Source-Level Debugging of Globally Optimized Code

Ali-Reza Adl-Tabatabai

April 9, 1996

CMU-CS-96-XXX

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

Contents

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 discrepencies between the specification documentation and the actual behavior of a program. Debuggers are thus an important analysis tools and should not replace or discourage the use of good programming methodology.

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 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 [98, 49, 83], 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, 66, 14, 97] 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 [94, 11, 42], the compiler can reorganize object code to increase the instruction fetch efficiency of pipelined or superscalar processors [54, 79]. Transformations such as blocking [67] and prefetching [22, 76] can improve a program's data cache locality, thus reducing stalls due to cache misses [76].

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, 78]. 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 [84] 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 advices 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 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 user 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 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 [82] at CMU do not enable optimizations during compilation, so that the system can be debugged in the field when necessary [86]. 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 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 and 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 [87, 88, 103] — 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. 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.

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

[1] Systems that generated code dynamically (e.g., [69] or [6]) are even more challenging to decipher.

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 have yet 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. There is no way that the debugger can provide the illusion that the statements are executing in the order prescribed by the source.

Instruction scheduling and register allocation are standard optimizations that are employed in almost all state-of-the-art compilers; the preceeding 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 thesis shows that accurate source-level debugging of fully optimized code is viable in practice, 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 thesis 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 [89, 46], or if compiler optimizations are constrained so that problems do not occur [104, 52]. But such modifications or limitations no longer qualify a program as optimized and defeat the goal of debugging *optimized* code. This thesis assumes 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 thesis, 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 source program. Most of this thesis is concerned with how the debugger can detect the effects of optimizations on source-level debugging, and how the debugger can convey the effects of optimizations to the user.

This research is oriented towards the debugging of imperative programming languages and considers the C programming language as the source language of the debuggee. This thesis does 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 industry and research standard compilers for imperative languages [7, 30, 70, 10]. The compiler first translates a source program into a suitable 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). This thesis considers the effects of transformations performed by both the machine-independent optimization and code generation phases of the compiler. This thesis does 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. This thesis does not investigate debugging of parallelized code (debugging of parellelized code is the subject of Cohn's thesis [33]).

In this thesis, 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 thesis 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 thesis in the context of the `cmcc` compiler. `cmcc` is a retargetable optimizing C compiler implemented as part of this thesis. This compiler is based on the compiler model outlined above and generates code that is roughly comparible 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 thesis

Chapter 2 presents background material on source-level debugging and describes in more detail the problems caused by compiler optimizations. Prior work in debugging of optimized code and the relationship of this thesis to prior work are also described.

Chapter 3 presents the experimental framework used in this thesis. The `cmcc` compiler and the suite of programs used for the measurements presented in later chapters are described. The quantitative methodology used to gather the measurements in this thesis is also discussed.

Chapters 4 and 5 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. Measurements of how often a user may be affected by optimizations when querying a variable's value are also presented in these chapters.

Chapter 6 describes in detail the problems relating to setting and reporting breakpoints in optimized code. The chapter presents 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, Chapter 7 summarizes the main contributions of this thesis.

Chapter 2

Background

In this chapter, I present background material on source-level debugging of optimized. In Section 2.1, I describe a debugger model that is used as a basis in the rest of this thesis. I concentrate on core debugger functionality and how optimizations affect this functionality. I discuss neither the design of the debugger user interface, nor the integration of a debugger into an editor; these are important in an implementation of a production debugger but are beyond the scope of this thesis. 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 why it is important to minimize perturbations to the debuggee during debugging. In Section 2.4, I discuss prior work on debugging optimized code and how this thesis relates to past approaches. In Section 2.5, I describe how source-level values can be managed by a debugger for optimized code. In Section 2.6, I introduce terminology that will be used in the rest of this thesis.

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 apects to the functionality of a source-level debugger:

1. The debugger must allow the user to manipulate the execution of the debuggee by providing functions for suspending and resuming debuggee execution. *User interrupts* allow the user to asynchronously suspend the execution of an executing program (e.g., by typing `^ 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 or software, or a combination. At this time, the efficient implementation of data breakpoints is a topic of active research [60, 92, 93, 59, 37] and beyond the scope of this thesis. A more general form of breakpoint, *conditional breakpoints*, suspends execution when a user specified predicate on the program's state become 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 [60] and beyond the scope of this thesis.

2. The debugger must convey 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 must convey 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 occured. 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 thesis 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* supend 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* supend 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 asynchronously suspend execution. 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; or 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.[1] 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 thesis.

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 control reference 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.

[1]An example is a VLIW machine that records the effects of integer operations before it is known if a memory access results in a page fault.

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. Current compiler tool chains allow debugging by producing 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 [102].

- 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, 5]. These problems are called *data-value* problems [102].

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 thesis 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 object code in this example is that of a hypothetical load–store architecture; destination operands are listed first. 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.[2] 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

[2] 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, resulting in these two variables having disjoint live ranges.

Source code (a)	Object code (b)
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`

Figure 2.1: Example

I_3, and register `R4` (the register assigned to `p`) is assigned a value of `b` at instruction I_4.

To hide the latency due to memory load and floating point 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 markers that seperate 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, and 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 — 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 compiler–debugger tool chains. 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 aim of 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. New ways of specifying breakpoints are necessary that capture the user's intent. New ways of conveying the control state and qualifying variable values are necessary that clearly convey the program state to the user.

2.2.2 Values in optimized code

Most research on debugging optimized code has concentrated on the data-value problem [49, 2, 1, 36, 99] and much terminology has been developed for this aspect of the problem. 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.

