Nonresident and Endangered Variables: The Effects of Code Generation Optimizations on Symbolic Debugging

Ali-Reza Adl-Tabatabai

December 1992

CMU-CS-92-221

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Instruction scheduling and register allocation/assignment are two optimizations that are commonly used in the code generation phase of modern compilers. These optimizations are important for processors with exposed instruction-level parallelism and large register files. These optimizations, however, impact the task of the symbolic debugger which attempts to present to the user a source-level view of program execution.

The debuggers for most systems today usually punt the issue of optimized code, either by turning optimizations off whenever the user asks for source level debugging, or by not detecting the effects of optimizations on the source-level state. To not mislead the user, the debugger must provide feedback of the effects of optimizations. In this paper, we investigate the effects of instruction scheduling and global register allocation/assignment on symbolic debugging and present approaches that a debugger can take.

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government

Keywords: Symbolic debugging, evicted variables, nonresident variables, endangered variables, suspect variables, noncurrent variables, code generation, register allocation, instruction scheduling, compiler optimization, data-flow analysis

1 Introduction

A debugger provides a user with mechanisms to control the execution of a program (e.g., to set breakpoints) and to inspect the state of the execution (e.g., to print the current value of a variable). To qualify as a *symbolic* debugger, all interactions must be in terms of the high-level language program that is the source for the object program.

Optimizations commonly employed by current compilers duplicate, eliminate or reorder operations and values so that it is difficult for a symbolic debugger to discover the correspondence between source and object code. Such optimizations may make it difficult to set breakpoints, e.g., because the execution of multiple source statements is overlapped, or to inspect variables, because some values may either be inconsistent with what a user expects based on the source code, or may be inaccessible in the run-time state. A symbolic debugger for optimized code must detect these values and respond appropriately to a user query.

A number of modern high-performance processors expose instruction-level parallelism as well as large register files to the compiler. Since the parallelism and the storage hierarchy are exposed, the compiler has the opportunity to exploit the parallelism in the program and to reduce the memory traffic by keeping the most frequently accessed variables in registers. Instruction scheduling and register allocation/assignment are two compiler optimizations that are commonly used to take advantage of these features. These optimizations, however, affect setting breakpoints and inspecting variables by a symbolic debugger.

Global register allocation and assignment effectively exploit modern architectures that have large register files and high memory access latencies. These optimizations affect debugging by making variables inaccessible at a breakpoint. By attempting to pack as many variables as possible into a limited number of registers, global register allocation re-assigns registers to different variables at different points in the program. Therefore, a source level variable V may be inaccessible at a breakpoint if the register assigned to V holds the value of some other variable at that point, and there is no other location that holds V's value. Such a variable V is called a *nonresident* variable.

Instruction scheduling can increase the efficiency of processors with instruction-level parallelism by statically scheduling independent operations for concurrent execution. However, scheduling changes the sequence in which source level values are computed by reordering or interleaving code sequences from different source statements. If assignments are executed out of order, the sequence in which source level values are computed will be different from that specified in the source program. Consequently, at a breakpoint, the run-time value of an inspected variable may not be the value expected in the source. Such a variable is called an *endangered* variable. If an indirect assignment or function call operation is executed out of sequence, the debugger may not be able to determine the operation's side effects on the source state. As a result, the debugger may not be able to determine with certainty whether a variable V's run-time value is the same as the value the user expects V to have. Hence, there are two types of endangered variables. If the debugger can determine with certainty that an endangered variable V's run-time value is not the same as V's expected source value, then V is a *noncurrent* variable. Otherwise, V is a *suspect* variable.

From the viewpoint of the user, nonresident and endangered variables are similar in that the debugger cannot display the variable's expected source value. However, 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 run-time 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 may be helpful to the user if the debugger can convey what source value an endangered variable's value corresponds to [13,9]. On the other hand, no source value can be displayed for a nonresident variable.

This paper investigates the problem of detecting nonresident and endangered variables in the presence of global register allocation and instruction scheduling. We present a solution to detecting nonresident variables based on using data-flow analysis techniques *for the debugger* to detect all points where source level values are in their assigned run-time locations. The problem of detecting nonresident variables is concerned only with whether a variable V's register contains *any* source value of V. We also present an approach to detecting and recovering endangered variables caused by instruction scheduling. Detection of endangered variables is concerned with whether the value in a resident variable V's run-time location is the expected source value of V. *Recovery* attempts to produce the expected source value of an endangered variable from the run-time state.

To evaluate the effectiveness of our approaches (and the seriousness of the problem) we have implemented the techniques in a production C compiler that performs code compaction and register allocation for a long instruction word (LIW) machine with a large register file. We compare the effects of nonresidency with the effects of endangerement on the debugger's ability to recover source values at a breakpoint. Our results indicate that nonresident variables are a serious problem; the assumption (made if endangerement is the only issue of concern to the debugger) that a variable is always accessible in its assigned run-time location presents a picture that is too optimistic. Furthermore, a separate data-flow analysis phase in the debugger for tracking a variable's run-time location significantly improves the number of variables accessible by the debugger; an overly conservative approach to tracking a variable's run-time location (as presented in [13]) misses many opportunities. We also measure the effectiveness of simple recovery strategies on the debugger's ability to report expected source values and to precisely determine which variables are endangered. Our results show that a simple recovery scheme is successful in increasing the number of variables for which the debugger can report the expected source value and improving the debugger's accuracy in detecting endangered variables.

2 Debugging optimized code

In this section we introduce the key concepts of our symbolic debugger and describe how optimizations affect a symbolic debugger's behavior.

2.1 Debugger model and terminology

The debugger can be invoked as a result of two types of *breaks*: *synchronous* and *asynchronous*. A *synchronous* break occurs when control reaches a *control breakpoint* that was set at a source statement by the user. When a synchronous break invokes the debugger, the debugger reports that execution has stopped *at* the statement S where the user has set a breakpoint. An *asynchronous* break occurs when an instruction raises an exception, or when the user interrupts program execution. The debugger maps the instruction I at which execution halted to the source statement S for which I was generated and reports that execution has stopped due to an exception *within* source statement S.

A *data break* occurs when the program writes to a storage location that is monitored by a *data breakpoint* set by the user. A source statement may contain several assignment expressions that may each cause a data break. Therefore, since a data break can occur *within* a source statement (as opposed to at a statement boundary), a data break is considered to be an asynchronous break. We do not discuss the mechanics of how control or data breakpoints are implemented. See [15] or [19] for possibilities.

When a break occurs, the point in the object at which execution has halted is called the *object breakpoint*, and the source statement where the breakpoint is reported is called the *source breakpoint*. Since there is always a source and object breakpoint associated with every break, a *breakpoint* refers to a pair $< S,O >$, where S is the source breakpoint and O is the object breakpoint.

The debugger is *non-invasive*; no modification of the program's code or data is allowed, except for modifications neccessary for setting breakpoints (as, for example, described in [15]), i.e., the code generated for debugging is identical to the code generated otherwise, and the storage layout of the program is not perturbed. Our model does not allow the compiler to insert extra code to make debugging easier. For example, the compiler does not insert *path determiners* [21] into the object code to determine the execution path leading to a breakpoint, even though such knowledge allows the debugger to perform better analysis while retrieving source values. Furthermore, registers are only saved when necessary for the execution of the program, the compiler does not save old values solely to assist the debugger. The compiler will, however, leave sufficient information describing correspondences between the object and source codes, such as a mapping of variables to storage locations.

Debugger functions can be classified into two groups: related to program flow (setting breakpoints: mapping a source-level location into a location in the object code; reporting exceptions: mapping a faulting machine operation into source code), and related to data (reporting the values of user variables). Problems related to the former are known as *code location* problems, while those related to the latter are known as *data value* problems [22]. In this paper, we only address the data value problem of retrieving source level values (e.g., values of source variables or locations in the heap) from the run-time state of a halted program. Our work assumes that breaks can occur anywhere in the object code and hence applies to both synchronous and asynchronous breaks. We do not discuss methods of mapping breakpoints in the source to breakpoints in the object. See [9] or [21] for a discussion of flow related issues.

Data modification by the user is complicated in optimized code since storage locations can be re-used, and expected uses of source variables can be re-ordered or eliminated by transformations such as instruction scheduling and common subexpression elimination. We do not address this problem in this paper, and data modification by the user is not supported in our debugger model.

2.2 Retrieving source values

In response to a query of a variable V's value at a source breakpoint S, the user expects the value from the latest source assignment to V, relative to S. This value is the *expected* value of V [9]. A variable's expected value is not always retrievable from the run-time state of an optimized program. Two conditions must be satisfied to retrieve a variable V's expected value at a breakpoint:

1. V must be accessible in a storage location (memory, register, condition codes, ...) of the machine. If the debugger determines that V is accessible in a storage location, V is called *resident*, otherwise V is called *nonresident*. The storage location where V is accessible is called V's *residence*, and the value in V's residence is called V's *actual* value [9].

2. V's actual value must be the same as V's expected value. If V's actual value may not correspond to V's expected value, V is called *endangered* [14]. Sometimes the debugger can determine with certainty that a variable's actual value does not correspond to the variable's expected value; such a variable is called a *noncurrent* variable. If a variable V is endangered, but the debugger cannot determine with certainty that V is noncurrent, then V is called a *suspect* variable. (An endangered variable is either noncurrent or suspect, but not both.) A nonresident variable does not have an actual value, hence endangerement can not apply to such a variable.

In unoptimized code, a variable has a home location whose value always matches the variable's expected value at a breakpoint. Thus the debugger can retrieve a variable's expected value from the variable's home location. Optimizing transformations complicate the retrieval of values by violating the conditions above; either a variable is inaccessible because the debugger determines the variable has no residence, or a variable is resident, but the variable's actual value may not be the same as its expected value.

2.3 Approaches to debugging optimized code

There are several approaches that a debugger can take to handle code that has been optimized. The debugger can try to completely hide the effect of optimizations, i.e., the debugger presents the *expected* behavior[21,9]. To provide expected behavior, the debugger must detect all endangered variables and recover their expected values.

Recovering the expected values is not always possible (Section 2.5), therefore most debuggers for optimized code settle for *truthful* behavior [21,9]: the debugger detects the set of variables that are nonresident or endangered, and either reports them as such in response to a user query, or attempts to recover their values, although recovery may not always be successful[14,9]. 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 an expected value cannot be presented, the debugger may provide additional guidance to the user by conveying how optimizations have affected source values. For example, the debugger may tell the user at which source assignmment(s) an endangered variable's actual value was (or may have been) assigned.

2.4 C source language and compiler issues

Detecting and recovering endangered variables at all possible object breakpoints requires analysis of both the source and object programs to accurately determine expected and actual variable values. Hence language features and semantics (e.g., pointers and evaluation order), effects of compiler transformations (e.g., re-ordering of operations), as well as target machine features (e.g., concurrent execution of multiple operations) must be precisely modelled. Earlier work in debugging optimized code focused on simple languages with restricted features, e.g., a subset of Pascal without pointers[14], allowed only synchronous breakpoints at statement boundaries, requiring only the modelling of inter-statement expected values [13,9], or considered only simple compiler models. However, C is nowadays a widely used programming language, and the practice of debugging has evolved to include asynchronous breakpoints (e.g., for data breaks, post-mortem analysis, or user interrupts like Control-C). Supporting asynchronous breaks requires modelling of what occurs *within* a statement, to correctly determine expected source values at a breakpoint.

Compiling the C language creates the following problems that must be addressed by the debugger:

1. **Indirect assignments and function calls.** If an indirect assignment (i.e., an assignment through a pointer) is executed out of order, the memory location targeted by the assignment is noncurrent. Therefore, to detect which memory location is affected by the assignment the debugger must recover the value of the indirect assignment's address expression. However, like recovery of endangered variables, recovery of indirect assignment address expressions is not always successful, in which case the debugger must conservatively assume that any memory location may be affected. Thus when the debugger detects that an indirect assignment has executed out of order and it cannot recover the assignment's address expression, it must report all variables with home locations in memory as suspect (and this includes the heap as well). Similarly, function calls can be reordered with respect to other language statements (e.g., other function calls or assignments), in which case the debugger must again report all variables with home locations in memory as suspect. As a compiler's ability to analyze the global effects of functions increases, it is more likely that an operation is executed out of order with respect to a function call.

