To appear in the
Proceedings of PLDI'93, ACM SIGPLAN/93 Conf. on Programming Language Design and Implementation, June 1993, Albuquerque, NM.

Detection and Recovery of Endangered Variables Caused by Instruction Scheduling

Ali-Reza Adl-Tabatabai and Thomas Gross

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Instruction scheduling re-orders and interleaves instruction sequences from different source statements. This impacts the task of a symbolic debugger, which attempts to present the user a picture of program execution that matches the source program. At a breakpoint B, if the value in the run-time location of a variable V may not correspond to the value the user expects V to have, then this variable is *endangered* at B. This paper describes an approach to detecting and recovering endangered variables caused by instruction scheduling. We measure the effects of instruction scheduling on a symbolic debugger's ability to recover source values at a breakpoint. This paper reports measurements for three C programs from the SPEC suite and a collection of programs from the Numerical Recipes, which have been compiled with a variant of a commercial C compiler.

1 Introduction

A debugger allows a user 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.

Optimizing compilers impact the task of the symbolic debugger by complicating the correspondence between the source code and object code. By re-ordering or eliminating the execution of source level expressions, optimizing compiler transformations complicate the mapping of breakpoints and values in the source code to those in the object code.

One important transformation that affects symbolic debugging is instruction scheduling. Instruction scheduling can increase the efficiency of modern processors with instruction-level parallelism by statically scheduling independent instructions for concurrent execution. However, scheduling changes the sequence in which source level expressions are computed by reordering or interleaving instruction sequences from different source statements. Consequently, at a breakpoint, the run-time value of an inspected variable may not correspond to the value expected in the source.

Consider the source and object codes shown in Figure 1. Variable `a` has been assigned register `R3`, while variables `b` and `p` have both been assigned register `R4`. Variables `c` and `d` have been assigned registers `R5` and `R6` respectively, and variables `f` and `g` both reside in memory.

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

Source code
(a)

```
I1:  load R1,f
I2:  load R2,g
I3:  store (R4),R3
I4:  fpmul R4,R5,R3
I5:  fpadd R6,R1,R2
```

Object code
(b)

Figure 1: Example

The first three columns of Table 1 show the correspondence between instructions and source expressions; it is easy to see how the instruction scheduler has interleaved and reordered the execution of instruction sequences from

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.

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

Table 1: Endangered variables at breakpoints in the code of Figure 1

different statements. For example, the assignment to `d` from statement S_1 occurs at instruction I_5, while the assignments to `b` and `*p` of statements S_2 and S_3 occur at instructions I_4 and I_3 respectively.

If a breakpoint occurs at instruction I_4 (e.g., due to a floating point exception), the debugger will report that execution has halted within statement S_2. At this breakpoint, the user expects `d` to have the value assigned by statement S_1. However, this assignment to `d` has not yet executed (it is executed at I_5), and the value in `d`'s run-time location (i.e., the value in `R6`) is the value from the last executed assignment to `d` prior to this block. Therefore, the value in `d`'s run-time location does not correspond to the value that the user expects `d` to have in the source, and `d` is said to be *noncurrent* at this breakpoint [8].

If there is a breakpoint at I_4, the assignment to `*p` of S_3 has executed prematurely at instruction I_3. Therefore, the memory location M, pointed to by `p`, is also noncurrent since the value in M is not the value expected by the user. M may be the home location of a variable, or it may be part of a dynamically allocated heap object. The debugger can determine exactly which location M is noncurrent, since the value of `p` that was used by the prematurely executed assignment of S_3 is available in `R4`. However, consider a breakpoint at instruction I_5, reported at statement S_1. At this breakpoint, the assignment of S_3 has again been executed prematurely, but the value of `p` used by S_3 is unavailable because `p`'s register (`R4`) is holding some other variable's value (the value of `b` assigned at I_4). Therefore, the debugger cannot determine the memory location M that is noncurrent and must conservatively assume any variable with a home locations in memory, i.e., `f` or `g`, may be noncurrent at this breakpoint. In this situation, `f` and `g` are said to be *suspect* variables because the debugger cannot determine with certainty whether these variables are noncurrent.

In general, noncurrent and suspect variables are referred to as *endangered* variables; i.e., an endangered variable is a variable whose run-time value *may* not correspond to the variable's expected source value. If a variable is not endangered, then it is *current*.

The debuggers for most systems in use today usually punt the issue of optimized code, by either turning optimization off whenever the user asks for source level debugging, or displaying the value in an inspected variable's assigned location without attempting to detect and warn the user that the variable is endangered. These approaches are unsatisfactory because:

1. The ability to debug optimized code is desirable and sometimes necessary and therefore should not be compromised [8,4,5].

