my friends outside of CMU, who made life in Pittsburgh much more enjoyable: Mehdi, Robbie, Amin, Ali, Shokran, Leela, Ario, Neda, Aalah, Mitra, and the rest of the "gang".

Finally, my biggest thanks go to my family, who put up with me being so far away for so long.

To my parents, Abdol-Reza and Pari.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a complicated and costly process, and the development of software tools for easing this process continues to be an important problem in both academia and industry. One valuable program development tool is the source-level debugger. A source-level debugger allows the software developer to monitor an executing program (the *debuggee*) at the source level. An *interactive* debugger provides the user with mechanisms to halt or resume execution and to inspect the state of the debuggee. All interactions between the user and the debugger are in terms of the debuggee's high-level language source.

Bugs are a normal part of the software life cycle and the source-level debugger aids the software developer in locating the source of a programming error after testing or usage of a program has shown the existence of such an error. Using a source-level debugger, the user can inspect the state of a crashed program (*post-mortem* debugging) and discover which source-level values caused the runtime error. When used interactively, the source-level debugger allows the developer to monitor the progress of a program, and to see how source-level values become corrupted, causing a program to produce incorrect results.

The source-level debugger also acts as an important analysis tool and aids the developer in understanding how a complex piece of software operates. While most software engineering tools allow the user to statically analyze a program, debuggers allow the user to better understand the dynamic behavior of a program. This is important, for instance, in helping the developer understand how a change affects a program, or how the behavior of a program changes when ported to a new hardware or software environment. A debugger also allows the user to discover discrepancies between the specification documentation and the actual behavior of a program.

Almost all modern compilers are available with companion source-level debuggers. The more advanced of these debuggers have graphical user interfaces and contain many features that make the debugging process easier and more efficient. But current debugger technology does not allow a software developer to debug an optimized translation of a program at the source level. Compiler optimizations complicate the correspondence between the source program and the object code, thus making the task of the source-level debugger difficult.

Compiler optimizations are playing an increasingly larger role in the performance of modern computer systems and the importance of using optimizations during software development has increased, as competitive forces require production software to execute as efficiently as possible. Optimizations, however, make it difficult to provide source-level debugging functionality. Although researchers and implementors have long acknowledged the need for source-level debugging of optimized code [102, 49, 86], compiler developers have generally avoided supporting source-level debugging of optimized code. Designing a source-level debugger for globally optimized code remains an open problem.

Current trends in processor design are towards an increasing reliance on compiler transformations to achieve high performance. Designers of modern processors take optimizing compiler technology into account and optimizing compilers play a large role in the performance of a processor. One way in which designers have given a large role to the compiler is by exposing the machine-level resources of a processor to the compiler. Such resources include instruction-level parallelism, large register files, instruction fetching mechanisms, and memory hierarchies. The compiler speeds up program execution by performing transformations that exploit the resources exposed by a target processor. For example, instruction scheduling [43, 51, 67, 14, 101] can increase the efficiency of processors with instruction-level parallelism by statically scheduling independent operations for concurrent execution. Global register allocation [25, 32] can effectively exploit architectures that have large registers files and high memory access latencies by keeping the most frequently accessed values in registers. Using software branch prediction [98, 11, 42], the compiler can reorganize object code to increase the instruction fetch efficiency of pipelined or superscalar processors [54, 82]. Transformations such as blocking [68] and prefetching [22, 77] can improve a program's data cache locality, thus reducing stalls due to cache misses [77].

The importance of optimizations is not limited to low-level machine-specific transformations. The modern RISC approach to designing processors has also increased the effectiveness and importance of classical (machine-independent) compiler optimizations. The simple load–store instruction sets of RISC processors not only provide a simple code selection target for the compiler, but also act as suitable compiler intermediate representations [57, 10, 80]. By modelling each RISC instruction in its intermediate representation, the compiler exposes most of the final instructions to classical optimizations — such as partial redundancy elimination, strength reduction, and so on — thus reducing path lengths in the object code. Even newer implementations of CISC architectures are providing a RISC core set of instructions that execute efficiently and are simple to model in a compiler. For example, Intel's Pentium [24] and Pentium Pro [47], AMD's K5 [88] and NexGen's Nx686 [48] processors are superscalar implementations of the x86 architecture, which can concurrently dispatch only RISC-like instructions [56]; complex instructions are dispatched one at a time. An Intel application note describing instruction selection and scheduling for the Pentium processor advises against selecting complex instructions and suggests using a load–store model of instruction selection [55]. The larger register files of RISC processors allow the use of good heuristic global register allocation techniques, alleviating the phase-ordering problems between register allocation and classical optimizations that increase register pressure (e.g., code motion, induction variable strength reduction, function inlining, etc.).

Although compiler optimizations are playing a larger role in the performance of computer systems, debugger technology has lagged behind in its support of optimizations. By duplicating, eliminating, and reordering operations and values, optimizing transformations make it impossible to provide the illusion that the source program is executing one source statement at a time. Current debugger technology disallows optimizations and requires that a program be compiled in a straightforward, unoptimized manner for it to be debugged using a source-level debugger; the compiler usually provides an option that produces such a debuggable translation of a program (e.g., the `-g` "debug flag" provided in UNIX C compilers).

The difficulties caused by optimizations have prevented most compiler systems from supporting the debugging of optimized code, although some systems provide debugger mechanics without warranties for the "debugging" of optimized code. To cite from the description of the options in the C manual of a major Unix vendor:

> **-g3** `Have the compiler produce additional symbol table information for full symbolic debugging for fully optimized code.  This option makes the debugger inaccurate.`

This approach to debugging optimized code does not take the effects of optimizations into account, and is inadequate because the debugger will sometimes provide misleading responses to the user. For example, in response to a user query of a variable's value, the debugger may erroneously show some other program value, leading the user to draw incorrect conclusions about the behavior of the debuggee. I will illustrate this problem in the later example of Figure 1.1. The reason for this inaccuracy is that the simplistic mappings generated by the compiler — and used by the debugger for debugging unoptimized code — are not powerful enough to take into account the effects of optimizations.

Rather than provide erroneous and misleading responses to the user, other debuggers disallow the debugging of optimized code. To give an example from the PC–Macintosh world:

> **Enable Debugging** turns off all optimizations. When the compiler optimizes it sometimes rearranges object code so that it does not correspond exactly to the source code. This rearrangement may confuse the debugger's source code view.

The lack of support for debugging optimized code has forced software developers to choose between either (1) enabling optimizations and foregoing the use of the debugger, or (2) compiling without optimizations so that a source-level debugger can be used when necessary. Because the ability to debug the production version of a software system is so important, the lack of support for debugging optimized code has discouraged some developers of large complex software systems from using optimizations. For instance, developers of the Coda file system [85] at Carnegie Mellon University do not enable optimizations during compilation, so that the system can be debugged in the field when necessary [90]. There are many instances, however, where debugging the unoptimized translation is unacceptable or debugging the optimized translation of a program is desirable:

1. In a production environment, it is desirable to use optimizations, and to debug the optimized production version of a program. The ability to debug optimized code is crucial to the software developer using optimizations, as he must ensure that the shipped version of a software system has been fully debugged. It is not sufficient to debug an unoptimized version of a program before enabling optimizations, because optimizations can change the behavior of a program. Bugs can surface when optimizations are enabled, even when compiler optimizations are correct. Optimizations change the data layout of a program and cause the program to execute faster; therefore, the debuggable translation of a program may mask a bug due to differences in storage layout and timing behavior [35]. For example, differences in timing behavior may cause the appearance of bugs due to race conditions. Or, differences in memory layout may cause pointer bugs that are unnoticed in the unoptimized version to cause a runtime error in the optimized version — or vice-versa.

2. Even if optimizations did not significantly change the behavior of a program, there are situations where it is impossible for the user to re-execute a crashed program under a source-level debugger, in which case all that is left for the debugger user is a core file. For instance, the sequence of inputs resulting in the runtime error may be irreproducible, or the program may have crashed in the field at a customer site and all that is available is a bug report and core file. In such situations, the program cannot be recompiled and re-executed under the source-level debugger; the program must be analyzed post-mortem.

3. Optimizations may be absolutely necessary to execute a program. Differences between the optimized and debuggable translations of a program can preclude the debuggable object code from running on a target platform — for example, because of memory limitations or constraints imposed on an embedded system.

In some situations where optimizations are necessary, developers have resorted to assembly-level debugging to debug their program. This scenario is common in the supercomputer world, where without optimizations applications can run for a prohibitively long time. Rather than deal with the problem of debugging optimized code, some supercomputer vendors have responded to the problem by providing debugger users with an

(a)	(b)	(c)
S_1: `d = f+g;`	S_1: `load R1,f`	I_1: `load R1,4(sp)`
S_2: `b = c*a;`	S_1: `load R2,g`	I_2: `load R2,8(sp)`
S_3: `*p = a;`	S_1: `fpadd Rd,R1,R2`	I_3: `store 0(R4),R3`
	S_2: `fpmul Rb,Rc,Ra`	I_4: `fpmul R4,R5,R3`
	S_3: `store 0(Rp),Ra`	I_5: `fpadd R6,R1,R2`

Figure 1.1: Example: The effects of instruction scheduling and register allocation on source-level debugging (a) Source code (b) Intermediate code (c) Object code

assembly dump window [91, 92, 110] — that is, a window that shows the instructions generated for the function being viewed at the source level. Optimizations make it very difficult for a user to debug a program at the assembly level. It is usually very tricky and time consuming — even for a compiler optimization expert — to look at optimized object code and relate individual instructions back to entities in the source code. Not only must the user take into account the net effect of cascaded optimizations, but he must also bridge the gap between assembly and source code[1]. For a software developer not versed in optimizing compiler technology, it can be impossibly difficult, tedious, and error prone to monitor a program at the assembly level, especially in the context of a large program.

The example of Figure 1.1 illustrates the difficulties introduced by two common code generation optimizations: global register allocation and instruction scheduling. Figure 1.1(a) shows a source code fragment and Figure 1.1(b) shows the translation of this code to an instruction sequence before register assignment and instruction scheduling. The instruction set in this example is that of a hypothetical load–store architecture; destination operands are listed first. All variables have been promoted to registers (shown symbolically as `Ra`, `Rb`, etc.) except for `f` and `g`, which are on the stack. Figure 1.1(c) shows the resulting object code after instruction scheduling and register assignment. Instruction scheduling has re-ordered and interleaved code from different statements, while register allocation has assigned the same register (`R4`) to two different variables (`p` and `b`). Note that register assignment has been done after instruction scheduling, since in Figure 1.1(b), `p` and `b` are simultaneously live at statement S_3's store instruction and thus cannot be assigned the same register.

By reusing registers that are assigned to source-level variables, register allocation com-

[1] Systems that generated code dynamically (e.g., [70] or [6]) are even more challenging to decipher.

plicates the debugger's basic task of retrieving the runtime value of a variable. In an unoptimized translation of a program, each variable is assigned a unique memory location (its home location); when the user queries a variable's value, the debugger can simply display the value in the variable's home location. Register allocation tries to minimize the use of registers by reusing registers as much as possible. This causes a problem for the debugger: if a variable has been allocated a register, there is no guarantee that the value in the variable's register is indeed a value of that variable. In the example of Figure 1.1, the variables `b` and `p` are both assigned register `R4`. If execution stops at or before instruction I_4, `b` has no runtime value, because `R4` holds a value of `p`. Similarly, if execution stops after I_4, `p` has no runtime value, because `R4` holds a value of `b`.

In the code of Figure 1.1(b), the boundaries between statements are clear and the debugger can map a breakpoint at a statement to the first instruction generated for the statement. Because of the reordering and interleaving introduced by instruction scheduling, it is difficult to decide where breakpoints should be set in the code of Figure 1.1(c). The execution of statement S_1 is overlapped with the execution of statements S_2 and S_3. If a breakpoint occurs at code for either statement S_2 or S_3, statement S_1 will not yet have completed execution. To further complicate matters, the three statements complete execution in reverse order from what is specified in the source. The debugger user will observe the side effects from the assignments in a completely different order than expected. In practice, it is impossible for the debugger to provide the illusion that the statements are executing in the order prescribed by the source, unless the debugger reorders the instructions back to their original source order.

Instruction scheduling and register allocation are standard optimizations that are employed in almost all state-of-the-art compilers; the preceding problems can be expected in any modern optimizing compiler. Other optimizations, such as code motion and dead code elimination, create additional problems for debugging.

This dissertation shows that accurate source-level debugging of fully optimized code is viable in practice. I demonstrate this in the context of a compiler that performs a complete set of global scalar optimizations for a commonly used source language such as C. The techniques presented in this dissertation do not constrain the optimizations performed by the compiler and do not instrument the debuggee to enable debugging. All interactions between the debugger and the user are in terms of the original source text using source variable names and expressions.

There are many approaches to the problem of debugging optimized code — in Section 2.4, I describe approaches that have been proposed and implemented in the past. Some approaches are *invasive*: the debugger requires modifications to the program or limitations on the transformations performed by the compiler. Source-level debugging can be made significantly easier if the compiler or debugger is allowed to insert additional code into the program before or after optimizations [93, 46], or if compiler optimizations are constrained so that problems do not occur [111, 52]. But such modifications or limitations no longer qualify a program as optimized and defeat the goal of debugging *optimized* code. In this dissertation, I assume that the debugger is *non-invasive* [2, 1]: the code generated by the compiler and debugged by the user is the default code generated with optimizations enabled. The compiler is not allowed to insert additional instructions into the object code to enable or simplify source-level debugging; I am interested in debugging the fully optimized version of a program.

To debug a program non-invasively, the debugger must somehow convey the effects of optimizations to the user. In general, it is impossible to make the effects of optimizations be transparent — that is, to hide the effects of optimizations from the user and to provide the same interactions as in unoptimized code. For example, in Figure 1.1, there are situations where in response to a user query for the value of variable `p` or `b`, the debugger must inform the user that no runtime value is available. The approach I present in this dissertation makes the effects of optimizations transparent whenever possible. Only when optimizations cannot be hidden from the user does the debugger inform the user of the effects of optimizations on the expected execution behavior of the program. Most of this dissertation is concerned with detecting the effects of optimizations on source-level debugging, and conveying the effects of optimizations to the user.

My research is oriented towards the debugging of imperative programming languages; I consider the C programming language as the source language of the debuggee. In this dissertation, I do not consider functional languages, although many of the techniques also apply to functional languages and their compilers. The compiler model used in this research is based on the traditional compiler organization used in most conventional industry and research compilers for imperative languages [7, 30, 71, 10]. The compiler first translates a source program into a machine-independent intermediate representation (IR). Classic machine-independent scalar optimizations — for example, code hoisting, dead code elimination, strength reduction, and so on — are performed on the IR. After optimizations, the code generation phase of the compiler maps the IR of the program into machine instructions and assigns machine resources, such as physical registers and functional units, to the instructions. Optimizations performed by the code generator include global register allocation and instruction scheduling (both local and global). In this dissertation, I consider the effects of transformations performed by both the machine-independent optimization and code generation phases of the compiler. I do not consider loop transformations such as interchange, skewing, reversal, blocking, splitting, unswitching, and so forth [12]. Such transformations have traditionally been used for vectorization and parallelization, and have only more recently been used in compilers for uni-processors. The effects of loop transformations on source-level debugging is left for future work; in Section 7.2, I suggest an approach for dealing with loop optimizations. This dissertation does not investigate debugging of parallelized code (debugging of parallelized code is the subject of Cohn's dissertation [33]).

In this dissertation, I concentrate on the low-level algorithms necessary to implement the core functionality of a source-level debugger. This core functionality includes retrieving and reporting variable values, and setting and reporting of breakpoints. This core functionality is affected by global optimizations; thus, I present algorithms that implement this functionality in the presence of optimizations — for example, algorithms for finding the latest runtime value of a variable, when register allocation has re-used storage locations assigned to variables. This dissertation does not deal with issues such as the debugger user interface, or the compiler–debugger interface — although these issues are important in the design of a debugger, problems relating to the core functionality of a debugger are more fundamental.

I have implemented all the algorithms I present in this dissertation in the context of the `cmcc` compiler (this compiler is described in detail in [5]). `cmcc` is a retargetable optimizing C compiler that I have implemented in collaboration with others as part of my research. The compiler is based on the compiler model outlined above and generates code that is roughly comparable in performance to the code generated by the native MIPS `cc` and `gcc` compilers on the DECstation 5000/200, and the native SPARC `cc` compiler on a SparcStation 20; Section 3.1 evaluates the performance of code generated by `cmcc` relative to code generated by these other compilers.

Structure of this dissertation

In Chapter 2, I present background material on source-level debugging and describe in more detail the problems caused by compiler optimizations. I also describe prior work in debugging of optimized code and the relationship of this dissertation to prior work.

In Chapter 3, I present the experimental framework used in this dissertation. I describe the `cmcc` compiler and the suite of programs used for the measurements presented in later chapters. I also discuss the quantitative methodology I used to gather the measurements in this dissertation.

In Chapters 4 and 5 I describe my algorithms for detecting the effects of optimizations on a debugger's ability to retrieve and report variable values from a program's runtime state. In these chapters, I also present measurements of how often a user may be affected by optimizations when querying a variable's value.

In Chapter 6, I describe in detail the problems relating to setting and reporting of breakpoints in optimized code. I present my approach to dealing with these problems as well as measurements of how often a user's ability to set breakpoints may be affected by optimizations.

Finally, in Chapter 7, I summarize the main contributions of this dissertation and suggest directions for future work.

Chapter 2

Background

In this chapter, I present background material on source-level debugging of optimized code. In Section 2.1, I describe a debugger model that is used as a basis in the rest of this dissertation. I concentrate on core debugger functionality and how optimizations affect this functionality. In Section 2.2, I illustrate with an example how optimizations affect debugger functionality. I also introduce terminology used when referring to variable values in optimized code. In Section 2.3, I discuss the importance of minimizing perturbations to the debuggee during debugging. In Section 2.4, I discuss prior work on debugging optimized code and the relationship of this dissertation to past approaches. In Section 2.5, I describe how source-level values can be managed by a debugger for optimized code.

2.1 Debugger model

The prime task of the debugger is to provide a source-level view of program execution; in this section I describe how the debugger accomplishes this task and the core functionality provided by the debugger. Additional functionality can be built based on the core set I describe here.

The debugger provides the user with the illusion that the source program is executing one statement at a time — on some abstract machine — in a manner defined by the operational semantics of the source language. The debugger allows the user to interact with the debuggee by providing mechanisms for controlling the execution of the debuggee and mechanisms for inspecting or changing the state of program execution. Examples of such mechanisms include functions for halting execution and printing the current value of a variable. All interactions between the debugger and the user are in terms of the high-level language program that is the source for the debuggee; thus variables are referenced by their user-given names, and breakpoints are set and reported at source statements or expressions.

There are three major aspects to the functionality of a source-level debugger:

1. The debugger allows the user to manipulate the execution of the debuggee by providing functions for suspending and resuming debuggee execution. *User interrupts* allow the user to suspend the execution of an executing program asynchronously (e.g., by typing `CONTROL-C` or hitting a `break` key). *Breakpoints* allow the user to specify conditions upon which debuggee execution is halted. *Control breakpoints* specify the breakpoint condition in terms of execution points in the source (e.g., halt when control reaches a particular statement). *Data breakpoints* specify the breakpoint condition in terms of source data objects (e.g., halt when a variable or memory location is referenced). Data breakpoints can be implemented in hardware, software, or a combination. The efficient implementation of data breakpoints is a topic of active research [61, 96, 97, 60, 37] and beyond the scope of this dissertation. A more general form of breakpoint, *conditional breakpoints*, suspends execution when a user specified predicate on the program's state becomes true. The clauses of the predicate can contain values of program variables, as well as references to either control or data breakpoints. The use of conditional breakpoints, as well as their efficient implementation, are an open research topic [61] and beyond the scope of this dissertation.

2. The debugger conveys to the user the control state of the halted debuggee in source-level terms by conveying which expressions have and have not executed. In unoptimized code this is easily done by pointing out the statement where execution has halted.

3. The debugger conveys to the user the data state of the halted debuggee in source-level terms. This is accomplished by allowing the user to query variable values.

The debugger is invoked as a result of a *break* that halts the execution of the debuggee. At a break, the halted program is either *suspended* or *terminated*. A suspended program is one that is halted as a result of a user request to intercept control — that is, a breakpoint or user interrupt. The user can resume the execution of a suspended program from the point where the break occurred. A terminated program is one that has been halted due to a runtime exception condition — for example, a memory access violation. If the debugger allows the user to modify program values at a break, then the user can try to resume the execution of a terminated program after first fixing the runtime value that caused the exception — for example, after changing the value of a nil pointer. In this dissertation, I do not allow variable modification; therefore, the user is not allowed to resume execution of a terminated program.

There are two ways in which the user can use the debugger: for *interactive* debugging and for *post-mortem* debugging. During interactive debugging, the user controls program execution by suspending and resuming the debuggee process. During post-mortem debugging, the debugger is invoked after the program has terminated, and the user cannot resume execution of the debuggee.

Breaks can be classified according to the runtime event causing the break:

- *Control breaks* suspend execution if the next instruction to be executed corresponds to a point in the source at which the user has set a control breakpoint.
- *Data breaks* suspend execution if the next instruction to be executed references a memory location at which the user has set a data breakpoint.
- *User interrupts* allow a user to suspend execution asynchronously. A user interrupt suspends execution at the instruction that was to be executed at the instant the user interrupt occured.
- *Faults* terminate program execution and occur if execution of an instruction causes a runtime exception (e.g., division by zero, overflow for fixed-sized arithmetic, segmentation fault, etc.). A fault terminates execution at the faulting instruction.

When the debugger is invoked, there exists a well-defined *stopping instruction*. A stopping instruction I is the instruction causing the break — that is, I is the instruction at which a control break occured; I is the instruction that caused a fault or data break; or I is the instruction to be executed when a user interrupt occured. At a stopping instruction I, all prior instructions have been executed (i.e., their effects are recorded), none of I's effects are visible (so I has not been executed), and none of the instructions subsequent to I have executed. If the target machine architecture exposes the effects of the individual operations of an instruction, then the definition of the stopping instruction can easily be extended to consist of an instruction I and the set of operations of I that have not been executed. Superscalar machines may allow instructions to complete out of order during execution of a program but usually provide *precise interrupts* [50] when an exception occurs. Recovering a precise machine state when the target machine's exceptions are imprecise is orthogonal to the topic of this dissertation. The techniques presented in this dissertation can be generalized to work in the context of imprecise interrupts as long as the interrupt-handling software can determine which instructions have completed execution (or *may* have completed execution).

When a break occurs, the debugger must convey the cause of the break in terms of the source program. In the source, breaks are distinguished according to whether they cause execution to halt *at* a statement boundary (i.e., at a point between two statements in the source) or *within* some statement S (i.e., during the execution of an operation within S). When a control break invokes the debugger, the debugger reports that execution has stopped *at* the statement S where the user has set a breakpoint. User interrupts, faults, and data breaks cause breaks during the execution of an operation *within* some statement, and are known as *asynchronous breaks*. When an asynchronous break occurs, the debugger maps the stopping instruction I, at which execution is halted, to the operation O in the source, for which I was generated. O is referred to as the *stopping operation*, and corresponds to the source operation where the fault, data break, or interrupt occured. When an asynchronous break occurs, the debugger reports that execution halted *within* the statement S containing O. The debugger may optionally report the source-level expression O to the user, depending on the granularity of the correspondences maintained by the compiler and debugger. In the case of either control or asynchronous breaks, the statement S is referred to as the *control reference statement*. Thus, a break is characterized by a pair $\langle S, I \rangle$, where S is the control reference statement, and I is the stopping instruction.

2.2 Debugging optimized code

Compiler optimizations complicate the task of the debugger by making it impossible for the debugger to provide the illusion that source statements are executed one at a time. To allow debugging, current compilers generate a debuggable translation of a program, in which boundaries between source code entities (e.g., statements, functions and variables) are preserved in the object code. More specifically, the ordering among source statements is preserved in the generated instruction sequences, and variables that are in scope at some statement S have an assigned storage location at the corresponding object code instruction(s) generated for S. In this manner, all that is required to implement debugger functionality is a mapping from each statement to an instruction, and each variable to a storage location. To implement a breakpoint at a source-level statement S, a breakpoint is set at the first instruction generated for S. Similarly, to implement source-level variable query and modification, a value in a runtime storage location is retrieved or modified. The task of the source-level debugger is simplified to that of mapping each statement to an instruction, and each variable to a storage location, using tables generated by the compiler. Because of the straightforward nature of the translation, these (one-to-one) mappings are easily generated by the compiler.

Transformations performed by optimizing compilers duplicate, eliminate, or reorder operations and values. As a result, the functionality of a source-level debugger is affected in the following ways:

- Because of optimizations, source execution points (i.e., source statement boundaries) are not well defined in the object code: code from a statement may be eliminated, moved, or merged with code from other statements. Consequently, optimizations make it difficult for a debugger to convey the exact control state of a halted program — for example, conveying which source-level expressions have completed execution, or which expression caused execution to halt. Moreover, providing source-level control breakpoints becomes difficult as mapping an execution point in the source program to an instruction in the object code becomes non-trivial. Similarly, mapping an instruction to the source code, in the case of an exception, becomes difficult. These problems are called *code location* problems [109].

- By reordering the execution of source expressions, and re-using runtime storage locations, optimizations make retrieval (modification) of source-level values from (in) the runtime state difficult (if not impossible) because some values may be either inaccessible in the runtime state [2] or inconsistent with what the user expects with respect to the source statement where execution has halted [1, 4]. These problems are called *data-value* problems [109].

In general, optimizations make it impossible for the debugger to provide the illusion that the source program is executing one source statement at a time, and that all in scope variables are available for inspection and modification. The goal of this dissertation is to extend the functionality of traditional debuggers with new source-level interactions that allow a user to manipulate or examine the state of an optimized program in a meaningful (and useful) way.

Support for debugging optimized code also complicates the implementation of the compiler: the compiler must track the correspondences between the source program and the intermediate representation, as optimizing transformations are performed. Moreover, as the mappings are no longer one-to-one, the compiler–debugger interface becomes more complicated.

2.2.1 Example: Global register allocation and instruction scheduling

To illustrate the difficulties caused by optimizations, let us consider again the source and object codes of Figure 1.1, shown again in Figure 2.1. The compiler has performed two common transformations: instruction scheduling and register allocation. To reduce memory accesses, the compiler has assigned registers `R3`, `R5`, and `R6` to variables `a`, `c`, and `d`, respectively. Variables `b` and `p` have both been assigned register `R4` since these variables have disjoint live ranges in the object code.[1] Variables `f` and `g` have not been assigned registers and they reside in the runtime stack. Upon entry to this block of code, `R4` contains the value of variable `p`. The value of `p` becomes dead after instruction I_3, and register `R4` (the register assigned to `p`) is assigned a value of `b` at instruction I_4.

[1]Note that register assignment has been performed after instruction scheduling since the assignment to `b` at S_2 and the last use of `p` at S_3 have been re-ordered in the object code. As a result of this reordering, these two variables having disjoint live ranges.

S_1: `d = f+g;`	I_1: `load R1,4(sp)`
S_2: `b = c*a;`	I_2: `load R2,8(sp)`
S_3: `*p = a;`	I_3: `store 0(R4),R3`
	I_4: `fpmul R4,R5,R3`
	I_5: `fpadd R6,R1,R2`
Source code (a)	Object code (b)

Figure 2.1: Example

To hide the latency of memory load instructions, the compiler has scheduled the instruction sequences from different source statements. Table 2.1 shows the correspondences between instructions, source expressions, and statements; it is easy to see how the instruction scheduler has interleaved and reordered the execution of instruction sequences from different statements. For example, the assignment to `d` from statement S_1 occurs at instruction I_5, while the assignments to `b` and `*p` of statements S_2 and S_3 occur at instructions I_4 and I_3, respectively.

Object Instruction	Source Expression Evaluated by Instruction	Source Statement
I_1	`f`	S_1
I_2	`g`	S_1
I_3	`*p = a`	S_3
I_4	`b = c*a`	S_2
I_5	`d = f+g`	S_1

Table 2.1: Execution order of code in Figure 2.1

Code location problems

To allow the user to halt execution at a source statement, the debugger must map a statement in the source to an instruction in the object where a breakpoint will be set. One obvious choice for mapping a breakpoint at a statement S is to map the breakpoint to the first instruction generated for S. Consider the case where the user sets a breakpoint at statement S_2 in the source of Figure 2.1(a). The debugger will map this breakpoint to instruction I_4. Note, however, that at I_4, statement S_3 has already completed execution while S_1 has not completed execution. This causes several anomalies during debugging:

1. The user may have set the breakpoint to halt execution before an exception at statement S_3's pointer assignment, but since S_3 completes execution before I_4, S_3 will fault before the breakpoint at S_2 is reached. It will appear to the user that the breakpoint at S_2 is missed.

2. After the breakpoint at S_2 is reached, the user may want to single step to statement S_3. Since S_3's breakpoint instruction has already been executed, the single step will not reach the breakpoint at S_3. It will appear to the user that S_3 is skipped by the single step.

3. If the user resumes execution from the breakpoint at S_2, and instruction I_5 subsequently causes an exception (e.g., floating-point overflow), the debugger will report an exception at S_1. It will appear to the user that execution proceeded backwards from S_2 to S_1.

The user expects that a breakpoint at a statement S suspends execution before any statement subsequent to S begins execution, and after all statements prior to S have completed execution. In unoptimized code, mapping a breakpoint to the first instruction of a statement achieves the desired effect since instruction sequences are neither overlapped, eliminated nor re-ordered. Moreover, the control state of a halted program can be precisely conveyed in the source: if a breakpoint or exception occurs during the execution of an instruction generated for a statement S, the debugger can report that execution has halted at or during the execution of S; the user is assured that all prior statements have completed execution and none of the subsequent statements have started execution. In optimized code, as the example above illustrates, mapping a breakpoint at a statement S to the first instruction generated for S may cause behavior that seems anomalous to the user. Due to the reordering and interleaving of instructions from different statements, there is no mapping from source statements to object instructions that will guarantee in general that breakpoints and exceptions are executed in order — this is evident in the example of Figure 2.1. Furthermore, the debugger can no longer accurately convey the control state simply by reporting that execution is at a particular statement.

The re-ordering and overlap caused by instruction scheduling also complicates keeping track of the correspondence between machine instructions and source lines. Rather than simply using labels that separate statements in the intermediate representation, the compiler must keep detailed information that maps each instruction back to an operation in the intermediate representation. Moreover, a single line number entry for each statement is no longer sufficient for reporting the statement where a fault occurs.

Data-value problems

In response to a query of a variable's value, a debugger typically retrieves and displays the value in the variable's runtime location. In unoptimized code, each variable has a unique runtime location; thus, runtime values exist for all variables that the user can query. Furthermore, because variable assignments are performed in the expected source order, a variable's runtime value will be identical to the value that the user expects the variable to have with respect to the statement at which execution has halted. In optimized code, on the other hand, the value in the runtime location of a variable — at a particular execution point in the object — may not be the value expected by the user. For example, register allocation can assign a single register to more than one variable; therefore, the runtime location of a variable may be holding the value of some other variable or a compiler temporary. Instruction scheduling can change the order in which assignments are performed; therefore, the value in the runtime location of a variable may not be an up-to-date value.

These data-value problems are illustrated with the example shown in Figure 2.1. If execution stops at instruction I_4 (e.g., due to a floating point exception), the debugger should report that execution has halted within statement S_2. At this instruction, the register assigned to `b` (register `R4`) is holding the value of `p`; therefore, no source value of `b` is available in the runtime state. The debugger will clearly mislead the user if it naively displays the value in `R4` in response to a query of `b`'s value. At statement S_2, the user expects `d` to have the value assigned by statement S_1. However, this assignment to `d` has not yet executed (it executes at I_5), and the value in `d`'s runtime location (i.e., the value in `R6`) is the value from the last executed assignment to `d` prior to this block. Therefore, the value in `d`'s runtime location (although it is *some* source-level value of `d`) does not correspond to the value that the user expects `d` to have. Again, the debugger may mislead the user if it displayed this unexpected value of `d`, without further qualification.

Sometimes, it is not possible to determine whether the value in the runtime location of a variable is an up-to-date value. For example, if the program stops at I_4, the assignment to `*p` of S_3 has executed prematurely at instruction I_3. Therefore, the memory location M, to which `p` points, also contains a value different from the one expected by the user. M may be the runtime location assigned to a variable, or it may be part of a dynamically allocated heap object. We can determine exactly which location M is not up to date, since the value of `p` that was used by the prematurely executed assignment of S_3 is available in `R4`. But consider stopping at instruction I_5, reported at statement S_1. The assignment of S_3 has again executed prematurely, but the value of `p` used by S_3 is unavailable because `p` itself is no longer available (`R4` now holds the value of `b` assigned at statement S_2). Therefore, we cannot determine the memory location M that is modified and must conservatively assume that the value of any variable with a runtime location in memory — that is, `f` or `g` — may not correspond to the value the user expects.

Discussion

The anomalous and unexpected behaviors described above are typical of what can be observed if a program is compiled with optimizations and subsequently debugged using current compilers and debuggers. Such behavior is clearly of limited use in debugging a program because the values and program points reported by the debugger do not accurately describe the program state to the user. The design goal for a debugger of optimized code should be to extend the functionality of a traditional debugger so that it can provide interactions that will guide the user in debugging optimized code.

2.2.2 Values in optimized code

Most research on debugging optimized code concentrates on the data-value problem and much terminology exists for this aspect of the problem [49, 2, 1, 36, 106]. In this section, I review this terminology.

At a break, the debugger conveys the state of source variables by allowing the user to query variable values. The control reference statement acts as a reference point in the source for reasoning about these values: the value that the debugger displays to the user is implicitly qualified as being the source-level value *at* the control reference statement. The value that a user expects a variable V to have, relative to a statement S, is V's *expected* value at S. For example, at statement S_3 in Figure 2.1, the expected value of `d` is the value assigned by statement S_1. Note that in the presence of a bug the expected value of a variable V may not be the value the user intended V to have. We are concerned only with the value the variable is supposed to have with respect to the source program, not with the value the intended for the variable. The debugger can only show the expected value of a variable; the user must make the determination whether the expected value of a variable is the intended value.

When a program is translated, each source variable V is assigned a runtime storage location (e.g., a register, or a slot in the activation record). In response to a query of V's value, the debugger can retrieve and display the value in this location. The value in V's runtime location at a stopping instruction I is V's *actual* value. In the example of Figure 2.1, at instruction I_3 the actual value of `d` — that is, the value in register `R6` — is the value assigned by the last assignment to `d` before this block of code.

In unoptimized code, the actual value of a variable is always identical to the variable's expected value. As illustrated in Section 2.2.1, however, several problems arise in optimized code. First, because of transformations such as global register allocation, there may sometimes be no runtime location holding a variable V's value at a break; consequently, no actual value of V may exist. For example, if a variable V has been assigned a register R, and the stopping instruction lies outside of V's live range, then R may be holding the value of some other variable (or temporary) at the break. In the example in Figure 2.1, `p` has no actual value at stopping instruction I_5, since `p`'s assigned register `R4` is holding the value of `b` (assigned at I_4) at this break. If, at a break B, the storage location assigned to a variable V is holding the value of some other variable, then V is *nonresident* at B [2]. If V is resident, the storage location where V is accessible is called V's *residence* and the value in V's residence is V's actual value [35].[2] The fourth column of Table 2.2 lists the nonresident variables at breaks in the code of Figure 2.1. The first three columns of this table again show the correspondences between instructions, source expressions, and statements. Note

[2]There are also situations where a variable may be resident in more than one location (e.g., due to live range splitting or renaming); or where a variable is eliminated by optimizations, and thus, has no runtime location anywhere in the program (e.g., due to value propagation and dead code elimination).

that residency depends only on the stopping instruction and is independent of the control reference statement.

Object Instruction	Source Expression Evaluated by Instruction	Source Statement	Nonresident Variables	Endangered Variables		
				Noncurrent		Suspect
I_1	`f`	S_1	`b`			
I_2	`g`	S_1	`b`			
I_3	`*p = a`	S_3	`b`	`d`		
I_4	`b = c*a`	S_2	`b`	`d`	`*p`	
I_5	`d = f+g`	S_1	`p`		`b`	`f,g`

Table 2.2: Endangered variables at breakpoints in the code of Figure 2.1

Even if a variable V is resident, the actual value of V may not correspond to the expected value of V. If the actual value of a variable V at a break B is identical to the expected value of V relative to the corresponding control reference statement, then V is *current* at B. Otherwise, if the actual value is not identical, then the variable is said to be *noncurrent* [49]. In Figure 2.1, `d` is noncurrent at break $\langle S_2, I_4 \rangle$. If it cannot be determined with certainty whether a variable is noncurrent, then this variable is called *suspect* [1]. At break $\langle S_1, I_5 \rangle$, both `f` and `g` are suspect: S_3 has prematurely executed, but the value of `p` is nonresident, so we cannot determine which memory location has been prematurely modified. In the absence of other information — for example, alias analysis information — we must make the worst-case assumption that any variable stored in memory can be noncurrent.

In general, noncurrent and suspect variables are referred to as *endangered* variables; that is, an endangered variable is a variable whose runtime value *may* not correspond to the variable's expected source value. (An endangered variable is either noncurrent or suspect, but not both.) Endangerment does not pertain to a nonresident variable, since such a variable does not have an actual value. The last two columns of Table 2.2 list noncurrent and suspect variables at breaks in the code of Figure 2.1. Note that endangerment depends on both the stopping instruction (which determines a variable's actual value) and the control reference statement (which determines a variable's expected value).

2.3 Perturbations caused by the debugger

To implement certain debugger features such as breakpoints, the debugger must sometimes modify the object code to realize a desired effect (e.g., stopping when machine instruction I is the next instruction to execute). In addition, a debugger or compiler can effect changes in the debuggee's address space to facilitate debugging. In this section, I discuss the importance of limiting or prohibiting changes in the debuggee's address space in the implementation of a debugger for optimized code.

By effecting changes in the debuggee's address space, the debugger can perturb the execution of the debuggee. However, the reasons for debugging *optimized* code require that the debugger modify the object code as little as possible. Perturbations in the storage layout and timing behavior can mask a bug, or prevent execution on a target platform (because of a size increase of either the object code or the data segment). Even if perturbations of the debuggee's address space do not prohibit execution on a target platform, program execution will slow down.

There are two types of modifications that a debugger can effect on a program's address space: changes of the data layout and changes of the code segment. Changes to the data layout should be avoided; references to illegal storage locations or undefined variables are among the most difficult bugs to find, and if the debugger modifies how the user variables and compiler temporaries are stored in memory, then the debugged program will differ significantly from the program run without the debugger.

Modifications to the code segment may be necessary to implement breakpoints. If the target architecture does not contain a "breakpoint" register (such a register contains an instruction address, and the runtime system is invoked when this address is to be executed next) or a break bit in the instruction (execution stops when an instruction with the break bit set is decoded), then code modification is unavoidable. A common approach is to use *code patching* to implement breakpoints [61] — for example, to implement a control breakpoint at an instruction I, I is replaced by a jump to a code patch that contains a trap instruction followed by I. In the case of control breakpoints, code patching does not increase the number of instructions executed, since each code patch transfers control to the debugger and is thus part of the context switch overhead.