2. **Undefined evaluation order.** To correctly determine expected source values at an asynchronous break, the debugger must accurately model the execution order of source side effects

within and among statements. The evaluation order of side effects from different source expressions is well defined. However, in a C language expression containing multiple side effecting expressions (i.e., assignments or function calls) the compiler is free to choose the evaluation order of the side effecting expressions. E.g., in the C code fragment of Figure 1, the assignment in statement S_1 must execute before the assignments in S_2, and all assignments in S_2 must execute before the assignment in S_3, but the compiler is free to choose the evaluation order of the three assignments within statement S_2. Thus, at an asynchronous breakpoint $< S_2, O >$ occuring at the expression y++ within S_2, the expected value of x depends on the evaluation order determined by the compiler and can be the value assigned by either S_1 or the expression x++ of S_2.

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

Figure 1: Undefined evaluation order in C

2.5 Example

Consider the source and object codes shown in Figures 2 (a)-(b). In this example, variables a and d have both been assigned register *r5*, while b and p have both been assigned register *r4*. Variable c has been assigned register *r3* and variables e,f and g reside in memory. Figure 2(c) shows the ranges of instructions during which register assigned variables are resident. c is resident in *r3* throughout this block. Registers *r4* and *r5* contain the values of p and a respectively upon entry to this block, hence b and d are nonresident. At I_6, *r5* is reassigned to d, and as a result a becomes nonresident, while at I_5, *r4* is reassigned to b causing p to become nonresident.

```
S1:  d = f+g;
S2:  b = f-a;
S3:  *p = a;
S4:  c = a*c;
S5:  e = b;
```

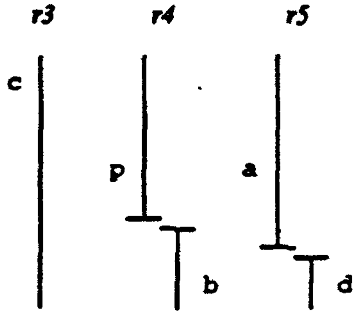
Source code

(a)

```
I1:  r1   <--  load f
I2:  r2   <--  load g
I3:  r3   <--  fpmul r5,r3
I4:  (r4) <--  store r5
I5:  r4   <--  fpsub r1,r5
I6:  r5   <--  fpadd r1,r2
I7:  e    <--  store r4
```

Object code

(b)



Residence ranges

(c)

Figure 2: Example of noncurrent and nonresident variables

Table 1 lists the nonresident and endangered variables at all possible object breakpoints in the code of Figure 2. We report an asynchronous break at an instruction I as a break occuring at the source statement for which I was generated. The first two columns of Table 1 show the mapping from object breakpoints to source breakpoints. The third column presents the source expression that is computed by each instruction. The last three columns of Table 1 list the nonresident, noncurrent and suspect variables if a break happens at a given instruction. Note that the value computed by I_1 is a common subexpression (f) of S_1 and S_2. This instruction, however, is mapped to S_1, since a break occuring at a common subexpression must logically happen at the earliest source statement in which the common expression occurs [14].

Breakpoint		Source Expression	Nonresident	Endangered Variables	
Object	Source	Evaluated by Instruction	Variables	Noncurrent	Suspect
I_1	S_1	f	b,d		
I_2	S_1	g	b,d		
I_3	S_4	c = a*c	b,d		e,f,g
I_4	S_3	*p = a	b,d		
I_5	S_2	b = f-a	b,d	c	e,f,g
I_6	S_1	d = f+g	p,d	c,b	e,f,g
I_7	S_5	e = b	p,a		

Table 1: Nonresident and noncurrent variables at breakpoints in code of Figure 2

Consider a breakpoint at instruction I_3, reported at statement S_4 in the source. At this breakpoint, the expected value of d is the value assigned at statement S_1 and the expected value of b is the value assigned at statement S_2. Both of these variables are nonresident at this breakpoint, because the registers assigned to these variables (*r5* and *r4*) are holding values of other variables (a and p). Therefore b and d have no actual values. The expected and actual values of a,c and p are the values assigned by the last assignments to these variables before this block. Therefore, these variables are current at this breakpoint. The indirect assignment of statement S_3 has not yet executed at this breakpoint. Consequently, the value in the memory location M pointed to by p is not the value the user expects M to have. Since M may be the home location of e,f or g, the expected value of one of these variables may be the value assigned by S_3. The actual values of e,f and g are the values assigned by the last assignments to these variables before this block. The value of p can be retrieved from *r4*, and therefore the debugger can precisely determine at run-time the location M. Consequently, the debugger will report one of e,f or g as noncurrent and the other two as current. But since this is a run-time value, we cannot statically determine which memory location is noncurrent. Therefore, for the purpose of this example, we show these memory variables as suspect. Note, however, that at a breakpoint at I_6, the assignment of S_3 has executed prematurely at I_4 and again the memory location pointed to by p is noncurrent. But p is nonresident at I_6 and the debugger will not be able to determine at run-time which memory location is noncurrent. Consequently, e,f and g will be reported as suspect.

The compiler may have information that describes which variables may be aliased by p at a breakpoint. However, program bugs, e.g., a pointer incremented past the end of an array (a typical bug in C programs), invalidate alias information gathered by the compiler. The debugger, therefore, cannot use alias information to refine the set of variables that are supect at a breakpoint, e.g., the debugger cannot rule out e,f or g as unaffected by I_4, even if compler alias information indicates that I_4 will not target these variables.

2.6 Nonresident and endangered variables

Observe that endangered and nonresident variables are different with regard to the possible behavior of the debugger. In the case of an endangered variable V, the debugger may provide additional information to the user by displaying V's actual value and attempting to explain what value is being displayed. E.g., consider a breakpoint occurring at I_6 and reported at statement S_1 in Figure 2. At this breakpoint, the assignment to b of statement S_2 has executed too early at I_5. In response to a user query of b, the debugger may 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. This approach was adopted in the DOC debugger [13]. Copperman [9] gives suggestions about what information should be presented to the user.

In the case of a nonresident variable, however, no actual value exists that can be presented to the user. For example, at a breakpoint at I_3, 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 can only inform the user that the requested variable has been optimized away (i.e., the variable is unavailable), as is done in DOC [13] and CXdb [5].

An approach to dealing with nonresident and endangered variables is to *recover* a variable's expected value from the actual values of other variables and temporaries [14]. E.g., at a breakpoint at I_3 (reported at statement S_4 in the source), d and b are nonresident. The expected values of these variables are from statements S_1 and S_2, which are computed by instructions I_6 and I_5. The operand values of these two instructions (the values in registers *r1*, *r2* and *r5*) are available at I_3 and thus the debugger can provide the expected values of b and d by interpreting instructions I_5 and I_6. In general though, recovering values in globally optimized code is difficult (see [14] for a discussion of the scenarios when recovery can be attempted) and is not always possible. E.g., to recover b's value at a break at I_1, the debugger must detect the latest source assignment to b that was executed before this block of code. Using reaching definitions [2], the debugger can determine which assignments to b reach the breakpoint S_1. However, there may be several definitions of b reaching this block, e.g., because of different assignments reaching on different execution paths leading to this breakpoint or because of ambiguous definitions of b, or there may be only one assignment to b that reaches, but not on all paths. Consequently, the debugger may not be able to determine which reaching source assignment computes b's expected source value. In Section 5 we discuss recovery of endangered variables caused by one case of code re-ordering.

2.7 Uninitialized variables

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. The debugger may either detect and warn the user of uninitialized variables, or it may let the user beware. Detecting and reporting uninitialized variables can reduce the number of variables that are reported as nonresident or endangered and provides additional information to the user.

In the absence of support provided by the run-time system (e.g., path determiners [21]) or the architecture (e.g., memory tags), detecting uninitialized variables requires that the debugger obtains program flow analysis information from the compiler. 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. Note that the debugger cannot help in the case that definitions reach on some but not all paths to S.

Referring back to the example of Figure 2, if no source assignment of b reaches the block of code, b can be reported as uninitialized rather than nonresident or endangered at any breakpoint reported at S_1. In this example, these are breakpoints that occur at I_1, I_2 and I_6.

3 Global register allocation

Register allocation and assignment attempt 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, we do not mention temporaries or constants any further.

Our model of register allocation is similar in style to Chaitin's [6] and is based on the optimizing compiler that we have used in our empirical studies. In our model, a variable is either promoted to a register (selected to reside in a register) or given a home location in memory. Register assignment binds physical registers to register promoted variables, and in our compiler, register assignment happens after instruction scheduling. A register is assigned for exclusive use by a variable during the variable's *live range* [7], which consists of instruction ranges between definitions and last uses of the variable.

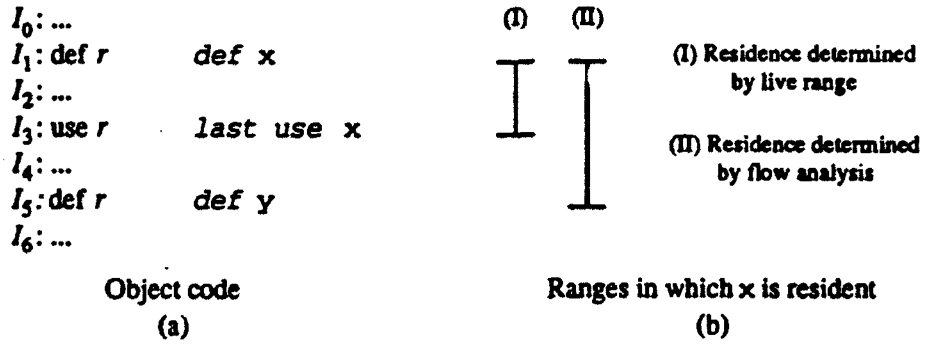
If spilling is required during register assignment, the whole live range of a variable is spilled to memory. Loads and stores are added to the schedule to access spilled variables. Disjoint segments of a live range are not renamed, nor are live ranges split during register assignment. Thus, each register promoted variable V is either spilled to memory (if there are not enough registers), or it is assigned a single physical register denoted $R(V)$ for the duration of its live range. A register promoted variable that is assigned a physical register is referred to as a *register assigned variable*. Variables that have home locations in memory (including those that are spilled to memory) are always resident, since their storage locations are not shared with other variables. Register assigned variables, however, may be nonresident since a register is usually assigned to many variables.

Coalescing, also known as subsumption [6], eliminates copy operations. This optimization coalesces two variables whose live ranges do not interfere and are connected by a copy operation. As a result, both variables are assigned the same physical register.

Other register allocation and assignment models allow a variable to exist in different storage locations at different points in the program, e.g., by splitting live ranges or by allocating registers seperately in different regions of a program. The approach to detecting nonresident variables described in this paper can also be extended to these models.

4 Detecting nonresident variables

There are several approaches that a debugger may take to determine if a variable is resident at a given object location. Since a variable is resident during its live range, one way to detect resident variables is to consider a variable as resident at an object breakpoint within the variable's live range. The advantage of this approach is that live range information is computed at compile time by the compiler's register assignment phase. E.g., in the DOC debugger [13], 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 assigner. The range records are passed to the debugger, which uses them to detect whether a breakpoint lies within a variable's live range.


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: ...

(I)
(II)
(I) Residence determined by live range
(II) Residence determined by flow analysis
Object code
(a)
Ranges in which x is resident
(b)

Figure 3: Example illustrating the approaches to detecting nonresident variables

Using a variable's live range for determining residency is simplistic and conservative; the debugger uses a simple rule that is always right but misses opportunities. A variable's assigned register may still be holding the variable's value after the variable's live range (e.g., after the last use of the variable).