2. Displaying values without detecting and warning the user of the effects of optimizations can mislead the user [5].

In this paper we present and evaluate an approach to detecting endangered variables caused by instruction scheduling. We also consider a simple but effective recovery scheme that improves the precision of the debugger in retrieving source values at a breakpoint. We present data measuring the effects of instruction scheduling on a symbolic debugger's ability to retrieve source level values at breakpoints. We also present data showing that our recovery strategy can be effective. The data have been gathered for a collection of C programs from the SPEC suite and the Numerical Recipes, which have been compiled with a variant of a commercial C compiler.

The rest of this paper is organized as follows. Section 2 describes our debugger model and approach to debugging optimized code. Section 3 presents our approach to detecting and recovering endangered variables caused by instruction scheduling. Section 4 presents experimental results, and Section 5 presents our conclusions.

2 Debugger model

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 is 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 [10,13].

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*. A *breakpoint* refers to a pair $< S, I >$, where S is the source breakpoint and I the object breakpoint.

The debugger can change the object code to implement control or data breakpoints, e.g., by using the techniques described in [10] and [13], but the compiler is not allowed to insert extra code to make debugging easier. The debugger is also not allowed to modify the object code to ease retrieval of values, e.g., by saving values before they are lost or re-ordering code back to the original source order. Any change to the object code must be limited to the implementation of breakpoints, i.e., the code generated by the compiler for debugging is the same as the code generated otherwise so that debugging is *non-invasive*.

There are many problems with allowing data modification by a debugger's user in the presence of optimizations [8]. We do not address this problem in this paper, and data modification by the user is not supported in our debugger model.

2.1 Retrieving source values

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 source expressions to execute in canonical execution order.

At a breakpoint $< S, I >$, the value that a user expects a variable V to have, relative to the source breakpoint S, is V's *expected* value, while the value in V's run-time location at I is V's *actual* value. In the example of Figure 1, at breakpoint $< S_3, I_3 >$ the expected value of `d` is the value assigned by statement S_1. However, the actual value of `d`, i.e., the value in register $R6$, is the value assigned by the last assignment to `d` before this block of code.

Sometimes there may be no location holding a variable V's value at a breakpoint, and consequently no actual value of V may exist. For example, if a variable V has been assigned a register R, and the breakpoint lies outside of V's live range, then R may be holding the value of some other variable at the breakpoint. In the example of Figure 1, `p` has no actual value at $< S_1, I_5 >$ since `p`'s assigned register `R4` is holding the value of `b` (assigned at I_4) at this breakpoint. If, at a breakpoint B, the debugger determines that the register assigned to a variable V is holding the value of some other variable, then V is *nonresident* at B [1]. The fourth column of Table 1 lists the nonresident variables at breakpoints in the code of Figure 1.

If the actual value of a variable V at a breakpoint B is identical to the expected value of V relative to the corresponding source breakpoint, then V is current at B. Otherwise, if the actual value of V *may* not correspond to the expected value of V, then V is endangered at the breakpoint. Endangerment does not pertain to a nonresident variable, since such a variable does not have an actual value. This implies that if V is a register assigned variable, then only after determining that V is resident can the debugger determine whether V is endangered.

If the debugger can determine with certainty that the actual value of V does not correspond to the expected value of V, then V is noncurrent at B. However, if a function call or indirect assignment operation is executed out of sequence, the debugger may not be able to determine the operation's side effects on the memory state. As a result, if a variable V has a home location in memory (rather than assigned a register), the debugger is not able to determine whether the actual value of V corresponds to the expected value of V at the breakpoint, and V is suspect at B. Objects allocated in the heap and referenced via pointers can also be endangered, noncurrent or suspect.

There are two ways in which a variable V may be noncurrent at a breakpoint B. Either an assignment to V has prematurely overwritten V's expected value, in which case V is a *roll back* variable, or the assignment that updates the expected value of V has been delayed until after the breakpoint, in which case V is a *roll forward* variable [8]. For example, at breakpoint $< S_2, I_4 >$ in Figure 1, `d` is a roll forward variable because the instruction corresponding to S_1's assignment (I_5) has not yet executed, while the object pointed to by `p` is a roll back variable because S_3 has been executed prematurely at this breakpoint. The last three columns of Table 1 list the roll forward, roll back and suspect variables at breakpoints in the code of Figure 1.

