Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging

Ali-Reza Adl-Tabatabai Thomas Gross

October 1992
CMU-CS-92-202

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this report appears in the *Proceedings of POPL'93 The Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 10-13, 1993 Charleston, SC

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, noncurrent variables, register allocation, code generation, compiler optimization, data-flow analysis

Abstract

A symbolic debugger allows a user to display the values of program variables at a breakpoint. However, problems arise if the program is translated by an optimizing compiler. This paper addresses the effects of global register allocation and assignment: a register assigned to a variable V may not be holding V's value at a breakpoint since the register can also be assigned to other variables. We define the problem of determining whether a variable is in its assigned register as the *residence* problem. Prior work on debugging of optimized code has focused on the currency problem; detecting whether a variable's run-time value is the expected value. Determining residence is a more serious problem than currency detection. We present a data-flow algorithm that accurately computes a variable's residency, by determining when a variable becomes *evicted* from its register. We measure the effectiveness of different approaches to determine variable residence for three C programs from the SPEC suite.

1 Introduction

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 or to inspect variables; 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.

The problems encountered by a symbolic debugger of optimized code have received considerable attention in the past. Starting with Hennessy [12], a number of studies [14,10,8,3,11] have investigated the detection and recovery of *noncurrent* variables, variables whose values are inconsistent with what the user expects from inspecting the source.

This paper discusses a more fundamental problem that a symbolic debugger has to handle: detection of *nonresident* variables. Global register allocation and assignment effectively exploit modern architectures that have large register files and high memory access latencies, and these optimizations are incorporated in almost all modern optimizing compilers. These optimizations, however, 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.

From the viewpoint of the user, noncurrent and nonresident variables are similar in that the debugger cannot display the expected value. However, a noncurrent variable has a value that is inconsistent with what the user expects, whereas a nonresident variable has *no* value. That is, the value in a noncurrent variable's run-time location is a source level value, but it is not 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 a noncurrent variable's value corresponds to [11,8]. On the other hand, no source value can be displayed for a nonresident variable.

This paper investigates the problem of detecting nonresident variables in the presence of global register allocation. Our solution is 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 resident variables is concerned only with whether a variable V's register contains *any* source value of V, and not whether V is current. Other mechanisms must exist that detect noncurrent variables. These mechanisms are orthogonal to our method of detecting nonresidence. Other compiler optimizations do not affect our method as long as the compiler can provide the necessary bookkeeping of source assignments.

To measure the effects of our approach (and the seriousness of the problem), we have implemented the techniques in a production C compiler that performs code compaction and register allocation for an LIW machine with a large register file. We have compared the effects of nonresidency with the effects of noncurrency 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 noncurrency 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 [11]) misses many opportunities.

The following section discusses the problem of debugging optimized code. Section 3 describes our global register allocation model. Section 4 discusses approaches to detecting nonresident variables and

describes our solution. Section 5 discusses the effects of optimizations employed by the code generator to eliminate register copy operations. Section 6 compares our work with prior work in debugging optimized code. Section 7 reports the results of using our approach on some sample programs, and Section 8 presents our conclusions.

2 Debugging optimized code

Our debugger model is the same as that used by a number of other researchers (see e.g. [8]). The debugger supports the base operations of setting control breakpoints, inspecting data, and resuming execution after a breakpoint. Control breakpoints are either synchronous, such as source level breakpoints, or asynchronous, such as program faults or user interrupts. Data inspection is limited to source variables, and the debugger does not change the state of a program except for setting breakpoints; data modification by the user is not supported. The debugger is *non-invasive*; no modification of the program's code or data is allowed, i.e., the code executed when debugging is identical to the code executed 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* [15] 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.

When the debugger is invoked as a result of a control breakpoint, the point in the object at which execution has halted is called the *object breakpoint location*, and the source statement where the breakpoint is reported is called the *source breakpoint location*.