Address-space perturbation is one aspect of the invasiveness of a debugger. A related issue is whether modifications to the object code are done at compile time or at debug time. Code patching can be performed either by the compiler tool chain before the program is run (e.g., as performed in [97] to implement data breakpoints), or by the debugger when the program is being debugged (e.g., to implement control breakpoints [61]). If modifications are done at compile time, the resulting code no longer qualifies as optimized (in the sense that this is the best code that can be generated for a given program). If the debugger is not invoked, then the execution has been unnecessarily slowed down. In practice, this situation tempts users to avoid the extra modifications and it is unlikely that the production version of a program will contain additional debugging code. Hence, the production version cannot be debugged (either interactively or post-mortem) and must be re-compiled for debugging. An extreme case is the use of a special flag to enable debugging, which avoids the issue of debugging optimized code.

I call a debugger *non-invasive* if it does not require modification to the data layout, and requires modifications to the code segment only to implement control breakpoints at debug time. Otherwise, I call the debugger *invasive*. Since perturbations caused by an invasive debugger are undesirable for reasons similar to those motivating the debugging of optimized code, this dissertation concentrates on non-invasive debuggers. Consequently, the compiler is not allowed to insert extra code to make debugging easier, or to ease retrieval of values — for example, the compiler is not allowed to re-order code back to the original source order or to insert code to save values before they are lost. The code generated by the compiler must be the same as the code generated otherwise. More specifically:

- I preclude the use of instrumentation code to enable or aid debugging. I will show in Section 5.5.1, that instrumentation code can help the debugger by providing runtime information that enables the debugger to eliminate suspect variables caused by code hoisting and dead code elimination. However, not only does instrumentation code perturb execution (thus possibly masking a bug), it also makes the code non-optimal and is thus unlikely to be part of the production version of a program. Requiring instrumentation code does not completely address the problem since the user is still faced with the choice between a fully optimized translation of a program and a debuggable but non-optimal translation.

- I preclude the insertion of *hidden breakpoints* [108] by the debugger; hidden breakpoints are inserted into the program to collect runtime information at key execution points in the program. This technique may be helpful in some interactive debugging sessions where execution speed is not an issue; but in general, the high cost of context switching between the debugger and the debuggee greatly perturbs timing behavior. Moreover, hidden breakpoints do not help in the case of post-mortem debugging.

- I preclude techniques that limit or constrain optimizations to allow debugging. For example, nonresident variables can be avoided if the compiler stores variables to memory at the end of live ranges. Endangered variables caused by instruction scheduling can be avoided by limiting the scope of instruction scheduling to instructions within individual source statements. Tradeoffs between optimizations and debuggability are left for future work.

2.4 Alternative approaches to debugging optimized code

There are a number of approaches that can be taken in the design of a source-level debugger for optimized code. In this section, I describe approaches that have been proposed or implemented in the past and relate these prior approaches to the one presented in this dissertation. I identify four general approaches to the problem of debugging optimized code: The first approach ignores the fact that optimizations have been performed, resulting in the anomalous behaviors described in Section 2.2.1. The second approach tries to hide all optimizations from the user, and to provide the exact functionality and behavior as expected from unoptimized code. In general, this approach cannot be applied non-invasively. The third approach exposes optimizations to the user, leaving the user with the task of sorting out the source state based on runtime values in the optimized program. The fourth approach — which is the approach taken in this dissertation — detects when optimizations can be made transparent and exposes optimizations to the user only when necessary; when optimizations must be exposed to the user, this approach manages the effects of optimizations by relating runtime values in the optimized program to source-level values in the original program.

2.4.1 Ignoring optimizations

There are some debuggers that allow debugging of optimized code but ignore the effects of optimizations. The compiler produces symbol table information that maps statements and variables to instructions and runtime locations in the optimized translation. The debugger interprets the symbol table information in the usual manner. Neither the compiler nor the debugger consider the effects of optimizations on the expected execution; for example, the debugger retrieves the value in the runtime location of a variable without regard to whether the variable is resident. These systems usually warn the user beforehand (in their documentation) that anomalous behavior will occur, or that some responses may be inaccurate during debugging of optimized code.

As the example of Section 2.2.1 illustrates, such an approach results in behavior that seems anomalous to the user. This approach is inadequate because the user can never be sure whether the displayed value of a variable is the expected value, or whether a breakpoint will be executed as expected. For an approach to be useful, the debugger (in conjunction with the compiler) must either (1) somehow hide the effects of optimizations by presenting the same behavior as when debugging an unoptimized program, or (2) take optimizations into account by detecting and conveying to the user the effects of optimizations on source-level debugging. Otherwise, the program points and values reported by the debugger will mislead the user.

2.4.2 Providing expected behavior

Ideally, we would like the debugger to hide the effects of optimizations by presenting to the user the expected values of variables and by providing the illusion that source statements are executed in source order. That is, we would like the debugger to present *expected* behavior [108] to the user: the debugger provides the same behavior when debugging the optimized version of a program as when debugging the unoptimized version.

To provide expected behavior without constraining optimizations or debugger functionality, the debugger must detect all nonresident and endangered variables, and recover their expected values. Moreover, the debugger must ensure that events, such as breakpoints and exceptions, occur in the order prescribed by the source (and expected by the user). However, it is not always possible to recover the values of endangered or nonresident variables, or to guarantee that events occur in source order, without constraining the optimizations performed by the debugger or inserting code to make expected behavior possible (e.g., by inserting code to store a register assigned variable to memory at the end of the variable's live range, even though the variable is dead). Therefore, providing expected behavior non-invasively is generally not feasible when debugging the optimized translation of a program.

Several approaches have been proposed that provide expected behavior at the expense of limiting the scope of optimizations [52, 111, 46, 93]. These approaches sacrifice optimizations for debugging and emphasize interactive debugging over post-mortem debugging. These approaches use a combination of techniques:

- The program points where the debugger can be invoked are limited and optimizations are constrained so that debugger functionality can be provided at these points [83, 52, 111, 46]. The compiler optimizes code between invocation points in a manner that allows the user to query program state when the debugger is invoked; for example, the compiler guarantees that all variables that are in scope at an invocation point are current. In the limit, this technique constrains the scope of compiler optimizations to individual source statements, thus allowing full debugging with limited optimizations.

- Instrumentation code is added before optimizations to collect runtime information for answering debugging queries. Instrumentation code can be placed at a few program points specified by the user [46] or at all source statements [93]. Optimizations do not interfere with debugging because the instrumentation code is inserted before optimizations.

- In response to a user request, the region of code affected by the request is incrementally recompiled by the debugger. Either new instrumentation code is inserted to implement the user request [46] or a function is deoptimized, allowing the user request to be carried out within that function [83, 52, 111].

In the following paragraphs, I first present each approach, and then discuss the limitations of the techniques.

Pollock and Soffa [83] propose an incremental compiler to disable optimizations that prohibit the debugger from satisfying a user request. The debugger user specifies all requests before program execution and the debugger incrementally recompiles the program in a manner that allows the requests to be satisfied. A program representation is described that keeps track of optimizing transformations and allows the compiler to derive the unoptimized version of the program after optimizations. No implementation of this proposal exists.

Holzle, Chambers and Ungar [52] describe the approach to debugging optimized code used in the SELF compiler system [27], an optimizing compiler for the object-oriented language SELF [94]. This approach restricts the program points at which the debugger can take control to discrete *interrupt* points. Optimizations are constrained so that the complete source-level state can be reconstructed at each interrupt point. Initially, interrupt points are at function prologues and at the end of each loop body; therefore, the debugger can be invoked only at a subset of the source statements. When the debugger is invoked at an interrupt point, the function (i.e., the SELF method) containing the interrupt point is deoptimized by the debugger (if it is not deoptimized already) so that debugger functionality such as single stepping and variable query can be provided; this is referred to as *dynamic deoptimization* [52]. Once a function is deoptimized the debugger may be invoked at any source point within that function. The SELF system generates code incrementally for each function at runtime, thus facilitating the use of dynamic deoptimization.

Zurawski and Johnson [111] describe a similar approach used in the TS compiler [58], an optimizing compiler for Typed Smalltalk. Like the SELF system, the debugger can be invoked only at pre-determined points that in the TS system are called *inspection points*. Inspection points are source points where the user can set a breakpoint, points where an exception may occur, or function call sites where the debugger may be invoked before the function returns. Optimizations are constrained so that the expected source-level state can be reconstructed at each inspection point. Recovery information is generated by the compiler and used by the debugger to map runtime state to the source-level state, at each inspection point. The compiler performs an optimization only if it can produce recovery information for each inspection point affected by the optimization. To allow modification of variable values, a function is converted to its unoptimized form at debug time (the TS compiler also performs incremental re-compilation).

Gupta [46] proposes a technique for debugging code transformed by trace scheduling compilers [43, 71]. During debugging, the user specifies commands for monitoring values and conditions at various points in the program. Code is then added to the program to collect the monitored values and the parts of the program containing the new monitor commands are incrementally recompiled. The units of re-compilation are program traces used by the compiler's trace scheduler; therefore, the compiler's trace scheduler must be integrated with the debugger. No implementation of this proposal exists.

Tolmach and Appel [93] present an approach to source-level debugging used in the Standard ML of New Jersey (SML-NJ) compiler [9], an optimizing compiler for Standard ML [75]. SML-NJ instruments the source program so that information is gathered at runtime in support of debugger queries. The instrumentation code is added to the abstract syntax tree representation of the source and is transformed along with the rest of the program by subsequent transformations. Although optimizations do not affect debugging in their approach, their technique seems motivated mainly by the fact that their compiler transforms the code through multiple representations, lambda-calculus, continuation passing style [8] and then assembly code, making the tracking back to the source code very difficult.

The techniques used by the above approaches have the following disadvantages:

- Constraining optimizations or adding instrumentation code do not solve the problem of debugging the optimized translation of a program; the user debugs a partially optimized translation of the program. A special switch is still needed in the compiler to enable debugging and the user is again faced with a choice between enabling full optimizations and enabling debugging.

- Limiting the set of program points where the debugger can be invoked constrains interactive debugging and precludes post-mortem debugging if execution terminates at a point where the debugger cannot be invoked. Similarly, deoptimization precludes post-mortem debugging if execution terminates in an optimized part of the program. From a user's perspective, it is valuable to have the ability to perform both full interactive and full post-mortem debugging. Post-mortem debugging is especially important in situations where either the user cannot re-execute the failed program, or it is inconvenient to repeatedly execute the failing program interactively until the cause of the fault is determined via breakpoints.

- Incremental re-compilation re-translates portions of the code at debug time, and is thus practical only in a programming environment where the compiler and debugger are closely integrated. Moreover, incremental re-compilation precludes post-mortem debugging: the user cannot use the debugger if execution aborts in a region of code that needs to be recompiled for debugging.

2.4.3 Exposing optimizations to the user

Since it is generally not possible to provide expected behavior non-invasively, the debugger must expose to the user the effects of optimizations on the expected execution. This exposure should be in terms of the source program (after all, we are interested in *source-level* debugging). One approach is to convey the control state of the program by exposing to the user the order in which source expressions actually execute in the object. Based on this information, the user can reason about the values of source variables. The debugger addresses both the code location and data-value problems by exposing them to the user; the user is left with the task of sorting out how optimizations have affected the state of source variables at a break.

In his Master's thesis [73], Lyle proposes to expose optimizations to the user by displaying a modified version of the source where operations have been re-ordered and eliminated to reflect the final ordering of operations after instruction scheduling. This is a difficult task because many transformations such as register allocation and software pipelining are difficult to express in source terms. Lyle's thesis specifically addresses scheduling for a VLIW machine. No implementation of this proposal exists. For source-to-source transformations (e.g., loop interchange) this approach may be feasible since such transformations are usually performed on a high-level (almost source-level) representation of a program, and can thus be easily reflected in the source.

The CXdb debugger [20] is an example of an alternative way to implement the exposed approach. CXdb animates execution by highlighting source expressions as the user single steps at the object level. CXdb only highlights the expression that is about to be executed and does not provide a control reference statement. No endangerment information is provided to the user; this is consistent, since no control reference statement is provided either. Highlighting alone does not allow the user to determine which variables are nonresident; therefore, CXdb partially addresses the data-value problem by addressing the residence problem only. A resident variable is implicitly suspect — it is up to the user to determine (using information discerned from execution animation) how the runtime value of a variable relates to the source value. The CXdb approach of only detecting nonresident variables is the most conservative behavior that can be expected of a debugger for optimized code. It is important that a source-level debugger provide at least this level of support for dealing with data-value problems, otherwise the user may be misled by the value displayed by the debugger[3] — that is, the user must be sure that the value displayed by the debugger is at least *some* source-level value of the queried variable.

The CXdb approach is applicable to a wide range of optimizations since the compiler needs only to maintain correspondences between source expressions and instructions at a very fine granularity. No analysis is required to detect how the actual values of source variables differ from their expected values since the user is responsible for determining how actual values relate to source-level values.

The exposed approach, and the CXdb approach in particular, is unsatisfactory for a number of reasons:

1. The burden of determining how the expected source-level state has been affected by optimizations is placed on the user. If the user is not versed in compiler optimizations, then this burden is unacceptable.

2. Visual annotation is useful only if the user interactively single-steps through the code; for each step, the source expression being executed is illuminated. It is not useful for post-mortem analysis or if a breakpoint is reached during the execution of a program: the debugger highlights the expression at which execution halts, but does not provide execution history, since the program was not single stepped.

3. Animation of some optimizations may not be easily expressed in source terms, in a manner that is easily understood by the user. For example, consider software

[3] Some commercial debuggers that claim to provide support for optimized code, naively display the value in a queried variable's storage location without detecting whether the variable is resident[35].

pipelining where the execution of multiple loop iterations are overlapped; highlighting alone cannot convey the iteration index of an operation that is being executed.

2.4.4 Managing the effects of optimizations

Even though hiding the effects of optimizations is generally not possible, there are still many situations where optimizations do not affect debugging and where the debugger can provide expected behavior to the user. When the debugger determines that optimizations need to be exposed to the user, the debugger can still manage the effects of optimizations by guiding the user in understanding how optimizations have affected the expected source-level state. In this manner, the burden of analysis is shifted to the debugger, and less knowledge about optimizations is required on the part of the user. This approach, however, increases the complexity of the debugger implementation because it requires the debugger to perform analysis to determine when and how optimizations affect source-level debugging.

Most of the prior work on detecting the effects of optimizations has focused on data-value problems; very little work has been done on managing code location problems. To manage the data-value problem, the debugger can detect the set of variables that are nonresident or endangered, and either report them as such in response to a user query, or attempt to recover their values, although recovery may not always be successful. The recovery strategy has an influence on how often the debugger is able to present the expected value to a user, but in either case, the debugger is never allowed to present erroneous data to the user. If a variable is endangered, the debugger displays the actual value of the variable with a message qualifying the variable as endangered. Zellweger [108] refers to debuggers that detect endangered variables as exhibiting *truthful* behavior. The debugger can provide additional guidance by conveying how optimizations have affected source values. For example, the debugger can tell the user at which source assignment(s) an endangered variable's actual value was (or may have been) assigned [3].

Several researchers have concentrated on the problems of detecting nonresident variables [2] and endangered variables [49, 1, 36, 106, 4]. In the following paragraphs, I summarize each of these prior works.

Hennessy [49] introduced the concept of endangered and noncurrent variables, and presented the first algorithms to detect endangered variables. It is difficult to extend the results of his work to modern compilers, however, because of limitations in the source-language and compiler used in his work: the source language is a subset of Pascal that does not include pointers, and the optimizations performed by the Pascal compiler are only at the machine-independent level. Therefore, the algorithms described by Hennessy in [49] (and corrected by Wall et al. in [99]) deal with noncurrency due to local dead code elimination, and re-ordering introduced by local common subexpressions that are assigned to variables; the algorithms do not consider effects of code generation optimizations such as instruction scheduling and register allocation. Hennessy also presents algorithms to recover the expected values of endangered variables; but these recovery algorithms recover values only from memory and do not consider partially computed results that are available in registers.

DOC [38] is a prototype debugger developed at HP, designed to demonstrate the feasibility of debugging optimized code. The optimizations that DOC handles are instruction scheduling, register allocation, constant propagation, and induction variable elimination. DOC can provide expected behavior for control breakpoints (control breakpoints are always executed in the expected source order), and can detect and warn the user of nonresident and endangered variables at breaks. DOC does not attempt recovery of endangered variables. To ease handling the problem of debugging optimized code, DOC makes two simplifications: First, DOC does not deal with global optimizations such as code hoisting and global dead code elimination. Second, DOC restricts breaks to only control breaks at pre-determined statement boundaries. Allowing interactive use only (i.e., allowing source-level control breakpoints and variable query only) can greatly simplify the implementation of a source-level debugger for optimized code, since the compiler and debugger know a priori the points at which execution will be halted in both the source and object codes: the compiler can pre-compute the set of endangered variables at each control breakpoint and pass this information on to the debugger. DOC cannot accurately report endangered variables if an asynchronous break occurs: it does not model what happens inside a statement and cannot precisely detect which variables are endangered at an asynchronous break. DOC's compiler first marks the set of valid break instructions corresponding to statement boundaries before code scheduling, and then computes the set of endangered variables at each valid break instruction, by detecting store instructions that were moved across statement boundaries by the code scheduler. This information is passed on to the debugger.

Copperman [35, 36] and Wismueller [107, 106] have investigated the data-value problem of detecting endangered variables caused by global optimizations. Their efforts concentrate only on detecting if a variable is endangered; they do not consider code location problems in any depth.

Copperman's approach to detecting endangered variables [35, 36] is based on a specialized data-flow analysis of an intermediate representation of the program. This representation captures the effects of optimizing transformations but does not consider nonresident variables. Copperman does not consider language issues such as undefined evaluation order. Moreover, Copperman's approach does not deal with asynchronous breaks and does not consider recovery. Copperman's techniques have not been implemented; therefore, it is difficult to evaluate the practicality of his techniques.

Wismueller [107, 106] provides a formal framework for detecting current variables. His algorithms concentrate only on detecting whether the expected value of a variable can be displayed to the user; he does not distinguish between nonresident, suspect, and noncurrent variables. Wismueller's algorithms depend on a specialized intermediate representation that is separate from the IR used by the compiler. This specialized IR must be maintained in parallel by the compiler as transformations are performed. Wismueller has implemented his algorithms in the context of the SUN C compiler, but his implementation does not include instruction scheduling.

The algorithms of Copperman and Wismueller are similar to some of the algorithms I present in this dissertation, to the extent that they also use data-flow analysis. There are, however, two key differences between my work and their's that have allowed me to develop simpler and more practical algorithms. First, of the many transformations that are performed by an optimizing compiler, only a few cause problems for the source-level debugger, namely transformations that move or eliminate assignments; my algorithms concentrate only on these transformations. Second, there are a number of invariants that are preserved when compilers transform programs — compilers do not perform arbitrary transformations; my algorithms take advantage of the invariants maintained by transformations that move or eliminate assignments. For example, if an assignment is hoisted to a different basic block, this basic block is post-dominated by the original block; this limits the range of breaks where a variable is endangered because of the hoisted assignment. Or, if an assignment is eliminated because of backward redundancy, the value must be available somewhere, and the debugger can provide this value to the user. Both Copperman [36] and Wismueller [106] assume arbitrary code movement and elimination, and fail to recognize that movement and elimination are not unconstrained.

Another major difference between this dissertation, and the earlier work by Copperman and Wismueller is that they attempt to capture a "summary effect" of all optimizations in auxiliary intermediate representations. They treat the compiler as a "black box" that takes a source program as input and produces object code along with the auxiliary intermediate representations. The auxiliary intermediate representations capture the source program both before and after optimizations; their algorithms then determine currency by comparing the optimized code with the original unoptimized source code.

In contrast, the approach I describe in this dissertation models each optimization step inside the compiler by annotating the single IR used by the compiler. Only the single representation used by the optimizer is necessary and the compiler propagates information about the effect of optimization steps through all optimization phases; when the program representation is lowered the annotations are transferred to the new representation. The final object code contains annotations that allow the debugger to determine nonresident, noncurrent, and suspect variables. In Section 5.7, I will discuss this issue in more detail.

2.5 Managing values in optimized code

From the viewpoint of the user, nonresident and endangered variables are similar in that the debugger cannot display the variables' expected source value. However, endangered and nonresident variables are different with regard to the information that the debugger can provide the user. An endangered variable has a value that may be inconsistent with what the user expects, whereas a nonresident variable has *no* value. That is, the value in an endangered variable's runtime location is some source-level value of the variable, but it may not be the value expected by the user. Therefore, since the value has *some* meaning in the source, it is helpful to the user if the debugger can convey what source value an endangered variable's value may correspond to. In the case of an endangered variable V, the debugger can provide additional information to the user by displaying V's actual value, and attempting to explain what value is being displayed. For example, consider the break $\langle S_1, I_5\rangle$ in Figure 2.1. At this break, the assignment to `b` of statement S_2 has executed early at I_4. In response to a user query of `b`, the debugger can display the value in register `R4` (`b`'s actual value) and explain to the user that the displayed value is the value of `b` assigned at S_2 because optimizations have caused statement S_2 to execute early.

In the case of a nonresident variable, however, no actual value exists that can be presented to the user. For example, at the break $\langle S_3, I_3\rangle$, `b` is nonresident because its assigned register `R4` is holding the value of `p`. This value has no relation to `b`, and therefore, will not be helpful to the user. The debugger informs the user that `b` is unavailable.

When the user inspects a variable, the variable's expected value may be immaterial because the variable has not been initialized during the execution of the program. Thus the question of whether an uninitialized variable V is resident or current is irrelevant, since V has no expected value. Detecting and reporting uninitialized variables reduces the number of variables that are reported as nonresident or endangered, and provides additional information to the user [2].

In the absence of support provided by the runtime system (e.g., path determiners [108]) or the architecture (e.g., memory tags), detecting uninitialized variables requires that the debugger perform program flow analysis on the source program. If no definition of a user variable *V* reaches a point *S* in the source, then *V* is uninitialized whenever the program breaks at *S*. In the case that definitions reach on some but not all paths to *S*, the debugger conservatively reports that V is initialized.

Referring back to the example of Figure 2.1, if no source assignment of `b` reaches the block of code, `b` can be reported as uninitialized rather than nonresident or endangered at any break reported at S_1. In this example, these are breaks that occur at I_1, I_2 and I_5.

Figure 2.2 illustrates the relationship between nonresident, endangered, and current variables. Each oval in this diagram represents one of four possible states of a queried variable: nonresident, suspect, noncurrent, and current. The arrows in this diagram show the direction in which the response from the debugger becomes more precise (and useful) for the user. In the worst case, the debugger cannot show any value in response to a user query — that is, the queried variable is nonresident. If a queried variable is resident, then a source-level value exists for the variable and this value can be displayed to the user. However, we would like the debugger not only to show the actual value of a variable, but also to provide additional information describing how the actual value relates to values in the source. The debugger can report conservatively that the variable is suspect and leave it up to the user to determine how this value relates to the source — that is, the debugger can display the actual value to the user with a warning that the value may not be the expected value, independent of the source reference statement; this is the approach taken by CXdb. Or, the debugger can qualify whether or not the actual value is the expected value with respect to a source statement — that is, the debugger can display not only the actual value to the user, but also inform the user whether the variable is current or noncurrent with respect to the control reference statement; this is the approach taken by DOC. Because of ambiguities due either to multiple paths reaching a breakpoint [3], or to pointer assignments that are executed out of order [1], the debugger may not always be able to determine whether a resident variable V is current or noncurrent; therefore, there are situations where the best the debugger can do is to report V as suspect.

The debugger can attempt to improve the response given to a user and determine whether a suspect variable is really current or noncurrent by using runtime values — that is, values in registers — to detect which path was taken to a break and which memory locations were updated by assignments that were executed out of order. To provide the expected value of a noncurrent variable V, the debugger can also use runtime values to construct V's expected value. For example, at break $\langle S_2, I_4 \rangle$ in Figure 2.1, the expected value of `d` is the value assigned by S_1, and computed by I_5. The values of I_5's source registers (`R1` and `R2`) are computed at instructions I_1 and I_2, and are thus available at I_4; therefore, the debugger can provide the expected value of `d` by interpreting I_5. This process is called *recovery* [49]. When attempting recovery, the debugger must be prepared to handle the case where the interpreted instructions cause an exception. Also, to allow execution to be resumed, the debugger must not overwrite runtime values during recovery.

Currency determination qualifies the value of a variable by indicating whether the actual value of the variable is the same as the expected value with respect to the control reference

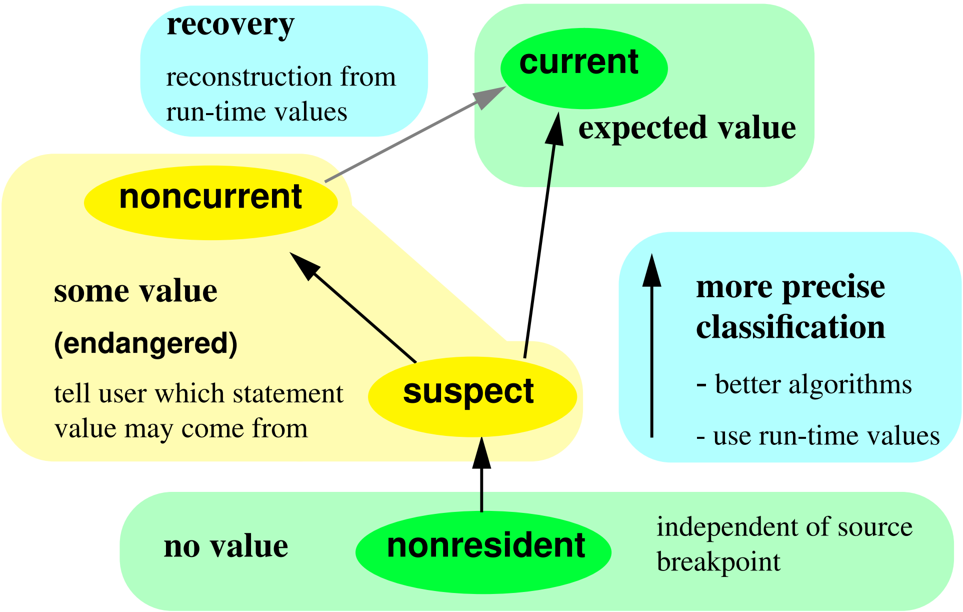


Figure 2.2: The data-value problem.

statement. An alternative way to qualify the value of a variable is to convey to the user which statement(s) may have assigned the actual value of the variable. This is similar to program slicing and can be accomplished using reaching definitions analysis to determine the set of assignments that may have affected a variable's value [103, 104]. The reaching definitions analysis depends only on the stopping instruction, and in contrast to noncurrency determination, does not require the existence of a control reference statement. The reaching assignments can be communicated to the user by some method of marking up the source (e.g., highlighting). This method of qualifying a variable's actual value by finding reaching assignments can be used in conjunction with noncurrency determination: the debugger can provide further information on an endangered variable V by telling the user which statements may have assigned V's actual value.

Note that the control reference statement has two roles: It conveys the control state of execution to the user and acts as an anchor in the source allowing the user to reason about source-level values. In unoptimized code, the control reference statement conveniently (and correctly) serves both of these purposes. First, since all statements before the control reference statement have completed execution and no statement following the control reference statement has begun execution, it accurately conveys the control state of execution. Second, since all variables are current with respect to the control reference statement, it acts as the most convenient anchor for reasoning about variable values. As we have seen, however, a single control reference statement cannot accurately convey the control state of execution in optimized code. Moreover, at a stopping instruction I, a variable V's actual value can be the expected value relative to one reference statement S, but not to another statement S'. Thus, V is current if the debugger uses S as the reference statement and noncurrent if the debugger uses S' as the reference statement. Therefore, a choice may exist in deciding on an anchor for reasoning about variable values. Alternatively, the debugger can qualify the value of a variable with a statement relative to which the variable is current (if possible). Hence, when dealing with optimized code, one possible approach is to decouple these two roles, and select one statement for conveying the control state and another statement as an anchor for noncurrency determination.

2.6 Summary

In this chapter, I have shown in detail how optimizations affect source-level debugging. I have introduced terminology related to variable values in optimized code. I have discussed various approaches that allow debugging in the presence of optimizations. In general, it is impossible to provide expected debugger behavior without constraining optimizations or modifying the debuggee (e.g., deoptimizing the debuggee at debug-time). But the same reasons that motivate debugging of optimized code also motivate non-invasive debugging. Non-invasive debugging of optimized code requires that the debugger user somehow be informed of the effects of optimizations. The debugger can go far in helping the user understand the effects of optimizations in source-level terms — for example, the debugger can determine when the runtime value of a variable does not correspond to the expected source-level value of the variable.

Chapter 3

Experimental Framework

In this chapter, I describe cmcc, the Carnegie Mellon University optimizing C compiler. I have implemented the algorithms that I describe in this dissertation in the context of this compiler. I also describe the methodology used to gather the measurements in this dissertation and the program suite I used in my measurements.

3.1 The cmcc compiler

cmcc accepts ANSI C as the source language and produces assembly language for a variety of target machine architectures. cmcc uses the lcc ANSI C compiler developed by Fraser and Hanson [44] as a front end. I have modified lcc to generate cmcc's internal representation. To preserve maximum flexibility, the cmcc optimizer and code generator runs as a separate C++ program, reading in the internal representation emitted by the lcc front end. Table 3.1 lists the optimizations performed by cmcc. At this time, cmcc has been targeted to the MIPS [59], SPARC [78], DLX [50], and iWarp [16] architectures. I obtained the results that I report in this dissertation using the code generator for the MIPS architecture.

cmcc consists of two major phases: (1) machine-independent global optimization, and (2) code generation. Each of these phases in part comprises a series of transformations. The global optimization phase first transforms the loop structure of the program by partially peeling loops and inserting loop preheader blocks. Some innermost loops are then peeled and unrolled, and induction variables are expanded inside of unrolled loops. Edges are split

Loop unrolling and peeling	Linear function test replacement
Induction variable expansion	Induction variable simplification
Constant propagation and folding	Induction variable elimination
Assignment propagation	Partial dead code elimination
Dead assignment elimination	Partial redundancy elimination
Strength reduction	Branch optimizations
Global register allocation (using graph coloring)	Register coalescing
Instruction scheduling	

Table 3.1: Optimizations performed by `cmcc`

in preparation for code motion transformations. After these control flow transformations, the optimizer performs constant propagation and folding using an algorithm based on abstract interpretation; this phase cleans up tests that were peeled out of loops. Copy propagation and dead code elimination are performed next. These two algorithms are based on the algorithms described by Chow [30] and are repeated until they effect no more changes on the program[1]. After copy propagation and dead code elimination, the optimizer performs partial redundancy elimination using the algorithm described by Knoop et al.[64]. Strength reduction is integrated with partial redundancy elimination using the algorithm described Knoop et al.[63]. Induction variable simplification is performed during partial redundancy elimination to reduce the number of address temporaries introduced by strength reduction. This optimization is also known as *base binding* [15]. Linear function test replacement and induction variable elimination are performed after partial redundancy elimination. Finally, partial dead code elimination (also known as assignment sinking) is performed using the algorithm described by Knoop et al.[65]. Partial dead code elimination is repeated until no more changes are detected in the program.

The code generation phase of the compiler performs global register allocation and local (i.e., intra-basic block) instruction scheduling. The global register allocator (described in detail in [72]) is a unique algorithm that integrates live range splitting with a Chaitin-style graph coloring register allocator [25]. The register allocation phase performs register coalescing and incorporates the improved coloring heuristics described by Briggs et al.[19]. The instruction scheduler is a list scheduler that can schedule in either the forward or backward directions. This scheduler is further parameterized to perform either cycle or

[1]This copy propagation algorithm propagates the right hand side of all assignments, not just assignments that are copies.

operation scheduling. To allow maximum scheduling freedom, the register allocator is run after the instruction scheduler.

The two major phases inside `cmcc` operate on different intermediate representations. Global optimizations operate on a simple machine-independent representation consisting of a control-flow graph of basic blocks, and a list of expression trees inside each basic block. The operators of these trees are based on the generic set of operators found in the `lcc` front end. Control-flow information is also encoded in loop-nest tree and dominator tree structures.

The code generation phase maps expressions to trees of abstract machine instructions, which are then linearized to schedules of instructions. All phases of the code generator (including the scheduler) operate on schedules of instructions. To keep the code generator retargetable, the representation of each machine instruction is kept machine independent and is encapsulated inside a C++ abstract class; the register allocator and the instruction scheduler operate on these abstract instructions in a machine-independent manner. Each new code generator tailors the abstract instruction class to the instructions of the target processor through inheritance: there are machine-specific instructions for each target machine; these machine-specific instructions inherit from the abstract instruction class and implement the interface defined by the abstract class' virtual functions. This mechanism allows the compiler to model each target machine instruction, and at the same time to keep the register allocator and instruction scheduler machine independent.

This strategy has proven successful as `cmcc` has been retargeted to four different architectures. These architectures are different in many respects: different calling conventions, different register file organizations (e.g., register windows versus no register windows, and separate versus unified floating-point and integer register banks), and different branch architectures (e.g., branch on condition code versus compare and branch). Moreover, the iWarp is a LIW machine with destructive two operand integer instructions, while the MIPS, SPARC, and DLX are RISC machines whose binary instructions have three operands.

Many of the global optimization phases of `cmcc` — such as partial redundancy elimination and dead assignment elimination — are based on bit-vector data-flow algorithms. Global register allocation also requires bit-vector data-flow algorithms to compute live and reaching sets, which are necessary for determining live ranges. The bit-vector data-flow

program	`gcc -O2`	`cc -O2`
`li`	0.98	1.05
`eqntott`	1.13	1.09
`espresso`	1.06	1.05
`gcc`	1.02	0.89
`alvinn`	1.06	0.94
`compress`	0.84	0.95
`ear`	1.07	0.95
`sc`	1.09	1.03

Table 3.2: Performance of optimized code generated by `cmcc`, relative to optimized code generated by `gcc` (version 2.3.2) and MIPS `cc` on a DECstation 5000/200.

algorithms used throughout `cmcc` are very similar in structure and a large amount of code reuse is achieved in `cmcc` through the use of a *data-flow analysis framework*; details of this framework can be found in [5].

To allow experimentation with profile-based optimizations, `cmcc` can generate an instrumented version of a program that, when executed, produces basic block and control-flow edge execution frequencies. The profiles generated by the instrumented program are used by `cmcc` to annotate the basic blocks and control-flow edges in the IR with the execution frequencies. Profile information is currently being used to guide spilling, splitting, and register assignment decisions in the register allocator [72].

The overall quality of the optimized code generated by `cmcc` is competitive with the code generated by the native MIPS `cc` and `gcc` compilers on a DECStation 5000/200. Table 3.2 shows the relative performance of the code generated by `cmcc` compared to the code generated by these other compilers. The benchmarks in this table are the C programs from the SPEC92 suite. These numbers are gathered on a DECStation 5000/200 and this table presents relative performance; a number of less than one means that `cmcc` produces better code. The native MIPS `cc` compiler is generally regarded as a very high-quality optimizing compiler. These performance numbers show that the optimized code generated by `cmcc` is roughly of the same quality as the optimized code generated by an industrial-strength optimizing compiler. Thus the experimental results presented in this dissertation can be reasonably generalized to other optimizing compilers.

There are a few benchmarks where `cmcc` performs better than the other compilers. This is due mainly to very aggressive partial redundancy elimination, constant propagation, and dead code elimination performed by `cmcc`. `cmcc` has not yet been tuned for compilation speed; thus, optimizations are performed as aggressively as possible. In an industrial compiler, optimization speed is an issue — an industrial compiler may not be as aggressive as `cmcc` in applying optimizations.

On the other hand, there are benchmarks where `cmcc` performs worse that the other compilers and this small performance difference is due to two factors: First, the global optimization phase works on a machine-independent representation. This representation does not expose a few machine-specific parameters such as sizes of immediates; thus, it is possible that not all instructions are subjected to global optimizations — for example, `cmcc` may miss an opportunity to hoist the load of an immediate out of a loop. Second, `cmcc` does not perform machine-dependent peephole optimizations — for example, aggressive branch delay filling and basic block placement. These differences should not have a major impact on debugging and should not limit the generality of the experimental results presented in this dissertation.

3.2 Quantitative methodology

In my approach to debugging optimized code, the debugger sometimes may not be able to provide the expected source-level value of a variable to the user; therefore, it is interesting to know how often a user can expect a query of a variable's value to result in a response where the debugger cannot display the variable's expected value. To answer such questions — regarding how often optimizations will affect a user's ability to debug in practice — a long-term user study that records the actual usage patterns of a debugger is necessary. Such user studies are, unfortunately, complicated to do because they require a set of developers willing to use an experimental compiler and debugger to develop their applications. Developers expect a fully-functional window-based source-level debugger, and an optimizing compiler with compilation times that are competitive with the native compiler. A user study is beyond the scope of this dissertation.

An approximation to the preceding question can be obtained with static measurements that assume all breakpoints are equally likely and all variables are equally likely to be queried. The charts in this dissertation show the average number of variables that are uninitialized, current, endangered, and nonresident at a breakpoint. These numbers are collected by counting the number of variables that fall into each category, at each possible breakpoint in the source program, and averaging the results by the number of breakpoints. In Chapter 5, I also present dynamic measurements wherein the effect of each breakpoint is weighed by the execution count of the breakpoint, gathered using a sample input data set. Neither static nor dynamic measurements indicate how often a debugger will be able to respond with an expected value during an actual debugging session, but as we will see, the measurements do identify general trends and the optimizations that are likely to cause the most trouble.

It is important to note that the numbers *are not* an evaluation of how well the algorithms work. The main contributions of this dissertation are the techniques that enable source-level debugging of optimized code inside a compiler and debugger. These algorithms are necessary so that the user is not misled during debugging. The measurements provide additional insight into how different optimizations affect debugging.

The empirical evaluation presented in this dissertation is based on the set of eleven C programs. Eight of these programs are from SPEC92. The SPEC programs are a suite of benchmarks created for comparative measurements of vendor system performance [95]. These programs are supposed to represent a variety of "typical" system workloads and have been used widely to report speedups from optimizing various system components, including the compiler. Although this dissertation is not about performance optimization, I have chosen the SPEC programs because they are widely available and have been studied in detail by compiler writers; thus, they provide a common point of reference for other researchers or developers who may implement my techniques in their compiler.

In addition to the SPEC92 C programs, I have included three (publicly-available) C programs: (1) `lcc`, version 1.9 of the retargetable ANSI C compiler developed by Fraser and Hanson [44]; (2) `tcl`, version 7.3 of the the Tcl (tool command language) scripting language system developed by John Ousterhout [81]; and (3) `triangle`, version 1.1 of the two-dimensional quality mesh generator and delaunay triangulator developed by Jonathan Shewchuk at Carnegie Mellon University [87].