To provide the expected value of a noncurrent variable V, the debugger may attempt to construct V's expected value from other run-time values. At breakpoint

$< S_2, I_4 >$ in Figure 1, the expected value of `d` is the value assigned by S_1, and computed by I_5. The values of I_5's source registers (`R1` and `R2`) are computed at instructions I_1 and I_2, and are thus available at I_4. Therefore the debugger can provide the expected value of `d` by interpreting I_5. This process is called *recovery*. When attempting recovery, the debugger must be prepared to handle the case where the interpreted instructions cause an exception. To simplify execution resumption from a breakpoint, we assume that the debugger buffers recovered values into its own data structures rather than modifying the halted program's run-time state.

2.2 Evaluation order

The expected value of a variable relative to a source breakpoint is not always well defined because the evaluation order may not be precisely defined by the source language semantics. To determine expected source values at breakpoints, the debugger must accurately model the canonical execution order of expressions that have side effects. Usually, the canonical execution order of side effects from different source statements is well defined, i.e., the semantics specify that statements execute in sequence. However, in a C language expression containing multiple side effecting expressions, the compiler is free to choose the evaluation order of the side effecting expressions. For example, in the code fragment of Figure 2, the assignment in statement S_1 must execute before the assignments in S_2, and all assignments in S_2 must execute before the assignment in S_3, but the compiler is free to choose the evaluation order of the three assignments within statement S_2. Thus, at an asynchronous break occurring during the execution of expression `y++` within S_2, the expected value of `x` is not uniquely defined since this value can be assigned by either S_1 or by the expression `x++` of S_2. I.e., if the actual value of `x` corresponds to either of these two values, then `x` is current.

In practice most compilers impose an evaluation order during the translation from source to intermediate form. The debugger should not consider the compiler defined evaluation order as the canonical execution order, otherwise the debugger will produce conservative and possibly misleading responses to the user. Considering again the example of Figure 2, the compiler may impose a left to right evaluation order so that `x++` is evaluated before `y++` at the intermediate representation level (i.e., before code generation). However, if the code generator delays the evaluation of `x++` until after the evaluation of `y++`, and the debugger uses the compiler defined evaluation order as the canonical execution order, then the debugger will report `x` as noncurrent at a break occurring within `y++`. If the actual value of `x` is the value assigned by S_1, then this response is not accurate since the actual value of `x` is a valid expected value.

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

Figure 2: Undefined evaluation order in C

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 [14,5]. To provide expected behavior non-invasively, the debugger must detect all nonresident and endangered variables and recover their expected values.

Recovering the expected values is not always possible [2], therefore like most other debuggers for optimized code we settle for *truthful* behavior [14,5] : 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. 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 assignment(s) an endangered variable's actual value was (or may have been) assigned [2,5].

Prior work on debugging optimized code has made simplifying assumptions for either the language model, the debugger model, or both. For example, [8] considers only a subset of Pascal without pointers, while [5] does not consider practical language aspects such as undefined evaluation order of assignment expressions [6]. Realistic languages like C do not completely specify the order of evaluation and allow aliasing, and the design of a debugger for optimized code must consider these language properties. Another frequent simplification is to allow breakpoints only at pre-determined statement boundaries (e.g., [7] and [5]), but in practice, debuggers are used for postmortem analysis of programs, or can be invoked as a result of run-time faults, data breaks or user interrupts such as Control-C. Therefore, a debugger for optimized code must provide accurate information at breakpoints occurring anywhere in the object code. Our work focuses on using C as the source language, and our debugger model is general, allowing breakpoints at arbitrary points in the object code. No prior work has dealt with these issues in a single

framework.

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 of the data space that matches the source code, these systems provide either raw information (e.g., the current machine state) to the user or convey the results of optimizations (e.g., statements eliminated). CXdb [4] 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) being executed at each breakpoint. 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 source expression being executed is illuminated. However, if the program is run until a break occurs, CXdb tells the user at which source expression the break occurred but fails to provide any history or context.

3 Detection and recovery

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 and re-use of storage locations), as well as target machine features (e.g., concurrent execution of multiple instructions), must be precisely modelled. In this section, we describe a model for detecting and recovering endangered variables.

3.1 Execution order

To detect endangered variables, the debugger must first determine which source expressions have executed out of order at a breakpoint. Thus it is necessary to capture the canonical execution order of source expressions and link this to the expressions' object execution order. Like all optimizing compilers we are aware of, our compiler 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 source expressions with instructions in the object code via the IR.

The canonical execution order of IR operations is captured by annotating each expression A in the IR with a sequence number $Seq(A)$. Given two IR operations A and B 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. In the example of Figure 2 the three assignment expressions of statement S_2 will all have the same sequence number.