When a program is translated, each source variable V is assigned a runtime storage location (e.g., a register or 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, and 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].[3] 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 that residency depends only on the stopping instruction and is independent of the control reference statement.

[3]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).

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 already executed, but the value of `p` is nonresident, so we cannot determine which memory location has been modified. In the absence of other 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 colums 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 why it is important to limit or prohibit 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 [60] — 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 [93] to implement data breakpoints), or by the debugger when the program is being debugged (e.g., to implement control breakpoints [60]). Such knowledge can assist a debugger for optimized code in performing analysis when retrieving source values. 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 thesis 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-ordering 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* [101] 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, we describe approaches that have been proposed or implemented in the past and relate these prior approaches to the one presented in this thesis. 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 completely hide optimizations from the user, and to provide the exact functionality and behavior as expected from unoptimized code. In general, this approach cannot be applied without limiting compiler optimizations, or the usability of the debugger. 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 thesis — exposes optimizations to the user, but 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

Some debuggers claim to 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 can mislead the user.

2.4.2 Providing expected behavior

Ideally, we would like the debugger to completely 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 [101] 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, 104, 46, 89]. 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 is limited and optimizations are constrained so that debugger functionality can be provided at these points [80, 52, 104, 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 single statements, thus allowing full debugging with some amount of 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 [89]. 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 [80, 52, 104].

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

Pollock and Soffa [80] 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 [90]. 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 [104] 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 recompilation).

Gupta [46] proposes a technique for debugging code transformed by trace scheduling [43, 41] compilers. 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 recompilation 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 [89] 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 [74]. 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.

- Increment recompilation 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 recompilation 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.

Lyle's masters thesis [72] 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 and meaningfully 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 (actual) 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[4] — 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 analyses are 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, 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 pipelining where the execution of multiple loop iterations are overlapped; highlighting alone cannot convey the iteration index of an operation that is being executed.

[4]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].

2.4.4 Managing the effects of optimizations

Even though hiding the effects of optimizations is generally not possible, 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 analyses to determine the effects of optimizations on source-level debugging.

Most of the prior work on managing the effects of optimizations has focused on managing 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 [101] 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, 99, 5]. 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 at the intermediate-representation level. Therefore, the algorithms Hennessy describes in [49] (and corrected in [95]) 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 reasonably handle 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 predetermined 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 apriori the points at which execution will be halted in both the source and object codes: the compiler can precompute 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 [100, 99] 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 variablers [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 [100, 99] 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 seperate from the IR used by the compiler. This specialized IR must be mainained 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 work of Copperman and Wismueller are similar the some of the algorithms I present in this thesis, to the extent that they also use data-flow analysis. There are, however, two key differences between my work and theirs 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 code; 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 these invariants. For example, if code is hoisted to a different basic block, the basic block is post-dominated by the original block; this limits the range of breaks where a variable may be endangered. Or, if an expression is eliminated due to redundancy, the value must be available somewhere, and the debugger can provide this value to the user. Both Copperman [36] and Wismueller [99] assume arbitrary code movement and elimination, and fail to recognize that movement and elimination are not unconstrained.

Another major difference between this thesis, and the earlier work by Copperman and Wismueller is that they attempt to capture a "summary effect" of all optimizations in an auxillary intermediate representation, and then attempt to relate the optimized code back to the source code. They treat the compiler as a "black box" that takes a source program as input and produces object code with the auxillary intermediate representation that captures the effects of optimizations. In contrast, the approach I describe in this thesis models each optimization step inside the compiler. Only the single representation used by the optimizer is maintained and the compiler propagates information about the effect of optimization steps through all optimization phases. 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 a source-level value, 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 too 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 too 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 [101]) or the architecture (e.g., memory tags), detecting uninitialized variables requires that the debugger performs 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 to not only show the actual value of a variable, but also provide additional information describing how the actual value relates to values in the source. The debugger can conservatively report 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 not only display 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 to either multiple paths reaching a breakpoint [3], or pointer assignment 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.

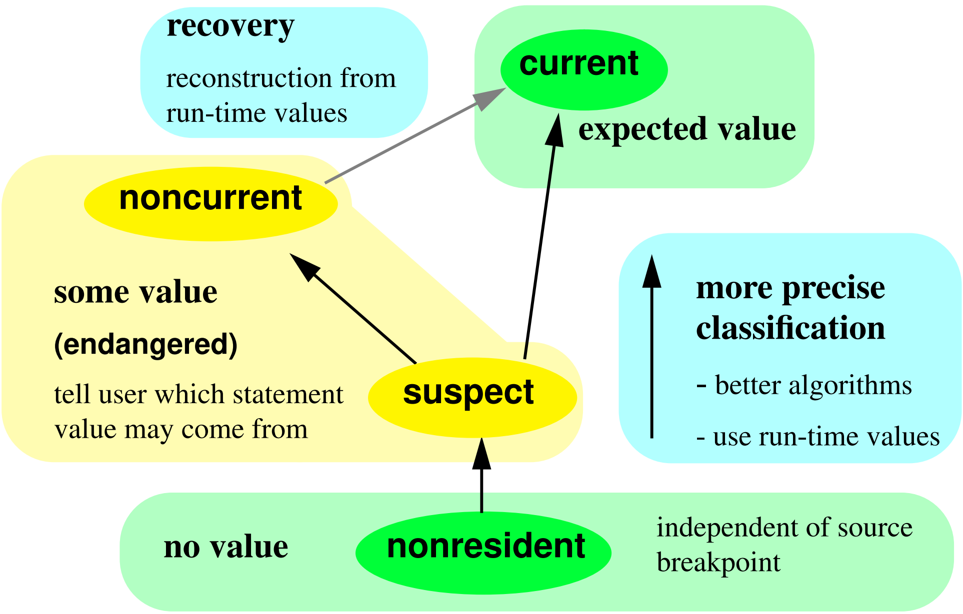


Figure 2.2: The data-value problem.