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 [16]. In this paper, we only address the data value problem of retrieving source level values from the run-time state of a halted program. Our work assumes that breakpoints can occur anywhere in the object code and hence applies to both synchronous and asynchronous breakpoints. See [8] or [15] for a discussion of flow related issues.

2.1 Retrieving source values

In response to a query of a variable V's value at a source breakpoint location S, the user expects the value from the latest source assignment to V, relative to S. This value is the *expected* value of V [8]. The debugger either presents V's expected value or detects and reports that V's expected value cannot be presented. If a value cannot be presented, the debugger may provide additional guidance to the user by conveying how optimizations have affected source values. Thus we assume the debugger exhibits *truthful* behavior [16].

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 [8].

2. V's actual value must be the same as V's expected value. If V's residence holds V's expected value, V is called *current*, otherwise V is called *noncurrent* [12]. A nonresident variable does not have an actual value, hence currency 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 its value is noncurrent because the variable's actual value is not the same (or *may* not be the same [1]) as its expected value. This paper focuses on the problem of detecting nonresident variables.

2.2 Example

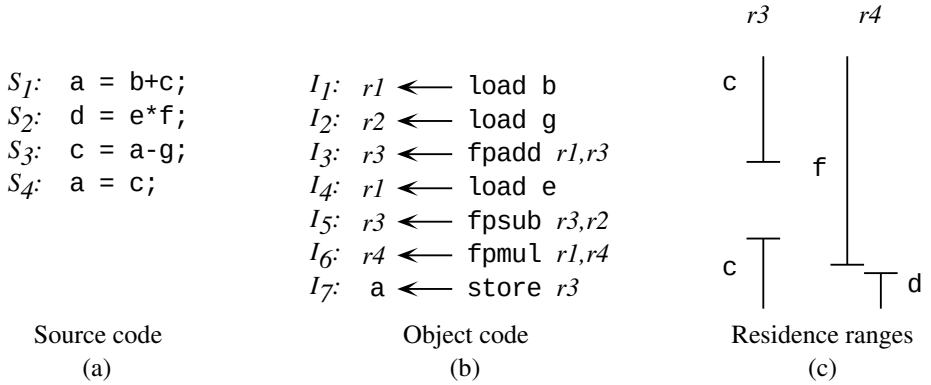

Figure 1: Example of noncurrent and nonresident variables

Consider the source and object codes shown in Figure 1(a) - (b). In this example, variable `c` has been assigned register *r3*, while `d` and `f` have both been assigned register *r4*. Register *r3* is also used as an expression temporary at instruction I_3. No code has been generated to perform a store to `a` for S_1, since S_1's assignment is redundant. The value computed by this statement is kept in register *r3* for use by statement S_3. Figure 1(c) shows the ranges of instructions during which register assigned variables are resident. `c` is resident upon entry to this block until instruction I_3, at which point `c` becomes dead and *r3* is reassigned to a temporary. `c` becomes resident again at I_5. *r4* contains a value of `f` upon entry, hence `d` is nonresident. At I_6, *r4* is reassigned to `d`, and as a result `f` becomes nonresident.

Table 1 lists the nonresident and noncurrent variables at all possible object breakpoint locations in the code of Figure 1. We report an asynchronous breakpoint at an instruction I as a breakpoint at the source statement for which I was generated. The first two columns of Table 1 show the mapping from object locations to source statements. The third column presents the source expression that is computed by each instruction, and the last two columns list the nonresident and noncurrent variables if a breakpoint happens at a given instruction.

Consider a breakpoint at instruction I_5, reported at statement S_3 in the source. At this breakpoint, the expected value of `a` is the value assigned at statement S_1, and the expected value of `d` is the value assigned at statement S_2. The actual value of `a` is the value in `a`'s memory location before entry into this block, since the store to `a` has been eliminated. Therefore, `a` is noncurrent. `d` is nonresident at this

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

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

breakpoint, because the register assigned to `d` (*r4*) is holding `f`'s value. `c` is also nonresident because its register (*r3*) is holding the temporary value computed at I_3. Therefore no actual values exist for `c` and `d`. The expected and actual values of `b,e` and `f` are the values assigned by the last assignments to these variables before this block. Therefore, these variables are current at this breakpoint.