The target machine model for our compiler and debugger is a processor 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.

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 (the one with the highest offset in the basic block schedule) 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 operation A, such that A assigns 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.

For each instruction I, we record its position in the basic block schedule, denoted *Offset(I)*. 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 and index registers) is denoted by *AddressRegs*(I).

[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.

3.2 Out-of-order operations

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 [8], however, our model considers values held in the physical registers of the machine as well as individual instructions generated for each IR operation.

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 occurs before M in the canonical execution order but is scheduled to complete after O, then N is a *roll forward operation* at breakpoint O, denoted $RFOp(N, O)$.
- If N occurs after M in the canonical execution order but is scheduled to complete 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 a 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 instructions. An instruction I that was generated for an out-of-order IR operation *IR(I)* is an *out-of-order instruction*. If *IR(I)* is a roll forward (roll back) operation at an object breakpoint O, then I is a *roll forward (roll back) instruction* at breakpoint O.

3.3 Noncurrent and suspect variables

Having discovered which operations are out-of-order, the debugger must then determine how the expected source state has been affected by such operations. If an IR operation A is out-of-order at a breakpoint B, and A assigns to a variable V, then V is noncurrent at B. However, if the debugger cannot determine the side effects of A, e.g., a function call, then all variables that are potentially affected by A are suspect at B. Operations that affect the source state are assignments and function calls. Therefore, only out-of-order assignment and function call operations are relevant to the detection of endangered variables.

We distinguish between two forms of IR assignments:

1. **Direct assignments.** Assignments with lvalue expressions that are identifiers [9]; e.g., the assignment expressions at S_1 and S_2 in Figure 1.

2. **Indirect assignments.** Assignments with lvalues that are of the form `*E` where `E` is an expression of pointer type; e.g., the assignment expression at S_3 in Figure 1. Assignments to array elements (e.g., `a[i]`) and assignments through structure pointers (e.g., `p->x`), also fall into this category since the lvalue expressions of such assignments are transformed by the compiler into equivalent expressions of the form `*E`. If A is an assignment of the form `*E1 = E2`, we call `E1` the *address expression* of A.

Out-of-order direct assignments cause noncurrent variables, since the debugger can determine from the lvalue identifier exactly which variable is being assigned to. In the case of an out-of-order indirect assignment A of the form `*E1 = E2`, the debugger must retrieve the value of the address expression `E1` to determine which memory location is affected by A. If the value of `E1` cannot be retrieved, the debugger must conservatively assume that all objects in memory may have been affected by A. Similarly, if a function call is out-of-order, the debugger cannot determine which memory locations may have been affected by the call. Thus, out-of-order function calls and indirect assignments (with non-recoverable address expressions) cause suspect variables. In the example of Figure 1, at breakpoint $< S_1, I_5 >$, the assignments of S_2 and S_3 are both roll back operations. The assignment of S_2 is a direct assignment to `b`, thus the debugger can determine that `b` is noncurrent. The assignment of S_3 is an indirect assignment with the address expression `p`. The value of `p`, however, cannot be recovered and the debugger cannot determine which memory location has been prematurely assigned by this assignment. Therefore, all variables in memory, in this case `f` and `g`, are reported as suspect.

Note that variables that have been assigned registers cannot be suspect since such variables are not affected by indirect assignments or function calls.

3.4 Recovery

The IR annotations provide a correspondence between run-time values and IR expressions. This correspondence allows the debugger to recover values of IR expressions from the run-time state, thus providing more precise information to the user in two ways:

1. 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. In Figure 1, the assignment of S_1 is a roll-forward assignment that assigns the expected value of `d` at breakpoint $< S_2, I_4 >$. The debugger can recover the expected value of `d` by interpreting I_5, the instruction that computes the value assigned by S_1. Recovery of values assigned by assignment operations is called *assignment recovery*.

2. Recovering address expressions of indirect assignments allows the debugger to refine the set of suspect variables, since it allows the debugger to determine precisely which memory location is affected by an out-of-order indirect assignment. For instance, in the example of Figure 1, at breakpoint $< S_2, I_4 >$, I_3 is a roll-back store instruction, corresponding to an indirect assignment. The value of I_3's address expression is available in `R4` and can be recovered, thus allowing the debugger to determine which memory location was affected by the store. The debugger then reports this location as noncurrent. At breakpoint $< S_1, I_5 >$, I_3 is again a roll-back store instruction, but I_3's address expression value cannot be recovered because it was overwritten at I_4. Recovery of address expression values is called *address recovery*.