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 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. 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 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 our 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. We define the set of predecessor instructions of I, denoted $pred(I)$, as the set of instructions from which control can immediately reach I. The basic block containing the instruction I is denoted $Block(I)$. 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. An instruction that targets a register R is a *definition* of R. The set of definitions of a register R that reach a point O is the set *ReachingDefs*$(R, O) = \{I : I$ reaches O and I is a definition of $R\}$.

2.7 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, CMU's optimizing C compiler. I have implemented the algorithms that I describe in this thesis, in the context of this compiler. I also describe the methodology used to gather the numbers in this thesis 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, SPARC, DLX, and iWarp architectures. I obtained the results that I report in this thesis 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 in preparation for code motion transformations. After these control flow transformations, the

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`

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 in [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 in [63]. Strength reduction is integrated with partial redundancy elimination using the algorithm described in [62]. 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 in [64]. 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 [71]) 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 in [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 operation scheduling. To allow maximum scheduling freedom, the register allocator is run after the instruction scheduler.

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

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 instructions is kept machine independent and is encapsulated inside a C++ abstract class; the register allocator and 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 inheritence: there are machine-specific instructions for each target machine which 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, while keeping 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 window versus no register windows, and seperate 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 assigmment 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 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

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.

framework can be found in [4].

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 assignment decisions in the register allocator [71].

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 thesis can be reasonably generalized to other optimizing compilers.

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

3.2 Quantitative methodology

In my approach to debugging optimized code, the debugger may sometimes 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 to result in a response where the debugger cannot display an expected value. To answer such questions — regarding how often optimizations will affect a users 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 program development tool chain 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 thesis.

An approximation to the above 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 thesis show the average number of variables that are uninitialized, current, endangered, and nonresident at a breakpoint. These numbers are collected by counting the

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

Table 3.3: Programs used in this study

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. Static measurements are no indication of how often a debugger will be able to respond with an expected value during an actual debugging session, but such measurements may help identify general trends and 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 thesis 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 numbers provide additional insight into how different optimizations affect debugging.

The empirical evaluation presented in this thesis is based on the set of eight C programs from SPEC92. The SPEC programs are a suite of benchmarks created for comparative measurements of vendor system performance [91]. These programs are supposed to represent a variety of "typical" system workloads and have been used widely for reporting speedups from optimizing various system components, including the compiler. Although this thesis 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.

It is difficult to say what a "typical" program that is being debugged under a source-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 thesis are more likely to be affected by application type and programming style than by whether a program is likely to require a source-level debugger. Therefore, even though the SPEC programs 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 these programs and statistics relevant to source-level debugging. 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 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 thesis 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 empiricial evaluation. The measurements I present in this thesis 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 progam 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. Global register allocation also complicates the mapping from variables to runtime storage location 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 describe how the debugger can attempt to improve the responses given to the user by detecting uninitialized variables. In Section 4.4, I present the data-flow analysis approach. 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 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 ranges; 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* [71] 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 verices 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 refered 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 which 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, 68].

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, 65] 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, 77, 71] 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 effecting 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 instruction 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, 40].

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 [96, 21, 81]. 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 variables, 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 4.4 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.

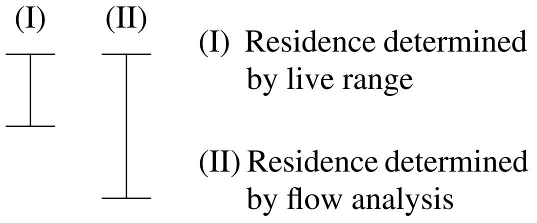
- When variables are spilled, the register allocator can minimize the size of the activation record by sharing spill locations. Therefore, spilled values can be nonresident, and the debugger must perform analysis even on spilled variables.

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) (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 4.4). 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 4.4, 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 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 4.3. 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 4.3: Example: Uninitialized variables.

a program.

4.4 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.

4.4.1 Terminology

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

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

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 $Residences(V)$ 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.)

4.4.2 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 4.4.4 solves for $Resident(V, L, I)$ by computing $AvailRes(\langle V, L \rangle, pre(I))$.

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 4.4: Example: Multiple available residences.

4.4.3 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 4.4. 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 $\langle \mathtt{x}, \mathtt{R1}\rangle$, while I_3 defines residence $\langle \mathtt{x}, \mathtt{R2}\rangle$. Both residences are available at instruction I_4 (assuming no definitions of `R1` between I_1 and I_3).

Since we are interested in only 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 4.4, because of source definition I_3. Residence $\langle \mathtt{x}, \mathtt{R2}\rangle$, however, still reaches I_4.

4.4.4 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.

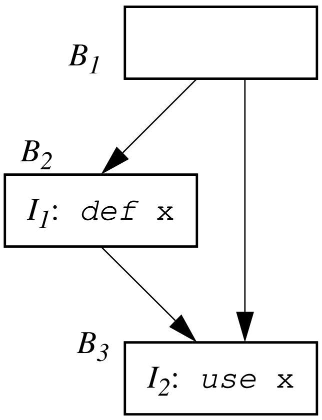


Figure 4.5: Example: Live but nonresident variable.

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 4.4.3, however, all such locations hold the same value of V.