2.3 Nonresident and noncurrent variables

Observe that noncurrent and nonresident variables are different with regard to the possible behavior of the debugger. In the case of a noncurrent 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_2 in Figure 1. At this breakpoint, the assignment to `c` of statement S_3 has executed too early at I_5. In response to a user query of `c`, the debugger may display the value in register *r3* (`c`'s actual value) and explain to the user that the displayed value is the value of `c` assigned at S_3 because optimizations have caused statement S_3 to execute too early. This approach was adopted in the DOC debugger [11]. Copperman [8] 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_5, `c` is nonresident because its assigned register *r3* is holding the temporary value computed at instruction I_3. This value has no relation to `c` (in fact, it is the value of `a` computed by statement S_1) 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 [11] and CXdb [4].

An approach to dealing with noncurrent and nonresident variables is to *recover* a variable's expected value from the actual values of other variables and temporaries [12]. E.g., at a breakpoint at I_5, the debugger may infer that `a`'s expected value is in *r3*. Recovering values in globally optimized code is difficult (see [12] for a discussion of the scenarios when recovery can be attempted), and the effectiveness of recovery in practice is unknown.

2.4 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 noncurrent and provides additional information to the user.

In the absence of support provided by the run-time system (e.g., path determiners [15]) 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. This data-flow problem is known as reaching definitions [2]. Note that the debugger cannot help in the case that definitions reach on some but not all paths to S.

eferring back to the example of Figure 1, if no source assignment of `d` reaches the block of code, `d` can be reported as uninitialized rather than nonresident at any breakpoint reported at S_1 or S_2. In this example, these are breakpoints that occur at I_1, I_3, I_4 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 [5] 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* [6], 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 [5], 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.

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 [11], 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.

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

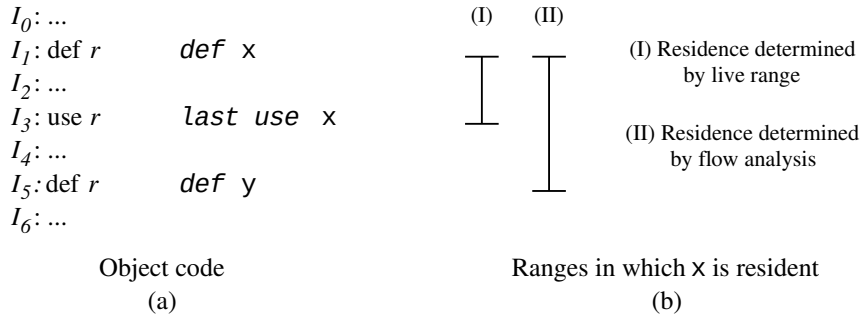

Figure 2: Example illustrating the approaches to detecting nonresident variables

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 2(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, S, E) where B is the set of basic blocks; $S \in B$ is the entry block; E is the set of edges between blocks such that if $(B_i, B_j) \in E$ then control may immediately reach B_j from B_i. Each basic block B_i contains a sequence of instructions generated by the compiler, as well as a special *preamble instruction* that appears before the other instruction 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* of the control flow graph is the point at the beginning of the source basic block S. The entry point dominates all other points in the control flow graph. A *path* is defined to be a sequence of points $p_1 ... p_n$ such that for each adjacent pair p_i, p_{i+1}, either $p_i = pre(I)$ and $p_{i+1} = post(I)$ for some instruction I, or p_i is a point at the end of a block B_j and p_{i+1} is a point at the beginning of a block B_k and $(B_j, B_k) \in E$.

We call an instruction that targets a register r a *definition* of r. An instruction I *reaches* a point p if there exists a path from $post(I)$ to p 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 2(a) and allows the debugger to display the value of `x` outside of `x`'s live range, as depicted by Figure 2(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 are to register assigned variables.