Both recovery strategies use the same techniques but have different objectives. Address recovery attempts to provide a more precise picture of the current run-time state to the debugger, while assignment recovery provides the user with the expected values of variables relative to the current breakpoint. Since assignment recovery is only useful if the variable is not suspect, the debugger first attempts address recovery of roll forward and roll back indirect assignments, and then attempts assignment recovery of roll forward assignments (either direct assignments or indirect assignments with recoverable address expressions). As has been noted in earlier work [8], roll forward variables are easier to recover than roll back variables. Therefore, we do not attempt recovery of roll back variables.

3.4.1 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:

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

3.4.2 Assignment recovery

Let A be a direct assignment of the form $x = N$, where x is an identifier. Let B be a breakpoint that occurs in the same block as A. If x is a register assigned 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. If x is a memory variable the instruction generated for A is a store instruction that stores the value computed by N. In both cases, the value assigned by A is recoverable if the value computed by N is recoverable. If A assigns x's expected value relative to breakpoint B, and N's value is recoverable, then the debugger can provide x's expected value.

On the other hand, if A is an indirect assignment of the form $*E = N$, in addition to N being recoverable, address recovery must also recover the address expression E for A to be recoverable.

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

1. $(\textit{Offset(I)} < O) \wedge (\textit{NextDef}_I(DestReg(I)) \geq O)$ or

2. $\forall R \in SourceRegs(I) : (\textit{LastDef}_I(R) < O) \wedge (\textit{NextDef}_I(R) \geq O)$ 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 and memory locations are endangered. We do not interpret function calls since they transfer control out of the current basic block; such interpretation is difficult. Also, we do not interpret loads since the values in memory may be endangered.

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

4 Experimental results

We have implemented detection and recovery of endangered variables as described above in a variant of a commercial C compiler: the iWarp C compiler pre-release version 2.7. This compiler is based on the PCC2 compiler from AT&T that has been enhanced with global optimizations.

The target machine is the iWarp microprocessor, a Long Instruction Word (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 [3].

The compiler performs global register allocation and assignment, branch optimizations, unreachable code elimination, and constant folding. 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.

The code generator was modified to emit the information and IR annotations described in Section 3; the object code is identical to the default code produced. Debugging is totally non-intrusive and the compiler does not effect any changes in the code, the memory layout, or any other aspect of the object program. For our experiments, the algorithms for detecting and recovering endangered variables are implemented in a separate program that analyzes the object code and the annotated IR. This program also computes the statistics presented in this section.

The set of programs for this evaluation consists of three C programs from the SPEC integer suite (li, espresso and eqntott)[12], and twelve programs from the Numerical Recipes collection (balanc, bessi, bessj, elmhes, fft, gaussj, hqr, jacobi, ludcmp, meo, svdcmp, and tred2)[11]. 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".

The information presented in this section is collected by analyzing the code generator output for all possible breakpoints and averaging the results by the number of breakpoints. Since we are considering asynchronous breakpoints, the analysis is done for each instruction and the results averaged by the number of instruction.

Since the number of endangered variables at a breakpoint is a function of the program *and* the code generator, we measured how often instructions from different statements are being interleaved. The second column of Table 2 shows the average number of out-of-order instructions at each breakpoint. For comparison, the average number of statements per basic block in the source programs is shown in the third column. Although there are on the average only a few statements per basic block, the compiler interleaves code quite frequently when the opportunity exists. Basic blocks with just a single statements contain no out-of-order operations, and therefore no endangered variables. 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.

Program	I	II
`li`	0.8	1.2
`espresso`	0.6	1.6
`eqntott`	0.7	1.5
`recipes`	1.1	2.1

Table 2: Effect of scheduling: Average number of out-of-order instructions per breakpoint (I), and average number of statements per basic block (II).

The first column of the charts in Figure 3 shows the number of breakpoints that contain endangered variables as a percentage of the total number of possible breakpoints in each benchmark program. These are breakpoints with out-of-order assignment or function call operations. The percentage of such breakpoints ranges from approximately 9% (li) to 21% (recipes).

Operations that can cause endangered variables are classified into five categories:

1. Direct assignments to local variables that have been assigned registers.
2. Direct assignments to local variables with home locations in memory.
3. Direct assignments to global variables with home locations in memory.
4. Indirect assignments.
5. Function calls.