4.4.5 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 situations where the available residence analysis is *more* conservative than using live range information. Consider the example shown in Figure 4.5. 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_3. The available residence analysis will detect x as resident after I_1 in block B_2, but nonresident inside block B_3, even though B_3 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 4.5, x is nonresident at the beginning of block B_3, 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 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 4.5, if execution reaches a break at B_3, without going through B_2, x is resident but its runtime value is undefined. Figure 4.6 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 a "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. Figure 4.7 shows a real example where the user can determine a variable's residence better than the debugger can.

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

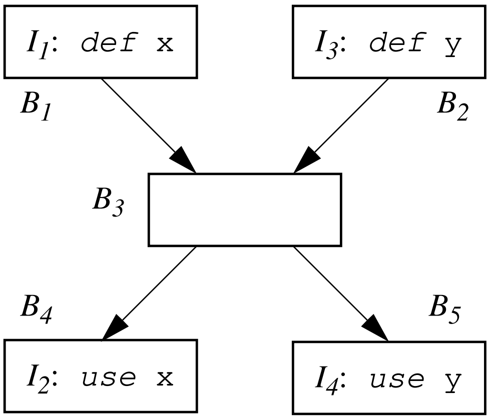


Figure 4.6: Example: Live but nonresident variables that overlap.

```
/* 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 has 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 4.6 could potentially happen.

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 conjuction 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 4.8 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 4.8(a)), between 30–60% of all possible breakpoints contain nonresident variables. A few programs, such as `ear`, `espresso`, `gcc`, and `sc`, have a significant number of breakpoints with four or more variables that are nonresident. Figure 4.8(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 4.8(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 4.8(d) shows the results when assignment sinking is performed along with global optimizations and instruction scheduling. For `espresso` and `gcc`, 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 4.9 show the average number of variables that are nonresident at each breakpoint. 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. Figure 4.10 shows a different view of the same data; this figure directly compares the average number of nonresident variables using different optimizations. The numbers in Figures 4.9 and 4.10 support the results of the preceeding paragraphs: scheduling has a minor effect on the average number of nonresident variables (Figure 4.9(b)); global optimizations almost double the number of nonresident variables at each breakpoint (Figure 4.9(c)); and assignment sinking further increases the number of nonresident variables (Figure 4.9(d)).