This is illustrated in Figure 3(a). This figure shows a sequence of definitions and uses of a register r in a straight line piece of object code. Register r has been assigned to source variables x and y. The definition at I_1 writes a value of x into r, and this definition 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 a breakpoint occurring at either 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 (eviction is discussed in Section 4.2). But the range of instructions after I_3 are not part of x's live range. Hence, at a breakpoint at 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.

4.1 Terminology

Before discussing the details of our approach, we introduce some terminology. A control flow graph is a directed graph (B, B_S, E) where B is the set of basic blocks; $B_S \in B$ 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 *preamble instruction* that appears before the other instructions in B_i, thus dominating them. A preamble instruction is an abstraction that is used by our algorithms, it is not generated by the compiler nor does it appear in the object code. Given an instruction 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. 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* O_S of the control flow graph is the point at the beginning of the source basic block B_S. The entry point dominates all other points in the control flow graph. A *path* is defined to be a sequence of points $O_1 ... O_n$ such that for each adjacent pair O_i, O_{i+1}, either $O_i = pre(I)$ and $O_{i+1} = post(I)$ for some instruction I, or O_i is a point at the end of a block B_j and O_{i+1} is a point at the beginning of a block B_k and $(B_j, B_k) \in E$.

We call an instruction that targets a register r a *definition* of r. An instruction I *reaches* a point O if there exists a path from *post*(I) to O along which the register defined by I is not redefined. The set of definitions of a register r that reach a point O is denoted by *ReachingDef(r,O)*.

4.2 Evicted variables

An approach to detecting nonresident variables that is more accurate 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 3(a) and allows the debugger to display the value of x outside of x's live range, as depicted by Figure 3(b). In the rest of this section, we describe a method based on data-flow analysis that realizes such an approach by detecting *evicted variables*. At a breakpoint, an evicted variable is a register assigned variable V whose assigned register $R(V)$ may contain a value that is not from a source assignment to V. Since only register assigned variables can be nonresident, all further references to variables in this section mean references to register assigned variables.

To track whether a variable V's value is held in $R(V)$, definitions that write a source value of V into $R(V)$ must be distinguished:

Definition 1 *Let E be a source assignment expression that assigns to a variable V. Of the instructions generated for E, the instruction that assigns V's value to $R(V)$ is referred to as a* **source definition** *of V and is denoted by $I_V(E)$.*

At the point immediately following a source definition of V, V is resident since $R(V)$ holds a value from a source assignment to V.

Referring back to Figure 2, instructions I_3, I_5 and I_6 are generated for statements S_4, S_2 and S_1 respectively. These instructious target the registers assigned to variables c,b and d with the source values of these variables computed at statements S_4, S_2 and S_1. Hence I_3, I_5 and I_6 are source definitions of c,b and d : $I_c(S_4) = I_3$, $I_b(S_2) = I_5$ and $I_d(S_1) = I_6$.

To detect whether a variable V is evicted, the debugger must analyze paths leading to a breakpoint to discover which value is being held by $R(V)$:

Definition 2 *A variable V is* **evicted along a path** *P in the object if execution of the path P results in $R(V)$ containing a value that is not a value from a source definition of V. If $R(V)$ is uninitialized along P, then $R(V)$ is considered as having no value, and V is not evicted along P.*

Definition 3 *A variable V is* **evicted at a point** *O in the object, if V is evicted along at least one path leading from O_S (the entry point) to O. The predicate $IsEvicted(V,O)$ is true if a variable V is evicted at point O in the object.*

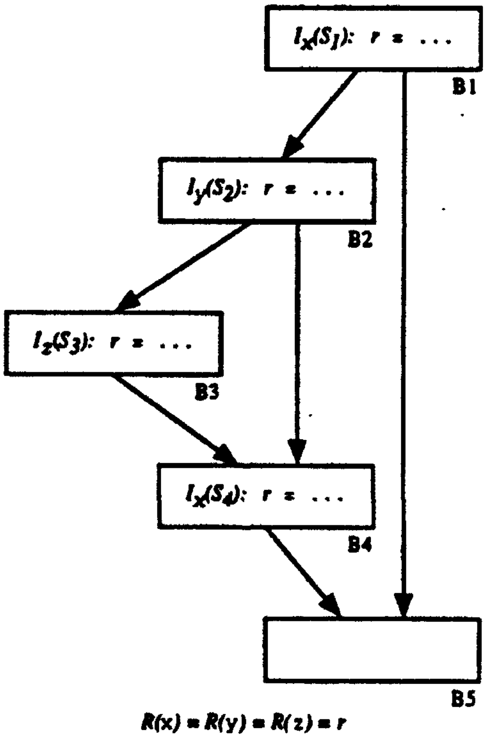


Figure 4: Object control flow graph

Note that the definition of an evicted variable does not depend on where the breakpoint is reported in the source. We are concerned only with whether $R(V)$ holds a value of V, not whether it holds the expected value of V.

Consider the control flow graph of Figure 4. In this figure, variables x, y and z have all been assigned the same register r. Each basic block contains at most one instruction that is a source definition of one of the variables. At the beginning of block B3, x and z are evicted, since all execution paths leading to this point result in r containing a value of y. Similarly, at the beginning of block B5, y and z are

Basic Block	Evicted Variables at Start of Block	Evicted Variables at End of Block
B1		y,z
B2	y,z	x,z
B3	x,z	x,y
B4	x,y,z	y,z
B5	y,z	y,z

Table 2: Evicted variables at the start and end of each block in Figure 4

evicted since all execution paths leading to this point result in r containing a value of x. Table 2 lists the evicted variables at the start and end of each basic block in Figure 4.

To detect whether a variable V is evicted at a point O, we must consider all values that may possibly be contained in $R(V)$ if execution is halted at O. This can be accomplished by examining the set *ReachingDef(R(V),O)*. If there exists any definition $D \in$ *ReachingDef(R(V),O)*, such that D is not a source definition of V, then there exists a path leading to O which results in $R(V)$ containing a value that is not a value from a source definition of V, and hence by Definition 3 *IsEvicted*(V,O) is true. Conversely, if *IsEvicted*(V,O) is true, some definition of $R(V)$ that is not a source definition of V must reach O. Thus, the eviction problem may be cast in terms of reaching definitions:

Lemma 1 *A variable V is evicted at a point O in the object iff there exists a definition D ∈ ReachingDef(R(V),O) such that D is not a source definition of V.*

4.3 Computing evicted variables

By Lemma 1, *IsEvicted*(V,O) can be solved for by computing the set *ReachingDef(R(V),O)*, and checking whether there is a definition in *ReachingDef(R(V),O)* that is not a source definition of V. A simpler approach to computing *IsEvicted*(V,O) is to track whether *any* definition of $R(V)$ that is not a source definition of V reaches O, using data-flow analysis. Whereas a source definition writes the value of a variable V into $R(V)$, an *evicting definition* of V writes the value of a variable other than V into $R(V)$:

Definition 4 *An* **evicting definition** *of a variable V is a definition of R(V) that is not a source definition of V.*

Given an evicting definition I of a variable V, we say V is *evicted* by I (or I *evicts* V). Table 3 lists the variables evicted by the instructions in Figure 4. For each register promoted variable V, we define the predicate *EvictReach*(V,O) as follows:

Definition 5 *The predicate EvictReach(V,O) is true at a point O in the object if any evicting definition of variable V reaches O.*

Note that by Lemma 1, *EvictReach*(V,O) is equivalent to *IsEvicted*(V,O). Hence, solving for *EvictReach*(V,O) is equivalent to solving for *IsEvicted*(V,O). Also, since no evicting definitions reach the entry point O_S of the flow graph, no variable is evicted at O_S.

Given an instruction I, let *EvictReachIn*(I) be the set of variables $\{V : \mathit{EvictReach}(V, \mathit{pre}(I))\}$ and let *EvictReachOut*(I) be the set of variables $\{V : \mathit{EvictReach}(V, \mathit{post}(I))\}$. Evicting definitions of a variable V reach the point immediately before an instruction I if evicting definitions of V reach

Instruction	Variables Evicted by Instruction
$I_x(S_1)$	y,z
$I_y(S_2)$	x,z
$I_z(S_3)$	x,y
$I_x(S_4)$	y,z

Table 3: Variables evicted by instructions in Figure 4

the points after *any* of I's predecessor instructions. Thus the *EvictReachIn* set of an instruction I is related to the *EvictReachOut* sets of I's predecessor instructions by the following data-flow equation:

$$EvictReachIn(I) = \bigcup_{J \in Pred(I)} EvictReachOut(J)$$

An instruction I that is an evicting definition of a variable V causes $EvictReach(V, post(I))$ to be true, and thus *generates* a reaching evicting definition of V. The set of variables that are evicted by an instruction I is denoted by $EvictReachGen(I)$. Similarly, an instruction J that is a source definition of V re-establishes V's residence, by *killing* all reaching evicting definitions of V, and causes $EvictReach(V, post(J))$ to be false. $EvictReachKill(J)$ denotes the set of variables for which J is a source definition. The sets *EvictReachGen* and *EvictReachKill* are defined for an instruction I that defines register $R(V)$:

- If I is a source definition of V, then $V \in EvictReachKill(I)$.
  Otherwise, $V \notin EvictReachKill(I)$.
- If I is an evicting definition of V, then $V \in EvictReachGen(I)$.
  Otherwise, $V \notin EvictReachGen(I)$.

The *EvictReachIn* and *EvictReachOut* sets of an instruction are related by the following data-flow equation:

$$EvictReachOut(I) = (EvictReachIn(I) \cup EvictReachGen(I)) \setminus EvictReachKill(I)$$

Function calls kill the contents of caller saved registers and therefore evict all variables that are assigned caller saved registers.

5 Detecting and recovering endangered variables

Noncurrency occurs when an assignment to a variable is executed out of source order. Such an assignment makes a variable's actual value unequal to its expected value by either pre-maturely overwriting the expected value with a future value or by delaying the update of the expected value. Therefore, there are two types of noncurrent variables for a given breakpoint: a *roll forward* variable is one whose expected value is assigned by a source assignment that has not been executed at the breakpoint, while a *roll back* variable is one whose actual value is from a source assignment that occurs after the source breakpoint [14].

To detect noncurrent variables, the debugger must detect which operations have executed out of order and how these operations affect the source level state (i.e., variables and memory locations). Operations that affect source state include assignments to source variables and function calls, but for conciseness, we only mention assignments in the rest of this section.

5.1 Execution order

We assume a target machine with a precise interrupt model; i.e., at an object breakpoint O, all instructions scheduled prior to O have executed, while no instructions scheduled at O or after O have completed execution or caused side effects on the run-time state. Either an instruction at O raised an exception, a user interrupt halted execution at O, or a breakpoint was reached at O.

We define the *canonical execution order* of source expressions to be the order in which expressions in the source program are supposed to execute according to the semantics of the source language. During symbolic debugging, the user expects variables to be updated according to the canonical execution order of assignments in the source.

Instruction scheduling changes the order in which source expressions execute, and as a result variables may not be updated in canonical execution order. To detect assignments that have executed out of source order, a model is necessary that records the source execution order of assignments and links it to the object execution order of the assignments. Our compiler (like all optimizing compilers we are aware of), however, first maps source expressions to an intermediate representation (IR), and then the code generator maps the IR into machine instructions. So our model must link the source assignments with the object assignment instructions via the IR.

We annotate a program's IR with information describing the canonical execution order of the IR operations and the code generated for each IR operation. The canonical execution order of assignments is captured by annotating each assignment operation A in the IR with a sequence number $Seq(A)$. IR operations within the left and right hand side expressions of A are given the same sequence number as A. Given two assignments A and B, both in the same basic block, $Seq(A) < Seq(B)$ implies that A executes before B in canonical execution order, while $Seq(A) = Seq(B)$ implies that the canonical execution order of A and B is undefined (their execution order is undefined in the source). E.g., in the example of Figure 1 the three assignment expressions of statement S_2 will all have the same sequence number. Note that the partial ordering defined by the *Seq* annotation captures the canonical execution order of statements within the same basic block, not the dependences that constrain the correct execution order(s).