To track whether a variable V's value is contained 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 1, instructions I_5 and I_6 are generated for statements S_3 and S_2 respectively. These instructions target the registers assigned to variables `c` and `d` with the source values of these variables computed at statements S_3 and S_2. Hence I_5 and I_6 are source definitions of `c` and `d` : $I_{\mathtt{c}}(S_3) = I_5$ and $I_{\mathtt{d}}(S_2) = I_6$.

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 the entry point to o. The predicate $IsEvicted(V, o)$ is true if a variable V is evicted at point o in the object.*

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 3. 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 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 3.

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 $d \in$ ReachingDef(R(V),o) such that d is not a source definition of V.*

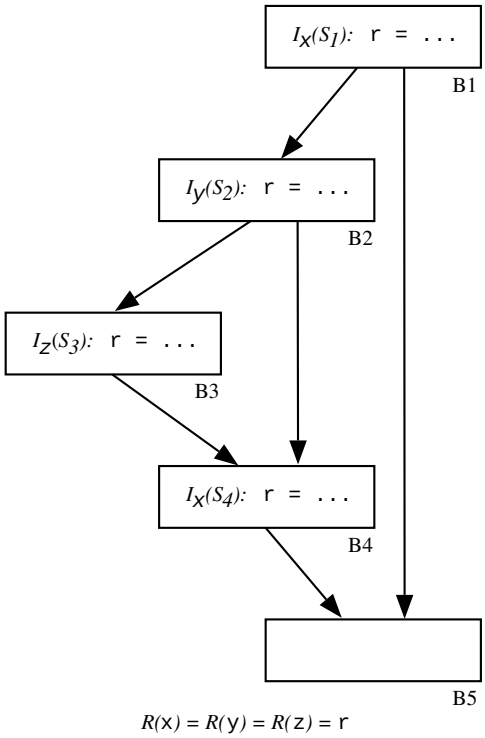

Figure 3: Object control flow graph

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 3

Instruction	Variables Evicted by Instruction
$I_{\mathtt{x}}(S_1)$	y,z
$I_{\mathtt{y}}(S_2)$	x,z
$I_{\mathtt{z}}(S_3)$	x,y
$I_{\mathtt{x}}(S_4)$	y,z

Table 3: Variables evicted by instructions in Figure 3

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 3. 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 $IsEvicted(V, o)$.

Since no evicting definitions reach the entry point s of the flow graph, no variable is evicted at s.

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

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

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

5 The effects of coalescing

Coalescing or subsumption [5] 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 4. 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, $I_{\mathtt{y}}(S_1) = I_1$ and $I_{\mathtt{x}}(S_2) = I_2$ 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`$)$).

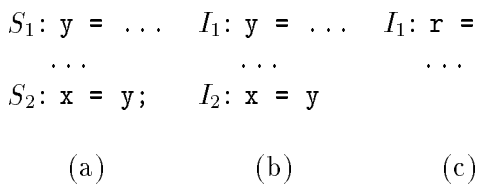

Figure 4: Effects of register subsumption

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 $I_{\mathtt{x}}(S_2) = I_{\mathtt{y}}(S_1) = I_1$ and $\{\mathtt{x},\mathtt{y}\} \subseteq EvictReachKill(I_1)$. 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 4 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 5 illustrates other cases that can occur. In this figure, coalescing eliminates the copy `x = y`. In Figure 5 (a) S post-dominates all reaching definitions of $R(V)$. Thus we may consider both reaching definitions as source definitions of S, adjusting $I_{\mathtt{x}}(S)$ 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 re-ordering

[1]. If there exists a reaching definition D that is not post-dominated by S, as shown in Figure 5(b), D cannot be considered a source definition of `x`.