It is difficult to say what a "typical" program that is being debugged under a source-

Program	Lines of code	Total source breakpoints	Breakpoints per function	Variables per breakpoint
li	7741	2594	7.4	5.2
eqntott	3483	1267	21.6	5.1
espresso	14842	7424	21.5	9.4
gcc	102389	28433	20.7	9.3
alvinn	322	140	8.3	6.3
compress	1503	429	26.9	5.8
ear	4466	1108	11.8	6.9
sc	8491	3400	23.1	7.1
lcc	10997	5970	20.1	4.7
tcl	26649	7820	29.2	10.0
triangle	10531	5312	57.7	33.2

Table 3.3: Programs used in this study

level debugger looks like. All programs, however, are candidates for debugging under a source-level debugger. Moreover, the process of debugging is likely to result only in minor changes to a program — that is, a debugged program will differ only in minor ways from its original, un-debugged form. The numbers in this dissertation are more likely to be affected by application type and programming style than by whether a program is likely to be further debugged under a source-level debugger. Therefore, even though the the programs I have selected have already been debugged and will probably not be further debugged under a source-level debugger, these programs serve well for the purposes of an empirical evaluation because they represent a range of (UNIX) applications, originate from different programmers, and represent a range of C programming styles.

Table 3.3 shows the sizes of the programs I have selected for my empirical evaluation. The third and and fourth column of this table show the total number of source-level breakpoints in each program and the average number of breakpoints per function. The last column shows the average number of local variables that are in scope at each source-level breakpoint.

3.3 Summary

To prove the practicality of an approach to debugging optimized code, it is necessary to have an optimizing compiler and an implementation of the debugger algorithms. Unfortunately, sources to a high quality, easily extensible, optimizing compiler are hard to come by: `gcc` is notoriously difficult to modify, `lcc` is not an optimizing compiler, and SUIF [105] was not available at the beginning of this project. Therefore, I have developed `cmcc`, a retargetable optimizing C compiler that generates code that is competitive with code generated by native optimizing compilers. The algorithms that I present in this dissertation have been implemented and evaluated in the context of this compiler.

A quantitative analysis of the effects of optimizations on source-level debugging is difficult, because such an analysis requires a user study. Moreover, it is unclear how to select a suite of programs for such an empirical evaluation. The measurements I present in this dissertation are static measurements that assume all breakpoints are equally likely and all variables are equally likely to be queried. I have chosen the SPEC92 programs for a program suite, because of the wide availability of these programs, and because of the range of applications and programming styles that these programs represent.

Chapter 4

Detecting Nonresident Variables

Global register allocation causes data-value problems by reusing registers that are assigned to variables. Register allocation also complicates the mapping from variables to runtime storage locations by assigning different storage locations to a variable at different points in a program. In this chapter, I present a data-flow analysis approach for finding those storage locations that hold source-level variable values at a break. In the first section, I present an overview of register allocation techniques and their related optimizations. In Section 4.2, I describe different approaches to detecting nonresident variables. In Section 4.3, I present the data-flow analysis approach. In Section 4.4, I describe how the debugger can attempt to improve the responses given to the user by detecting uninitialized variables. In Section 4.5, I present measurements of how register allocation may affect a debugger's ability to present a source-level value of a variable to the user.

4.1 Global register allocation

Register allocation attempts to speed up program execution by keeping frequently accessed values in high-speed registers. Such values include variables, temporaries, and constants, but since we are concerned with source-level debugging, I do not mention temporaries or constants any further.

4.1.1 Graph coloring

The most common approach models register allocation as a graph-coloring problem. The goal is to assign a register for exclusive use by a variable during the variable's *live range*, which consists of those basic blocks or instructions that lie between definitions and last uses of the variable [25, 32]. Live ranges that overlap in the program *conflict* (or *interfere*) and are assigned different physical registers. For each compilation unit, the compiler constructs an *interference graph* whose vertices represent live ranges, and whose edges connect live ranges that conflict. Finding a valid assignment of physical registers to live ranges is modelled as coloring the interference graph with N colors in such a manner that two live ranges connected by an interference edge do not get the same color, for a target machine with N physical registers managed by the compiler.

There are two common heuristic graph coloring algorithms used in the context of register allocation: *priority-based coloring* [32] and *graph simplification* [25, 19]. Priority-based coloring assigns registers (colors) to live ranges (vertices) in an order determined by the *priority* of each live range; the priority of a live range L is an estimation of the execution time saved by keeping L in a register. When no legal color exists for a live range L, the compiler *splits* L by segmenting it into smaller live ranges L_i; each L_i is then independently assigned a register, or left in memory if no legal color can be assigned to L_i. *Shuffle code* [72] is inserted to move L's data value when control passes from a segment L_a to another segment L_b.

Graph simplification is based on the observation that an *unconstrained* vertex V (i.e., a vertex with degree $< N$) can be trivially colored, since no matter what colors are assigned to V's neighbors, a legal color exists for V. Simplification successively removes unconstrained vertices from the interference graph; each time a vertex V is removed, the edges that are incident upon V are also removed, which decrements the degree of V's neighbors. Once all vertices have been removed, colors are assigned to vertices in the *reverse order* in which they were removed. If, during vertex removal, all the remaining vertices are constrained, then a vertex (i.e., live range) is picked to be *spilled*.

Spilling assigns a memory location (in the activation record) to a live range L and replaces all references to L with references to the memory location; L is removed from the interference graph. The additional memory references (i.e., loads and stores) are referred to as *spill code*. To minimize the size of a function's activation record, the spill locations in the activation record can be shared by non-overlapping live ranges (i.e., coloring can also be applied to spilled live ranges). The decision on which live range to spill is based on a heuristic cost function that takes into account the cost of spilling a live range as well as the benefit of removing the live range from the interference graph.

In both graph coloring approaches the units of allocation are live ranges. The live range of a variable V comprises either those *instructions* where V is live and reaching, or those *basic blocks* where V is live and reaching. The basic block representation of a live range is at a coarser and thus less precise granularity than the instruction-level representation: a register is prohibited from holding more than one live range's value within a basic block [32]. In practice, this is overly conservative when basic blocks are long, so implementations that use the basic blocks representation usually split large basic blocks into smaller ones [32, 69].

Priority-based coloring uses the basic block representation of a live range, because this representation facilitates live range splitting: when a color cannot be assigned to a live range L, a new live range L' is formed from L by incrementally adding basic blocks from L until adding one more block renders L' uncolorable.

Although graph simplification can also be implemented using the basic blocks representation of live ranges, graph simplification has traditionally been implemented using the instruction-level representation [17]. Renumbering [26] and web analysis [57] are techniques that isolate disjoint segments of a live range; these techniques further refine the instruction-level representation of live ranges, and result in interference graphs of potentially lower degree. These two techniques split a variable's live range, but require no shuffle code since the segments are disconnected.

There are several variations on the graph simplification approach: The RS/6000 compiler [13] optimizes spill code placement and refines the heuristic spill cost function. Some approaches [17, 66] have implemented live range splitting within the graph simplification framework by splitting live ranges *before* graph coloring; register allocation then tries to minimize the runtime cost of shuffle code by eliminating unnecessary splits via *coalescing* (coalescing is discussed in the next section). Other approaches [23, 45, 79, 72] have developed spilling and splitting heuristics that are sensitive to program structure and execution probabilities.

4.1.2 Register coalescing

Coalescing or subsumption [25] is an important register allocation optimization that eliminates copy operations by assigning the same physical register to the source and destination operands of a move instruction. Coalescing is an effective way to eliminate move instructions generated for copying register arguments and function return values, for copies between variables and temporaries (left over from global optimizations), and for destructive two-operand instructions (e.g., two-operand add instructions).

An important consequence of coalescing is that, at a stopping instruction, a register (or spill location) may hold the value of more than one variable. Consider the example shown in Figure 4.1. Part (a) of this figure shows the source code, while parts (b) and (c) show the object code before and after register allocation, respectively. In Figure 4.1(b), instructions I_2 and I_4 are generated from the assignments of statements S_2 and S_4, respectively; these instructions assign source-level values of variables `y` and `x`. Assume that the live ranges of `x` and `y` do not interfere. The register allocator coalesces `x` and `y`, and assigns the same register `R1` to both variables. Instruction I_4 is subsequently eliminated and no object code exists for statement S_4 (Figure 4.1(c)). Although the assignment to `x` is eliminated, instruction I_2 assigns the same value that is assigned by S_4, and at any stopping instruction after I_2, register `R1` holds two source-level values: the value of `y` assigned by statement S_2, and the value of `x` assigned by statement S_4. That is, I_2 assigns to `y` the value assigned by S_2, and assigns to `x` the value assigned by S_4. If the coalesced live range of `x` and `y` had been spilled to some memory location M, then M would have contained the value of both `x` and `y`.

Figure 4.1 illustrates another important consequence of coalescing: the instruction implementing the source-level assignment to `x` (i.e., S_4) is now executed earlier at I_2, causing `x` to be noncurrent at I_3. In general, coalescing causes endangered variables by eliminating move instructions that assign a source-level value to a variable. I address the endangerment problems caused by coalescing in Section 5.4 of the next chapter.

(a)		(b)		(c)	
S_1:	`...`	I_1:	`...`	I_1:	`...`
S_2:	`y=u+v;`	I_2:	`add Ry,Ru,Rv`	I_2:	`add R1,R2,R3`
S_3:	`...`	I_3:	`...`	I_3:	`...`
S_4:	`x=y;`	I_4:	`mov Rx,Ry`		
S_5:	`...`	I_5:	`...`	I_5:	`...`

Figure 4.1: Example: Register coalescing (a) Source code (b) Instructions before register allocation (c) Instructions after register allocation and coalescing.

4.1.3 Calling conventions

When assigning physical registers to live ranges, the register allocator must take into account conventions related to saving and restoring of caller- and callee-saved registers. If a live range L spans one or more function calls, and is assigned a caller-saved register, then *caller-saved shuffle code* is required to save and reload the value of L before and after each function call that is crossed by L, respectively. If a callee-saved register R is used inside a function (i.e., assigned to a live range), then *callee-saved shuffle code* is necessary to save and reload the value of R at the prologue and epilogue of the function, respectively.

Several optimizations are possible to reduce the cost of caller- and callee-saved shuffle code. Caller-saved shuffle code for a live range L can be optimized by eliminating unnecessary saves (reloads) to (from) a memory location M: a save is unnecessary if M already contains the value of L, and a reload is unnecessary if the value of L is not used before being saved again. The placement of callee-saved shuffle code can be optimized by saving and reloading the value of a callee-saved register R only along those paths where R is actually used [31, 41].

The values that are saved and reloaded by callee-saved shuffle code are values from calling functions. Callee-saved values have no relationship to values in the context of the callee function; therefore, callee-saved shuffle code does not play a role when determining the residence of a local variable. When performing a stack trace, however, the debugger must use the saved values of callee-saved registers to reconstruct callee-saved register values in the contexts of calling functions. Therefore, the debugger must be aware of the storage locations where callee-saved registers have been saved, and of whether any callee-save shuffle code optimizations have been performed.

4.1.4 Shuffle code

Shuffle code transfers a value from one runtime location to another, and is generated for two reasons: First, when splitting partitions a live range L into several live ranges L_i, shuffle code is needed to transfer L's value at those program points where control passes from one live range L_a to another L_b; the shuffle code transfers L's value from the location assigned to L_a to the location assigned to L_b. Second, when a live range L spans one or more function calls, and L is assigned a caller-saved register, shuffle code is needed around each function call to save (reload) L's value to (from) memory. There are three types of shuffle code instructions:

Saves: A save is a store instruction that transfers a live range's value from a register to a memory location.

Reloads: A reload is a load instruction that transfers a live range's value from a memory location to a register.

Moves: A move instruction transfers a live range's value from one register to another.

4.1.5 Global variables and aggregates

Global variables and aggregates (i.e., array and structure elements) can also be register allocation candidates [100, 21, 84]. When included as register allocation candidates, globals and aggregates are usually assigned a home location in memory and are promoted to registers over a limited region of the program (e.g., those regions where they are not aliased and are frequently accessed [53]). Allocation of globals and aggregates requires alias and dependence analysis, and many compilers consider only local scalar variables (besides temporaries and constants) as register allocation candidates.

The type and storage class of the register promoted variable, is orthogonal to residency detection. We do not mention globals and aggregates in the rest of this chapter; the residency detection analysis described in Section 4.3 is applicable to any register promoted value.

4.1.6 Effects on source-level debugging

In summary, the various techniques associated with register allocation have the following effects on debugging:

- Live ranges can be split and spilled; as a result, a variable may be assigned different registers at different points in the program, or be assigned a register at one point and be spilled at another. Therefore, the mapping from variables to runtime locations, generated by the compiler, can be one to many; at a break, the debugger must be able to determine which register or stack location, if any, holds the latest runtime value of a source-level variable.

- Because of coalescing, a register (or spill location) may hold more than one variable value at a stopping instruction. Therefore, the mapping from variables to runtime locations can be many to one.

- When variables are spilled, the register allocator can minimize the size of the activation record by sharing spill locations. Therefore, spilled values can be nonresident, and the debugger must perform analysis even on spilled variables.

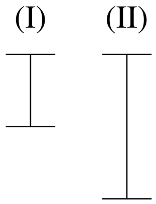
4.2 Approaches to detecting nonresident variables

There are several approaches that a debugger can take to determine if a variable is resident at a stopping instruction. Since a variable is resident during its live range, one approach is to consider a variable as resident at a stopping instruction within the variable's live range. The advantage of this approach is that live range information is already computed by the compiler's register allocation phase. For example, in the DOC debugger [38], the address ranges of instructions in a variable's live range are recorded in the *range record* data structure at the same time as the interference graph is built by the register allocator. The range records are passed to the debugger, which uses them to determine whether a stopping instruction lies within a variable's live range. The CXdb debugger [20] also uses live ranges to determine residency.

Using a variable's live range for determining residency is simplistic and conservative:

I_0: . . .
I_1: def R `def x`
I_2: . . .
I_3: use R `last use x`
I_4: . . .
I_5: def R `def y`
I_6: . . .

(a)



(I) Residence determined by live range

(II) Residence determined by flow analysis

(b)

Figure 4.2: Example: Approaches to detecting nonresident variables (a) Object code (b) Ranges in which x is resident.

the debugger uses a simple rule that is always right but misses opportunities. A storage location assigned to a variable V may still hold the value of V after the live range of V (i.e., after the last use of V); this is illustrated in Figure 4.2. Figure 4.2(a) shows a sequence of definitions and uses of a register R in a straight-line piece of object code. Register R is assigned to source variables x and y. Instruction I_1 writes a value of x into R and marks the beginning of x's live range, whereas the use of R at I_3 is the last use of x and establishes the end of x's live range. x is definitely resident at stopping instructions I_2 or I_3, since these instructions lie within x's live range. x remains resident until I_5 writes y's value in R, thus *evicting* x from R (I discuss eviction in Section 4.3). But the range of instructions after I_3 are not part of x's live range. Hence, at stopping instruction I_4, a debugger that bases x's residency on x's live range will report x as being nonresident, even though R still contains x's value.

An approach to detecting nonresident variables that is more aggressive than using live ranges is to precisely detect *all* points where V becomes nonresident. This implies detecting that x is still resident at I_4 in Figure 4.2(a) and allows the debugger to display the value of x outside of x's live range, as depicted by Figure 4.2(b). In Section 4.3, I describe a method based on data-flow analysis that realizes such an approach by detecting *evicted variables*.

When there are a large number of physical registers, it is unlikely that a live range's register is re-used immediately after the end of the live range; thus, it is likely that a live range's values will persist well beyond the end of the live range. The effectiveness of using data-flow analysis to extend beyond a live range's boundaries also depends on how aggressively the register allocator re-assigns registers.

4.3 Detecting evicted variables

My method for detecting nonresident variables accurately tracks the values assigned by individual instructions, along all possible execution paths in the program. Using data-flow analysis, this algorithm determines which runtime locations hold the latest source-level values with respect to the stopping instruction. I start the description of the data-flow analysis by defining some terminology.

4.3.1 Terminology

A control flow graph is a directed graph (B, S, E) where B is the set of basic blocks; $S \in B$ is the entry block; E is the set of edges between blocks such that if $(B_i, B_j) \in E$ then control may immediately reach B_j from B_i. Each basic block B_i contains a sequence of instructions generated by the compiler, as well as a special *pre-amble instruction* that appears before the other instructions in B_i thus dominating them and a *post-amble instruction* that appears after the other instruction in B_i thus post-dominating them. The pre-amble and post-amble instructions are abstractions used by my algorithms; they are not generated by the compiler nor do they appear in the object code. The pre-amble instruction of a block B_i is denoted by $Preamble(B_i)$, while the post-amble instruction of a block B_i is denoted by $Postamble(B_i)$. Given an instruction I, $Block(I)$ is the basic block containing I. I define the set of predecessor instructions of I, denoted $pred(I)$, as the set of instructions from which control can immediately reach I. A *point* is defined as being either between two adjacent instructions, before the first instruction in a basic block, or after the last instruction in a basic block. The point immediately before an instruction I is denoted $pre(I)$, and the point immediately after I is denoted $post(I)$. The *entry point* of the control flow graph is the point at the beginning of the source basic block S, and is denoted by $start$. The entry point dominates all other points in the control flow graph. A *path* in the object code is defined to be a sequence of points $\langle p_1, ..., p_n \rangle$ such that for each adjacent pair p_i, p_{i+1}, either $p_i = pre(I)$ and $p_{i+1} = post(I)$ for some instruction I, or p_i is a point at the end of a block B_j and p_{i+1} is a point at the beginning of a block B_k and $(B_j, B_k) \in E$. An instruction I is part of a path P, denoted $I \in P$, if $pre(I)$ and $post(I)$ both occur in P, and $pre(I)$ occurs before $post(I)$ along P. A basic block B_i is part of a path P, denoted $B_i \in P$, if

$Preamble(B_i) \in P$.

An instruction I *reaches along a path* P in the object code, if $I \in P$ and the destination register of I is not defined by any other instruction in P after the last occurrence of $post(I)$. An instruction I *reaches a point* O in the object code, if there exists a path $P = \langle start, ..., O\rangle$ such that I reaches along P.

Since nonresident variables are caused by global register allocation, only the storage locations that are assigned by the register allocator and the variables that are candidates for allocation are germane to our discussion. A *storage location* L is either a physical register or a slot in the activation record (stack frame) of the function being compiled. A register R can be from any register bank of the target machine (e.g., an integer, floating-point, or condition-code register). Some architectures hold a double-precision floating-point quantity in a pair of consecutive registers; thus, R can represent a register pair.

Slots are assigned from the *register spill area* of the activation record and can be single- or double-word sized. A live range L is assigned a stack slot either because L cannot be assigned a register and is thus spilled by the register allocator, or because L is assigned a caller-saved register and caller-saved shuffle code is required for L (L spans one or more function calls and must be saved and restored around these calls). We are not interested in the static data area or dynamically allocated data (i.e., heap) because these memory locations are not assigned by the register allocator[1]. If global variables are promoted to registers, then static memory locations can be included as relevant storage locations.

A *register variable* V is a variable that has been promoted to a register by the compiler. V is a candidate for register allocation and will be assigned one or more storage locations (i.e., registers and/or activation record slots). Variables that are not promoted to registers have an assigned *home location* in memory and are not discussed further; these variables are always resident since their home locations are not shared with other variables. Register variables, however, can be nonresident since a storage location can be assigned to more than one variable.

A *residence* is a pair $\langle V, L\rangle$ where V is a register variable and L a storage location assigned to V. A register variable V can be assigned more than one storage location and

[1] Some language implementations allocate activation records from the heap rather than from a runtime stack. What is important here is the activation record slots.

$Residences(V)$ is the set of residences of V:

$$Residences(V) = \{\langle V', L\rangle : V' = V\}.$$

$Locations(V)$ is the set of storage locations assigned to V:

$$Locations(V) = \{L : \langle V, L\rangle \in Residences(V)\}.$$

Note that since storage locations are shared among register allocation candidates, it is not necessarily the case that $Locations(V_1) \cap Locations(V_2) = \emptyset$ for any pair of variables V_1 and V_2. $Variables(L)$ is the set of variables to which storage location L is assigned:

$$Variables(L) = \{V : L \in Locations(V)\}.$$

At a stopping instruction I, a variable V is *resident* in a storage location $L \in Locations(V)$, denoted $Resident(V, L, I)$, if, at I, the value in L is a source-level value of V; that is, V is resident at I if there exists an L such that $Resident(V, L, I)$.

An instruction I that defines a storage location L is a *definition* of L; if L is a register R then I is an instruction that targets R, otherwise if L is a stack slot S then I is a store instruction that writes to S. I distinguish instructions that write a source-level value of V into a storage location $L \in Locations(V)$: A *source definition* of a variable V is an instruction I that assigns a source-level value of V to a storage location $L \in Locations(V)$ (I is a definition of L); I is generated from a source assignment expression that assigns to V and establishes V's residence in L. A *definition of a residence* $\langle V, L\rangle$ is a source definition of V that defines L; I will use the superscript notation $I_i^{\langle V,L\rangle}$ as shorthand for an instruction I_i that defines a residence $\langle V, L\rangle$. Note, that because of coalescing, an instruction can be a source definition of more than one variable and thus can define more than one residence.

Let I be a definition of a storage location L, where $L \in Locations(V)$ for some variable V. If I is not a source definition of V, then I is an *evicting definition* of V and we say that I *evicts* V from L. (I writes a value into L that is not a source-level value of V.)

4.3.2 Available residences

At the point immediately following a source definition $I^{\langle V,L\rangle}$, V is resident since L holds a value from a source assignment to V. V remains resident in L until a later evicting definition evicts V from L. Therefore, V is resident along those execution paths where a source definition $I^{\langle V,L \rangle}$ reaches:

Definition 1 *A residence* $\langle V, L\rangle$ **reaches along a path** $P = \langle start, ..., O\rangle$ *in the object iff a source definition* $I^{\langle V,L\rangle}$ *reaches along* P.

Lemma 1 *If a residence* $\langle V, L\rangle$ *reaches along a path* $P = \langle start, ..., O\rangle$ *and* P *is the execution path traversed to a stopping instruction* I*, then* $Resident(V, L, I)$ *is true.*

Since the debugger does not know which path is executed to reach a break, all paths leading to a stopping instruction must be considered:

Lemma 2 *If* $\langle V, L\rangle$ *reaches along all paths leading to a stopping instruction* I*, then* $Resident(V, L, I)$ *is true at break* $\langle S, I\rangle$ *for any* S.

I define a predicate $AvailRes(\langle V, L\rangle, O)$ that is true when a residence $\langle V, L\rangle$ is *available* at a point O in the object:

Definition 2 *A residence* $\langle V, L\rangle$ *is* **available at a point** O *in the object iff* $\langle V, L\rangle$ *reaches along all paths leading to* O*. The predicate* $AvailRes(\langle V, L\rangle, O)$ *is true if a residence* $\langle V, L\rangle$ *is available at a point* O.

By Lemma 2, $AvailRes(\langle V, L\rangle, pre(I))$ implies $Resident(V, L, I)$. The data-flow analysis I present in Section 4.3.4 solves for $Resident(V, L, I)$ by computing $AvailRes(\langle V, L\rangle, pre(I))$.

4.3.3 Multiple available residences

Because of live-range splitting, a variable V may be assigned more than one storage location; therefore, it is possible that several residences of V are available at a break. Consider, for example, the code of Figure 4.3. In this figure, variable `x` is assigned two registers: `R1` and `R2`. Instructions I_1 and I_3 are source definitions of `x`, generated from the assignments of statements S_1 and S_3, respectively. I_1 defines residence $\langle \texttt{x}, \texttt{R1}\rangle$, while I_3 defines residence $\langle \texttt{x}, \texttt{R2}\rangle$. Both residences are available at instruction I_4 (assuming no definitions of `R1` between I_1 and I_3).

S_1:	`x = ...`	$I_1^{\langle \mathtt{x},\mathtt{R1} \rangle}$:	`add R1,...`
S_2:	`...`	I_2:	`...`
S_3:	`x = ...`	$I_3^{\langle \mathtt{x},\mathtt{R2} \rangle}$:	`sub R2,...`
S_4:	`...`	I_4:	`...`
	Source		Object

Figure 4.3: Example: Multiple available residences.

Since we are interested only in the latest value assigned to a variable V, we sharpen Definition 1 so that only the residence(s) holding the latest value of V can reach along a path:

Definition 3 *A residence* $\langle V, L \rangle$ **reaches along a path** $P = \langle start, ..., O \rangle$ *iff a source definition* $I^{\langle V,L \rangle}$ *reaches along* P *and no other source definitions of* V *occur along* P *after* $I^{\langle V,L \rangle}$.

Based on this new definition, residence $\langle \mathtt{x}, \mathtt{R1} \rangle$ does not reach instruction I_4 along any path in Figure 4.3, because of source definition I_3. Residence $\langle \mathtt{x}, \mathtt{R2} \rangle$, however, still reaches I_4.

4.3.4 Data-flow analysis

Given an instruction I, I define the data-flow sets $AvailResIn(I)$ and $AvailResOut(I)$ as follows:

$$AvailResIn(I) = \{\langle V, L \rangle : AvailRes(\langle V, L \rangle, pre(I))\}$$

$$AvailResOut(I) = \{\langle V, L \rangle : AvailRes(\langle V, L \rangle, post(I))\}$$

A residence $\langle V, L \rangle$ is available at the point immediately before an instruction I only if $\langle V, L \rangle$ reaches the points after *all* of I's predecessor instructions. Thus the $AvailResIn$ set of an instruction I is related to the $AvailResOut$ sets of I's predecessor instructions by the following data-flow equation:

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

The set of residences that are made available by an instruction I is denoted by $AvailResGen(I)$, while the set of residences whose availability is killed by I is denoted by

$AvailResKill(I)$. The $AvailResIn$ and $AvailResOut$ sets of an instruction I are related by the following data-flow equation:

$$AvailResOut(I) = (AvailResIn(I) \setminus AvailResKill(I)) \cup AvailResGen(I)$$

An instruction I that defines a residence $\langle V, L\rangle$ causes $AvailRes(\langle V, L\rangle, post(I))$ to be true and thus generates the availability of $\langle V, L\rangle$. I, however, kills the availability of all other residences of V (since by Definition 3 we are interested in the latest source definition of V). Similarly, an instruction I that defines a storage location L kills the availability of all residences associated with L and causes $AvailRes(\langle V, L\rangle, post(I))$ to be false for all $\langle V, L\rangle \in Residences(L)$. Given an instruction I that defines a residence $\langle V, L\rangle$, $AvailResGen$ and $AvailResKill$ are the smallest sets defined as follows:

- If I defines a residence $\langle V, L\rangle$:
  - $\langle V, L\rangle \in AvailResGen(I)$
  - $\forall L' \in Locations(V) : \langle V, L'\rangle \in AvailResKill(I)$
- If I defines a location L:
  - $\forall V \in Variables(L) : \langle V, L\rangle \in AvailResKill(I)$

Function calls

Function calls kill the contents of caller-saved registers and therefore kill the availability of all residences $\langle V, R\rangle$ where R is a caller-saved register.

Shuffle code

Let I be a shuffle code instruction that transfers the value of a variable V from a storage location L_a to another location L_b. The effect of I is to duplicate V's value in L_b: after the execution of I, L_a and L_b both hold the same value of V. Therefore, I generates the residence $\langle V, L_b\rangle$, but does not kill the residence $\langle V, L_a\rangle$. Note that because of shuffle code, a variable V can be resident in more than one storage location — unlike the example of Section 4.3.3, however, all such locations hold the same value of V.

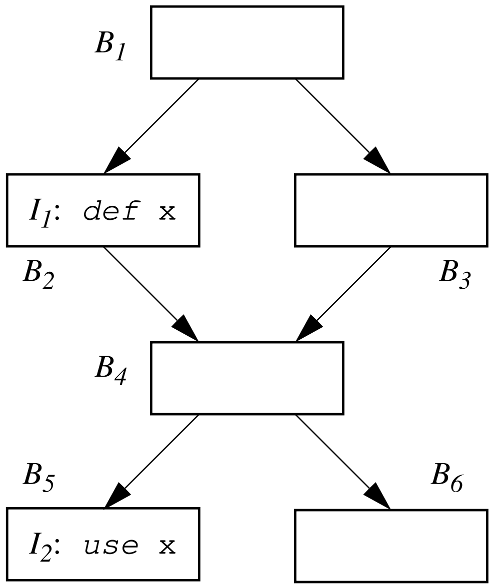


Figure 4.4: Example: Live but nonresident variable.

4.3.5 Nonresident but live variables

The impetus for using the available residence data-flow analysis is to extend beyond the live range of a variable V the points where V is detected as resident. But there are rare situations where the available residence analysis is *more* conservative than using live range information. Consider the example shown in Figure 4.4. In this figure, the live range of variable x extends from instruction I_1 in block B_2 to instruction I_2 in block B_5. The available residence analysis will detect x as resident after I_1 in block B_2, but nonresident inside block B_4, even though B_4 is part of the live range of x.

This conservatism is a consequence of using static data-flow analysis: the debugger cannot tell which execution path was taken to reach a stopping instruction and must therefore consider all paths as likely. The available residence analysis guarantees that a resident variable is initialized with a source-level value. Thus, a variable V that is live but uninitialized along some execution path(s), is detected as nonresident by this analysis — for example, in Figure 4.4, x is nonresident at the beginning of block B_5, even though a use of x is anticipated at instruction I_2. If there exists at least one path where a variable V is nonresident, then it is possible that the value in the runtime location of V is not a valid source-level value of V; using the available residence analysis in this situation, the debugger takes the conservative view that V is nonresident, rather than take the chance of showing

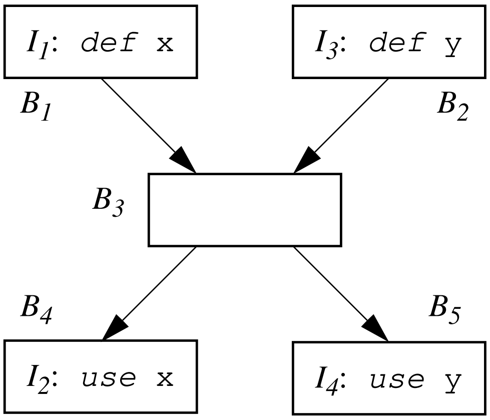


Figure 4.5: Example: Live but nonresident variables that overlap.

an erroneous value to the user.

In contrast, the live range of a variable can contain points where the variable is potentially uninitialized at runtime. Therefore, a debugger that uses live range information to detect resident variables may display an undefined value to the user — for example, in Figure 4.4, if execution reaches a break at B_4, without going through B_2, x is resident (using the live range approach) but its runtime value is undefined. Figure 4.5 shows a more complicated example. In this figure, variables x and y are nonresident at blocks B_3, B_4, and B_5 even though they are both live at these blocks. To further complicate matters, x and y can be assigned the same register, even though their live ranges overlap at block B_3.

If a variable is live but not reaching, then clearly an "uninitialized variable" bug exists and the user should be warned. On the other hand, if a variable is live and reaching along only *some* paths, the debugger can regard the variable as resident but with a warning to the user that the runtime value of the variable may be undefined — the user may be able to determine (based on other runtime values) whether the variable is indeed resident. The example of Figure 4.7, discussed later in this section, shows a real example where the user can determine a variable's residence better than the debugger can.

To eliminate the ambiguity due to multiple paths reaching a break, the compiler can specialize code to particular execution paths by performing code duplication. Figure 4.6 shows the example of Figure 4.4 after the compiler has duplicated blocks B_4, B_5, and B_6; the copies of these blocks are labeled B_4', B_5', and B_6', respectively. At block B_4, x is clearly resident, whereas at block B_4', x is clearly uninitialized. Similarly, at the beginning of block

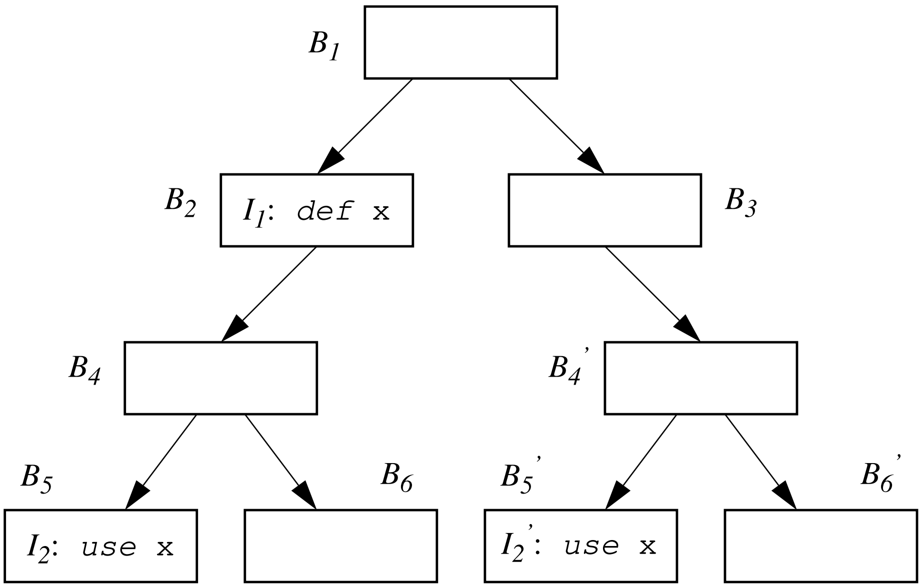


Figure 4.6: Example: Eliminating ambiguity via code duplication.

B_5, x is clearly resident, whereas at the beginning of block B_5', x is clearly uninitialized — it is clear at the beginning of block B_5' that an "uninitialized variable" bug exists.

Code expansion can have a negative effect on the cache behavior of a program; therefore, it is probably not worth performing code duplication specifically to eliminate those ambiguous cases where a variable is resident but potentially uninitialized. Although global optimizations can benefit from code duplication [29], in the case of eliminating resident but potentially uninitialized variables, code duplication is performed at a very late stage of compilation (during or after register allocation). Unless another global optimization phase is performed after register allocation, the code duplication will not benefit optimization.

The C language code fragment shown in Figure 4.7 illustrates a real situation in which the available residence analysis will determine that a variable is nonresident at a break, even though the variable is live and initialized (i.e., even though the break occurs within the variable's live range). This code fragment is from the `xlisp` program of the SPEC'92 suite; it is extracted from the function `binary()`, at line 106 of the file `xlmath.c`. There are three variables that are relevant to our discussion: `imode`, `ival`, and `fval`; these variables are all assigned registers by the compiler. This code fragment consists of two parts: (1) lines 118–127, which contain assignments to variables `imode`, `ival`, and `fval`; and (2) lines 130–134, which contain references to these variables. The tests at lines

```
/* set the type of the first argument */
if (fixp(arg)) {
    ival = getfixnum(arg);
    imode = TRUE;
}
else if (floatp(arg)) {
 fval = getflonum(arg);
 imode = FALSE;
}
else
 xlerror("bad argument type",arg);

/* treat '-' with a single argument as a special case */
if (fcn == '-' && args == NIL)
 if (imode)
     ival = -ival;
    else
     fval = -fval;

```

Figure 4.7: Extract from `xlisp` illustrating live but nonresident variables.

118 and 122 test whether the variable `arg` represents an integer or floating-point number, respectively. If the test at line 118 evaluates to true, then the variable `ival` is set to the integer value represented by `arg` (line 119), and the variable `imode` is set to true (line 120). Otherwise, if the test at line 122 evaluates to true, then the variable `fval` is set to the floating-point value represented by `arg` (line 123), and the variable `imode` is set to false (line 124). If the tests at line 118 and 122 both fail, then an error routine (`xlerror()`) is called (line 127); this routine handles fatal errors and does not return. The rest of this function references one of either `ival` or `fval` depending on whether the flag `imode` is true or false, respectively; for example, lines 130–134, negate the value of either `ival` when `imode` is true, or `fval` when `imode` is false.

Consider a break at line 130 in this code fragment. A static analysis by the compiler shows that any path leading to this break must pass through one of three points: line 119, line 123, or line 127. Assuming no optimizations other than register allocation have been performed, this break clearly lies within the live ranges of `imode`, `ival`, and `fval`, since these variables are both live and initialized at line 130. The available residence analysis, however, will determine that `imode` is nonresident because there is a path — the path that passes through line 127 — along which this variable is nonresident in the object code; the compiler does not know that the function `xlerror()` does not return, and thus, cannot determine that execution will never reach line 130 through line 127.

Even if the compiler could determine that `xlerror()` never returns, the available residence analysis will detect that variables `ival` and `fval` are nonresident at line 130, because each of these two variables is defined along only some paths leading to this line: `ival` is defined only on paths that pass through line 119, and `fval` is defined only on paths that pass through line 123. The second part of this code fragment, however, uses only one of these two variables, depending on the value of `imode`; therefore, even though the uses of these variables at lines 132 and 134 are potential uses of uninitialized values, the programmer has made sure that there these variables are used only if they are initialized. In fact, `ival` and `fval` could be assigned the same register (assuming that the target architecture had a unified integer and floating-point register bank like the AMD AM29000 or the iWarp architectures) — this is an example where the situation depicted in Figure 4.5 could potentially happen.

4.4 Uninitialized variables

The C programming language does not define the initial values of automatic variables; therefore, the source-level value of an uninitialized variable is undefined. Typically, if the debugger user queries the value of an uninitialized variable V, the debugger will show whatever value happens to be in the runtime location of V. In optimized code, a variable that is uninitialized in the source will most likely be nonresident in the object (unless an assignment executes prematurely due to code movement optimizations and causes the variable to be endangered). If the debugger can determine that a nonresident variable is actually uninitialized at break — in other words, the variable has no valid source-level value — then the debugger can provide a more precise response to the user, by reporting the variable as uninitialized rather than nonresident. That is, the debugger can provide a better response to the user by reporting that a variable has no valid source-level value rather than reporting that the variable's runtime value has been eliminated by optimizations.

The debugger can perform reaching analysis to detect which variables are uninitialized at a break. However, this analysis can only find variables that are uninitialized along all paths leading to a break, and cannot help in the case that a variable is uninitialized only on some paths. For example, consider the source code shown in Figure 4.8. There are no assignments to `x` other than the one at statement S_2. At statement S_3, `x` is uninitialized if the value of `p` is false, otherwise the value of `x` is the value assigned by statement S_2.

In languages where the initial value of a variable is defined, the compiler must insert an explicit initialization assignment into the intermediate representation. In this case, reaching analysis will not help because a variable is never uninitialized in the source. If the initial value of a variable V is never used, or used only along some paths, then optimizations may eliminate or move the initialization code of V, thus causing V to be nonresident at some breakpoints.

To aid in determining which variables are uninitialized at a break, a language implementation can initialize a variable's runtime location to a special out-of-band value representing the uninitialized state (i.e., uninitialized values are tagged). This helps the debugger detect variables that are uninitialized along only some paths. But such initialization is generated strictly for debugging purposes and will probably not be part of an optimized translation of

```
S1:  if (p)
S2:      x = ...
S3:  ..x..
```

Figure 4.8: Example: Uninitialized variables.

a program.