The order in which IR operations are executed in the object is determined by the code scheduler, and the code scheduler must pass this information to the debugger. Each IR operation may translate into multiple instructions, which are placed by the scheduler at different positions in the basic block schedule. Therefore, each IR operation N is annotated with the list of instructions generated for N.

The position of the last instruction generated for an IR operation N is denoted by $Sched(N)$. $Sched(N)$ captures the relative order in which IR operations complete execution in the object: given two IR operations N and M, $Sched(N) < Sched(M)$ implies that N completes execution before M in the object, while $Sched(N) = Sched(M)$ implies that N and M complete execution concurrently[1]. For an IR assignment operation A, such that A assigns a value to a variable V, $Sched(A)$ is the basic block position of the instruction I that performs the assignment. If V has been assigned a register, then I targets the register assigned to V, otherwise I stores to V's home location. Thus, the order in which assignments are executed is recorded.

When a break occurs at an instruction I, the object breakpoint is mapped to the IR operation N for which I was generated, denoted *IR(I)*, and referred to as the *IR breakpoint operation*. The annotated IR is then examined to detect which IR operations have executed out of sequence with respect to the IR breakpoint operation. This approach is similar to Hennessy's [14], however, our model considers values held in the physical registers of the machine as well as individual instructions generated for each

[1] This definition covers statically scheduled machines that can execute multiple instructions (or machine operations) concurrently. Hence the need for the case where two instructions are scheduled at the same block offset.

IR operation.

For each instruction I, we record its position in the basic block schedule. Since our code generator performs local instruction scheduling only, it is sufficient to record I's offset from the start of the basic block. Each instruction I is also annotated with the position (in the basic block) of the last and next *local* definitions of its source and destination registers, denoted $LastDef_I(R)$ and $NextDef_I(R)$, where R is a source or destination register of I. If no local last definition of R exists, then $LastDef_I(R)$ is -1. Similarly, if no local next definition of R exists, then $NextDef_I(R)$ is set to a value beyond the last position in I's basic block.

The set of source registers of an instruction I is denoted by $SourceRegs(I)$, and the destination register of an instruction I is denoted by $DestReg(I)$. If I is a load or store instruction, then I's set of address registers (e.g., base or index registers) is denoted by $AddressRegs(I)$.

5.2 Out-of-order operations

To determine if there are noncurrent or suspect variables at a breakpoint, we have to find out if any operations with side effects on the user-visible state executed out of order. Side effects on temporaries (e.g., for address arithmetic) do not cause noncurrent variables.

Given an object breakpoint O and the IR breakpoint operation M, we call an IR operation N that is performed out of source order with respect to M an *out-of-order IR operation* at breakpoint O. There are two types of out-of-order IR operations:

- If N executes before M in the canonical execution order but is scheduled to complete execution after O, then N is a *roll forward operation* at breakpoint O, denoted $RFOp(N,O)$.
- If N executes after M in the canonical execution order but is scheduled to complete execution before O, then N is a *roll back operation* at breakpoint O, denoted $RBOp(N,O)$.

Using the IR annotations, the debugger can determine which operations are executed out of order. Let I be the instruction causing an asynchronous break, $M = IR(I)$ be the IR breakpoint operation, $O = Offset(I)$ be the position of I in the basic block (the object breakpoint), and N an IR operation in the same block as M:

- $[(Seq(N) < Seq(M)) \wedge (Sched(N) \geq O)] \Rightarrow RFOp(N,O)$
- $[(Seq(N) > Seq(M)) \wedge (Sched(N) < O)] \Rightarrow RBOp(N,O)$

We extend the above terminology to machine operations. A machine operation I that was generated for an out-of-order IR operation $IR(I)$ is an *out-of-order machine operation.* If $IR(I)$ is a roll forward (roll back) operation at an object breakpoint O, then I is a *roll forward (roll back) machine operation* at breakpoint O.

5.3 Endangered variables

Having discovered which operations are scheduled for out of order execution, the debugger must then determine how the expected source level state has been affected. Source level state is affected by assignment and function call operations. Therefore, if such operations are out-of-order at a breakpoint then the variables that they affect are endangered.

A source assignment can either assign directly to a variable by specifying the variable's identifier in its left hand side, e.g.

```
x = ...
```

or it can assign indirectly through an address expression, e.g.

```
*(p+4) = ...
```

In the former case, the debugger can precisely determine which storage location is being assigned to, since the run-time locations of variables are known. Thus, out-of-order direct assignments cause noncurrent variables. In the latter case, however, the variable assigned to is not explicit in the expression, but rather depends on the value of the address expression. The debugger must be able to recover the address expression value to determine the run-time location affected. In the case of function calls, the debugger cannot determine which memory locations have been accessed. Thus, out-of-order indirect assignments or function calls cause suspect variables.

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

5.4 Recovery

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

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

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

As has been noted in earlier work[14], roll forward variables are easier to recover than roll back variables. Therefore, our debugger implements the following strategy. First, the debugger attempts address recovery of roll forward and roll back indirect assignments. Since address recovery may make suspect variables either noncurrent or current, the status of suspect variables is re-evaluated after this recovery is performed. Then, the debugger attempts assignment recovery of roll forward assignments, possibly enabling the debugger to provide the expected values of some roll forward variables.

Address recovery

Let A be an indirect assignment expression, and let I be the store instruction generated for A. A's address expression is recoverable at an object breakpoint O if the values in I's address registers have been computed but not subsequently overwritten:

$$\forall R \in AddressRegs(I) : (LastDef_I(R) < O) \wedge (NextDef_I(R) \geq O)$$

Assignment recovery

Let A be the assignment operation in the IR expression $x = E$, where E is an expression computed by the IR operation N. If x is a memory variable, the instruction generated for A is a store instruction that stores the value computed by N. If x is a register variable, the compiler generates either a register move operation that copies the value computed by N into the register assigned to x, or the last instruction generated for N targets the register assigned to x. Let B be the breakpoint $< S, O >$ where S is in the same block as A. If A is a roll forward assignment operation at B, the value assigned by A is recoverable if the value computed by N is recoverable. If A assigns x's expected value at B, and N's value is recoverable, then the debugger can provide x's expected value.

On the other hand, if x is an indirect address expression (e.g., *(p + 4)), and hence A is an indirect assignment operation, the address expression of x must also be recoverable for A to be recoverable.

We take a simple approach to recovering the value of an IR operation N. Let I be the last instruction generated for N, and let O be the object breakpoint. N's value is reproducible from the run-time state if either

1. $(Offset(I) < O) \wedge (NextDef_I(DestReg(I)) \geq O)$ or
2. $\forall R \in SourceRegs(I) : (LastDef_I(R) < O) \wedge (NextDef_I(R) \geq O)$ and I is safe to execute.

In the first case, IR operation N has executed and its result is available in a register. In the second case, the last instruction of N has not executed but can be executed (actually interpreted by the debugger) since its source registers are available. Hence we can compute I's value from I's source registers and obtain N's value. However, the debugger must be prepared to handle the case that I may fault. The debugger cannot perform this computation if I is a function call instruction, or if I is a load instruction, since memory locations may be noncurrent or suspect.

Note that we can extend the above scheme to recover values from more than one instruction by tracing instruction dependences back further. However, we are interested in how well we can do with a simple scheme.

6 The effects of coalescing

Coalescing or subsumption [6] is an optimization that eliminates copy operations by assigning the same physical register to the source and destination operands of a copy operation. Coalescing affects debugging when the eliminated copy operation is a source definition $I_V(S)$ of a variable V. E.g., consider the source and object codes depicted in Figure 5. Part (a) of this figure shows the source code, while parts (b) and (c) show the object code before and after register assignment respectively. In this example, $I_y(S_1) = I_1$ and $I_x(S_2) = I_2$ before register assignment. Assume that the live ranges of x and y do not interfere. The register assigner coalesces x and y, eliminating the copy operation I_2 and assigning the same register r to both x and y ($r = R(x) = R(y)$).

S_1: y = ...	I_1: y ← ...	I_1: r ← ...
...	...	...
S_2: x = y;	I_2: x ← y	
(a)	(b)	(c)

Figure 5: Effects of register subsumption

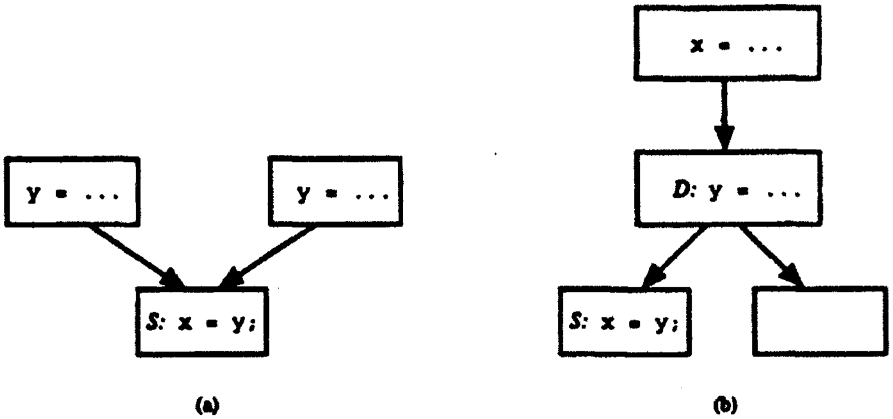


Figure 6: Global register coalescing

To capture the effects of coalescing, we consider the source definition corresponding to S_2 as being executed by I_1, at the same time as S_1, so that $I_x(S_2) = I_y(S_1) = I_1$ and $\{x,y\} \subseteq EvictReachKill(I_1)$. The IR annotations are adjusted to reflect ths change. If execution stops somewhere between S_1 and S_2 in the source, but after I_1 in the object, x and y are both resident in r. The actual value of y is the value assigned by S_1, while the actual value of x is the value assigned by S_2. Hence, y is current and x is noncurrent. If execution stops after S_2 in the source, but after I_1 in the object, both x and y are current.

A less precise model is to consider S_2 as an eliminated assignment. Thus at a breakpoint after S_2 in the source and after I_1 in the object, x will be detected as nonresident since I_1 is a reaching evicting definition of x. However, this is conservative since r contains the value that would have been assigned to x by S_2, which is x's expected value.

In general, when coalescing eliminates an instruction $I = I_V(S)$, the source definition instruction $I_V(S)$ is changed to an earlier definition of $R(V)$ that reaches $pre(I)$. Figure 5 illustrates the simple case where the earlier definition of $R(V)$ is within the same basic block as the eliminated copy operation. However, no earlier definition of $R(V)$ may exist within the same basic block as the eliminated instruction. Figure 6 illustrates other cases that can occur. In this figure, coalescing eliminates the copy x = y. In Figure 6 (a) S post-dominates all reaching definitions of $R(V)$. Thus we may consider all reaching definitions as source definitions of S, adjusting $I_x(S)$ accordingly. Note that this moves the definition of x into different basic blocks and results in multiple source definitions. This has ramifications on the noncurrency detection algorithms which now have to address global code movement. If there exists a reaching definition D that is not post-dominated by S, as shown in Figure 6(b), D cannot be considered a source definition of x.

To avoid these two problems all together, we model the source definition of S in both cases to be the *pre-amble* instruction of S's basic block. In other words, if there does not exist a prior definition of $R(V)$ in the basic block, the pre-amble instruction is used.

7 Prior work

Prior work on debugging optimized code has mostly assumed that variables are always accessible in a run-time storage location. With the exception of the DOC [13] and CXdb [5] debuggers, previous research has overlooked the problem of nonresident variables. In [1], we defined the problem of nonresident variables and investigated approaches to detecting when a variable's assigned register holds a value of that variable.

Hennessy's work [14] and later refinements of Wall et. al [20] deal with detection and recovery of noncurrent variables in the presence of local optimizations and code generation using DAGs. The model of [14] assumes that all variables have home locations in memory and does not consider values held in registers. The code generator is cast at the intermediate representation level (before variables are bound to machine resources) and operates without reference to any specific features of the target machine (like load delays or a horizontal instruction format). However, code generators in modern compilers are typically tightly tuned to the instruction-level parallelism and storage hierarchy of the target architecture, as modern architectures rely on instruction scheduling and register allocation for performance.