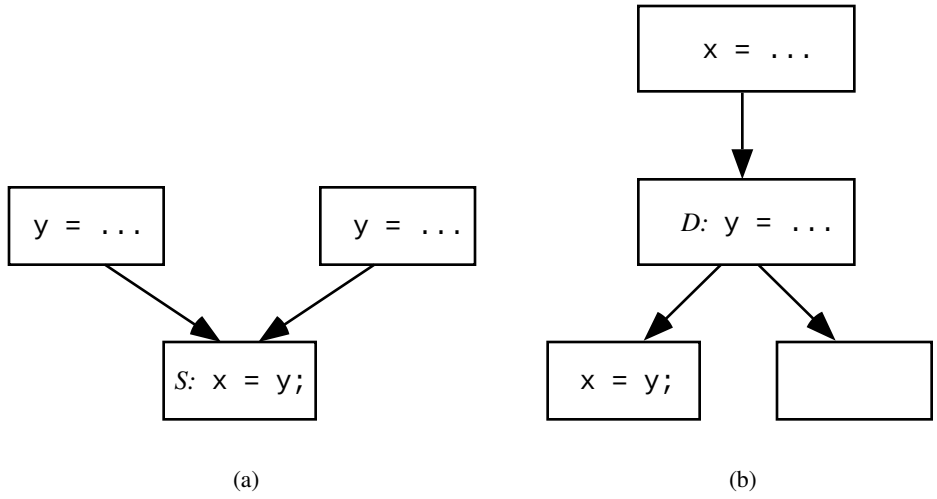

Figure 5: Global register coalescing

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.

6 Comparison with other 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 [11] and CXdb [4] debuggers, previous research has overlooked the problem of nonresident variables.

Hennessy's work [12] and later refinements of Wall et. al [14] deal with detection and recovery of noncurrent variables in the presence of local optimizations and code generation using DAGs. The model of [12] 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 typically are 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.

Copperman ([8], [7]), proposes a method of detecting noncurrent variables using global data-flow analysis but does not consider nonresident variables. A similar approach has been proposed by Bemmerl [3].

In another work, Copperman and McDowell point out that Hennessy's model does not consider values held in registers [10]. 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.

DOC [11] is the first 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 [11] 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 [10], DOC reports a variable as inaccessible after its last use. CXdb [4] is another system that detects nonresidency using the live range approach[13].

7 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 (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 processor, an LIW machine, with 128 registers, of which 94 are available to the compiler.

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. The register allocation model is the one described in Section 3. Due to the large number of registers in our machine, none of the benchmarks requires live ranges to be spilled to memory.

The compiler annotates each operation in the intermediate representation (IR) of the program with the list of machine instructions generated for the operation. The IR of assignments that have been eliminated due to coalescing are annotated as described in Section 5. Register assignments are recorded in a table that maps register assigned variables to physical registers. This information is used by our algorithms to detect nonresident and noncurrent variables (details of our algorithm for detecting noncurrent variables are described in [1]). These algorithms 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.

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.