The charts in Figure 4.9 also show that, on average, between 12–22% of the variables are detected as uninitialized at a break, using reaching analysis. Without detecting uninitialized variables, these variables would be detected as nonresident by the debugger (note, that these

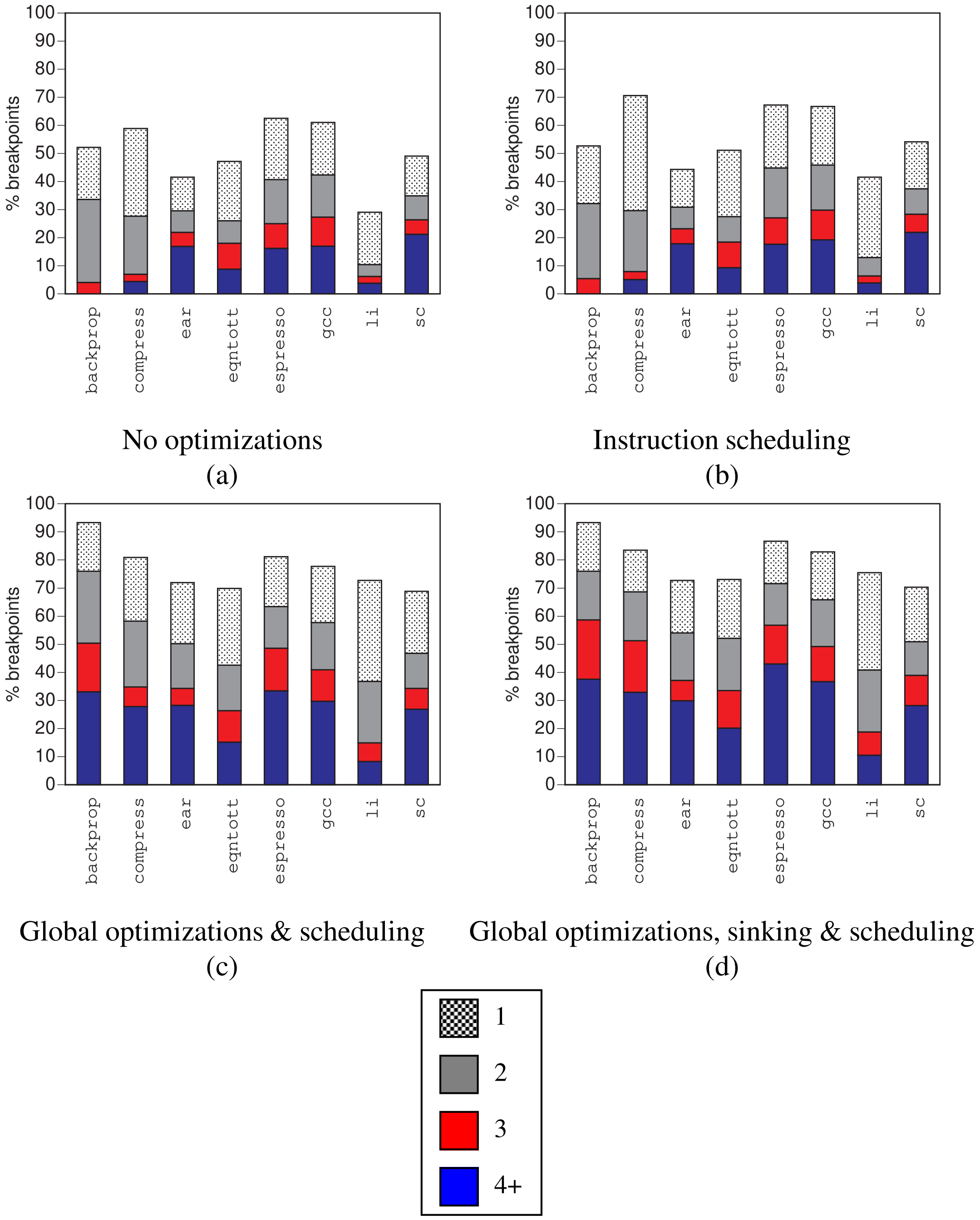


Figure 4.8: Percentage of breakpoints with 1, 2, 3, and 4 or more nonresident variables using available residence data-flow analysis.

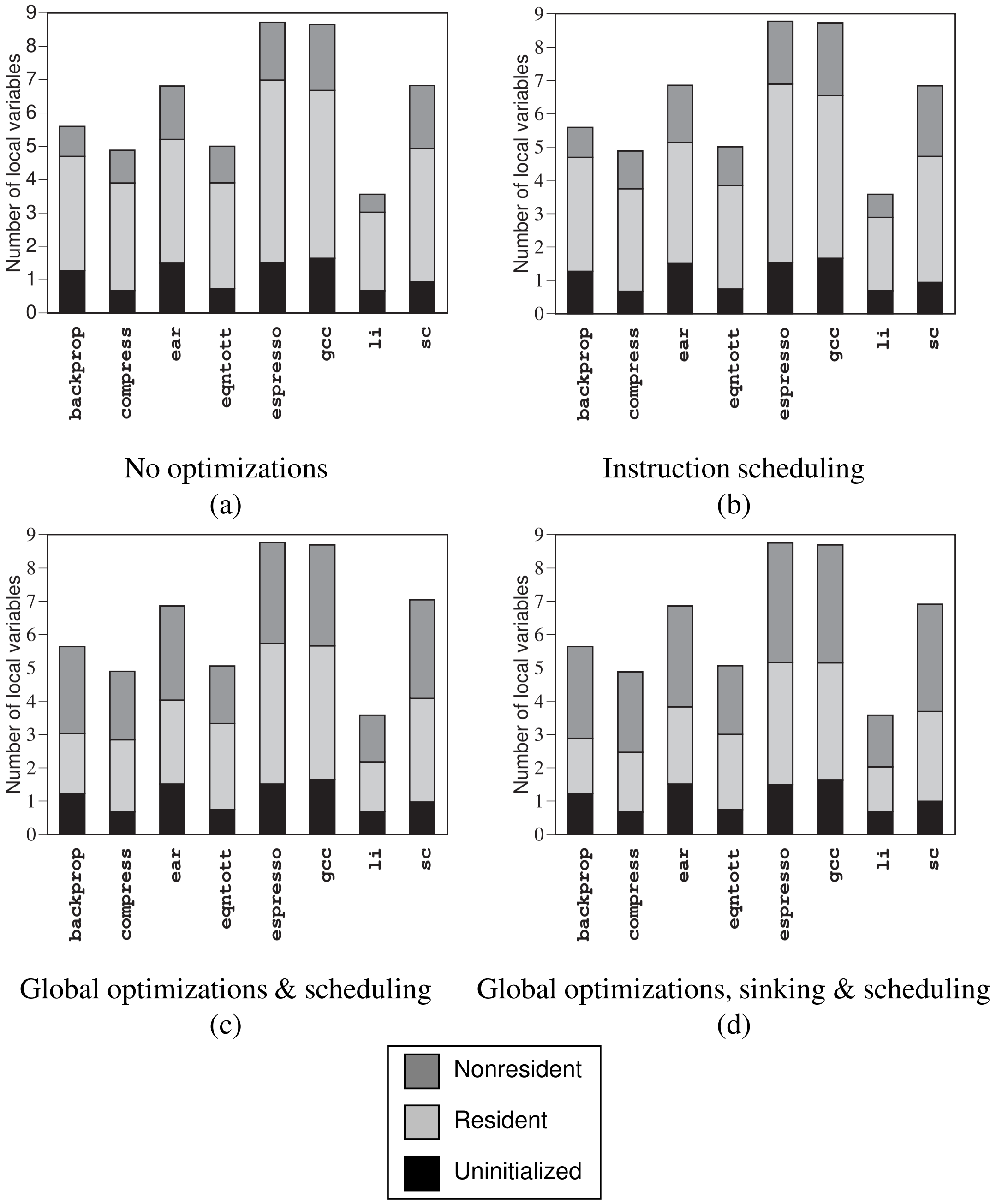


Figure 4.9: Average number of nonresident variables using available residence data-flow analysis.

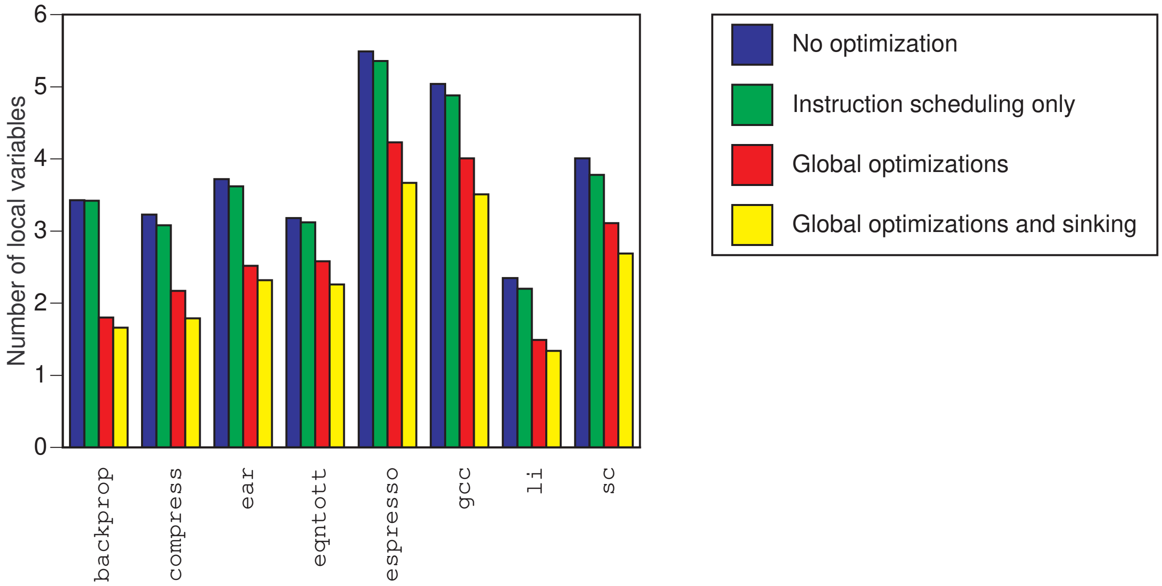


Figure 4.10: Comparison of average number of resident variables at a breakpoint, using available residence data-flow analysis.

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 4.11 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 4.12 shows the average number of breakpoints that a variable is resident using available residence analysis. For a few programs (`compress`, `backprop`, and `sc`) the decrease caused by global optimizations are more dramatic that those in Figure 4.11. 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.

The charts in Figure 4.13 measure the effectiveness of using the available residence data-

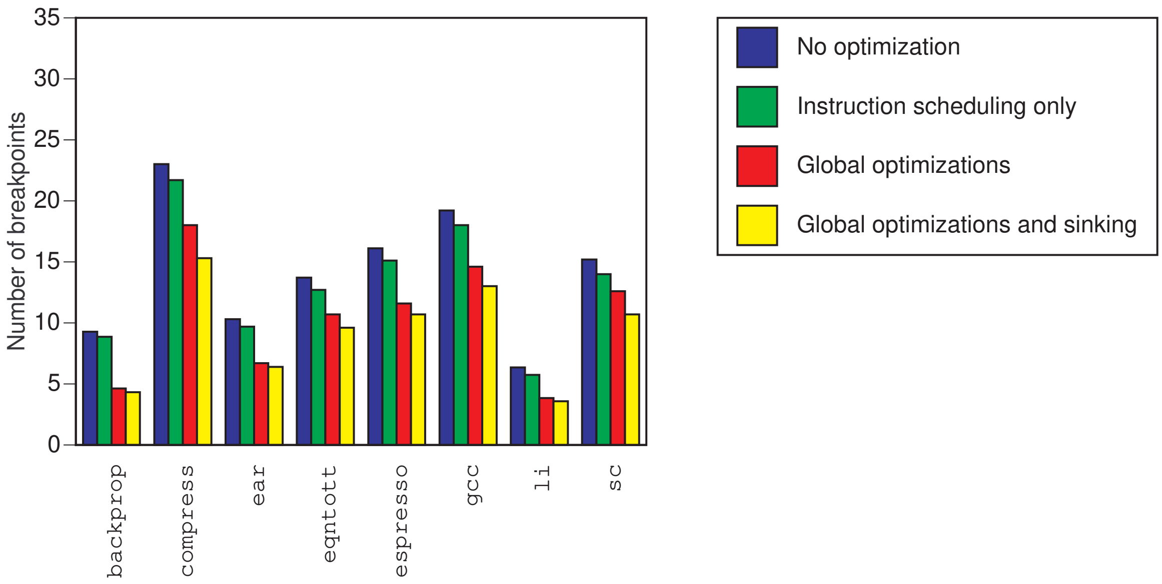


Figure 4.11: Comparison of average length of a variable's live range.

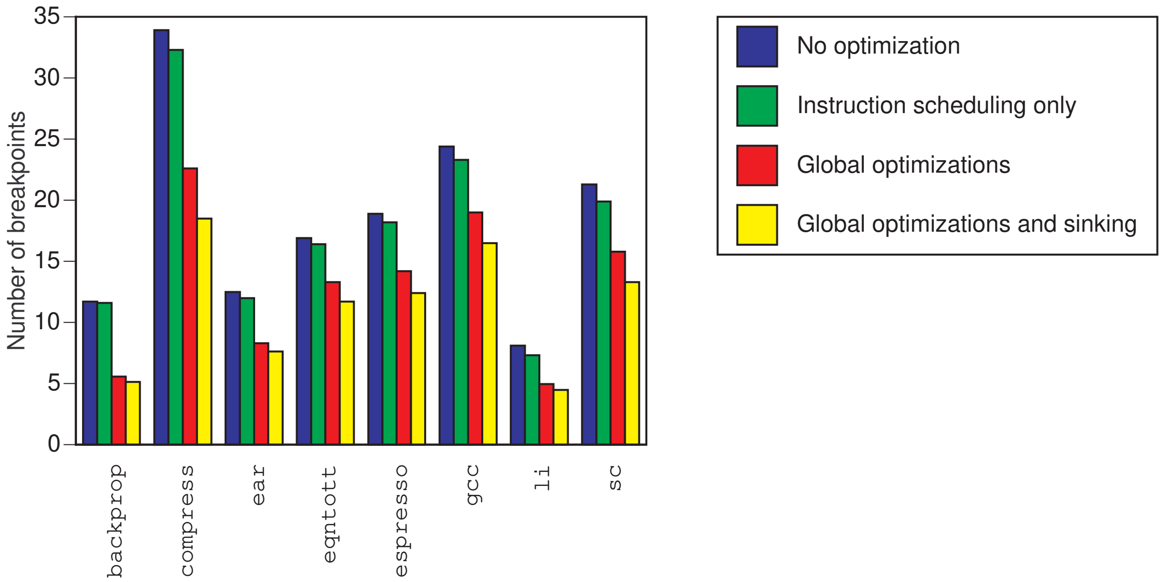


Figure 4.12: Comparison of average number of breakpoints a variable is resident using available residence data-flow analysis.

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 4.13(a) and 4.13(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 4.13(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 debugger can provide a more accurate classification for 12–22% of the local variables that

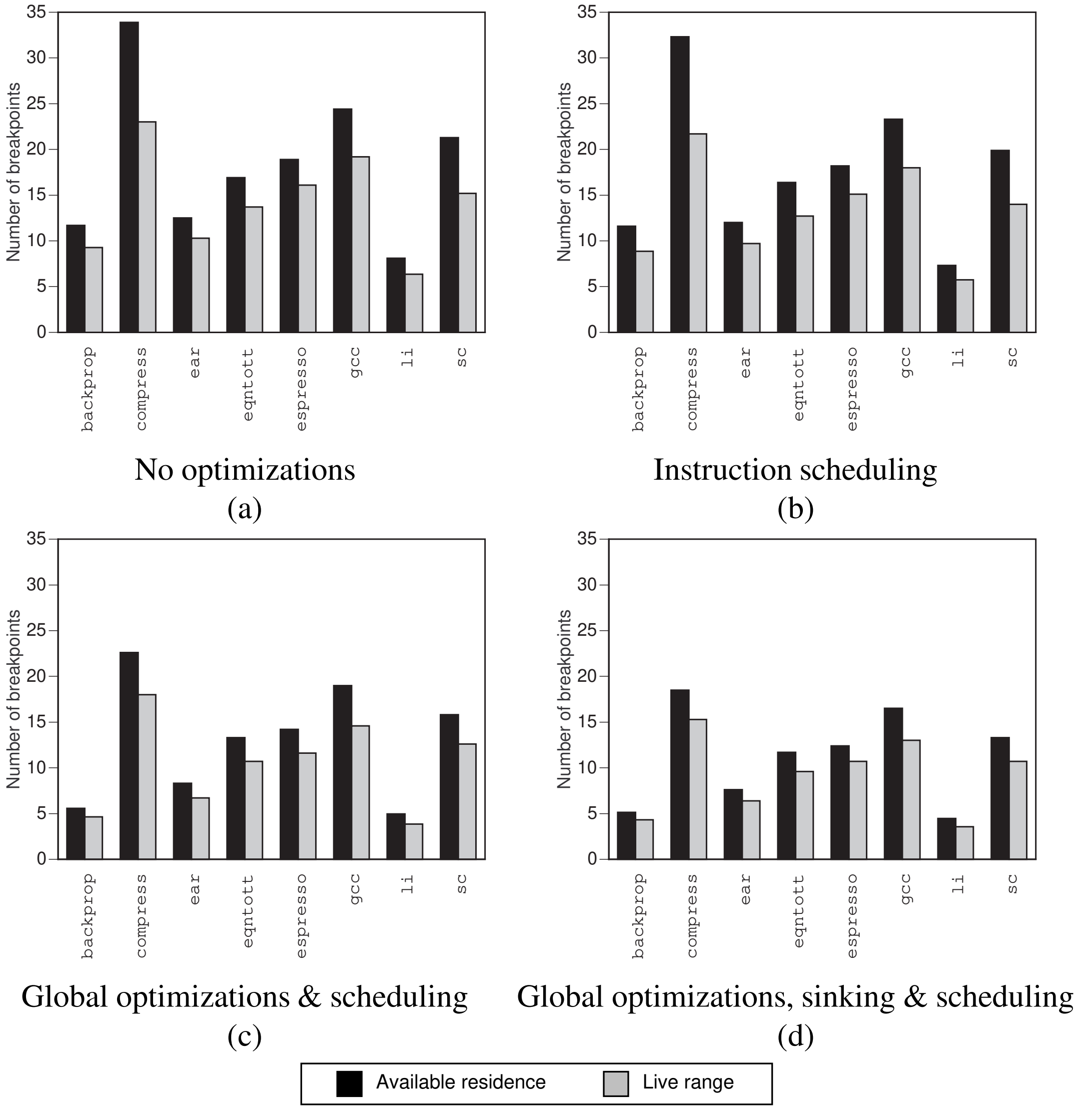


Figure 4.13: Average number of breakpoints a variable is resident.

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.

Bibliography

[1] A. Adl-Tabatabai and T. Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *Proc. ACM SIGPLAN'93 Conf. on Prog. 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 *Conf. Record of the 20th Annual ACM Symp. on Principles of Prog. Lang.*, 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. Engineering a global optimizer and code generator for reuse. Proc. Fifth Workshop on Compilers for Parallel Languages, Malaga, Spain (Tech Report UMA-DAC-95/09), pages 395–407, 1995.