In [14], Hennessy introduced the concept of endangered and noncurrent variables and proposed algorithms to recover their expected values. However, the source language is a subset of Pascal that does not include pointers. The optimizations performed by the Pascal compiler are at the intermediate representation level, and the algorithms described in [14] (and corrected in [20]) deal with noncurrency due to dead code elimination, and re-ordering introduced by common subexpressions that are assigned to variables. The effects of code generation techniques like instruction scheduling or register allocation are not considered. Furthermore, the recovery algorithms only recover values from memory and do not consider partially computed results that are available in registers. Our recovery algorithm considers values computed by individual instructions and held in physical registers.

The only type of endangered variable that is considered in [14] are *path-endangered* variables. A path-endangered variable is a variable that is noncurrent along (at least) one path through the program leading to the breakpoint B. Only global (inter-block) optimizations can cause such a situation. This paper considers endangered variables that occur in the presence of local (intra-block) optimizations: even if all global optimization is suppressed, reordering of load and store operations can cause endangered variables.

DOC [13] is a prototype debugger, designed to demonstrate the feasibility of debugging optimized code. The optimizations that DOC handles include instruction scheduling and register allocation. It is the first debugger system that we are aware off that deals with multiple storage locations for variables. DOC tackles this problem by computing the location and currency of a variable in the compiler; this information is then passed to the debugger in *range records.* A range record provides the debugger with the storage location assigned to a variable or, in the case that a variable has been eliminated and replaced with a constant, the variable's value. The range of object code addresses during which a range record is valid is also specified in the record. The description in [13] states that range records of a register promoted variable span only the variable's live range, but the paper does not indicate how the debugger responds to a user query of a register promoted variable at breakpoints outside of the variable's live range, e.g., at a breakpoint before the variable has been defined or after its last use. According to Copperman and McDowell [12], DOC reports a variable as inaccessible after its last use.

DOC simplifies the problem of noncurrent variables by allowing only breakpoints that are set in the source by the user. That is, the DOC approach to detecting noncurrent variables assumes predetermined breakpoint locations in the object code that correspond to source statement entry points. If an exception occurs at an instruction I that is generated for a statement S in the source, and I does not correspond to S's statement entry point, then DOC may not precisely detect which variables are noncurrent. Hence DOC does not model what happens inside a statement and supports only synchronous breakpoints. Re-ordering of function calls is not an issue in DOC, since the compiler does not perform such a transformation. The user is warned when an assignment through a pointer has occurred out of order, but no recovery (of the address or the value) is attempted.

Copperman and McDowell (in [11] and [9]) propose a method of detecting noncurrent variables in the presence of global transformations that re-order or eliminate operations. Their method uses global data flow analysis techniques and is based on comparing variable assignments that reach a breakpoint in the source with those that reach the breakpoint in the object. Their method assumes predetermined breakpoint locations at instructions that correspond to source assignment operations. Thus, like the DOC approach, their method does not handle asynchronous breaks at arbitrary points in the object code. They do not consider nonresident variables, nor do they consider recovery of endangered variables. Moreover, they do not consider the problems caused by function calls that are executed out of order, and they do not consider practical aspects like the impact of the undefined evaluation order of assignments within a statement[10]. There is no implementation of the proposed methods. Bemmerl [3] also proposes a method of detecting noncurrent variables in the presence of global transformations, using global data flow analysis.

In another work, Copperman and McDowell point out that Hennessy's model does not consider values held in registers [12]. However, they still consider the problem at the intermediate representation level without reference to registers. They suggest that allowing multiple assignments to a variable within a basic block addresses this problem. However, this does not handle the case of nonresidence caused by register re-use.

Other work in debugging optimized code addresses the *code location* problem: how to construct a mapping between breakpoints in the source and object code. Zellweger [21] deals with these mappings in the presence of control flow optimizations such as function inlining and cross jumping. Our work deals with retrieving source values at breakpoints and is orthogonal to the code location problem. Also, we do not consider the mechanics of how data or control breakpoints are set; see [15] or [19] for some options.

Some debugger systems try to avoid the problem of dealing with optimized code by leaving it to the user to sort things out. Instead of trying to present the user with a view f the data space that matches the source code, these systems provide either raw information (e.g., the current state) to the user or convey the results of optimizations (e.g., statements eliminated). CXdb [5] is a recent example of such a debugger. It animates the execution of a source program by highlighting the source expression(s) that is (are) executed. Based on this information, the user can determine how source values are affected by optimizations. Such a visual annotation is useful if the user single-steps through the code; for each step, the current source unit is illuminated. However, if the program is run until a break occurs, CXdb tells the user which source unit the break occurred at but fails to provide any history or context. Although CXdb does not detect endangered variables, it detects and warns the user of nonresident variables. CXdb uses the live range approach to detecting nonresident variables [17].

8 Experimental results

To compare the problems caused by nonresidency with those caused by noncurrency, and to evaluate the effectiveness of using data-flow analysis, we have implemented our approach using the iWarp C compiler. This compiler is based on the PCC2 compiler from AT&T that has been enhanced with global optimizations. The target machine is the iWarp processor, an LIW machine, with a large number of registers.

The set of programs for this evaluation consists of three C programs from the SPEC integer suite (li, espresso and eqntott)[18], and twelve programs from the Numerical Recipes collection (balanc, bessi, bessj, elmhes, fft, gaussj, hqr, jacobi, ludcmp, meo, svdcmp, and tred2)[16]. Each program from the latter set consists of a small number of functions, and the results do not vary significantly between the individual programs. Therefore, we average the results from the Numerical Recipes programs. Each figure in this section contains four charts, one for each SPEC program, and one for the averaged results of the Numerical Recipes programs (referred to in the figures as "recipes").

8.1 Compiler framework

The iWarp C compiler (pre-release version 2.7) performs local code compaction for the iWarp processor. iWarp is an LIW machine, with 128 registers, of which 94 are available to the compiler. In a single cycle, the iWarp can execute a floating point multiplication, a floating point addition, 2 integer operations or memory accesses, as well as a loop termination test [4].

The compiler performs global register allocation and assignment, branch optimizations, unreachable code elimination, common subexpression elimination, value propagation, constant folding, and instruction scheduling. Common subexpression elimination, value propagation, and instruction scheduling are performed at the basic block level. Function calls do not delimit basic blocks, and scheduling may cause function calls to be reordered with respect to other operations.

Local variables that are not aliased are promoted to a register by the optimizer. These variables along with compiler temporaries are allocated registers from an infinite pool of virtual registers. Virtual registers are assigned physical registers after code scheduling, using graph coloring. Live ranges are not split, and promoted variables have *no* home locations in memory. Therefore, a promoted variable resides in its assigned register throughout its live range. The assigner attempts to assign caller saved registers to live ranges that do not span function calls. Register subsumption or coalescing[6] is performed to minimize the number of register moves. This optimization assigns the same register to two virtual registers whose live ranges do not conflict, but are connected by a register move. Due to the large number of registers in our machine, none of the benchmarks requires live ranges to be spilled to memory.

The code scheduler and register assigner of the iWarp C compiler create two problems for a debugger. First, because of code scheduling, the debugger must detect which assignments and function call operations have executed (or not executed) out of order with respect to the source stopping point, and how source level values have been affected. Second, because registers may be reassigned, the debugger must detect which of the promoted variables are resident in their assigned registers at a breakpoint. A promoted variable may have been evicted either because its assigned register was re-assigned to another variable or killed by a function call (if the variable was assigned a caller saved register). Since promoted variables do not have home locations in memory, recovery of their values is difficult.

The code generator was modified to emit the information and IR annotations described in Sections 4 and 5. The IR of assignments that have been eliminated due to coalescing are annotated as described in Section 6. Register assignments are recorded in a table that maps register assigned variables to physical registers. Note that the object code is identical to the default code produced. The debugger is totally non-intrusive and does not effect any changes in the code, the memory layout, or any other aspect of the object program. The algorithms for detecting nonresident and endangered variables can be performed either in the compiler for the debugger, or in the debugger. For our experiments, these algorithms were implemented in a separate program that gathers statistics.

The information presented in this section is collected by analyzing the code generator output for all possible breakpoint locations.

8.2 Nonresident variables

Our experiments compare the effects of data-flow analysis techniques for finding evicted variables to a simple approach, which tracks a variable's location only during the variable's live range. We also investigate the effects of using reaching analysis to find uninitialized variables at breakpoints.

We look at the following four approaches to detecting nonresident variables:

1. A variable is resident during its live range only.

2. A variable is resident during its live range only, and reaching analysis detects uninitialized variables.

3. A variable is resident wherever it is not evicted.

4. A variables is resident wherever it is not evicted, and reaching analysis detects uninitialized variables.

In the first approach, the debugger is successful in recovering a variable V's value if a breakpoint occurs inside V's live range. The second approach augments the first approach by using reaching analysis to detect uninitialized variables. In this approach, the debugger is successful in recovering a variable V's value if the breakpoint occurs inside V's live range. At a source breakpoint where V is not reaching, the debugger reports V as uninitialized. This reduces the number of variables reported nonresident, since uninitialized variables are not reported as nonresident.

The third approach uses the data-flow analysis technique described in Section 4.3 to find all points where a variable V's assigned register $R(V)$ contains V's value. However, at a breakpoint, an *uninitialized* variable V is reported as nonresident if an evicting definition of V reaches the object breakpoint location. Recall that such an evicting definition is not a source definition of V. The fourth approach adds reaching analysis to the third approach to detect uninitialized variables and therefore only reports variables as evicted if there is a source definition that reaches the breakpoint. This approach is the most aggressive and least conservative of the four.

Figure 7 compares the percentage of breakpoints that contain endangered variables with the percentage of breakpoints that contain nonresident variables using each of the above four approaches. The first column shows the percentage of instructions for which there are endangered variables, i.e., we map each instruction to its source statement and determine if there are any assignments or function calls that have been scheduled out of source order. The other four columns show the percentage of breakpoints with nonresident variables. For each approach listed above, we compute the number of instructions for which there is at least one nonresident variable. The raw instruction counts are normalized by the total number of instructions in the program. These metrics do not reflect the number of noncurrent or nonresident variables at a breakpoint, nor do they consider the dynamic behavior of programs. Note that these metrics also assume queries to all variables to be equally likely.

The breakpoint model used is one that considers all instructions as potential breakpoints and corresponds to the situation where the user can interrupt program execution at an arbitrary point in the object. We also considered a breakpoint model where only instruction that can generate a fault are considered as breakpoints. Note that these models capture the state of the user program for each machine instruction in the object code and not for each source-level statement in the user program.

The results in Figure 7 for these benchmarks show that there are significantly more breakpoints containing nonresident variables than there are breakpoints with endangered variables. Thus nonresident variables are potentially a more serious problem than endangered variables when debugging optimized code.

A comparison of the results of using the first approach with the results of using the third approach, as well as a comparison of the second approach with the fourth approach show that using data-flow analysis to detect evicted variables significantly increases the chances of recovering a register assigned variable's value.

The effects of using reaching analysis can be measured by comparing the results of using the first and second approach and by comparing the results of using the third and fourth approach. Our results show that using reaching analysis decreases the number of breakpoints with nonresident variables for both the live range approach and the evicted data-flow analysis approach.

Benchmark	Average number of out-of-order machine operations per breakpoint	Average number of statements per basic block
li	0.8	1.2
espresso	0.6	1.6
eqntott	0.7	1.5
recipes	1.1	2.1

Table 4: Effect of scheduling

Another way to evaluate the effectiveness of the various techniques for detecting nonresident variables is to look at the number of variables that are nonresident at a given breakpoint. Figure 8 presents this information using the same breakpoint model as discussed above. The leftmost column of each graph in this figure shows the average number of register assigned variables; this number presents a baseline for comparison. A naive debugger that does not handle register assigned variables at all has to report all of these variables as inaccessible, so this number is the upper bound on nonresident variables for each program. The rightmost columns of these graphs depict the average number of variables that the different strategies discussed above report as inaccessible. The results from this figure again illustrate the effectiveness of using data-flow analysis to detect nonresident and uninitialized variables.