4.5 Empirical results

In this section, I present measurements of how often nonresident variables are likely to occur and how optimizations influence the number of nonresident variables. I also present measurements that compare the effectiveness of using the available residence data-flow analysis against using live range information, to detect nonresident variables. I use three metrics in these measurements: (1) the percentage of breakpoints that contain nonresident variables, (2) the average number of nonresident variables at each breakpoint, and (3) the average number of breakpoints that a variable is resident.

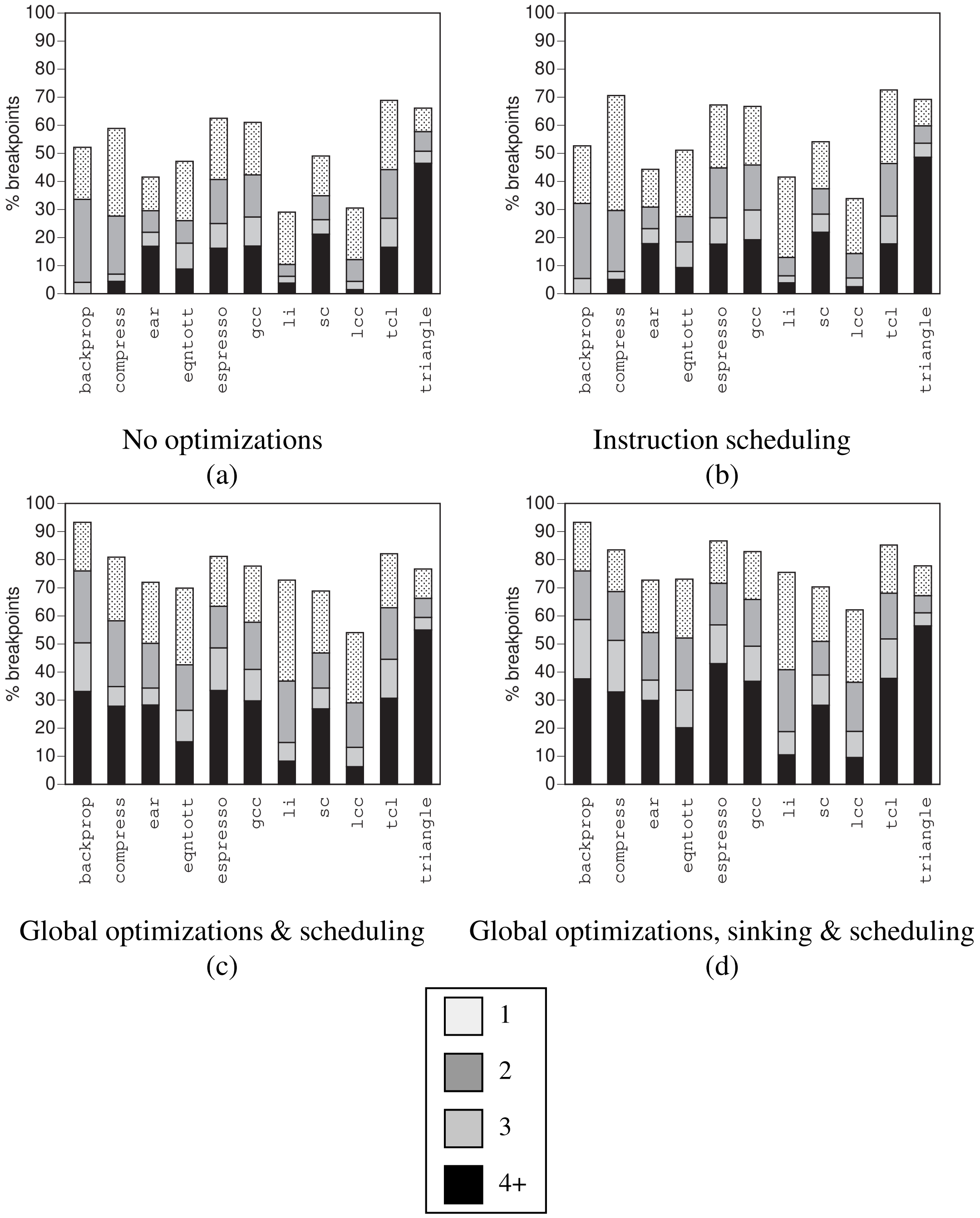
To measure the effects of different optimizations on the number of nonresident variables, each chart in this section shows the results when a different set of optimizations is enabled in conjunction with register allocation. I consider four combinations of optimization: (1) no optimizations (i.e., register allocation only); (2) instruction scheduling; (3) instruction scheduling and global optimizations; and (4) instruction scheduling, global optimizations, and partial dead code elimination (sinking).

The charts in Figure 4.9 show the percentage of breakpoints with nonresident variables; each column in these charts is broken down according to the number of breakpoints with one, two, three, and four or more nonresident variables, respectively. When only register allocation is done (Figure 4.9(a)), between 30–60% of all possible breakpoints contain nonresident variables. A few programs, such as `ear`, `espresso`, `gcc`, `sc`, `tcl`, and `triangle`, have a significant number of breakpoints with four or more variables that are nonresident. Figure 4.9(b) shows the results when instruction scheduling is also performed. Scheduling has a minor effect on the number of nonresident variables: a few of the programs have additional breakpoints where one variable is nonresident. This slight increase in the number of nonresident variables is due to the increase in register pressure caused by instruction scheduling. `cmcc` currently performs only local instruction scheduling; by hoisting instructions across basic blocks, global instruction scheduling is likely to have a more dramatic effect on register pressure, and thus, is likely to increase the number of nonresident variables.

Figure 4.9(c) shows the results when global optimizations are also performed. The number of breakpoints with nonresident variables increases significantly. The number of variables that are nonresident at these breakpoints also increases significantly. There are two reasons for this increase in nonresident variables: First, dead assignment elimination and induction variable elimination — coupled with transformations, such as assignment propagation, constant propagation, and linear function test replacement, that increase the effectiveness of these optimizations — eliminate live ranges, thus increasing the number of breakpoints where a variable is nonresident. Second, partial redundancy elimination and strength reduction inside loops, both increase register pressure by introducing new live ranges that span several basic blocks. This increased register pressure forces the register allocator to reuse registers more aggressively; as a result, the register assigned to a variable is likely to be reused soon after the end of the variable's live range.

Figure 4.9(d) shows the results when assignment sinking is performed along with global optimizations and instruction scheduling. For `espresso`, `gcc`, `lcc`, and `tcl`, assignment sinking increases the number of breakpoints with nonresident variables; most of these additional breakpoints have four or more nonresident variables. For other programs, there is an increase in the number of breakpoints with two or more variables that are nonresident. The reason for these increases is that partial dead code elimination shrinks the size of live ranges, by moving assignments to a variable closer to uses of the variable; therefore, there are fewer breakpoints comprising a variable's live range.

The charts in Figure 4.10 show the average number of variables that are nonresident at each breakpoint for all programs in the suite except `triangle`. These charts show that, on average, approximately 20–30% of all local variables are nonresident at a breakpoint when only instruction scheduling is performed, and approximately 30–60% of all local variables are nonresident at a breakpoint when optimizations are performed along with scheduling. `triangle` has a significantly greater average number of local variables than the other programs; thus, the results for `triangle` are shown separately in Figure 4.11. About



No optimizations
(a)

Instruction scheduling
(b)

Global optimizations & scheduling
(c)

Global optimizations, sinking & scheduling
(d)

Figure 4.9: Percentage of breakpoints with 1, 2, 3, and 4 or more nonresident variables using available residence data-flow analysis.

35% of the variables in this program are nonresident when only instruction scheduling is performed, and 46% of the variables are nonresident when optimizations are performed along with scheduling.

Figure 4.12 shows a different view of the same data shown in Figures 4.10 and 4.11; this figure directly compares the average number of resident variables using different optimizations. The numbers in Figures 4.10, 4.11, and 4.12 support the results of the earlier paragraphs: scheduling has a minor effect on the average number of nonresident variables (Figure 4.10(b)); global optimizations almost double the number of nonresident variables at each breakpoint (Figure 4.10(c)); and assignment sinking further increases the number of nonresident variables (Figure 4.10(d)).

The charts in Figure 4.10 also show that, on average, between 12–22% of the variables are detected as uninitialized at a break, using reaching analysis. `triangle` (Figure 4.11) has a significantly larger number of uninitialized variables compared to the other programs: the functions in this program are very large and many of the variables declared at the beginning of a function are unused for large portion of the function. Without detecting uninitialized variables, uninitialized variables would be detected as nonresident by the debugger (note, that these are variables that are uninitialized with respect to where the break is reported in the source). Thus, detecting uninitialized variables may help provide more accurate information to the user.

The influence of assignment sinking on the lengths of live ranges can be further quantified by measuring the length of each variable's live range. Figure 4.13 shows the average length of a variable's live range, in source-level breakpoints. As more global optimizations are enabled, the lengths of live ranges decrease because of dead code elimination and assignment sinking. Note, that the increased register pressure, caused by global optimizations, does not affect length of live ranges.

Figure 4.14 shows the average number of breakpoints that a variable is resident using available residence analysis. For a few programs (`compress`, `backprop`, `sc`, and `triangle`) the decrease caused by global optimizations are more dramatic than those in Figure 4.13. The reason for this is that the increase in register pressure caused by optimizations causes registers to be reused sooner, thus reducing the number of breakpoints where a variable is resident.

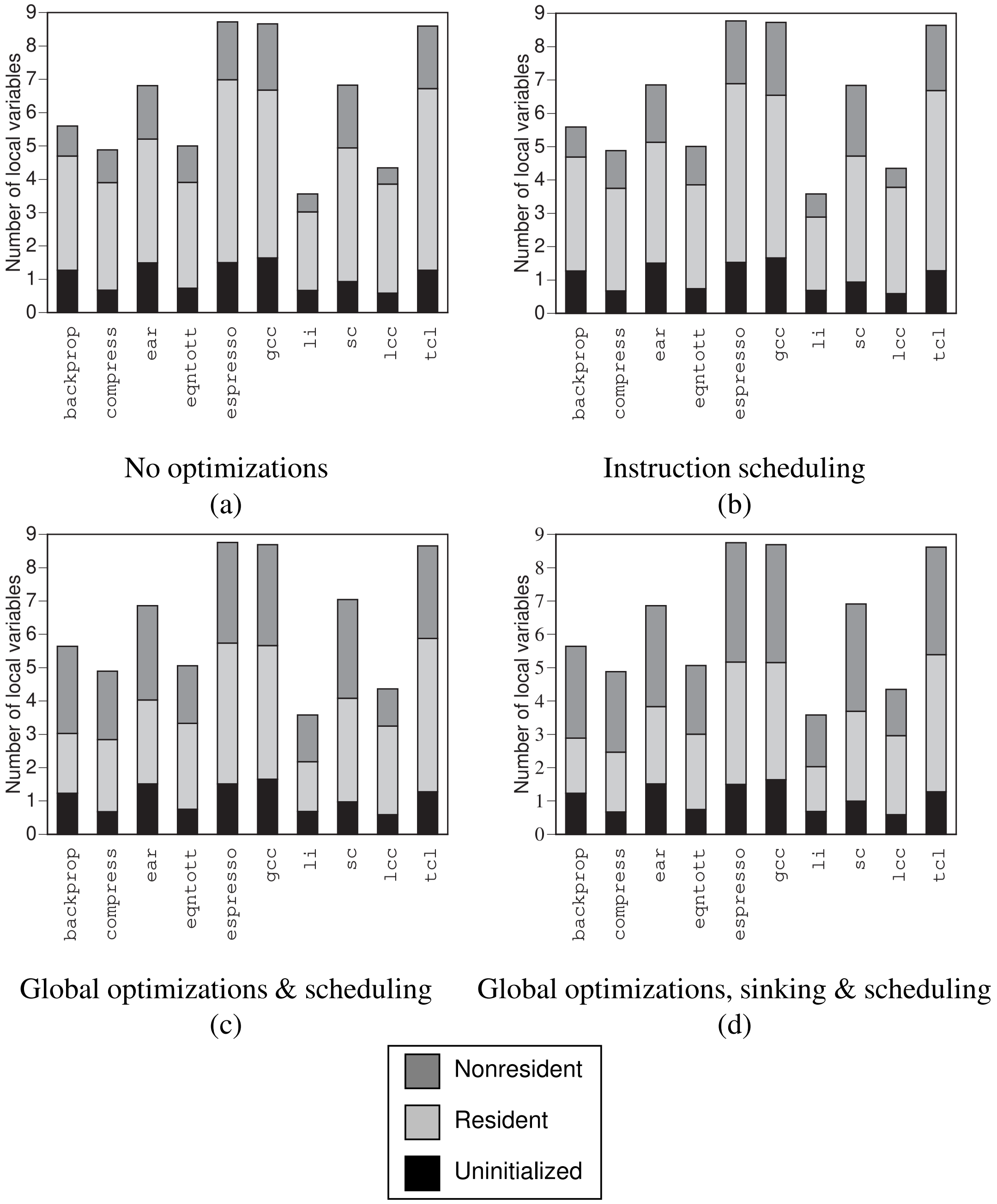


Figure 4.10: Average number of nonresident variables using available residence data-flow analysis.

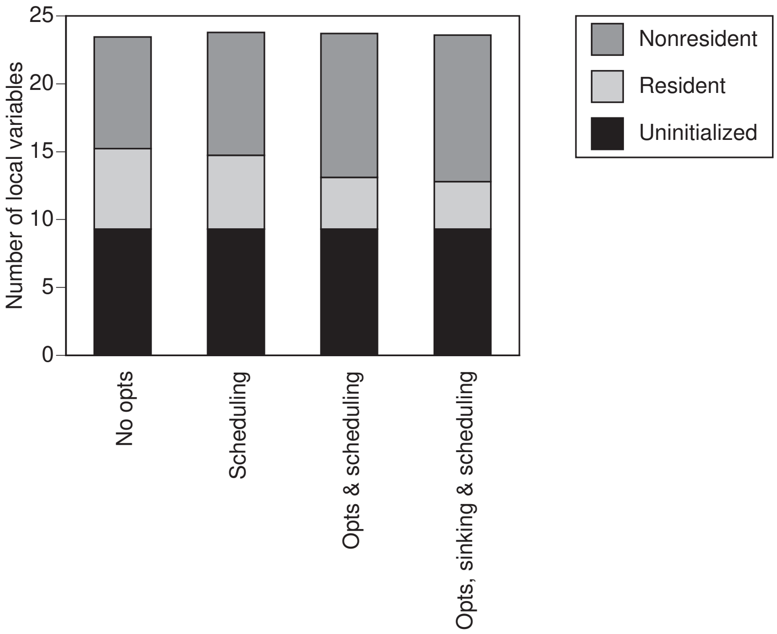


Figure 4.11: Average number of nonresident variables using available residence data-flow analysis for `triangle`.

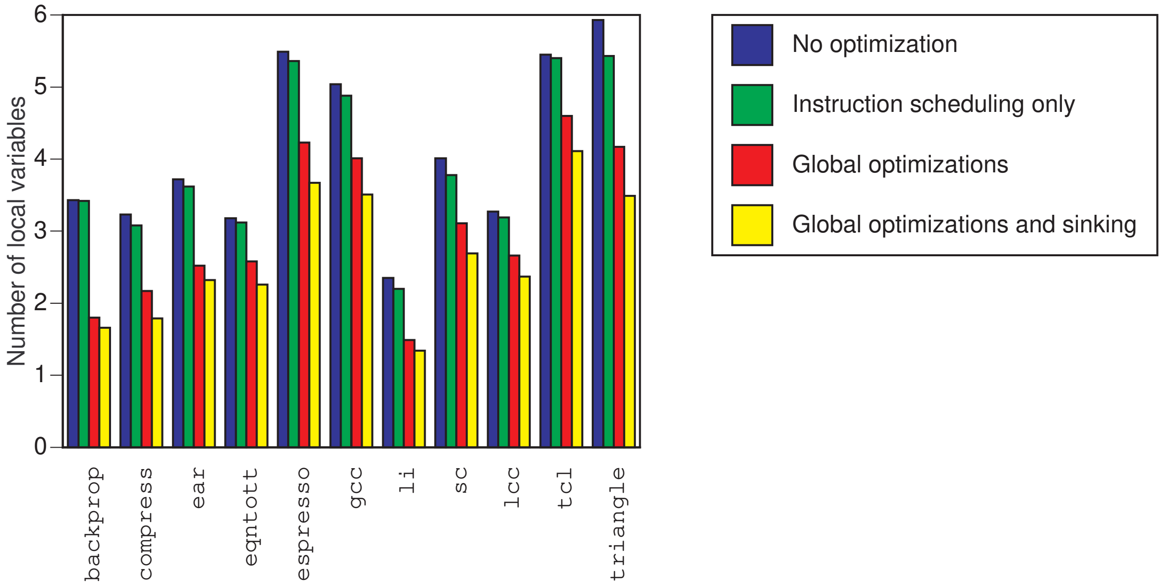


Figure 4.12: Comparison of average number of resident variables at a breakpoint, using available residence data-flow analysis.

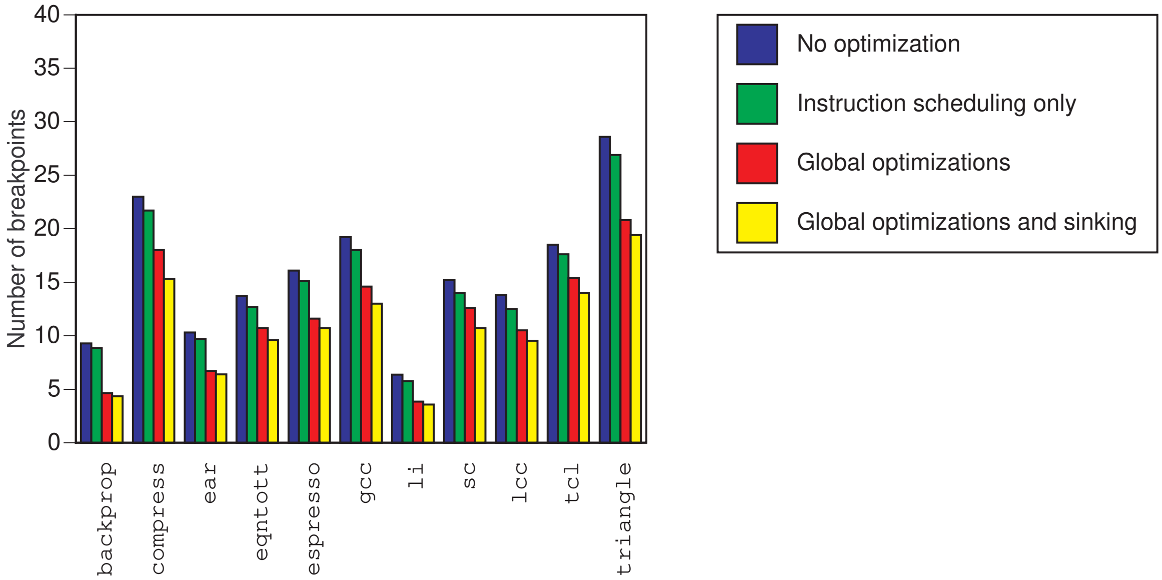


Figure 4.13: Comparison of average length of a variable's live range.

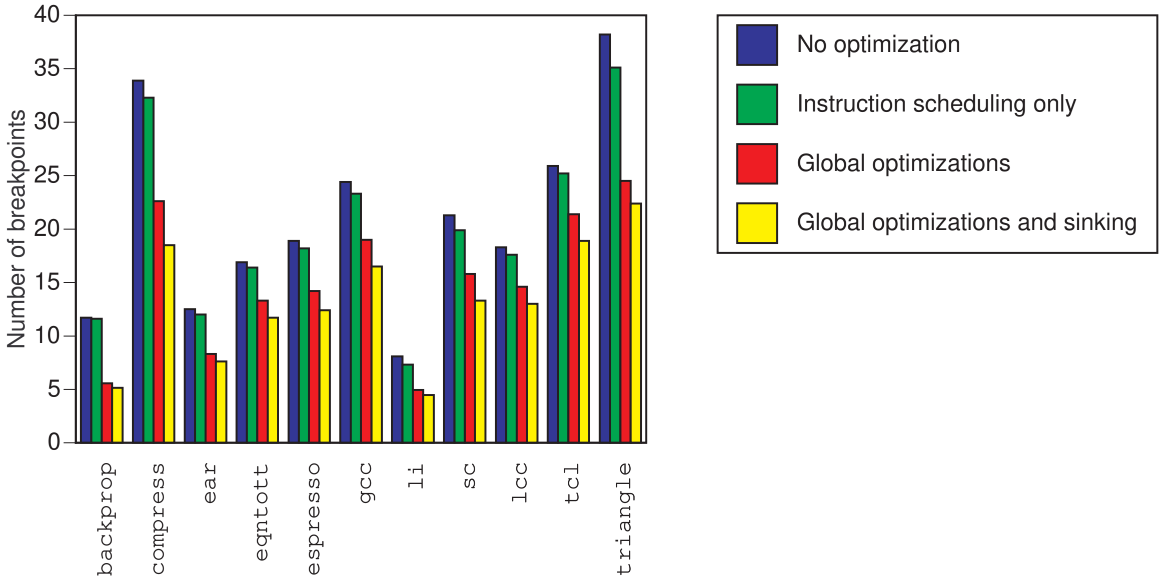


Figure 4.14: Comparison of average number of breakpoints a variable is resident using available residence data-flow analysis.

The charts in Figure 4.15 measure the effectiveness of using the available residence data-flow analysis over using live range information. These charts directly compare the average number of breakpoints where a variable is resident using the available residence analysis, against the number of breakpoints that fall within a variable's live range. Without global optimizations (Figures 4.15(a) and 4.15(b)), the available residence data-flow analysis is able to extend by 20–50% the number of breakpoints where a variable is detected as resident. When global optimizations are enabled (Figures 4.15(c)–(d)), the available residence data-flow analysis is able to extend by about 20% the number of breakpoints where a variable is detected as resident. The reason for the reduced effectiveness of available residence is that the extra register pressure induced by global optimizations causes the register allocator to re-use physical registers sooner.

4.6 Summary

In response to a user query of a variable's value, the debugger must first determine whether there exists a runtime value of the variable. Therefore, residency detection is a fundamental component of any debugger for optimized code. A simple approach to determining whether a variable V is resident is to use live range information computed by the compiler. Using live range information is conservative, however, because the compiler — in an attempt to reuse registers as aggressively as possible — tries to build live ranges that are as concise as possible; this is in contrast to the goals of the debugger, which are to provide accurate information about a variable across the largest region of a program.

In this chapter, I have described analyses that allow the debugger to provide the user with precise information at those program points that lie outside of a variable's live range. I have introduced a data-flow analysis algorithm that allows the debugger to extend beyond the last uses of a variable's value, the points where the variable is reported as resident; the measurements I present in Section 4.5 show that this analysis can increase by 20–50% the number of breakpoints where a variable is reported as resident. Furthermore, I have shown that by using reaching analysis to detect uninitialized variables, the debugger can significantly reduce the number of variables that are reported as nonresident; the measurements I present in Section 4.5 show that by detecting uninitialized variables, the

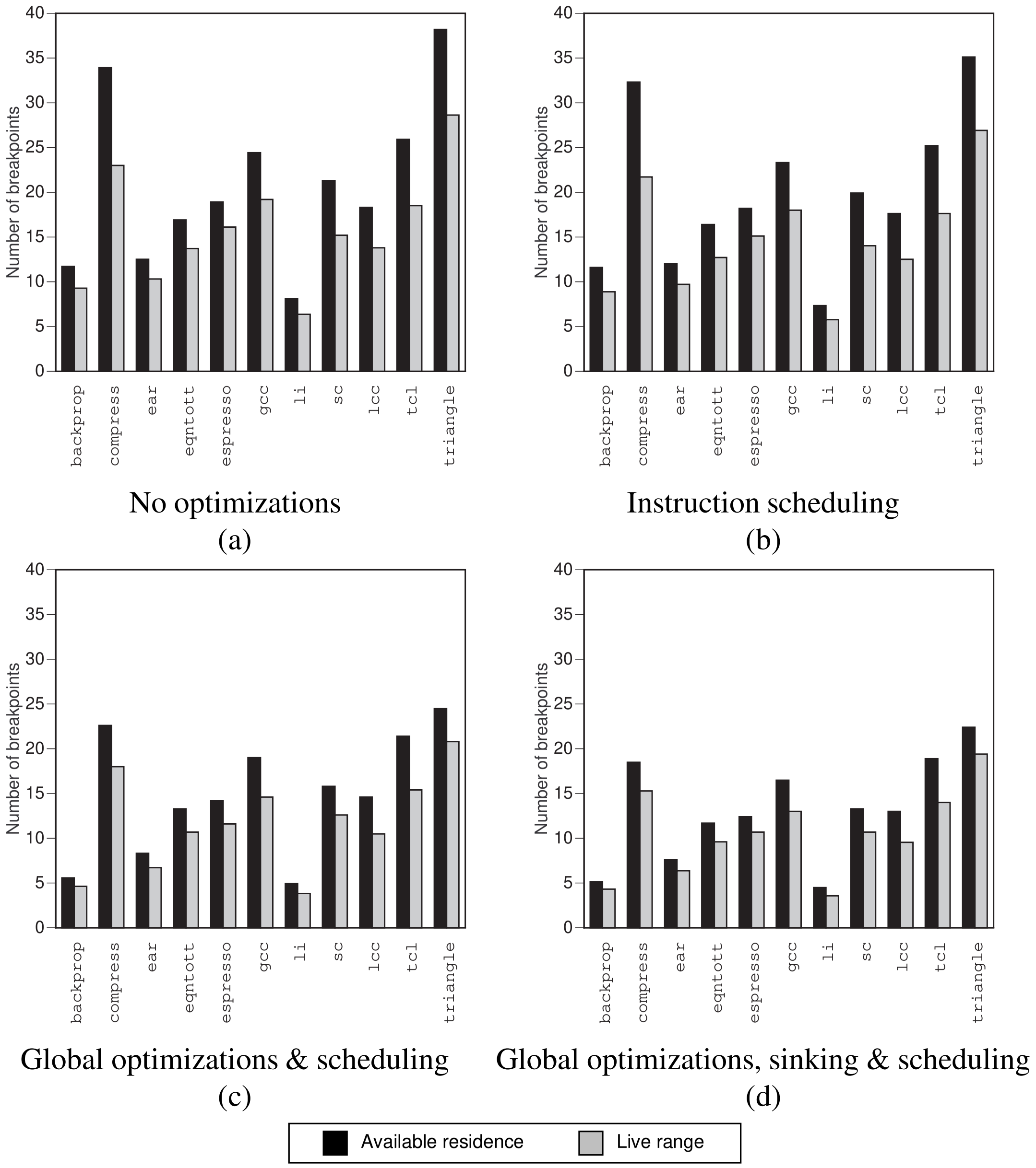


Figure 4.15: Average number of breakpoints a variable is resident.

debugger can provide a more accurate classification for 12–22% of the local variables that are in scope at a breakpoint. These analyses significantly reduce the number of breakpoints at which a variable is reported as nonresident; therefore, these analyses enable the debugger to perform aggressively its basic task of retrieving source-level values from the runtime state.

The measurements I present in Section 4.5, also show that optimizations other than register allocation, such as partial redundancy elimination, dead code elimination, and so on, influence the number of nonresident variables: by eliminating and shrinking live ranges, and by increasing register pressure, optimizations make it more difficult for the debugger to find a runtime value for a variable. In the next chapter, I explore the other data-value problems that these optimizations cause for the debugger.

Chapter 5

Detecting Endangered Variables

In this chapter, I concentrate on algorithms for detecting variables that are endangered at a break. I describe how compiler optimizations cause endangered variables and identify core transformations that cause endangerment. In the first section I provide an overview of optimizing compiler transformations. In the following sections, I present algorithms for detecting endangered variables. The description of the algorithms for detecting endangered variables is split into several sections. In Section 5.2, I describe algorithms for detecting endangered variables caused by local instruction scheduling. In Section 5.3, I describe algorithms for detecting endangered variables caused by local optimizations. In Section 5.4, I describe algorithms for detecting endangered variables caused by register coalescing. In Section 5.5, I describe algorithms for detecting endangered variables caused by dead code elimination and code hoisting; these two are the core global transformations that capture the effects of *global* optimizations that cause endangered variables. In Section 5.6, I describe how my algorithms can be extended to handle optimizations that hoist code speculatively. In Section 5.7 I describe the extensions necessary in the `cmcc` compiler to gather the information for detecting endangered variables. At the end of this chapter, I present results of an empirical study of how optimizations may affect a debugger's ability to present to the user the expected value of a variable.

5.1 Compiler transformations

Modern optimizing compilers perform a large number of optimizations, but very few of these optimizations actually cause endangered variables. Since the debugger interacts with the user, only values of *source program* variables are of interest; compiler-internals (temporaries) are never visible to the user. Therefore, only transformations that affect assignments to source-level variables cause endangerment. Transformations cause endangered variables by either *eliminating* or *moving* assignments, so that variable values are not updated in the manner specified by the source.

5.1.1 Code elimination

Compiler optimizations may eliminate an assignment A for one of three reasons:

1. If the value computed by A is never used, then A is *useless* and can be eliminated. Elimination of assignments whose value is never used is called *dead assignment elimination*. Dead assignment elimination causes endangerment when an assignment that assigns the expected value of a variable at a break has been eliminated.

2. If the value computed by A is already computed on all paths leading to A, then A is *available* and can be eliminated (A would not have changed the value of the variable being assigned). This transformation is usually performed by partial redundancy elimination (which subsumes common subexpression elimination). Elimination of an available assignment *does not* cause endangerment since the value of the assigned variable would not have been changed by the eliminated assignment.

3. Register move instructions that have the same source and destination register are ineffectual and can be eliminated by optimizations. To eliminate move instructions, register allocation attempts to assign the same register to the source and destination operands of a move. This optimization is known as *register coalescing* or *subsumption*. Register coalescing causes endangered variables when an eliminated move is a source definition.

Elimination of useless assignments is generally called *dead code elimination*, which also refers to elimination of code that is *unreachable*. I distinguish elimination of unreachable code from elimination of useless assignments by explicitly referring to the former transformation as unreachable code elimination. In the context of this thesis, the term dead code elimination is used to refer to the elimination of useless assignments.

Induction variable elimination is another transformation that eliminates useless assignments. This transformation eliminates assignments that are induction variable updates (e.g., `i++`), when the assigned value is used only by the eliminated update. I consider induction variable elimination to be a special form of dead assignment elimination[1].

Programmers generally don't write code with explicit dead assignments and opportunities for dead code elimination are usually exposed by other transformations such as copy propagation. But when a combination of transformations leads dead code elimination to eliminate an assignment to a variable, a portion of the variable's live range is in effect eliminated, allowing the register allocator to reuse any register assigned to the variable. Thus rather than creating an endangered variable, dead code elimination may instead indirectly create a nonresident variable. The empirical results of Section 5.8 support this observation.

5.1.2 Code movement

Compiler transformations that move assignments include instruction scheduling and global code motion. Instruction scheduling reorders and interleaves instruction sequences from different source statements, causing variable updates to occur out of the order expected in the source. I have already illustrated the effects of instruction scheduling in the example of Figure 2.1 (Section 2.2.1). Instruction scheduling can also reorder the execution of function calls; this issue is discussed in more detail in Section 5.2.

Code motion algorithms move an expression either upwards against the direction of control flow (code hoisting) or downwards towards the direction of control flow (code sinking). Hoisting of assignments causes endangerment by prematurely updating variable values, while sinking of assignments causes endangerment by delaying variable updates. Examples of code hoisting optimizations include partial redundancy elimination [76, 30, 39, 64],

[1] In fact in `cmcc` and other compilers, induction variable elimination is implemented as a simple variation on dead assignment elimination [30].

loop invariant code motion [7], and global instruction scheduling algorithms that perform non-speculative hoisting of instructions [14]. Examples of code sinking optimizations include partial dead code elimination [65], unspeculation [41], global instruction scheduling algorithms that sink code past conditional branches [28], superblock dead code elimination [29], and forward propagation [18].

Code motion and elimination are related, since some code motion algorithms operate by computing the set of program points where insertions of expressions render other expressions either available [76] or dead [65]. For instance, the Morel and Renvoise partial redundancy algorithm [76] and its variants [30, 39, 62] compute the set of program points where the insertion of an expression E will make another partially redundant instance of E fully redundant. The net effect of this transformation is a hoisting of E. Similarly, the partial dead code elimination algorithm described in [65] computes the set of program points where the insertion of an assignment A will make other partially dead instances of A candidates for dead assignment elimination. The net effect of this transformation is a sinking of A.

Without loss of generality, I consider only partial redundancy elimination in the rest of this thesis, although I will continue to use the term code hoisting in some places. In other words, I assume that code hoisting *eliminates* an available expression from one point in the program after *inserting* one or more copies of the expression at other locations. The other code hoisting transformations explicitly relocate either a particular operation (e.g., loop invariant code motion [7]) or a particular instruction (e.g., global scheduling [14] or superblock optimization [29]). In these cases, I assume that the moved operation is *eliminated* from its original location and *inserted* into its new location. Similarly, I consider only partial dead code elimination [65] while continuing to use the term code sinking; I make the same elimination and insertion assumptions about an explicitly sunk operation, with the difference that the eliminated assignments are considered useless (rather than available).

Data-value problems caused by global optimizations can be managed by concentrating on only two key transformations: code hoisting and dead code elimination. We do not need to deal with code sinking explicitly because dead code elimination captures the effects of code sinking: the assignments that are *eliminated* by code sinking are the ones that cause variables to be endangered. In the case of code hoisting, on the other hand, assignments that are *inserted* cause endangerment by prematurely updating the value of a variable; the assignments that become redundant and are consequently eliminated do not cause endangerment.

5.1.3 Code motion invariants

Code motion is constrained by *safety* considerations; that is, a computation cannot be introduced into a path where it did not exist before. Therefore, code hoisting optimizations copy an expression E from a block B to one or more blocks that are *post-dominated* by B, while code sinking optimizations move an expression E from a block B to one or more blocks that are *dominated* by B. By taking advantage of these code motion invariants, the algorithms for detecting endangered variables are greatly simplified. In fact, it is these invariants that have allowed me to produce a solution to the problem that is significantly simpler than the approaches described by Wismueller [107] and Copperman [36].

Not all compiler optimizations satisfy these safety constraints. Global instruction scheduling algorithms that schedule instructions for *speculative execution* [43, 14, 28], can hoist an instruction I from a block B to another block that is not post-dominated by B. Since I is being introduced into a program path where it did not exist before, I should be selected from the execution path with the highest execution probability. Therefore, speculative code motion is usually guided by profile information. It is assumed that I will not cause a fault, or if it does, the fault is suppressed until the control dependencies of I are resolved [34, 89, 74]. As the amount of instruction-level parallelism offered by processors increases, the compiler must look harder for instructions that can be scheduled for parallel execution. Speculative instruction scheduling is becoming an increasingly important technique for extracting such parallelism from a program. I address speculative code motion in Section 5.6.

5.1.4 Optimizations that do not cause endangerment

Many scalar optimizations, such as strength reduction, constant folding, and constant propagation [7], do not affect assignments to source variables but create new opportunities for dead assignment elimination and thus indirectly contribute to creating nonresident and endangered variables. For example, constant propagation or copy propagation may eliminate all uses of an assignment's left hand side, thus making the assignment a candidate for elimination by dead assignment elimination. Induction variable strength reduction and linear function test replacement may eliminate all uses of an induction variable inside of a loop except for the induction variable update, making the update assignment a candidate for elimination by induction variable elimination.

Some optimizations allow the debugger to infer source values from compiler temporaries that replace eliminated variables. For example, strength reduction can replace address expressions that are linear functions of a loop induction variable (e.g., `A[i+1]`) with a new address temporary (e.g., `tmp = &A[i+1]`). If the only remaining use of the induction variable is the update expression (e.g., `i++`), then the variable can be eliminated (loop induction variable elimination). However, the value of the eliminated induction variable can be derived from the new address temporary (i.e., `i=((tmp-A)>>2)-1`). Such an approach is used in the DOC [38] and CXdb [20] debuggers. There are other situations where the overall effect of a series of transformations is the replacement of a source-level variable with a compiler temporary, again allowing the debugger to infer values from compiler temporaries; I address this issue in more detail in Section 5.5.5.

Control-flow transformations, such as loop unrolling and tail duplication, change the control flow graph by duplicating basic blocks, but do not reorder the execution of source-level assignments and therefore do not cause data-value problems. Code duplicating control-flow transformations, are usually performed to expand the scope of other optimizations.

5.2 Endangered variables caused by instruction scheduling

Instruction scheduling changes the sequence in which source-level expressions are computed by reordering or interleaving instruction sequences from different source statements. Endangerment occurs when an assignment to a variable is executed out of order with respect to the control reference statement. Such an assignment makes a variable's actual value different from its expected value by either prematurely overwriting the expected value with a future value or by delaying the update of the expected value. Therefore, there are two ways in which a variable V may be noncurrent at a breakpoint B. Either an assignment to V has prematurely overwritten V's expected value, in which case V is a *roll back* variable, or the assignment that updates the expected value of V has been delayed until after the breakpoint, in which case V is a *roll forward* variable [49].

To detect noncurrent variables, the debugger must detect which operations have executed out of order and how these operations affect the source-level state (i.e., variables and memory locations). Operations that affect source state include assignments to source variables and function calls; for conciseness, I only mention assignments in the rest of this section. Detecting which assignments have executed out of order requires precise modeling of the order in which assignments are expected to execute in the source, as well as the order in which they actually execute in the object code. Since the effects of instruction scheduling are localized, I need only model what occurs within basic blocks rather than what arises globally among basic blocks. My approach is similar to Hennessy's [49]; but my model also considers values held in the physical registers of the machine and computed by individual instructions, whereas Hennessy's approach operates strictly on the machine-independent intermediate representation of the program.

5.2.1 Source execution order

I define the *canonical execution order* of source expressions to be the order in which expressions in the source program are supposed to execute according to the semantics of the source language. During source-level debugging, the user expects variables to be updated according to the canonical execution order of assignments in the source. Thus, to determine expected source values at breaks, the debugger must accurately model the canonical execution order of source assignments.

The evaluation order of source assignments may not always be precisely defined by the source language semantics; thus, the expected value of a variable relative to a source breakpoint is not always well defined. Usually, the canonical execution order of side effects from different source statements is well defined — that is, the semantics specify that statements execute in sequence. However, in a C language statement containing multiple side effecting expressions, the compiler is free to choose the evaluation order of the side effecting expressions. For example, in the code fragment of Figure 5.1, the assignment in statement S_1 must execute before the assignments in S_2, and all assignments in S_2 must execute before the assignment in S_3, but the compiler is free to choose the evaluation order of the three assignments within statement S_2. Thus, at an asynchronous break occurring during the execution of expression `y++` within S_2, the expected value of `x` is not uniquely defined since this value can be assigned by either S_1 or by the expression `x++` of S_2; that is, if the actual value of `x` corresponds to either of these two values, then `x` is current.