[5] A. Adl-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. In *Proc. ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*. ACM, May 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 Languages Design and Implementation*. ACM, May 1996. To appear.

[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 *Proc. 16th POPL Conf.*, 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 *Proc. ACM SIGPLAN'82 Symp. of 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 Languages 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 *Proc. ACM SIGPLAN '89 Conf. on Prog. Language Design and Implementation*, pages 258–263. ACM, July 1989.

[14] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. ACM SIGPLAN '91 Conf. on Prog. 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 *Proc. ACM SIGPLAN'94 Conf. on Prog. 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 *Proc. ACM SIGPLAN'89 Conf. on Prog. 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 *Proc. ACM SIGPLAN'92 Conf. on Prog. 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 Languages Design and Implementation*, pages 53–65. ACM, June 1990.

[22] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of ASPLOS IV*, pages 40–52, Santa Clara, April 1991. ACM.

[23] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. ACM SIGPLAN'91 Conf. on Prog. 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 *Proc. ACM SIGPLAN 1982 Symp. 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 *Proc. 18th Intl. Symp. 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 Languages 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 Trans. on Prog. Lang. Syst.*, 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 Trans. on Prog. Lang. Syst.*, 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 *Proc. ACM SIGPLAN'88 Conf. on Prog. 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] D. Ebcioglu, R. Groves, K. Kim, G. Silberman, and I. Ziv. Vliw compilation techniques in a superscalar environment. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 36–48. ACM, June 1994.

[41] J. R. Ellis. Bulldog: A compiler for vliw architectures. Technical Report DCS/RR-364, Yale Univ., Feb. 1985. published by MIT Press, Cambridge MA., 1986.

[42] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch direction from previous runs of a program. In *Proc. Fifth Intl. Conf. on Architectural Support for Prog. 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 Trans. on Prog. Lang. Syst.*, 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 Trans. on Prog. Lang. Syst.*, 5(3):422 –448, July 1983.

[52] U. Holzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. ACM SIGPLAN'92 Conf. on Prog. 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 *Proc. ACM SIGPLAN '86 Symp. on Compler 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] D. Keppel. Fast data breakpoints. Technical Report UWCSE 93-04-06, University of Washington, April 1993.