To minimize saving and restoring of callee-saved registers in function prologues and epilogues, our compiler attempts to assign caller-saved registers to live ranges that do not span function calls. However, the programs from the SPEC suite contain a large number of function calls. Since function calls evict variables that have been assigned caller-saved registers, assigning caller-saved registers to variables could affect the number of variables that are evicted. We investigated the effects of assigning only callee-saved registers to variables, and our results showed that the choice of assigning caller-saved vs. callee-saved registers to variables does effect the number of nonresident variables.

8.3 Endangered variables

Clearly, the number of endangered variables at a breakpoint is a function of the program *and* the code generator. (A code generator without reordering does not cause any endangered variables). We performed a simple sanity check to ensure that indeed the code generator interleaved the execution of multiple statements where appropriate. Table 4 shows the average number of out-of-order machine operations at each breakpoint. Although there are on the average only a few statements per basic block, the compiler interleaves code quite frequently when the opportunity exists. (For comparison, the average number of statements per basic block in the source programs is shown in the right column. Basic blocks with just a single statements contain no endangered variables due to reordering, and no out-of-order operations. For li, only 1/5 of the basic blocks contains more than one statement on average, but nevertheless, *on average*, 0.8 operations are out-of-order.)

Figure 9 shows the number of breakpoints that contain out-of-order operations as a percentage of the total number of possible breakpoints (i.e., machine operations) in each benchmark program. Only out-of-order operations that cause noncurrent or suspect variables are included in Figure 9. The percentage of breakpoints with such out of order operations ranges from approximately 9% (li) to 21% (recipes).

Columns 2 to 6 of Figure 9 show what kind of operation causes endangered variables. Note that at a breakpoint B there can be more than one out-of-order operation, so the sum of the percentages of these columns may be more than the total shown by the first column. At the majority of breakpoints with endangered variables, an assignment to a register assigned variable is out-of-order and causes this variable to be noncurrent. The exception to this is li, which contains a large number of function calls, and where out-of-order function calls are the main cause of endangered variables. The low number of out-of-order assignments to local variables in memory is due to the success of the register allocator in assigning registers to local variables. Out-of-order function calls are the second largest contributor for the other SPEC benchmarks. This shows that if a scheduler operates on basic blocks that are not delimited by function calls, a significant number of function calls may be out-of-order at a breakpoint. The numerical recipes benchmark consists mainly of computations on arrays, with relatively few function calls. Therefore this benchmark has a higher percentage of breakpoints with endangered variables caused by out-of-order pointer assignments and a low percentage of breakpoints with out-of-order function calls. In the SPEC programs, out of order assignments through pointers occur at about 2% of all breakpoints, while in the numerical recipes such assignments occur at about 4% of the breakpoints.

Out-of-order function calls are problematic in that the debugger cannot determine a function call's effect on the program state in memory. Thus a debugger must report a memory variable as suspect at a breakpoint B, if there exists an out-of-order function call at B. Out-of-order indirect assignments are similarly problematic if the debugger cannot recover the address of the assigned memory location. The variables effected by direct assignments can be precisely determined by the debugger, and hence these types of assignments do not cause suspect variables.

Figure 10 depicts the effect of address recovery. Since we may not be able to count the number of variables that are suspect (all variables on the heap may be included, and their number can only be determined at runtime), we have to use the number of breakpoints with suspect (or noncurrent) variables to illustrate the effect of recovery.

The first column in the charts of Figure 10 shows the number of breakpoints with either noncurrent or suspect variables (from Figure 9) and is repeated here for reference. The second column in the charts shows the percentage of breakpoints with out-of-order direct assignments. These are breakpoints with noncurrent variables. The third column shows the percentage of breakpoints with out-of-order indirect assignments or function calls. These are breakpoints with suspect variables. This data indicates that there are more breakpoints with noncurrent variables than there are breakpoints with suspect variables. However, there are still a significant number of breakpoints with suspect variables. In the case of li there are almost as many breakpoints with suspect variables as there are breakpoints with noncurrent variables. Furthermore, the number of suspect variables can be quite large at such a breakpoint. Therefore, any success in identifying the effect of the indirect assignments improves the quality of the debugger, since such recovery of the addresses allows the debugger to provide the user with more precise responses.

The last two columns in the graphs of Figure 10 show the effects of recovering addresses of out-of-order indirect assignments. Address recovery reduces the number of breakpoints with suspect variables to approximately the number of breakpoints with out-of-order function calls, which is the limit on how well address recovery can do.

Figure 11 breaks down the percentage of breakpoints with out-of-order assignments and function calls into breakpoints with roll forward and roll back operations. Again, the first column is repeated from Figure 9. We see that there are more breakpoints with roll forward operations (Column 2) than ones with roll back operations (Column 3). This is good news for the user interested in obtaining the expected value of a variable: for a roll forward operation M, either the debugger may attempt to recover the value using the approach described in Section 5.4, or the user may set a breakpoint at the point where M is eventually computed to recover the value computed by M.

This figure also breaks down the breakpoints with roll forward/roll back operations according to

Benchmark	% Breakpoints with out-of-order function calls or assignments	% Breakpoints with out-of-order function calls or assignments that are not recoverable
li	8.8	6.1
espresso	12.9	9.0
eqntott	14.5	10.7
recipes	20.8	16.2

Table 5: Effects of recovery

whether they cause suspect or noncurrent variables. Column 7 (the right most column) shows the percentage of breakpoints with a roll back function call or indirect assignment. The debugger may be limited in allowing variable inspection at such a breakpoint, since some or all variables in memory may be suspect. But almost all roll back operations are direct assignment operations; roll back function calls or indirect assignments are very rare. That is, almost all suspect variables are caused by roll forward operations, and those can be resolved at runtime by the debugger.

A user of a debugger may want to know how many variables are suspect or noncurrent at the average breakpoint. As explained above, this number cannot be obtained using the static analysis tools developed, since the number of suspect memory variables is dynamic (and may differ from one run of a program to another). However, for other out-of-order operations, a better breakdown is possible and is presented in Figure 12. This figure shows the average number of roll forward operations per breakpoints with noncurrent or suspect variables, for the different types of assignments and for function calls. Columns 1, 3, 5, 7, and 10 show the average number of roll forward assignments to registers, locals (direct), and globals (direct), of indirect assignments, and of roll forward function calls, respectively. Also shown is the average number of roll-forward assignments whose values are recoverable using our simple approach to assignment recovery (Columns 2, 4, 6, 9). In many of the cases, most of the roll forward assignments are recoverable. Note that to recover an out-of-order indirect assignment, both the address expression as well as the assigned value must be recoverable. We see that address recovery is successful in recovering the addresses of most roll forward indirect assignments (Column 8), but there are some for which only the the address expressions can be recovered (compare Column 8 with Column 9).

Address recovery is also successful in recovering the addresses of the majority of roll back indirect assignments. However, these results were not shown because roll back indirect assignments are so rare that their effects do not show in the graph (see Figure 11 for the overall frequency).

In the case of roll forward operations that are direct assignments or indirect assignments with recoverable addresses, the numbers in this graph are also upper bounds on the number of roll forward variables that are caused by these assignments, since each assignment can assign to only one variable (but more than one may assign to the same variable).

In summary, there exist a noticeable number of breakpoints with endangered variables (between 9% and 20% in our suite), but their number is no obstacle to symbolic debugging. Furthermore, even a simple recovery scheme as described in this paper reduces the number of breakpoints with endangered variables. Table 5 presents the bottom line and shows how recovery effects the number of breakpoints where the debugger can provide the expected values of source variables.

9 Conclusions

Global register allocation/assignment and instruction scheduling are important aspects of code generation and a large number of modern compilers include some form of these optimizations. Register allocation/assignment causes nonresident variables by re-assigning registers to many variables. Instruction scheduling causes endangered variables by re-ordering the execution of side effecting source operations. There exist two mutually exclusive classes of endangered variables: those for which the debugger can deduce that the actual value is not the expected value, and those for which the debugger is not sure. A debugger that wants to exhibit truthful behavior must detect all endangered and nonresident variables and warn the user.

Prior work in debugging of optimized code has concentrated mainly on the problem of detecting and recovering endangered variables and has either ignored the problem of evicted variables by assuming that a variable is always resident in a storage location, or used a simplistic approach to tracking variable locations. However, evicted variables are a serious problem for a symbolic debugger if the compiler optimizations include global register allocation and assignment (as is commonly done in modern compilers today). Our results indicate that evicted variables are far more serious problem than endangered variables. In our sample programs a large number of breakpoints have evicted variables.

To detect evicted variables, it is necessary to consider the data-flow at the level of machine instructions since it is at this level that register re-use occurs Thus a detailed model of the machine resources is required. Debugger models proposed in previous studies are at a higher level than the machine level and are not sufficient for detecting evicted variables.

Detecting evicted variables requires analysis *for the debugger*. Our results show that at an approach that uses compiler collected data-flow information (e.g., live range information for register allocation) to track a variable's location is conservative. The impact of data-flow analysis on the number of resident variables is significant, and our results clearly show that performing data-flow analysis to detect evicted and uninitialized variables increases the number of variables that are correctly reported by the debugger.

Detecting noncurrent variables caused by instruction scheduling requires accurate modeling of source and object execution orders. When debugging a realistic language such as C, out-of-order pointer assignments and function calls may inhibit the debugger from precisely detecting which variables are noncurrent. Our results show that the impact of out-of-order function calls can be significant.

A simple recovery scheme that rolls forward a single instruction is effective for recovering the values assigned by out-of-order assignments and can reduce the number of variables for which the debugger cannot report the expected value. Further work is required to investigate whether better results can be obtained by rolling forward through more instructions. Recovering address values from the physical registers can reduce the number of out-of-order assignments that cause suspect variables. By recovering the address, the debugger can determine the effect of the indirect assignment on the programmer-visible state of the machine. Both techniques are effective and reduce the number of breakpoints with endangered variables.

References

[1] A. Adl-Tabatabai and T. Gross. Evicted variables and symbolic debugging of optimized code. In *Proc. 20th POPL Conf.*,pages 371–383. ACM, January 1993.

[2] A. V. Aho, R. Sethi, and Ullman J. D. *Compilers*. Addison-Wesley, 1986.

[3] T. Bemmerl and R. Wismueller. Quellcode debugging von global optimierten programmen. Presented at 1992 Dagstuhl Seminar, Feb. 1992. (in German).

[4] 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 *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[5] G. Brooks, G. Hansen, and S. Simmons. A new approach to debugging optimized code. In *Proc. SIGPLAN'92 Conf. on PLDI*, pages 1–11. ACM SIGPLAN, June 1992.

[6] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the SIGPLAN 1982 Symposium on Compiler Construction*, pages 98–105, June 1982. In SIGPLAN Notices, v. 17, n. 6.

[7] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12:501–535, Oct. 1990.

[8] M. Copperman. Debugging optimized code: Currentness determination with data flow. In *Proc. Supercomputer Debugging Workshop '92*, Dallas, October 1992. Los Alamos National Laboratory.

[9] M. Copperman. Debugging optimized code without being misled. Technical Report 92-01, UC Santa Cruz, May 1992.

[10] M. Copperman. Personal communication. 1992.

[11] M. Copperman and C. McDowell. Detecting unexpected data values in optimized code. Technical Report 90-56, UC Santa Cruz, October 1990.

[12] M. Copperman and C. McDowell. A further note on Hennessy's "Symbolic debugging of optimized code ". Technical Report 92-24, UC Santa Cruz, April 1992.

[13] D. S. Coutant, S. Meloy, and M. Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proc. SIGPLAN 1988 Conf. on PLDI*, pages 125–134. ACM, June 1988.