In practice, most compilers impose an evaluation order during the translation from source to intermediate form. The debugger should not consider the compiler-defined evaluation order as the canonical execution order, otherwise the debugger will produce conservative and possibly misleading responses to the user. Considering again the example of Figure 5.1, the compiler may impose a left-to-right evaluation order so that `x++` is evaluated before `y++` at the intermediate representation level (i.e., before code generation). However, if the code generator delays the evaluation of `x++` until after the evaluation of `y++`, and the debugger uses the compiler-defined evaluation order as the canonical execution order, then the debugger will report `x` as noncurrent at a break occurring within `y++`. If the actual value of `x` is the value assigned by S_1, then this response is not accurate since the actual value of `x` is a valid expected value.

```
S1:  x = y + 2;
S2:  z = x++ + y++;
S3:  y = x + y;
```

Figure 5.1: Undefined evaluation order in C

To capture the canonical execution order of source assignments, I assign a sequence number $Seq(A)$ to each assignment A in the intermediate representation, where A corresponds to an assignment in the source. Given two assignments A_i and A_j, both in the same basic block, $Seq(A_i) < Seq(A_j)$ implies that A_i executes before A_j in the canonical execution order, while $Seq(A_i) = Seq(A_j)$ implies that the canonical execution order of A_i and A_j is undefined (their execution order is undefined in the source).

In the example of Figure 5.1, the three assignment expressions of statement S_2 all have the same sequence number. Note that the partial ordering defined by the Seq annotation captures the canonical execution order of statements within the same basic block, not the dependences that constrain the correct execution order(s).

5.2.2 Instruction execution order

To capture the execution order of instructions, I define a partial ordering among the instructions inside a single basic block: given two instruction I_i and I_j inside the same basic block, $I_i < I_j$ implies that I_i completes execution before I_j. The machine model I use in this thesis, assumes that $I_i < I_j$ if and only if I_i is scheduled before I_j (in other words, I assume precise interrupts); therefore, this partial ordering can be easily calculated by sequentially numbering all instructions inside of a basic block schedule. (V)LIW or statically scheduled superscalar machines (e.g., the iWarp [16]) can execute multiple instructions concurrently; therefore, $I_i = I_j$ implies that I_i and I_j complete execution concurrently.

The order in which source definition instructions complete execution determines the order in which source-level assignments actually execute. For each source definition I, the most important information is the sequence number of the assignment(s) for which I is generated and the variable(s) that is (are) assigned by I. I encapsulate this information in an *assignment descriptor*. An assignment descriptor is a triple $\langle I, N, V \rangle$ where I is the source definition, N is the sequence number of the assignment for which I is generated, and V is the variable that is assigned. Given an assignment descriptor $D = \langle I, N, V \rangle$, I define $Inst(D) = I$, $Seq(D) = N$ and $Var(D) = V$. Indirect assignments (e.g., `*p = ...`) may assign to any memory location and the debugger cannot tell with static information alone which memory location is affected by an indirect source definition. If the source definition I is generated for an indirect assignment, then $V = *$, where the special symbol $*$ represents the set of variables that reside in memory.

Note that the compiler may have alias information describing the set of variables potentially aliased by an address expression. Such information, for example, can be used by the scheduler to re-order indirect loads and stores. However, the debugger should not use such information for refining the set of suspect variables, since a program bug may invalidate the assumptions made when gathering alias information.

5.2.3 Endangered variables

To determine if there are noncurrent or suspect variables at a break, the debugger has to find out if any operations with side effects on the user-visible state executed out of sequence with respect to the control reference statement. Let A be an assignment to a source variable V; there are two ways in which A may have executed out of order at a break $B = \langle S, O \rangle$:

- If A is supposed to execute before S in the canonical execution order but is scheduled to complete execution after O, then the execution of A has been delayed at B and V is a roll forward variable at B, denoted $ForwardVar(V, B)$.
- If A is supposed to execute after S in the canonical execution order but is scheduled to complete execution before O, then A has executed prematurely at B and V is a roll back variable at B, denoted $BackVar(V, B)$.

The assignment descriptors inside $Block(O)$ contain sufficient information to determine whether source assignments have executed out of order and how the source-level state has been affected. Given a break $\langle S, O \rangle$ and an assignment descriptor D in the same basic block as O:

$$[(Seq(D) < Seq(S)) \land (Inst(D) \geq O)] \Rightarrow ForwardVar(Var(D), \langle S, O \rangle) \tag{5.1}$$

$$[(Seq(D) > Seq(S)) \land (Inst(D) < O)] \Rightarrow BackVar(Var(D), \langle S, O \rangle) \tag{5.2}$$

In the case that $Var(D) = *$, the variable that is noncurrent depends on the memory address value used by $Inst(D)$ ($Inst(D)$ in this case must be a store instruction). The debugger must be able to recover this address value to determine the runtime location affected, otherwise $Inst(D)$ causes suspect variables. In the case of function calls, the debugger cannot determine which memory locations have been accessed. Thus, out-of-order function calls and out-of-order indirect assignments whose address expressions cannot be recovered both cause suspect variables.

5.2.4 Recovery

The debugger can use partially computed results available in the physical registers to perform recovery. Recovery can allow the debugger to provide more precise information to the user, in several ways:

1. Recovering address expressions of indirect assignments allows the debugger to reclassify some suspect variables as either noncurrent or current. An out-of-order

indirect assignment A causes all memory variables to be suspect. The number of variables in memory can be potentially large (e.g., consider the number of heap objects). Recovery of A's address expression will re-classify the affected variable to noncurrent. An unaffected suspect variable V may be re-classified as current if A is the only out-of-order operation causing V to be suspect. Recovery of address expression values is called *address recovery*.

2. Recovering the values assigned by source definitions allows the debugger to provide the expected values of some roll forward variables. If a source definition I assigns a roll forward variable V's expected value and the value assigned by I is recoverable, then the debugger can provide V's expected value. Recovery of values assigned by source definitions is called *assignment recovery*.

To perform recovery, the debugger can take the following strategy: First, the debugger attempts address recovery of indirect source assignments that have executed out of order. Since address recovery may make suspect variables either noncurrent or current, the status of suspect variables is re-evaluated after this recovery is performed. Then, the debugger attempts assignment recovery of source definitions whose execution has been delayed (as determined by Equation 5.1), possibly enabling the debugger to provide the expected values of some noncurrent variables.

To allow recovery, I record for each source definition I the position (in the basic block) of the last and next *local* definitions of its source and destination registers, denoted $\textit{LastDef}_I(R)$ and $\textit{NextDef}_I(R)$, where R is a source or destination register of I. If no local last definition of R exists, then $\textit{LastDef}_I(R)$ is -1. Similarly, if no local next definition of R exists, then $\textit{NextDef}_I(R)$ is set to a value beyond the last position in I's basic block. Note that the last and next definition information can be easily computed with a simple local analysis of the object code. This analysis can be done by the debugger and thus the information does not need to be communicated to the debugger by the compiler.

The set of source registers of an instruction I is denoted by $\textit{SourceRegs}(I)$ and the destination register of an instruction I is denoted by $\textit{DestReg}(I)$[2]. If I is a load or store

[2]Some machines have instructions with multiple destination registers. For example, the iWarp and PowerPC have loads with auto-update addressing modes, requiring multiple destination registers (one for

instruction, then I's set of address registers (e.g., base and index registers) is denoted by *AddressRegs*(I).

Address recovery

Let I be an indirect source definition (a store instruction). At a stopping instruction O, the address of the location stored by I is recoverable if the values in I's address registers are available (i.e., have been computed but not subsequently overwritten):

$$\forall R \in AddressRegs(I) : (\textit{LastDef}_I(R) < O) \wedge (\textit{NextDef}_I(R) \geq O)$$

Assignment recovery

Let I be a source definition. At a stopping instruction O, the value assigned by I can be reconstructed from the runtime state if the values used by the source registers of I are available and execution of I does not cause an exception:

$\forall R \in SourceRegs(I) : (\textit{LastDef}_I(R) < O) \wedge (\textit{NextDef}_I(R) \geq O)$ and I is safe to execute

If the values used by the source registers of I are available, then I can be executed (actually interpreted by the debugger). However, the debugger must be prepared to handle the case that I may fault. The debugger cannot perform this computation if I is a function call instruction, or if I is a load instruction and there exists memory locations that are endangered.

Note, that this recovery scheme can be extended to recover values from more than one instruction by computing an executable backward slice [103, 104].

the loaded value and one for the updated address). The definition of $DestReg$ can be easily extended to accommodate a set of registers without major changes to the rest of our discussion. But for the sake of simplicity, I stick to the assumption of only one destination register.

S_1:	`d = f+g;`	I_1:	`load R1,4(sp)`	
S_2:	`b = c*a;`	I_2:	`load R2,8(sp)`	
S_3:	`*p = a;`	I_3:	`store 0(R4),R3`	$\langle I_3, 3, *\rangle$
		I_4:	`fpmul R4,R5,R3`	$\langle I_4, 2, \texttt{b}\rangle$
		I_5:	`fpadd R6,R1,R2`	$\langle I_5, 1, \texttt{d}\rangle$

(a) (b) (c)

Figure 5.2: Example: Instruction scheduling (a) Source code (b) Object code after scheduling and register allocation (c) Assignment descriptors.

5.2.5 Example

The example of Figure 1.1 is reproduced in Figure 5.2, and is used to illustrate the techniques introduced in this section. Sequence numbers are assigned to source statements as follows: $Seq(S_1) = 1$, $Seq(S_2) = 2$ and $Seq(S_3) = 3$. There are three source definitions with assignment descriptors: $\langle I_3, 3, *\rangle$, $\langle I_4, 2, \texttt{b}\rangle$ and $\langle I_5, 1, \texttt{d}\rangle$.

Break	Roll forward	Roll back	Recoverable
$\langle S_3, I_3\rangle$	$\langle I_4, 2, \texttt{b}\rangle$, $\langle I_5, 1, \texttt{d}\rangle$		I_4, I_5
$\langle S_2, I_4\rangle$	$\langle I_5, 1, \texttt{d}\rangle$	$\langle I_3, 3, *\rangle$	I_5, $AddressRegs(I_3)$
$\langle S_1, I_5\rangle$		$\langle I_3, 3, *\rangle$, $\langle I_4, 2, \texttt{b}\rangle$	

Table 5.1: Assignment descriptors of out-of-order assignments and recoverable source definitions at breaks in code of Figure 5.2.

Table 5.1 shows the assignment descriptors that cause endangered variables at various breaks in Figure 5.2. At break $\langle S_3, I_3\rangle$, the executions of I_4 and I_5 have been delayed, causing variables `b` and `d` to be noncurrent. However, the source values used by these instructions are available (the values in registers `R1`, `R2`, `R3` and `R5`). Therefore values computed by both of these instructions are recoverable (although the debugger must take care because floating-point computations may fault) and the expected values of `b` and `d` can be presented to the user. At break $\langle S_2, I_4\rangle$, I_5 is again delayed causing `d` to be noncurrent; I_3 is now executed prematurely. I_3 is an indirect store instruction, but fortunately, the value used by the base address register of I_3 (register `R4`) is still available; thus, I_3 causes a noncurrent variable (rather than suspect variables). At break $\langle S_1, I_5\rangle$, I_3 is again executed prematurely, but unfortunately, the value in register `R4` has been overwritten by instruction I_4 (which is also executed prematurely); thus, I_3 causes variables that are in memory to be suspect.

5.3 Endangered variables caused by local optimizations

In this section, I focus on local optimizations that cause endangered variables. Compilers usually perform local optimizations in preparation for global optimizations and to improve the quality of generated instruction sequences (peephole optimizations). Local optimizations are inexpensive to perform and are typically the first optimizations enabled when the user specifies a low level of optimization (e.g., `-O1`).

Local optimizations cause endangered variables by eliminating source assignments. As I mentioned in Section 5.1, an assignment may be eliminated because it is either available or dead. I first look at assignments that are eliminated because they are available locally; such elimination does not cause data-value problems for the debugger. I then concentrate on assignments that are eliminated because they are dead locally. Elimination of locally dead assignments causes endangered variables, but the endangered variables can be detected easily with a simple extension to the approach presented in Section 5.2.

5.3.1 Locally available assignments

Locally available expressions are typically discovered by a local common subexpression elimination phase, and by peephole optimizations that eliminate instructions that are redundant because of an exact copy of the eliminated instruction earlier in the basic block. Local availability optimizations can also be performed on extended basic blocks (i.e., straight-line code sequences with a single entry and one or more exit points).

Elimination of a locally available assignment A does not cause endangerment since the value assigned by A has already been assigned earlier in the block and thus execution of A is ineffectual. That is, the value assigned by A has already been assigned once earlier and the value of the assigned variable will not be changed by A.

Consider the example source and object codes shown in Figure 5.3. There are no other assignments to the variable `x` in the source other than the assignments in statements S_2 and S_4. Instruction I_i has been generated for statement S_2 and there is only one assignment

Source		Object		
S_1:	`...`	I_1:	`...`	
S_2:	`x=y+z;`		`...`	
S_3:	`...`	I_i:	`add Rx,Ry,Rz`	$\langle I_i, Seq(S_2), \mathtt{x}\rangle$
S_4:	`x=y+z;  /* available */`		`...`	
S_5:	`...`	I_n:	`...`	

Figure 5.3: Example: Locally available assignment.

descriptor: $\langle I_i, Seq(S_2), \mathtt{x}\rangle$. The assignment of statement S_4 is redundant because an identical copy of this assignment is executed earlier in the block at statement S_2; therefore, no code has been generated for S_4.

Table 5.2 lists the noncurrent variables assuming various breaks $\langle S, O\rangle$ in the example of Figure 5.3. I consider all possible breaks by considering stopping instructions O occuring before and after instruction I_i ($O < I_i$ and $O > I_i$ respectively) and considering control reference statements occuring before, in between, and after statements S_2 and S_4 (S_1, S_3 and S_5, respectively).

The interesting case to consider is the case where the control reference statement S is statement S_5 (the last row of Table 5.2) because at this statement the expected value of x is the value assigned by the eliminated statement S_4 (at all other breaks, the actual and expected values of x are not affected by optimizations). At S_5 the expected value of x is the value assigned by statement S_4, which is the same as the value assigned by S_2. If the stopping instruction O is before I_i ($O < I_i$), then the debugger (correctly) detects that x is a roll forward variable and recovering the value assigned by I_i will correctly recover the expected value of x (even though the value recovered is the value assigned by S_2). If the stopping instruction O is after I_i ($O > I_i$), then the actual value of x is the value assigned by I_i which is the source-level value assigned by statement S_2. Statements S_2 and S_4, however, assign the same value to x and thus x is current.

As this example illustrates, locally available assignments can be eliminated without worrying about endangered variables.

S	Stopping instruction O	
	$O < I_i$	$O > I_i$
S_1	—	$BackVar(\mathtt{x}, \langle S, O\rangle)$
S_3	$ForwardVar(\mathtt{x}, \langle S, O\rangle)$	—
S_5	$ForwardVar(\mathtt{x}, \langle S, O\rangle)$	—

Table 5.2: Noncurrent variables at various breaks $\langle S, O\rangle$ in Figure 5.3.

Source		Object	
S_1:	`...`	I_1:	`...`
S_2:	`x=w-v;   /* dead */`		`...`
S_3:	`...`	I_i:	`add Rx,Ry,Rz` $\langle I_i, Seq(S_4), \mathtt{x}\rangle$
S_4:	`x=y+z;`		`...`
S_5:	`...`	I_n:	`...`

Figure 5.4: Example: Locally dead assignment.

5.3.2 Locally dead assignments

Local dead code elimination eliminates an assignment A that assigns to a variable V, when there are no uses of V after A and there is a later assignment A' that also assigns to V in the same basic block. V is noncurrent at breaks $\langle S, O\rangle$ where the control reference statement S is between A and A' and there are no source definitions that have prematurely assigned to V at O — in other words, the expected value of V is from A, and no assignment to V has prematurely executed at O.

Figure 5.4 shows an example of a locally dead assignment. There are no uses of x between statements S_2 and S_4 and local dead code elimination has eliminated the assignment of statement S_2. I_i is generated for statement S_4 and there is only one descriptor: $\langle I_i, Seq(S_4), \mathtt{x}\rangle$.

Table 5.3 lists the various breaks possible in Figure 5.4. The interesting cases are those

S	Stopping instruction O	
	$O < I_i$	$O > I_i$
S_1	—	$BackVar(\mathtt{x}, \langle S, O\rangle)$
S_3	$Dead(\mathtt{x}, \langle S, O\rangle)$	$BackVar(\mathtt{x}, \langle S, O\rangle)$
S_5	$ForwardVar(\mathtt{x}, \langle S, O\rangle)$	—

Table 5.3: Noncurrent variables at various breaks $\langle S, O\rangle$ in Figure 5.4.

where the expected value of x is assigned by the eliminated assignment of statement S_2; these are breaks where the control reference statement is statement S_3 (second row of Table 5.3). If the stopping instruction O is before I_i ($O < I_i$), the actual value of x is the value assigned by the last executed assignment to x before this code fragment. x is noncurrent because the expected value is assigned by S_2 which is eliminated. If the stopping instruction is after I_i ($O > I_i$) then x is noncurrent because I_i has prematurely updated x with the value assigned by S_4.

When local dead code elimination eliminates an assignment A because of a later assignment A', a *dead assignment descriptor* is created for A. A dead assignment descriptor D is a triple $D = \langle I, N, V \rangle$ where I is the source definition generated for A', N is the sequence number of the dead assignment with which D is associated ($N = Seq(A)$), and V is the variable assigned by the dead assignment. Given a descriptor $D = \langle I, N, V \rangle$, I define $Inst(D) = I$, $Seq(D) = N$ and $Var(D) = V$.

The dead assignment descriptor can be used to detect variables that are noncurrent because of local dead code elimination.

Let $Dead(V, B)$ denote a variable V that is noncurrent because of an eliminated assignment, at a break B. (In Figure 5.4, $Dead(\mathtt{x}, \langle S_3, O \rangle)$ for $O < I_i$.) Given a break $\langle S, O \rangle$ and a dead assignment descriptor D in the same basic block as O:

$$[(Seq(D) < Seq(S)) \wedge (Inst(D) \geq O) \quad \wedge \quad \neg ForwardVar(Var(D), \langle S, O \rangle)] \\ \Rightarrow Dead(Var(D), \langle S, O \rangle) \tag{5.3}$$

The three clauses in formula 5.3 are justified as follows: First, the dead assignment A represented by D causes endangerment only if A was supposed to execute with respect to S; thus, the condition $Seq(D) < Seq(S)$. Second, the actual value of $Var(D)$ must not be the value assigned by A', the assignment that causes A to be dead; thus, the condition $(Inst(D) \geq O)$. Finally, the expected value of $Var(A)$ may be assigned by A', and therefore, $Var(D)$ may be noncurrent because A' has not yet executed; the condition $\neg ForwardVar(Var(D), \langle S, O \rangle)$ makes sure that $Var(D)$ is not already noncurrent because of A'.

S	Stopping instruction O	
	$O < I_2$	$O > I_2$
S_1	—	$BackVar(\{\texttt{x,y}\}, \langle S, O\rangle)$
S_3	$ForwardVar(\texttt{y}, \langle S, O\rangle)$	$BackVar(\texttt{x}, \langle S, O\rangle)$
S_5	$ForwardVar(\{\texttt{x,y}\}, \langle S, O\rangle)$	—

Table 5.4: Noncurrent variables at various breaks $\langle S, O\rangle$ in Figure 5.5(a) after register coalescing.

5.4 Endangered variables caused by coalescing

Coalescing causes endangered variables when it eliminates a register move instruction that is a source definition. Consider again the source and object codes depicted in Figure 4.1, redrawn in Figure 5.5. Part (a) of this figure shows the source code, while parts (b) and (c) show the object code before and after register allocation, respectively. Before register allocation (Figure 5.5(b)), I_2 and I_4 are source definitions of variables `y` and `x` and there are two assignment descriptors: $\langle I_2, Seq(S_2), \texttt{y}\rangle$ and $\langle I_4, Seq(S_4), \texttt{x}\rangle$. Assume that the live ranges of `x` and `y` do not interfere. The register allocator coalesces `x` and `y`, assigning the same register `R1` to both variables. The source definition I_4 is subsequently eliminated and no instructions are generated for the assignment of statement S_4 (Figure 5.5(c)).

(a)	(b)	(c)
S_1: `...`	I_1: `...`	I_1: `...`
S_2: `y=u+v;`	I_2: `add Ry,Ru,Rv`	I_2: `add R1,R2,R3`
S_3: `...`	I_3: `...`	I_3: `...`
S_4: `x=y;`	I_4: `mov Rx,Ry`	
S_5: `...`	I_5: `...`	I_5: `...`

Figure 5.5: Example: Register coalescing (a) Source code (b) Instructions before register allocation (c) Instructions after register allocation and coalescing.

To capture the effects of coalescing, we can consider I_2 to be a source definition generated from *both* S_2 and S_4. That is, I_2 assigns to `y` the value assigned by S_2 and assigns to `x` the value assigned by S_4; $\langle I_4, Seq(S_2), \texttt{x}\rangle$ is changed to $\langle I_2, Seq(S_2), \texttt{x}\rangle$ to reflect this. Now, if execution stops somewhere between S_2 and S_4 in the source, but after I_2 in the object, `x` and `y` are both resident in `R1`. The actual value of `y` is the value assigned by S_2, while the actual value of `x` is the value assigned by S_4. Hence, `y` is current and `x` is

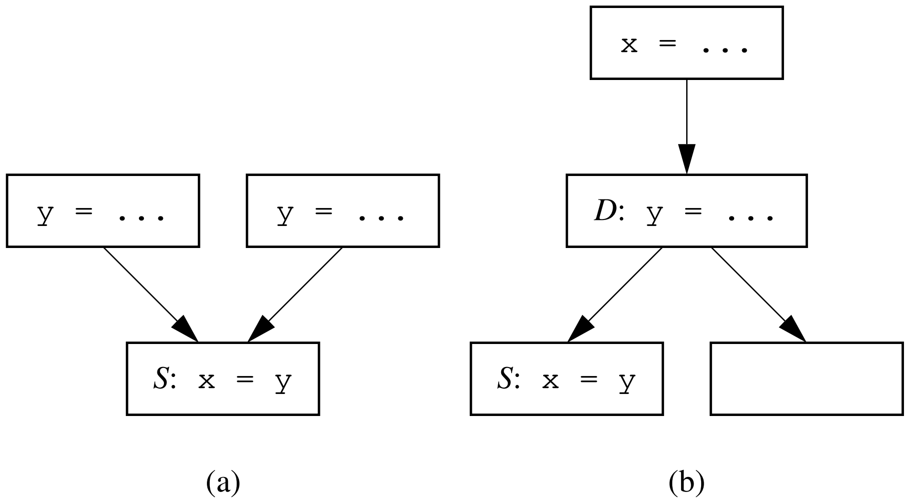


Figure 5.6: Global register coalescing

noncurrent. If execution stops after S_4 in the source, but after I_2 in the object, both x and y are current. Table 5.4 lists the noncurrent variables at various breaks in Figure 5.5.

A less precise way to deal with this situation is to consider S_4 as an eliminated (dead) assignment. Thus, at a break $\langle S, O \rangle$, where $Seq(S) > Seq(S_4)$ in the source and $O > I_2$ in the object, x is detected as nonresident since I_2 is an evicting definition of x. This is conservative, however, since R1 contains the value that would have been assigned to x by S_4, which is x's expected value.

In general, when coalescing eliminates a source definition I, an earlier instruction I' that defines $DestReg(I)$ is found that reaches $pre(I)$; I' then replaces I as the source definition. Figure 5.5 illustrates the simple case where the earlier definition of $DestReg(I)$ is within the same basic block as the eliminated copy operation. There are cases, however, where no earlier definition of $DestReg(I)$ may exist within the same basic block as the eliminated instruction I. Figure 5.6 illustrates other cases that can occur. In this figure, coalescing eliminates the copy x = y of statement S. In Figure 5.6(a), S post-dominates all reaching definitions of the register assigned to both x and y. Thus we can consider all reaching definitions as source definitions of x. However, this moves the definition of x into different basic blocks and results in multiple source definitions. This has ramifications on the endangerment detection algorithms, which now have to address global code movement. That is, we now have to deal with this situation as a global code motion problem: S has been hoisted out of its basic block. Moreover, if there exists a reaching definition D that is not post-dominated by S, as shown in Figure 5.6(b), D cannot be considered a source definition of x (we can, however, consider D as a speculative assignment to x).

Rather than perform a reaching definitions data-flow analysis or a graph search for each move instruction eliminated by coalescing, I avoid these two problems with a simple local solution: I model the source definition of S in both cases to be the *pre-amble* instruction of S's basic block. In other words, if there does not exist a prior reaching definition in the basic block, the pre-amble instruction is used as the source definition of S.

5.5 Endangered variables caused by global optimizations

In this section, I present a solution to the data-value problems caused by global transformations. I focus on the two core global transformations that caused endangerment: code hoisting and dead code elimination. I present an approach based on data-flow analysis to analyze and propagate the effects of these transformations. The data-flow analysis required to support the debugger is similar to the data-flow analysis performed for global optimization; in cmcc, this analysis uses the same modules. Compiler extensions are necessary only to generate the information required to analyze the impact of optimizing transformations on the data-value problems. Moreover, the algorithms work on the final object representation of a program; this is in contrast to other approaches that keep around a copy of the original source program representation (e.g., [106]) or work on an auxiliary intermediate representation (e.g., [36]).

5.5.1 Example

I illustrate with an example the data-value problems caused by global optimizations and how these problems can be managed by the debugger.

Figure 5.7(a) shows the intermediate representation (IR) flow graph of a program fragment before optimizations and code generation. This program fragment contains exactly five expressions $E_1 \ldots E_5$ that assign to a source-level variable x and exactly two uses of x, one after E_4 inside block B7 and one in block B9. There are no assignments to variables y and z in this fragment. Figure 5.7(b) shows the flow graph of the object code after code hoisting and dead code elimination transformations and code generation. The destination register of an instruction is the first operand of the instruction. I assume that other IR expressions and object instructions exist in the program. Variable `x` is assigned a register represented symbolically by `Rx` in Figure 5.7(b), and no other definitions of the register `Rx` exist other that the ones shown in the figure.

In this example, assignment expressions E_1 and E_4 have been translated to instructions I_1 and I_3 respectively. Variable `x` is dead after expressions E_2 and E_3. Dead code elimination has eliminated these assignment expressions, so that no code has been generated for E_2 or E_3 in Figure 5.7(b). Partial redundancy elimination [76] has eliminated expression E_5 and has inserted instruction I_2 into block B4. The expression `x=y+z` is partially available [76] at B8 since this expression is available along paths that reach B8 from B7, but not along paths that reach B8 from B6. In Figure 5.7(b), a copy of the expression `x=y+z` has been inserted into block B4 (instruction I_2). This insertion makes expression `x=y+z` available along all paths that reach B8 and thus E_5 has been eliminated. In effect, this insertion has hoisted E_5 up from B8 to B4.

Endangered variables caused by code hoisting

Code hoisting causes endangered variables by hoisting an expression that assigns to a source-level variable V, thus causing V to be updated prematurely. If the debugger can detect that the value in the runtime location of a variable V is *definitely* from a source-level assignment that has executed prematurely at the break, the debugger reports that V is noncurrent. However, control flow introduces ambiguities and the debugger in general does not know which execution path is traversed to reach a break. Thus, the debugger sometimes detects that the actual value of a variable V is *possibly* from a source-level assignment that has executed prematurely and V is reported as suspect. In the case that a variable V is either noncurrent or suspect due to the (possible) premature execution of a source-level assignment, the debugger conveys the actual value of V in source-level terms.

Consider a stopping instruction I that occurs after I_2 in block B4, B5 or B6 (Figure 5.7(b)). The actual value of `x` at I is the value computed by instruction I_2 which corresponds to the source-level value assigned by E_5. The expected value of `x` at the various breaks

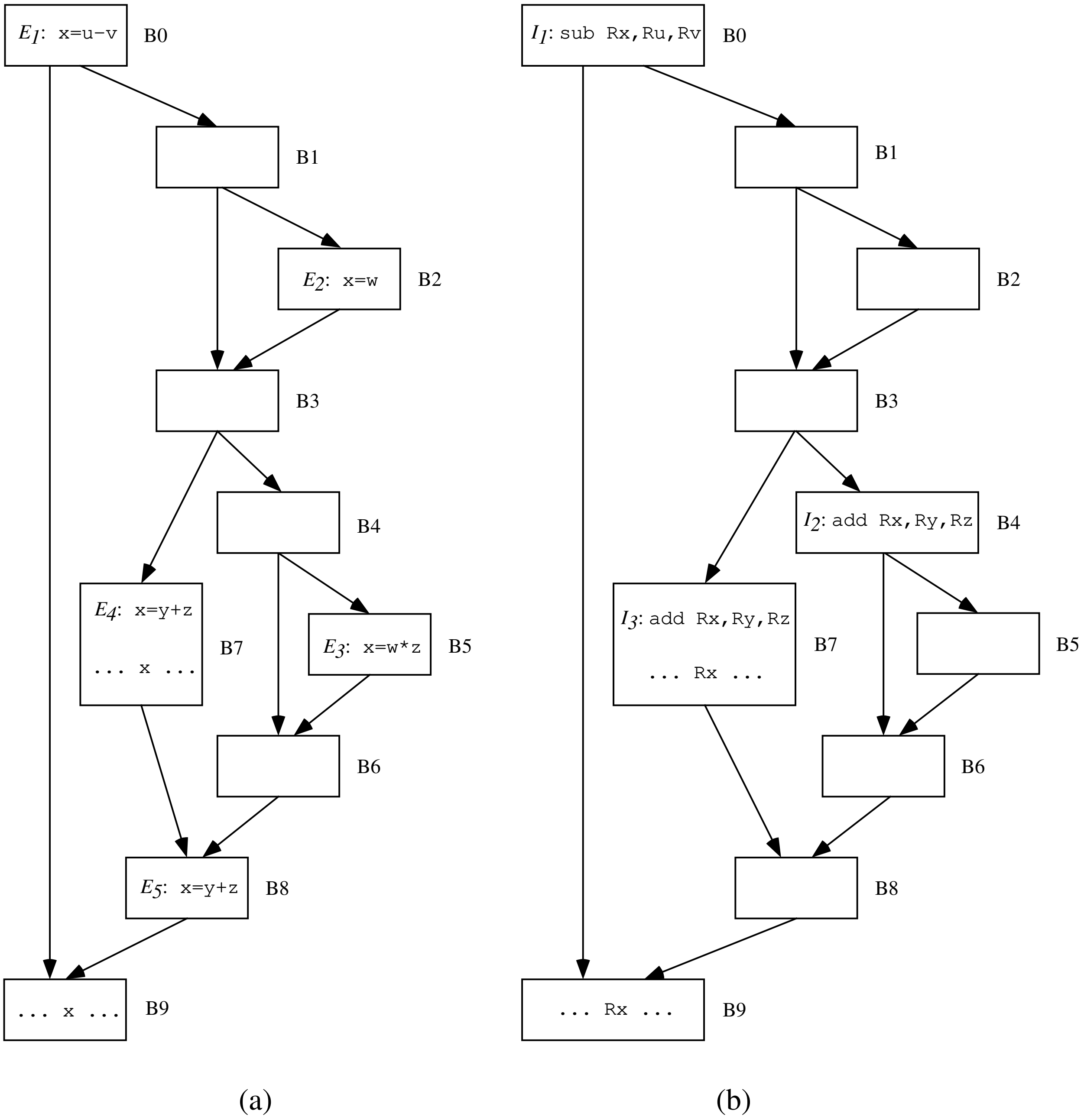


Figure 5.7: Control flow graphs (a) IR after translation from source (b) Object program after optimizations and code generation

within these blocks (Figure 5.7(a)) is the value assigned by E_1, E_2, or E_3, depending on the control reference statement and the execution path leading to the break. Clearly, at all of the above breaks, x is noncurrent, since the actual value of x is different from the expected value of x due to instruction I_2 prematurely assigning to Rx the source-level value assigned by E_5. If the user queries the value of x at any of these breaks, the debugger can display the actual value of x and warn the user that this value is the value assigned by expression E_5, which has executed prematurely.

Now consider a break B inside block B8. Since the expression x=y+z is available at B8, the value in x's runtime location at any stopping instruction inside B8 is the value assigned in the source by E_5. Assume that the control reference statement S associated with break B occurs after E_5 inside B8. At break B, x is current, since the expected value of x is the value assigned by E_5, which corresponds to the actual value of x. Now assume that S is before E_5. If execution reaches B8 from B7, then x's expected value is the value assigned by E_4. The values assigned by E_4 and E_5 are identical and therefore x is current at break B. However, if execution reaches B8 from B6 then the expected value of x is the value assigned by any of the expressions E_1, E_2, or E_3 (depending on the path taken) and thus x is noncurrent at break B. In the absence of knowledge regarding execution history, the debugger cannot determine whether execution reaches B8 via B7 or B6, and therefore the debugger cannot determine whether x is current or noncurrent and must report x as being suspect at break B. If the user queries the value of x at this break, the debugger can display the actual value of x and warn the user that this value *may* be from expression E_5, which *may* have executed prematurely at block B4.[3] The user may be able to determine whether block B4 has executed (e.g., based on the values that determine the outcome of block B3's conditional branch) and thus whether E_5 has indeed executed prematurely. Note that the compiler can instrument the object code or the debugger can install hidden breakpoints to collect runtime information (e.g., path determiners [109]), allowing the debugger to determine which path is taken to B8 and thus report x as either noncurrent or current. This instrumentation is disallowed, however, in our non-invasive debugger model.

[3] Of course block B4 will be translated to some program point in the source. Also, the actual value may stem from several hoisted expressions and the debugger may have to report several such program points to the user. If the user determines that execution reached the break from any of these points, then the user can infer that the suspect variable is noncurrent.

Endangered variables caused by dead code elimination

Dead code elimination causes endangered variables by eliminating dead assignment expressions, thus eliminating updates to variables. In the case of an endangered variable V caused by dead code elimination, the expected value of V is the value that would have been assigned by a source-level assignment E_d that was eliminated by dead code elimination, while the actual value of V is the value assigned by a source-level assignment other than E_d. When both dead code elimination and code hoisting are performed, it is possible that the expected value of a variable V stems from a dead assignment, while the actual value of V stems from an assignment that has been executed prematurely due to code hoisting; in this case V is endangered due to code hoisting. For example, in Figure 5.7, the dead expression E_3 will never directly cause endangerment because I_2 has prematurely updated x at all breaks where E_3 assigns the expected value of x.

Because of control flow ambiguities, dead code elimination can cause both noncurrent and suspect variables. When a variable is endangered due to dead code elimination, the debugger can convey in source-level terms what the actual value of an endangered variable V (possibly) corresponds to, as well as the fact that the assignment(s) that would have (possibly) assigned the expected value of V has (have) been eliminated.

Consider a break B inside block B2. The actual value of x at break B is the value assigned by I_1, which corresponds to the source-level value assigned by E_1. If the control reference statement S associated with break B is before E_2, then the expected value of x is the value assigned by E_1 and x is current. On the other hand, if S is after E_2, then the expected value of x is the value assigned by E_2, and thus x is noncurrent at break B; i.e., the actual value of x is a stale value since x should have been updated by E_2. If the user queries the value of x at break B, the debugger can display the actual value of x and warn the user that this value does not correspond to the expected value of x, which is the value that would have been assigned by the eliminated expression E_2.

Now consider a break B inside block B3. The actual value of x is again the source-level value assigned by E_1. Depending on the path that was taken to this break, the expected value of x is the value assigned by either E_1 or E_2. If control reaches break B without going through B2, then both the expected and actual values of x are the value assigned by E_1, and thus `x` is current at break B. However, had the path through B2 been taken, then `x` would be noncurrent at break B. Since the debugger cannot disambiguate which path was actually traversed to reach break B, it cannot determine whether `x` is current or noncurrent and must report `x` as suspect at break B. If the user queries the value of `x` at break B, the debugger can display the actual value of `x` and warn the user that this value *may* not correspond to the expected value of `x`, which *may* be the value that would have been assigned by the eliminated expression E_2. The user may be able to determine whether E_2 has executed (e.g., based on the values that determine the outcome of block B1's conditional branch). Variable `x` is similarly suspect at a break inside B4, with the stopping instruction before instruction I_2.

In the case where a variable V is either noncurrent or suspect due to dead code elimination, the debugger can perform reaching definitions analysis in the object code, to determine which source definition(s) reach a break and thus which source expression(s) (could have) assigned the actual value of V. This information can then be presented to the user. For example, in Figure 5.7, only the source definition I_1 reaches a break at B3, so in addition to telling the user that `x` is suspect, the debugger can tell the user that the actual value of `x` is the value assigned by E_1.

5.5.2 Terminology

The Morel and Renvoise partial redundancy algorithm [76] and its variants [30, 39, 62] compute the set of program points where the insertion of an expression E will make another partially redundant instance of E fully redundant. I refer to all such expressions that are inserted by code hoisting transformations as *hoisted* expressions; the expressions that are made redundant by the insertions and thus eliminated from the program, are referred to as *redundant* expressions. If a hoisted expression E_h is inserted to make one other expression E_r redundant, then E_r is referred to as the *redundant copy* $RedCopy(E_h)$ of the hoisted expression E_h, and E_h is referred to as a *hoisted copy* of the redundant expression E_r.

Referring back to Figure 5.7(b), since I_1 and I_3 assign a source-level value of `x` to register `Rx`, these instructions are source definitions of `x` [2]. I_2 also assigns a source-level value of `x` to `Rx` and is a source definition of `x`, but this source definition is one that has been generated for a hoisted expression, and so I_2 is referred to as a *hoisted definition* of `x`.

That is, a hoisted definition of a variable V is a source definition of V that was generated for a hoisted assignment expression. I extend $RedCopy$ to apply to hoisted definitions: if D is a hoisted definition generated for a hoisted expression E_h, then $RedCopy(D)$ is the redundant copy of E_h (i.e., $RedCopy(D) = RedCopy(E_h)$). Note that a hoisted definition is an instruction, whereas the redundant copy of a hoisted definition is an expression in the IR. Note also, that the value computed by a hoisted definition D corresponds to the source-level value computed by the redundant expression $RedCopy(D)$. In Figure 5.7(b), $E_5 = RedCopy(I_2)$ and thus the value assigned to `Rx` by the hoisted definition I_2, corresponds to the source-level value assigned by the redundant expression E_5.

5.5.3 Detecting endangered variables caused by code hoisting

In this section, I concentrate on detecting endangerment caused by code hoisting. To aid in the description of my algorithm, I use the simple example of Figure 5.8, which shows the intermediate representation (IR) flow graph of a program fragment. Code hoisting has inserted expression E_3 inside block B2, rendering expression E_2 in block B3 redundant. Thus, E_3 is a hoisted expression, E_2 is a redundant expression, and $E_2 = RedCopy(E_3)$. Three possible breakpoints are depicted: Bkpt1, Bkpt2, and Bkpt3. At Bkpt1, `x` is definitely noncurrent, since the actual value of `x` is different from the expected value of `x` because of E_3 prematurely assigning to `x` the source-level value assigned by E_2. At breakpoint Bkpt2, `x` is current if execution reaches block B3 from block B1, and noncurrent if execution reaches from block B2. Since the debugger cannot determine how execution reached B3, it qualifies `x` as *suspect* at breakpoint Bkpt2. Finally, at breakpoint Bkpt3, the expected value of `x` is the value assigned by E_2 while the actual value of `x` is the value assigned by either E_1 or E_3, depending on the path traversed to this breakpoint. The values assigned by either E_1 or E_3 are the same as the value that would have been assigned by E_2 (otherwise E_2 would not have been eliminated due to redundancy); thus, `x` is now current.