Columns 2 to 6 of Figure 3 show the percentage of breakpoints with endangered variables caused by each of these different kinds of operations. At a breakpoint there can be more than one out-of-order operation, therefore 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 (4%) and a low percentage of breakpoints with out-of-order function calls. In the SPEC programs, out of order indirect assignments occur at about 2% of all 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 affected by direct assignments can be precisely determined by the debugger, and hence these types of assignments do not cause suspect variables.

Figure 4 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 run-time), 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 4 shows the number of breakpoints with endangered variables (from Figure 3) and is repeated here for reference. The second column of the charts shows the percentage of breakpoints with noncurrent variables, i.e., breakpoints with out-of-order direct assignments. The third column shows the percentage of breakpoints with suspect variables, i.e., breakpoints with out-of-order indirect assignments or function calls. 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.

When a function call or indirect assignment is out-of-order at a breakpoint, the number of suspect variables can be quite large since all memory variables and objects will be suspect. Any success in identifying the effect of the indirect assignments allows the debugger to provide the user with more precise information, and thus improves the quality of the debugger. The last two columns in the graphs of Figure 4 show the effects of recovering addresses of out-of-order indirect assignments. As expected, the number of breakpoints with suspect variables variables decreases while the number of breakpoints with noncurrent variables increases. 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 5 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 3. 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 3.4, 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 whether they cause suspect or noncurrent variables. Almost all roll back operations are direct assignment operations (Columns 3 and 5); roll back function calls or indirect assignments are very rare (Column 7). That is, almost all suspect variables are caused by roll forward operations, and those can be potentially recovered 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 6. 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 direct assignments to register assigned variables, local variables in memory, and global variables in memory, 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 (compare Column 7 with Column 8), but there are some for which only the the address expressions can be recovered (compare Column 8 with Column 9).

Benchmark	I	II
li	8.8	6.1
espresso	12.9	9.0
eqntott	14.5	10.7
recipes	20.8	16.2

Table 3: Effects of recovery: Breakpoints with out-of-order function calls or assignments total (I) and after recovery (II).

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 5 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 static breakpoints with endangered variables (between 9% and 21% in our suite). A simple recovery scheme as described in this paper reduces the number of breakpoints with endangered variables. Table 3 presents the bottom line and shows how recovery affects the number of breakpoints where the debugger can provide the expected values of source variables.

5 Conclusions

Instruction scheduling is an important aspect of code generation, and a large number of modern compilers include some form of instruction scheduling. This optimization causes endangered variables by re-ordering the execution of source expressions. 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 such variables and warn the user if such a variable is queried.

Detecting noncurrent variables caused by instruction scheduling requires accurate modelling 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 ongoing 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.

Acknowledgements

We thank Amer Diwan, Scott Nettles, David Tarditi, Bob Wheeler, and Po-Jen Yang for helpful comments on an earlier draft of this paper.

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] Ali-Reza Adl-Tabatabai. Nonresident and endangered variables: The effects of code generation optimizations on symbolic debugging. Technical Report CMU-CS-92-221, CMU, December 1992.

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

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

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

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

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

[14] 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.