[14] J. L. Hennessy. Symbolic debugging of optimized code. *ACM Trans. on Programming Languages and Systems*, 4(3):323 – 344, July 1982.

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

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1991.

[17] S. Simmons. Personal communication. 1992.

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

[19] R. Wahbe. Efficient data breakpoints. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–212. October 1992.

[20] D. Wall, A. Srivastava, and F. Templin. A note on Hennessy's "Symbolic debugging of optimized code". *ACM Trans. on Programming Languages and Systems*, 7(1):176–181, January 1985.

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

[22] 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.

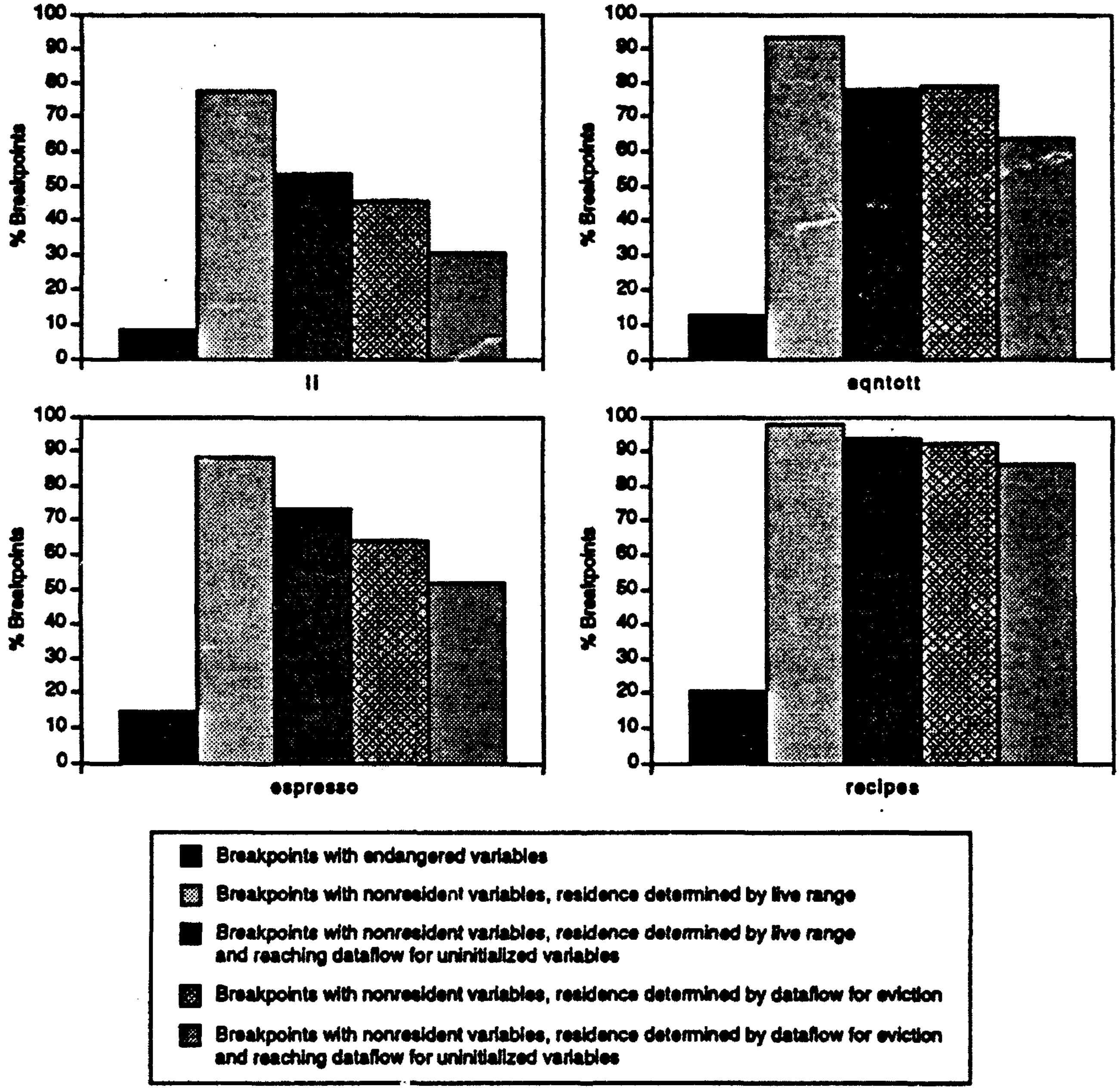


Figure 7: Breakpoints with noncurrent or nonresident variables

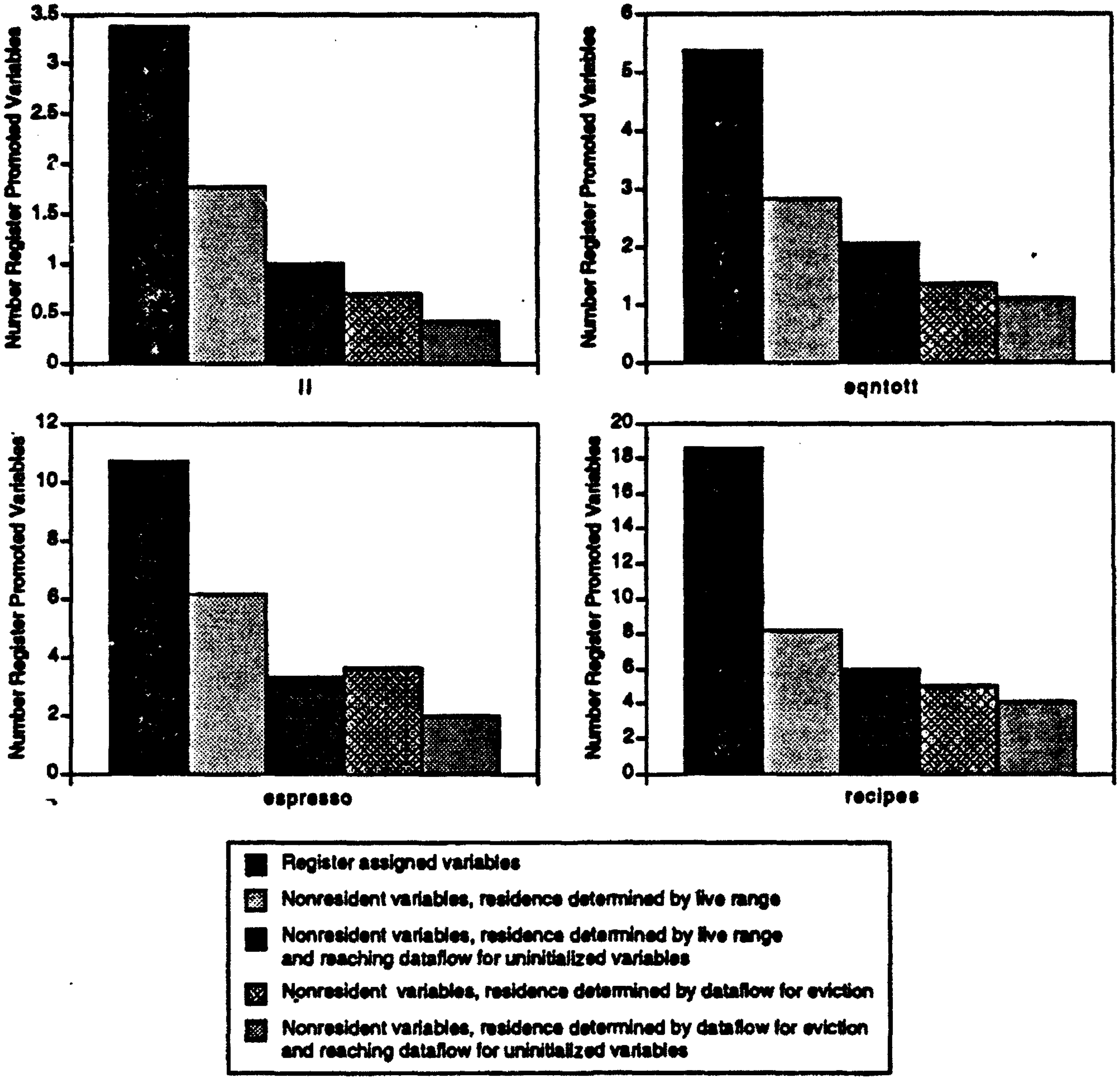


Figure 8: Avg. number of nonresident register assigned variables

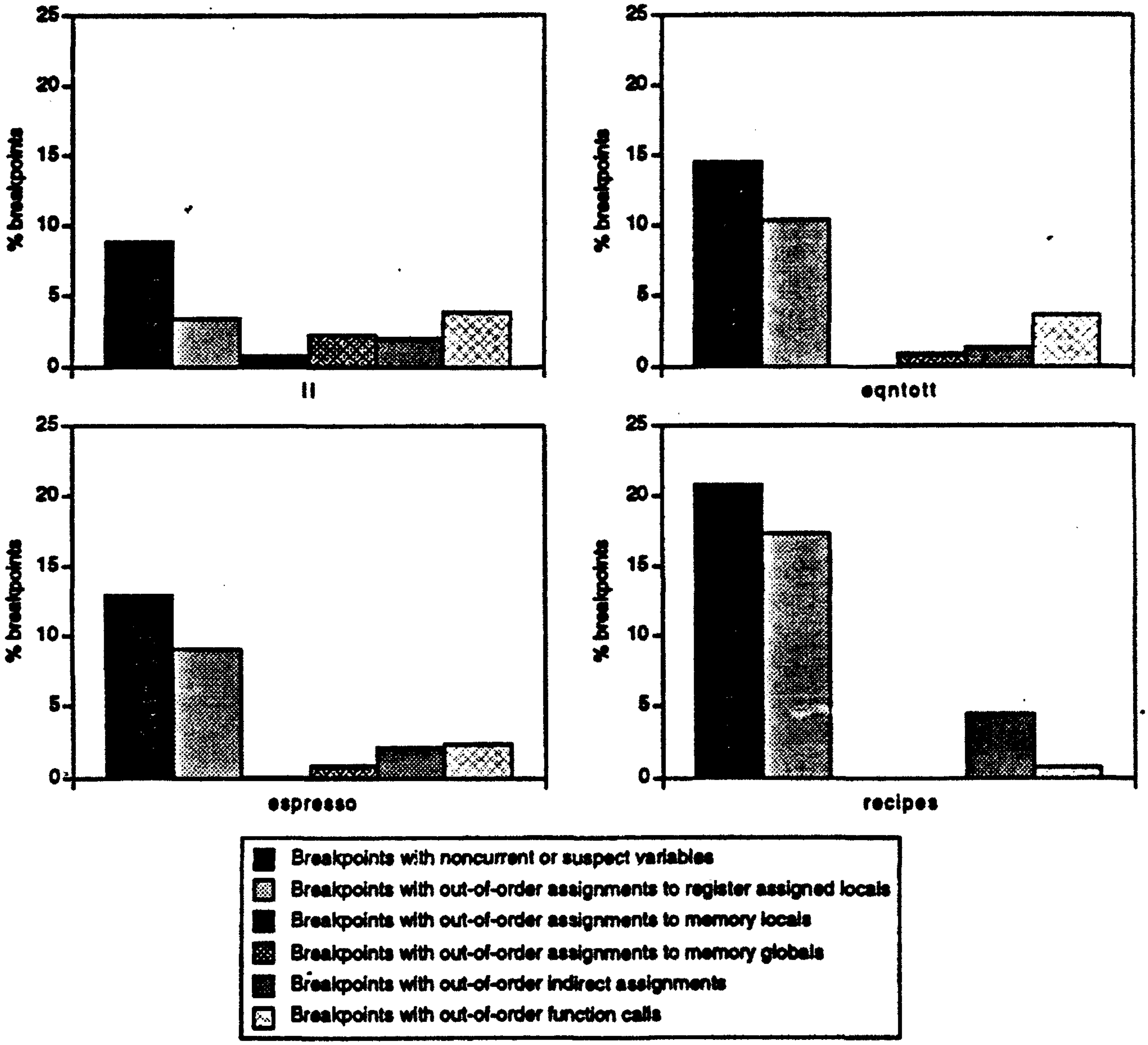


Figure 9: Breakpoints with operations causing endangered variables

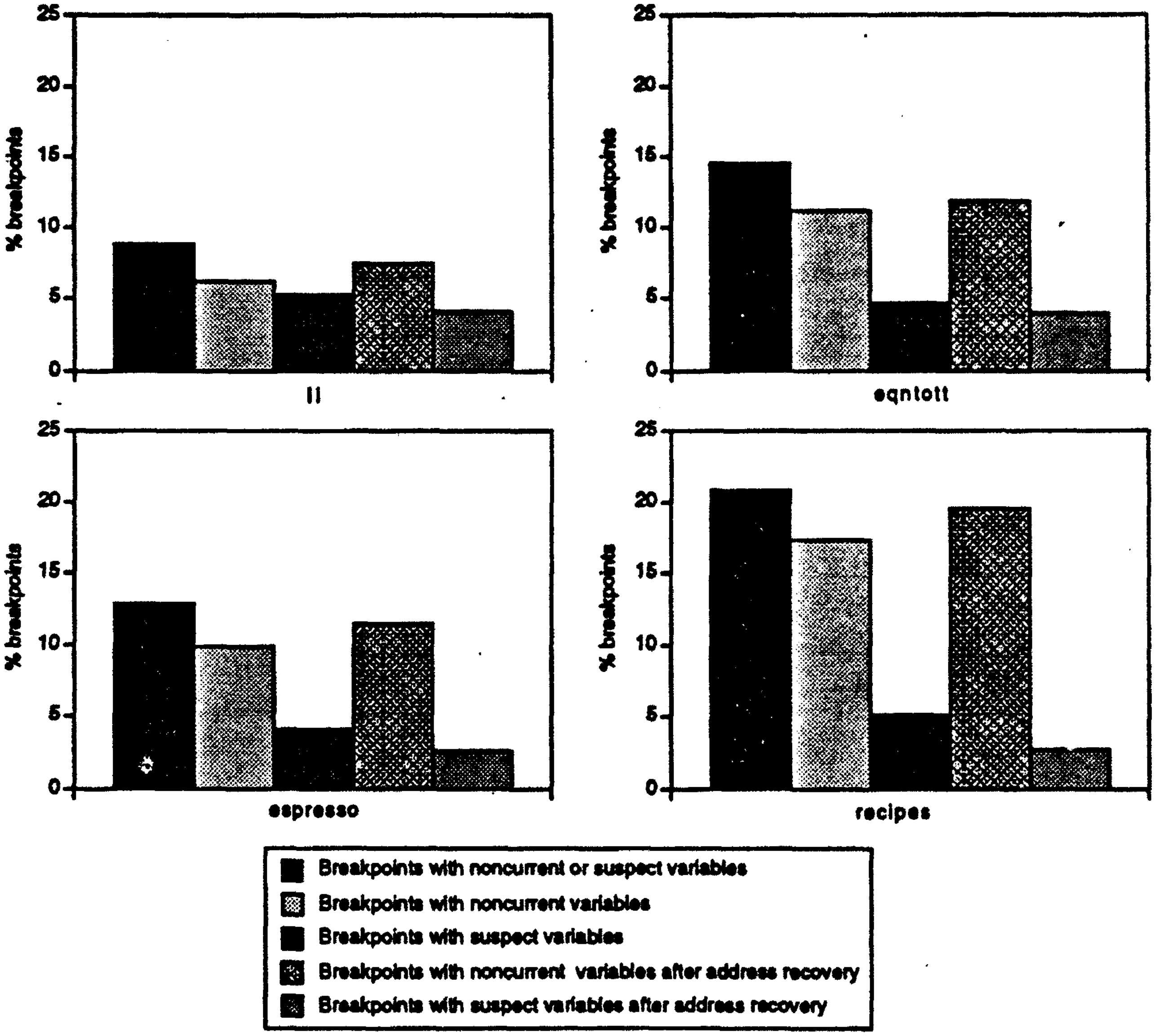


Figure 10: Breakpoints with suspect and noncurrent variables

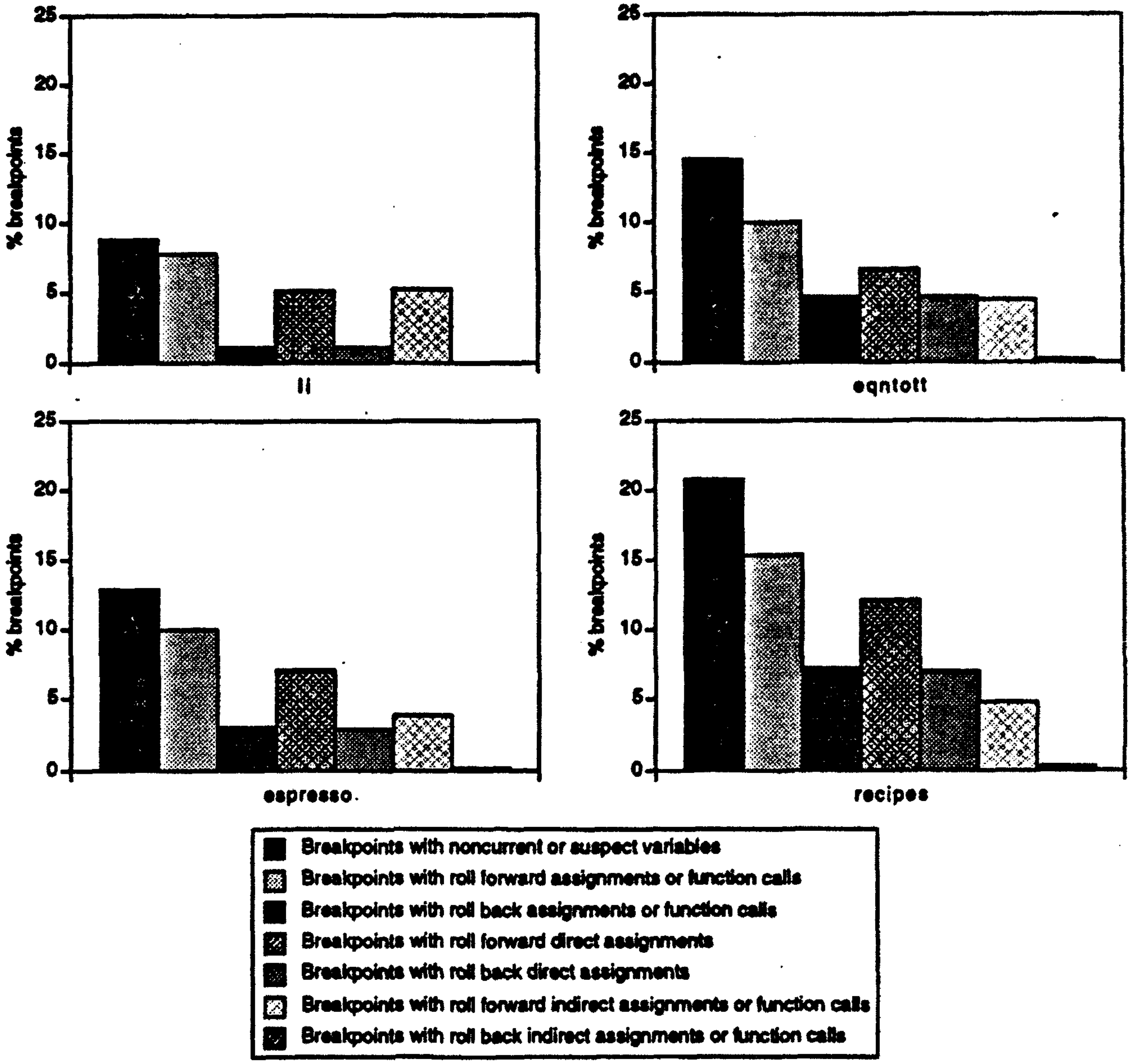


Figure 11: Breakpoints with roll forward and roll back operations

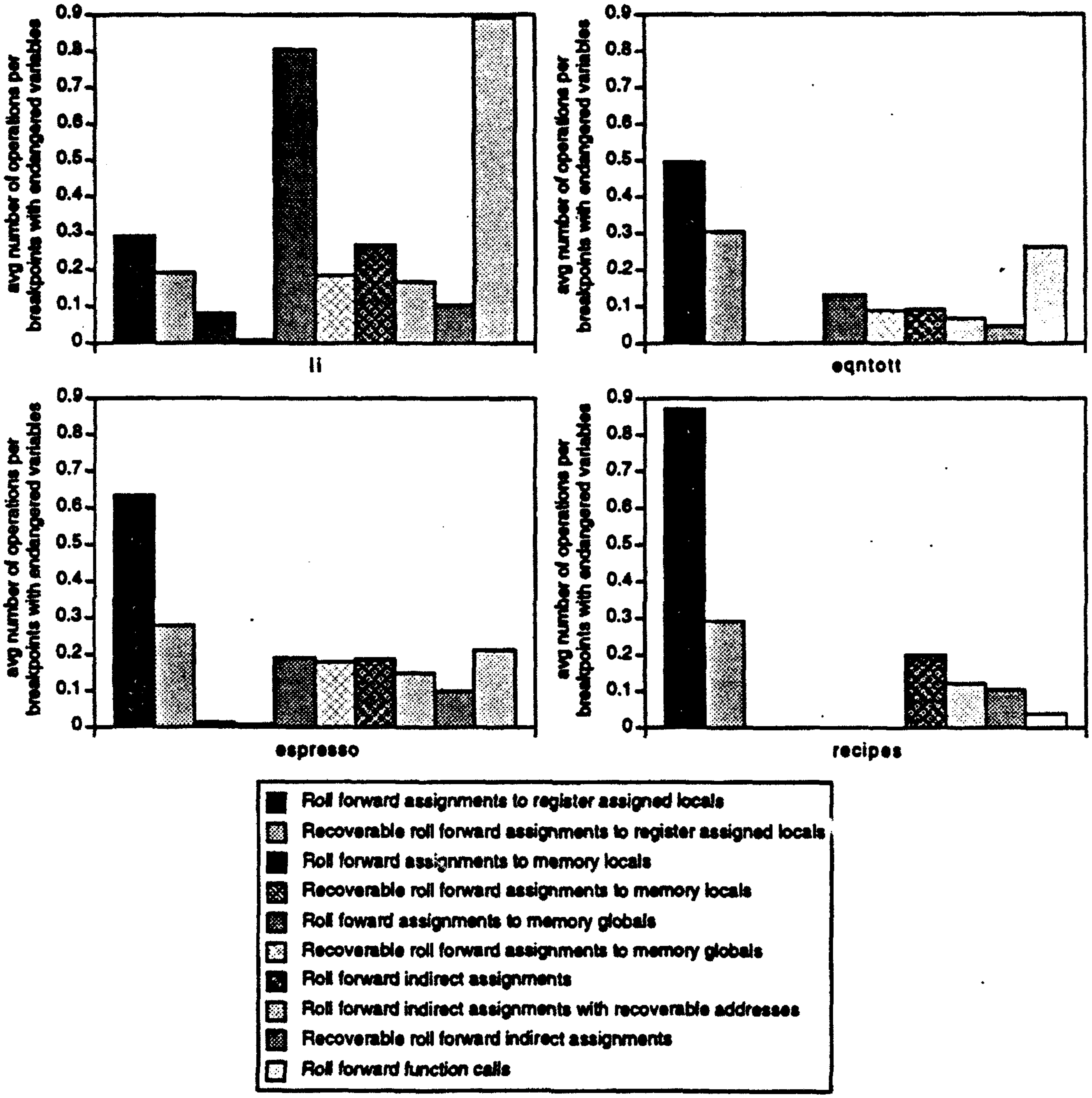


Figure 12: Avg. number of roll forward and recoverable operations