The key idea behind my algorithm is to determine if there exists a path to a break that includes a hoisted expression that is not followed (on the same path) by a redundant copy. Once execution has progressed to the point that all paths include a redundant copy, the variable is current, since the actual value of the variable is the expected value. Path problems are easily solved by an appropriate data flow framework.

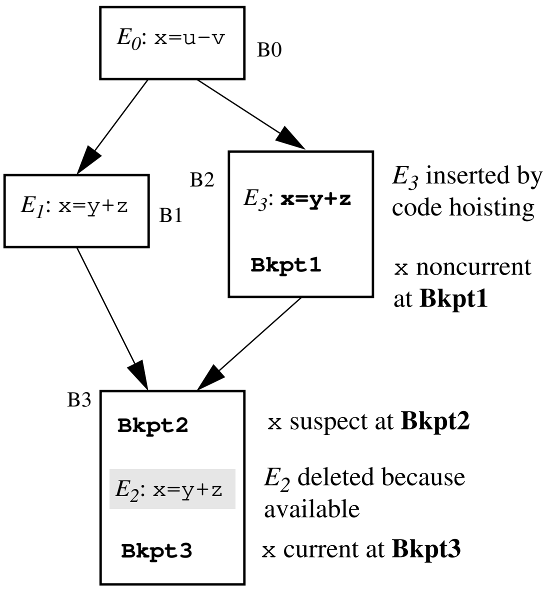


Figure 5.8: Example: Code hoisting.

The intuition behind my approach to detecting endangered variables caused by code hoisting is as follows. Let E_h be a hoisted definition of a variable V, let E_r be the redundant copy of E_h and let P be an execution path in the object code, that passes once through both E_h and $Block(E_r)$. Consider a point O along this path where a break B has occurred. Assume that O occurs after E_h and that along P, there are no other instructions that define the register assigned to V, so that the actual value of V at B is the source-level value assigned by E_r. We can make the following observations:

1. If, along path P, O is *before* the beginning of $Block(E_r)$ then E_r has executed prematurely and V is noncurrent at break B.

2. If O is *after* the end of $Block(E_r)$ along path P, then the expected value of V at break B (assuming no other assignments to V exist in the IR) is the value assigned by E_r; thus, V becomes current after $Block(E_r)$. In other words, once execution has passed through $Block(E_r)$, E_r has no longer been executed prematurely by E_h, so E_h no longer causes V to be noncurrent.

3. If O is *inside* $Block(E_r)$ then depending on whether the control reference statement is before or after E_r, V is either noncurrent or current respectively at break B. When O is within $Block(E_r)$, detecting whether the control reference statement is before or

after E_r requires only a local analysis, similar to the analysis I presented in Section 5.2 for local instruction scheduling.

To illustrate using the example of Figure 5.8, consider the path that extends from E_3 to Bkpt3 and beyond. Bkpt1 corresponds to case 1 above, and a point after Bkpt3 corresponds to case 2 above. Breaks within block B3 correspond to case 3: at Bkpt2 `x` is noncurrent while at Bkpt3 `x` is current.

My approach to detecting endangered variables uses two analyses:

1. **Global analysis.** Data-flow analysis is used to detect endangerment caused by E_h at breaks in blocks other than $Block(E_r)$ (cases 1 and 2 above). This data-flow analysis determines whether possible execution paths leading to a break could have executed E_h without subsequently passing through $Block(E_r)$. Depending on whether this analysis finds all, only some, or no paths along which this property holds, V is either noncurrent due to hoisted definitions, suspect due to hoisted definitions, or unaffected by hoisted definitions.

2. **Local analysis.** The local analysis of Section 5.2 is used to detect whether V is endangered at breaks inside $Block(E_r)$. Since E_r is redundant, it has been eliminated from the program and no instructions have been generated for it. However, at any break inside $Block(E_r)$, the runtime location of V contains the source-level value that would have been assigned by E_r, since the value of E_r is available upon entry to $Block(E_r)$ (assuming no other assignments to V's runtime location). Thus, E_r can be modelled as executing at the pre-amble instruction of $Block(E_r)$ — that is, regardless of where the stopping instruction is in $Block(E_r)$, E_r can be considered as having executed since the value in the runtime location of V is the value that would have been assigned by E_r. For example, in Figure 5.8, since the expression `x=y+z` is available upon entry to block B3, E_2 can be modelled as executing at the pre-amble instruction of block B3, even though no instructions have been generated for E_2. By modelling the execution of E_2 in this manner, the algorithm described in Section 5.2 will detect E_2 as having executed prematurely only at control reference statements before E_2.

   For each assignment expression E_r eliminated by code hoisting, I create an assignment descriptor $\langle I, N, V\rangle$ where $I = Preamble(Block(E_r))$, $N = Seq(E_r)$ and V is the variable assigned by E_r. For any stopping instruction O inside $Block(E_r)$, we have $Preamble(Block(E_r)) < O$; therefore, by Equation 5.2, V is endangered at all control references statements S where $Seq(S) < Seq(E_r)$.

The data-flow analysis used to detect endangered variables due to code hoisting is similar to the *reaching definitions* analysis [7]. In the remainder of this section, I describe this data-flow analysis solution.

After the execution of a hoisted expression E_h that assigns to a variable V, the *actual* value of V corresponds to the source-level value that would have been assigned by the redundant expression $RedCopy(E_h)$. For example, in Figure 5.8, the actual value of x at breakpoint Bkpt1 is the value assigned by the hoisted assignment E_3, which is the source-level value assigned by E_2.

Let $P = \langle start, ..., O\rangle$ be the execution path in the object code, that is traversed to a breakpoint B. If at B the actual value of a variable V is the value assigned by a hoisted expression E_h, then V is noncurrent at B if the *expected* value of V does not correspond to the source-level value that would have been assigned by $RedCopy(E_h)$. V is definitively noncurrent if E_h reaches along P and after the last occurrence of E_h, P does not include $RedCopy(E_h)$. Or expressed positively, V is current whenever a path P includes $RedCopy(E_h)$, and E_h does not occur on P after $RedCopy(E_h)$. For example, in Figure 5.8, E_3 reaches breakpoints Bkpt1 and Bkpt3. x is noncurrent at Bkpt1 since any path taken to this breakpoint does not go through E_2. x is current at Bkpt3, however, because all paths to Bkpt3 must go through E_2.

Therefore, given paths along which a hoisted expression E_h reaches, those paths that do not go through $RedCopy(E_h)$ are distinguished:

Definition 4 *A redundant expression E_r* **hoist reaches along a path** $P = \langle start, ..., O\rangle$ *in the object code, if there exists a hoisted expression E_h such that $E_r = RedCopy(E_h)$, E_h reaches along P, and E_r does not occur after the last occurrence of E_h along P.*

Note that hoist reach is a property of redundant expressions only — that is, expressions that have been eliminated due to partial redundancy elimination. In Figure 5.8, the redundant expression E_2 hoist reaches Bkpt2 on the path from block B2. E_2 does not hoist reach on paths that reach via block B1.

Lemma 3 *Let E_r be a redundant assignment expression that assigns to a variable V. If E_r hoist reaches along a path $P = \langle start, ..., O\rangle$ and P is the execution path traversed to a breakpoint B, then V is noncurrent at B due to the premature execution of E_r.*

Proof: After execution of P, the actual value of V corresponds to the source-level value assigned by E_r. However, since execution has not yet reached $Block(E_r)$, the expected value of V cannot correspond to the value that would have been assigned by E_r. Hence V is noncurrent at any break at O due to the premature execution of E_r.

Because the debugger does not know which execution path was actually taken to reach a breakpoint, all possible paths must be considered. The following lemmas describe the two cases where a redundant assignment expression hoist reaches along all or only some of the paths that lead to a point O, where a breakpoint has occurred. Let E_r be a redundant assignment expression that assigns to a variable V:

Lemma 4 *If E_r hoist reaches along all paths leading to a point O, then at any breakpoint occurring at O, V is noncurrent due to the premature execution of E_r.*

Proof: Lemma 3 holds for all possible execution paths leading to O. Thus, V is noncurrent due to the premature execution of E_r at any break B occurring at O regardless of the execution path traversed to reach break B.

Lemma 5 *If E_r hoist reaches along at least one but not all paths leading to a point O, then at any breakpoint occurring at O, V is suspect due to the possible premature execution of E_r.*

Proof: Lemma 3 holds for all least one but not all execution paths leading to O. Therefore, the debugger cannot determine statically whether V is current or noncurrent at any break B occurring at O and V is suspect at breaks occurring at O.

In Figure 5.8, `x` is noncurrent at Bkpt1, because the redundant assignment expression E_2 hoist reaches on all paths to Bkpt1. At Bkpt2, `x` is suspect because E_2 hoist reaches on only some paths. At Bkpt3, `x` is current because no expressions that assign to `x` hoist reach.

Hoisted indirect assignments

An indirect assignment assigns to a memory location specified by a pointer expression (e.g., `*p = ...`). If an indirect assignment A hoist reaches a point O, then the variable pointed to by the pointer expression in the left hand side of A is noncurrent at O. If the value of the pointer used by A is available at O, then the debugger can tell precisely which variable has been prematurely updated by A. In the absence of such knowledge, however, the debugger must conservatively assume that all variables that are potentially aliased by A are suspect.

I can easily extend the above formulation to take into account hoisted indirect assignments:

Lemma 6 *Let E_r be a redundant expression that is an indirect assignment and let V be a variable that is possibly aliased by E_r. If E_r hoist reaches along a path $P = \langle start, ..., O\rangle$ and P is the execution path traversed to a breakpoint B, then V is suspect at B due to the premature execution of E_r.*

Indirect assignments cause variables to be suspect if they hoist reach on *any* path. Let E_r be a redundant expression that is an indirect assignment (i.e., an assignment through a pointer) and let V be a variable that is possibly aliased by E_r:

Lemma 7 *If E_r hoist reaches along any path leading to a point O, then at any breakpoint occurring at O, V is suspect due to the possible premature execution of E_r.*

Optimizing expressions involving pointer accesses requires expensive alias analysis and is performed by few optimizing compilers. Rather than perform alias analysis some compilers optimize pointer accesses but with very pessimistic assumptions, e.g., `cmcc` assumes that all local variables whose addresses are taken are aliased along with all global variables.

When a hoisted indirect assignment causes suspect variables, the debugger can try to recover the runtime value of the pointer expression, thus detecting which variable has been prematurely updated. However, our measurements using the `cmcc` compiler show that hoisting of indirect assignments occur very rarely (I have not yet seen a case). In practice, it is highly unlikely that a user will encounter endangerment due to hoisted indirect assignments and thus I do not bother with formulating how the runtime value of a pointer can be recovered.

The hoist reaching formulation described in this section computes the set of assignment expressions that hoist reach; the sets of noncurrent and suspect variables are derived from the set of hoist reaching assignments. By computing the set of hoist reaching assignment expressions, the algorithms can detect whether any of the hoist reaching expressions are indirect assignments and thus whether any variables are suspect due to indirect assignments. If we assume that indirect assignments are never hoisted, then the formulation of the problem can be changed to directly compute the set of variables that are noncurrent and suspect due to code hoisting. In Section 5.5.4, I illustrate this alternative by formulating the algorithm for computing endangered variables caused by dead code elimination, assuming indirect assignments are not eliminated.

Data-flow analysis

Detecting whether a redundant expression hoist reaches along all or only some paths can easily be done using data-flow analysis. This data-flow analysis is performed on the final instruction-level intermediate representation of a program, which includes information describing the effects of optimizations. In Section 5.7 I describe how this representation is built and maintained by the `cmcc` compiler. The hoist reach data flow attribute of an expression E is generated by any code inserted by code hoisting that computes E and killed by any eliminated redundant code that also computes E. Two data-flow analysis problems, $AllHoistReach$ and $AnyHoistReach$, that together represent the conditions of Lemma 4 and Lemma 5 are defined:

Definition 5 *The predicate $AllHoistReach(E_r, O)$ is true at a point O in the object code if the redundant assignment expression E_r hoist reaches along all paths leading to O.*

Definition 6 *The predicate $AnyHoistReach(E_r, O)$ is true at a point O in the object code if the redundant assignment expression E_r hoist reaches along any path leading to O.*

Note that the data-flow analysis problems involve properties of redundant assignment expressions in the intermediate representation, yet they are performed on the object code.

Note also, that the algorithm does not need to determine which instance of an expression hoist reaches, but rather that *some* expression hoist reaches; that is, the compiler need not determine that $E_2 = RedCopy(E_3)$, but rather that E_3 is a hoisted instance of the syntactic expression `x=y+z`, and that E_2 is redundant. If a break B occurs at a point O in the object code where $AllHoistReach(E_r, O)$ is true, then the variable assigned by E_r is noncurrent at break B due to the premature execution of E_r. Otherwise, if $AnyHoistReach(E_r, O)$ is true, then the variable assigned by E_r is suspect at break B due to the possible premature execution of E_r.

Indirect assignments cause variables to be suspect if they hoist reach on any path. Thus if E_r is an indirect assignment and $AnyHoistReach(E_r, O)$ is true, then memory variables are suspect.

The In and Out sets of the data-flow analysis problem, with respect to an instruction I, are as follows:

- $AllHoistReachIn(I)$ is the set $\{E_r : AllHoistReach(E_r, pre(I))\}$
- $AllHoistReachOut(I)$ is the set $\{E_r : AllHoistReach(E_r, post(I))\}$
- $AnyHoistReachIn(I)$ is the set $\{E_r : AnyHoistReach(E_r, pre(I))\}$
- $AnyHoistReachOut(I)$ is the set $\{E_r : AnyHoistReach(E_r, post(I))\}$

The only difference between $AllHoistReach$ and $AnyHoistReach$ is the confluence operator; the $Kill$ and Gen sets of the two analyses are the same and are represented by $HoistReachKill$ and $HoistReachGen$. The confluence operator for $AllHoistReach$ is set intersection, while that of $AnyHoistReach$ is set union:

$$AllHoistReachIn(I) = \bigcap_{J \in pred(I)} AllHoistReachOut(J)$$

$$AnyHoistReachIn(I) = \bigcup_{J \in pred(I)} AnyHoistReachOut(J).$$

The set $HoistReachGen$ is defined as follows. Let D be a hoisted definition, and let E_r be a redundant assignment expression such that $E_r = RedCopy(D)$. E_r hoist reaches the point $post(D)$ immediately after D, thus D generates $AllHoistReach(E_r, post(D))$ and $AnyHoistReach(E_r, post(D))$:

- If D is a hoisted definition, then $HoistReachGen(D) = \{RedCopy(D)\}$, otherwise $HoistReachGen(D) = \{\}$.

The set $HoistReachKill$ is the smallest set defined as follows. Because a redundant expression E_r is no longer hoist reaching along a path that passes through $Block(E_r)$, the pre-amble of $Block(E_r)$ kills $AllHoistReach$ and $AnyHoistReach$ for a redundant expression E_r:

- If E_r is a redundant assignment expression,
  then $E_r \in HoistReachKill(Preamble(Block(E_r)))$.

Given a redundant expression E_r that is the redundant copy of a hoisted expression E_h, E_r post-dominates E_h and the hoist reach of E_r is eventually killed on any path leading from E_h. Therefore, the region of endangerment caused by code hoisting is limited.

Any instruction that defines a register R kills $AllHoistReach$ and $AnyHoistReach$ for all E_r such that E_r is a redundant assignment expression that assigns to a variable V that has been assigned register R:

- Let I is an instruction that defines a register R. $\forall E_r$ such that

  1. E_r is a redundant assignment expression that assigns to a variable V and
  2. the variable V has been assigned register R

  $E_r \in HoistReachKill(I)$.

The In and Out sets of the data-flow equations for an instruction I are related by:

$$AllHoistReachOut(I) = (AllHoistReachIn(I) \backslash HoistReachKill(I)) \cup HoistReachGen(I)$$

$$AnyHoistReachOut(I) = (AnyHoistReachIn(I) \backslash HoistReachKill(I)) \cup HoistReachGen(I).$$

Providing additional information to the user

As described in Section 5.5.1, if a variable is suspect due to the possible premature execution of a redundant assignment expression E_r, then the debugger can provide additional information to the user by displaying in source-level terms the execution points at which hoisted copies of E_r execute. For instance, in Figure 5.8, if the user queries the value of x at breakpoint Bkpt1, the debugger can display the actual value of x and warn the user that this value is the value assigned by the source-level assignment E_2, which has executed prematurely. If the user queries the value of x at breakpoint Bkpt2, the debugger can display the actual value of x and warn the user that this value *may* be from expression E_2, which *may* have executed prematurely at block B2.

This information can be gathered at a point O in the object code using a reaching definitions data-flow analysis in the object code to compute the set

$$ReachingHoistAssigns(E_r, O) = \{D : DreachesO \wedge E_r = RedCopy(D)\}.$$

If a break occurs at O, the debugger informs the user of source execution points where hoisted copies of E_r execute, after mapping each instruction $I \in ReachingHoistAssigns(E_r, O)$ to an execution point in the source.

5.5.4 Detecting endangered variables caused by dead code elimination

This section focuses on detecting endangered variables caused by dead code elimination. The program fragment of Figure 5.9 is used to demonstrate the effects of dead code elimination on debugging. In this figure, assignment sinking has inserted E_2 and expression E_0 has been deleted because sinking makes E_0 dead. At breakpoint Bkpt1, x's expected value is the value that would have been assigned by E_0, while x's actual value is the value assigned by the last assignment that was executed prior to this program fragment. Therefore, the actual value of x is a stale value, and x is noncurrent. x is similarly noncurrent at Bkpt2 and Bkpt3. At Bkpt4, x is current since expression E_2 assigns the expected value of x (i.e., the value that would have been assigned by E_0). At Bkpt5, x is noncurrent if execution has reached this breakpoint from block B1 and current if execution has reached from block B2. Therefore, x is suspect at Bkpt5. Finally at Bkpt6, both the expected and actual values of x are from E_1, and thus, x is current.

The approach to detecting endangered variables caused by dead code elimination uses a data-flow analysis similar to that described in Section 5.5.3. The data-flow analysis for detecting endangered variables at a point O in the object code solves for the predicates

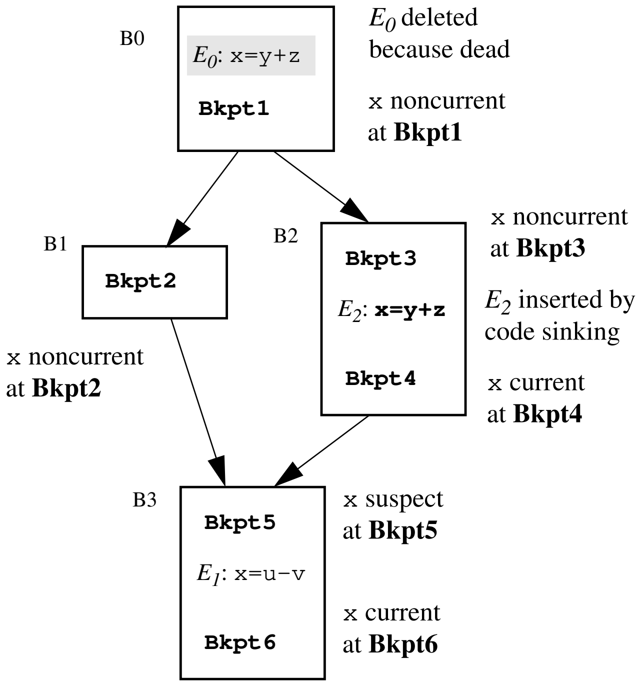


Figure 5.9: Example: Code sinking and dead code elimination.

$AllDeadReach(V, O)$ and $AnyDeadReach(V, O)$ for a variable V. $AllDeadReach(V, O)$ is true if along all paths leading to O the expected value of V is assigned by a dead assignment and the actual value of V is stale. $AnyDeadReach(V, O)$ is true if the conditions holds along only some paths. Thus, at a break occurring at O, V is noncurrent due to dead code elimination if $AllDeadReach(V, O)$ is true; otherwise, V is suspect if $AnyDeadReach(V, O)$ is true. If neither predicate is true then V is not endangered due to dead code elimination.

Let E_d be a dead assignment expression that assigns to a variable V. Let D be a source definition of V and P be an execution path in the object code, such that D and $Block(E_d)$ occur once in P, D occurs after $Block(E_d)$, and $Block(D) \neq Block(E_d)$. Assume that between $Block(E_d)$ and D along P there are no other blocks containing dead assignments to V and no other source definitions of V. Consider a point O along P, where a break B has occurred. We can make the following observations:

1. If O is *inside* $Block(E_d)$ then depending on whether the control reference statement is before or after E_d, V is either current or noncurrent respectively at break B.

2. If, along path P, O is *after* $Block(E_d)$ but *before* D then at break B, the expected value

of V is the value that would have been assigned by E_d which does not correspond to the actual value of V. V is noncurrent at break B due to the elimination of the dead assignment expression E_d.

3. If O is *after* D along path P, then the actual value of V is the value assigned by D. At break B, this value may either be the current value of V, or the value from a prematurely executed assignment to V. In any case, once execution has passed D, the elimination of E_d no longer causes V to be noncurrent.

My approach to detecting endangered variables caused by dead code elimination again divides the problem into a global and local component:

1. **Global analysis.** Data-flow analysis is used to detect endangerment caused by dead code elimination of E_d at breaks in blocks other than $Block(E_d)$ (cases 2 and 3 above). This data-flow analysis determines whether possible execution paths leading to a break could have passed through $Block(E_d)$ for some dead assignment expression E_d that assigns to a variable V, without subsequently executing a source definition D of V. Depending on whether this analysis finds all, only some, or no paths along which this property holds, V is either noncurrent because dead code elimination, suspect because of dead code elimination, or unaffected by dead code elimination.

2. **Local analysis.** The local analysis of Sections 5.2 and 5.3 is used to detect whether V is endangered at breaks inside $Block(E_d)$ for some dead assignment expression E_d that assigns to variable V. Because E_d is dead, it has been eliminated from the program and no instructions have been generated for it. However, at any break inside $Block(E_d)$, the runtime location of V will never contain the source-level value that would have been assigned by E_d. Thus, E_d can be modelled as executing at the postamble instruction of $Block(E_d)$; that is, regardless of where the stopping instruction is in $Block(E_d)$, the execution of E_d has been delayed until the end of $Block(E_d)$. For example, in Figure 5.7, by modelling the execution of E_2 as occurring at the end of block B2, the algorithm described in Section 5.2 (with the extension described in Section 5.3) will detect E_2 as having not executed when it should have only at control reference statements within block B2 that occur after E_2.

For each assignment expression E_d eliminated by global dead code elimination, the compiler creates a dead assignment descriptor $\langle I, N, V\rangle$ where $I = Postamble(Block(E_d))$, $N = Seq(E_d)$ and V is the variable assigned by E_d. For any stopping instruction O inside $Block(E_d)$, we have $Postamble(Block(E_d)) > O$; thus, by Equation 5.3, V is endangered at all control references statements S where $Seq(S) > Seq(E_d)$.

Unlike the hoist reaching data-flow algorithm where we solved for whether a *redundant IR expression* is hoist reaching, the data flow algorithm for detecting endangered variables caused by dead code elimination solves for whether a *variable* V is endangered due to the elimination of some dead assignment to V. After execution passes through a dead assignment to a variable V, the actual value of V becomes stale until another assignment to V is executed. For example, in Figure 5.9, x becomes noncurrent after E_0, until after the assignments E_1 and E_2. Therefore, I distinguish those paths where a variable's value is stale due to a dead assignment:

Definition 7 *A variable V is* **dead reaching along a path** $P = \langle start, ..., O\rangle$ *in the object code, if there exists a dead assignment expression E_d that assigns to V such that $Block(E_d)$ occurs in P and no assignments to V occur along P after the last occurrence of $Block(E_d)$.*

If a variable V is dead reaching along a path P and P is the execution path traversed to a breakpoint B, then V is clearly noncurrent at B:

Lemma 8 *If a variable V is dead reaching along a path $P = \langle start, ..., O\rangle$, and P is the execution path traversed to a breakpoint B, and V is not noncurrent due to the premature execution of a redundant assignment, then V is noncurrent at B because the actual value of V is stale.*

Proof: After execution of P, the expected value of V is from some dead assignment expression E_d. The actual value of V is from an assignment that occurs earlier than E_d in the source. Hence the expected and actual values of V do not correspond and V is noncurrent at break B.

The development of the data-flow problem is similar to that of Section 5.5.3 (proofs of lemmas are very similar to those of Lemmas 4 and 5, and are omitted for the sake of conciseness):

Lemma 9 *If a variable V is dead reaching along all paths leading to a point O, then V is noncurrent at any breakpoint occurring at O.*

Lemma 10 *If a variable V is dead reaching along at least one but not all paths leading to a point O, then V is suspect at any breakpoint occurring at O.*

In Figure 5.9, x is dead reaching along all paths leading to Bkpt1, Bkpt2 and Bkpt3; thus, x is noncurrent at these breakpoints. At Bkpt5, x is dead reaching only along those paths that pass through B1; thus, x is suspect. At Bkpt6, x is not dead reaching; thus, x is current.

Data-flow analysis

The data-flow analyses to detect whether a variable is dead reaching on only some or all paths can be derived in a straight-forward manner from the above definitions and lemmas. The dead reach of a variable V is generated by a dead assignment to V and killed by any other kind of assignment to V.

Definition 8 *The predicate $AllDeadReach(V, O)$ is true at a point O in the object code if the variable V is dead reaching along all paths leading to O.*

Definition 9 *The predicate $AnyDeadReach(V, O)$ is true at a point O in the object code if the variable V is dead reaching along any path leading to O.*

The definitions of the sets $AllDeadReachIn$, $AllDeadReachOut$, $AnyDeadReachIn$ and $AnyDeadReachOut$, as well as the confluence operators and flow equations for these sets are similar to the corresponding sets of the data flow analysis in Section 5.5.3; I omit the details for conciseness. The set $DeadReachGen$ is the smallest set defined by:

- If E_d is a dead assignment expression that assigns to a variable V, then $V \in DeadReachGen(Postamble(Block(E_d)))$.

The set $DeadReachKill$ is the smallest set defined by:

- If I is a source definition of V, then $V \in DeadReachKill(I)$.

Providing additional information to the user

If a variable V is endangered due to dead code elimination, then it is useful to inform the user which source statements assign the actual and expected values of V (as discussed in Section 5.5.1). At a break, the debugger performs a reaching definitions analysis *in the object code* to detect the set of reaching instructions that are source definitions. Using this information, the debugger can inform the user which source-level expression assigns (or which expressions may assign) the *actual* value of V. To inform the user of which expressions assign the *expected* value of V, the debugger performs reaching definitions analysis *in the source*.

5.5.5 Recovery

If dead code elimination eliminates an assignment to a variable V, it may be possible to recover the expected value of V from the values of compiler temporaries. Consider the example in Figure 5.10(a). The right hand side of the expression `x=y+z` at statement S_1 is propagated to the two uses of `x` at statements S_2 and S_3. After this assignment propagation, no uses of `x` remain, and thus dead code elimination eliminates S_1 (Figure 5.10(b)). Common subexpression elimination detects the common subexpression `y+z`, replacing the two computations of `y+z` with fetches from the temporary `tmp` (Figure 5.10(c)).

S_1: `x = y+z`		`tmp = y+z`
S_2: `...x...`	S_2: `...y+z...`	S_2: `...tmp...`
S_3: `...x...`	S_3: `...y+z...`	S_3: `...tmp...`
(a)	(b)	(c)

Figure 5.10: Recovery example (a) Original source program (b) After copy propagation and dead code elimination (c) After common subexpression elimination.

In `cmcc`, this assignment propagation is performed to improve partial redundancy elimination [30, 18], and the situation described above occurs quite often. The final effect of this series of transformations is that the source-level variable `x` is replaced with `tmp`. If the user queries the value of `x` at a breakpoint that occurs after statement S_1, the debugger could display the value of `tmp`, since these two variables are aliased. This is one form of *recovery*, where the debugger reconstructs the expected value of a variable from other runtime values.

Recovery is performed by checking each expression E that was inserted by code replacement transformations (Section 5.7 describes how the compiler can keep track of such transformations). If E replaced a fetch from a source-level variable V in the original program, then the value computed by E aliases V, and V can be recovered from the storage location holding the value of E. E may be a constant, a fetch from a temporary, or some more general computation such as addition. In the case that E is a fetch from a temporary T, the compiler generates the residence [2] of V in the storage location assigned to T. If E is a constant, the compiler generates a special constant residence for V, indicating that the value of V is a constant. If E is neither a constant nor a fetch, then the compiler generates the residence of V in the storage location assigned to the result register of the instruction that computes E's value. In all cases, the dead reach of V is killed by E. A similar approach is used to recover the value of a source-level induction variable that is replaced by a compiler-synthesized induction variable.

5.6 Speculative code motion

Speculative code motion hoists an operation O from a block B_1 to another block B_2 that is not post-dominated by B_1; as a result O is introduced into an execution path in which it did not exist before. To maintain correct semantics, the operation O must not fault or incorrectly overwrite values that are read on paths where O did not exist before. Speculative code motion is most commonly performed by global instruction scheduling and loop-invariant code motion algorithms that hoist operations that are not executed on all possible paths through a loop. `cmcc` (like many other compilers) does not perform speculative code motion. The algorithms described thus far for detecting endangered variables do not work with speculative code hoisting. But speculative code motion can be easily handled with simple extensions to the earlier algorithms; I include these extensions here for completeness. I do not deal with the case where an indirect store is speculatively hoisted; indirect stores may cause a fault and are usually never executed speculatively unless special hardware support is provided to buffer and squash the stored value [89]. In general, stores are not hoisted by instruction scheduling since they are at the end of a dependence chain; memory resources are better utilized by hoisting load instructions.

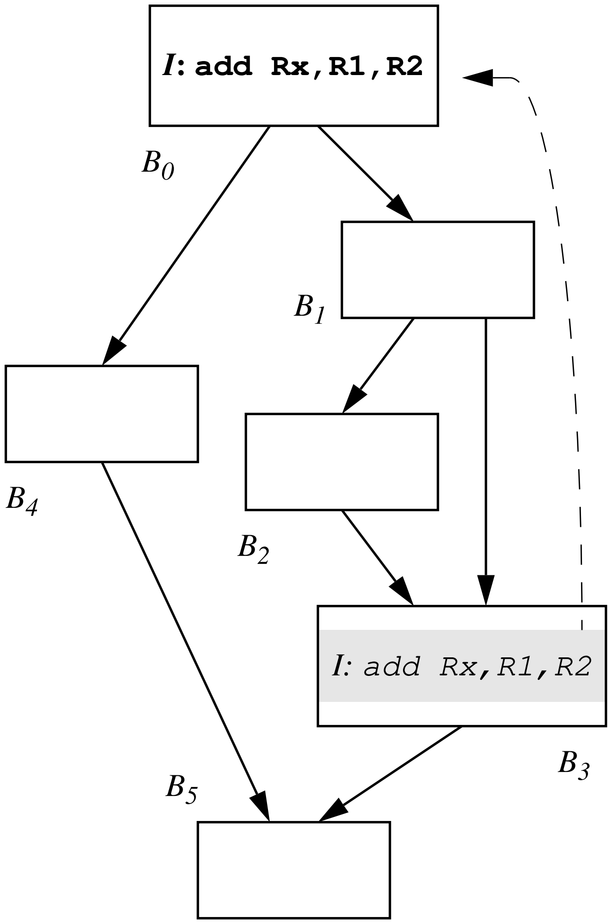


Figure 5.11: Example: Speculative code motion.

Figure 5.11 shows an example of speculative code motion; the compiler has speculatively hoisted instruction I from block B_3 to block B_0. Instruction I is a source definition of variable `x` and there are no other definitions of `x`. By hoisting I to block B_0, the compiler has introduced I into the path $\langle B_0, B_4, B_5\rangle$ where I did not exist before.

Because I did not exist in the path $\langle B_0, B_4, B_5\rangle$, I cannot really be considered a source definition of `x`. At block B_4, the value in `Rx` is not a valid source-level value of `x`; that is, the value in `Rx` is not a value of `x` that is computed on any valid execution path in the source. Therefore, with respect to block B_4 `x` is nonresident. At blocks B_1 and B_2, however, the value in `Rx` is from an assignment to `x` that has executed prematurely. At these blocks, the assignment to `x` at block B_3 is inevitable. At block B_0, however, it is unknown whether B_3

will execute and thus the value in Rx is a *speculative* value of x; since the debugger does not know whether B_3 will execute x is reported as nonresident.

In general, when speculative code hoisting hoists a source definition I of a variable V from a block B_i to a block B_j, V is nonresident on any path leading from B_j to any block B_k that is post-dominated by B_i. At B_k, V becomes a noncurrent variable. Thus I is marked as an evicting definition of V and the first block B_k that is post-dominated by B_j and is reachable on any path leading from B_i is marked as causing V to become resident. Moreover, B_k generates the hoist reach of instruction I while block B_i kills the hoist reach. This ensures that on all paths from B_k to B_i V is correctly detected as endangered.

5.7 Tracking the effects of compiler transformations

To allow the debugger analyses described in Section 5.5, the compiler must perform bookkeeping to record the effects of optimizations in the program representation. In the cmcc compiler, this bookkeeping is performed by annotating the nodes of cmcc's IR. These annotations record whether an operation was inserted by optimizations (and if so by which). Bookkeeping also inserts special *IR marker nodes* to mark points of interest to the debugger. These annotations and markers are ignored by optimizations and optimizations are not constrained in any way.

This method of tracking the correspondence between the optimized code and the source code is in contrast to the approach taken by Wismeuller [106] and Copperman [36]. In those approaches, a representation of the original source program is kept as a copy, and links are maintained between the intermediate representation used for optimizations and the original representation (e.g., an abstract syntax tree). Copperman maintains a data structure called a *DS-graph*, which is essentially a control flow graph containing assignments in the source code with links to instructions in the control flow graph of the final object code. The DS-graph is maintained in parallel to the IR used by the compiler and edited as optimizations transform the IR. The DS-graph is passed to the debugger for data-flow analysis. Wismueller's algorithm for detecting current variables requires two representations, a control flow graph of the source program and a control flow graph of the object program with mappings between the two.

The different ways in which global optimizations may transform a program, and the manner in which bookkeeping is performed for these transformations in `cmcc`, are as follows:

Code insertion Code motion and common subexpression elimination transformations insert new code into the program representation. Expressions that are inserted by either code hoisting or code sinking are marked as hoisted or sunk expressions. Assignment expressions that are marked as hoisted will generate the hoist reach of variables.

Code replacement Copy propagation and redundancy elimination replace one expression with another. Copy propagation replaces a reference to a variable with a propagated expression, while redundancy elimination replaces an available expression with a fetch from a common subexpression temporary. When an expression E replaces a fetch from a variable V, a reference to V is kept in E. This information is needed only for the recovery algorithm described in Section 5.5.5 and can otherwise be omitted.

Code deletion Dead code elimination and partial redundancy elimination delete assignment expressions that are dead or available. When an assignment to a variable V is eliminated, it is replaced with a special IR *marker* node, unless this assignment has been previously marked as sunk or hoisted. A marker node contains a reference to the variable V and an indication why the assignment to V was eliminated (i.e., whether the assignment was dead or available). Markers are ignored by optimization phases and are used only for the debugger analysis algorithms. Markers that indicate an available variable V will kill the hoist reach of V, while markers indicating a dead variable V will generate the dead reach of V.

Code duplication Control flow optimizations such as loop peeling duplicate code. Code duplication, however, does not create data-value problems since no movement or elimination of assignments occur. Therefore, the effects of this transformation need not be recorded. However, marker nodes inside a block B must also be duplicated when B is duplicated. Moreover, if an IR node containing debugging annotations is duplicated, the information must be duplicated along with the node.

Basic block deletion A block of code can be eliminated if the optimizer determines that this code is unreachable. This transformation may occur after a conditional branch expression is folded. Since the code that is eliminated would not have executed in the original program, this transformation does not cause data-value problems and thus the effects need not be recorded.

Basic blocks may also be deleted because they become empty after other optimizations, or because they contain only unconditional branches (and are deleted by branch chaining). In this case, if the deleted basic block contains any information relevant to debugging (i.e., markers), then such information must be retained, and is transferred to the deleted block's successor.

Basic block insertion Edge splitting and preheader insertion insert new (empty) basic blocks into the program representation. This transformation does not cause data-value problems.

Only after the final object code has been produced are all optimizations exposed; thus, the analyses for detecting endangered variables must be performed on an instruction-level representation of the program [2, 1]. Like most compilers, however, `cmcc` has a two level intermediate representation consisting of a machine-independent IR used for global optimizations (e.g., partial redundancy elimination), and an instruction-level representation used for machine dependent optimizations (e.g., instruction scheduling and register allocation). Most of the bookkeeping is performed on the machine-independent IR (since most optimizations operate on this IR), and thus the annotations and markers must be passed along to the instruction-level representation as the program representation is lowered. This is similar to passing high level information such as aliasing information along to a compiler back end for use by an instruction scheduler. During code selection, annotations are transferred from nodes in the machine independent IR, to the selected instructions. IR marker nodes are lowered to special *marker instructions*, which convey essentially the same information as the IR marker nodes. Instructions are also annotated with information indicating which instructions correspond to source-level assignments.

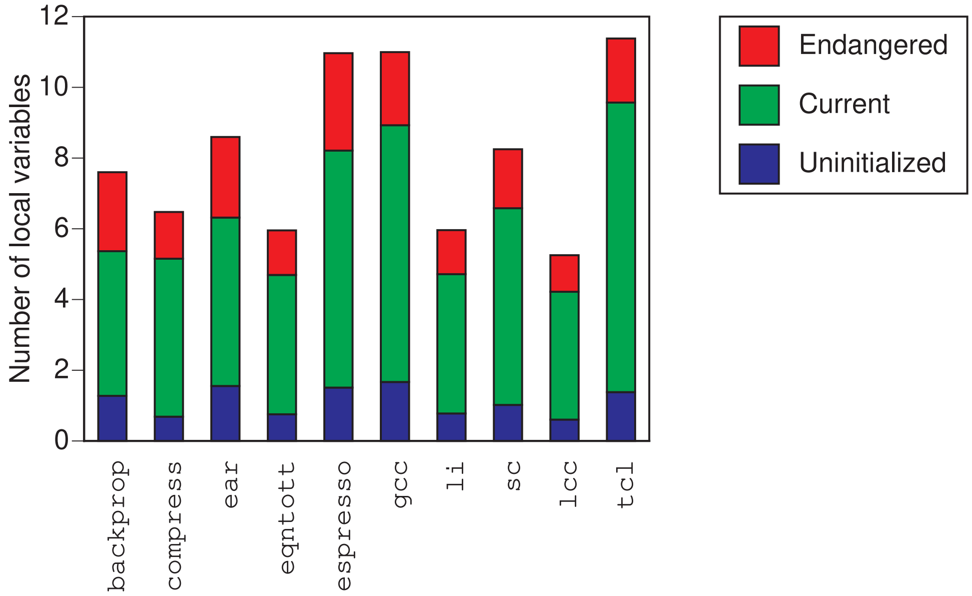


Figure 5.12: Average number of endangered variables at each breakpoint, all optimizations except register allocation enabled.