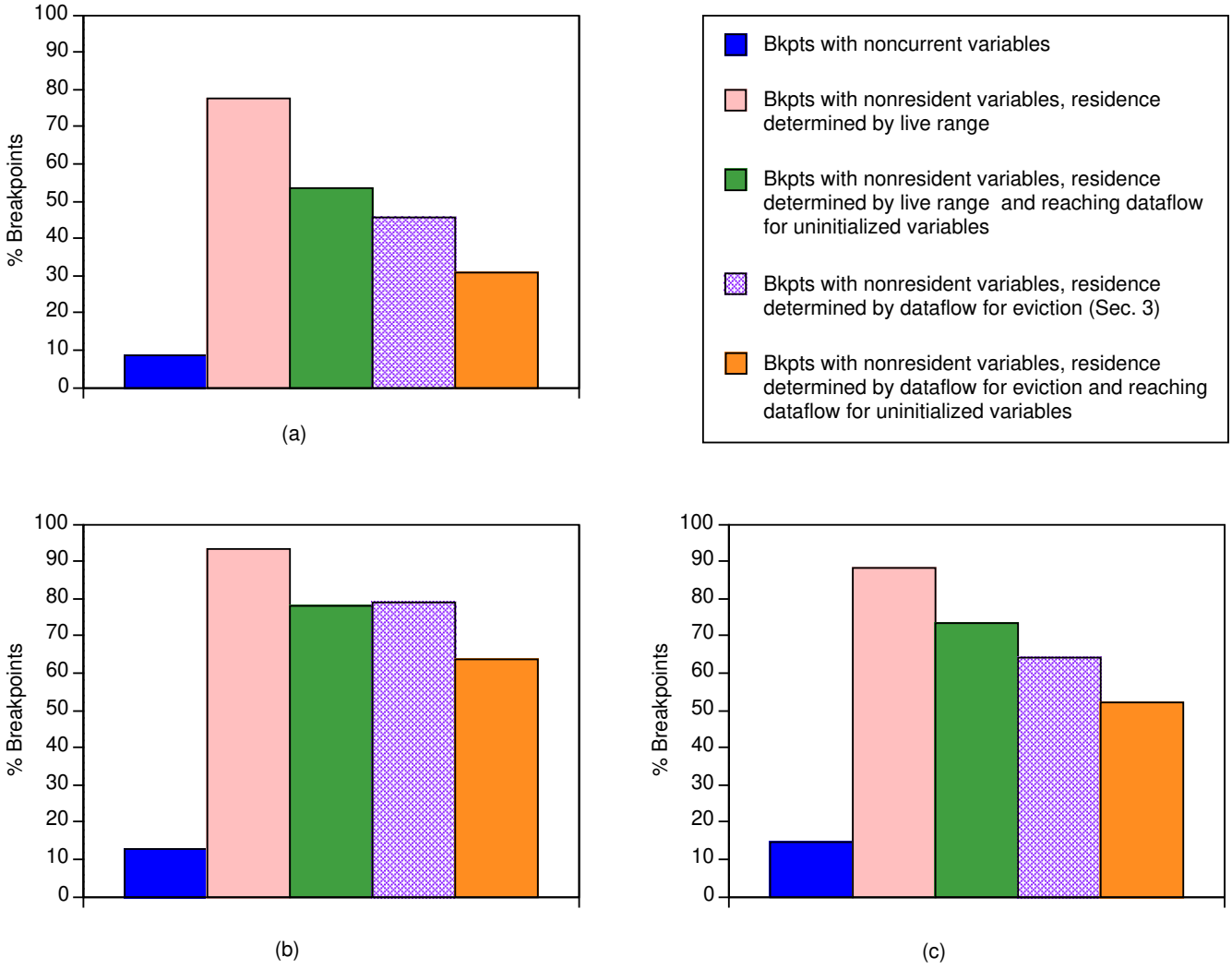

Figure 6: Percentage of breakpoints with noncurrent or nonresident variables for three C benchmarks from the SPEC suite: (a) li , (b) eqntott, and (c) espresso.

We present the results for three C benchmarks from the SPEC suite in Figure 6. This figure compares the percentage of breakpoints that contain noncurrent 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 noncurrent 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. The results for both models are close; more details can be found in [1]. 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 6 for these benchmarks show that there are significantly more breakpoints containing nonresident variables than there are breakpoints with noncurrent variables. Thus nonresident variables are potentially a more serious problem than noncurrent 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.

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 7 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.

8 Conclusions

Prior work in debugging of optimized code has concentrated mainly on the problem of detecting and recovering noncurrent 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 noncurrent variables. In our sample programs from the SPEC suite, 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 instruc-