[60] P. Kessler. Fast breakpoints: Design and implementation. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 78–84. ACM, June 1990.

[61] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN'92 Conf. on Prog. Language Design and Implementation*, pages 224–234. ACM, June 1992.

[62] J. Knoop, O. Ruthing, and B. Steffen. Lazy strength reduction. *Journal of Prog. Languages*, 1(1):71–91, 1993.

[63] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Prog. Lang. Syst.*, 16(4):1117–1155, July 1994.

[64] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 147–158. ACM, June 1994.

[65] P. Kolte and M. J. Harrold. Load/store range analysis for global register allocation. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Language Design and Implementation*, pages 268–277. ACM, June 1993.

[66] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Language Design and Implementation*, pages 318–328, June 1988.

[67] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS IV*, pages 63–74, Santa Clara, April 1991. ACM.

[68] 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.

[69] P. Lee and M. Leone. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Languages Design and Implementation*. ACM, May 1996. To appear.

[70] 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.

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

[72] E. C. Lyle. Debugging vliw code after instruction scheduling. Master's thesis, Oregon Graduate Institute, July 1992.

[73] 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 ASPLOS V*, pages 238–247, Boston,MA, October 1992. ACM.

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

[75] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[76] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. Fifth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, October 1992. ACM/IEEE.

[77] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Language Design and Implementation*, pages 266–277. ACM, June 1994.

[78] K. O'Brien, K.M. O'Brien, M. Hopkins, A. Shepherd, and R. Unrau. Xil and yil: The intermediate languages of tobey. In *Proc. ACM/SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 71–82, San Francisco, Jan 1995. ACM.

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

[80] 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.

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

[82] 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.