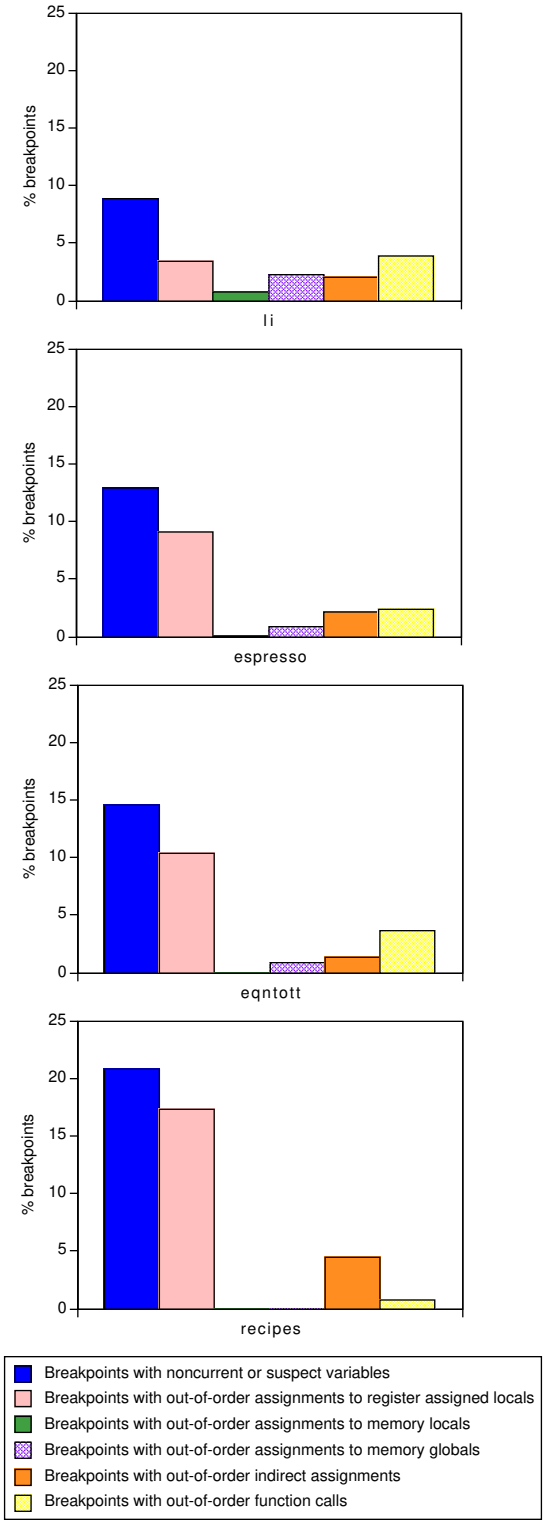

Figure 3: Breakpoints with operations causing endangered variables

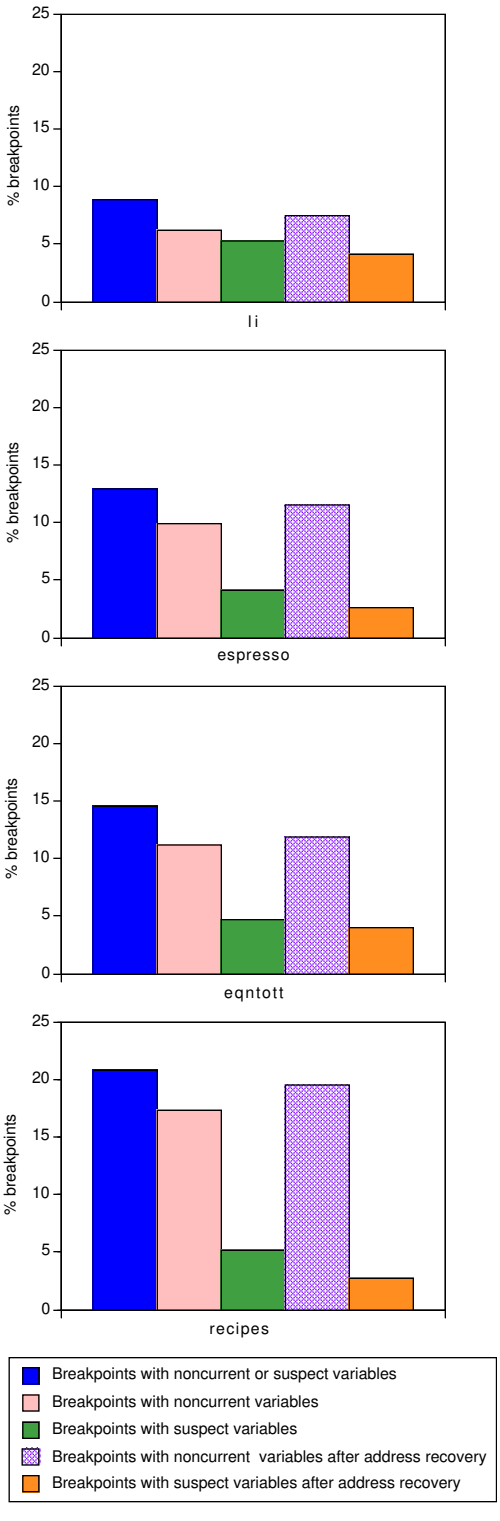

Figure 4: Breakpoints with suspect and noncurrent variables, with and without address recovery