5.8 Empirical results

In this section, I present measurements of how often endangered variables are likely to occur. The metric I use in these measurements is the average number of variables that are uninitialized, current, endangered, and nonresident at a breakpoint. These numbers are gathered by considering all possible source-level control breakpoints. Although there are a large number of *global* variables that can be queried at each breakpoint, very few global variables are endangered on the average. Therefore, I show only the results for *local* variables. (Including global variables would greatly reduce the percentage of nonresident and endangered variables.) The average number of local variables in these charts is greater than in the charts of Section 4.5 because the charts in this section include *all* local variables; the charts of Section 4.5 show only locals that are register allocation candidates.

Figure 5.12 shows the results when the programs are compiled *with* global optimization and instruction scheduling, but *without* global register allocation. Since register allocation is not performed, nonresident variables cannot occur. On average, only about 10-30% of the variables are endangered at each breakpoint. The same measurements for `triangle` are shown in the first column of Figure 5.14 (the results for this program are shown separately

Program	% Suspect
`li`	13.6%
`eqntott`	36.7%
`espresso`	33.6%
`gcc`	22.0%
`alvinn`	9.0%
`compress`	13.2%
`ear`	30.4%
`sc`	30.8%
`lcc`	13.3%
`tcl`	19.8%
`triangle`	25.5%

Table 5.5: Percentage of endangered variables that are suspect in Figure 5.12.

because this program has a significantly higher average number of local variables compared to the other programs). About 16% of the variables are endangered at each breakpoint in this program.

Table 5.5 shows the percentage of endangered variables that are suspect. This table shows that the majority of endangered variables are noncurrent. Thus, the debugger can successfully classify the majority of resident variables as either current or noncurrent.

Figure 5.13 shows the results when the programs are compiled with global optimizations *and* with register allocation. (The second column of chart in Figure 5.14 shows the results for `triangle`). Over half the variables are current or uninitialized; these are the "good" cases, because the debugger can provide accurate and meaningful information to the user. Almost all the variables that cause problems for the debugger are nonresident. Therefore, when the user queries the value of a variable V, the debugger will in most cases respond with either the expected value of V or with no value at all (i.e., V is nonresident) — very seldom will the debugger report a variable as endangered. This suggests that using analysis to detect current variables can be a big win: by performing analysis to detect current variables, the debugger certifies most resident variables as current (rather than suspect) and can make optimizations transparent in many situations. If the debugger only detected whether a variable is nonresident, then the best answer that the debugger could give a user is that a variable is suspect — the user is faced with the task of determining whether the variable is suspect. If the debugger also performs analysis to determine whether a variable

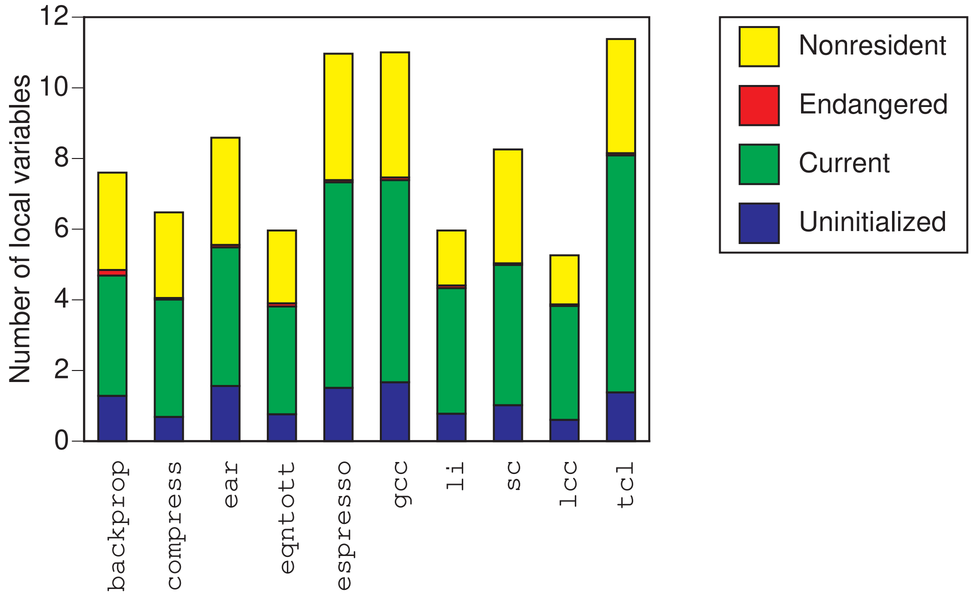


Figure 5.13: Average number of endangered variables at each breakpoint, all optimizations enabled.

is current, then in most cases where a variable is resident, the debugger need not bother the user in any way and can provide expected behavior.

It is worthwhile to compare Figures 5.12 and 5.13 to see the influence of register allocation. When global register allocation is enabled, we see two effects: First, the number of current variables decreases (the number of uninitialized variables should be unaffected since uninitialized variables depend only on the source reference statement). The reason for this is that some of the variables that are current in Figure 5.12, are allocated registers and become nonresident in Figure 5.13. Second, the number of endangered variables decreases to an almost negligible amount. The reason for this is that a majority of the endangered variables in Figure 5.12 are dead; thus, the registers allocated to these variables are re-used by the register allocator. The results of these two figures suggest that if register allocation is performed with dead code elimination, the effects of dead code elimination are manifest in the form of nonresident variables, rather than endangered variables.

Figure 5.15 shows the results weighted by dynamic execution count, for a subset of the SPEC C programs compiled with both global optimizations and register allocation enabled. Compared to the results of Figure 5.13, substantially fewer variables are uninitialized on average; this is because almost all variables are initialized at the breakpoints where the

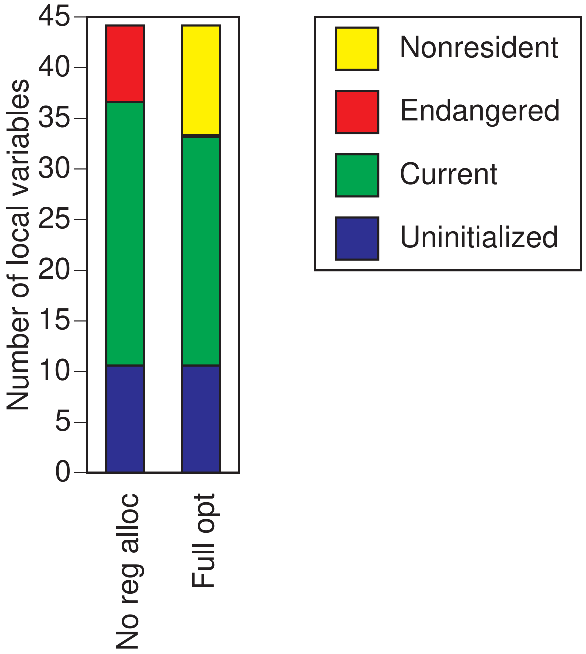


Figure 5.14: Average number of endangered variables at each breakpoint for `triangle`.

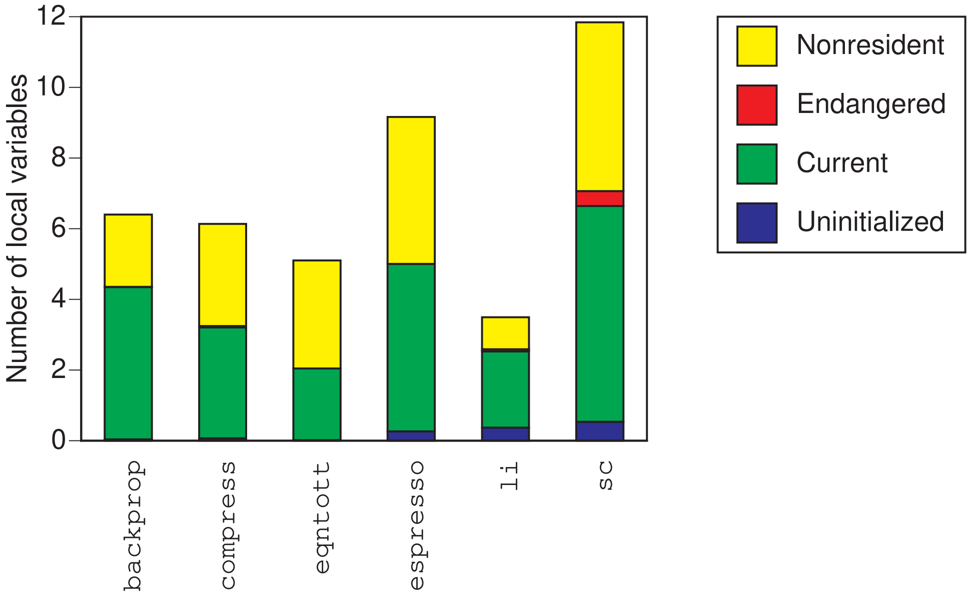


Figure 5.15: Average number of endangered variables at each breakpoint weighted by execution count, all optimizations enabled.

majority of execution time is spent. In general, however, the results of Figure 5.15 support the conclusions of the previous paragraphs: the majority of variables are either current or nonresident — endangered variables are rare.

In Section 2.4.2, I mentioned that one way in which expected behavior can be provided is by constraining optimizations. To eliminate data-value problems, optimizations can be constrained such that endangered and nonresident variables do not occur. The user can be given the option of trading off optimizations and sacrificing performance, in order to gain debugability. The measurements in this section show that the most critical optimization that must be addressed first is register allocation, because nonresident variables are the most serious data-value problem. To trade-off optimizations for debugability, the register allocator can either disallow variables from being register-residing candidates, or it can assign a register to a variable for the duration of a function. Alternatively, after register assignment, store instructions that spill the values of source-level variables to home locations in memory, can be introduced into the program. These spills can be inserted immediately after each definition of a variable, at the end of a variable's live range, or at the point where a variable becomes nonresident. Code motion techniques can be used to optimize the placement of spills by eliminating partial redundancies among spills or by scheduling spills into empty schedule slots (thus, executing spills "for free"). Future work can investigate the trade-offs in performance using these approaches.

5.9 Summary

In this chapter, I have described an approach to detecting endangered variables caused by global scalar optimizations and instruction scheduling. My approach uses two analyses. First, a local analysis is used to detect endangered variables caused by instruction scheduling and local optimizations. To gather the information required for this analysis, the compiler annotates the instruction-level representation of the program with information that describes the order in which assignments are supposed to executed in the source. These annotations allow the debugger to detect which assignments have been executed out of order at a breakpoint.

Second, a global data-flow analysis is used to detect endangered variables caused by global scalar optimizations. Since data-flow analysis is a well-understood technique, there are limited obstacles to overcome for an implementation of these techniques in a compiler. The analyses are very similar to other analyses that are done by the compiler and can thus take advantage of an infrastructure that is already present. This is in contrast to other approaches that require specialized data-flow analyses and program representations [36, 106]. To gather the information required for the data-flow analyses, the program intermediate representation is annotated during optimizations to mark hoisted and sunk assignments, and additional markers are inserted to indicate points from which source-level assignments are eliminated. The data-flow analyses can be performed either by the compiler after optimizations and code generation, or by the debugger.

I have presented measurements of the effects of optimizations on a source-level debugger's ability to retrieve variable values. Measurements show that a debugger is more likely to be affected by register allocation than by other global optimizations. Moreover, by performing the analyses described in this chapter, the debugger can in many cases provide expected behavior when responding to a user variable query.

Chapter 6

Breakpoints in Optimized Code

To accurately set and report breakpoints, the debugger requires two mappings: (1) the *the object-to-source* mapping and (2) the *source-to-object* mapping. The object-to-source mapping maps an instruction I to the source-level statement for which I is generated. When an asynchronous break occurs at an instruction I, the debugger uses the object-to-source mapping to report to the user the statement in which the asynchronous break occured (asynchronous breaks are defined in Section 2.1). The source-to-object mapping maps a statement S to one or more instructions. The debugger uses this mapping to determine the instructions at which breakpoints are actually set when the user sets a breakpoint at S.

The object-to-source and source-to-object mappings are constructed by the compiler and transmitted to the debugger via the object file. To construct the object-to-source mapping, the compiler must maintain enough information in both the machine-independent and instruction-level intermediate representations to trace each instruction I back to the source statement for which I is generated. The compiler must update this information as transformations are performed so that the mapping from instructions in the optimized translation to statements in the source program remains accurate. For example, when partial redundancy elimination inserts an expression E into the program representation, the compiler must find the statement from which E is hoisted by searching forward in the control-flow graph.

To construct the source-to-object mapping, the compiler must keep track of the effects of optimizations on the mapping of source breakpoints to instructions. The compiler must update the source-to-object mapping to reflect the effects of optimizing transformations;

transformations may require that a breakpoint be moved, eliminated, or duplicated. For example, loop peeling duplicates the instruction sequence that is generated for a statement S that lies within the peeled loop body. When the user sets a breakpoint at S, the debugger must map this breakpoint to all copies of S's instruction sequences.

In this chapter, I discuss the effects of optimizations on the object-to-source and source-to-object mappings. I describe the construction of these mappings in the presence of global optimizations. In the next section, I describe in detail the representation of the object-to-source and source-to-object mappings inside the compiler's intermediate representation. In the following sections, I describe how the compiler updates this representation as it performs optimizations. In Section 6.2, I describe how the compiler updates the source-to-object mapping when it performs code duplicating optimizations such as loop unrolling and peeling. In Section 6.3, I describe the effects of code eliminating optimizations on the source-to-object mapping and describe the necessary bookkeeping in the compiler. In Section 6.4, I describe how transformations that insert code require accurate updating of both the source-to-object and object-to-source mappings by the compiler. In Section 6.5, I describe the effects of code motion on the source-to-object mapping and how this may lead to unexpected behavior. In Section 6.6, I describe the effects of instruction scheduling on setting and reporting of breakpoints. Finally, in Section 6.7, I present measurements.

6.1 Representing the mapping to the source

To map from an instruction back to the source, the compiler must keep track of the correspondence between source statements and operations in the intermediate representation (IR), at all phases of compilation. Depending on the form of the IR, an IR operation can be either a machine-independent operation, or a target machine instruction. A source statement can be represented by a file name, line number, and line offset that together mark the beginning of the source statement in the source text. The compiler may keep more detailed information about each statement S — such as a character count that indicates the extent of S, or an additional line number and offset indicating the end of S — such that the debugger user interface can highlight S in the source. The exact representation of source line number information, however, is not the topic of this thesis.

The compiler can also keep very fine-grained mappings by maintaining the correspondence between each IR operation O and the source-level *expression* for which O is generated. Such a fine-grained mapping can be used to accurately pin-point to the user the expression where an asynchronous break occurs. The CXdb debugger [20] uses such an expression-level mapping to highlight source expressions that are executed as the user single steps through the object code. In the rest of this chapter, I assume a statement-level mapping to the source.

Typically, the mapping from IR operations to source statements is maintained via *statement labels* that label sequences of IR operations with the source statement for which the sequence is generated. If an expression-level mapping is kept by the compiler, then additional *expression labels* are necessary in the IR. Alternatively, each IR operation O can be annotated with the source expression for which O is generated; that is, an additional field containing the source expression is added to the representation of O. Annotations are redundant for statement-level mappings because sequences of IR operations have the same annotations — statement labels factor this information. As we will see in Section 6.6, however, annotations are necessary to keep track of the correspondence between instructions and source statements during instruction scheduling.

Each time the compiler lowers the program representation — for example, from an abstract syntax tree to a machine-independent representation, or from a machine-independent representation to a machine-specific instruction-level representation — the compiler must lower statement labels to the new representation, and pass annotations along to operations in the new representation. During final code emission, statement labels and annotations are passed along to the debugger via the object file's symbol table.

In the absence of optimizations, the code sequences generated for source statements are disjoint and there is exactly one label for each source statement. Therefore, statement labels are sufficient for implementing both the source-to-object and object-to-source mappings: To set a breakpoint at a statement S, the debugger sets a breakpoint at the instruction labelled by the statement label for S. To find the statement for which an instruction I is generated, the debugger searches backward in the object code for the nearest statement label.

In the presence of optimizations, statement labels are no longer sufficient for implementing both the source-to-object and object-to-source mappings. Optimizations such as code motion and instruction scheduling can fragment the IR generated from a statement and interleave its IR with IR fragments from other statements. Consequently, there may exist several sequences of IR operations for a single statement S. Each of these code sequences must be separately labelled to identify them with statement S; this is necessary to report the source location of an asynchronous break accurately at any of these code sequences. When the user sets a breakpoint at S, however, this breakpoint cannot be mapped to all of the code fragments belonging to S, because the breakpoint will then be executed multiple times for a single statement. Moreover, a breakpoint cannot be mapped to only one of the code sequences, because some optimizations, such as loop unrolling and function inlining, require that a breakpoint be mapped to more than one instruction. Also, because of code elimination, an instruction I generated for a statement S may be labelled by statement labels belonging to statements other than S (the statement label of a statement that is completely eliminated is moved to be adjacent to the next statement label). If an exception occurs at I, then the debugger may incorrectly report the exception to be at a statement other than S.

Thus, in the presence of optimizations, separate labelling mechanisms are needed to implement the source-to-object and object-to-source mappings. To identify the statement for which an IR fragment is generated, IR fragments are labelled with *marker* labels (i.e., marker labels implement the object-to-source mapping). Like the statement labels, marker labels are lowered with the representation and passed on to the debugger (the object file format must somehow distinguish marker labels from statement labels). When an asynchronous break occurs at an instruction I, the debugger searches backwards in the object code for the earliest marker label that identifies the statement to which the instruction belongs. The source-to-object mapping is still implemented with statement labels: When the user sets a breakpoint at a statement S, the debugger maps this breakpoint to the instruction labelled by the statement label of S. As we will see later, a statement can generate multiple statement labels because of code duplicating optimizations, in which case the debugger maps the breakpoint to all of these labels. Also, a label L may be moved because no instructions are generated for the L's statement; the user must be warned of L's new location.

Each time the compiler performs a transformation that affects the mapping from IR operations to source statements (e.g., loop unrolling or code hoisting), the compiler must update the source-to-object and object-to-source mappings encoded in the IR labels. This update is necessary to enable the debugger both to report the location of an asynchronous break, and to set breakpoints accurately. This update may involve moving or duplicating a statement/marker label, or introducing a new marker label to identify the source statement of a new IR fragment. Also, some optimizations may move a statement label such that the user must be warned of the label's new location; therefore, the compiler must annotate a moved statement label L with information indicating that the debugger must warn the user of L's location when a breakpoint is set at L.

In the following sections, I look at the transformations that affect the object-to-source and source-to-object mappings. I describe the updates necessary to these mappings that the compiler must perform for each of these transformations.

6.2 Code duplication

Code duplicating transformations duplicate basic blocks in the control-flow graph. Examples of code duplicating transformations include loop unrolling and peeling, function inlining and cloning, and tail duplication. When basic blocks are duplicated, the source-to-object mapping becomes one to many: if the user sets a breakpoint at a statement S, the debugger must map this breakpoint to all duplicated copies of the code generated for S. When the compiler performs code duplicating transformations, it must correctly update the source-to-object mapping. If a duplicated basic block B contains statement or marker labels, then the compiler must also duplicate these labels along with the rest of the code in B. Similarly, if a duplicated operation O contains an annotation, then the annotation must also be duplicated and attached to the newly created copy of O.

Figures 6.1 and 6.2 show an example illustrating the effects of code duplication on the source-to-object mapping. Figure 6.1 shows a C code fragment containing a while loop. Figure 6.2(a) shows the control-flow graph of this source before any transformations; the loop is translated into two basic blocks: one containing statements `S3` and `S4`, and another containing statement `S2` (the loop termination test). Each statement in the source maps to exactly one statement label in the IR. This translation of the loop makes it impossible for partial redundancy elimination to hoist code from statements `S3` and `S4`, because these statements are executed on only some paths through the loop; code can only be hoisted speculatively out of these statements. Figure 6.2(b) shows the control-flow graph after control-flow transformations have partially peeled the loop. A duplicate of the block containing statement `S2` has been merged with the block containing statement `S1`, and the block containing the original copy of `S2` has been merged with the block containing statements `S3` and `S4`. The loop body now comprises only one basic block, enabling partial redundancy elimination to hoist loop-invariant code from `S3` and `S4`.

If the user sets a breakpoint at statement `S2` (the while loop test), the debugger must map this breakpoint to two locations in the object code, corresponding to the two copies of this statement in Figure 6.2(b). The first instance of this breakpoint is executed on initial entry to the loop (i.e., when `i == 0`); the second instance is executed on each subsequent iteration (i.e., when `i > 0`).

```
S1: i = 0;
S2: while (i < 10) {
      S3: A[i] = 0;
      S4: i++;
    }
S5: foo(A);
```

Figure 6.1: Source code for Figure 6.2.

6.3 Code elimination

Optimizations that eliminate code — for example, dead code elimination and partial redundancy elimination — cause a problem for debugging when they eliminate all the code associated with a statement label. If the user sets a breakpoint at a statement S and all the code associated with the statement label of S has been eliminated by optimizations, then there are no instructions unique to S to which the debugger can map the breakpoint.

To deal with the situation where the user sets a breakpoint at a statement S whose code has been completely eliminated, the debugger can map the breakpoint to the statement label S' that is within the same block as S and occurs immediately after S. The user can then be warned that the breakpoint has been mapped to the new statement S'; this warning can be issued either when the user sets the breakpoint, or when the breakpoint is executed.

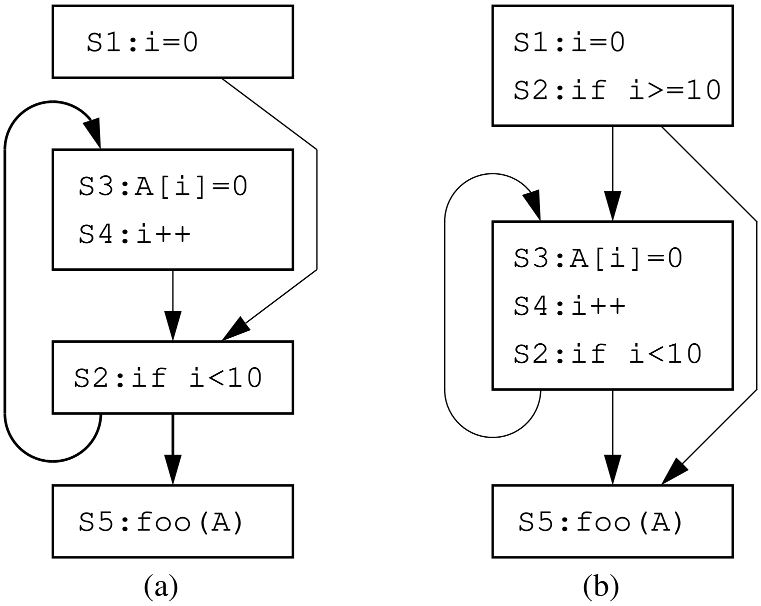


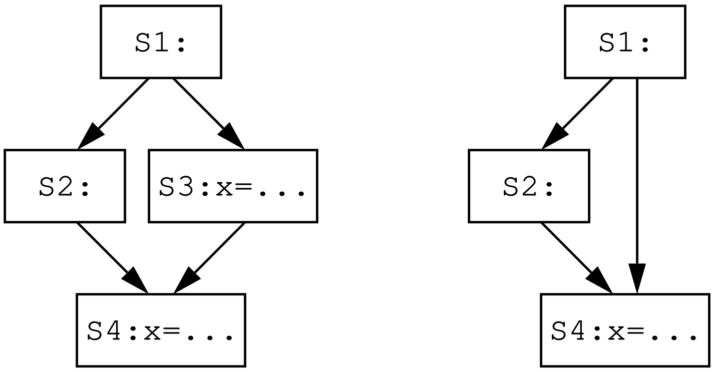
Figure 6.2: Example: Code duplication (a) Control-flow graph before transformation (b) Control-flow graph after loop peeling.

There are several complications, however, that the debugger must be prepared to deal with when the user sets a breakpoint at an eliminated statement S. One complication is that the statement label for S may be at the end of the basic block; thus, there may be no other statement label occuring after S to which the breakpoint at S can be mapped. In this case, the debugger can map the breakpoint at S such that the break is taken *after* the execution of the last instruction in the basic block of S.

Another complication is that code elimination may eliminate all the code in the basic block B containing the statement label of S, so that B is eliminated from the control flow graph; as a result, there are no instructions to map the breakpoint at S. This is illustrated in the example shown in Figures 6.3 and 6.4. Figure 6.3 shows a C code fragment containing an if statement. On the else branch of this if statement, there is an assignment statement (`S3`) that assigns to a variable `x`. There is another assignment to `x` at statement `S4`, immediately after the if statement. Figure 6.4(a) shows the control-flow graph of the IR generated for this source. The assignment to `x` at statement `S3` is dead because `x` is re-assigned at statement `S4` (i.e., the value assigned by statement `S3` is never used). Figure 6.4(b) shows the control-flow graph after optimizations; dead code elimination eliminates statement `S3`, and the block containing `S3` subsequently becomes empty and is eliminated.

```
S1: if (...)
       S2:
    else
       S3: x = ...
S4: x = ...
```

Figure 6.3: Source code for Figure 6.4.



(a) (b)

Figure 6.4: Example: Code elimination (a) Control-flow graph before transformation (b) Control-flow graph after code elimination.

If the user sets a breakpoint at `S3`, there are no instructions to which the debugger can map this breakpoint. The debugger has three options in this situation: (1) disallow the user from setting a breakpoint at `S3`, (2) map the breakpoint to statement `S1` in the previous basic block, or (3) map the breakpoint to statement `S4` in the next basic block. In the last two cases, the breakpoint will execute more often than expected: the user expects that a breakpoint at statement `S3` is executed only when the if statement's test expression evaluates to false. Since the user must be warned of unexpected behavior, the debugger should warn the user of the new location to which the breakpoint has been moved. Based on the new location of the breakpoint, the user will observe that the break will occur more often that it is supposed to. Note, that even if the debugger stops execution at `S1` or `S4`, data-value problems may prevent the user from being able to query the values of the variables that determine the outcome of the if statement test.

Since the branch instruction that implements the test expression is not eliminated, the debugger can determine at runtime the value of the if statement's test. One way that the debugger can accomplish this is by setting a conditional breakpoint at the branch instruction that implements the if statement's test. When the breakpoint is executed, the debugger evaluates the source values of the branch instruction and passes control to the user only when it determines that the branch will evaluate to false (i.e., only when the branch evaluates to the direction of statement `S3`); otherwise, the debugger resumes execution. Code patching techniques [61] could be used for a faster implementation of this conditional breakpoint: the branch instruction is replaced with an unconditional branch to an out-of-line code sequence that evaluates whether the debugger should be invoked. Note, that such a conditional breakpoint is invasive, because it changes the timing of the debuggee. During interactive debugging, however, this invasiveness may not be a problem.

The problems caused by code elimination are further complicated by the interaction that can occur between code duplication and elimination. When performed after code duplication, code elimination can eliminate a copy of a statement's label; that is, code duplication may duplicate a statement S's statement label and one of these duplicate labels can be subsequently eliminated. Figure 6.5 illustrates this problem. This figure shows the effects of code eliminating optimizations on the code of Figure 6.2(b). Constant propagation has propagated the constant zero value of `i` from statement `S1` to statement `S2`. Constant folding has subsequently eliminated the first instance of statement `S2`'s loop test expression. After this transformation, there is no code associated with the first instance of statement `S2`'s label in the loop preheader, but there is still code associated with the second instance in the loop body. Assume the user sets a breakpoint at statement `S2` (in the source of Figure 6.1). If the debugger sets a breakpoint only at the instance of `S2` that occurs within the loop body, then the breakpoint is executed only after the loop body has been executed at least once — this is unexpected behavior because the user expects the breakpoint to also execute before the first time the loop body is executed. To deal with this situation, the debugger can set another breakpoint after the last instruction in the preheader basic block.

Note another difficulty illustrated by this example: the debugger cannot convey this optimization by simply telling the user that statement `S2` has been eliminated (e.g., by using a different font for `S2`).

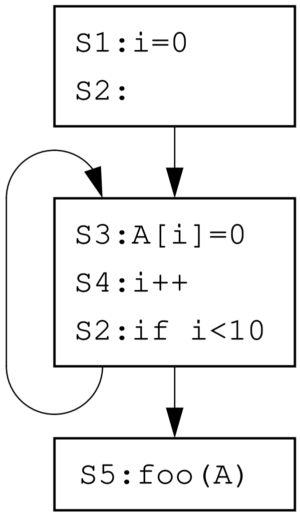


Figure 6.5: Example: Elimination of code associated with a duplicate statement label.

6.4 Code insertion

Some optimizations insert code into the program representation. For example, partial redundancy elimination inserts code that saves the results of a computation into a compiler temporary; partial dead code elimination introduces source-level assignments at new points in the program (rendering other assignments dead); strength reduction inside of loops introduces new compiler-synthesized induction variables that are initialized at a loop's preheader and updated within the loop's body. Code inserted by optimizations can usually be logically associated with some source-level statement; for instance, the update of a compiler-synthesized induction variable can be associated with a source-level update of a user variable. To report the source location of a runtime exception that occurs at inserted code, the debugger must accurately determine the source statement associated with the inserted code.

Figures 6.6 and 6.7 illustrate how partial redundancy elimination can insert code into a program. Figure 6.6 shows a C code fragment containing an if statement. Figure 6.7(a) shows the control-flow graph of the IR generated for this C code; Figure 6.7(b) shows this IR after partial redundancy elimination. In Figure 6.7(a), the expression `y+z` that occurs within statement `S3` is partially redundant because this expression is computed on only some paths that lead to this statement (i.e., those paths that go through statement `S2`). In

Figure 6.7(b), partial redundancy elimination has introduced a copy of the expression `y+z` into a new basic block; now the expression `y+z` is fully redundant at statement `S3` because this expression is evaluated on all paths leading to `S3`. The compiler has inserted the label `$h` before this new instance of `y+z` to mark that this expression has been inserted by partial redundancy elimination (i.e., code hoisting). Both instances of the expression `y+z` are stored into a compiler temporary (this temporary is named `$1` in the figure), and the expression `y+z` that occurs within statement `S3` is replaced by a reference to this temporary.

There are several different forms of intermediate representation that lead to slightly different implementations of partial redundancy elimination. In the `cmcc` compiler, the compiler explicitly assigns the value of an expression E whose value is later recomputed, into a compiler temporary T and replaces the later computation of E with a fetch from T. Expressions that save values into compiler temporaries (e.g., `$1=y+z`) are referred to as *saves*. Some compilers use a different form of IR wherein all instances of an expression E target the same temporary (or symbolic register); in this manner, all instances of E are also implicit saves of E. The discussion I present here, also generalizes to these intermediate forms.

As Figure 6.7(b) illustrates, partial redundancy elimination can insert a computation into the IR for one of two reasons: (1) to save the value of an expression occuring within a source-level statement (e.g., the save inserted at statement `S2`); and (2) to compute and save the value of an expression at an earlier point in the program (e.g., the save labeled `$h`). Saves that are inserted to save the value of an expression within a source-level statement S are considered part of S; thus, the optimizer must ensure that the save of an expression E is inserted after the statement or marker label of the statement containing E. In Figure 6.7(b), the expression `$1=y+x` is inserted immediately after the statement label `S2` so that the code for this save is correctly associated with statement `S2`. An asynchronous break occuring within this save is correctly reported at `S2`.

Saves that are inserted to compute and save an expression E at an earlier point in the program are considered part of later statements that are hoisted to earlier points in the program; if such save expressions cause an exception, the debugger must report that the exception occured within the statement from which the expression E is hoisted. In Figure 6.7(b), the save `$1=y+z` labelled `$h` is computing the expression `y+z` of statement `S3` at an earlier point in the program; thus, this save is associated with statement S3.

When partial redundancy elimination hoists an expression E to a new location in the program, the compiler must perform a forward search in the control-flow graph to find the original copy of E. The compiler inserts a marker before the inserted copy of E to identify the source statement from which E is hoisted. For example, in Figure 6.7(b), after the compiler inserts the save labeled $h, it must perform a forward search to find statement S3, the statement from which this expression is hoisted. Since partial redundancy elimination allows many-to-one hoistings, the expression E may be associated with more than one statement — many-to-one hoistings, however, tend to be very rare. Note that the forward search is guaranteed to find the original copy of E. If the original copy of E contains annotations (e.g., an annotation indicating the location of the expression in the source code), these annotations must be transferred to the inserted instance of E.

Partial dead code elimination causes problems similar to those for partial redundancy elimination, but in the reverse direction: a backward search is necessary to find the source statement(s) from which an assignment is sunk.

```
S1: if (...)
       S2: x = y+z;
S3: a = y+z;
```

Figure 6.6: Source code for Figure 6.7.

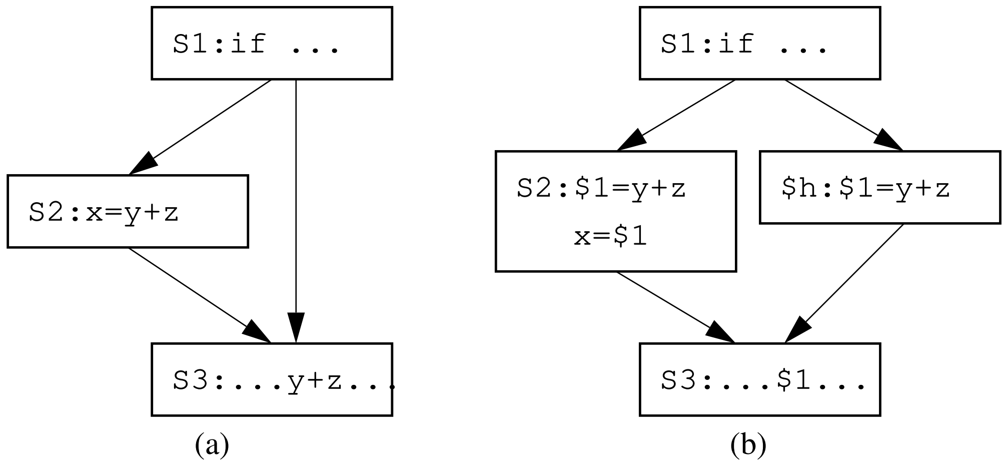


Figure 6.7: Example: Partial redundancy elimination.

Strength reduction inside of loops introduces a new compiler-synthesized induction variable that is a linear function of a source-level induction variable; this synthesized variable is then initialized at the same points that the source-level induction variable is initialized and updated at the same points that the source-level induction variable is updated. Figure 6.8 illustrates the bookkeeping needed for induction variable strength reduction. Figure 6.8(a) shows the IR of Figure 6.5 after induction variable strength reduction: the address expression `A[i]` at statement `S3` has been replaced with a compiler-synthesized temporary (`$1`); this temporary is initialized to the base address `A` immediately after the initialization of `i` at statement `S1` in the loop preheader, and incremented by four each time the induction variable `i` is incremented at statement `S4` (each element in array `A` is of size 4). The variable `$1` is always linearly related to variable `i`; if `i` is eliminated the debugger can recover the value of `i` from `$1`. Since `$1` and `i` are so closely related, it is logical to associate the code that initializes and updates `$1` with the corresponding statements that initialize and update `i`. In Figure 6.8(a), this association is made by inserting this code immediately after the initialization and update code of statements `S1` and `S4`, respectively; the statement labels for these two statements now also label these inserted assignments as part of statements `S1` and `S4`. Note that, alternatively, the compiler could have inserted these assignments in between the statement labels and the source assignments.

There is an additional benefit to associating the assignments to compiler-synthesized induction variables in this manner. Figure 6.8(b), shows the code of Figure 6.8(a) after further optimizations. Linear function test replacement has replaced the use of `i` at the test expression of statement `S2` with the use of `$1`. After this replacement, the only use of `i` is the update expression of statement `S4`; thus, induction variable elimination eliminates `i` altogether by eliminating the initialization of `i` at statement `S1` and the update of `i` at statement `S4`. Note, however, that because of the way the initialization and update of `$1` are associated with statement labels, the original source code that initializes and updates `i` is in effect replaced with the corresponding code for `$1`; now, the debugger can accurately consider `$1` as replacing `i`.

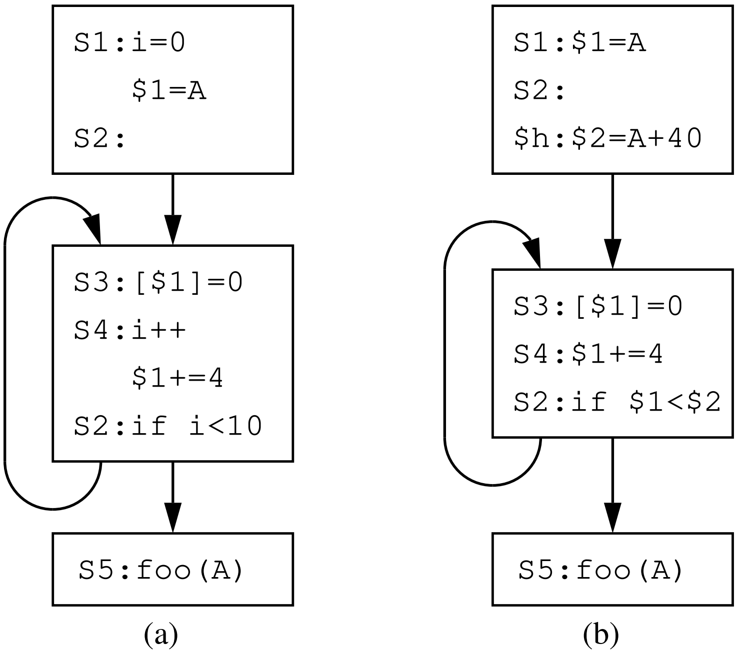


Figure 6.8: Example: (a) Induction variable strength reduction (b) Linear function test replacement and induction variable elimination.

6.5 Setting breakpoints in the presence of code motion

In the preceding sections, I have described a policy for maintaining the mappings from source statements to labels in the IR when the compiler performs classical machine-independent optimizations. As a result of this policy (and in the absence of instruction scheduling), if the program resulting after global optimizations is executed by the user, source breakpoints are mostly executed in the order expected in the source. As we saw in Section 6.3, however, sometimes breakpoints may execute unexpectedly: because of code elimination, some breakpoints may be moved next to other source breakpoints; sometimes such a moved breakpoint executes more often than expected in the unoptimized translation.