[83] R. Seidner and N. Tindall. Interactive debug requirements. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 9–22. ACM, 1983.

[84] M. Slater. Amd's k5 designed to outrun pentium. *Microprocessor Report*, pages 1–11, October 1994.

[85] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Fifth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS V)*, pages 248–259. ACM, Oct 1992.

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

[87] 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.

[88] R. Title. Thinking machines vendor update. In *Proc. Supercomputer Debugging Workshop '92*, pages 63–82, Dallas, TX, October 1992. Los Alamos National Laboratory.

[89] A. P. Tolmach and A. Appel. Debugging standard ml without reverse engineering. In *Proc. 1990 ACM Conf. on Lisp and Functional Prog.*, pages 1–12. ACM, June 1990.

[90] D. Ungar and R.B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241. ACM, October 1987.

[91] J. Uniejewski. Spec benchmark suite: Designed for today's advanced systems. *SPEC Newsletter*, 1(1), Fall 1989.

[92] R. Wahbe. Efficient data breakpoints. In *Proc. Fifth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS V)*, pages 200–212, Boston, October 1992. ACM/IEEE.

[93] R. Wahbe, S. Lucco, and S. Graham. Practical data breakpoints: Design and implementation. In *Proc. ACM SIGPLAN'93 Conf. on Prog. Language Design and Implementation*, pages 1–12. ACM, June 1993.

[94] D. Wall. Predicting program behavior using real or estimated profiles. In *Proc. ACM SIGPLAN '91 Conf. on Compiler Construction*, pages 59–70. ACM, June 1991.

[95] D. Wall, A. Srivastava, and F. Templin. A note on hennessy's "symbolic debugging of optimized code". *ACM Trans. on Prog. Lang. Syst.*, 7(1):176–181, January 1985.

[96] D. W. Wall. Global register allocation at link time. In *Proc. ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 264–275, Palo Alto, June 1986. ACM.

[97] H. Warren. Instruction scheduling for the ibm risc system/6000 processor. *IBM J. Research and Development*, 34(1):85–92, Jan 1990.

[98] 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.

[99] R. Wismueller. Debugging of globally optimized programs using data flow analysis. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 278–289. ACM, June 1994.

[100] 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.

[101] P. Zellweger. An interactive high-level debugger for control-flow optimized programs. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging*, pages 159–171. ACM, 1983.

[102] 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.

[103] 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.

[104] 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.