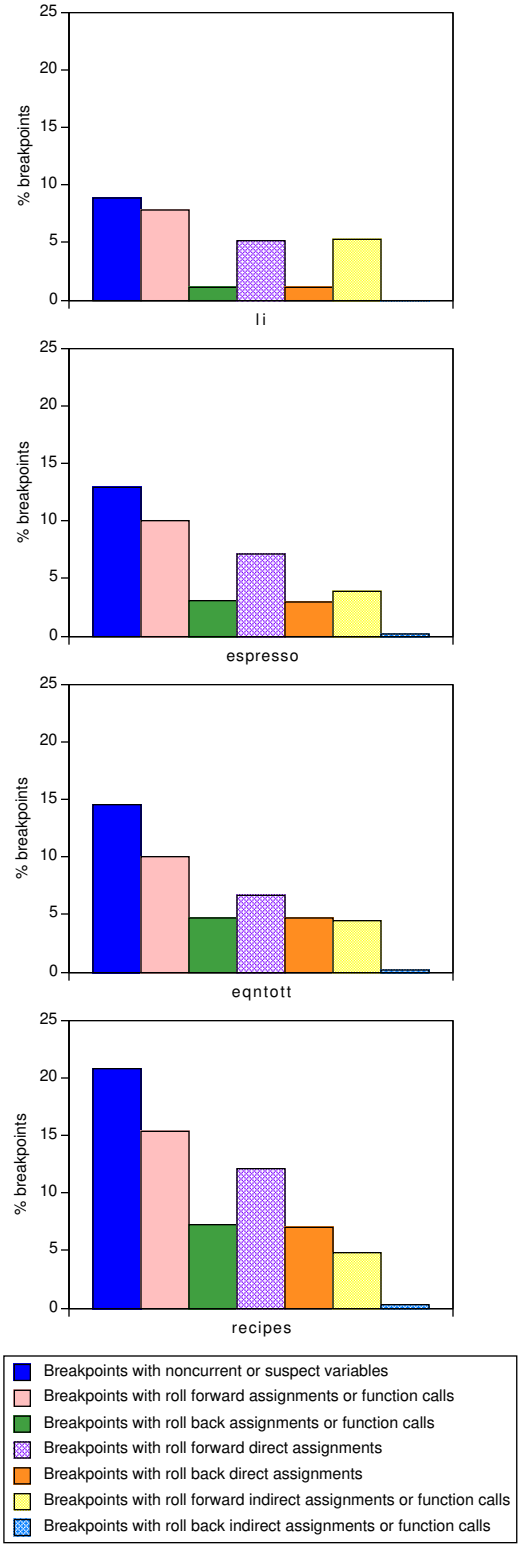

Figure 5: Breakpoints with roll forward and roll back operations

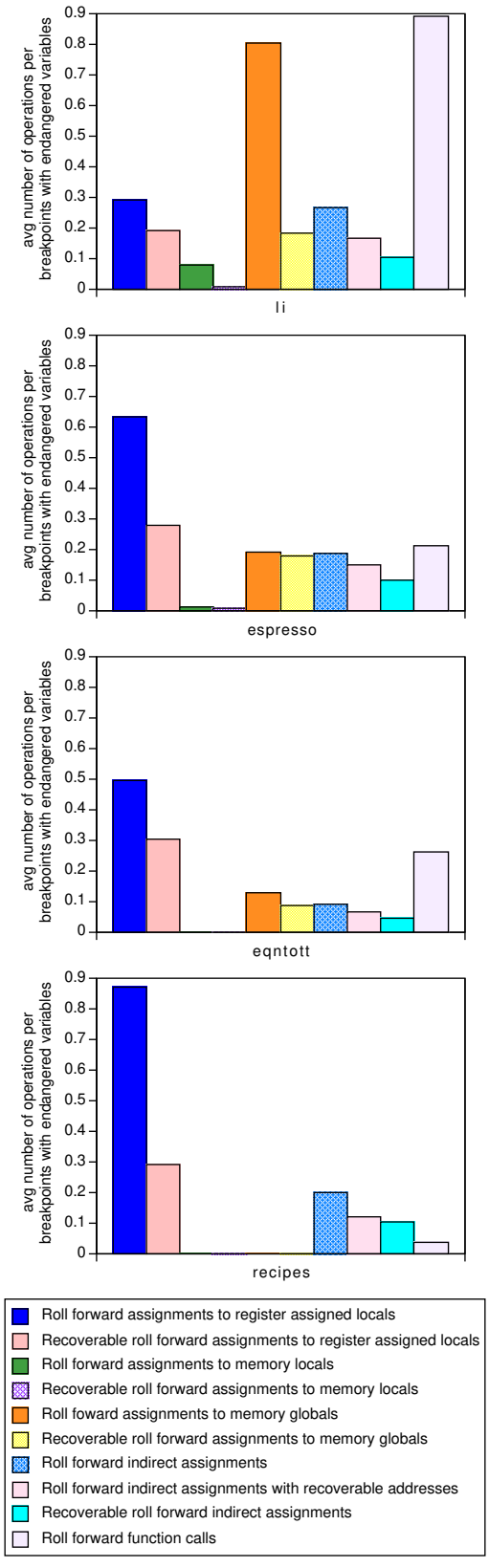

Figure 6: Average number of roll forward and recoverable operations