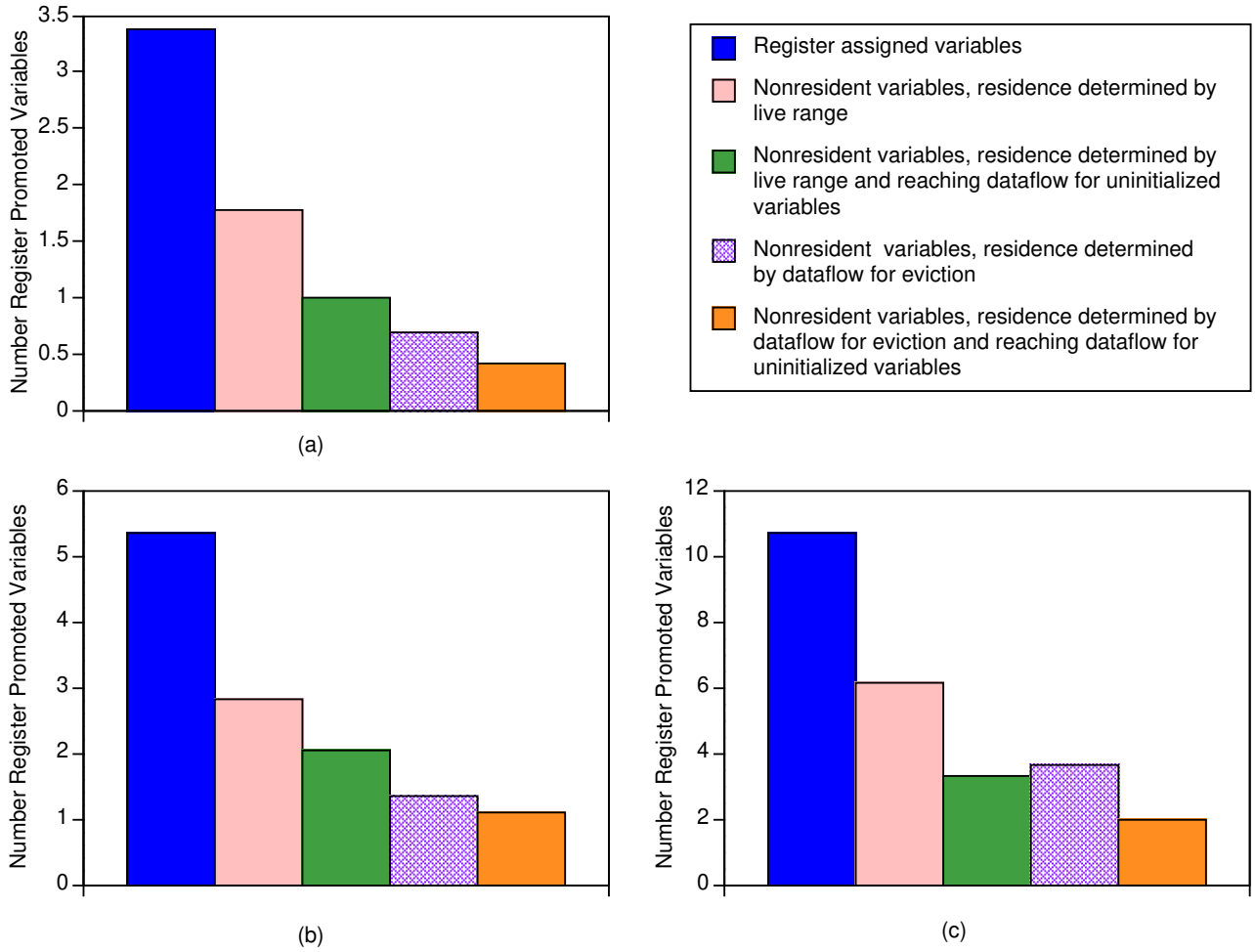

Figure 7: Number of nonresident register allocated variables for three C benchmarks from the SPEC suite: (a) li , (b) eqntott, and (c) espresso.

tions 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 [12,9,8,7] 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.

References

[1] A. Adl-Tabatabai. Symbolic debugging of optimized C code. Unpublished draft from School of Computer Science, Carnegie Mellon University, 1992.

[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] 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] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] J. L. Hennessy. Symbolic debugging of optimized code. *ACM TOPLAS*, 4(3):323 – 344, July 1982.

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

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

[15] 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.

[16] 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.