Another cause for unexpected behavior, however, is due to partial redundancy elimination. Because of partial redundancy elimination, an exception may occur at an instruction that has been hoisted out of a source-level statement S, before a breakpoint at S is reached. Partial redundancy elimination operates as a series of deletions and insertions of code into the program representation. In the preceding sections, I have described how the mapping between source and IR is updated when the compiler performs code insertions and deletions. As a result of this bookkeeping, when partial redundancy elimination hoists an expression

E from a statement S, the breakpoint for S remains in the original basic block from which E was hoisted. For example, in Figure 6.7(b), the statement label for statement `S3` is not moved even though the expression `y+z` of this statement executes at an earlier point in the program. This breakpoint mapping will lead to unexpected behavior if the expression E that is hoisted causes an exception and the user has set a breakpoint at `S3`; that is, exceptions may execute out of order with respect to source breakpoints. For example, in Figure 6.7(b), if the user sets a breakpoint at statement `S3`, and the expression `y+z` causes an exception (e.g., floating-point overflow), then the exception will occur before the breakpoint is executed; this is unexpected behavior.

This situation is further illustrated in the example of Figure 6.9. In Figure 6.9, statement S_3 contains a load operation (`*p`) that can potentially trap (e.g., due to an illegal address) and this load has been hoisted out of the loop by partial redundancy elimination. The user may set a breakpoint at statement S_3 to halt execution before this trap occurs, but the trap will occur before the breakpoint is executed. If the user knew a priori that statement S_3 would trap (e.g., the program had been run once before resulting in the trap) then the user could have performed a post-mortem analysis of the program with the debugger, in which case the debugger could have informed the user that a breakpoint at S_1 would halt execution before the trap.

Source program	After code hoisting
S_1:	S_1:
	`tmp = *p`
`do {`	`do {`
S_2:	S_2:
S_3: `...*p...`	S_3: `...tmp...`
S_4:	S_4:
S_5: `} while(`E`)`	S_5: `} while(`E`)`

Figure 6.9: Example: Hoisting of potentially trapping operations

Some researchers [109, 35, 106] have suggested that an alternative strategy exists for setting breakpoints at statements from which code has been hoisted. An alternative strategy for setting a breakpoint at statement S is to set a breakpoint at the earliest instruction generated for S (Zellweger [109] calls this a *semantic* breakpoint mapping). This strategy guarantees that the breakpoint for S is taken before any exceptions (or other side effects) inside S. Applying this strategy to Figure 6.9, a breakpoint at S_3 is set before the loop where the load from p is executed. This strategy results in the following behavior: the breakpoint for statement S_3 occurs out of order with respect to statement S_2. Moreover, the breakpoint is taken only once before the loop, rather than on every trip through the trip as expected. The debugger could set a breakpoint both before the loop and inside the loop, but then the breakpoint at S_3 executes one additional time before the loop starts.

This issue of where to map the breakpoint for statement S_3 arises only if the user wants to halt execution before a side effect from S_3 happens. In our language model (i.e., C), the only side effects that can happen at the source level are assignments and exceptions. A compelling reason to map a statement S's breakpoint to the earliest points where instructions from S have been hoisted would be to halt execution before a side effect of S occurs. However, in Section 6.7, I show that while code hoisting occurs often (that is why it is a useful optimization), only very rarely does it hoist code that either assigns to a source-level variable or potentially traps. Thus the approach I propose is to retain the breakpoint at the original statement from which code is hoisted, and to provide guidance to the user in the rare case that an exception occurs at hoisted code.

6.6 Instruction scheduling

By interleaving instruction sequences generated from different source statements, instruction scheduling causes several problems. Because the execution of statements is overlapped, there are no clear execution points in the object code that correspond to boundaries between the instruction sequences generated from different statements. In general, it becomes impossible for the debugger to map the breakpoint at a statement S to an instruction, such that all instructions generated from statements prior to S have completed execution and no instructions generated from subsequent statements have started execution. Providing source-level breakpoints thus becomes difficult.

After scheduling, the compiler must make a decision on where to map the breakpoint for a statement S (i.e., where to insert the statement label of S). This decision is not a clear one since any of the instructions generated for S are potential candidates — many different possibilities exist. One obvious choice for mapping a breakpoint at a statement S is to set this breakpoint at the earliest scheduled instruction I that has been generated for S. This is not necessarily the best choice, however, because the first instruction will not always guarantee that events such as assignments, exceptions and breakpoints will execute in the expected source order. Another possibility is to set the breakpoint at the earliest scheduled side-effecting instruction I generated for S, where I is an assignment, function call or branch instruction. The rationale behind this strategy is that side-effecting instructions are more likely to execute close to their original source order. This strategy, however, still does not guarantee that exceptions and breakpoints are executed in their expected order.

When mapping a breakpoint at a statement S, there is no real reason why the breakpoint should be mapped to an instruction that has been generated for S. For example, to guarantee that breakpoints are at least taken in their original order, the compiler can insert source labels in their original basic block offsets after scheduling. Yet another possibility, is to simply place labels according to their original source order but evenly spaced apart.

The point is that after instruction scheduling, there is, in general, no mapping of breakpoints that guarantees that events are executed in their expected source order. Although many different possibilities exist and the strategy used to map the breakpoint at a statement S may influence the number of variables that are endangered at S (i.e., there may different numbers of variables endangered at different stopping instructions), there does not seem to be one clearly superior strategy. In Section 6.7, I consider two different strategies for mapping breakpoints and measure the effects of each strategy on the number of endangered variables.

To maintain the mapping of instructions to source statements, instruction scheduling also requires careful bookkeeping by the compiler. Since instruction scheduling does not schedule statement or marker labels, each instruction I must be annotated with information that identifies the statement from which I is generated, before instruction scheduling is performed. Because the instruction sequence associated with a statement or marker label is fragmented by scheduling, marker labels are necessary to identify the source statement from which an instruction is generated. Therefore, after instruction scheduling, the compiler must insert marker labels into the scheduled instruction sequence.

Program	Total breakpoints source	object	deleted
li	2594	2723	10
eqntott	1267	1334	24
espresso	7424	8047	49
gcc	28433	28951	354
alvinn	140	146	0
compress	429	435	3
ear	1108	1173	0
sc	3400	3528	49
lcc	5970	6171	32
tcl	7820	8188	152
triangle	5312	5419	21

Table 6.1: Breakpoint statistics

6.7 Empirical results

The second column of Table 6.1 shows the number of source breakpoints for each program in our application suite. These numbers correspond to the number of statement labels generated by the front end (i.e., the number of statement labels before optimizations). The `lcc` front end generates a statement label for each C language sequence point (i.e., semicolon and comma) as well as for `if` and `while` loop test expressions. The third column of this table shows the number of statement labels in the object code. In almost all of the programs, the number of labels in the object code is more than the number of source breakpoints; therefore, for some source-level breakpoints the debugger will set more than one breakpoint in the object code. This is caused by code duplication performed by the loop peeling phase of `cmcc`. The fourth column of Table 6.1 shows the number of breakpoints that were deleted by code eliminating optimizations and were moved to an adjacent basic block. As I described in Section 6.3, these are the breakpoints that will cause unexpected behavior. This measurement shows that the debugger user will probably very seldom encounter such problematic breakpoints.

Table 6.2 shows the number of instructions that are inserted by code hoisting and sinking. The second, third and fourth column of this table show the total number of instructions, the number of instructions inserted by code hoisting and the number of instructions inserted by code sinking, respectively. There are a significant number of instructions that are inserted

Program	Instructions total	Instructions hoist	Instructions sunk	Avg. distance hoist	Avg. distance sunk	Trapping instructions total	Trapping instructions hoist	Trapping instructions sunk	Avg. distance hoist	Avg. distance sunk
`li`	11550	387	1328	1.35	1.36	835	16	71	1.00	0.90
`eqntott`	5808	669	1666	2.04	1.40	482	13	70	1.10	0.96
`espresso`	40116	3631	5618	2.09	2.06	6016	202	561	1.40	0.97
`gcc`	148534	8141	23200	1.69	2.92	18627	669	1865	1.10	2.20
`alvinn`	561	167	162	2.28	1.79	90	1	40	1.00	1.30
`compress`	2180	130	999	2.27	2.03	96	3	24	1.00	0.17
`ear`	6503	537	940	3.20	1.88	951	26	144	2.70	2.20
`sc`	19658	2593	6839	2.50	3.83	842	18	70	1.60	1.30
`lcc`	34064	2295	3545	1.53	1.84	4791	250	293	1.10	1.80
`tcl`	27725	1256	3531	1.66	3.73	3128	99	276	1.50	2.60
`triangle`	23638	1166	5327	2.69	3.85	3204	32	296	1.60	4.10

Table 6.2: Number of hoisted and sunk trapping instructions

by code motion, implying that the compiler found ample opportunity to perform these transformations. The fifth and sixth columns in this table show the average number of source breakpoints across which an instruction was hoist and sunk, respectively. These numbers show that even though the optimizer found many opportunities for performing code hoisting and sinking, code was hoisted through an average of only about 1.2 source statements, while sinking sunk code through an average of about 2.1 source statements.

The next three columns of Table 6.2, show the total number of instructions that can potentially trap and the number of such instructions that are inserted by code hoisting and sinking. Very few of the instructions that are been inserted by code hoisting or sinking can potentially trap. Comparing the number of hoist and sunk trapping instructions with the total number of potentially trapping instructions, we see that it is very unlikely that the user will run into a trap at a hoisted instruction. The last two columns of this table show the average number of source breakpoints that these trapping hoist/sunk instructions were moved. These measurements shows that (1) the user can halt execution before an exception at a hoisted instruction by setting a source breakpoint only a few statements away, and (2) an exception at a sunk instruction I will occur very close to the statement from which I is sunk.

To measure the influence of the breakpoint mapping after instruction scheduling, I consider two different strategies for mapping breakpoints:

1. Insert the statement label before the first scheduled instruction generated for S.

2. Insert the statement label before the first scheduled side-effecting instruction generated for S (i.e., assignment, branch, or function call instruction).

Figure 6.10 shows the percentage of breakpoints with one and two or more endangered variables using the two strategies. This figure shows the influence of the breakpoint mapping using two different combinations of optimizations (in addition to register allocation): (1) instruction scheduling with no global optimizations (i.e., instruction scheduling only), and (2) instruction scheduling with full global optimizations (including sinking). Except for `backprop`, the number of endangered variables does not vary significantly with the breakpoint mapping. These measurements show that the influence of using a different breakpoint mapping on the number of endangered variables is negligible. Thus either strategy may work well in practice and the user can be given a choice of breakpoint mapping.

6.8 Summary

To set breakpoints and report asynchronous breaks, the debugger must accurately keep track of the correspondence between instructions and source statements. In this chapter, I have described how this relatively straight-forward task can be accomplished using special statement and marker labels in the intermediate representation. Statement labels mark the points to which source breakpoints are mapped. Marker labels identify the statement from which an instruction is generated and are used for reporting the source location of a runtime exception.

I consider four different transformations: (1) code duplication, which duplicates statement and marker labels; (2) code elimination, which may delete all the code associated with a statement label requiring that the label be moved; (3) code insertion, which requires that the compiler identify the inserted code with a source statement; and (4) instruction scheduling, which requires that the compiler decide on how to insert statement labels after code scheduling. Measurements show that the user is unlikely to encounter unexpected behavior in the presence of the first three transformations. Measurements also show that

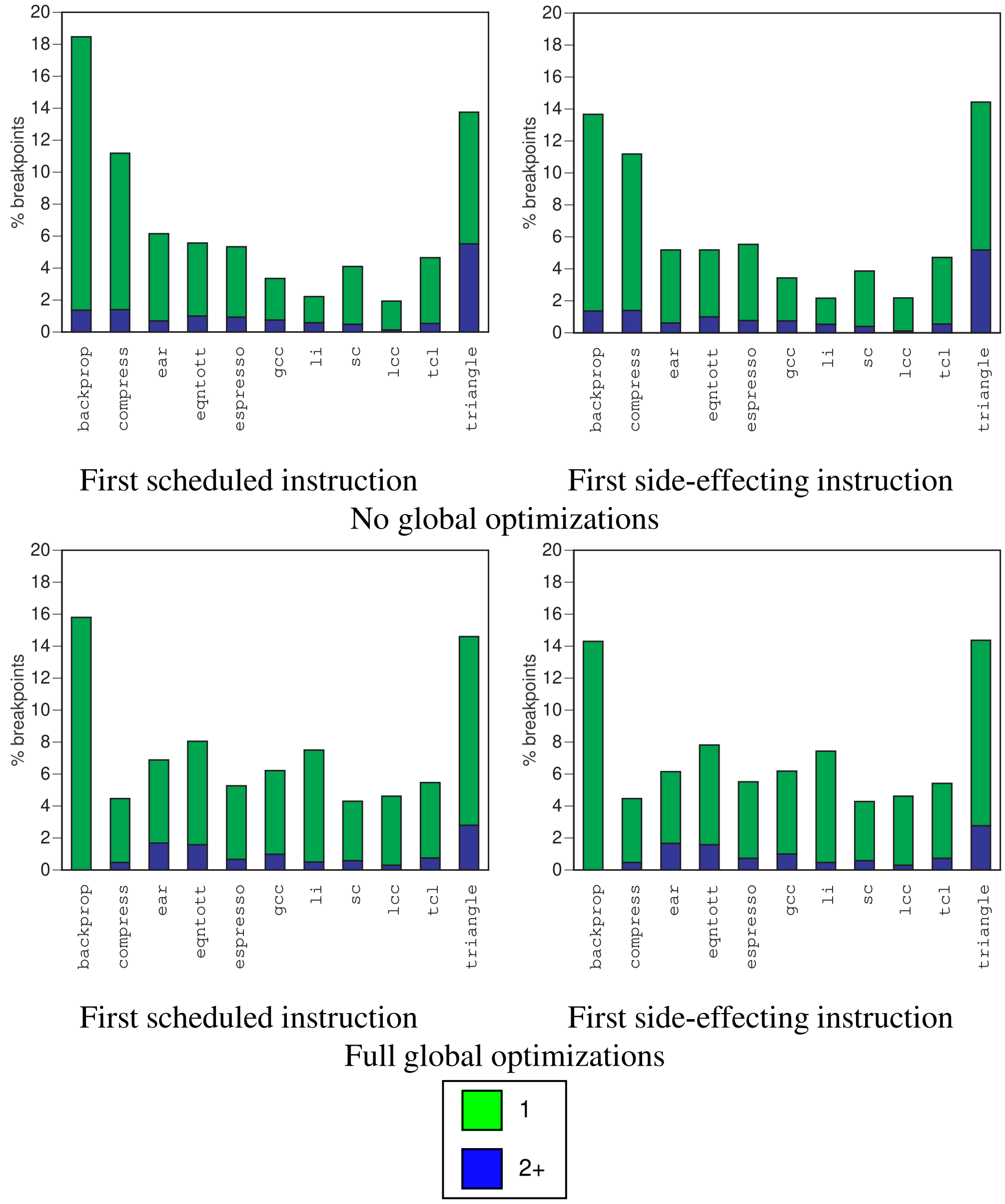


Figure 6.10: Percentage of breakpoints with 1, and 2 or more endangered variables.

the breakpoint mapping used after instruction scheduling may have very little effect on the number of endangered variables.

Chapter 7

Conclusion

In this dissertation, I have described necessary techniques to enable accurate source-level debugging of globally optimized code. I have shown how accurate source-level debugging of optimized code can be supported non-invasively for a wide range of scalar optimizations employed in state-of-the-art compilers. I have addressed both the code location and data-value problems in depth. For the code location problem, I have identified the problems that optimizations cause for setting and reporting breakpoints, and described techniques to handle these problems. I have presented the bookkeeping necessary in the compiler to maintain an accurate mapping between source and object programs. This mapping enables the debugger to set and report breakpoints precisely in the presence of optimizations.

I have presented a concise model for the data-value problem. This model precisely classifies the status of a queried variable according to how optimizations have affected the variable's value. This model identifies two new classes of variables: nonresident variables and suspect variables. I have also presented algorithms for classifying the status of a variable according to this model. I have described in detail the extensions necessary in the compiler to track the effects of optimizations and to generate the information needed by the debugger algorithms.

Finally, I have proven the practicality of the techniques presented in this dissertation by implementing them inside of `cmcc`, an optimizing C compiler that I have built in collaboration with others. This compiler performs an aggressive set of scalar optimizations, and generates code that is competitive with the code generated by the native MIPS `cc` and `gcc` compilers on the DECstation 5000/200. Using this implementation, I have quantified the effects of different optimizations on source-level debugging and identified the optimizations that are likely to cause the most problems for debugging.

7.1 Debugging in context: Advice to compiler writers

Based on my experience developing the `cmcc` compiler and implementing the algorithms for debugging of optimized code within this compiler, I offer the following advice to compiler writers:

- Elements in the intermediate representation should be easy to annotate with generic information. These elements include basic blocks, nodes within expressions, and instructions within schedules. Examples of annotations include comments, pointers to source-level statements, or flags marking that a node or instruction was inserted by a particular optimization phase. The idea of providing a flexible annotation mechanism is not new and has been used in other compilers (e.g., the SUIF compiler [105]). But annotations are crucial for debugging optimized code because they allow the compiler to keep a trail of how optimizations have transformed a program. Annotating the IR to provide a history of optimizations also helps in debugging the compiler.

- The intermediate representation should include support for nodes that are purely for bookkeeping purposes and are ignored by optimizations. There are several examples where such nodes are necessary for debugging of optimized code: statement labels, marker labels, and markers that indicate points from which code has been eliminated. Such nodes can also help in debugging the compiler.

- Source-level debugging should be considered from the very start when implementing optimizations. It is much simpler to implement the bookkeeping necessary for an optimization at the same time as the optimization is implemented.

- Frameworks provide a very powerful form of code reuse inside an optimizing compiler [5]. A framework captures the control structure of a class of computations but leaves specification of the exact functionality to the client of the framework. The best example of how frameworks can be used effectively inside a compiler is for data-flow analysis. The data-flow analysis framework within `cmcc` is reused 15 times, including the data-flow analyses for debugging optimized code.

7.2 Future work

There are several issues that are left open by this dissertation that should be addressed by future work on debugging of optimized code. One open issue is optimization coverage. This dissertation covers the large majority of standard optimizations found in modern production compilers, but does not address software pipelining and loop-nest transformations (e.g., interchange, skewing, etc.). Most production compilers are only beginning to include these optimizations. With the increase in available instruction-level parallelism, as well as the increasing gap in access time between different levels of the memory hierarchy, these optimizations are becoming more important and will sooner or later have to be addressed by a debugger for optimized code.

The general truthful approach of detecting and warning the user of endangered variables is likely the best approach to dealing with software pipelining. The general approach of Chapter 5 can be used to develop an algorithm for detecting endangered variables caused by software pipelining: first determine which assignments have been executed out of order at a breakpoint, and then detect endangered variables by determining the side effects of out-of-order assignments. Since software pipelining overlaps multiple loop iterations, we also need to discover variables that are endangered across loop iterations (*inter-iteration* endangered variables). That is, if execution halts inside a software-pipelined loop at some iteration index i, then a variable may be endangered because an assignment from a prior iteration (e.g., $i-1$) has not yet executed, or because an assignment from a subsequent iteration (e.g., $i+1$) has executed prematurely. Thus, typical responses involving endangered variables may be that several elements of an array A (e.g., elements $A[i-3]$ to $A[i-1]$) have not yet been updated, or that several elements (e.g., $A[i+1]$ to $A[i+3]$) have been prematurely assigned.

The best approach to handling loop transformations may be to expose these optimizations to the user by rewriting the source to reflect the effects of loop transformations (as discussed in Section 2.4.3). Loop transformations are typically performed on an almost abstract syntax tree-level representation of a program (e.g., a representation with explicit for loop constructs). In fact some loop transformation systems operate as source-to-source preprocessors. Since these transformations operation at such a high level, their results may still be meaningfully represented to the user.

Another open issue is the integration into a debugger of the techniques for debugging optimized code. Several engineering issues must be addressed in a production implementation. One such issue is the interface between the compiler and debugger. The information collected by the compiler must be passed to the debugger. Traditional object file formats are clearly not powerful enough to support this interface. The newer DWARF 2.0 debug format [40] supports the communication of residence information: the compiler can specify the storage location associated with a variable for a given range of instructions. DWARF, however, does not define a mechanism for communicating endangerment information to the debugger. One possible mechanism for communicating this information are the assignment descriptors described in Chapter 5.

A related engineering issue is the partitioning of the task of detecting nonresident and endangered variable between the compiler and debugger. Because the communication of residence information is defined by DWARF, residence determination should be done by the compiler using either live range information or the available resident data-flow analysis described in Chapter 4. The implementation of this data-flow analysis can take advantage of any data-flow analysis framework that may be available in the compiler. When the user queries the value of a variable V, the debugger uses the stopping instruction as an index into V's list of address ranges to find V's residence.

The data-flow analysis required for detecting endangered variables can also take advantage of available data-flow frameworks and be done in the compiler. The results of this data-flow analysis can be communicated to the debugger via the assignment descriptors. The debugger then detects endangered variables using the local analysis presented in Section 5.2.

An important practical concern in the context of a production debugger is testability. In the absence of optimizations, the testing of a standard source-level debugger can be automated easily. A given set of breakpoints and queries should always produce the same result regardless of changes to the compiler or debugger. Thus, testing can easily be automated using a script. Testing a source-level debugger for optimized code, however, is more difficult to automate because the results of a given set of breakpoints and queries can differ with changes to the compiler's optimization phases. If the results of a test are different after a change to the compiler or debugger, then human intervention is necessary to verify the results. Quality assurance for a debugger of optimized code may be difficult.

This dissertation addresses only the low-level problems that optimizations cause for debugging and leaves open user-interface issues. One important user interface issue is how to report a nonresident or endangered variable to the user. When a variable V is endangered, the debugger can provide additional information to the user describing which source statement(s) assigned the actual value of V. This information can be gathered by computing the reaching definitions of V in the object file or by performing a backward slice for V [103, 104].

The static measurements presented in this dissertation provide useful insight into the effects of different optimizations on debugging, but these measurements do not indicate how likely it is for a debugger user to encounter unexpected behavior because of optimizations. A necessary future work is a user study that measures how often users actually run into debugging problems because of optimizations. Such a user study can measure how often users query variables that are nonresident, suspect, or noncurrent, and how often users set control breakpoints at statement labels that have been moved because of code elimination.

In addition to measuring how often a debugger user is affected by optimizations, a user study can generate source-level debugging profiles that characterize typical debugging sessions. For example, to profile the locations of source breakpoint, the debugger can be instrumented to classify breakpoints according to whether they are at function entry points, loop entry points (i.e., loop preheader), loop exit points, initial loop statements, or any other type of location that may be interesting. To profile variable queries, one possibility is to measure how often variables are queried at breakpoints within their source live ranges. Another possibility, is to characterize variable queries in relation to the control reference statement. For example, the debugger can measure how often a variable is queried at a control reference statement that immediately follows an assignment to the variable, at a control reference statement that immediately precedes a use of the variable, at a control reference statement that immediately follows a last use of the variable, and so on.

User profiles can be used to extrapolate the effects of optimization on source-level debugging by biasing the static measurements presented in this thesis. User profiles can also be used to develop optimization heuristics that allow aggressive optimizations while minimizing the effects of optimizations on source-level debugging. For example, if measurements indicate that users are very likely to query the value of an induction variable at statements immediately following a loop, then the register allocator can extend the live ranges of induction variables by a few statements to minimize the chances that a user will run into a nonresident variable.

Another area of future work is minimizing the impact of optimizations on debugging by relaxing the invasiveness constraint. One of the main contributions of this dissertation is an in-depth insight into the effects of each optimization on source-level debugging. This insight can be gained only by looking at the most constrained case of non-invasive debugging. The motivation for non-invasiveness was the same as for debugging optimized code: eliminating perturbations that could change program behavior and maintaining the ability to debug the production (i.e., fully optimized) version of a program. There are cases, however, where some amount of invasiveness may be acceptable. For example, during interactive debugging, small amounts of perturbations to execution time may be acceptable, since invocation of the debugger and interaction with the user drastically perturbs execution time anyway. Or, in languages where unrestricted use of pointers is disallowed — for example, ML or Java — perturbation of the storage layout to enable debugging may be acceptable. In all cases, however, we would like to minimize perturbations since such perturbations are likely to affect performance. The insights we have gained from this research now allow us to understand how to minimize the effects of optimizations on source-level debugging while allowing only a minimal amount of invasiveness.

One way the invasiveness constraint can be relaxed is by modifying compiler optimizations to reduce the effect of optimizations on debugging. The interesting question is how much performance is sacrificed to gain debuggability. A good example of how this can help minimize the impact of optimizations is the elimination of nonresident variables. The results of Section 5.8 show that the most problematic optimization is potentially register allocation. One way to eliminate the problems caused by this optimization is to save a variable's value before the register assigned to the variable is re-used by the register allocator. The runtime cost of these saves can be minimized by using redundancy elimination techniques, or by scheduling the saves into empty slots in the final schedule; the cost of executing these additional saves may be minimal on a superscalar architecture. Another possibility is to extend an integrated register allocation and instruction scheduling framework to also handle saves at the end of live ranges.

Another example is the elimination of endangered variables. Optimizations that cause endangered variables can be constrained so that problems do not occur. For example, partial redundancy elimination (i.e., code hoisting) can be constrained so that assignments to source variables are not moved; since assignments are very rarely moved by this optimization anyway, this constraint is likely to have only a minimal impact on performance. Instruction scheduling can be constrained such that only the instructions within a statement are reordered. Or, scheduling can be constrained such that side-effecting instructions such as assignments, function calls, and branches are executed in their original source order. Dead code elimination can be constrained such that it eliminates an assignment to a variable only when the assigned value can be reconstructed from other runtime values; for example, an induction variable is eliminated only when its value can be recovered from a compiler-synthesized induction variable.

Another way in which invasiveness can be relaxed is by allowing the debugger to collect runtime information at key program points during interactive debugging. This can be implemented with "hidden breakpoints" that transparently suspend and resume execution. This technique can be used to eliminate nonresident variables: the debugger saves away variable values at the end of their live ranges (or before they become nonresident). To minimize the impact on execution time, this can be done for a subset of variables specified by the user or for all variables that are in scope at the source breakpoints set by the user. By allowing invasiveness only during interactive debugging, performance is affected only during interactive debugging rather than on production runs of a program. Of course, these techniques do not help post-mortem debugging.

By allowing invasiveness during interactive debugging, the debugger can also eliminate the impact of instruction scheduling on source-level debugging. To eliminate both the code location and data-value problems caused by instruction scheduling, when the user sets a breakpoint inside a basic block B, the debugger can halt execution at the first instruction of

B and then interpret the instructions inside B one at a time in original source order until all instructions from statements prior to the breakpoint statement have been executed. In effect, the debugger re-orders instructions back to their original source order before a breakpoint is taken. This technique is, of course, subject to constraints. For example, function calls should probably be executed rather than interpreted. And the debugger must analyze the dependences induced by the reuse of registers to maintain correct semantics.

Finally, optimizations also affect other components of the program development tool chain; the techniques developed in this dissertation may be applicable to these other components. For example, profile-based performance analysis tools must understand the correspondence between the source and object codes to convey profiling results to the user. This dissertation presents the bookkeeping necessary inside the compiler to map instructions and runtime locations accurately back to source statements and variables, respectively. These mappings are also useful for performance analysis tools.

7.3 Concluding remarks

Source-level debuggers are valuable program development tools that help a user locate the source of a programming error and analyze the dynamic behavior of a program. When compiler optimizations have been performed, current compiler tool chains either preclude source-level debugging, or allow debugging without considering the effect of optimizations on the execution behavior expected by the user — in the later case, the responses from the debugger are often inaccurate and misleading.

Accurate source-level debugging of optimized code is important because there are many instances where debugging the optimized translation of a program is necessary. For example, bugs may surface when optimizations are enabled, even when optimizations are correct; debugging the unoptimized translation will not locate the source of the bug. Or, it may be impossible to re-execute a crashed program under the control of a source-level debugger (e.g., the shipped optimized version of a program may crash in the field allowing only post-mortem debugging); in this case, debugging the unoptimized translation is impossible.

Most importantly, debugging of optimized code is important because optimizations are becoming “default”. Designers of modern computer systems are relying more and more on compiler optimizations to deliver higher performance in new processor generations. It has become very expensive — both in development costs and processor cycle time — to rely purely on hardware techniques to achieve higher performance. Thus processor designers are investigating techniques such as predicated execution, VLIW, and software-controlled memory hierarchies that require extensive compiler support and rely on the compiler's ability to analyze globally the source program. As the reliance on compiler optimizations increases so will the impetus for providing some level of compiler optimization by default; otherwise, there will be no immediate gain from adopting a new processor generation. This in turn will drive the demand from application writers for better and more accurate debugging support. Even if optimizations are not performed by default, optimizations should not preclude accurate source-level debugging. Without debugging support, application writers are less likely to enable optimizations; thus, systems are less likely to perform close to their full potential.

There is a common belief among compiler and debugger implementors that debugging of optimized code is a difficult and infeasible task. No commercially available program development tool chain to date supports accurate and *truthful* debugging of optimized code (as described in this dissertation). CXdb, the only commercial debugger that provides support for optimization, unnecessarily burdens the user by exposing all optimizations and by leaving it to the user to figure out the effects of optimizations. For the typical application writer this burden is unacceptable.

This dissertation establishes that accurate source-level debugging of optimized code is feasible. This dissertation shows that the debugger can go far in analyzing the effects of optimizations on source-level debugging. The measurements that I have presented show that in many cases the debugger can make optimizations transparent. Using the techniques developed in this dissertation it is finally practical to implement a source-level debugger for globally optimized code. No longer is it necessary to support a "debug-mode" for the compiler; it is possible to debug the best code produced by the compiler.

Bibliography

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

[2] A. Adl-Tabatabai and T. Gross. Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 371–383. ACM, January 1993.

[3] A. Adl-Tabatabai and T. Gross. Symbolic Debugging of Globally Optimized Code: Data Value Problems and Their Solutions. Technical Report CMU-CS-94-105, CMU, January 1994.

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

[5] A. Adl-Tabatabai, T. Gross, and G.Y. Lueh. Code Reuse in an Optimizing Compiler. In *OOPSLA '96 Conference Proceedings*. ACM, October 1996.

[6] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136. ACM, May 1996.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[8] A.W. Appel. Continuation-Passing, Closure-Passing Style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302. ACM, October 1989.

[9] A.W. Appel and D.B. MacQueen. A Standard ML Compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 301–324, Portland, OR, September 1987. ACM, Springer Verlag.

[10] M. Auslander and M. Hopkins. An Overview of the PL8 Compiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*. ACM, June 1982.

[11] T. Ball and J.R. Larus. Branch Prediction For Free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313. ACM, June 1993.

[12] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic Program Parallelization. *Proc. IEEE*, 81(2):211–243, Feb 1993.

[13] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263. ACM, July 1989.

[14] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255. ACM, June 1991.

[15] D.S. Blickstein, P.W. Craig, C.S. Davidson, R.N. Faiman, K.D. Glossop, R.B. Grove, S.O. Hobbs, and W.B. Noyce. The GEM Optimizing Compiler System. *Digital Technical Journal*, 4(4), 1992.

[16] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proc. Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[17] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[18] P. Briggs and K. Cooper. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170. ACM, June 1994.

[19] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284. ACM, July 1989.

[20] G. Brooks, G. Hansen, and S. Simmons. A New Approach to Debugging Optimized Code. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 1–11. ACM, June 1992.

[21] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65. ACM, June 1990.

[22] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, Santa Clara, April 1991. ACM.

[23] D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, June 1991. ACM.

[24] B. Case. Intel Reveals Pentium Implementation Details. *Microprocessor Report*, pages 9–17, March 1993.

[25] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982. In SIGPLAN Notices, v. 17, n. 6.

[26] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation by Coloring. Research Report 8395, IBM Watson Research Center, 1981.

[27] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, A Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pages 49–70. ACM, October 1989.

[28] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275. ACM/IEEE, May 1991.

[29] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.

[30] F. Chow. *A Portable, Machine-Independent Global Optimizer — Design and Measurements*. PhD thesis, Stanford University, 1984.

[31] F. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94. ACM, June 1988.

[32] F. C. Chow and J. L. Hennessy. A Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12:501–535, Oct. 1990.

[33] R. Cohn. *Source-level Debugging of Automatically Parallelized Programs*. PhD thesis, School of Computer Science, Carnegie Mellon, Oct 1992.

[34] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Trans. on Computers*, 37(8):967–979, August 1988.

[35] M. Copperman. Debugging Optimized Code Without Being Misled. Technical Report UCSC-CRL-93-21, UC Santa Cruz, June 1993.

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

[37] M. Copperman and J. Thomas. Poor Man's Watchpoints. Technical Report UCSC-CRL-93-12, UC Santa Cruz, March 1993.

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

[39] D. M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[40] DWARF Debugging Information Format. Industry review draft, UNIX International, Programming Languages SIG, Parsippany, NJ, July 1993.

[41] D. Ebcioglu, R. Groves, K. Kim, G. Silberman, and I. Ziv. VLIW Compilation Techniques in a Superscalar Environment. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 36–48. ACM, June 1994.

[42] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Direction From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 85–97. ACM, October 1992.

[43] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers*, C-30(7):478–490, July 1981.

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

[45] S. Freudenberger and J. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 146–170. Springer Verlag, 1992.

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

[47] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, pages 9–15, February 1995.

[48] L. Gwennap. Nx686 Goes Toe-to-Toe with Pentium Pro. *Microprocessor Report*, pages 1–10, October 1995.

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

[50] J. L Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman, 1990.

[51] J.L. Hennessy and T.R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422 –448, July 1983.

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

[53] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1,2):229–248, March 1993.

[54] W.W. Hwu and P.P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 242–251. ACM and IEEE Computer Society, June 1989.

[55] Intel. *Optimizations for Intel's 32-Bit Processors*. Application Note AP-500, Intel Corp., February 1994.

[56] K. Johnson. RISC-like Design Fares Well for x86 CPUs. *Microprocessor Report*, pages 26–27, November 1995.

[57] M. S. Johnson and T. C. Miller. Effectiveness of a Machine-Level Global Optimizer. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 99–108. ACM, July 1986.

[58] R.E. Johnson, J.O. Graver, and L.W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*, pages 18–26. ACM, September 1988.

[59] G. Kane and J Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[60] D. Keppel. Fast Data Breakpoints. Technical Report UWCSE 93-04-06, University of Washington, April 1993.

[61] P. Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 78–84. ACM, June 1990.

[62] J. Knoop, O. Ruthing, and B. Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234. ACM, June 1992.

[63] J. Knoop, O. Ruthing, and B. Steffen. Lazy Strength Reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

[64] J. Knoop, O. Ruthing, and B. Steffen. Optimal Code Motion: Theory and Practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[65] J. Knoop, O. Ruthing, and B. Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158. ACM, June 1994.

[66] P. Kolte and M. J. Harrold. Load/Store Range Analysis for Global Register Allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 268–277. ACM, June 1993.

[67] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[68] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74, Santa Clara, April 1991. ACM.

[69] J.R. Larus and P.N. Hilfinger. Register Allocation in the SPUR Lisp Compiler. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 255–263. ACM, June 1986.

[70] P. Lee and M. Leone. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148. ACM, May 1996.

[71] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1,2):51–142, March 1993.

[72] G.Y. Lueh, T. Gross, and A. Adl-Tabatabai. Global Register Allocation Based on Graph Fusion. CMU-CS 96-106, School of Computer Science, Carnegie Mellon University, 1996.

[73] E. C. Lyle. Debugging VLIW Code After Instruction Scheduling. Master's thesis, Oregon Graduate Institute, July 1992.

[74] S.A. Mahlke, W.Y. Chen, W.W. Hwu, B.R. Rau, and M.S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 238–247, Boston,MA, October 1992. ACM.

[75] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA., 1990.

[76] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[77] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, October 1992. ACM/IEEE.

[78] Masood Namjoo and Anant Agrawal. Implementing SPARC: A High-Performance 32-Bit RISC Microprocessor. *SunTechnology*, Winter, 1988.

[79] C. Norris and L. L. Pollock. Register Allocation over the Program Dependence Graph. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 266–277. ACM, June 1994.

[80] K. O'Brien, K.M. O'Brien, M. Hopkins, A. Shepherd, and R. Unrau. XIL and YIL: The Intermediate Languages of TOBEY. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 71–82, San Francisco, Jan 1995. ACM.

[81] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison–Wesley, 1994.

[82] K. Pettis and R.C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, June 1990.

[83] L.L. Pollock and M.L. Soffa. High-Level Debugging With the Aid of an Incremental Optimizer. *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, 1988.

[84] V. Santhanam and D. Odnert. Register Allocation Across Procedure and Module Boundaries. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39. ACM, June 1990.

[85] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

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

[87] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *First Workshop on Applied Computational Geometry*. ACM, May 1996.

[88] M. Slater. AMD's K5 Designed to Outrun Pentium. *Microprocessor Report*, pages 1–11, October 1994.

[89] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 248–259. ACM, Oct 1992.

[90] D. Steere. Personal communication. 1995.

[91] L.V. Streepy et al. CXdb A New View On Optimization. In *Proc. Supercomputer Debugging Workshop '91*, Albuquerque, NM, November 1991. Los Alamos National Laboratory.

[92] R. Title. Thinking Machines Vendor Update. In *Proc. Supercomputer Debugging Workshop '92*, pages 63–82, Dallas, TX, October 1992. Los Alamos National Laboratory.

[93] A. P. Tolmach and A. Appel. Debugging Standard ML Without Reverse Engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 1–12. ACM, June 1990.

[94] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241. ACM, October 1987.

[95] J. Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems. *SPEC Newsletter*, 1(1), Fall 1989.

[96] R. Wahbe. Efficient Data Breakpoints. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 200–212, Boston, October 1992. ACM/IEEE.

[97] R. Wahbe, S. Lucco, and S. Graham. Practical Data Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 1–12. ACM, June 1993.

[98] D. Wall. Predicting Program Behavior using Real or Estimated Profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Compiler Construction*, pages 59–70. ACM, June 1991.

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

[100] D. W. Wall. Global Register Allocation at Link Time. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, Palo Alto, June 1986. ACM.

[101] H. Warren. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM J. Research and Development*, 34(1):85–92, Jan 1990.

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

[103] M. Weiser. Programmers Use Slices When Debugging. *Communications of the ACM*, 25(7), July 1982.

[104] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[105] R. Wilon, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.

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

[107] R. Wismueller. *Quellsprachorientiertes Debugging von optimierten Programmen*. PhD thesis, Technische Universitaet Muenchen, Munich, Germany, Dec. 1994. (in German). Published (1995) by Shaker Verlag, Aachen (Germany), ISBN 3-8265-0841-6.

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

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

[110] S. Zimmerman. UDB: A Parallel Debugger for the KSR1. In *Proc. Supercomputer Debugging Workshop '92*, pages 95–102, Dallas, TX, October 1992. Los Alamos National Laboratory.

[111] L. Zurawski and R. Johnson. Debugging Optimized Code With Expected Behavior. Unpublished draft from Department of Computer Science, University of Illinois at Urbana-Champaign, April 1991.